CaSCaDE: (Time-Based) Cryptography from Space Communications DElay

Carsten Baum[1], Bernardo David[2], Elena Pagnin[3], and Akira Takahashi[4]

[1] Technical University of Denmark
[2] IT University of Copenhagen
[3] Chalmers University
[4] The University of Edinburgh

March 21, 2023

Abstract. Time-based cryptographic primitives such as Time-Lock Puzzles (TLPs) and Verifiable Delay Functions (VDFs) have recently found many applications to the efficient design of secure protocols such as randomness beacons or multiparty computation with partial fairness. However, current TLP and VDF candidate constructions rely on the average hardness of sequential computational problems. Unfortunately, obtaining concrete parameters for these is notoriously hard, as there cannot be a large gap between the honest parties' and the adversary's runtime when solving the same problem. Moreover, even a constant improvement in algorithms for solving these problems can render parameter choices, and thus deployed systems, insecure - unless very conservative and therefore highly inefficient parameters are chosen.
In this work, we investigate how to construct time-based cryptographic primitives from communication delay, which has a known lower bound given the physical distance between devices: the speed of light. In order to obtain high delays, we explore the sequential communication delay that arises when sending a message through a constellation of satellites. This has the advantage that distances between protocol participants are guaranteed as positions of satellites are observable, so delay lower bounds can be easily computed. At the same time, building cryptographic primitives for this setting is challenging due to the constrained resources of satellites and possible corruptions of parties within the constellation.
We address these challenges by constructing efficient proofs of sequential communication delay to convince a verifier that a message has accrued delay by traversing a path among satellites. As part of this construction, we propose the first ordered multisignature scheme with security under a version of the the discrete logarithm assumption, which enjoys constant-size signatures and, modulo preprocessing, computational complexity independent of the number of signers. Building on our proofs of sequential communication delay, we show new constructions of Publicly Verifiable TLPs and VDFs whose delay guarantees are rooted on physical communication delay lower bounds. Our protocols as well as the ordered multisignature are analysed in the Universal Composability framework using novel models for sequential communication delays and (ordered) multisignatures. A direct application of our results is a randomness beacon that only accesses expensive communication resources in case of cheating.

Table of Contents

1 Introduction . . . 3
 1.1 Our Contributions . . . 3
 1.2 Technical Overview . . . 5
2 Preliminaries . . . 8
 2.1 Modelling Time and Global Clocks . . . 8
3 Modeling Communication Delays . . . 9
4 Proofs of Sequential Communication Delays . . . 11
 4.1 Modelling Proofs of Sequential Communication Delay . . . 12
5 Ordered Multi-Signatures . . . 19
6 Verifiable Delay Functions . . . 24
7 Publicly Verifiable Time-Lock Puzzles . . . 26
8 Delay Encryption and Stateless VDF . . . 30
A Auxiliary Functionalities and other Preliminaries . . . 37
 A.1 UC Secure Public-Key Encryption with Plaintext Verification . . . 40
 A.2 Global Clocks and Global tickers . . . 42
B Delayed Communication - Proofs and more details . . . 43
 B.1 Realizing $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ . . . 43
 B.2 Proof of Theorem 4.1 . . . 44
 B.3 Computing channel delays . . . 45
 B.4 Proof of Theorem 4.4 . . . 46
C Proof of Theorem 7.1 . . . 47
D UC Treatment of Delay Encryption . . . 47
E More on OMS . . . 49
 E.1 Preliminaries for our OMS construction . . . 49
 E.2 Proof of Theorem 5.4 . . . 51
 E.3 Three OMS Variants . . . 57
F Multi-Signatures in the UC Framework . . . 60
 F.1 Ideal Functionality for Multi-Signatures . . . 60
 F.2 UC Security of Interactive Multi-Signatures . . . 61
 F.3 UC Security of Interactive Ordered Multi-Signature Scheme . . . 70
 F.4 Integrating $\mathcal{F}_{\mathsf{OMS}}$ into $\pi_{\mathsf{Multi-SCD}}$ . . . 72

1 Introduction

Time-based primitives such as Time-Lock Puzzles (TLPs) [65] and Verifiable Delay Functions (VDFs) [15] have received a lot of attention recently as building blocks for efficient protocols that e.g. realize randomness beacons and multiparty computation with partial fairness. TLPs allow a sender to commit to a message in such a way that a receiver can obtain it only after a certain (polynomial) amount of time. A VDF works as a pseudorandom function whose evaluation requires at least a certain (polynomial-bound) amount of time, after which it generates both an output and a proof that the output was obtained after at least this amount of time has elapsed. Similarly, a publicly verifiable TLP (PV-TLP) also produces a proof that a certain message was contained in the puzzle. In both cases, verifying these proofs takes time essentially independent of the time needed to evaluate the VDF or solve the PV-TLP.

Although a lot of theoretical work has been done on constructing TLPs [9,12, 18,40,47,65] and VDFs [8,15,36,38,61,68], all of these constructions are based on the average hardness of sequential computational problems. The rationale behind these constructions is that the the minimum delay in evaluating a VDF or solving a TLP is obtained by forcing parties to solve computational problems that require a number of sequential steps that cannot be computed more efficiently in parallel. Hence, the more steps needed to solve the problem, the higher the minimum delay provided by the VDF/TLP. Unfortunately, very little is known about concrete parameters and lower bounds for such computational problems. At the same time, slight inaccuracies in estimating the hardness of the underlying problems can easily render constructions insecure - even constant improvements in algorithms which solve them can break parameters. Given this small room for errors, finding constructions which realize TLPs or VDFs from different assumptions is deemed prudent.

Since computational assumptions inherently have this disadvantage when being used for time-based primitives, an alternative is to rely on physical assumptions instead. First, the inherent noise in communication channels was shown to yield primitives such as Oblivious Transfer [33], Commitments [33] and Key Exchange [55]. Similar results were later obtained from physically-unclonable functions [21,60,66] and tamper-proof tokens [45,46]. More recently, secure vaults and one-time programs were constructed from protein polymers [3]. However, none of these assumptions seems to be useful when building time-based cryptography. Hence, we consider the minimal communication delay guaranteed by special relativity, which was first proposed as a way to construct commitments by Kent [49] in 1999. Recently, this assumption was used to construct more efficient commitments [52] and multi-prover Zero-Knowledge proofs [34], which have been experimentally demonstrated [2,67]. However, these constructions only guarantee security to verifiers who interact with provers via ideal secure channels, whereas the primitives we consider require non-interactive public verifiability.

1.1 Our Contributions

In this work, we investigate a new way of constructing time-based cryptographic primitives from physical assumptions (while additionally leveraging a classical

trust assumption). Our constructions derive their delay guarantees from special relativity, which posits that communication cannot happen faster than the speed of light. Thus, the communication delay between two parties is precisely lower bounded by their relative distance. However, this well known lower bound is practically meaningless when considering devices in close proximity. On the other hand, this fundamental physical delay becomes apparent when transmitting data over large distances, as it is the case for satellites in space. Delay guarantees may not hold if both devices are corrupted, but are achievable with at least one honest participant.

In order to obtain a delay lower bound sufficiently high for the usual applications of time-based cryptography, we consider the delay incurred by sending messages across a constellation of satellites placed far from each other. While this may seem far-fetched, the advent of relatively cheap CubeSats [63] has made it possible to easily deploy sizable constellations for specific applications, sparking initiatives towards satellite-based cryptographic applications [1]. It is therefore not unthinkable that satellite time could be rented from different satellite providers in the future, similar to how one rents cloud servers today[5]. Using satellites also allows any third party verifiers to ascertain communication delay lower bounds, since satellite positions are publicly observable[6]. However, our focus on satellites directly imposes a number of challenges. This is because their computational and communication resources are extremely limited, so our construction must be implementable from "cheap" cryptographic primitives. At the same time, potential corruptions of a subset of satellites within a larger constellation have an impact on the minimal delay that can be guaranteed.

From a cryptographic perspective, our work makes the following contributions:

Modelling dynamic delayed channels: We introduce a model for communication channels whose delay evolves with time in the UC framework [24]. For this, we use a Global clock to establish synchrony. As we rely on messages being transmitted through a constellation of satellites, our formal model can express communication delay among parties whose position changes over time, thus affecting the delay when transmitting a message between them.

Proofs of Sequential Communication Delay: Building on our model, we introduce techniques for proving that a certain message has been sequentially transmitted among a number of parties. We analyse the delay bounds obtained by composing delayed channels and propose the notion (and a construction) of proofs of sequential communication delay using signatures.

The first Ordered Multisignature based on OMDL: As a tool for efficient proofs of sequential communication delay, we introduce the first ordered

[5] Having satellites owned by different companies participating also shows that an assumed threshold on corruptions in a constellation of satellites is realistic.

[6] There are many ways to track satellites and learn their positions. Even spy satellites are tracked by amateur enthusiasts (e.g. `https://gizmodo.com/how-you-can-track-every-spy-satellite-in-orbit-1685316357`). This shows it is possible to keep track of satellite positions that determine communication delays.

multisignature (OMS) scheme based on the hardness of the One-More Discrete Log assumption. Assuming broadcasting and preprocessing, our OMS enjoys constant-size signatures and computational complexity independent of the number of signers in the online phase. Previous non-interactive OMS or Sequential Aggregate Signature (SAS) constructions with constant-size signatures rely either on bilinear pairings [10,14,39,51,53] or trapdoor permutations [20,43,53,57]. The existing Schnorr-based SAS [32] has a linear growth of the signature size. We take a different approach by lifting the bandwidth-efficient Schnorr-based interactive multi-signature of [58] to OMS. We also revisit the OMS security model of [14] and put forth a general framework that handles complex corruption and forgery patterns. Our security model and construction are contributions of independent interest.

UC (Ordered) Multisignatures: We present the first formalization of (Ordered) Multisignatures in the UC model, and prove that our game-based construction fulfills it. This allows us to easily integrate it into our UC-secure proofs of sequential communication delay. Along the way, we also propose UC formalisations of multisignatures (MS) and interactive multisignatures (IMS).

(Stateful) VDF and PV-TLP based on Communication Delay: We use our proofs of communication delay to introduce the first constructions of VDFs and PV-TLPs based on communication delay and analyse their security in the UC framework. We directly construct VDFs from proofs of sequential communication delay (in the random oracle model) by extracting randomness from such proofs. While trapdoor VDFs [68] are sufficient for constructing a PV-TLP [8,40], we opt for constructing those directly from sequential communication delays in order to obtain more efficient schemes. As an application, we show that our PV-TLPs can be used to efficiently instantiate the randomness beacon from [8] so that expensive resources are only used in case of cheating.

Delay Encryption and Stateless VDF from Threshold Identity Based Encryption (IBE): We obtain Delay Encryption [22] by combining our proofs of sequential communication delay and an IBE scheme endowed with a threshold identity secret key generation protocol. To the best of our knowledge, this is the first Delay Encryption scheme not based on supersingular isogeny assumptions. We also use a similar technique to obtain a more efficient construction of VDFs.

1.2 Technical Overview

Modelling Communication Delays: We model delays in the Abstract Composable Time [9] framework with synchrony provided by a global clock $\mathcal{G}_{\mathsf{Clock}}$ inspired by [50], which we realize in the aforementioned framework. We start by modelling a single-use ideal functionality $\mathcal{F}_{\mathsf{dmt}}^{\Delta_{\mathtt{lo}},\Delta_{\mathtt{hi}}}$ for delayed message transmission with fixed minimum ($\Delta_{\mathtt{lo}}$) and maximum ($\Delta_{\mathtt{hi}}$) delay parameters. This functionality guarantees that the adversary does not learn the message until at least $\Delta_{\mathtt{lo}}$ ticks after it is sent by the sender. At the same time, the functionality guarantees that the receiver gets the message at most $\Delta_{\mathtt{hi}}$ ticks after sending. This means that as long as at least one of the channel participants is honest, the channel guarantees delay. Next, we model a multiple use delayed channel functionality $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta}}$ where the minimum and maximum delays for a message sent at

time t are dynamically determined by a function $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow \mathsf{f}_{\Delta}(t)$ according to the current time provided by $\mathcal{G}_{\mathsf{Clock}}$. We show that $\mathcal{F}^{\mathsf{f}_{\Delta}}_{\mathsf{mdmt}}$ can be realized based on $\mathcal{F}^{\Delta_{\mathtt{lo}},\Delta_{\mathtt{hi}}}_{\mathsf{dmt}}$. These functionalities can be composed in order to obtain a minimum delay guarantee for a message transmitted among multiple parties.

Proofs of Sequential Communication Delay: We define the notion of a proof of sequential communication delay $\pi_{\mathtt{lo}}$, which allows a third party verifier $\mathcal{V}_i$ to check that a given message m has been sent from $\mathcal{P}_S$ to $\mathcal{P}_R$ while incurring a minimum delay of $\Delta_{\mathtt{lo}}$. This is modelled using the functionality $\mathcal{F}^{\mathsf{f}_{\Delta}}_{\mathsf{SCD}}$. As we construct this functionality from $\mathcal{F}^{\mathsf{f}_{\Delta}}_{\mathsf{mdmt}}$, we inherit security guarantees of minimal delay even if one of the participants (sender or receiver) is dishonest. We then construct a simple protocol realizing $\mathcal{F}^{\mathsf{f}_{\Delta}}_{\mathsf{SCD}}$ in a synchronized setting with a global clock $\mathcal{G}_{\mathsf{Clock}}$, public key infrastructure $\mathcal{F}_{\mathsf{Reg}}$, a unique digital signature scheme $\mathcal{F}_{\mathsf{Sig}}$ and a delayed channel $\mathcal{F}^{\mathsf{f}_{\Delta}}_{\mathsf{mdmt}}$. In this protocol, $\mathcal{P}_S$ signs (m, t), *i.e.* the message to be sent concatenated with the time t when the message is sent, obtaining a signature $\sigma_{\mathcal{P}_S}$. $\mathcal{P}_S$ sends $(m, t, \sigma_{\mathcal{P}_S})$ through $\mathcal{F}^{\mathsf{f}_{\Delta}}_{\mathsf{mdmt}}$ to $\mathcal{P}_R$, who checks that $\sigma_{\mathcal{P}_S}$ is valid w.r.t. (m, t) and $\mathcal{P}_S$'s verification key. Moreover, $\mathcal{P}_R$ checks that it has received the message at a time t' such that $t + \Delta_{\mathtt{lo}} \leq t' \leq t + \Delta_{\mathtt{hi}}$, where $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow \mathsf{f}_{\Delta}(t)$, *i.e.* it checks that the claimed sending time t is consistent with the channels parameters and the time t' when it actually receives the message. If the checks pass, $\mathcal{P}_R$ generates a signature $\sigma_{\mathcal{P}_R}$ on $(m, t, \sigma_{\mathcal{P}_S})$, and a proof $\pi^{\mathcal{P}_S \to \mathcal{P}_R}_{\Delta_{\mathtt{lo}}} = (\sigma_{\mathcal{P}_S}, \sigma_{\mathcal{P}_R})$ along with $m, t, \Delta_{\mathtt{lo}}$. Any third party can verify whether the proof is valid w.r.t. a message m and parameters $t, \Delta_{\mathtt{lo}}$ by checking that $\mathcal{P}_S, \mathcal{P}_R$ are the parties transmitting through $\mathcal{F}^{\mathsf{f}_{\Delta}}_{\mathsf{mdmt}}$, that $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow \mathsf{f}_{\Delta}(t)$ and that $\sigma_{\mathcal{P}_S}, \sigma_{\mathcal{P}_R}$ are valid. While it is clear that the guarantees of $\pi_{\mathtt{lo}}$ do not hold if both $\mathcal{P}_S$ and $\mathcal{P}_R$ collude (and e.g. pool their signing keys), we show that delay of proof is still guaranteed if at most one of the parties is corrupted.

We then show how this simple protocol can be generalized to a chain of delay channels with multiple intermediate parties and optimized to obtain proofs of constant size (the size of a signature) by using SAS [53] or OMS [14]. Our definition of $\mathcal{F}^{\mathsf{f}_{\Delta}}_{\mathsf{SCD}}$ is broad enough that this functionality can also model this setting. Here, the parameters of the individual delay channels of the participating parties, their number as well as the threshold of parties that can be corrupted allow us to prove which delay function $\mathsf{f}_{\Delta}(\cdot)$ the sequential proof will guarantee.

Interactive Ordered Multi-Signature from OMDL: Our OMS construction can be seen as a round-efficient interactive Schnorr-based multisignature where order is enforced by having each signer check the so-far aggregated signature (created by the preceding co-signers). It is obtained by carefully adjusting the recent MuSig2 [58] to comply with the ordered-signing setting. This yields constant-size signatures while keeping the computational complexity of each signer in the online phase (i.e. once they receive a message to be signed) independent of the total number of parties. We inherit the offline-online setup of MuSig2, and signers can preprocess the first round of communication before receiving the message. This motivates us to revisit existing (game-based) security models for OMS and introduce a generalized syntax that captures interactive OMS and explicitly supports this setting.

UC-security for (Ordered) Multi-Signatures: While it is possible to directly use our OMS construction to obtain an efficient proof of sequential communication, reducing security of the overall UC protocol to the game-based security is cumbersome. To circumvent this, we present formalizations of Multisignatures, Interactive Multisignatures as well as OMS in the UC framework, and show that our Interactive OMS from OMDL realizes the functionality. These differ quite strongly from existing models of UC-secure signatures [4, 23, 26] or threshold signatures [29], so we devise a new framework to model dynamic key registration, preprocessing as well as order during signing accurately.

Verifiable Delay Functions from Sequential Communication: We obtain a direct construction of a VDF from proofs of sequential communication delay. Our construction departs from $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$, a random oracle and a bulletin board $\mathcal{F}_{\mathsf{BB}}$, which is used to keep the state of VDF evaluations. The core idea is to send the VDF input in from a sender $\mathcal{P}_S$ to a receiver $\mathcal{P}_R$ via $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$, obtaining a proof of sequential communication delay $\pi_{\mathtt{lo}}$, which is also the proof of VDF evaluation. The output of the VDF is determined by querying the random oracle on $in|\pi_{\mathtt{lo}}$. Verification can be done by checking $\pi_{\mathtt{lo}}$ is valid for a given minimal delay $\Delta_{\mathtt{lo}}$ and recomputing the output. The first $\pi_{\mathtt{lo}}$ is written to the bulletin board and retrieved for future evaluations of the VDF to avoid multiple valid evaluations (*i.e.* sending in through $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ again to get a different $\pi_{\mathtt{lo}}$).

Publicly Verifiable Time-Lock Puzzles from Sequential Communication: We construct a PV-TLP protocol where a puzzle is a ciphertext $\mathtt{puz}$ obtained by encrypting a message m under the public key pk of a threshold encryption scheme. The parties $\mathcal{P}_i$ who have the corresponding secret key shares sk_i are connected via delayed channels $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$. A PV-TLP is solved by threshold decrypting $\mathtt{puz}$ via delayed channels $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ following a specific sequence of parties $\{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n\}$ where $\mathcal{P}_i$ aggregates its decryption share to $\mathcal{P}_{i-1}$'s decryption share before passing it on to $\mathcal{P}_{i+1}$. The delay guarantee comes from our analysis of sequential communication delay, as honest parties $\mathcal{P}_i$ check that the ciphertext has traversed the path from $\mathcal{P}_1$ to $\mathcal{P}_{i-1}$ before aggregating their decryption share, which guarantees a minimum delay. In order to obtain a publicly verifiable proof that a puzzle $\mathtt{puz}$ contained a message m, we employ the random oracle based transformation of [42, 62], where decryption yields not only m but the unique randomness used to generate $\mathtt{puz}$. This randomness constitutes our proof, since together with m it can be used to do a re-encryption check.

Delay Encryption and Stateless VDF from Threshold IBE: We use an IBE scheme with a protocol that allows parties who hold shares of the master secret key to efficiently generate the secret key for a given identity. In order to construct Delay Encryption, we define encryption under a given identity as IBE encryption under that identity. Later on, key extraction is done by having parties jointly generate the secret key for that identity in a round-robin manner along with a proof of sequential communication delay showing that this key generation had a certain minimum delay. We then adapt this technique to obtain a unique signature using Naor's transform from IBE to signatures, which yields a threshold unique signature with a proof of sequential communication delay

attesting the minimum delay for signature generation. The final VDF can be constructed by applying a random oracle to the signature and using the signature of communication delay and the signature itself to verify the VDF output. This construction solves the caveat of our first simple construction, since it always yields the same output for each input without requiring any parties to keep state.

2 Preliminaries

Notation We denote the computational (resp. statistical) security parameter by τ (resp. λ), the concatenation of two strings a and b by $a|b$, and compact multiple concatenations by $(a_i)_{i=1}^n = a_1|a_2|\dots|a_n$.

Auxiliary Background Material. In Appendix A, we give an overview of the UC framework [24] and present standard functionalities for Public Key Infrastructures ($\mathcal{F}_{\mathsf{Reg}}$), (unique) digital signatures ($\mathcal{F}_{\mathsf{Sig}}$), bulletin boards ($\mathcal{F}_{\mathsf{BB}}$) and global random oracles ($\mathcal{G}_{\mathsf{rpoRO}}$), which we will use in our constructions.

UC Secure Public-Key Encryption with Plaintext Verification. It is observed in [7] that it is possible to UC-realize public-key encryption with a plaintext verification property using the random oracle-based IND-CCA secure public-key encryption schemes of [42, 62]. This plaintext verification property allows a party who decrypts a ciphertext to generate a non-interactive publicly verifiable proof that a certain plaintext was obtained. We will apply the approach of [7] to obtain a threshold public-key encryption scheme with the same plaintext verification property. In order to do so, we use the fact that the encryption schemes of [42,62] can be obtained from any partially trapdoor one-way function, which allows us to depart from a simple threshold version of El Gamal to obtain a UC-secure theshold encryption scheme with plaintext verification. In Appendix A.1 we recall in verbatim form the definitions of the schemes from [42, 62] and the necessary properties for obtaining plaintext verification as presented in [7].

2.1 Modelling Time and Global Clocks

Previous works on universally composable PV-TLPs and VDFs based on sequential computation [8,9] have been cast in the abstract composable time model from TARDIS [9]. This model expresses time within the GUC framework in such a way that protocols can be made oblivious to clock ticks, which allows for modelling the passage of time without implying synchronicity, a generality that has also been exploited, *e.g.*, in the context of delayed adaptive corruptions [54]. Although we work with physical delay rather than sequential computation assumptions, we cast our results in the TARDIS model so that our results can be compared and/or combined[7] with those of [8,9] and extended to models of [54].

[7] For example, since it is hard and costly to tamper with a satellite in orbit, one can combine the physical delay guarantees of special relativity with computational delay guarantees provided by an on-board non-programmable computational device (*e.g.* an ASIC) that solves hard sequential problems (*e.g.* iterated squaring) with well-known runtimes. Since it is infeasible to update the internal device, later advances that speed up these runtimes do not affect the on-board computation.

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

$\mathcal{G}_{\mathsf{Clock}}$ is parameterized by a variable ν, sets $\mathcal{P}, \mathcal{F}$ of parties and functionalities respectively. It keeps a Boolean variable $d_{\mathcal{J}}$ for each $\mathcal{J} \in \mathcal{P} \cup \mathcal{F}$, a counter ν as well as an additional variable $\mathtt{u}$. All $d_{\mathcal{J}}$, ν and $\mathtt{u}$ are initialized as 0.

Clock Update: Upon receiving a message (UPDATE) from $\mathcal{J} \in \mathcal{P} \cup \mathcal{F}$: Set $d_{\mathcal{J}} = 1$. If $d_F = 1$ for all $F \in \mathcal{F}$ and $d_p = 1$ for all honest $p \in \mathcal{P}$, set $\mathtt{u} \leftarrow 1$ if it is 0.

Clock Read: Upon receiving a message (READ) from any entity: If $\mathtt{u} = 1$ then first send $(\textsc{Tick}, \mathsf{sid})$ to $\mathcal{S}$. Next set $\nu \leftarrow \nu + 1$, reset $d_{\mathcal{J}}$ to 0 for all $\mathcal{J} \in \mathcal{P} \cup \mathcal{F}$ and reset $\mathtt{u}$ to 0. Answer the entity with $(\textsc{Read}, \nu)$.

Fig. 1: Functionality $\mathcal{G}_{\mathsf{Clock}}$ for a Global Clock.

Global Tickers: In [8, 9], a global ticker functionality $\mathcal{G}_{\mathsf{ticker}}$ (see Appendix A.2) keeps track of “ticks” representing a discrete unit of time. When activated by another ideal functionality, the global ticker answers whether or not a new “tick” has happened since the last time it was activated by this ideal functionality but *does not provide a synchronized clock value.* To ensure that all honest parties can observe all relevant timing-related events, $\mathcal{G}_{\mathsf{ticker}}$ only progresses if all honest parties have signaled that they have been activated (in arbitrary order). Parties do not get outputs from $\mathcal{G}_{\mathsf{ticker}}$. Ticked functionalities can freely interpret ticks and perform arbitrary internal state changes. Upon each activation, any ticked ideal functionality first checks with $\mathcal{G}_{\mathsf{ticker}}$ if a new tick has happened and if yes, executes code in a special **Tick** interface. In a protocol realizing a ticked functionality, parties activate the global ticker after executing their steps, so that a new tick is allowed to happen. We refer to [9] for more details.

Global Clocks: We need to assume that honest parties have synchronized clocks. This is necessary to argue about evolving communication delays with respect to specific instants in time, which we need to construct proofs of sequential communication delays. We capture this notion of synchronicity by using a global clock functionality $\mathcal{G}_{\mathsf{Clock}}$ (see Fig. 1), following the ideas of [5, 48, 50]. $\mathcal{G}_{\mathsf{Clock}}$ allows parties and functionalities to request the current value of a synchronized time counter, which is only incremented if all honest parties agree to update the clock. This also means that *e.g.* ticks cannot happen randomly in protocol steps, unless parties in the protocol explicitly query $\mathcal{G}_{\mathsf{Clock}}$ to continue. We explain in Appendix A.2 how $\mathcal{G}_{\mathsf{Clock}}$ can be realized in the framework of [9].

3 Modeling Communication Delays

We model physical communication between two parties as authenticated message transmission ideal functionalities that ensure both minimal and maximal communication delays. This is in line with communication in the UC framework, that always happens through channel functionalities. Moreover, we allow any third party to observe the minimum and maximum delay bounds for a message transmitted through the functionality. This implicitly assumes that the parties know each others’ positions (in order to compute the delays) which is a reasonable assumption for satellites and base stations as outlined in the introduction.

Functionality $\mathcal{F}_{\mathsf{dmt}}^{\Delta_{\mathtt{lo}},\Delta_{\mathtt{hi}}}$

This functionality is parameterized by a minimal delay $\Delta_{\mathtt{lo}} > 0$ and a maximal delay $\Delta_{\mathtt{hi}} > \Delta_{\mathtt{lo}}$; it interacts with a sender $\mathcal{P}_S$, a receiver $\mathcal{P}_R$, an adversary $\mathcal{S}$, and the clock $\mathcal{G}_{\mathsf{Clock}}$. At initialisation t is set to 0, and the flags msg, $\mathsf{released}$, done to $\bot$.

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

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

Release message: Upon receiving an input $(\textsc{ok}, \mathsf{sid})$ from $\mathcal{S}$, do:
- If $\mathsf{msg} = \bot$ or $t < \Delta_{\mathtt{lo}}$, then send $(\textsc{None}, \mathsf{sid})$ to $\mathcal{S}$.
- If $\mathsf{msg} = \top$, $t \geq \Delta_{\mathtt{lo}}$ and $\mathsf{released} = \bot$, then set $\mathsf{released} = \top$.
- If $\mathsf{released} = \top$, then send $(\textsc{None}, \mathsf{sid})$ to $\mathcal{S}$.

Tick: Sends $(\textsc{Read})$ to $\mathcal{G}_{\mathsf{Clock}}$, receiving $(\textsc{Read}, \bar{t})$ as answer. If $\bar{t}$ has changed since the last activaction:
- If $\mathsf{msg} = \bot$, then send $(\textsc{None}, \mathsf{sid})$ to $\mathcal{S}$.
- If $\mathsf{msg} = \top$ and $\mathsf{released} = \bot$, then set $t = t + 1$:
  - If $t = \Delta_{\mathtt{lo}}$ then send $(\textsc{Sent}, \mathsf{sid}, m, t)$ to $\mathcal{S}$.
  - If $t = \Delta_{\mathtt{hi}}$, set $\mathsf{released} = \top$ and send $(\textsc{Released}, \mathsf{sid})$ to $\mathcal{S}$.

Fig. 2: Ticked ideal functionality $\mathcal{F}_{\mathsf{dmt}}^{\Delta_{\mathtt{lo}},\Delta_{\mathtt{hi}}}$ for authenticated message transmission with minimal delay $\Delta_{\mathtt{lo}}$ and maximal message delay $\Delta_{\mathtt{hi}}$.

We start by modelling a single-use delayed channel with fixed minimum and maximum delay parameters for simplicity. This channel captures the transmission of a single message between two parties at an specific point in time, which determines the delay parameters. As parties' relative positions evolve with time, so do the communication delay bounds as their relative distances change. We therefore, based on the single-use delayed channel, construct a multi-use functionality whose delay bounds can evolve with the ticks of $\mathcal{G}_{\mathsf{Clock}}$. This multi-use channel allows other parties to observe the delay bounds for a message transmitted at a given (past or future) point in time, which will later be necessary for verifying the output of a time-based primitive constructed over the channels, as well as estimating the delay guaranteed by a future evaluation of such a primitive.

Single-Use Channel ideal functionality $\mathcal{F}_{\mathsf{dmt}}^{\Delta_{\mathtt{lo}},\Delta_{\mathtt{hi}}}$: As a warm-up example, we present the functionality $\mathcal{F}_{\mathsf{dmt}}^{\Delta_{\mathtt{lo}},\Delta_{\mathtt{hi}}}$ for delayed authenticated message transmission in Fig. 2. The message delivery is at least $\Delta_{\mathtt{lo}}$ ticks (*i.e.* the physical bound for message transmission), and this delay holds also for an adversarial receiver. The adversary cannot force transmission to be delayed by more than $\Delta_{\mathtt{hi}}$ ticks if it is the sender, and cannot force delivery before $\Delta_{\mathtt{lo}}$ ticks.

Multiple-Use Channel ideal functionality $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$: Manually keeping track of what instance of $\mathcal{F}_{\mathsf{dmt}}^{\Delta_{\mathtt{lo}},\Delta_{\mathtt{hi}}}$ to use (along with its parameters $\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}$) every time a message needs to be sent between two parties, as well as the current time, would

make protocol descriptions very cumbersome. Hence, we present a higher level abstraction of a multiple-use delayed authenticated channel that automatically assigns minimum and maximum delays to each message according to the time it is sent. In Fig. 3 we present the functionality $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ for multiple-use delayed authenticated message transmission. The main parameter of this functionality is a function f_Δ that takes as input a time t and outputs the minimum delay $\Delta_{\mathtt{lo}}$ and maximum delay $\Delta_{\mathtt{hi}}$ for a message sent at time t. When it is requested to transmit a message, $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ determines the current time by contacting $\mathcal{G}_{\mathsf{Clock}}$ and computes $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(t)$. Next, the functionality registers the message in a list and ensures that it is not revealed to the adversary before a minimum delay $\Delta_{\mathtt{lo}}$, while guaranteeing delivery to an honest receiver within a maximum delay $\Delta_{\mathtt{hi}}$. Moreover, $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ allows for any third party to obtain the delay parameters for messages sent at a given clock tick, as f_Δ is a public parameter of the functionality (similar to $\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}$ in $\mathcal{F}_{\mathsf{dmt}}^{\Delta_{\mathtt{lo}},\Delta_{\mathtt{hi}}}$).

To model predictability of delay, we require that the variance between any two ticks in delay - as modeled by f_Δ - cannot be too much: no adversary should be able to send a message *faster by waiting until a later tick* (i.e. time travel of messages is not possible). To capture this, we give the following definition:

Definition 3.1 (Permissible Delay Function). *A function* $f_\Delta : \{0, \ldots, \mathsf{poly}(\tau)\} \rightarrow \mathbb{N} \times \mathbb{N}$ *models permissible delay if*

$$\forall t \in \mathbb{N} : (\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow f_\Delta(t), (\Delta'_{\mathtt{lo}}, \Delta'_{\mathtt{hi}}) \leftarrow f_\Delta(t+1) \Rightarrow \Delta'_{\mathtt{lo}} - \Delta_{\mathtt{lo}} > -1.$$

Realizing $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ from $\mathcal{F}_{\mathsf{dmt}}^{\Delta_{\mathtt{lo}},\Delta_{\mathtt{hi}}}$ In Appendix B.1 we present a protocol that realises the multiple-use ideal functionality for authenticated delayed message transmission using $\mathcal{G}_{\mathsf{Clock}}$ and multiple $\mathcal{F}_{\mathsf{dmt}}^{\Delta_{\mathtt{lo}},\Delta_{\mathtt{hi}}}$, and prove its security. The protocol uses one instance of $\mathcal{F}_{\mathsf{dmt}}^{\cdot}$ for each possible timestamp. The sender simply picks the correct instance for message transmission, while the verifier for every clock tick tests 1. if any of the instances delivers a message to him; and 2. if the message's time of sending and delay are consistent.

4 Proofs of Sequential Communication Delays

In this section, we introduce techniques for producing a publicly verifiable proof $\pi_{\mathtt{lo}}$ that a message m has incurred a certain minimum delay due to being transmitted from party $\mathcal{P}_S$ to party $\mathcal{P}_R$. Such a proof, when using the delay channel functionalities from Section 3, requires that at least one of the two parties involved in the process was honest. The idea is to have both the sender $\mathcal{P}_S$ and receiver $\mathcal{P}_R$ of a delayed channel sign the input message and the initial timestamp when this message was sent (provided that the message is received within reasonable time constraints such that the initial timestamp is not too far in the future or past). Both signatures and the initial timestamp form the proof $\pi_{\mathtt{lo}}$ showing that the message was sent from $\mathcal{P}_S$ to $\mathcal{P}_R$ incurring a given minimum delay as observed by an honest party. This is guaranteed by the delayed channel, whose minimum delay is determined by the timestamp.

Functionality $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$

This functionality is parameterized by a computational security parameter τ and a permissible delay function $\mathsf{f}_\Delta : \{0, \ldots, \mathsf{poly}(\tau)\} \to \mathbb{N} \times \mathbb{N}$; it interacts with $\mathcal{G}_{\mathsf{Clock}}$, sender $\mathcal{P}_S$, receiver $\mathcal{P}_R$ and adversary $\mathcal{S}$. At initialisation the list L is empty.
In any call below, $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ first sends (READ) to $\mathcal{G}_{\mathsf{Clock}}$ and obtains $(\text{READ}, \bar{t})$.

Send: Upon first message $(\text{SEND}, \mathsf{sid}, m)$ for $\bar{t}$ from party $\mathcal{P}_S$ add $(m, \bar{t}, \perp)$ to L.
Receive: Upon receiving $(\text{REC}, \mathsf{sid})$ from $\mathcal{P}_R$, for every $(m, t, \mathsf{released}) \in L$, if $\mathsf{released} = \top$ (*i.e.* the maximum delay has passed or the adversary released the message), remove $(m, t, \mathsf{released})$ from L and send $(\text{SENT}, \mathsf{sid}, m, t)$ to $\mathcal{P}_R$.
Release message: Upon receiving an input $(\text{OK}, \mathsf{sid}, t)$ from $\mathcal{S}$ compute $(\Delta_{\mathtt{lo}}, \cdot) \leftarrow \mathsf{f}_\Delta(t)$. If there is $(m, t, \mathsf{released}) \in L$ such that $\bar{t} \geq t + \Delta_{\mathtt{lo}}$ then set $\mathsf{released} = \top$.
Tick: For every $(m, t, \mathsf{released}) \in L$ compute $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(t)$ and do as follows:
- If $t + \Delta_{\mathtt{lo}} = \bar{t}$, send $(\text{SENT}, \mathsf{sid}, m, t)$ to $\mathcal{S}$.
- If $t + \Delta_{\mathtt{hi}} = \bar{t}$, set $\mathsf{released} = \top$.

Fig. 3: Ticked functionality $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ for authenticated message transmission with evolving delays.

We then use a sequence of consecutive communication channels between multiple parties in order to obtain a larger provable minimum delay than that provided by a single channel without intermediaries. Here, a message m travels from sender $\mathcal{P}_1$ to receiver $\mathcal{P}_n$, through hops $\mathcal{P}_i$. Each intermediate party $\mathcal{P}_i$ sends to $\mathcal{P}_{i+1}$ not only the original message m, but also a proof showing that m travelled from $\mathcal{P}_1$ to $\mathcal{P}_i$. If m and the proof arrive at $\mathcal{P}_i$ at a certain time that is not consistent with the minimum and maximum delays of the channels connecting $\mathcal{P}_1$ to $\mathcal{P}_i$ (*i.e.* it is too far in the future or in the past), $\mathcal{P}_i$ aborts. This construction can be leveraged to obtain a final proof of sequential communication delay consisting of $(m, t, \sigma_1, \ldots, \sigma_{i-1})$, where signature σ_i is generated by party $\mathcal{P}_i$, and t is the initial timestamp when m was sent. Finally, we discuss how to use sequentially aggregate signatures (SAS) and ordered multi signatures (OMS) to optimize this construction and avoid proofs of sequential communication delay of size linear in the number of network nodes.

4.1 Modelling Proofs of Sequential Communication Delay

We begin by modeling a publicly verifiable proof of delay through an ideal functionality $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ depicted in Fig. 4. This functionality incorporates the delayed channel modelled by $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$, and proof generation/verification mechanisms similar to those of the unique digital signature functionality $\mathcal{F}_{\mathsf{Sig}}$ (Fig. 16). Departing from $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$, which allows for a $\mathcal{P}_S$ to send a message m to $\mathcal{P}_R$ with minimum and maximum delays $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(t)$ depending on time t, $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ delivers to $\mathcal{P}_R$ the proof $\pi_{\mathtt{lo}}$ that m was sent at time t with a minimum delay $\Delta_{\mathtt{lo}}$.

In $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$, the adversary may only generate valid proofs of delay after the minimal delay of $\pi_{\mathtt{lo}}$, but it learns m earlier than the honest receiver. This makes sense because the statement that the message m has traveled for a certain delay does not mean that m was only learnt by the adversary with that delay. For

Functionality $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$

$\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ keeps initially empty lists L, L_π, and is parameterized by a computational security parameter τ and a permissible delay function f_Δ. $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ interacts with $\mathcal{G}_{\mathsf{Clock}}$, sender $\mathcal{P}_S$, receiver $\mathcal{P}_R$, verifiers $\mathcal{V}$ and adversary $\mathcal{S}$.

In any call below, $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ first sends (READ) to $\mathcal{G}_{\mathsf{Clock}}$ and obtains $(\text{READ}, \bar{t})$.

Send: Upon receiving an input $(\text{SEND}, \mathsf{sid}, m)$ from an honest $\mathcal{P}_S$ and if this is the first such message in this tick-round:

1. Compute $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(\bar{t})$ and add $(\bar{t}, m, \perp, \Delta_{\mathtt{lo}})$ to L.
2. Output $(\text{MESSAGE}, \mathsf{sid}, \bar{t}, m)$ to $\mathcal{S}$.

If $\mathcal{P}_S$ is corrupted, then upon input $(\text{SEND}, \mathsf{sid}, m, t)$ from $\mathcal{S}$ compute $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(t)$. If $\Delta_{\mathtt{hi}} + t - \bar{t} \geq \Delta_{\mathtt{lo}}$ then add $(t, m, \perp, \Delta_{\mathtt{lo}})$ to L.

Receive: Upon receiving $(\text{REC}, \mathsf{sid})$ from $\mathcal{P}_R$, for every $(t, m, \top, \mathtt{cnt}) \in L$:

1. Remove $(t, m, \mathsf{released}, \mathtt{cnt})$ from L and recompute $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(t)$.
2. If $(m, t, \Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}, \pi_{\mathtt{lo}}, 1) \in L_\pi$ send $(\text{SENT}, \mathsf{sid}, m, t, \bar{t} - t, \pi_{\mathtt{lo}})$ to $\mathcal{P}_R$.
3. Else, send $(\text{PROOF}, \mathsf{sid}, m, t, \bar{t} - t)$ to $\mathcal{S}$. Upon receiving $(\text{PROOF}, \mathsf{sid}, m, t, \pi_{\mathtt{lo}})$ from $\mathcal{S}$, check that $(m, t, \Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}, \pi_{\mathtt{lo}}, 0) \notin L_\pi$. If yes, output $\perp$ to $\mathcal{P}_S/\mathcal{P}_R$ and halt. Else, add $(m, t, \Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}, \pi_{\mathtt{lo}}, 1)$ to L_π and send $(\text{SENT}, \mathsf{sid}, m, t, \bar{t} - t, \pi_{\mathtt{lo}})$ to $\mathcal{P}_R$. If $\mathcal{S}$ sends $(\text{NOPROOF}, \mathsf{sid})$ then output $(\text{NOPROOF}, \mathsf{sid})$.

Release message: Upon receiving an input $(\text{OK}, \mathsf{sid}, t)$ from $\mathcal{S}$ compute $(\Delta_{\mathtt{lo}}, \cdot) \leftarrow \mathsf{f}_\Delta(t)$. If there is $(t, m, \mathsf{released}, \mathtt{cnt}) \in L$ such that $\bar{t} \geq t + \Delta_{\mathtt{lo}}$ and $\mathtt{cnt} = 0$ then set $\mathsf{released} = \top$.

Verify: Upon receiving $(\text{VERIFY}, \mathsf{sid}, m, t, \Delta, \pi_{\mathtt{lo}})$ from $\mathcal{V}_i$, send $(\text{VERIFY}, \mathsf{sid}, m, t, \Delta, \pi_{\mathtt{lo}})$ to $\mathcal{S}$. Upon receiving $(\text{VERIFIED}, \mathsf{sid}, m, t, \Delta, \pi_{\mathtt{lo}}, \phi)$ from $\mathcal{S}$ do:

1. If $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(t)$ and $\Delta \notin [\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}]$ or then set $f = 0$. Otherwise set $f = 1$. (is delay in allowed interval?)
2. If $t + \Delta_{\mathtt{lo}} > \bar{t}$ then set $f = 0$ (no verification request can be positive, unless m has circulated for at least $\Delta_{\mathtt{lo}}$ ticks)
3. If $\phi = 1$ and there is an entry $(m, t, \Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}, \pi'_{\mathtt{lo}}, 1) \in L_\pi$ where $\pi'_{\mathtt{lo}} \neq \pi_{\mathtt{lo}}$ then set $f = 0$. (any proof of delay must be unique)
4. If there is an entry $(m, t, \Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}, \pi_{\mathtt{lo}}, f') \in L_\pi$, let $b = f \wedge f'$. (All verification requests with identical parameters will result in the same answer.)
5. If no such entry is present, set $b = f \wedge \phi$ and add $(m, t, \Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}, \pi_{\mathtt{lo}}, b)$ to L_π. (Add for consistency)

Output $(\text{VERIFIED}, \mathsf{sid}, m, t, \Delta, b)$ to $\mathcal{V}_i$.

Tick: For every $(t, m, \mathsf{released}, \mathtt{cnt}) \in L$ compute $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(t)$. If $t + \Delta_{\mathtt{hi}} = \bar{t}$, set $\mathsf{released} = \top$. If $\mathtt{cnt} > 0$ then reduce $\mathtt{cnt}$ by 1.

Fig. 4: Ticked functionality $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ for proofs of sequential communication delay.

an example in practice, consider a chain of 4 parties with 3 intermediate delay channels. If e.g. $\mathcal{P}_2$ and $\mathcal{P}_4 = \mathcal{P}_R$ are both corrupted, then the adversary must of course learn m once it arrives at $\mathcal{P}_2$. The guarantee of the functionality is that *the proof of delay* will only arrive at the adversary with the required minimal delay, and that an *honest* receiver will have to potentially wait longer to receive it and m. This is because the message still has to pass through channels that have honest parties as senders and receivers before a proof is generated.

A second interesting property is that a corrupted sender is allowed to *date back* message sending by a certain amount of ticks, i.e. at time $\bar{t}$ it is allowed

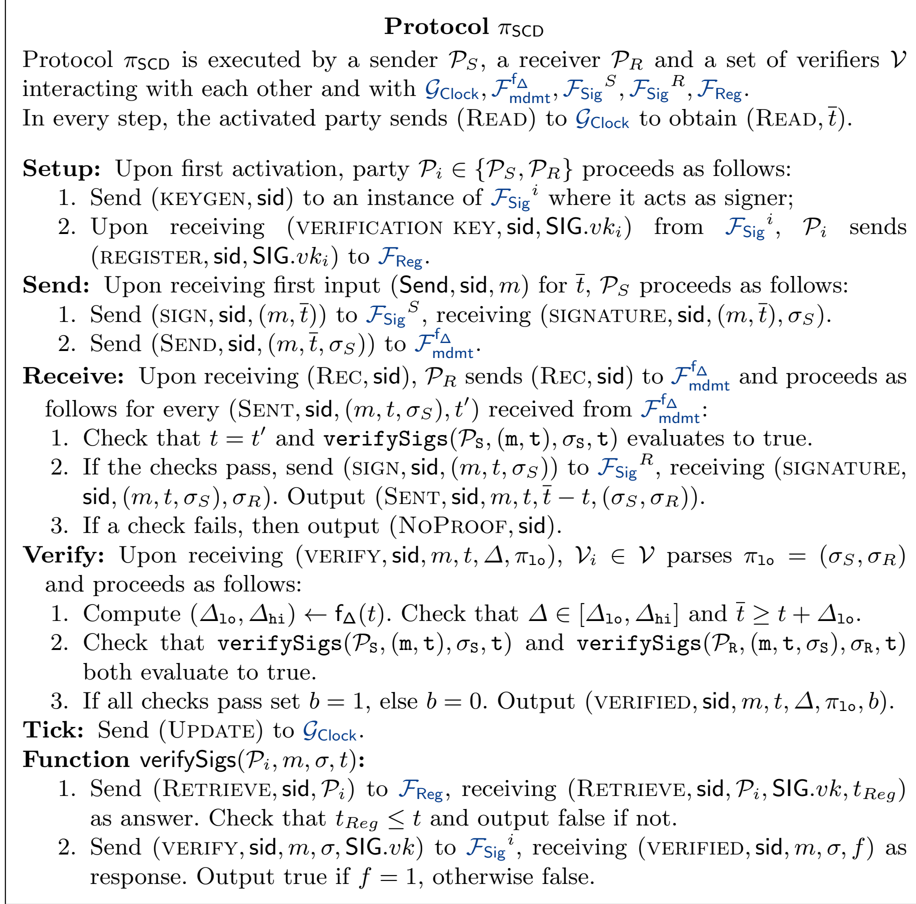
to say that it sent the message already at time $t < \bar{t}$. It can do so as long as $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ can still delay proof (and message) delivery by $\Delta_{\mathtt{lo}}$ ticks without exceeding time $t + \Delta_{\mathtt{hi}}$ during delivery. The reason for this "time traveling" of dishonest senders is the multiparty protocol. For example, consider a chain of parties where $\mathcal{P}_1 = \mathcal{P}_S$ and $\mathcal{P}_2$ are corrupted. In that case the simulator cannot extract any information from the channel between $\mathcal{P}_1$ and $\mathcal{P}_2$ as the adversary is of course not bound to use this channel. But it can still guarantee message delay as parties later on in the chain are honest, so their delay channels must have been used.

A third important property is that a proof that m was sent through the channel with a certain delay that is within $[\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}]$ is *unique* to the tuple (m, t), where t is the time when m was supposed to be sent. Moreover, $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ allows any verifier $\mathcal{V}_i$ to check that a proof $\pi_{\mathtt{lo}}$ of delay in $[\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}]$ for message m sent at time t is indeed valid (*i.e.* it has been generated honestly). Here, the adversary may define validity of a proof during verification even if $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ did not output the proof itself at that time. This is an artifact of our protocol, as a dishonest receiver $\mathcal{P}_R$ must not make his contributions public until when the proof gets verified. This is standard behavior in other UC functionalities, such as the signature functionality $\mathcal{F}_{\mathsf{Sig}}$.

Proofs of Sequential Communication Delay with 2 parties. We construct a simple protocol that realizes $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ between two parties by leveraging a delayed channel $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$, a Public Key Infrastructure $\mathcal{F}_{\mathsf{Reg}}$ and a unique digital signature $\mathcal{F}_{\mathsf{Sig}}$ on a synchronized network (with synchrony maintained by $\mathcal{G}_{\mathsf{Clock}}$). In it, both the sender $\mathcal{P}_S$ and receiver $\mathcal{P}_R$ sign the message m being transmitted. However, we need to take steps to guarantee that an honest $\mathcal{P}_R$ does not inadvertently help a corrupted $\mathcal{P}_S$ forge a proof for an invalid initial timestamp t or minimum delay $\Delta_{\mathtt{lo}}$. In order to to avoid this issue, $\mathcal{P}_R$ needs to verify that m has been received through an instance of $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ where $\mathcal{P}_S$ acts as sender at a timestamp between $t + \Delta_{\mathtt{lo}}$ and $t + \Delta_{\mathtt{hi}}$, where t is the initial timestamp when the message was sent and $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(t)$. Since $\mathcal{P}_R$ needs to know t in order to obtain $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}})$, we have $\mathcal{P}_S$ sign not only m but (m, t), allowing $\mathcal{P}_R$ to perform its delay consistency checks. If $\mathcal{P}_R$ is satisfied, it then signs (m, t, σ_S), where σ_S is $\mathcal{P}_S$'s signature, and outputs both $\mathcal{P}_S$'s signature and its own as the proof of sequential communication delay. Verifying such a proof of sequential communication delay can be done by any third party by simply verifying the signatures generated $\mathcal{P}_S$ and $\mathcal{P}_R$, as well as checking consistency of the timestamps. The protocol is presented in Figure 5.

Theorem 4.1. π_{SCD} *(Fig. 5) UC-realizes* $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ *in the* $\mathcal{G}_{\mathsf{Clock}}, \mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}, \mathcal{F}_{\mathsf{Sig}}, \mathcal{F}_{\mathsf{Reg}}$*-hybrid model against a static active adversary corrupting at most one of* $\mathcal{P}_S, \mathcal{P}_R$.

The proof can be found in Appendix B.2 and is rather straightforward. For a corrupted sender, extract the message m from $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ but ensure that verification keys are registered and that it would later be accepted by an honest receiver. For a corrupted receiver, program $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ to output the correctly signed message at the right time. In this case, verification is more involved as upon querying **Verify** the signature used by the dishonest receiver might be undefined.



Protocol π_{SCD}

Protocol π_{SCD} is executed by a sender $\mathcal{P}_S$, a receiver $\mathcal{P}_R$ and a set of verifiers $\mathcal{V}$ interacting with each other and with $\mathcal{G}_{\mathsf{Clock}}, \mathcal{F}^{\mathsf{f}_\Delta}_{\mathsf{mdmt}}, {\mathcal{F}_{\mathsf{Sig}}}^S, {\mathcal{F}_{\mathsf{Sig}}}^R, \mathcal{F}_{\mathsf{Reg}}$.
In every step, the activated party sends (Read) to $\mathcal{G}_{\mathsf{Clock}}$ to obtain (Read, $\bar{t}$).

Setup: Upon first activation, party $\mathcal{P}_i \in \{\mathcal{P}_S, \mathcal{P}_R\}$ proceeds as follows:

1. Send (keygen, sid) to an instance of ${\mathcal{F}_{\mathsf{Sig}}}^i$ where it acts as signer;
2. Upon receiving (verification key, $\mathsf{sid}, \mathsf{SIG}.vk_i$) from ${\mathcal{F}_{\mathsf{Sig}}}^i$, $\mathcal{P}_i$ sends (register, $\mathsf{sid}, \mathsf{SIG}.vk_i$) to $\mathcal{F}_{\mathsf{Reg}}$.

Send: Upon receiving first input ($\mathsf{Send}, \mathsf{sid}, m$) for $\bar{t}$, $\mathcal{P}_S$ proceeds as follows:

1. Send (sign, $\mathsf{sid}, (m, \bar{t})$) to ${\mathcal{F}_{\mathsf{Sig}}}^S$, receiving (signature, $\mathsf{sid}, (m, \bar{t}), \sigma_S$).
2. Send (Send, $\mathsf{sid}, (m, \bar{t}, \sigma_S)$) to $\mathcal{F}^{\mathsf{f}_\Delta}_{\mathsf{mdmt}}$.

Receive: Upon receiving (Rec, sid), $\mathcal{P}_R$ sends (Rec, sid) to $\mathcal{F}^{\mathsf{f}_\Delta}_{\mathsf{mdmt}}$ and proceeds as follows for every (Sent, $\mathsf{sid}, (m, t, \sigma_S), t'$) received from $\mathcal{F}^{\mathsf{f}_\Delta}_{\mathsf{mdmt}}$:

1. Check that $t = t'$ and $\mathtt{verifySigs}(\mathcal{P}_\mathtt{S}, (\mathtt{m}, \mathtt{t}), \sigma_\mathtt{S}, \mathtt{t})$ evaluates to true.
2. If the checks pass, send (sign, $\mathsf{sid}, (m, t, \sigma_S)$) to ${\mathcal{F}_{\mathsf{Sig}}}^R$, receiving (signature, $\mathsf{sid}, (m, t, \sigma_S), \sigma_R$). Output (Sent, $\mathsf{sid}, m, t, \bar{t} - t, (\sigma_S, \sigma_R)$).
3. If a check fails, then output (NoProof, sid).

Verify: Upon receiving (verify, $\mathsf{sid}, m, t, \Delta, \pi_{\mathtt{lo}}$), $\mathcal{V}_i \in \mathcal{V}$ parses $\pi_{\mathtt{lo}} = (\sigma_S, \sigma_R)$ and proceeds as follows:

1. Compute $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(t)$. Check that $\Delta \in [\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}]$ and $\bar{t} \geq t + \Delta_{\mathtt{lo}}$.
2. Check that $\mathtt{verifySigs}(\mathcal{P}_\mathtt{S}, (\mathtt{m}, \mathtt{t}), \sigma_\mathtt{S}, \mathtt{t})$ and $\mathtt{verifySigs}(\mathcal{P}_\mathtt{R}, (\mathtt{m}, \mathtt{t}, \sigma_\mathtt{S}), \sigma_\mathtt{R}, \mathtt{t})$ both evaluate to true.
3. If all checks pass set $b = 1$, else $b = 0$. Output (verified, $\mathsf{sid}, m, t, \Delta, \pi_{\mathtt{lo}}, b$).

Tick: Send (Update) to $\mathcal{G}_{\mathsf{Clock}}$.

Function $\mathsf{verifySigs}(\mathcal{P}_i, m, \sigma, t)$**:**

1. Send (Retrieve, $\mathsf{sid}, \mathcal{P}_i$) to $\mathcal{F}_{\mathsf{Reg}}$, receiving (Retrieve, $\mathsf{sid}, \mathcal{P}_i, \mathsf{SIG}.vk, t_{Reg}$) as answer. Check that $t_{Reg} \leq t$ and output false if not.
2. Send (verify, $\mathsf{sid}, m, \sigma, \mathsf{SIG}.vk$) to ${\mathcal{F}_{\mathsf{Sig}}}^i$, receiving (verified, $\mathsf{sid}, m, \sigma, f$) as response. Output true if $f = 1$, otherwise false.

Fig. 5: Protocol π_{SCD} realizing $\mathcal{F}^{\mathsf{f}_\Delta}_{\mathsf{SCD}}$.

Proofs of Sequential Communication Delay with > 2 parties. We will now realize $\mathcal{F}^{\mathsf{f}_\Delta}_{\mathsf{SCD}}$ using a *longer chain of parties*. There, the sender $\mathcal{P}_1 = \mathcal{P}_S$ is connected to $\mathcal{P}_2$ using a delayed channel $\mathcal{F}^{\mathsf{f}_\Delta}_{\mathsf{mdmt}}$ with delay function $\mathsf{f}_{\Delta,1}$, $\mathcal{P}_2$ is connected to $\mathcal{P}_3$ via $\mathcal{F}^{\mathsf{f}_\Delta}_{\mathsf{mdmt}}$ with $\mathsf{f}_{\Delta,2}$ until $\mathcal{P}_{n-1}$, which is connected via $\mathcal{F}^{\mathsf{f}_\Delta}_{\mathsf{mdmt}}$ to $\mathcal{P}_n = \mathcal{P}_R$ with delay function $\mathsf{f}_{\Delta,n}$. As before, $\mathcal{P}_1$ signs m, t before sending it through $\mathcal{F}^{\mathsf{f}_\Delta}_{\mathsf{mdmt}}$, while $\mathcal{P}_2$ signs the output of $\mathcal{F}^{\mathsf{f}_\Delta}_{\mathsf{mdmt}}$ if it is valid and then forwards it with the signature via $\mathcal{F}^{\mathsf{f}_\Delta}_{\mathsf{mdmt}}$ to $\mathcal{P}_3$ etc. We will prove that such a chain again realizes an instance of $\mathcal{F}^{\mathsf{f}_\Delta}_{\mathsf{SCD}}$, but with different delay parameters.

We consider malicious adversaries that can *interrupt signature generation* by refusing to execute the protocol. We assume that each party in the chain knows all the delay functions $\mathsf{f}_{\Delta,i}$ for each of the $\mathcal{F}^{\mathsf{f}_\Delta}_{\mathsf{mdmt}}$ instances in the chain, which allows them to compute delay bounds for incoming messages. In our protocol, $\mathcal{P}_i$ must establish that the message m that it obtained – which was supposedly initially sent at time t by $\mathcal{P}_1$ – *could be delivered to* $\mathcal{P}_{i-1}$ via instances of $\mathcal{F}^{\mathsf{f}_\Delta}_{\mathsf{mdmt}}$

with delay functions $\mathsf{f}_{\Delta 1}, \ldots, \mathsf{f}_{\Delta i-2}$ and incurring the respective delay, such that $\mathcal{P}_{i-1}$ sending it at time t_{i-1} via $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ with a delay modeled by $\mathsf{f}_{\Delta i-1}$ is plausible.

As an example, assume a chain of 3 parties where only $\mathcal{P}_3$ is honest. Let $(1,3) = \mathsf{f}_{\Delta,1}(t) = \mathsf{f}_{\Delta,2}(t)$ for every t, and assume that $\mathcal{P}_3$ obtains m from $\mathcal{P}_2$, which was supposedly sent at $t = 0$ by $\mathcal{P}_1$. $\mathcal{P}_3$ knows that m must travel a minimum time of 1 tick from $\mathcal{P}_1$ to $\mathcal{P}_2$ or at most 3 ticks. If the channel from $\mathcal{P}_2$ to $\mathcal{P}_3$ incurs delay between 1 and 3 ticks, but $\mathcal{P}_3$ obtains m at tick 7, then $\mathcal{P}_2$ has sent m the earliest at tick 4. This means that $\mathcal{P}_2$ is cheating as it delayed delivery of m. Alternatively, if $\mathcal{P}_3$ had obtained m at tick 1 then $\mathcal{P}_2$ must have sent the message at tick 0 (by the minimum delay $\mathsf{f}_{\Delta,2}$), which is also impossible as the message would have needed at least 1 tick from $\mathcal{P}_1$ to $\mathcal{P}_2$. Hence, we carefully specify *what each party verifies before signing* about timestamps and delivery times and how it impacts the proven delay given the corruption thresholds.

Plausible delays. We now introduce the $\mathsf{plausible}$ delay predicate $\mathsf{isP}(t_1, \mathsf{f}_{\Delta,1}, \ldots, \mathsf{f}_{\Delta,\ell-1}, t_\ell)$. It is defined for $\ell > 1$ as follows:

$\ell = 2$: true if $\exists \Delta \in \mathsf{f}_{\Delta,1}(t_1) : t_1 + \Delta = t_2$.
$\ell > 2$: true if $\exists \Delta \in \mathsf{f}_{\Delta,1}(t_1)$: $\mathsf{isP}(t_1 + \Delta, \mathsf{f}_{\Delta,2}, \ldots, \mathsf{f}_{\Delta,\ell-1}, t_\ell)$.

As we constrain the output of each $\mathsf{f}_{\Delta,i}$ to only be defined on polynomially many inputs, isP can be computed in polynomial time as long as $\ell = O(\log(\tau))$. This can be improved if, e.g. all f_Δ functions are constant in an obvious way.

We now show that we can combine two instances of isP into one:

Proposition 4.2. *Let $\mathsf{f}_{\Delta,1}, \ldots, \mathsf{f}_{\Delta,n-1}$ be permissible delay functions and let t_1, t_i, t_n be such that $\mathsf{isP}(t_1, \mathsf{f}_{\Delta,1}, \ldots, \mathsf{f}_{\Delta,i-1}, t_i)$ and $\mathsf{isP}(t_i, \mathsf{f}_{\Delta,i}, \ldots, \mathsf{f}_{\Delta,n-1}, t_n)$. Then $\mathsf{isP}(t_1, \mathsf{f}_{\Delta,1}, \ldots, \mathsf{f}_{\Delta,n-1}, t_n)$ holds.*

Conversely, we can also decompose every isP chain into its parts.

Proposition 4.3. *Let $\mathsf{f}_{\Delta,1}, \ldots, \mathsf{f}_{\Delta,n-1}$ be permissible delay functions and t_1, t_n be such that $\mathsf{isP}(t_1, \mathsf{f}_{\Delta,1}, \ldots, \mathsf{f}_{\Delta,n-1}, t_n)$ holds. For every $i \in \{2, \ldots, n-1\}$ there exists a t_i such that $\mathsf{isP}(t_1, \mathsf{f}_{\Delta,1}, \ldots, \mathsf{f}_{\Delta,i-1}, t_i)$ and $\mathsf{isP}(t_i, \mathsf{f}_{\Delta,i}, \ldots, \mathsf{f}_{\Delta,n-1}, t_n)$ hold.*

Proof. (of Propositions 4.2 & 4.3) The definition of isP implies that $\mathsf{isP}(t_1, \mathsf{f}_{\Delta,1}, \ldots, \mathsf{f}_{\Delta,n-1}, t_n)$ returns true if and only if $\exists\ \Delta_1, \ldots, \Delta_{n-1} : t_1 + \sum_{i=1}^{n-1} \Delta_i = t_n \wedge \Delta_i \in \mathsf{f}_{\Delta,i}\left(t_1 + \sum_{j=1}^{i-1} \Delta_j\right)$. Proposition 4.2 follows by combining both existential statements into one. Proposition 4.3 follows from setting $t_{i+1} = t_1 + \Delta_1 + \cdots + \Delta_i$. $\square$

We stress that verifying that a message *arrived at the receiver with plausible delay* does not imply that it indeed incurred the delay during delivery. The reason for this is that if a sequence of parties are corrupted, then they may not use delayed channels for communication among each other. Going back to the aforementioned example, if m arrives at tick-round 2 at $\mathcal{P}_3$ and is claimed to have been sent at tick round $t = 0$ by $\mathcal{P}_1$, then this is not what must have happened as we first must consider the corruption threshold. If both $\mathcal{P}_1, \mathcal{P}_2$ are corrupted then an adversary could have only gotten m at tick round 1, signed

$(m, 0)$ using both signing keys and make $\mathcal{P}_2$ send it to $\mathcal{P}_3$. Hence, if we consider a corruption model where 2 parties out of 3 can be corrupted, the overall channel built by $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ cannot guarantee a minimum delay that is longer than 1, if by minimum delay we mean time spent for m to travel as observed by honest parties. This is of course different if only 1 out of $\mathcal{P}_1, \mathcal{P}_2$ can be corrupted.

We now describe how the proven minimal delivery time can be computed. If both $\mathcal{P}_1$ & $\mathcal{P}_n$ are honest, then $\mathcal{P}_n$ would only sign if isP is true when the message arrives at it. This means that the message must have incurred a delay from $\mathcal{P}_1$ to $\mathcal{P}_n$ that is at least the sum of minimal delays on each intermediate channel: $\mathcal{P}_1$ is honest and must have sent it at the right time. Therefore, the longest chain of delay observed by the honest parties in this case spans the whole message delay from $\mathcal{P}_1$ to $\mathcal{P}_n$ and is the lower-bound on provable message delay. This observation extends to any chain between the first $\mathcal{P}_i$ and last $\mathcal{P}_j$ honest party within $\mathcal{P}_1, \ldots, \mathcal{P}_n$, if either of $\mathcal{P}_1, \mathcal{P}_n$ was not honest. Therefore, to determine the minimal guaranteed delay in case of k corruptions, we only need to consider the cases *where all of* $\mathcal{P}_1, \ldots, \mathcal{P}_{i-1}$ *are dishonest* and send the message later than allowed, or *where* $\mathcal{P}_{j+1}, \ldots, \mathcal{P}_n$ *are all dishonest* and sign the messages earlier than allowed, or both. Only these can reduce proven delay time.

Next, consider the setting where honest parties appear in sequences of at least $n - k > 1$ consecutive parties in the network, i.e. there is no isolated honest party. Let $\mathcal{P}_i, \ldots, \mathcal{P}_{i+n-k-1}$ be such an honest chain of parties. Then the minimal delay cannot be reduced by placing a dishonest party within this chain. This follows because then either $\mathcal{P}_{i-1}$ or $\mathcal{P}_{i+n-k}$ become honest, and the minimal honest delay then consists of the minimal delay on $\mathcal{P}_i, \ldots, \mathcal{P}_{i+n-k-1}$ plus the extra party (as the additional delay due to f_Δ will be non-negative. Therefore, to reduce the minimal delay to a minimum, exactly $n-k$ consecutive parties must be honest.

Moreover, it is not sufficient if only $\mathcal{P}_1$ or $\mathcal{P}_n$ is dishonest, followed or preceded by honest parties. This is because an honest $\mathcal{P}_2$ by observing $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ would ensure that the message was sent early enough given the delay of the channel (similarly for an honest $\mathcal{P}_{n-1}$ and corrupt $\mathcal{P}_n$). Thus, to minimize delay, an adversary will not only corrupt the first or last party in the chain, but also the adjacent one.

Bounding the channel delays. Using Propositions 4.2, 4.3 and the aforementioned observations, we can compute the minimal and maximal delay by decomposing an isP sequence into all possible partitions of up to 3 plausible subsequences, one of which is of length $n - k$ and represents the honest parties. There are at most $\mathsf{poly}(\tau)$ many such decompositions. In Appendix B.3 we show how to find sequences that realize the shortest observable minimal delay, or the maximal delay, in time polynomial in the number of isP calls.

Putting things together. We present a detailed description of our protocol for sequential communication delays $\pi_{\mathsf{Multi-SCD}}$ in Fig. 6. The protocol realizes the delay function delays computable as outlined previously.

Theorem 4.4. *The protocol* $\pi_{\mathsf{Multi-SCD}}$ *UC-securely implements* $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ *in the* $\mathcal{G}_{\mathsf{Clock}}, \mathcal{F}_{\mathsf{Reg}}, \mathcal{F}_{\mathsf{Sig}}, \mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$*-hybrid model with security against any adversary actively corrupting up to* $k = n-1$ *parties with permissible delay function given by* delays.

The proof can be found in Appendix B.4 and follows a similar outline as the one for Theorem 4.1. The key difference is that there might be a dishonest $\mathcal{P}_S$, followed by a chain of dishonest $\mathcal{P}_2, \mathcal{P}_3, \dots$ that do not necessarily have to communicate via their $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ instances. Hence, when the first honest (simulated) party obtains an output from $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$, then the message that $\mathcal{S}$ enters into $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ has to have an earlier timestamp than the current one, based on the claim when the dishonest $\mathcal{P}_1$ originally "sent" the message.

Protocol $\pi_{\mathsf{Multi-SCD}}$

This protocol is executed by a sender $\mathcal{P}_1$, a set of intermediate parties $\mathcal{P}_2, \dots, \mathcal{P}_{n-1}$ and a receiver $\mathcal{P}_n$, as well as a set of verifiers $\mathcal{V}$ interacting with each other and with $\mathcal{G}_{\mathsf{Clock}}, \mathcal{F}_{\mathsf{Reg}}, \mathcal{F}_{\mathsf{Sig}}{}^1, \dots, \mathcal{F}_{\mathsf{Sig}}{}^n$. Each pair $\mathcal{P}_i, \mathcal{P}_{i+1}$ is connected by $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,i}}$.
In every step, the activated party sends (READ) to $\mathcal{G}_{\mathsf{Clock}}$ to obtain $(\text{READ}, \bar{t})$.

Setup: Upon first activation, each $\mathcal{P}_i$ proceeds as follows:

1. Send $(\text{KEYGEN}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{Sig}}{}^i$ where $\mathcal{P}_i$ acts as signer.
2. Upon receiving $(\text{VERIFICATION KEY}, \mathsf{sid}, \mathsf{SIG}.vk_i)$ from $\mathcal{F}_{\mathsf{Sig}}{}^i$, $\mathcal{P}_i$ sends $(\text{REGISTER}, \mathsf{sid}, \mathsf{SIG}.vk_i)$ to $\mathcal{F}_{\mathsf{Reg}}$.

Send: Upon receiving first input $(\text{SEND}, \mathsf{sid}, m)$ for $\bar{t}$, $\mathcal{P}_1$ proceeds as follows:

1. Send $(\text{SIGN}, \mathsf{sid}, (m, \bar{t}))$ to $\mathcal{F}_{\mathsf{Sig}}{}^1$, receiving $(\text{SIGNATURE}, \mathsf{sid}, (m, \bar{t}), \sigma_1)$.
2. Send $(\text{SEND}, \mathsf{sid}, (m, \bar{t}, \sigma_1))$ to $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,1}}$.

Receive: Upon receiving $(\text{REC}, \mathsf{sid})$, $\mathcal{P}_n$ sends $(\text{REC}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,n-1}}$ and proceeds as follows for the first $(\text{SENT}, \mathsf{sid}, (m, t, \sigma_1, \dots, \sigma_{n-1}), t')$ received from $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,n-1}}$:

1. Check if $\mathsf{isP}(t, \mathsf{f}_{\Delta,1}, \dots, \mathsf{f}_{\Delta,n-2}, t')$.
2. For each $i \in [n-1]$ check if $\mathsf{verifySigs}(i, (m, t, \sigma_1, \dots, \sigma_{i-1}), \sigma_i, t)$ is true.
3. If all checks pass, send $(\text{SIGN}, \mathsf{sid}, (m, t, \sigma_1, \dots, \sigma_{n-1}))$ to $\mathcal{F}_{\mathsf{Sig}}{}^n$ to obtain $(\text{SIGNATURE}, \mathsf{sid}, (m, t, \sigma_1, \dots, \sigma_{n-1}), \sigma_n)$. Output $(\text{SENT}, \mathsf{sid}, m, t, \bar{t} - t, (\sigma_1, \dots, \sigma_n))$. If a check fails, then output $(\text{NOPROOF}, \mathsf{sid})$.

Verify: Upon receiving $(\text{VERIFY}, \mathsf{sid}, m, t, \Delta, \pi_{\mathsf{1o}})$, $\mathcal{V}_i \in \mathcal{V}$ parses $\pi_{\mathsf{1o}} = (\sigma_1, \dots, \sigma_n)$ and proceeds as follows:

1. Check that $t + \Delta \geq \bar{t}$ and $\mathsf{isP}(t, \mathsf{f}_{\Delta,1}, \dots, \mathsf{f}_{\Delta,n}, t + \Delta)$ is true.
2. For each $i \in [n]$ check if $\mathsf{verifySigs}(i, (m, t, \sigma_1, \dots, \sigma_{i-1}), \sigma_i, t)$ is true.
3. If all checks pass set $b = 1$, else $b = 0$. Output $(\text{VERIFIED}, \mathsf{sid}, m, t, \Delta, \pi_{\mathsf{1o}}, b)$.

Tick: Proceed as follows and then send (UPDATE) to $\mathcal{G}_{\mathsf{Clock}}$.

1. Each $\mathcal{P}_i \in \{\mathcal{P}_2, \dots, \mathcal{P}_{n-1}\}$ sends $(\text{REC}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,i-1}}$.
2. If $\mathcal{P}_i$ obtains $(\text{REC}, \mathsf{sid}, (m, t, \sigma_1, \dots, \sigma_{i-1}), t_{i-1})$ then check if $\mathsf{isP}(t, \mathsf{f}_{\Delta,1}, \dots, \mathsf{f}_{\Delta,i-2}, t_{i-1})$ is true and if for each $j \in [i-1]$ it holds that $\mathsf{verifySigs}(j, (m, t, \sigma_1, \dots, \sigma_{j-1}), \sigma_j, t)$ is true.
3. If the checks pass, send $(\text{SIGN}, \mathsf{sid}, (m, t, \sigma_1, \dots, \sigma_{i-1}))$ to $\mathcal{F}_{\mathsf{Sig}}{}^i$ to obtain $(\text{SIGNATURE}, \mathsf{sid}, (m, t, \sigma_1, \dots, \sigma_{i-1}), \sigma_i)$ if this is the first message for t.
4. Send $(\text{SEND}, \mathsf{sid}, (m, t, \sigma_1, \dots, \sigma_i))$ to $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,i}}$.

Function $\mathsf{verifySigs}(\ell, m, \sigma, t)$**:**

1. Send $(\text{RETRIEVE}, \mathsf{sid}, \mathcal{P}_\ell)$ to $\mathcal{F}_{\mathsf{Reg}}$, receiving $(\text{RETRIEVE}, \mathsf{sid}, \mathcal{P}_\ell, \mathsf{SIG}.vk, t_{Reg})$ as answer. Check that $t_{Reg} \leq t$ and output false if not.
2. Send $(\text{VERIFY}, \mathsf{sid}, m, \sigma, \mathsf{SIG}.vk)$ to $\mathcal{F}_{\mathsf{Sig}}{}^\ell$, receiving $(\text{VERIFIED}, \mathsf{sid}, m, \sigma, f)$ as response. Output true if $f = 1$, otherwise false.

Fig. 6: Protocol $\pi_{\mathsf{Multi-SCD}}$ realizing $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ with multiple intermediate parties.

Optimizing $\pi_{\mathsf{Multi-SCD}}$ While $\pi_{\mathsf{Multi-SCD}}$ realizes $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ using only simple primitives, it incurs a large overhead for the proof of sequential communication: one proof consists of n nested signatures, and each party $\mathcal{P}_i$ forwards i signatures to party $\mathcal{P}_{i+1}$. We want to obtain a proof size and communication complexity independent from the number of parties, preferably close to the size of a single signature. To do so, we face a main hurdle: it seems that we cannot eliminate a signature by any intermediate party, since that would allow the adversary to forge proofs by making the eliminated party be the honest party in $\pi_{\mathsf{Multi-SCD}}$. Hence, we focus on techniques that allow us to aggregate signatures by each party $\mathcal{P}_i$ involved in $\pi_{\mathsf{Multi-SCD}}$ in such a way that we obtain a compact proof of size independent from n. A conceptually simple way to achieve this is using a sequentially aggregate signature scheme [53] (SAS), which allows for aggregating a number of signatures generated in sequence into a single signature (*i.e.* with the same size as a single signature). However, the SAS notion is too strong, since it allows the messages of each signer to be different. A better approach is using the notion of Ordered Multisignature (OMS) schemes [14]. An OMS scheme allows a sequence of signers to compute a compact representation of signatures on a single message under each of their signing keys. Later on, a verifier can check both that the OMS is valid w.r.t. the message and public keys, and that it was generated in a certain order. This directly fits our use of signatures in $\pi_{\mathsf{Multi-SCD}}$, where enforcing the order of signing is solved by the OMS property of allowing verifiers to check the order with which each party generated its signature on (m, t). In the next section we construct an efficient OMS under weak assumptions and show how to use it to improve $\pi_{\mathsf{Multi-SCD}}$.

5 Ordered Multi-Signatures

In this section, we provide the syntax for Ordered Multi-Signatures (OMS), a game-based security notion for OMS and a construction secure under OMDL. Appendix F.3 contains the corresponding formulations in the UC model and a proof that any scheme secure according to our game-based OMS security notion also UC-realizes $\mathcal{F}_{\mathsf{OMS}}$ (Fig. 29, Theorem F.4).

Syntax and Security Model. For the syntax, our starting point are the (unordered) multi-signatures MuSig2 [58] and MuSig-L [19], where we shed algorithms related to key aggregation but keep the two-phase signing (through the algorithms SignOff and SignOn). This allows for offloading most of the overhead to the offline phase (SignOff), which can be preprocessed before the messages are known. Unlike the usual multi-signatures, SignOn now takes an *aggregate so-far* as input and updates it with contribution by the current signer. For the the security model, we adapt the security game of [14] so as to capture *interactive* and *offline-online* signing as required by our syntax. Notably, in OMS signatures not only need to be unforgeable in the standard sense but must also enforce the ordering on the signers. In [14], this last requirement is considered for the worst case where all but one co-signer are corrupted. In our application, however, this setting is too limited. Thus our security model is more complex and takes into account forgeries that involve *at least one, but potentially more* honest co-signers, a more realistic and general setting than the one in [14].

Definition 5.1 (Ordered Multi-Signature Scheme (OMS)). *An ordered multi-signature* OMS *consists of a tuple of algorithms* $\mathsf{OMS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{SignOff}, \mathsf{SignOn}, \mathsf{Vrfy})$ *with the following input-output behavior. To formally handle the interactive signing,* $\mathsf{SignOff}$ *and* SignOn *share a common state* $\mathtt{st}$*.*

$\mathsf{Setup}(1^\tau)$**:** *on input the security parameter* τ*, this algorithm outputs a handle of public parameters* pp*. Throughout, we assume that* pp *is given as implicit input to all other algorithms.*

$\mathsf{KeyGen}()$**:** *on input the public parameters, the key generation algorithm outputs a key pair* $(\mathsf{pk}, \mathsf{sk})$*.*

$\mathsf{SignOff}_{\mathtt{st}}(\mathsf{sk}, \mathsf{aux})$**:** *on input a secret key* sk *and a string of auxiliary information* aux*, the offline signing algorithm outputs an offline token* off*. This is a stateful algorithm, and updates* $\mathtt{st}$ *at every execution. According to the construction,* aux *may be an ordered list of public keys* L*, or an empty string. We note that this algorithm is agnostic of the message* m *to be signed, and runs independently of* m*.*

$\mathsf{SignOn}_{\mathtt{st}}(\mathsf{sk}, m, L, \mathsf{offs}, \sigma)$**:** *on input a secret key* sk*, a message* m*, an ordered list of public keys* L*, some offline-token information* offs*, and a so-far-aggregated online signature* σ*, the online signing algorithm outputs a new so-far-aggregated signature* σ'*, which may contain a special symbol:* $\sigma' = \varepsilon$ *if the inputs are not well-formed, or* $\sigma' = \perp^*$ *if the input* σ *does not verify w.r.t.* L *and* m*. If* $\sigma' \notin \{\perp^*, \varepsilon\}$*, it is assumed to be a valid so-far aggregate.*

$\mathsf{Vrfy}(L, m, \sigma)$**:** *on input an ordered list of public keys* L*, a message* m*, and a signature* σ*, the verification algorithm outputs* 1 *(accept) or* 0 *(reject).*

In the remainder of the paper, we assume the order of the signing parties is known a-priori and given in input to $\mathsf{SignOff}$ as $\mathsf{aux} = L = (\mathsf{pk}_1, \dots, \mathsf{pk}_n)$. This is in line with our specific application scenario, since in $\pi_{\mathsf{Multi-SCD}}$ all participants must agree on complete signing order (route) before launching the OMS signing protocol. Moreover, it allows us to label the offline tokens and the so-far-aggregated online signatures according to the position of the signer in the chain L. For instance, for the signer in position i of $L = (\mathsf{pk}_1, \dots, \mathsf{pk}_n)$ we get: $\mathsf{off}_i \leftarrow \mathsf{SignOff}_{\mathtt{st}}(\mathsf{sk}, \mathsf{aux} = L)$ and $\sigma_i \leftarrow \mathsf{SignOn}_{\mathtt{st}}(\mathsf{sk}_i, m, L, \mathsf{offs} = (\mathsf{off}_1, \dots, \mathsf{off}_n), \sigma_{i-1})$.

We remark that $\mathsf{SignOff}$ and SignOn share the state information of signer i. While $\mathsf{SignOff}$ can run without a specified signer order, e.g., in cases where $\mathsf{aux} = \varepsilon$, honest executions of OMS demand SignOn be run in a sequential manner. That is to say, SignOn takes in input offline tokens offs (usually one for each co-signer in L) and the online output σ_{i-1} of co-signer $i-1$, and passes their output σ_i on to the co-signer in position $i+1$. The signer in position $i = 1$ receives σ_0 as initial input, where σ_0 is some constant specified in pp, and the signer in position $i = n$ outputs $\sigma_n = \sigma$ as the final OMS. Finally we highlight that our syntax is general and accommodates for several settings, including: *identifying the order of signing parties on-the-fly*, so that each signer only learns the partial order $\mathsf{aux} = L_{i-1}$ in the offline phase, and then checks that the full list L received in the online phase has L_{i-1} as prefix, similar to [14]; *unsynchronized offline-phase*, when $\mathsf{aux} = \varepsilon$; *known co-signer order*, $\mathsf{aux} = L$ both for out-and-back topologies and round-robin. We discuss some of variants in more detail in Appendix E.3.

Correctness demands that Vrfy accepts as long as all signers receive the same message m and L and execute KeyGen, $\mathsf{SignOff}$, and SignOn honestly. The definition is straightforward and deferred to Appendix E.1.

Security Model. Our security model for OMS aims at capturing the two following conditions. The signature shall be *unforgeable*, that is, as long as at least one signer in L is honest (i.e., the adversary has no access to the signer's secret key) it should be computationally infeasible to generate a signature σ^* that verifies for a *new* message (not queried during the game) or against a *new* list of co-signers L^*. Moreover *order matters*, so an adversary $\mathcal{A}$ that has access to a subset C of the keys identified by an ordered signer list L should be unable to generate a signature σ^* that verifies for a list $L^* \neq L$ where the public keys differ, or the position of some honest signers is changed. That is to say, we expect to detect all order changes except for shuffling of dishonest parties. This latter property separates OMS from MS: in the OMS security game, $\mathcal{A}$ may win by outputting a valid signature on $(m, L^* = (\mathsf{pk}, \mathsf{pk}'))$ after querying $(m, L = (\mathsf{pk}', \mathsf{pk}))$ to the signing oracle; whereas in the usual MS security game L^* is not considered new, and thus the adversary cannot win with such an output.

Following [14], we require the key pairs to be *registered*, that is, $\mathcal{A}$ must inform the knowledge of its secret keys in advance, moreover any query or forgery attempt that involves at least one unregistered key in L gets rejected. This requirement essentially models the situation where users are asked to prove knowledge of their secret keys during public-key registration with a CA. Such registered key (RK) model is weaker than the typical plain public key (PPK) model, where $\mathcal{A}$ may use arbitrary public keys for which it may not know the corresponding secret keys. In our case, this is not a drawback since we will use OMS in the protocol $\pi_{\mathsf{Multi-SCD}}$ that already employs a key registration functionality $\mathcal{F}_{\mathsf{Reg}}$ and it can easily be extended to $\mathcal{F}_{\mathsf{vReg}}$ performing an additional well-formedness check. In Appendix E.3 we also sketch how to strengthen this basic construction to meet security under the PPK model using existing tricks of MuSig2.

Definition 5.2 (OMS Security). *An ordered multi signature* OMS *is said to be secure if for any probabilistic polynomial time adversary* $\mathcal{A}$ *and* $\forall \tau$ *it holds that:* $\mathbf{Adv}_{\mathsf{OMS}}^{\mathsf{OMS\text{-}UF\text{-}CMA}}(\mathcal{A}, \tau) := \Pr[1 \leftarrow \mathsf{OMS\text{-}UF\text{-}CMA}(\mathcal{A}, \tau)] \leq \mathsf{negl}(\tau)$, *where* $\mathsf{OMS\text{-}UF\text{-}CMA}$ *is the security-game for unforgeability under chosen-message attack of ordered multi-signatures defined in Figure 7.*

Remark 5.3. While we take inspiration from the OMS security model of [14], we also identify a weakness that motivates our new model. In [14] security holds for one single honest signer in L. While this setting may seem stronger than ours, it does not capture more complex scenarios where the adversary tries to swap the signing order of honest parties, which is the goal of this work. In particular, in [14] a signature output by an honest signer authenticates the message m and the signer's position i in the chain, but not the order of honest signers that contribute before and after position i. In more detail, [14] considers a forgery to be successful if (among other trivial checks) $\mathcal{A}$ never queried the signing oracle with (m^*, σ', L') for the same m^* as in the forgery, $|L'| = i^* - 1$ and i^* is the

Security Model for OMS

GAME OMS-UF-CMA($\mathcal{A}, \tau$)

1: $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\tau)$
2: $\mathcal{K} := \emptyset$; $\mathcal{T} := \emptyset$; $\mathcal{Q} := \emptyset$; $\mathcal{S} := \emptyset$;
3: $\mathsf{win} := 0$
4: $\mathcal{O} := \{\mathsf{OKeyReg}, \mathsf{OSignOff}, \mathsf{OSignOn}\}$ ▷ RO in $\mathcal{O}$ if needed
5: $(L^*, m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{pp})$
6: **if** $\mathsf{win} = 1$ **then** [**return** 1] ▷ win is updated by OSignOn
7: **parse** $L^* = (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$
8: **if** $\exists \mathsf{pk} \in L^*$ s.t. $(\cdot, \mathsf{pk}, \cdot) \notin \mathcal{K}$ **then return** 0 ▷ Unregistered keys
9: **for** $i, j \in [1, |L^*|]$ **do**
10: **if** $\mathsf{pk}_i = \mathsf{pk}_j$ and $i \neq j$ **then return** 0 ▷ Same signer
11: $H := \varepsilon$ ▷ Honest signer key list
12: **for** $i = 1, \ldots, n$ **do**
13: **if** $(1, \mathsf{pk}_i, \cdot) \in \mathcal{K}$ **then** $H := H|\mathsf{pk}_i$
14: **if** $H = \varepsilon$ **then return** 0 ▷ L^* must contain one honest signer
15: **if** $\exists \mathsf{pk} \in H : (\mathsf{pk}, m^*, L^*) \notin \mathcal{Q}[\cdot]$ **then** [**return** $\mathsf{Vrfy}(L^*, m^*, \sigma^*)$] ▷ Standard unforgeability
16: **else return** 0

ORACLES

$\mathsf{OKeyReg}(uid, \mathsf{aux})$

1: **if** $\mathcal{K}[uid] \neq \epsilon$ **then return** uid `already registered`
2: **if** $\mathsf{aux} = \epsilon$ **then**
3: $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$
4: $\mathcal{K}[uid] := (1, \mathsf{pk}, \mathsf{sk})$ ▷ honest
5: **else** use aux to run $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$
6: $\mathcal{K}[uid] := (0, \mathsf{pk}, \mathsf{sk})$ ▷ corrupt
7: **return** $(uid$, `pk) registered`

$\mathsf{OSignOff}(\mathsf{sid}, L, i)$

1: **if** $(1, \mathsf{pk}_i, \cdot) \notin \mathcal{K}$ **then return** `illicit signer`
2: **if** $(\mathsf{pk}_i, \cdot) \in \mathcal{T}[\mathsf{sid}]$ **then return** `session already initialised`
3: **if** $\exists \mathsf{pk} \in L : (*, \mathsf{pk}, *) \notin \mathcal{K}$ **then return** `unregistered key`
4: $\mathsf{st}_i \leftarrow \varepsilon$
5: $\mathsf{off}_i \leftarrow \mathsf{SignOff}_{\mathsf{st}_i}(\mathsf{sk}_i, L)$ ▷ state update
6: $\mathcal{T}[\mathsf{sid}] := \mathcal{T}[\mathsf{sid}]|(\mathsf{pk}_i, \mathsf{st}_i)$
7: **return** off_i

$\mathsf{OSignOn}(\mathsf{sid}, m, L, \mathsf{offs}, \sigma_{i-1}, i)$

1: **if** $(1, \mathsf{pk}_i, \cdot) \notin \mathcal{K}$ **then return** `illicit signer`
2: **if** $(\mathsf{pk}_i, \cdot) \notin \mathcal{T}[\mathsf{sid}]$ **then return** `offline session not initialised`
3: **if** $\exists \mathsf{pk} \in L : (*, \mathsf{pk}, *) \notin \mathcal{K}$ **then return** `unregistered key`
4: **if** $(\mathsf{pk}_i, \cdot) \in \mathcal{Q}[\mathsf{sid}]$ **then return** `used session`
5: **retrieve** $(\mathsf{pk}_i, \mathsf{st}_i)$ from $\mathcal{T}[\mathsf{sid}]$
6: **retrieve** $(1, \mathsf{pk}_i, \mathsf{sk}_i)$ from $\mathcal{K}$
7: $\sigma_i \leftarrow \mathsf{SignOn}_{\mathsf{st}_i}(\mathsf{sk}_i, m, L, \mathsf{offs}, \sigma_{i-1})$ ▷ Lines 8-13: check winning conditions
8: $H := \varepsilon$ ▷ Honest signer key list
9: **for** $j = 1, \ldots, i-1$ **do**
10: **if** $(1, \mathsf{pk}_j, \cdot) \in \mathcal{K}$ **then**
11: $H := H|\mathsf{pk}_j$
12: **if** $\sigma_i \notin \{\varepsilon, \perp^*\} \wedge \exists \mathsf{pk} \in H : (\mathsf{pk}, m, L) \notin \mathcal{Q}[\cdot]$ **then**
13: [$\mathsf{win} := 1$] ▷ order forgery: party i signs but honest party $j < i$ has not signed yet
14: $\mathcal{Q}[\mathsf{sid}] := \mathcal{Q}[\mathsf{sid}]|(\mathsf{pk}_i, m, L)$
15: **return** σ_i

Fig. 7: Security definition for ordered multi-signatures: unforgeability under chosen message attack. L is always assumed to be parsed as $L = (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$. We [frame] the branches in the security experiment where $\mathcal{A}$ may win.

position of the honest signer in L^*, and $\sigma' \in \{0,1\}^*$ is arbitrary. This constraint is needed as in [14] extensions of an ordered multi signature to a longer set of signers $L' = L|\mathsf{pk}'$ are trivial, which is not the case in our construction.

Ordered Multi-Signature from the One-More Discrete Log Assumption

$\mathsf{Setup}(1^\tau)$:

1: $(\mathbb{G}, p, g) \leftarrow \mathsf{GroupGen}(1^\tau)$
2: $(n, \ell) \leftarrow \mathsf{poly}(\tau)$
3: $\mathsf{H}_{\text{non}}, \mathsf{H}_{\text{ch}} : \{0,1\}^* \to \mathbb{Z}_p^*$
4: $\sigma_0 := (1_{\mathbb{G}}, 0) \in \mathbb{G} \times \mathbb{Z}_p$
5: **return** $\mathsf{pp} := (\mathbb{G}, g, p, n, \ell, \mathsf{H}, \sigma_0)$

$\mathsf{KeyGen}()$:

1: $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_p$; $\mathsf{pk} := g^{\mathsf{sk}}$
2: **return** $(\mathsf{pk}, \mathsf{sk})$

$\mathsf{Vrfy}(L, m, \sigma)$:

1: **Parse** $L = (\mathsf{pk}_1, \ldots, \mathsf{pk}_n) \in \mathbb{G}^n$
2: **for** $i, j \in [1, n]$ **do**
3: **if** $\mathsf{pk}_i = \mathsf{pk}_j \wedge i \neq j$ **then return** 0
4: **Parse** $\sigma = (R, z) \in \mathbb{G} \times \mathbb{Z}_p$
5: $c \leftarrow \mathsf{H}_{\text{ch}}(m, R, L)$
6: **if** $g^z = R \cdot (\prod_{i=1}^n \mathsf{pk}_i)^c$ **then return** 1
7: **return** 0

Two-phase interactive singing run in a loop determined by $L = (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$. Note that the first part of SignOn (lines 1-8) can be preprocessed given all offline tokens from the co-signers and independently of the message.

For $i \in [1, n]$ **do**

$\mathsf{SignOff}_{\mathtt{st}_i}(\mathsf{sk}_i, L)$:

1: **for** $j \in [1, \ell]$ **do**
2: $r_{i,j} \xleftarrow{\$} \mathbb{Z}_p$
3: $R_{i,j} := g^{r_{i,j}}$
4: $\mathtt{st}_i := \mathtt{st}_i | r_{i,j} | R_{i,j}$
5: $\mathsf{off}_i := (R_{i,1}, \ldots, R_{i,\ell})$
6: **return** off_i

$\mathsf{offs} := (\mathsf{off}_1, \ldots, \mathsf{off}_n)$

For $i \in [1, n]$ **do**

$\mathsf{SignOn}_{\mathtt{st}_i}(\mathsf{sk}_i, m, L, \mathsf{offs}, \sigma_{i-1})$:

▷ Lines 1-8: processing independent of m, σ_{i-1}
1: **Parse** $\mathtt{st}_i = (r_{i,j} | R'_{i,j})_{j=1}^{\ell} \in (\mathbb{Z}_p \times \mathbb{G})^{\ell}$
2: **Parse** $\mathsf{offs} = ((R_{1,j})_{j=1}^{\ell}, \ldots, (R_{n,j})_{j=1}^{\ell}) \in \mathbb{G}^{\ell \times n}$
3: **if** $R'_{i,j} \neq R_{i,j}$ **then return** ε
4: $\widetilde{\mathsf{pk}} := \prod_{k=1}^{i-1} \mathsf{pk}_k$
5: **for** $j \in [1, \ell]$ **do**
6: $R_j := \prod_{k=1}^{n} R_{k,j}$
7: $\tilde{R}_j := \prod_{k=1}^{i-1} R_{k,j}$ ▷ So-far aggregation
8: **if** $R_j = 1_{\mathbb{G}}$ or $\tilde{R}_j = 1_{\mathbb{G}}$ **then return** ε
▷ Lines 9-20: processing that depends on m, σ_{i-1}
9: **Parse** $\sigma_{i-1} = (R, \tilde{z}) \in \mathbb{G} \times \mathbb{Z}_p$
10: $v \leftarrow \mathsf{H}_{\text{non}}(m, \mathsf{offs}, L)$
11: **if** $i = 1$ **then**
12: $R := \prod_{j=1}^{\ell} R_j^{v^{j-1}}$
13: $c \leftarrow \mathsf{H}_{\text{ch}}(m, R, L)$
14: **else** ▷ Check so-far aggregation
15: **if** $R \neq \prod_{j=1}^{\ell} R_j^{v^{j-1}}$ **then return** ε
16: $\tilde{R} := \prod_{j=1}^{\ell} \tilde{R}_j^{v^{j-1}}$
17: $c \leftarrow \mathsf{H}_{\text{ch}}(m, R, L)$
18: **if** $g^{\tilde{z}} \neq \tilde{R} \cdot \widetilde{\mathsf{pk}}^c$ **then return** $\perp^*$
19: $z_i := c \cdot \mathsf{sk}_i + \sum_{j=1}^{\ell} v^{j-1} \cdot r_{i,j}$
20: $z := \tilde{z} + z_i$ **return** $\sigma_i = (R, z)$

Fig. 8: Our OMS construction. In the description, L is always assumed to be parsed as $L = (\mathsf{pk}_1, \ldots, \mathsf{pk}_n) \in \mathbb{G}^n$ for some known public parameter $n \geq 2$.

Schnorr-based OMS Construction. We present a modified version of MuSig2 in Figure 8. This construction is tailored to the application scenario considered in this work: the set of co-signers and the signing order are known a priori. This information is conveyed to $\mathsf{SignOff}$ via $\mathsf{aux} = L = (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$. Following the route given by L in a round-Robin fashon, the signer in position i is able to send

messages to the signer in position $i+1$. This setting makes the algorithms simpler and more efficient. In particular, each signer can compute its *so-far accumulated public key* $\widetilde{\mathsf{pk}} = \prod_{k=1}^{i-1} \mathsf{pk}_k$, and reuse it for later OMS signing instances, making running time in the online phase independent of n, the total number of co-signers, and i, the signer's position in L.

Following the security model defined earlier we assume that an adversary only outputs a forgery with a key list L^* consisting of pre-registered public keys. That is, the reduction knows secret keys associated with all public keys in L^* except for the challenge key pk^*. In the proof we will make use of the General Forking Lemma and the One-More Discrete Log (OMDL) Assumption available in Appendix E.1. While this proof strategy closely follows that of MuSig2, guaranteeing correct order of honest signers turns out non-trivial. In fact, it is easy to see that the original MuSig2 doesn't qualify as a secure OMS, since an adversary can easily swap the order of honest parties' contribution by querying signing oracles in different order than what's specified in L. To overcome this hurdle, our proof makes use of the so-far aggregation check (L.18). The full proof is deferred to Appendix E.2.

Theorem 5.4 (Informal, see Theorem E.4 for the formal statement). *Our* OMS *scheme with* $\ell = 2$ *is* $\mathsf{OMS\text{-}UF\text{-}CMA}$ *secure under the OMDL assumption, in the programmable random oracle model.*

Integrating the OMS into $\pi_{\mathsf{Multi-SCD}}$ Integrating our OMS construction into the protocol $\pi_{\mathsf{Multi-SCD}}$ from the previous section is rather straightforward, so we will only sketch it here. During **Setup**, each party will register its key pair with an extended key registration functionality (that also requires proof of secret key knowledge). Then, each $\mathcal{P}_i$ will run $\mathsf{SignOff}$ for a nonce $\mathsf{sid} = 0$. Upon completion, during **Send** $\mathcal{P}_1$ will sign m, t using SignOn for the last initialized sid as offline state, and additionally run $\mathsf{SignOff}$ for $\mathsf{sid}+1$. Thereafter, during **Tick** each party will do the same. Finally, during **Receive** $\mathcal{P}_n$ will output its $\mathsf{SignOff}$ output for $\mathsf{sid} + 1$ to $\mathcal{P}_1$, and will output the proof for m, t if it verifies. The modified protocol will only check that all keys were registered at the correct time during VERIFYSIGS, as the aggregation check is already done during SignOn. In Appendix F.4 we give a more detailed construction using the UC OMS functionality and prove it secure. We want to stress that $\mathsf{SignOff}$ can run during idle time and must not be parallelized with proof-generation instances.

6 Verifiable Delay Functions

We construct a VDF from proofs of sequential communication delays. Our construction can be obtained in a black-box manner from any proof of sequential delay, yielding a VDF with a proof size equal to that of the underlying proof of communication delay. The main idea is to sequentially send the input of the VDF among nodes in a network while having them compute a proof of sequential communication delay for this message. The output is computed by querying a global random oracle on the input concatenated with the proof of sequential communication delay. Verification can be easily achieved by first verifying the proof of sequential communication delay and then recomputing the output. We realize a VDF functionality presented in Figure 9, which is adapted from [8].

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

$\mathcal{F}_{\mathsf{VDF}}$ is parameterized by a computational security parameter τ, and input space $\mathcal{ST}$, a proof space $\mathcal{PROOF}$, a slack parameter $0 < \epsilon \leq 1$ and a delay parameter Γ. $\mathcal{F}_{\mathsf{VDF}}$ interacts with a set of parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, and an adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{VDF}}$ maintains a initially empty lists L (proofs being computed), and OUT (outputs).

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

Tick: For each $(\mathcal{P}_i, \mathsf{sid}, in, c, b) \in L$, update $(\mathcal{P}_i, \mathsf{sid}, in, c, b) \in L$ by setting $c = c+1$ and proceed as follows:

1. If $c \geq \epsilon\Gamma$ sample $out \xleftarrow{\$} \mathcal{ST}$, send $(\mathsf{GetStsPf}, \mathsf{sid}, in, out)$ to $\mathcal{S}$ and wait for an answer. If $\mathcal{S}$ answers with $(\textsc{Abort}, \mathsf{sid})$, update $(\mathcal{P}_i, \mathsf{sid}, in, c, b) \in L$ by setting $b = \bot$. If $\mathcal{S}$ answers with $(\mathsf{GetStsPf}, \mathsf{sid}, \pi)$, $\mathcal{F}_{\mathsf{VDF}}$ halts if $\pi \notin \mathcal{PROOF}$ or there exists $(in', out', \pi) \in \mathsf{OUT}$, else, it appends (in, out, π) to OUT.
2. If $c = \Gamma$, remove $(\mathcal{P}_i, \mathsf{sid}, in, \Gamma, b) \in L$. If there was an abort (*i.e.* $b = \bot$), send $(\mathsf{NoProof}, \mathsf{sid}, in)$ to $\mathcal{P}_i$. Otherwise, send $(\mathsf{Proof}, \mathsf{sid}, in, out, \pi)$ to $\mathcal{P}_i$.

Verification: Upon receiving $(\mathsf{Verify}, \mathsf{sid}, in, out, \pi)$ from $\mathcal{P}_i \in \mathcal{P}$, set $b = 1$ if $(in, out, \pi) \in \mathsf{OUT}$, otherwise set $b = 0$ and output $(\mathsf{Verified}, \mathsf{sid}, in, out, \pi, b)$ to $\mathcal{P}_i$.

Fig. 9: Ticked Functionality $\mathcal{F}_{\mathsf{VDF}}$ for Verifiable Delay Functions.

We present our VDF protocol in Figure 10. The construction assumes access to a bulletin board where we store attempts at jointly evaluating the VDF by sending a message via $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$. When evaluating the VDF we consider as valid only the first evaluation attempt registered in the bulletin board with a valid proof of sequential delay generated by $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$. This significantly simplifies our analysis since the adversary can no longer send the same input to $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ multiple times and obtain multiple proofs of sequential delay and thus produce several valid VDF outputs, which deviates from the standard behavior expected from this primitive. The same effect could be obtained by assuming either $\mathcal{P}_S$ or $\mathcal{P}_R$ are honest and do not accept to interact with $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ to transmit the same message more than once, thus guaranteeing only one proof of sequential delay is generated, which means a single valid VDF output exists.

Theorem 6.1. *Protocol π_{VDF} UC-realizes $\mathcal{F}_{\mathsf{VDF}}$ in the $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}, \mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{BB}}$-hybrid model against an active static adversary corrupting a majority of parties in $\mathcal{P}$. The delay parameter is $\Gamma = \Delta_{\mathtt{hi}}$ and the slack parameter is $\epsilon = \frac{\Delta_{\mathtt{lo}}}{\Delta_{\mathtt{hi}}}$ where $(\cdot, \Delta_{\mathtt{hi}}) = \max_{t \in \{0,\ldots,\mathsf{poly}(\tau)\}} \{\mathsf{f}_\Delta(t)\}$ and $(\Delta_{\mathtt{lo}}, \cdot) = \min_{t \in \{0,\ldots,\mathsf{poly}(\tau)\}} \{\mathsf{f}_\Delta(t)\}$.*

Proof. It is simple to construct a simulator $\mathcal{S}$ for π_{VDF} by having $\mathcal{S}$ interact with an internal copy of $\mathcal{A}$ towards which it simulates honest parties executing exactly as in π_{VDF} and simulating $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}, \mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{BB}}$ exactly as they are described except when explicitly stated. $\mathcal{S}$ forwards every message sent to simulated $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ to be evaluated by $\mathcal{F}_{\mathsf{VDF}}$ and provides matching proofs to $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ and $\mathcal{F}_{\mathsf{VDF}}$ when requested. If $\mathcal{A}$ causes an evaluation to abort, $\mathcal{S}$ correspondingly aborts the same evaluation at $\mathcal{F}_{\mathsf{VDF}}$. Whenever $\mathcal{F}_{\mathsf{VDF}}$ leaks to $\mathcal{S}$ that an evaluation on a new input has been requested, $\mathcal{S}$ simulates this evaluation in the simulation.

Protocol π_{VDF}

Protocol π_{VDF} is executed by a set of parties $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$ interacting with a bulletin board functionality $\mathcal{F}_{\mathsf{BB}}$ and with $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$, where party $\mathcal{P}_R \in \mathcal{P}$ acts as receiver and party $\mathcal{P}_S \in \mathcal{P}$ as sender. They additionally use a random oracle $\mathcal{G}_{\mathsf{rpoRO}}$.

Solve: A party $\mathcal{P}_i$ interacts with $\mathcal{P}_S, \mathcal{P}_R$ as follows to evaluate the VDF on in:

1. On input $(\mathsf{Solve}, \mathsf{sid}, in)$, $\mathcal{P}_i$ sends $(\textsc{Read}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{BB}}$ and checks whether a record $(c, in, t, \Delta, \pi_{1\circ})$ is returned (if multiple $(c, in, t, \Delta, \pi_{1\circ})$ for different c and $\pi_{1\circ}$ are returned, consider the one with the lowest c and a valid $\pi_{1\circ}$ w.r.t $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$). If yes, skip to step 5.
2. $\mathcal{P}_i$ sends $(\textsc{Send}, \mathsf{sid}, in)$ to $\mathcal{P}_S$ and $\mathcal{P}_S$ forwards $(\textsc{Send}, \mathsf{sid}, in)$ from $\mathcal{P}_i$ to $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$.
3. Upon receiving $(\textsc{Sent}, \mathsf{sid}, in, t, \Delta, \pi_{1\circ})$ from $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$, $\mathcal{P}_R$ send $(\textsc{Write}, \mathsf{sid}, (in, t, \Delta, \pi_{1\circ}))$ to $\mathcal{F}_{\mathsf{BB}}$. If instead $\mathcal{P}_R$ receives $(\textsc{NoProof}, \mathsf{sid})$, it forwards this message to all parties in $\mathcal{P}$.
4. If it received $(\textsc{NoProof}, \mathsf{sid})$ from $\mathcal{P}_R$, $\mathcal{P}_i$ outputs $(\textsc{NoProof}, \mathsf{sid}, in)$. Otherwise, it sends $(\textsc{Read}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{BB}}$ and retrieves $(c, in, t, \Delta, \pi_{1\circ})$.
5. $\mathcal{P}_i$ sends $(\textsc{Hash-Query}, in|\pi_{1\circ})$ to $\mathcal{G}_{\mathsf{rpoRO}}$, receiving $(\textsc{Hash-Confirm}, out)$. $\mathcal{P}_i$ sends $(\textsc{IsProgrammed}, in|\pi_{1\circ})$ and aborts if the response is $(\textsc{IsProgrammed}, 1)$. $\mathcal{P}_i$ outputs $(\textsc{Proof}, \mathsf{sid}, in, out, \pi = \pi_{1\circ})$.

Verification: On input$(\textsc{Verify}, \mathsf{sid}, in, out, \pi)$, $\mathcal{P}_i$ proceeds as follows:

1. Send $(\textsc{Read}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{BB}}$ and check that there is a record (c, in, t, Δ, π), if multiple (c, in, t, Δ, π) for different c are returned, consider the one with the lowest c and a valid π w.r.t. to $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$.
2. Send $(\textsc{Verify}, \mathsf{sid}, in, t, \Delta, \pi)$ to $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ expecting $(\textsc{Verified}, \mathsf{sid}, in, t, \Delta, 1)$.
3. Send $(\textsc{Hash-Query}, in|\pi)$ to $\mathcal{G}_{\mathsf{rpoRO}}$, receiving $(\textsc{Hash-Confirm}, out')$. Check that $out = out'$. Send $(\textsc{IsProgrammed}, in|\pi)$ expecting $(\textsc{IsProgrammed}, 0)$.
4. If all checks pass set $b = 1$, else set $b = 0$, and output $(\textsc{Verified}, \mathsf{sid}, in, out, \pi, b)$

Fig. 10: Protocol π_{VDF}.

Moreover, $\mathcal{S}$ programs $\mathcal{G}_{\mathsf{rpoRO}}$ so that outputs of simulated VDF evaluations match the outputs provided by $\mathcal{F}_{\mathsf{VDF}}$. □

7 Publicly Verifiable Time-Lock Puzzles

We construct a publicly verifiable time-lock puzzle (PV-TLP) based on sequential communication delays. The main idea is to use a threshold encryption scheme and generate a puzzle by encrypting a message under the public key. The secret key is in turn shared among a set of nodes connected by delayed channels. The TLP is opened by having these nodes perform threshold decryption via sequential communication. By having the nodes which hold the key shares communicate in a round-robin manner, the individual channel delays then add up to the overall delay of the TLP.

In our construction, the sizes of both the proof and the messages exchanged among each pair of parties involved in solving the puzzle are independent from the number of parties. In order to do so, we relax our output guarantee by only detecting dishonest behavior after the decryption protocol is finished without

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

$\mathcal{F}_{\mathsf{DKG}}$ is parameterized by a cyclic group $\mathbb{G}$ of order q with generator g and interacts with a set of parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, among which a subset of solvers $\mathcal{W} \subset \mathcal{P}$.

Key Generation The first time it is activated, $\mathcal{F}_{\mathsf{DKG}}$ samples $\mathsf{sk}_i \xleftarrow{\$} \mathbb{G}$ for $i \in \mathcal{W}$, computes $\mathsf{sk} = \sum_{i \in \mathcal{W}} \mathsf{sk}_i$ and $\mathsf{pk} = g^{\mathsf{sk}}$.
SK Request: Upon $(\textsc{SecKey}, \mathsf{sid})$ from $\mathcal{P}_i \in \mathcal{W}$, return $(\textsc{SecKey}, \mathsf{sid}, \mathsf{sk}_i)$.
PK Request Upon $(\textsc{PubKey}, \mathsf{sid})$ from $\mathcal{P}_i \in \mathcal{P}$, return $(\textsc{PubKey}, \mathsf{sid}, \mathsf{pk})$.

Fig. 11: Functionality $\mathcal{F}_{\mathsf{DKG}}$ for distributed key generation.

identifying cheaters, which allows for the adversary to cause aborts without revealing the corrupted parties. In case aborts happen, we can fall back to a more expensive protocol using NIZKs of valid decryption share generation in order to identify the corrupted parties and eliminate them. This yields low overhead in the optimistic case (which is the most likely to happen in practice) while still attaining guaranteed output delivery.

In order to achieve constant communication, we have each decryption node aggregate its decryption share to the share received from the previous party along with a proof of sequential communication showing that the ciphertext being decrypted has traversed a pre-defined path through a certain sequence of decryption nodes. This step avoids attacks where the adversary obtains several decryption shares from different honest nodes in parallel or out of order.

We use the generic Public Key Cryptosystem with Plaintext Verification construction from Definition A.4 together with a simple threshold version of El Gamal to verify that the final decrypted message is indeed the message that was originally encrypted (i.e. the message inside the PV-TLP). Hence, the verifier only has to perform a re-encryption check in order to assert that a given PV-TLP has been correctly solved. This optimized construction realizes the PV-TLP functionality defined in Fig. 12, which follows [8] but supports only a fixed delay Γ. Our construction, $\pi_{\mathsf{TLP-Light}}$, is depicted in Fig. 13 and employs a Distributed Key Generation functionality, $\mathcal{F}_{\mathsf{DKG}}$, in the setup (Fig. 11). The $\mathcal{F}_{\mathsf{DKG}}$ functionality can be UC-realized by a number of protocols that compute a public key g^{sk} and secret key shares sk_i such that $\mathsf{sk} = \mathsf{sk}_1 + \cdots + \mathsf{sk}_n$.

We capture the security of Protocol $\pi_{\mathsf{TLP-Light}}$ in Theorem 7.1. The proof obtains loose bounds for the minimum and maximum delay guarantees provided by this protocol since $\pi_{\mathsf{TLP-Light}}$ only uses the decryption validity proof as a publicly verifiable proof of a TLP solution, which allows for a unique and easily verifiable proof. If the TLP proof instead also consisted of the proofs provided by the parties in the set $\mathcal{W}$ by using $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ instead of $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ and for correct decryption, we would be able to condition the minimum and maximum delays guaranteed by a TLP solution on the exact time when it is solved, which would give tighter delay bounds. However, the latter approach requires an intricate reworking of $\mathcal{F}_{\mathsf{tlp}}$ that would also require a more expensive protocol to realize as the communication per party becomes linear in $|\mathcal{W}|$. Hence, we present this simpler construction in order to highlight our main techniques.

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

$\mathcal{F}_{\mathsf{tlp}}$ is parameterized by a computational security parameter τ, a message space $\{0,1\}^\tau$, a tag space $\mathcal{TAG}$, a proof space $\mathcal{PROOF}$, a slack parameter $0 < \epsilon \leq 1$ and a delay parameter Γ. $\mathcal{F}_{\mathsf{tlp}}$ interacts with a set of parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and an adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{tlp}}$ maintains initially empty lists omsg (output messages and proofs) and L (puzzles being solved).

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

1. If $\mathcal{P}_i$ is honest, sample $\mathsf{puz} \xleftarrow{\$} \mathcal{TAG}$ and proof $\pi \xleftarrow{\$} \mathcal{PROOF}$.
2. If $\mathcal{P}_i$ is corrupted, let $\mathcal{S}$ provide puz and π. If $(\mathsf{puz}, \pi) \notin \mathcal{TAG} \times \mathcal{PROOF}$ or there exists $(\mathsf{puz}', m', \pi) \in \mathsf{omsg}$, then $\mathcal{F}_{\mathsf{tlp}}$ halts.
3. Append (puz, m, π) to omsg, set and output $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathsf{puz}, \pi)$ to $\mathcal{P}_i$ and $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathsf{puz})$ to $\mathcal{S}$.

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

Public Verification: Upon receiving $(\mathsf{Verify}, \mathsf{sid}, \mathsf{puz}, m, \pi)$ from a party $\mathcal{P}_i \in \mathcal{P}$, set $b = 1$ if $(\mathsf{puz}, m, \pi) \in \mathsf{omsg}$, otherwise set $b = 0$ and output $(\mathsf{Verified}, \mathsf{sid}, \mathsf{puz}, m, \pi, b)$ to $\mathcal{P}_i$.

Tick: For all $(\mathsf{sid}, \mathsf{puz}, c) \in L$, update $(\mathsf{sid}, \mathsf{puz}, c) \in L$ by setting $c = c + 1$ and proceed as follows:

- If $c \geq \epsilon\Gamma$ and $(\mathsf{puz}, m, \pi) \in \mathsf{omsg}$, output $(\mathsf{Solved}, \mathsf{sid}, \mathsf{puz}, m, \pi)$ to $\mathcal{S}$.
- If $c \geq \epsilon\Gamma$ and there does not exist $(\mathsf{puz}, m, \pi) \in \mathsf{omsg}$, let $\mathcal{S}$ provide $\pi \in \mathcal{PROOF}$ and add $(\mathsf{puz}, \perp, \pi)$ to omsg.
- If $c = \Gamma$, remove $(\mathsf{sid}, \mathsf{puz}, c) \in L$ and send $(\mathsf{Proceed?}, \mathsf{sid}, \mathsf{puz}, m, \pi)$ to $\mathcal{S}$, where m, π are such that there is $(\mathsf{puz}, m, \pi) \in \mathsf{omsg}$ and proceed as follows:
  - If $\mathcal{S}$ sends $(\textsc{Abort}, \mathsf{sid}, \pi')$, output $(\mathsf{Solved}, \mathsf{sid}, \mathsf{puz}, \perp, \pi')$ to all $\mathcal{P}_i$.
  - If $\mathcal{S}$ sends $(\textsc{Proceed}, \mathsf{sid})$, output $(\mathsf{Solved}, \mathsf{sid}, \mathsf{puz}, m, \pi)$ to all $\mathcal{P}_i$.

Fig. 12: Ticked Functionality $\mathcal{F}_{\mathsf{tlp}}$ for publicly verifiable time-lock puzzles.

Theorem 7.1. *Protocol $\pi_{\mathsf{TLP-Light}}$ UC-realizes $\mathcal{F}_{\mathsf{tlp}}$ in the $\mathcal{G}_{\mathsf{Clock}}, \mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{DKG}}, \mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$-hybrid model against an active static adversary $\mathcal{A}$ corrupting a majority of parties in $\mathcal{W}$. The parameters of $\mathcal{F}_{\mathsf{tlp}}$ are tag space $\mathcal{TAG} = \mathbb{G} \times \mathbb{G} \times \{0,1\}^{2\tau}$, proof space $\mathbb{G} \times \{0,1\}^\tau$, slack parameter $\epsilon = \frac{\Delta_{\mathtt{lo}}}{\Delta_{\mathtt{hi}}}$ and delay parameter $\Gamma = \Delta_{\mathtt{hi}}$ where $(\Delta_{\mathtt{lo}}, \cdot) \leftarrow \min_{t \in \{0,\ldots,\mathsf{poly}(\tau)\}}\{\mathtt{delays}(t, \mathsf{f}_{\Delta,1}, \ldots, \mathsf{f}_{\Delta,|\mathcal{W}|-1}, |\mathcal{W}| - 1)\}$, $(\cdot, \Delta_{\mathtt{hi}}) \leftarrow \max_{t \in \{0,\ldots,\mathsf{poly}(\tau)\}}\{\mathtt{delays}(t, \mathsf{f}_{\Delta,1}, \ldots, \mathsf{f}_{\Delta,|\mathcal{W}|-1}, |\mathcal{W}| - 1)\}$ and $\mathsf{f}_{\Delta,1}, \ldots, \mathsf{f}_{\Delta,|\mathcal{W}|-1}$ are the delay functions of the instances of $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,1}}, \ldots, \mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,|\mathcal{W}|-1}}$ where $\mathcal{P}_j \in \mathcal{W}$ acts as receiver.*

The proof can be found in Appendix C. The core tasks of its simulator $\mathcal{S}$ are making sure that: 1) every puzzle generated by $\mathcal{A}$ is created at $\mathcal{F}_{\mathsf{tlp}}$; and 2) every puzzle that is solved by $\mathcal{F}_{\mathsf{tlp}}$ in the ideal world is simulated towards $\mathcal{A}$. The first task is accomplished by $\mathcal{S}$ by extracting the message m and proof π from every puzzle generated by $\mathcal{A}$ and sending it to $\mathcal{F}_{\mathsf{tlp}}$. The second task is achieved by simulating an execution of $\pi_{\mathsf{TLP-Light}}$ for solving TLPs provided by $\mathcal{F}_{\mathsf{tlp}}$ and later using the leakage of m, π from $\mathcal{F}_{\mathsf{tlp}}$ to program the restricted programmable random oracles such that the output of the protocol matches m, π.

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

$\pi_{\mathsf{TLP-Light}}$ is parameterized by a cyclic group $\mathbb{G}$ of order q with generator g. $\pi_{\mathsf{TLP-Light}}$ is executed by parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, among which a subset of solvers $\mathcal{W} \subset \mathcal{P}$, interacting with $\mathcal{G}_{\mathsf{Clock}}$, $\mathcal{G}_{\mathsf{rpoRO}}{}^1$ with output in $\mathbb{Z}_{|\mathbb{G}|}$, $\mathcal{G}_{\mathsf{rpoRO}}{}^2$ with output in $\{0,1\}^{2\tau}$, $\mathcal{F}_{\mathsf{DKG}}$ and instances $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,i}}$ where $\mathcal{P}_i$ is sender and $\mathcal{P}_{i+1}$ is receiver for all $\mathcal{P}_i \in \mathcal{W}$.

Setup: When first activated, all $\mathcal{P}_i \in \mathcal{P}$ send $(\textsc{PubKey}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{DKG}}$, receiving pk, and all $\mathcal{P}_i \in \mathcal{W}$ additionally send $(\textsc{SecKey}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{DKG}}$, receiving sk_i.

Create puzzle: On input $(\mathsf{CreatePuzzle}, \mathsf{sid}, m)$, $\mathcal{P}_i$ encrypts m using pk following the steps of Definition A.4:

1. Sample $r \xleftarrow{\$} \mathbb{G}, s \xleftarrow{\$} \{0,1\}^\tau$ and send $(\textsc{Hash-Query}, r)$ to $\mathcal{G}_{\mathsf{rpoRO}}{}^2$, receiving $(\textsc{Hash-Confirm}, \mathsf{pad})$. Then send $(\textsc{Hash-Query}, m|s)$ to $\mathcal{G}_{\mathsf{rpoRO}}{}^1$, receiving $(\textsc{Hash-Confirm}, \rho)$.
2. Send $(\textsc{IsProgrammed}, m|s)$ (resp. $(\textsc{IsProgrammed}, r)$) to $\mathcal{G}_{\mathsf{rpoRO}}{}^1$ (resp. $\mathcal{G}_{\mathsf{rpoRO}}{}^2$) and abort if either of the responses is $(\textsc{IsProgrammed}, 1)$.
3. Compute $\mathsf{puz} = (c_1 = g^\rho, c_2 = r \cdot \mathsf{pk}^\rho, c_3 = (m|s) \oplus \mathsf{pad})$.
4. Output $(\mathsf{CreatedPuzzle}, \mathsf{sid}, \mathsf{puz}, \pi = (\mathsf{pk}, r, s))$.

Solve: On input $(\textsc{Solve}, \mathsf{sid}, \mathsf{puz})$, $\mathcal{P}_i$ sends $(\textsc{Solve}, \mathsf{sid}, \mathsf{puz})$ to the first $\mathcal{P}_j \in \mathcal{W}$ (*i.e.* $j = \min\{j \mid \mathcal{P}_j \in \mathcal{W}\}$). Upon receiving $(\textsc{Solved}, \mathsf{sid}, \mathsf{puz}, m, \pi)$ from the last $\mathcal{P}_\ell \in \mathcal{W}$ (*i.e.* $\ell = \max\{\ell \mid \mathcal{P}_\ell \in \mathcal{W}\}$), perform **Public Verification** on puz, m, π and set $m = \perp$ if it does not succeed. Output $(\textsc{Solved}, \mathsf{sid}, \mathsf{puz}, m, \pi)$.

Public Verification: On input $(\textsc{Verify}, \mathsf{sid}, \mathsf{puz} = (c_1, c_2, c_3), m, \pi = (\mathsf{pk}, r, s))$, $\mathcal{P}_i$ executes Steps 2 to 5 of **Create Puzzle** with pk, m, r, s to obtain puz'. If $\mathsf{puz}' = \mathsf{puz}$, $\mathcal{P}_i$ sets $b = 1$, else, it sets $b = 0$, outputting $(\textsc{Verified}, \mathsf{sid}, \mathsf{puz}, m, \pi, b)$.

Tick: Parties in $\mathcal{W}$ proceed as follows and then send $(\mathtt{Update})$ to $\mathcal{G}_{\mathsf{Clock}}$:

Starting Solution: For all $(\textsc{Solve}, \mathsf{sid}, \mathsf{puz} = (c_1, c_2, c_3))$ received in this tick, the first $\mathcal{P}_i \in \mathcal{W}$ proceeds as follows: 1. Send $(\textsc{Read})$ to $\mathcal{G}_{\mathsf{Clock}}$, obtaining $(\textsc{Read}, \nu_1)$; 2. Compute $\widehat{c}_2 = c_2 \cdot c_1^{-\mathsf{sk}_i}$; 3. Send $(\textsc{Send}, \mathsf{sid}, (\nu_1, \mathsf{puz}, \widehat{c}_2))$ to $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,1}}$.

Ongoing Solution: Every party $\mathcal{P}_j \in \mathcal{W} \setminus \mathcal{P}_i$ sends $(\textsc{Rec}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ where they act as receivers and, for every message $(\textsc{Sent}, \mathsf{sid}, (\nu_1, \mathsf{puz}, \widehat{c}_2), \nu)$ received as answer, proceed as follows:

1. Given the current time $\overline{\nu}$ obtained from $\mathcal{G}_{\mathsf{Clock}}$, ν and all the delay functions $\mathsf{f}_{\Delta,1}, \ldots, \mathsf{f}_{\Delta,j-1}$ associated to the previous instances of $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$, check that $\mathsf{isP}(\nu_1, \mathsf{f}_{\Delta,1}, \ldots, \mathsf{f}_{\Delta,j-1}, \overline{\nu})$ is true, aborting otherwise.
2. Parse $\mathsf{puz} = (c_1, c_2, c_3)$ and compute $\tilde{c}_2 = \widehat{c}_2 \cdot c_1^{-\mathsf{sk}_j}$.
3. If $\mathcal{P}_j$ is not the last party $\mathcal{P}_\ell \in \mathcal{W}$, send $(\textsc{Send}, \mathsf{sid}, (\nu_1, \mathsf{puz}, \tilde{c}_2))$ to $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,j}}$.

Delivering Result: The last party $\mathcal{P}_\ell \in \mathcal{W}$ obtains $r = \tilde{c}_2 = c_2 \cdot c_1^{-\sum_{j\in\mathcal{W}} \mathsf{sk}_j}$, sends $(\textsc{Hash-Query}, r)$ to $\mathcal{G}_{\mathsf{rpoRO}}{}^1$, receiving $(\textsc{Hash-Confirm}, \mathsf{pad})$, computes $m|s = c_3 \oplus \mathsf{pad}$ and broadcasts $(m, \pi = (\mathsf{pk}, r, s))$ to all $\mathcal{P}_i \in \mathcal{P}$.

Fig. 13: Protocol $\pi_{\mathsf{TLP-Light}}$ for Publicly Verifiable Time Lock Puzzles.

Constructing a Random Beacon. Notice that our $\mathcal{F}_{\mathsf{tlp}}$ can be used to instantiate the random beacon construction of [8]. In this construction, parties generate randomness by broadcasting (or posting to a public ledger) a PV-TLP containing a random input. After a majority of parties have provided their PV-TLPs, these PV-TLPs are opened by their owners, who present their random

input along with a proof that it was contained in their PV-TLP. In case one of the owners does not follow the protocol, the other parties can solve the unopened PV-TLP to obtain the remaining random input. Finally all parties hash all random inputs to obtain a random output. In our setting, this is particularly advantageous, since potentially sequential communication delay channels only needs to be used in case a party misbehaves. When there is no misbehavior, randomness can be obtained cheaply by locally verifying PV-TLP proofs without accessing delayed channels. Otherwise, if sequential communication delay must be used, a party who failed to open their PV-TLP is identified, so it can be excluded in future executions and/or made to pay for access to delay channels.

8 Delay Encryption and Stateless VDF

In this section, we extend our PV-TLP construction to obtain a related primitive called Delay Encryption [22]. A Delay Encryption scheme allows for encrypting many messages under a certain identity in such a way that a secret key allowing for decrypting all such messages can be obtained after a certain delay, a notion akin to an "identity based TLP". We construct this primitive by combining an IBE scheme with a distributed (identity) key generation protocol and our proofs of sequential communication delay. This Delay Encryption construction can also be converted into a stateless VDF. Since we combine standard results in order to obtain this construction, we only informally sketch it here.

Assume $\texttt{IBE} = (\texttt{Setup}, \texttt{KG}, \texttt{Enc}, \texttt{Dec})$ is an Identity-based encryption scheme where: $\texttt{IBE.Setup}$ on input the security parameter τ outputs the master secret key msk and the public key pk; $\texttt{IBE.KG}$ on input an identity string $ID \in \{0,1\}^*$ and msk outputs the identity decryption key sk_{ID}; $\texttt{IBE.Enc}$ on input the plaintext m, public key pk and identity ID outputs the ciphertext c; $\texttt{IBE.Dec}$ on input the identity decryption key sk_{ID} and the ciphertext c outputs either a message m or $\perp$. First, observe that many IBE schemes (*e.g.* [17]) are essentially a version of El Gamal. This means that $\texttt{Setup}, \texttt{KG}$ can easily be "thresholdized" to allow for generating identity secret keys from shares of msk, and that sk_{ID} is unique for each ID. As an example, consider [17] which uses two source groups G, a target group G_T and a pairing $e : G \times G \mapsto G_T$. $\texttt{Setup}$ creates $pk = g^{msk}$ for master secret key msk using a public generator $g \in G$. $\texttt{KG}$ creates a random generator $h = H(ID) \in G$ based on a hash of the identity ID using a random oracle H to G, and lets $sk_{ID} = h^{msk}$. Clearly, sk_{ID} is unique for ID. $\texttt{Enc}$ generates a ciphertext $c = (c_1, c_2)$ from m and ID by computing $c_1 = g^r$ $c_2 = m \cdot e(H(ID)^r, pk)$, and $\texttt{Dec}$ decrypts c by computing $m = c_2 \cdot e(c_1, sk_{ID})^{-1}$.

It is easy to "thresholdize" such an $\texttt{IBE}$ scheme with UC security. To implement $\texttt{Setup}$, parties use standard semi-honest El Gamal distributed key generation to create a Shamir sharing of a random secret msk and then raise g to msk using standard techniques. Additionally, they commit to their shares of msk and use UC NIZKs to prove execution correctness. Implementing $\texttt{KG}$ as a distributed protocol is again straightforward as ID is public, since each protocol participant can compute $H(ID)$ locally, raise it to its committed share of msk and prove correctness of this using a UC NIZK. Then, by reconstruction in the exponent,

one can obtain the unique $H(id)^{msk}$. By using a CCA secure version of $\mathtt{Enc}, \mathtt{Dec}$, *e.g.* [17] as shown in [59], we obtain UC security for the full encryption scheme.

Our crucial observation is that we can run a distributed key generation (DKG) protocol outputting the secret key for a given ID via delayed channels $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ that generate proofs of sequential communication. By letting intermediate parties check the key shares and proofs of delay, we can provably lower-bound the delay for creating sk_{ID}. Notice that this idea gives us a natural construction of Delay Encryption. To encrypt, we let a party knowing pk first choose an identity ID and let the ciphertext be $ID, \mathtt{Enc}(m, pk, ID)$. To decrypt one or more ciphertexts for the same ID, parties obtain the secret key sk_{ID} by running the DKG and then decrypt using sk_{ID}. The delay directly follows from the bound on the execution time of the DKG. We provide an ideal functionality for Delay Encryption in Appendix D and formalize this observation in the following theorem, which is conservatively phrased in terms of the [17] IBE, although it can be generalized to any IBE that supports distributed key generation.

Theorem 8.1. *If the IBE scheme of [17] is* $\mathrm{IND-ID-CCA2}$ *secure, there exists a protocol that UC-realizes* $\mathcal{F}_{\mathsf{DE}}$ *in the* $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}, \mathcal{F}_{\mathrm{NIZK}}, \mathcal{G}_{\mathsf{rpoRO}}$*-hybrid model against an active static adversary corrupting a majority of parties in* $\mathcal{P}$*. The delay parameter is* $\Gamma = \Delta_{\mathtt{hi}}$ *and the slack parameter is* $\epsilon = \frac{\Delta_{\mathtt{lo}}}{\Delta_{\mathtt{hi}}}$ *where* $(\cdot, \Delta_{\mathtt{hi}}) = \max_{t \in \{0,\ldots,\mathsf{poly}(\tau)\}}\{\mathsf{f}_\Delta(t)\}$ *and* $(\Delta_{\mathtt{lo}}, \cdot) = \min_{t \in \{0,\ldots,\mathsf{poly}(\tau)\}}\{\mathsf{f}_\Delta(t)\}$.

Stateless VDF: In Section 6 we have described a VDF construction that creates the random value from a proof of sequential delay. Unfortunately, in order to achieve uniqueness we have to use a bulletin board to keep track of previous VDF inputs. Departing from our Delay Encryption construction, obtaining a stateless VDF is possible as follows: assume that a threshold instance of $\mathtt{IBE}$ is set up such that $\mathtt{Setup}$ was run and pk is known. To evaluate the VDF, consider the VDF input x as an ID and run the threshold version of $\mathtt{KG}$ to generate sk_x. Then, hashing x, sk_x using a random oracle yields the VDF output, while sk_x serves as the publicly verifiable proof[8]. Unpredictability follows due to the Naor transform [35], since each sk_x can be considered as a signature of an EUF-CMA secure signature scheme (which is therefore UC secure). Uniqueness of the signature follows from the El Gamal-type of $\mathtt{IBE}$, as each sk_x is unique. The VDF delay is then identical with the runtime of $\mathtt{KG}$. We formalize this result in the following theorem, which is conservatively phrased in terms of the [17] IBE, although it can be generalized to any IBE that yields a unique signature via the Naor Transform and supports distributed key generation.

Theorem 8.2. *If the IBE scheme of [17] is* $\mathrm{IND-ID-CCA2}$ *secure, there exists a protocol that UC-realizes* $\mathcal{F}_{\mathsf{VDF}}$ *in the* $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}, \mathcal{F}_{\mathrm{NIZK}}, \mathcal{G}_{\mathsf{rpoRO}}$*-hybrid model against an active static adversary corrupting a majority of parties in* $\mathcal{P}$*. The delay parameter is* $\Gamma = \Delta_{\mathtt{hi}}$ *and the slack parameter is* $\epsilon = \frac{\Delta_{\mathtt{lo}}}{\Delta_{\mathtt{hi}}}$ *where* $(\cdot, \Delta_{\mathtt{hi}}) = \max_{t \in \{0,\ldots,\mathsf{poly}(\tau)\}}\{\mathsf{f}_\Delta(t)\}$ *and* $(\Delta_{\mathtt{lo}}, \cdot) = \min_{t \in \{0,\ldots,\mathsf{poly}(\tau)\}}\{\mathsf{f}_\Delta(t)\}$.

[8] Which can be checked by encrypting a random value to identity x, decrypting using sk_x and checking for consistency

Acknowledgment

The work described in this paper has received funding from: the Protocol Labs Research Grant Program PL-RGP1-2021-064, the Protocol Labs-CryptoSat SpaceVDF program, and the Independent Research Fund Denmark (IRFD) grants number 9040-00399B (TrA^{2}C), 9131-00075B (PUMA) and 0165-00079B.

References

1. Cryptosat. https://cryptosat.io. Accessed: 2022-10-07.
2. Pouriya Alikhani, Nicolas Brunner, Claude Crépeau, Sébastien Designolle, Raphaël Houlmann, Weixu Shi, Nan Yang, and Hugo Zbinden. Experimental relativistic zero-knowledge proofs. Nat., 599(7883):47–50, 2021.
3. Ghada Almashaqbeh, Ran Canetti, Yaniv Erlich, Jonathan Gershoni, Tal Malkin, Itsik Pe'er, Anna Roitburd-Berman, and Eran Tromer. Unclonable polymers and their cryptographic applications. In Orr Dunkelman and Stefan Dziembowski, editors, Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part I, volume 13275 of Lecture Notes in Computer Science, pages 759–789. Springer, 2022.
4. Christian Badertscher, Ueli Maurer, and Björn Tackmann. On composable security for digital signatures. In Michel Abdalla and Ricardo Dahab, editors, PKC 2018, Part I, volume 10769 of LNCS, pages 494–523. Springer, Heidelberg, March 2018.
5. Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Jonathan Katz and Hovav Shacham, editors, CRYPTO 2017, Part I, volume 10401 of LNCS, pages 324–356. Springer, Heidelberg, August 2017.
6. Ali Bagherzandi, Jung Hee Cheon, and Stanislaw Jarecki. Multisignatures secure under the discrete logarithm assumption and a generalized forking lemma. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, ACM CCS 2008, pages 449–458. ACM Press, October 2008.
7. Carsten Baum, Bernardo David, and Rafael Dowsley. (Public) Verifiability for Composable Protocols Without Adaptivity or Zero-Knowledge. In Chunpeng Ge and Fuchun Guo, editors, Provable and Practical Security - 16th International Conference, ProvSec 2022, Nanjing, China, November 11-12, 2022, Proceedings, volume 13600 of LNCS, pages 249–272. Springer, 2022.
8. Carsten Baum, Bernardo David, Rafael Dowsley, Ravi Kishore, Jesper Buus Nielsen, and Sabine Oechsner. CRAFT: Composable randomness beacons and output-independent abort mpc from time. Cryptology ePrint Archive, Paper 2020/784, 2020. To Appear at PKC 2023. https://eprint.iacr.org/2020/784.
9. Carsten Baum, Bernardo David, Rafael Dowsley, Jesper Buus Nielsen, and Sabine Oechsner. TARDIS: A foundation of time-lock puzzles in UC. In Anne Canteaut and François-Xavier Standaert, editors, EUROCRYPT 2021, Part III, volume 12698 of LNCS, pages 429–459. Springer, Heidelberg, October 2021.
10. Mihir Bellare, Chanathip Namprempre, and Gregory Neven. Unrestricted aggregate signatures. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, ICALP 2007, volume 4596 of LNCS, pages 411–422. Springer, Heidelberg, July 2007.

11. Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, ACM CCS 2006, pages 390–399. ACM Press, October / November 2006.
12. Nir Bitansky, Shafi Goldwasser, Abhishek Jain, Omer Paneth, Vinod Vaikuntanathan, and Brent Waters. Time-lock puzzles from randomized encodings. In Madhu Sudan, editor, ITCS 2016, pages 345–356. ACM, January 2016.
13. Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Yvo Desmedt, editor, PKC 2003, volume 2567 of LNCS, pages 31–46. Springer, Heidelberg, January 2003.
14. Alexandra Boldyreva, Craig Gentry, Adam O'Neill, and Dae Hyun Yum. Ordered multisignatures and identity-based sequential aggregate signatures, with applications to secure routing. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, ACM CCS 2007, pages 276–285. ACM Press, October 2007.
15. Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, CRYPTO 2018, Part I, volume 10991 of LNCS, pages 757–788. Springer, Heidelberg, August 2018.
16. Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In Thomas Peyrin and Steven Galbraith, editors, ASIACRYPT 2018, Part II, volume 11273 of LNCS, pages 435–464. Springer, Heidelberg, December 2018.
17. Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, CRYPTO 2001, volume 2139 of LNCS, pages 213–229. Springer, Heidelberg, August 2001.
18. Dan Boneh and Moni Naor. Timed commitments. In Mihir Bellare, editor, CRYPTO 2000, volume 1880 of LNCS, pages 236–254. Springer, Heidelberg, August 2000.
19. Cecilia Boschini, Akira Takahashi, and Mehdi Tibouchi. MuSig-L: Lattice-based multi-signature with single-round online phase. In Yevgeniy Dodis and Thomas Shrimpton, editors, CRYPTO 2022, Part II, volume 13508 of LNCS, pages 276–305. Springer, Heidelberg, August 2022.
20. Kyle Brogle, Sharon Goldberg, and Leonid Reyzin. Sequential aggregate signatures with lazy verification from trapdoor permutations - (extended abstract). In Xiaoyun Wang and Kazue Sako, editors, ASIACRYPT 2012, volume 7658 of LNCS, pages 644–662. Springer, Heidelberg, December 2012.
21. Christina Brzuska, Marc Fischlin, Heike Schröder, and Stefan Katzenbeisser. Physically uncloneable functions in the universal composition framework. In Phillip Rogaway, editor, CRYPTO 2011, volume 6841 of LNCS, pages 51–70. Springer, Heidelberg, August 2011.
22. Jeffrey Burdges and Luca De Feo. Delay encryption. In Anne Canteaut and François-Xavier Standaert, editors, EUROCRYPT 2021, Part I, volume 12696 of LNCS, pages 302–326. Springer, Heidelberg, October 2021.
23. Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, EUROCRYPT 2018, Part I, volume 10820 of LNCS, pages 280–312. Springer, Heidelberg, April / May 2018.
24. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In 42nd FOCS, pages 136–145. IEEE Computer Society Press, October 2001.

25. Ran Canetti. Universally composable signatures, certification and authentication. Cryptology ePrint Archive, Report 2003/239, 2003. https://eprint.iacr.org/2003/239.
26. Ran Canetti. Universally composable signature, certification, and authentication. In 17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA, page 219. IEEE Computer Society, 2004.
27. Ran Canetti. Universally composable signature, certification, and authentication. In 17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA, page 219. IEEE Computer Society, 2004.
28. Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, TCC 2007, volume 4392 of LNCS, pages 61–85. Springer, Heidelberg, February 2007.
29. Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, ACM CCS 2020, pages 1769–1787. ACM Press, November 2020.
30. Ran Canetti and Jonathan Herzog. Universally composable symbolic analysis of mutual authentication and key-exchange protocols. In Shai Halevi and Tal Rabin, editors, TCC 2006, volume 3876 of LNCS, pages 380–403. Springer, Heidelberg, March 2006.
31. Pyrros Chaidos and Aggelos Kiayias. Mithril: Stake-based threshold multisignatures. Cryptology ePrint Archive, Report 2021/916, 2021. https://eprint.iacr.org/2021/916.
32. Yanbo Chen and Yunlei Zhao. Half-aggregation of schnorr signatures with tight reductions. Cryptology ePrint Archive, Report 2022/222, 2022. https://eprint.iacr.org/2022/222.
33. Claude Crépeau and Joe Kilian. Achieving oblivious transfer using weakened security assumptions (extended abstract). In 29th FOCS, pages 42–52. IEEE Computer Society Press, October 1988.
34. Claude Crépeau, Arnaud Massenet, Louis Salvail, Lucas Shigeru Stinchcombe, and Nan Yang. Practical relativistic zero-knowledge for NP. In Yael Tauman Kalai, Adam D. Smith, and Daniel Wichs, editors, ITC 2020, pages 4:1–4:18. Schloss Dagstuhl, June 2020.
35. Yang Cui, Eiichiro Fujisaki, Goichiro Hanaoka, Hideki Imai, and Rui Zhang. Formal security treatments for signatures from identity-based encryption. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, ProvSec 2007, volume 4784 of LNCS, pages 218–227. Springer, Heidelberg, November 2007.
36. Luca De Feo, Simon Masson, Christophe Petit, and Antonio Sanso. Verifiable delay functions from supersingular isogenies and pairings. In Steven D. Galbraith and Shiho Moriai, editors, ASIACRYPT 2019, Part I, volume 11921 of LNCS, pages 248–277. Springer, Heidelberg, December 2019.
37. Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. On the security of two-round multi-signatures. In 2019 IEEE Symposium on Security and Privacy, pages 1084–1101. IEEE Computer Society Press, May 2019.
38. Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. Continuous verifiable delay functions. In Anne Canteaut and Yuval Ishai, editors, EUROCRYPT 2020, Part III, volume 12107 of LNCS, pages 125–154. Springer, Heidelberg, May 2020.

39. Marc Fischlin, Anja Lehmann, and Dominique Schröder. History-free sequential aggregate signatures. In Ivan Visconti and Roberto De Prisco, editors, SCN 12, volume 7485 of LNCS, pages 113–130. Springer, Heidelberg, September 2012.
40. Cody Freitag, Ilan Komargodski, Rafael Pass, and Naomi Sirkin. Non-malleable time-lock puzzles and applications. In Kobbi Nissim and Brent Waters, editors, TCC 2021, Part III, volume 13044 of LNCS, pages 447–479. Springer, Heidelberg, November 2021.
41. Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, CRYPTO 2018, Part II, volume 10992 of LNCS, pages 33–62. Springer, Heidelberg, August 2018.
42. Eiichiro Fujisaki and Tatsuaki Okamoto. How to enhance the security of public-key encryption at minimum cost. In Hideki Imai and Yuliang Zheng, editors, PKC'99, volume 1560 of LNCS, pages 53–68. Springer, Heidelberg, March 1999.
43. Craig Gentry, Adam O'Neill, and Leonid Reyzin. A unified framework for trapdoor-permutation-based sequential aggregate signatures. In Michel Abdalla and Ricardo Dahab, editors, PKC 2018, Part II, volume 10770 of LNCS, pages 34–57. Springer, Heidelberg, March 2018.
44. Shafi Goldwasser and Rafail Ostrovsky. Invariant signatures and non-interactive zero-knowledge proofs are equivalent (extended abstract). In Ernest F. Brickell, editor, CRYPTO'92, volume 740 of LNCS, pages 228–245. Springer, Heidelberg, August 1993.
45. Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In Daniele Micciancio, editor, TCC 2010, volume 5978 of LNCS, pages 308–326. Springer, Heidelberg, February 2010.
46. Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In Moni Naor, editor, EUROCRYPT 2007, volume 4515 of LNCS, pages 115–128. Springer, Heidelberg, May 2007.
47. Jonathan Katz, Julian Loss, and Jiayu Xu. On the security of time-lock puzzles and timed commitments. In Rafael Pass and Krzysztof Pietrzak, editors, TCC 2020, Part III, volume 12552 of LNCS, pages 390–413. Springer, Heidelberg, November 2020.
48. Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, TCC 2013, volume 7785 of LNCS, pages 477–498. Springer, Heidelberg, March 2013.
49. Adrian Kent. Unconditionally secure bit commitment. Physical Review Letters, 83(7):1447, 1999.
50. Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In Marc Fischlin and Jean-Sébastien Coron, editors, EUROCRYPT 2016, Part II, volume 9666 of LNCS, pages 705–734. Springer, Heidelberg, May 2016.
51. Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. Sequential aggregate signatures and multisignatures without random oracles. In Serge Vaudenay, editor, EUROCRYPT 2006, volume 4004 of LNCS, pages 465–485. Springer, Heidelberg, May / June 2006.
52. Tommaso Lunghi, Jedrzej Kaniewski, Felix Bussières, Raphael Houlmann, Marco Tomamichel, Stephanie Wehner, and Hugo Zbinden. Practical relativistic bit commitment. Physical Review Letters, 115(3):030502, 2015.
53. Anna Lysyanskaya, Silvio Micali, Leonid Reyzin, and Hovav Shacham. Sequential aggregate signatures from trapdoor permutations. In Christian Cachin and

Jan Camenisch, editors, EUROCRYPT 2004, volume 3027 of LNCS, pages 74–90. Springer, Heidelberg, May 2004.

54. Christian Matt, Jesper Buus Nielsen, and Søren Eller Thomsen. Formalizing delayed adaptive corruptions and the security of flooding networks. In Yevgeniy Dodis and Thomas Shrimpton, editors, CRYPTO 2022, Part II, volume 13508 of LNCS, pages 400–430. Springer, Heidelberg, August 2022.
55. Ueli M. Maurer. Protocols for secret key agreement by public discussion based on common information. In Ernest F. Brickell, editor, CRYPTO'92, volume 740 of LNCS, pages 461–470. Springer, Heidelberg, August 1993.
56. Silvio Micali, Kazuo Ohta, and Leonid Reyzin. Accountable-subgroup multisignatures: Extended abstract. In Michael K. Reiter and Pierangela Samarati, editors, ACM CCS 2001, pages 245–254. ACM Press, November 2001.
57. Gregory Neven. Efficient sequential aggregate signed data. In Nigel P. Smart, editor, EUROCRYPT 2008, volume 4965 of LNCS, pages 52–69. Springer, Heidelberg, April 2008.
58. Jonas Nick, Tim Ruffing, and Yannick Seurin. MuSig2: Simple two-round Schnorr multi-signatures. In Tal Malkin and Chris Peikert, editors, CRYPTO 2021, Part I, volume 12825 of LNCS, pages 189–221, Virtual Event, August 2021. Springer, Heidelberg.
59. Ryo Nishimaki, Yoshifumi Manabe, and Tatsuaki Okamoto. Universally composable identity-based encryption. In Phong Q. Nguyen, editor, Progress in Cryptology - VIETCRYPT 2006, pages 337–353, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
60. Ravikanth Pappu, Ben Recht, Jason Taylor, and Neil Gershenfeld. Physical one-way functions. Science, 297(5589):2026–2030, 2002.
61. Krzysztof Pietrzak. Simple verifiable delay functions. In Avrim Blum, editor, ITCS 2019, volume 124, pages 60:1–60:15. LIPIcs, January 2019.
62. David Pointcheval. Chosen-ciphertext security for any one-way cryptosystem. In Hideki Imai and Yuliang Zheng, editors, PKC 2000, volume 1751 of LNCS, pages 129–146. Springer, Heidelberg, January 2000.
63. J. Puig-Suari, C. Turner, and W. Ahlgren. Development of the standard cubesat deployer and a cubesat class picosatellite. In 2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542), volume 1, pages 1/347–1/353 vol.1, 2001.
64. Thomas Ristenpart and Scott Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In Moni Naor, editor, EUROCRYPT 2007, volume 4515 of LNCS, pages 228–245. Springer, Heidelberg, May 2007.
65. Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto, 1996.
66. Ulrich Rührmair and Marten van Dijk. On the practical use of physical unclonable functions in oblivious transfer and bit commitment protocols. Journal of Cryptographic Engineering, 3:17–28, 2013.
67. Ephanielle Verbanis, Anthony Martin, Raphaël Houlmann, Gianluca Boso, Félix Bussières, and Hugo Zbinden. 24-hour relativistic bit commitment. Phys. Rev. Lett., 117:140506, Sep 2016.
68. Benjamin Wesolowski. Efficient verifiable delay functions. In Yuval Ishai and Vincent Rijmen, editors, EUROCRYPT 2019, Part III, volume 11478 of LNCS, pages 379–407. Springer, Heidelberg, May 2019.

A Auxiliary Functionalities and other Preliminaries

We use the (Global) Universal Composability or (G)UC model [24, 28] for analyzing security and refer interested readers to the original works for more details.

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

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

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

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

The Global Random Oracle. In Figure Fig. 14 we present the restricted observable and programmable global random oracle ideal functionality from [23]. It follows the standard notion of a random oracle, when defined in the UC framework.

Key Registration Ideal Functionality $\mathcal{F}_{\mathsf{Reg}}$. The key registration functionality $\mathcal{F}_{\mathsf{Reg}}$ is presented in Figure 15. This ideal functionality captures a public key infrastructure, allowing parties to register their public keys in such a way that other parties can retrieve public keys with the guarantee that they belong to the party who originally registered them. $\mathcal{F}_{\mathsf{Reg}}$ is inspired by the functionality from [30], but additionally supports timestamps on registered keys.

Unique Digital Signatures Ideal Functionality $\mathcal{F}_{\mathsf{Sig}}$. The standard digital signature functionality $\mathcal{F}_{\mathsf{Sig}}$ from [27] captures a randomized signature scheme where the signer may influence the generation of a signature by choosing the randomness used by the signing algorithm. This particularity is captured by allowing the ideal adversary $\mathcal{S}$ choose a new string σ to represent a signature on a message m every time the signer $\mathcal{P}_s$ (a special party who has the right to generate signatures, *i.e.*, who holds the signature key) makes a new request for a signature on m. This process allows for multiple valid signatures to be produced for the same message. However, we require a unique signature scheme [44] for

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

$\mathcal{G}_{\mathsf{rpoRO}}$ is parameterized by an output size function ℓ and a security parameter τ, and keeps initially empty lists $\mathsf{List}_{\mathcal{H}}$,$\mathsf{prog}$.

Query: On input $(\textsc{Hash-Query}, m)$ from party $(\mathcal{P}, \mathsf{sid})$ or $\mathcal{S}$, parse m as (s, m') and proceed as follows:

1. Look up h such that $(m, h) \in \mathsf{List}_{\mathcal{H}}$. If no such h exists, sample $h \xleftarrow{\$} \{0,1\}^{\ell(\tau)}$ and set $\mathsf{List}_{\mathcal{H}} = \mathsf{List}_{\mathcal{H}} \cup \{(m, h)\}$.
2. If this query is made by $\mathcal{S}$, or if $s \neq \mathsf{sid}$, then add (s, m', h) to the (initially empty) list of illegitimate queries $\mathcal{Q}_s$.
3. Send $(\textsc{Hash-Confirm}, h)$ to the caller.

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

Program: On input $(\textsc{Program-RO}, m, h)$ with $h \in \{0,1\}^{\ell(\tau)}$ from $\mathcal{S}$, ignore the input if there exists $h' \in \{0,1\}^{\ell(\tau)}$ where $(m, h') \in \mathsf{List}_{\mathcal{H}}$ and $h \neq h'$. Otherwise, set $\mathsf{List}_{\mathcal{H}} = \mathsf{List}_{\mathcal{H}} \cup \{(m, h)\}$, $\mathsf{prog} = \mathsf{prog} \cup \{m\}$ and send $(\textsc{Program-Confirm})$ to $\mathcal{S}$.

IsProgrammed: On input $(\textsc{IsProgrammed}, m)$ from a party $\mathcal{P}$ or $\mathcal{S}$, if the input was given by $(\mathcal{P}, \mathsf{sid})$ then parse m as (s, m') and, if $s \neq \mathsf{sid}$, ignore this input. Set $b = 1$ if $m \in \mathsf{prog}$ and $b = 0$ otherwise. Then send $(\textsc{IsProgrammed}, b)$ to the caller.

Fig. 14: Restricted observable and programmable global random oracle functionality $\mathcal{G}_{\mathsf{rpoRO}}$ from [23].

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

$\mathcal{F}_{\mathsf{Reg}}$ interacts with a set of parties $\mathcal{P}$ and an ideal adversary $\mathcal{S}$ as well as a global clock $\mathcal{G}_{\mathsf{Clock}}$ as follows:

Key Registration: Upon receiving a message $(\textsc{register}, \mathsf{sid}, \mathsf{pk})$ from a party $\mathcal{P}_i \in \mathcal{P}$:

1. Send $(\textsc{Read})$ to $\mathcal{G}_{\mathsf{Clock}}$, waiting for response $(\textsc{Read}, \nu)$.
2. Send $(\textsc{Registering}, \mathsf{sid}, \mathsf{pk}, \mathcal{P}_i, \nu)$ to $\mathcal{S}$. Upon receiving $(\mathsf{sid}, ok, \mathcal{P}_i)$ from $\mathcal{S}$, and if this is the first message from $\mathcal{P}_i$, then record the tuple $(\mathcal{P}_i, \mathsf{pk}, \nu)$.

Key Retrieval: Upon receiving a message $(\textsc{Retrieve}, \mathsf{sid}, \mathcal{P}_j)$ from a party $\mathcal{P}_i \in \mathcal{P}$, send message $(\textsc{Retrieve}, \mathsf{sid}, \mathcal{P}_j)$ to $\mathcal{S}$ and wait for it to return a message $(\textsc{Retrieve}, \mathsf{sid}, ok)$. Then, if there is a recorded tuple $(\mathcal{P}_j, \mathsf{pk}, \nu)$ output $(\textsc{Retrieve}, \mathsf{sid}, \mathcal{P}_j, \mathsf{pk}, \nu)$ to $\mathcal{P}_i$. Otherwise, if there is no recorded tuple, return $(\textsc{Retrieve}, \mathsf{sid}, \mathcal{P}_j, \perp)$.

Fig. 15: Functionality $\mathcal{F}_{\mathsf{Reg}}$ for Key Registration.

our applications to proofs of sequential communication. In a unique signature scheme, only one signature may be produced for a given message m under a signing key. In the UC formalization of signature schemes, an instance of the functionality $\mathcal{F}_{\mathsf{Sig}}$ itself represents each different signing key by allowing only a special party $\mathcal{P}_s$ (*i.e.* the holder of a signing key) to produce signatures. Hence, we capture the notion of unique signatures by only allowing one signature on a

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

Given an ideal adversary $\mathcal{S}$, verifiers $\mathcal{V}$ and a signer $\mathcal{P}_s$, $\mathcal{F}_{\mathsf{Sig}}$ performs:

Key Generation: Upon receiving a message $(\textsc{keygen}, \mathsf{sid})$ from $\mathcal{P}_s$, verify that $\mathsf{sid} = (\mathcal{P}_s, \mathsf{sid}')$ for some sid'. If not, ignore the request. Else, hand $(\textsc{keygen}, \mathsf{sid})$ to the adversary $\mathcal{S}$. Upon receiving $(\textsc{verification key}, \mathsf{sid}, \mathsf{SIG}.vk)$ from $\mathcal{S}$, output $(\textsc{verification key}, \mathsf{sid}, \mathsf{SIG}.vk)$ to $\mathcal{P}_s$, and record the pair $(\mathcal{P}_s, \mathsf{SIG}.vk)$.

Signature Generation: Upon receiving a message $(\textsc{sign}, \mathsf{sid}, m)$ from $\mathcal{P}_s$, verify that $\mathsf{sid} = (\mathcal{P}_s, \mathsf{sid}')$ for some sid' . If not, then ignore the request. Else, if an entry $(m, \sigma, \mathsf{SIG}.vk, 1)$ is recorded, output $(\textsc{signature}, \mathsf{sid}, m, \sigma)$ to $\mathcal{P}_s$ and ignore the next steps (this condition guarantees uniqueness). Else, send $(\textsc{sign}, \mathsf{sid}, m)$ to $\mathcal{S}$. Upon receiving $(\textsc{signature}, \mathsf{sid}, m, \sigma)$ from $\mathcal{S}$, verify that no entry $(m, \sigma, \mathsf{SIG}.vk, 0)$ is recorded. If it is, then output an error message to $\mathcal{P}_s$ and halt. Else, output $(\textsc{signature}, \mathsf{sid}, m, \sigma)$ to $\mathcal{P}_s$, and record the entry $(m, \sigma, \mathsf{SIG}.vk, 1)$.

Signature Verification: Upon receiving a message $(\textsc{verify}, \mathsf{sid}, m, \sigma, \mathsf{SIG}.vk')$ from some party $\mathcal{V}_i \in \mathcal{V}$, hand $(\textsc{verify}, \mathsf{sid}, m, \sigma, \mathsf{SIG}.vk')$ to $\mathcal{S}$. Upon receiving $(\textsc{verified}, \mathsf{sid}, m, \phi)$ from $\mathcal{S}$ do:

1. If $\mathsf{SIG}.vk' = \mathsf{SIG}.vk$ and the entry $(m, \sigma, \mathsf{SIG}.vk, 1)$ is recorded, then set $f = 1$. (This condition guarantees completeness: If the verification key $\mathsf{SIG}.vk'$ is the registered one and σ is a legitimately generated signature for m, then the verification succeeds.)
2. Else, if $\mathsf{SIG}.vk' = \mathsf{SIG}.vk$, the signer $\mathcal{P}_s$ is not corrupted, and no entry $(m, \sigma', \mathsf{SIG}.vk, 1)$ for any σ' is recorded, then set $f = 0$ and record the entry $(m, \sigma, \mathsf{SIG}.vk, 0)$. (This condition guarantees unforgeability: If $\mathsf{SIG}.vk'$ is the registered one, the signer is not corrupted, and never signed m, then the verification fails.)
3. Else, if there is an entry $(m, \sigma, \mathsf{SIG}.vk', f')$ recorded, then let $f = f'$. (This condition guarantees consistency: All verification requests with identical parameters will result in the same answer.)
4. Else, let $f = \phi$ and record the entry $(m, \sigma, \mathsf{SIG}.vk', \phi)$.

Output $(\textsc{verified}, \mathsf{sid}, m, f)$ to $\mathcal{V}_i$.

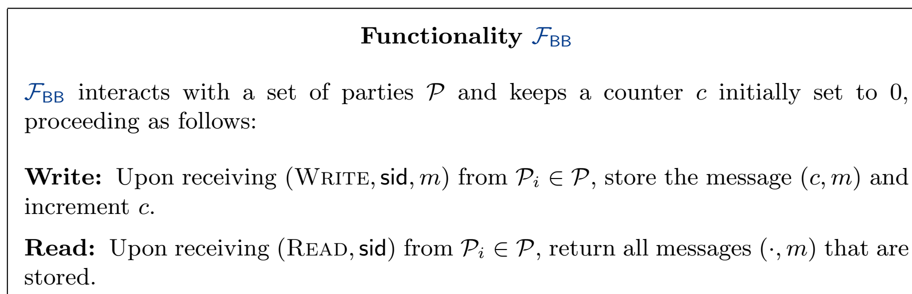
Fig. 16: Functionality $\mathcal{F}_{\mathsf{Sig}}$ for Unique Digital Signatures. Modifications with respect to the functionality of [27] are written in this font.

given message m to be produced by the same instance of $\mathcal{F}_{\mathsf{Sig}}$. The remainder of this functionality still follows the same steps as the standard one from [27]. Our modified $\mathcal{F}_{\mathsf{Sig}}$ capturing unique signatures is presented in Figure 16, where modifications with respect to [27] are written in this font.

It is shown in [27] that any EUF-CMA signature scheme UC realizes the standard signature functionality where multiple valid signatures may be produced for the same message under the same signing key (*i.e.* the same instance of $\mathcal{F}_{\mathsf{Sig}}$ may generate multiple signatures for the same message, as long as they have not been flagged as invalid signatures by a previous unsuccessful verification procedure). We observe that this fact trivially extends to the case of unique signatures, *i.e.*, any EUF-CMA signature scheme UC realizes our $\mathcal{F}_{\mathsf{Sig}}$ capturing

unique signatures, since the only restriction in this case is that a single signature is produced for each message by a single instance of $\mathcal{F}_{\mathsf{Sig}}$ (which represents a signer's signing key).

Bulletin Board Ideal Functionality $\mathcal{F}_{\mathsf{BB}}$. In Fig. 17 we describe an authenticated bulletin board functionality which is used throughout this work. Authenticated Bulletin Boards can be constructed from regular bulletin boards using $\mathcal{F}_{\mathsf{Sig}}, \mathcal{F}_{\mathsf{Reg}}$ and standard techniques.



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

$\mathcal{F}_{\mathsf{BB}}$ interacts with a set of parties $\mathcal{P}$ and keeps a counter c initially set to 0, proceeding as follows:

Write: Upon receiving $(\textsc{Write}, \mathsf{sid}, m)$ from $\mathcal{P}_i \in \mathcal{P}$, store the message (c, m) and increment c.

Read: Upon receiving $(\textsc{Read}, \mathsf{sid})$ from $\mathcal{P}_i \in \mathcal{P}$, return all messages $(\cdot, m)$ that are stored.

Fig. 17: Functionality $\mathcal{F}_{\mathsf{BB}}$ for an authenticated Bulletin Board.

A.1 UC Secure Public-Key Encryption with Plaintext Verification

Semantics of a public-key encryption scheme. We consider public-key encryption schemes PKE that have public-key $\mathcal{PK}$, secret key $\mathcal{SK}$, message $\mathcal{M}$, randomness $\mathcal{R}$ and ciphertext $\mathcal{C}$ spaces that are functions of the security parameter τ, and consist of a PPT key generation algorithm KG, a PPT encryption algorithm Enc and a deterministic decryption algorithm Dec. For $(\mathsf{pk}, \mathsf{sk}) \stackrel{\$}{\leftarrow} \mathsf{KG}(1^\tau)$, any $\mathsf{m} \in \mathcal{M}$, and $\mathsf{ct} \stackrel{\$}{\leftarrow} \mathsf{Enc}(\mathsf{pk}, \mathsf{m})$, it should hold that $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) = \mathsf{m}$ with overwhelming probability over the used randomness.

Moreover, we extend the semantics of public-key encryption by adding a plaintext verification algorithm $\{0, 1\} \leftarrow \mathsf{V}(\mathsf{ct}, \mathsf{m}, \pi)$ that outputs 1 if m is the plaintext message contained in ciphertext ct given a valid proof π that also contains the public-key pk used to generate the ciphertext. Furthermore, we modify the encryption and decryption algorithms as follows: $(\mathsf{ct}, \pi) \stackrel{\$}{\leftarrow} \mathsf{Enc}(\mathsf{pk}, \mathsf{m})$ and $(\mathsf{m}, \pi) \leftarrow \mathsf{Dec}(\mathsf{sk}, \mathsf{ct})$ now output a valid proof π that m is contained in ct. The security guarantees provided by the verification algorithm are laid out in Definition A.2.

Definition A.2 (Plaintext Verification). *Let* $\mathsf{PKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{V})$ *be a public-key encryption scheme and* τ *be a security parameter. Then* PKE *has plaintext verification if for every PPT adversary* $\mathcal{A}$*, it holds that:*

$$\Pr\left[\mathsf{V}(\mathsf{ct},\mathsf{m}',\pi')=1 \middle| \begin{array}{c} \mathsf{pk} \xleftarrow{\$} \mathcal{PK}, (\mathsf{m},\pi,\mathsf{m}',\pi') \xleftarrow{\$} \mathcal{A}(\mathsf{pk}), \\ \pi=(\mathsf{pk},r), \pi'=(\mathsf{pk},r') \in \mathcal{PK}\cup\mathcal{R}, \\ \mathsf{m},\mathsf{m}' \in \mathcal{M}, (\mathsf{ct},\pi) \leftarrow \mathsf{Enc}(\mathsf{pk},\mathsf{m};r), \mathsf{m}' \neq \mathsf{m} \end{array}\right] \in \mathsf{negl}(\tau)$$

IND-CCA secure Cryptosystem with Plaintext Verification based on [62] from [7]. This cryptosystem can be constructed from any Partially Trapdoor One-Way Injective Function in the random oracle model. Moreover, as observed in [7], it can be instantiated in the restricted observable and programmable global random oracle model of [23]. First we recall the definition of Partially Trapdoor One-Way Functions.

Definition A.3 (Partially Trapdoor One-Way Function [62]). *The function $f : \mathcal{X} \times \mathcal{Y} \rightarrow \mathcal{Z}$ is said to be partially trapdoor one-way if:*

- *For any given $z = f(x, y)$, it is computationally impossible to get back a compatible x. Such an x is called a partial preimage of z. More formally, for any polynomial time adversary A, its success, defined by $\mathtt{Succ}_{\mathtt{A}} = Pr_{x,y}[\exists y'$, $f(x', y') = f(x, y)|x' = A(f(x, y))]$, is negligible. It is one-way even for just finding partial-preimage, thus partial one-wayness.*
- *Using some extra information (the trapdoor), for any given $z \in f(\mathcal{X} \times \mathcal{Y})$, it is easily possible to get back an x, such that there exists a y which satisfies $f(x, y) = z$. The trapdoor does not allow a total inversion, but just a partial one and it is thus called a partial trapdoor.*

As observed in [62], the classical El Gamal cryptosystem is a partially trapdoor one-way injective function under the Computational Diffie Hellman (CDH) assumption, implying an instantiation of this cryptosystem under CDH. We will later exploit this fact to apply this transformation to a simple threshold version of El Gamal where the encryption procedure and the public key are exactly the same as in the standard scheme, allowing for the construction below to be instantiated. We now recall this generic construction.

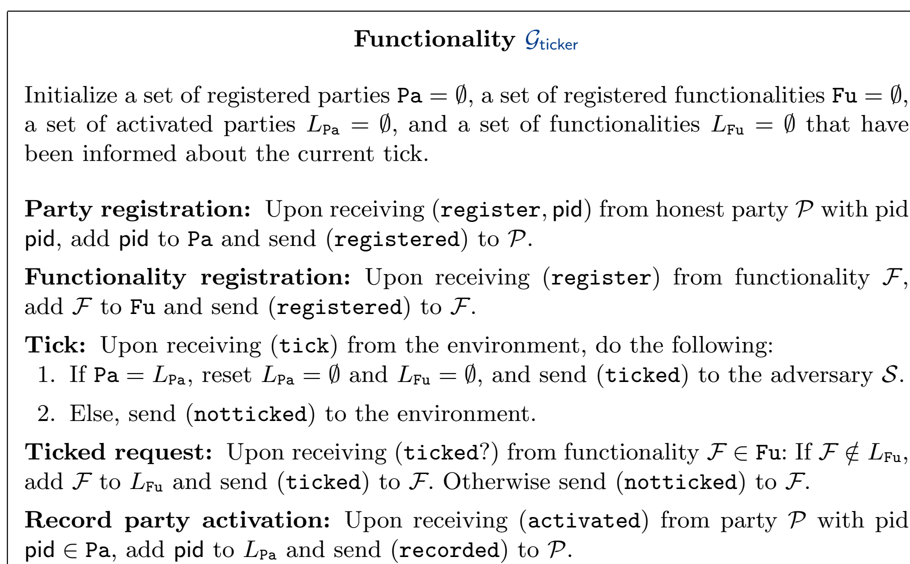
Definition A.4 (Pointcheval [62] IND-CCA Secure Cryptosystem with Plaintext Verification). *Let $\mathcal{TD}$ be a family of partially trapdoor one-way injective functions and let $H : \{0,1\}^{|m|+\tau} \rightarrow \mathcal{Y}$ and $G : \mathcal{X} \rightarrow \{0,1\}^{|m|+\tau}$ be random oracles, where $|m|$ is message length. This cryptosystem consists of the algorithms $\mathsf{PKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec}\,\mathsf{V})$ that work as follows:*

- $\mathsf{KG}(1^\tau)$*: Sample a random partially trapdoor one-way injective function f : $\mathcal{X} \times \mathcal{Y} \rightarrow \mathcal{Z}$ from $\mathcal{TD}$ and denote its inverse parameterized by the trapdoor by $f^{-1} : \mathcal{Z} \rightarrow \mathcal{X}$. The public-key is $\mathsf{pk} = f$ and the secret key is $\mathsf{sk} = (f, f^{-1})$.*
- $\mathsf{Enc}(\mathsf{pk}, \mathsf{m})$*: Sample $r \xleftarrow{\$} \mathcal{X}$ and $s \xleftarrow{\$} \{0,1\}^\tau$. Compute $a \leftarrow f(r, H(\mathsf{m}|s))$ and $b = (\mathsf{m}|s) \oplus G(r)$, outputting $\mathsf{ct} = (a, b)$ as the ciphertext and $\pi = (\mathsf{pk}, r, s)$ as the proof.*

– $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct})$*: Given a ciphertext* $\mathsf{ct} = (a, b)$ *and secret key* $\mathsf{sk} = f^{-1}$*, compute* $r \leftarrow f^{-1}(a)$ *and* $M \leftarrow b \oplus G(r)$*. If* $a = f(r, H(M))$*, parse* $M = (m|s)$ *and output* m *and the proof* $\pi = (\mathsf{pk}, r, s)$ *. Otherwise, output* $\bot$*.*
– $\mathsf{V}(\mathsf{ct}, \mathsf{m}, \pi)$*: Parse* $\pi = (\mathsf{pk}, r, s)$*, compute* $\mathsf{ct}' \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{m}, (r, s))$ *and output* 1 *if and only if* $\mathsf{ct} = \mathsf{ct}'$*.*

A.2 Global Clocks and Global tickers

The TARDIS [9] global ticker functionality $\mathcal{G}_{\mathsf{ticker}}$ is depicted in Fig. 18.



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

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

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

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

Tick: Upon receiving $(\mathtt{tick})$ from the environment, do the following:
1. If $\mathtt{Pa} = L_{\mathtt{Pa}}$, reset $L_{\mathtt{Pa}} = \emptyset$ and $L_{\mathtt{Fu}} = \emptyset$, and send $(\mathtt{ticked})$ to the adversary $\mathcal{S}$.
2. Else, send $(\mathtt{notticked})$ to the environment.

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

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

Fig. 18: Global ticker functionality $\mathcal{G}_{\mathsf{ticker}}$(from [9]).

Synchronicity and Global Clocks As mentioned in Section 2 we need to assume that honest parties have synchronized clocks. This is necessary to argue about communication delays that depend on the relative position of two parties, which evolves in time. We capture this notion of synchronicity by using a global clock functionality $\mathcal{G}_{\mathsf{Clock}}$ (see Fig. 1). In the definition that we use throughout the main body, $\mathcal{G}_{\mathsf{Clock}}$ allows users to query the current time and increments an internal time counter once all functionalities and honest parties activate a clock update interface after the last update. However, to realize $\mathcal{G}_{\mathsf{Clock}}$ in the abstract composable time model we update its internal time counter when $\mathcal{G}_{\mathsf{ticker}}$ issues a new tick.

Global Clocks in the TARDIS [9] model: In order to integrate the global functionality $\mathcal{G}_{\mathsf{Clock}}$ into the abstract composable time model, we modify it as outlined above. This modification captures the fact that $\mathcal{G}_{\mathsf{Clock}}$ exposes towards the parties and other ideal functionalities the number of ticks issues by $\mathcal{G}_{\mathsf{ticker}}$

since the beginning of the execution. However, it is not a separate clock that is executed independently from $\mathcal{G}_{\mathsf{ticker}}$. Since we wish $\mathcal{G}_{\mathsf{Clock}}$ to count the ticks issued by $\mathcal{G}_{\mathsf{ticker}}$, our modified version of $\mathcal{G}_{\mathsf{Clock}}$ requires all honest parties to activate the global ticker every time they would update the global clock (*i.e.* when they have executed all their instructions for a given round). This modification can be seen in Fig. 19. It is immediate how this clock functionality can be used to replace the global $\mathcal{G}_{\mathsf{Clock}}$ throughout our protocols by replacing UPDATE messages to $\mathcal{G}_{\mathsf{Clock}}$ by ACTIVATED calls to $\mathcal{G}_{\mathsf{ticker}}$.

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

$\mathcal{G}_{\mathsf{Clock}}$ interacts with a sets $\mathcal{P}, \mathcal{F}$ of parties and functionalities, respectively, as well as with $\mathcal{G}_{\mathsf{ticker}}$. It keeps a counter ν initially set to 0.

Clock Read: Upon receiving (READ) from any entity, answer with (READ, ν).

Tick: Increment ν, *i.e.* set $\nu \leftarrow \nu + 1$.

Fig. 19: Functionality $\mathcal{G}_{\mathsf{Clock}}$ for a Global Clock in the TARDIS model.

B Delayed Communication - Proofs and more details

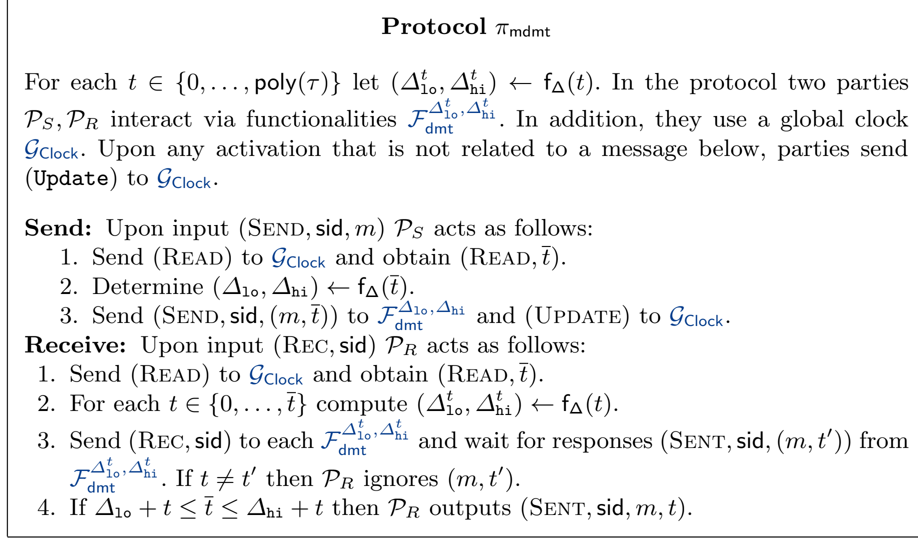
B.1 Realizing $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$

The multiple-use ideal functionality $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ for authenticated delayed message transmission can be realized in the $\mathcal{G}_{\mathsf{Clock}}, \mathcal{F}_{\mathsf{dmt}}^{\Delta_{\mathtt{lo}},\Delta_{\mathtt{hi}}}$-hybrid model. Assume access to many instances of the single-use functionality $\mathcal{F}_{\mathsf{dmt}}^{\Delta_{\mathtt{lo}},\Delta_{\mathtt{hi}}}$, one fresh instance of $\mathcal{F}_{\mathsf{dmt}}^{\Delta^t_{\mathtt{lo}},\Delta^t_{\mathtt{hi}}}$ associated to t for each message to be sent at time $t \in \{0, \dots, \mathsf{poly}(\tau)\}$ with parameters $(\Delta^t_{\mathtt{lo}}, \Delta^t_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(t)$. Upon receiving an input $(\text{SEND}, \mathsf{sid}, m)$, a sender $\mathcal{P}_S$ determines $(\Delta^t_{\mathtt{lo}}, \Delta^t_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(t)$ and uses the instance of $\mathcal{F}_{\mathsf{dmt}}^{\Delta^t_{\mathtt{lo}},\Delta^t_{\mathtt{hi}}}$ to send (m, t). Upon receiving input $(\text{REC}, \mathsf{sid})$, a receiver $\mathcal{P}_R$ queries all instances of $\mathcal{F}_{\mathsf{dmt}}^{\Delta^{t'}_{\mathtt{lo}},\Delta^{t'}_{\mathtt{hi}}}$ associated to a time t' smaller than current time t in order to retrieve messages that might have been sent. It then has to establish correctness of the delay.

Theorem B.1. *The protocol π_{mdmt} in Figure 20 GUC-securely implements $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ in the $\mathcal{G}_{\mathsf{Clock}}, \mathcal{F}_{\mathsf{dmt}}^{\cdot}$-hybrid model against a static active adversary.*

Proof. We now construct a PPT simulator $\mathcal{S}$ for a corrupted sender or receiver. In both cases, the simulator will simulate all hybrid instances of $\mathcal{F}_{\mathsf{dmt}}^{\cdot}$, which can be done in time polynomial in τ as there are only $\mathsf{poly}(\tau)$ such instances.

If $\mathcal{P}_S$ is corrupted then we construct $\mathcal{S}$ as follows: $\mathcal{S}$ acts like an honest $\mathcal{P}_R$, but it additionally observes all inputs $(\text{SEND}, \mathsf{sid}, m)$ to any instance of $\mathcal{F}_{\mathsf{dmt}}^{\cdot}$ that it simulates. Any input of the form (m, t) to $\mathcal{F}_{\mathsf{dmt}}^{\Delta_{\mathtt{lo}},\Delta_{\mathtt{hi}}}$ with $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) = \mathsf{f}_\Delta(t)$ is forwarded as $(\text{SEND}, \mathsf{sid}, m)$ to $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ during the same tick of $\mathcal{G}_{\mathsf{Clock}}$. When



Protocol π_{mdmt}

For each $t \in \{0, \ldots, \mathsf{poly}(\tau)\}$ let $(\Delta^t_{\mathtt{lo}}, \Delta^t_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(t)$. In the protocol two parties $\mathcal{P}_S, \mathcal{P}_R$ interact via functionalities $\mathcal{F}_{\mathsf{dmt}}^{\Delta^t_{\mathtt{lo}}, \Delta^t_{\mathtt{hi}}}$. In addition, they use a global clock $\mathcal{G}_{\mathsf{Clock}}$. Upon any activation that is not related to a message below, parties send $(\mathtt{Update})$ to $\mathcal{G}_{\mathsf{Clock}}$.

Send: Upon input $(\textsc{Send}, \mathsf{sid}, m)$ $\mathcal{P}_S$ acts as follows:
1. Send $(\textsc{Read})$ to $\mathcal{G}_{\mathsf{Clock}}$ and obtain $(\textsc{Read}, \bar{t})$.
2. Determine $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(\bar{t})$.
3. Send $(\textsc{Send}, \mathsf{sid}, (m, \bar{t}))$ to $\mathcal{F}_{\mathsf{dmt}}^{\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}}$ and $(\textsc{Update})$ to $\mathcal{G}_{\mathsf{Clock}}$.

Receive: Upon input $(\textsc{Rec}, \mathsf{sid})$ $\mathcal{P}_R$ acts as follows:
1. Send $(\textsc{Read})$ to $\mathcal{G}_{\mathsf{Clock}}$ and obtain $(\textsc{Read}, \bar{t})$.
2. For each $t \in \{0, \ldots, \bar{t}\}$ compute $(\Delta^t_{\mathtt{lo}}, \Delta^t_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(t)$.
3. Send $(\textsc{Rec}, \mathsf{sid})$ to each $\mathcal{F}_{\mathsf{dmt}}^{\Delta^t_{\mathtt{lo}}, \Delta^t_{\mathtt{hi}}}$ and wait for responses $(\textsc{Sent}, \mathsf{sid}, (m, t'))$ from $\mathcal{F}_{\mathsf{dmt}}^{\Delta^t_{\mathtt{lo}}, \Delta^t_{\mathtt{hi}}}$. If $t \neq t'$ then $\mathcal{P}_R$ ignores (m, t').
4. If $\Delta_{\mathtt{lo}} + t \leq \bar{t} \leq \Delta_{\mathtt{hi}} + t$ then $\mathcal{P}_R$ outputs $(\textsc{Sent}, \mathsf{sid}, m, t)$.

Fig. 20: Protocol π_{mdmt} for authenticated message transmission with evolving delay bounds.

the adversary makes this $\mathcal{F}_{\mathsf{dmt}}^{\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}}$ output the message (m, t'), then $\mathcal{S}$ makes $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ output m in the same tick round of $\mathcal{G}_{\mathsf{Clock}}$ by sending $(\textsc{ok}, \mathsf{sid}, t')$. This simulation is perfect, as $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ will output any message in the same round where the respective instance of $\mathcal{F}_{\mathsf{dmt}}^{\cdot}$ would have released it to an honest receiver. Moreover, only those messages are forwarded by $\mathcal{S}$ to $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ that wouldn't be ignored by an honest receiver.

If $\mathcal{P}_R$ is corrupted then $\mathcal{S}$ sends $(\textsc{Rec}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ in every tick round. Upon obtaining $(\textsc{Sent}, \mathsf{sid}, m, t')$ from $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ in tick round t, $\mathcal{S}$ computes $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(t')$ and programs the respective instance $\mathcal{F}_{\mathsf{dmt}}^{\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}}}$ to contain the message (m, t') and have $\mathtt{msg} = \mathtt{released} = \top$ so that the honest receiver can pick up the message. Again, the simulation is perfect because the instance that is reprogrammed by $\mathcal{S}$ is the one an honest sender would provide the respective input to. Moreover, given the construction of $\mathcal{F}_{\mathsf{dmt}}^{\cdot}$ the dishonest receiver would not be able to obtain the message any earlier than in this round in the real protocol. □

B.2 Proof of Theorem 4.1

Proof. We construct a PPT simulator $\mathcal{S}$ that emulates the protocol interaction for a corrupted $\mathcal{P}_S$ or $\mathcal{P}_R$. $\mathcal{S}$ will simulate the instances of $\mathcal{F}_{\mathsf{Sig}}, \mathcal{F}_{\mathsf{Reg}}, \mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$. During **Setup**, $\mathcal{S}$ will in either case of corruption act like an honest party, setting up both instances ${\mathcal{F}_{\mathsf{Sig}}}^S, {\mathcal{F}_{\mathsf{Sig}}}^R$ and will simulate posting its key on $\mathcal{F}_{\mathsf{Reg}}$.

If $\mathcal{P}_S$ is corrupted, then $\mathcal{S}$ during **Send** extracts the message m from $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ and checks that $\mathcal{P}_S$ has a key $\mathsf{SIG}.vk_S$ registered with $\mathcal{F}_{\mathsf{Reg}}$ before the current tick round. If the signature verifies with ${\mathcal{F}_{\mathsf{Sig}}}^S$, then forward it to $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ as $(\textsc{Send},$

$\mathsf{sid}, m)$ in the same tick round. When the adversary makes $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ output the message and if an honest verifier would have accepted it, then $\mathcal{S}$ computes $\pi_{\mathtt{lo}}$ as in the protocol using ${\mathcal{F}_{\mathsf{Sig}}}^R$ for its signature. Finally, it lets $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ deliver the message to the honest receiver and sends $(\textsc{Proof}, \mathsf{sid}, m, t, \pi_{\mathtt{lo}})$ to $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$. If the timestamp in $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ does not coincide with when the message was sent, it instead lets $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ deliver the message and sends $(\textsc{NoProof}, \mathsf{sid})$. For any message $(\textsc{verify}, \mathsf{sid}, m, t, \varDelta, \pi_{\mathtt{lo}})$ where $\mathcal{S}$ did generate $\pi_{\mathtt{lo}}$ for this m, t and $(\varDelta_{\mathtt{lo}}, \varDelta_{\mathtt{hi}}) \leftarrow \mathsf{f}_\Delta(t), \varDelta \in [\varDelta_{\mathtt{lo}}, \varDelta_{\mathtt{hi}}]$ send $(\textsc{verify}, \mathsf{sid}, m, t, \varDelta, \pi_{\mathtt{lo}}, 1)$, otherwise send $(\textsc{verify}, \mathsf{sid}, m, t, \varDelta, \pi_{\mathtt{lo}}, 0)$ to $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$.

If instead $\mathcal{P}_R$ is corrupted, wait until $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ outputs $(\textsc{Sent}, \mathsf{sid}, m, t)$, then create a valid signature σ_S using $\mathcal{F}_{\mathsf{Sig}}$ and make $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ output $(\textsc{Sent}, \mathsf{sid}, (m, t, \sigma_S), t)$ to $\mathcal{P}_R$ in the same tick round. In addition, send $(\textsc{NoProof}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$. Then, upon query $(\textsc{Verify}, \mathsf{sid}, m, t, \varDelta, \pi_{\mathtt{lo}})$ from $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ and if $\pi_{\mathtt{lo}}$ can be parsed as (σ'_S, σ_R), check that $\sigma_S = \sigma'_S$. If not, then send $(\textsc{verified}, \mathsf{sid}, m, t, \varDelta, \pi_{\mathtt{lo}}, 0)$ to $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$. Otherwise emulate the call $(\textsc{verify}, \mathsf{sid}, (m, t, \sigma_S), \sigma_R, \mathsf{SIG}.vk_R)$ on ${\mathcal{F}_{\mathsf{Sig}}}^R$ with the adversary, which will ultimately output $(\textsc{verified}, \mathsf{sid}, (m, t, \sigma_S), f)$ to $\mathcal{S}$. Send $(\textsc{verified}, \mathsf{sid}, m, t, \varDelta, \pi_{\mathtt{lo}}, f)$ to $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$.

Clearly, the simulation runs in polynomial time. For a corrupted $\mathcal{P}_S$, we only make $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ output a proof (and let it later verify a proof positively) if the message from $\mathcal{P}_S$ via $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$ was well-formed. This is identical to the protocol, and also the proof $\pi_{\mathtt{lo}}$ is identical. For the corrupt $\mathcal{P}_R$ we make the simulated protocol output the correctly signed message to it in the same round as it would in the real protocol. Moreover, $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$'s **Verify** responses are consistent with the outputs from the protocol by letting $\mathcal{S}$ verify signatures with $\mathcal{F}_{\mathsf{Sig}}$ first. Hence both cases are perfectly indistinguishable. □

B.3 Computing channel delays

We now define the algorithm $\mathsf{delays}(t_1, \mathsf{f}_{\varDelta,1}, \ldots, \mathsf{f}_{\varDelta,n-1}, k)$ that works for any threshold $k < n$ of corrupted parties to determine the minimal and maximal observable delay as follows:

1. For $\imath \in [n-1]$ let $\varDelta_{\mathtt{hi}}^i = \max_{j \in \mathsf{poly}(\tau)} \{\varDelta_{\mathtt{hi}} \mid (\varDelta_{\mathtt{lo}}, \varDelta_{\mathtt{hi}}) \leftarrow \mathsf{f}_{\varDelta,i}(j)\}$. Then
$$\varDelta_{\mathtt{hi}} = \max_{j \in [\varDelta_{\mathtt{hi}}^1 + \cdots + \varDelta_{\mathtt{hi}}^{n-1}]} \{j \mid \mathsf{isP}(t_1, \mathsf{f}_{\varDelta,1}, \ldots, \mathsf{f}_{\varDelta,n-1}, t_1 + j)\}$$
2. First party honest:
$$a_1 = \min_{t_1 \leq t \leq t_1 + \varDelta_{\mathtt{hi}}} \{t - t_1 \mid \mathsf{isP}(t_1, \mathsf{f}_{\varDelta,1}, \ldots, \mathsf{f}_{\varDelta,n-k}, t_1 + t)\}$$
3. Last party honest:
$$a_2 = \min_{t_1 \leq t < t_n \leq t_1 + \varDelta_{\mathtt{hi}}} \left\{ t_n - t \;\middle|\; \begin{matrix} \mathsf{isP}(t_1, \mathsf{f}_{\varDelta,1}, \ldots, \mathsf{f}_{\varDelta,k}, t) \wedge \\ \mathsf{isP}(t, \mathsf{f}_{\varDelta,k+1}, \ldots, \mathsf{f}_{\varDelta,n-1}, t_n) \end{matrix} \right\}$$

4. First and last two corrupt:

$$a_3 = \min_{i\in\{2,\dots,k-2\}, t_1\le t<t'\le t_1+\Delta_{\mathtt{hi}}} \left\{ t'-t \;\middle|\; \begin{array}{c} \mathsf{isP}(t_1, \mathsf{f}_{\Delta,1},\dots,\mathsf{f}_{\Delta,i}, t)\wedge \\ \mathsf{isP}(t, \mathsf{f}_{\Delta,i+1},\dots,\mathsf{f}_{\Delta,i+n-k}, t')\wedge \\ \mathsf{isP}(t', \mathsf{f}_{\Delta,i+n-k+1},\dots,\mathsf{f}_{\Delta,n-1}, t_n) \end{array} \right\}$$

5. Set $\Delta_{\mathtt{lo}} = \min\{a_1, a_2, a_3\}$ and output $(\Delta_{\mathtt{lo}}, \Delta_{\mathtt{hi}})$.

Clearly, each step of delays makes only polynomially many calls to isP, so the algorithm remains efficient for $n = \mathsf{poly}(\log \tau)$.

Proposition B.2. *The algorithm* delays *computes the minimal and maximal observable delay for* k *corruptions of* n *parties given delay functions* $\mathsf{f}_{\Delta,1},\dots,\mathsf{f}_{\Delta,n-1}$.

Proof. Clearly, $\Delta_{\mathtt{hi}}$ cannot be larger than the sum of the largest individual delays that any $\mathsf{f}_{\Delta,i}$ can contribute. Hence, $\Delta_{\mathtt{hi}}$ as computed is the largest achievable delay in any observable protocol.

a_1 considers the case where the first $n-k$ parties are honest. That the given statement finds the smallest possible delay in this case follows directly.

Step a_2 considers the case where the last $n-k$ parties are honest. Here, since $\mathcal{P}_{k+1}$ can observe the behavior of $\mathcal{P}_k$ (which is dishonest), the minimal delay includes the delay from $\mathcal{P}_k$ to $\mathcal{P}_{k+1}$.

Finally, step a_3 considers all cases where there are two parties in the beginning and the end of the chain that are corrupted, and picks the best way of having i corrupted in the beginning and $k-i$ in the end so that the honest parties have minimal observable delay. Then, the minimal of all these 3 mutually exclusive cases yields the minimal channel delay. □

B.4 Proof of Theorem 4.4

Proof. We construct a simulator $\mathcal{S}$ that works for every set of corrupted parties. Let $\mathcal{P}_i$ be the first honest party and $\mathcal{P}_j$ be the last honest party (where $\mathcal{P}_i = \mathcal{P}_j$ is possible). In general, $\mathcal{S}$ will run a simulation of $\pi_{\mathsf{Multi-SCD}}$ with the adversary where it lets every uncorrupted $\mathcal{P}_i$ act honestly, subject to the modifications outlined below.

If $\mathcal{P}_1$ is honest then $\mathcal{S}$ already initially obtains m from $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ and honestly generates messages and signatures for $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,i}}$ where an honest party is a sender. If $\mathcal{P}_1$ is corrupted then wait until the first honest party $\mathcal{P}_i$ obtains the first valid message m, t from $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,i-1}}$. If an honest $\mathcal{P}_i$ would sign and forward the message, then send $(\textsc{Send}, \mathsf{sid}, m, t)$ to $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ and continue to simulate the protocol honestly.

Continue simulation for each honest intermediate party until the last honest party $\mathcal{P}_j$. If $\mathcal{P}_j = \mathcal{P}_n$ then $\mathcal{S}$ makes message delivery of $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,n-1}}$ coincide with output delivery in $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ by using **Release message** and chooses the proof string according to all signatures as in the protocol. If some signatures are not valid or delivery appears too late at the simulated $\mathcal{P}_n$ or any honest intermediate receiver

then $\mathcal{S}$ makes $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ output $(\textsc{NoProof}, \mathsf{sid})$. Finally, reject all **Verify** queries in case $(\textsc{NoProof}, \mathsf{sid})$ was sent and accept only those for the chosen proof string otherwise. If $\mathcal{P}_j \neq \mathcal{P}_n$, let all honest parties act like in the protocol. For each query of **Verify**, reject if the proof string disagrees with the honestly generated signatures for the specific message and delay. For all signatures of adversarially controlled parties $\mathcal{P}_i$, check with $\mathcal{F}_{\mathsf{Sig}}{}^i$ if they are valid for m, t and only set $\phi = 1$ iff all are valid.

The messages that the adversary obtains in the protocol are perfectly indistinguishable from those in the simulation. Moreover, the output of **Verify** both in the simulation and in the protocol coincides. If the receiver is honest, then delivery of message and output is simultaneous with what happens in the protocol by $\mathcal{S}$ using the **Release message** interface. Moreover the message and its timestamp are consistent with the simulation, and exactly those get delivered to an honest receiver that don't make the protocol abort. If a protocol instead fails, then $\mathcal{S}$ uses $(\textsc{NoProof}, \mathsf{sid})$ to let $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ abort. All **Verify** responses of $\mathcal{S}$ are consistent with what an honest verifier would output in the protocol. ◻

C Proof of Theorem 7.1

Proof. We prove Theorem 7.1 by constructing a simulator $\mathcal{S}$ (presented in Figure 21) that executes an internal copy of $\mathcal{A}$ and interacts with $\mathcal{F}_{\mathsf{tlp}}$ in an ideal world execution that is indistinguishable for the environment $\mathcal{Z}$ from the real world execution of $\pi_{\mathsf{TLP-Light}}$ with $\mathcal{A}$. The core tasks of $\mathcal{S}$ are making sure that every puzzle generated by $\mathcal{A}$ in the simulation is created at $\mathcal{F}_{\mathsf{tlp}}$ and that every puzzle that is solved by $\mathcal{F}_{\mathsf{tlp}}$ in the ideal world is simulated towards $\mathcal{A}$. The first task is accomplished by $\mathcal{S}$ by extracting the message m and proof π from every puzzle generated by $\mathcal{A}$ and creating a TLP containing m by contacting $\mathcal{F}_{\mathsf{tlp}}$. The second task is achieved by simulating an execution of $\pi_{\mathsf{TLP-Light}}$ for solving TLPs provided by $\mathcal{F}_{\mathsf{tlp}}$ and later using the leakage of m, π from $\mathcal{F}_{\mathsf{tlp}}$ to program the restricted programmable random oracles such that the output of the protocol matches m, π. Both simulation strategies are clearly possible and indistinguishable from a real execution since $\mathcal{S}$ has the shared secret key sk provided by $\mathcal{F}_{\mathsf{DKG}}$ (which is simulated) and since it can rely on the properties of the IND-CCA secure (and thus UC-secure) encryption scheme in Definition A.4, which is used to generate ciphertexts containing TLP messages in $\pi_{\mathsf{TLP-Light}}$. ◻

D UC Treatment of Delay Encryption

The notion of Delay Encryption (DE) was introduced in [22], where a game based security definition is presented. In order to use our proof of sequential communication delay machinery, we first introduce a treatment of DE in the UC framework, upon which we have defined and constructed our results. In Figure 22, We provide an ideal functionality $\mathcal{F}_{\mathsf{DE}}$ for DE that captures this notion.

Simulator $\mathcal{S}$ for $\pi_{\mathsf{TLP-Light}}$

$\mathcal{S}$ interacts with an internal copy of $\mathcal{A}$, towards which it simulates the honest parties in $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$ and functionalities $\mathcal{G}_{\mathsf{ticker}}, \mathcal{G}_{\mathsf{Clock}}, \mathcal{G}_{\mathsf{rpoRO}}{}^1, \mathcal{G}_{\mathsf{rpoRO}}{}^2, \mathcal{F}_{\mathsf{DKG}}, \mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,1}}, \dots, \mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,|\mathcal{W}|-1}}$. Unless explicitly stated, $\mathcal{S}$ simulates all functionalities exactly as they are described.

Setup: $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{DKG}}$ towards $\mathcal{A}$ and honest parties in $\mathcal{P}$ interacting with $\mathcal{F}_{\mathsf{DKG}}$, learning all sk_j and $\mathsf{sk} = \sum_{\mathcal{P}_j \in \mathcal{W}} \mathsf{sk}_j$.

Create puzzle: When $\mathcal{A}$ outputs $\mathsf{puz} = (c_1, c_2, c_3)$, $\mathcal{S}$ proceeds as follows:

1. Extract the message m and proof $\pi = (\mathsf{pk}, r, s)$: (a) Extract message m by computing $r = \tilde{c}_2 = c_2 \cdot c_1^{\mathsf{sk}}$, sending $(\textsc{Hash-Query}, r)$ to $\mathcal{G}_{\mathsf{rpoRO}}{}^1$, receiving $(\textsc{Hash-Confirm}, \mathsf{pad})$ and computing $m|s = c_3 \oplus \mathsf{pad}$. (b) Check that the puzzle is valid by sending $(\textsc{Hash-Query}, m|s)$ to $\mathcal{G}_{\mathsf{rpoRO}}{}^1$, receiving $(\textsc{Hash-Confirm}, \rho)$ and checking that $\mathsf{puz} = (c_1 = g^\rho, c_2 = r \cdot \mathsf{pk}^\rho, c_3 = (m|s) \oplus \mathsf{pad})$. (c) Send $(\textsc{IsProgrammed}, m|s)$ (resp. $(\textsc{IsProgrammed}, r)$) to $\mathcal{G}_{\mathsf{rpoRO}}{}^1$ (resp. $\mathcal{G}_{\mathsf{rpoRO}}{}^2$) and abort if either of the responses is $(\textsc{IsProgrammed}, 1)$.
2. If all checks on m, π passed, send $(\mathsf{CreatePuzzle}, \mathsf{sid}, m)$ to $\mathcal{F}_{\mathsf{tlp}}$ and provide puz, π when requested.

Solve: Simulate honest parties in $\mathcal{P}$ executing as in $\pi_{\mathsf{TLP-Light}}$. Upon receiving $(\mathsf{Solve}, \mathsf{sid}, \mathsf{puz})$ from $\mathcal{F}_{\mathsf{tlp}}$, $\mathcal{S}$ forwards $(\mathsf{Solve}, \mathsf{sid}, \mathsf{puz})$ to the first $\mathcal{P}_i \in \mathcal{W}$.

Public Verification: Simulate honest parties in $\mathcal{P}$ executing as in $\pi_{\mathsf{TLP-Light}}$.

Tick: $\mathcal{S}$ simulates honest parties in $\mathcal{W}$ executing as in $\pi_{\mathsf{TLP-Light}}$, additionally performing the following steps:

Starting Solution: When a corrupted party in $\mathcal{P}$ sends $(\mathsf{Solve}, \mathsf{sid}, \mathsf{puz})$ to the first $\mathcal{P}_i \in \mathcal{W}$, $\mathcal{S}$ forwards $(\mathsf{Solve}, \mathsf{sid}, \mathsf{puz})$ to $\mathcal{F}_{\mathsf{tlp}}$.

Ongoing Solution: $\mathcal{S}$ answers requests from $\mathcal{F}_{\mathsf{tlp}}$ as follows:

- Upon receiving $(\mathsf{Solved}, \mathsf{sid}, \mathsf{puz}, m, \pi)$ from $\mathcal{F}_{\mathsf{tlp}}$, $\mathcal{S}$ programs $\mathcal{G}_{\mathsf{rpoRO}}{}^1$ and $\mathcal{G}_{\mathsf{rpoRO}}{}^2$ such that solving puz via the steps of $\pi_{\mathsf{TLP-Light}}$ yields message m with proof π.
- Upon receiving a request from $\mathcal{F}_{\mathsf{tlp}}$ for π for a $\mathsf{puz} = (c_1, c_2, c_3)$, $\mathcal{S}$ answers with $\pi = (\mathsf{pk}, r, s)$ obtained by computing $r = \tilde{c}_2 = c_2 \cdot c_1^{\mathsf{sk}}$, sending $(\textsc{Hash-Query}, r)$ to $\mathcal{G}_{\mathsf{rpoRO}}{}^1$, receiving $(\textsc{Hash-Confirm}, \mathsf{pad})$ and computing $m|s = c_3 \oplus \mathsf{pad}$.

Fig. 21: Simulator $\mathcal{S}$ for $\pi_{\mathsf{TLP-Light}}$.

Similarly to other timed functionalities in our work, this functionality is defined in the abstract composable time model of TARDIS [9], previously discussed in Section 2 and Appendix A.2. We essentially adapt our PV-TLP functionality $\mathcal{F}_{\mathsf{tlp}}$ to generate a DE ciphertext as if it was a time-lock puzzle connected to a certain ID represented by a sub-session ID ssid. Analogously, we modify the puzzle solving interface to instead implicitly extract the secret key corresponding to a ssid, which in the functionality is reflected by allowing honest parties to obtain the messages in ciphertexts corresponding to that ssid. As is the case in $\mathcal{F}_{\mathsf{tlp}}$ and $\mathcal{F}_{\mathsf{VDF}}$, we allow the adversary to decrypt ciphertexts connected to a given ssid slightly before the same access is given to honest parties (*i.e.* at time $\epsilon\Gamma < \Gamma$).

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

$\mathcal{F}_{\mathsf{DE}}$ is parameterized by a computational security parameter τ, a message space $\mathcal{MSG}$, a tag space $\mathcal{TAG}$, a slack parameter $0 < \epsilon \leq 1$ and a delay parameter Γ. $\mathcal{F}_{\mathsf{DE}}$ interacts with a set of parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and an adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{DE}}$ maintains initially empty lists omsg (encrypted messages), L (keys being extracted), EXT (extracted keys).

Encrypt Message: Upon receiving a message $(\mathsf{CreatePuzzle}, \mathsf{sid}, \mathsf{ssid}, m)$ from $\mathcal{P}_i$ where $m \in \mathcal{MSG}$, send $(\mathsf{CreatePuzzle}, \mathsf{sid}, \mathsf{ssid})$ to $\mathcal{S}$ and let $\mathcal{S}$ provide puz. If $\mathsf{puz} \notin \mathcal{TAG}$ or there exists $(\mathsf{ssid}, \mathsf{puz}, m') \in \mathsf{omsg}$, then $\mathcal{F}_{\mathsf{DE}}$ halts. Append $(\mathsf{ssid}, \mathsf{puz}, m)$ to omsg, set and output $(\mathsf{Encrypt}, \mathsf{sid}, \mathsf{ssid}, \mathsf{puz})$ to $\mathcal{P}_i$ and to $\mathcal{S}$.

Extract Key: Upon receiving $(\mathsf{Extract}, \mathsf{sid}, \mathsf{ssid})$ from $\mathcal{P}_i \in \mathcal{P}$, add $(\mathsf{ssid}, 0)$ to L and send $(\mathsf{Extract}, \mathsf{sid}, \mathsf{ssid})$ to $\mathcal{S}$.

Decrypt Ciphertext: Upon receiving $(\mathsf{Decrypt}, \mathsf{sid}, \mathsf{ssid}, \mathsf{puz})$ from a party $\mathcal{P}_i \in \mathcal{P}$, ignore the message if $\mathcal{P}_i$ is honest and there does not exist a record $\mathsf{ssid} \in \mathsf{EXT}$ or if $\mathcal{P}_i$ is corrupted and there does not exist a record $(\mathsf{ssid}, c) \in L$ for $c \geq \epsilon\Gamma$. Otherwise, proceed as follows:

- If $(\mathsf{ssid}, \mathsf{puz}, m) \in \mathsf{omsg}$, output $(\mathsf{Decrypt}, \mathsf{sid}, \mathsf{ssid}, \mathsf{puz}, m)$ to $\mathcal{P}_i$.
- If there does not exist $(\mathsf{ssid}, \mathsf{puz}, m) \in \mathsf{omsg}$, let $\mathcal{S}$ provide $m \in \mathcal{MSG}$, add $(\mathsf{ssid}, \mathsf{puz}, m)$ to omsg and output $(\mathsf{Decrypt}, \mathsf{sid}, \mathsf{ssid}, \mathsf{puz}, m)$ to $\mathcal{P}_i$.

Tick: For all $(\mathsf{ssid}, c) \in L$, update $(\mathsf{ssid}, c) \in L$ by setting $c = c + 1$ and:

- If $c \geq \epsilon\Gamma$, send $(\mathsf{Extracted}, \mathsf{sid}, \mathsf{ssid})$ to $\mathcal{S}$.
- If $c = \Gamma$, remove $(\mathsf{ssid}, c) \in L$, send $(\mathsf{Proceed?}, \mathsf{sid}, \mathsf{ssid})$ to $\mathcal{S}$ and proceed as follows:
  - If $\mathcal{S}$ sends $(\textsc{Abort}, \mathsf{sid}, \mathsf{ssid})$, output $(\mathsf{Abort}, \mathsf{sid}, \mathsf{ssid})$ to all $\mathcal{P}_i$.
  - If $\mathcal{S}$ sends $(\textsc{Proceed}, \mathsf{sid}, \mathsf{ssid})$, add ssid to EXT and output $(\mathsf{Extracted}, \mathsf{sid}, \mathsf{ssid})$ to all $\mathcal{P}_i$.

Fig. 22: Ticked Functionality $\mathcal{F}_{\mathsf{DE}}$ for Delay Encryption.

E More on OMS

E.1 Preliminaries for our OMS construction

Correctness of OMS We formally define correctness for a OMS scheme. By inspection, our construction in Fig 8 satisfies correctness with negligible error, where an error occurs if aggregated R_j or $\tilde{R}_j$ which are uniform in $\mathbb{G}$ happens to be $1_{\mathbb{G}}$ (L. 8).

Definition E.1 (OMS Correctness). *An ordered multi signature* OMS *is said to be correct if* $\forall \tau, n > 0$ *and for any message* m *it holds that:*

$$\Pr[1 \leftarrow \mathsf{OMS}.\mathbf{Correctness}(\tau, n, m)] = 1 - \mathsf{negl}(\tau)$$

where the $\mathsf{OMS}.\mathbf{Correctness}$ *experiment is defined in Fig. 23.*

Forking Lemma We restate a variant of the general forking lemma from [58]. It differs from the widely used version of [11] in that an algorithm rewound by the forking algorithm Fork takes additional side inputs v_j's.

$\mathsf{OMS}.\mathbf{Correctness}\ (\tau, n, m)$	
1: $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\tau)$	7: $\mathsf{off}_i \leftarrow \mathsf{SignOff}_{\mathtt{st}_i}(\mathsf{sk}_i, L)$
2: **for** $i \in [1, n]$ **do**	8: $\mathsf{offs} := (\mathsf{off}_1, \ldots, \mathsf{off}_n)$
3: $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}()$	9: **for** $i \in [1, n]$ **do**
4: $\mathtt{st}_i := \varepsilon$	10: $\sigma_i \leftarrow \mathsf{SignOn}_{\mathtt{st}_i}(\mathsf{sk}_i, m, L, \mathsf{offs}, \sigma_{i-1})$
5: $L = (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$	11: $\sigma = \sigma_n$
6: **for** $i \in [1, n]$ **do**	12: **return** $\mathsf{Vrfy}(L, m, \sigma)$

Fig. 23: Correctness for ordered multi signatures

Lemma E.2 (General Forking Lemma with side inputs [58]). *Fix integers Q and m and sets C and V of size greater than 2. Let IGen be a randomized algorithm that we call the input generator. Let $\mathcal{B}$ be a randomized algorithm that on input inp, $c_1, \ldots, c_Q \in C$ and $v_1, \ldots, v_m \in V$ returns indices $i \in [0, Q]$, $j \in [0, m]$ and a side output out. Let Fork be a forking algorithm that works as in Fig. 24 given inp as input and given black-box access to $\mathcal{B}$. Suppose the following probabilities.*

$$\mathsf{acc} := \Pr\left[i \geq 1 \ :\ \begin{array}{c} \mathsf{inp} \leftarrow \mathsf{IGen}(1^\tau); c_1, \ldots, c_Q \xleftarrow{\$} C; v_1, \ldots, v_m \xleftarrow{\$} V; \\ (i, j, \mathsf{out}) \leftarrow \mathcal{B}(\mathsf{inp}, (c_1, \ldots, c_Q), (v_1, \ldots, v_m)) \end{array}\right]$$

$$\mathsf{frk} := \Pr\left[b = 1 \ :\ \begin{array}{c} \mathsf{inp} \leftarrow \mathsf{IGen}(1^\tau); v_1, \widehat{v}_1 \ldots, v_m, \widehat{v}_m \xleftarrow{\$} V; \\ (b, \mathsf{out}, \widehat{\mathsf{out}}) \leftarrow \mathsf{Fork}(\mathsf{inp}, v_1, \widehat{v}_1, \ldots, v_m, \widehat{v}_m) \end{array}\right]$$

Then

$$\mathsf{frk} \geq \mathsf{acc} \cdot \left(\frac{\mathsf{acc}}{Q} - \frac{1}{|C|}\right).$$

Alternatively,

$$\mathsf{acc} \leq \frac{Q}{|C|} + \sqrt{Q \cdot \mathsf{frk}}.$$

$\mathsf{Fork}(\mathsf{inp}, v_1, \widehat{v}_1, \ldots, v_m, \widehat{v}_m)$
1: $\rho \leftarrow \{0,1\}^*$
2: $c_1, \ldots, c_Q \leftarrow C$
3: $(i, j, \mathsf{out}) \leftarrow \mathcal{B}(\mathsf{inp}, (c_1, \ldots, c_Q), (v_1, \ldots, v_m); \rho)$
4: **if** $i = 0$ **then return** $(0, \bot, \bot)$
5: $\widehat{c}_i, \ldots, \widehat{c}_Q \leftarrow C$
6: $(\widehat{i}, \widehat{j}, \widehat{\mathsf{out}}) \leftarrow \mathcal{B}(\mathsf{inp}, (c_1, \ldots, c_{i-1}, \widehat{c}_i, \ldots, \widehat{c}_Q), (v_1, \ldots, v_j, \widehat{v}_{j+1}, \ldots, \widehat{v}_m); \rho)$
7: **if** $i = \widehat{i} \wedge c_i \neq \widehat{c}_i$ **then return** $(1, \mathsf{out}, \widehat{\mathsf{out}})$
8: **else return** $(0, \bot, \bot)$

Fig. 24: Forking algorithm

One-More Discrete Log Assumption We state the one-more discrete log assumption which our OMS construction inherits from MuSig2 Schnorr-based (unordered) multi-signature [58].

Definition E.3 (OMDL Assumption). *Let $(\mathbb{G}, p, g)$ be group parameters generated by a group generator algorithm $\mathsf{GGen}(1^\tau)$. Let $\mathcal{O}_{ch}$ be an oracle that returns a uniformly random element of $\mathbb{G}$, and $\mathcal{O}_{dl}$ be an oracle that on input $X \in \mathbb{G}$ returns its discrete logarithm $x \in \mathbb{Z}_p$ in base g, i.e., $g^x = X$. The one more discrete log assumption (OMDL) is true if for any PPT adversary $\mathcal{A}$, the probability $\mathbf{Adv}^{\mathsf{OMDL}}_{\mathsf{GGen}}(\mathcal{A})$ that, $\mathcal{A}$ outputs solutions to $q+1$ DLog instances produced by $\mathcal{O}_{ch}$ while having made at most q queries to $\mathcal{O}_{dl}$, is negligible in τ.*

E.2 Proof of Theorem 5.4

Here we first provide a high level overview. The reduction has the same structure (except additional guessing arguments explained below) as the one used in the proof of MuSig2 [58], with three wrappers around $\mathcal{A}$. The inner-most wrapper, $\mathcal{B}$, simulates the OMS security game to $\mathcal{A}$ by embedding a Dlog challenge in one of the honest keys registered by $\mathcal{A}$ and by answering signing queries using queries to the Dlog solver oracle through the outer wrappers. Next there is an algorithm Fork of the forking lemma [11, 58], that passes all Dlog challenges to $\mathcal{B}$ as well as inputs to be used to answer some of $\mathcal{A}$'s oracle queries. The main task of Fork is to rewind $\mathcal{B}$ (and thus $\mathcal{A}$, see Fig. 24) and carefully match inputs to $\mathcal{B}$ with the series of OMDL values obtained by Fork via the final wrapper $\mathcal{C}$, which is playing the OMDL game (see Def. E.3).

Our proof differs from MuSig2 essentially in two ways due to a different security model. Since the OMS scheme additionally guarantees a correct sequence of honest signers unlike usual multi-signatures, we consider a more complex situation where multiple signing oracles co-exist instead of one. The reduction embeds a DLog challenge in one of the honest signing oracles and answers queries to the remaining oracles using self-generated keys. In this way, the reduction still succeeds using standard guessing arguments. While MuSig2 incurs a quartic security loss due to double-forking, our proof does not suffer from it since the security notion of OMS we inherit from [14] is not in the plain-public key model. Outside the current application context, one may want to guarantee the security without the key registration requirement, while withstanding the so-called *rogue-key attacks* [56]. To retain security in the plain-public key model, the present construction should be modified by having each party use distinct Fiat-Shamir challenge insead of single c as in Fig. 26.

Theorem E.4. *The* OMS *scheme of Fig. 8 with $\ell = 2$ is* $\mathsf{OMS\text{-}UF\text{-}CMA}$ *secure (Fig.7) under the OMDL assumption and in the programmable random oracle model. Concretely, for any adversary $\mathcal{A}$ against* $\mathsf{OMS\text{-}UF\text{-}CMA}$ *security that receives at most Q_k honest public keys from the key generation oracle, makes at most Q_h queries to the random oracles, and starts at most Q_s sessions in total with* $\mathsf{OSignOff}, \mathsf{OSignOn}$ *oracles, there exists an adversary $\mathcal{C}$ against* OMDL

assumption such that

$$\mathbf{Adv}^{\mathsf{OMS\text{-}UF\text{-}CMA}}_{\mathsf{OMS}}(\mathcal{A}) \leq \frac{Q(Q+Q_k^2)}{p} + Q_k^2 \cdot \sqrt{Q \cdot \left(\mathbf{Adv}^{\mathsf{OMDL}}_{\mathsf{GGen}}(\mathcal{C}) + \frac{Q^2}{p}\right)}$$

where $Q = Q_s + 2Q_h + 1$.

Proof. We prove that any successful forger $\mathcal{A}$ can be used to break the OMDL assumption. We consider two types of forgeries: (1) $(m^*, \sigma^* = (R^*, z^*), L^*)$ output at the end of the game, and (2) $(m^*, L^*, \mathsf{offs}^*, \sigma^*_{i^*-1})$ sent to the $\mathsf{OSignOn}$ oracle with index i^* (and which triggers the $\mathsf{win} := 1$ flag).

Type-(1) Forgery If the adversary $\mathcal{A}$ causes a Type-(1) forgery, we construct the following reduction to OMDL. Let Q_k be the number of queries to $\mathsf{OKeyReg}$ with $\mathsf{aux} = \varepsilon$ (i.e., maximum number of honest keys). The reduction goes through multiple layers of wrappers below.

- $\mathcal{B}(\mathsf{pk}^*, U_1, \ldots, U_{2Q_s}, c_1, \ldots, c_Q, v_1, \ldots, v_Q)$: A wrapper algorithm that simulates the view of $\mathcal{A}$ without using a secret key belonging to one of the uncorrupted (honest) parties. $\mathcal{B}$ initially picks uniform $t \in [1, Q_k]$. Whenever $\mathcal{A}$ submits a new honest signer key request (uid, ϵ), if this is the t-th query, then it answers with the challenge public key pk^*. Otherwise, it generates a new key pair using KeyGen algorithm and returns the corresponding public key as $\mathsf{OKeyReg}$ would. In this way, $\mathcal{B}$ knows $Q_k - 1$ secret keys of honest parties. $\mathcal{B}$ answers queries to H_{ch}, $\mathsf{H}_{\mathrm{non}}$, $\mathsf{OSignOff}$ and $\mathsf{OSignOn}$ in the natural way (explained below). Upon receiving a valid forgery $(m^*, L^*, \sigma^* = (R^*, z^*))$ at the end of the game, it outputs $(\mathsf{ctr_ch}^*, \mathsf{ctr_non}^*, \mathsf{out})$, which are as described below.
- $\mathsf{Fork}(\mathsf{pk}^*, U_1, \ldots, U_{2Q_s}, v_1, \ldots, v_Q, \widehat{v}_1, \ldots, \widehat{v}_Q)$: A forker algorithm that rewinds $\mathcal{B}$ as in Figure 24.
- $\mathcal{C}$: A OMDL adversary again that runs Fork internally. It initially makes $2Q_s + 1$ queries to $\mathcal{O}_{\mathrm{ch}}$ to obtain DLog instances $(\mathsf{pk}^*, U_1, \ldots, U_{2Q_s})$. After sampling $v_1, \widehat{v}_1, \ldots, v_Q, \widehat{v}_Q \in \mathbb{Z}_p$ uniformly, $\mathcal{C}$ runs $\mathsf{Fork}(\mathsf{inp}, v_1, \widehat{v}_1, \ldots, v_Q, \widehat{v}_Q)$ where $\mathsf{inp} = (\mathsf{pk}^*, U_1, \ldots, U_{2Q_s})$. Then on receiving $(1, \mathsf{out}, \widehat{\mathsf{out}})$ from Fork, it outputs DLog solutions $(\mathsf{sk}^*, u_1, \ldots, u_{2Q_s})$ such that $\mathsf{pk}^* = g^{\mathsf{sk}^*}$ and $U_j = g^{u_j}$ for $j = 1, \ldots, 2Q_s$. Note that $\mathcal{C}$ is allowed to make at most $2Q_s$ queries to $\mathcal{O}_{\mathrm{dl}}$ in order to win the OMDL game.

Description of $\mathcal{B}$ $\mathcal{B}$ perfectly simulates the view of $\mathcal{A}$ in the $\mathsf{OMS\text{-}UF\text{-}CMA}$ game as follows, except at a few abort events highlighted.

- Initialization: $\mathcal{B}$ runs $\mathcal{A}$ on input pp. It initially samples uniform $t \in [1, Q_k]$ and sets $\mathsf{ctr_u} = 0$, $\mathsf{ctr_ch} = 0$ and $\mathsf{ctr_non} = 0$. It also initializes empty key-value lookup tables $\mathrm{HT}_{\mathrm{ch}}$ and $\mathrm{HT}_{\mathrm{non}}$.
- $\mathsf{H}_{\mathrm{non}}$: On receiving a query to $\mathsf{H}_{\mathrm{non}}$ with input $X := (m, (R_{1,1}, R_{1,2}, \ldots, R_{n,1}, R_{n,2}), L)$, if $\mathrm{HT}_{\mathrm{non}}[X]$ is already defined $\mathcal{B}$ returns this value. Otherwise, $\mathcal{B}$ increments $\mathsf{ctr_non}$, and computes $R = \prod_{i=1}^n R_{i,1} \cdot (\prod_{i=1}^n R_{i,2})^{v_{\mathsf{ctr_non}}}$,

using the v_i values in its input. *If* $\prod_{i=1}^{n} R_{i,2} \neq 1_{\mathbb{G}}$ *and* $\mathrm{HT}_{ch}[m, R, L]$ *is already defined,* $\mathcal{B}$ *aborts. Assuming* $v_{\mathsf{ctr_non}}$ *is sampled uniformly from* $\mathbb{Z}_p$*, this happens with probability at most* $1/p$ *for each query and thus overall at most* Q^2/p *by the union bound.* Otherwise, it makes a query to H_{ch} with input (m, R, L), sets $\mathrm{HT}_{\mathrm{non}}[X] := v_{\mathsf{ctr_non}}$, and returns $v_{\mathsf{ctr_non}}$. Here we use the programmability of the random oracle.

- H_{ch}: On receiving a query to H_{ch} with input $X := (m, R, L)$, if $\mathrm{HT}_{\mathrm{ch}}[X]$ is defined $\mathcal{B}$ returns this value. Otherwise, $\mathcal{B}$ increments $\mathsf{ctr_ch}$, sets $\mathrm{HT}_{\mathrm{ch}}[X] := c_{\mathsf{ctr_ch}}$, and returns $c_{\mathsf{ctr_ch}}$. Here we use the programmability of the random oracle.
- $\mathsf{OKeyReg}$: Whenever $\mathcal{A}$ queries $\mathsf{OKeyReg}$ with new *uid* with empty aux, if this is the t-th query with empty aux, $\mathcal{B}$ registers $(1, \mathsf{pk}^*, 0)$ in $\mathcal{K}$, and returns pk^*. Otherwise, it proceeds as the actual $\mathsf{OKeyReg}$ would.
- $\mathsf{OSignOff}$: Whenever $\mathcal{A}$ queries $\mathsf{OSignOff}$ with a new session identifier sid and L with an index i corresponding to one of the honest keys, if $\mathsf{pk}^* = \mathsf{pk}_i \in L$ then $\mathcal{B}$ lets $R_{i,j} := U_{\mathsf{ctr_u}+j}$ for $j = 1, 2$, and then sets $\mathsf{ctr_u} = \mathsf{ctr_u} + 2$. Otherwise, it proceeds as the actual $\mathsf{OSignOn}$ would. $\mathcal{B}$ also updates $\mathcal{T}$ as $\mathsf{OSignOff}$ would and returns $\mathsf{off}_i = (R_{i,1}, R_{i,2})$.
- $\mathsf{OSignOn}$: Whenever $\mathcal{A}$ queries $\mathsf{OSignOn}$ with a valid sid, m, L, $\mathsf{offs} = (R_{1,1}, R_{1,2}, \ldots, R_{n,1}, R_{n,2})$, $\sigma_{i-1} = (R, \tilde{z})$ with an index i corresponding to one of the honest keys, $\mathcal{B}$ runs all the sanity checks performed by $\mathsf{OSignOn}$, and executes the operations of SignOn, except at line 19 where $\mathcal{B}$'s behavior differs depending on the type of the key:
  - If $\mathsf{pk}_i = \mathsf{pk}^*$, $\mathcal{B}$ makes a query to $\mathcal{O}_{\mathrm{dl}}$ with input $R_{i,1} \cdot R_{i,2}^{v} \cdot \mathsf{pk}_i^{c}$ to obtain its discrete logarithm: z_i.
  - Else, $\mathcal{B}$ computes z_i using the knowledge of sk_i and $r_{i,1}, r_{i,2}$ as the actual SignOn would.

  Whenever any of the checks performed by $\mathsf{OSignOn}$ fails, $\mathcal{B}$ also returns $\perp$ to $\mathcal{A}$ as $\mathsf{OSignOn}$ would.
- When $\mathcal{A}$ outputs a forgery tuple $(m^*, L^*, \sigma^* = (R^*, z^*))$, $\mathcal{B}$ returns $(0, \perp, \perp)$ if any of the validity checks in the $\mathsf{OMS\text{-}UF\text{-}CMA}$ game fails. *Moreover, if* $\mathsf{pk}^* \notin L^*$ *or* (m^*, L^*) *has been signed by* pk^* *previously,* $\mathcal{B}$ *aborts. Since the challenge* pk^* *is embedded in one of* Q_k *honest keys uniformly at random, the probability that* $\mathcal{B}$ *does* ***not*** *abort here is at least* $1/Q_k$. Otherwise, let $\mathsf{ctr_ch}^*$ and $\mathsf{ctr_non}^*$ be the values of $\mathsf{ctr_ch}$ and $\mathsf{ctr_non}$ at the moment $\mathrm{HT}_{\mathrm{ch}}[m^*, R^*, L^*]$ is defined, respectively. That is, $\mathrm{HT}_{\mathrm{ch}}[m^*, R^*, L^*] = c_{\mathsf{ctr_ch}^*}$ and $v_1, \ldots, v_{\mathsf{ctr_non}^*}$ have been used by the time $\mathrm{HT}_{\mathrm{ch}}[m^*, R^*, L^*]$ is set. Moreover let $j^* \in [n]$ be the index such that $\mathsf{pk}^* = \mathsf{pk}_{j^*} \in L$ and K^* be the list of secret keys corresponding to other public keys $L^* \setminus \{\mathsf{pk}_{j^*}\}$. Finally, $\mathcal{B}$ outputs $(\mathsf{ctr_ch}^*, \mathsf{ctr_non}^*, \mathsf{out} = (\mathsf{ctr_ch}^*, \mathsf{ctr_non}^*, j^*, L^*, K^*, R^*, c^*, z^*)$.

Description of $\mathcal{C}$

- $\mathcal{C}$ runs Fork on its input and uniformly sampled $v_1, \ldots, v_Q, \widehat{v}_1, \ldots, \widehat{v}_Q \in \mathbb{Z}_q$. Whenever $\mathcal{B}$ asks for a query to $\mathcal{O}_{\mathrm{dl}}$ through Fork, $\mathcal{C}$ forwards that query and the response accordingly, but in case the second run of $\mathcal{B}$ makes a query

identical to one of the previous queries, $\mathcal{C}$ uses a cached response from the previous run instead of redundantly querying $\mathcal{O}_{\mathrm{dl}}$. After obtaining $(b, \mathsf{out}, \widehat{\mathsf{out}})$ from Fork, if $b = 0$ $\mathcal{C}$ aborts. If $b = 1$, then $\mathcal{C}$ parses

$$\mathsf{out} = (\mathsf{ctr_ch}^*, \mathsf{ctr_non}^*, (j^*, L^*, K^*, R^*, c^*, z^*))$$

and

$$\widehat{\mathsf{out}} = (\mathsf{ctr_ch}^*, \mathsf{ctr_non}^*, (j^*, L^*, K^*, R^*, \widehat{c}^*, \widehat{z}^*)),$$

from Fork (where $\mathsf{ctr_ch}^*, \mathsf{ctr_non}^*, j^*, L^*, K^*, R^*$ from two outputs are guaranteed to be identical thanks to the way $\mathcal{B}$ simulates the view and the successful run of Fork). Due to the verification condition, if $b = 1$ we have that

$$g^{z^*} = R^* \cdot \left(\prod_{i=1}^{n} \mathsf{pk}_i\right)^{c_{\mathsf{ctr_ch}^*}}$$
$$g^{\widehat{z}^*} = R^* \cdot \left(\prod_{i=1}^{n} \mathsf{pk}_i\right)^{\widehat{c}_{\mathsf{ctr_ch}^*}} .$$

– $\mathcal{C}$ extracts the secret key for the challenge public key pk^* as follows.

$$\mathsf{sk}^* = \left((z^* - \widehat{z}^*) \cdot (c_{\mathsf{ctr_ch}^*} - \widehat{c}_{\mathsf{ctr_ch}^*})^{-1} - \sum_{i=1, i \neq j^*}^{n} \mathsf{sk}_i\right)$$

where co-signers' secret keys sk_i for $i \neq j^*$ are indeed known thanks to the key registration requirement.

– Now that $\mathcal{C}$ knows DLog of pk, what's left is computing DLogs of $(U_1, \ldots, U_{2Q_s})$. For each $\mathsf{ctr_u} \in [0, Q_s - 2]$, consider the response z_i computed inside $\mathsf{OSignOn}$ by consuming $U_{\mathsf{ctr_u}+1}, U_{\mathsf{ctr_u}+2}$ and querying $\mathcal{O}_{\mathrm{dl}}$ during the first run of $\mathcal{B}$. For some $\mathsf{ctr_ch}$, $\mathsf{ctr_non}$, z_i satisfies

$$z_i = r_{i,1} + v_{\mathsf{ctr_non}} \cdot r_{i,2} + c_{\mathsf{ctr_ch}} \cdot \mathsf{sk}^* \tag{1}$$

where $r_{i,1}$ and $r_{i,2}$ are DLogs of $R_{i,1} = U_{\mathsf{ctr_u}+1}$ and $R_{i,2} = U_{\mathsf{ctr_u}+2}$, and $v_{\mathsf{ctr_non}} = \mathsf{H}_{\mathrm{non}}(m, \mathsf{offs}, L)$. If $\mathsf{ctr_ch} < \mathsf{ctr_ch}^*$ at that moment, then $\mathcal{C}$ hasn't made any extra query to $\mathcal{O}_{\mathrm{dl}}$ during the second run of $\mathcal{B}$. Thus, $\mathcal{C}$ simply queries $\mathcal{O}_{\mathrm{dl}}$ with $R_{i,1}$ to learn $r_{i,1}$ and uses this information to compute $r_{i,2} := dLog(R_{i,2}) = dLog(U_{\mathsf{ctr_u}+2})$. If $\mathsf{ctr_ch} > \mathsf{ctr_ch}^*$ at that moment[9], the response $\widehat{z}_i$ computed in the second run also satisfies

$$\widehat{z}_i = r_{i,1} + \widehat{v}_{\widehat{\mathsf{ctr_non}}} \cdot r_{i,2} + \widehat{c}_{\widehat{\mathsf{ctr_ch}}} \cdot \mathsf{sk}^* \tag{2}$$

where $\widehat{\mathsf{ctr_non}} > \mathsf{ctr_non}^*$ and $\widehat{v}_{\widehat{\mathsf{ctr_non}}} = \mathsf{H}_{\mathrm{non}}(\widehat{m}, \widehat{\mathsf{offs}}, \widehat{L})$. If $v_{\mathsf{ctr_non}} = \widehat{v}_{\widehat{\mathsf{ctr_non}}}$ (which happens with probability at most $1/p$ for each combination of $\mathsf{ctr_non}$

[9] It must be that $\mathsf{ctr_ch} \neq \mathsf{ctr_ch}^*$ if $\mathcal{A}$ creates a successful forgery. Otherwise, $\mathsf{ctr_ch}^* = \mathsf{H}_{\mathrm{ch}}(m^*, R^*, L^*)$ is used while computing the response, contradicting the winning condition that (R^*, L^*) has never been signed by the owner of pk^*.

and $\widehat{\mathsf{ctr_non}}$ and thus overall at most Q^2/p by the union bound), $\mathcal{C}$ aborts. Otherwise, $\mathcal{C}$ can successfully solve the system of two linear equations with two unknowns $r_{i,1}$ and $r_{i,2}$ (recall that at this point $\mathsf{sk}^*, v_{\mathsf{ctr_non}}, \widehat{v}_{\widehat{\mathsf{ctr_non}}}, c_{\mathsf{ctr_ch}}$ and $\widehat{c}_{\widehat{\mathsf{ctr_ch}}}$ are all known to the reduction).

We now invoke the forking lemma (Lemma E.2). By construction we have that

$$\mathsf{acc}(\mathcal{B}) = \left(\mathbf{Adv}^{\mathsf{OMS\text{-}UF\text{-}CMA}}_{\mathsf{OMS}}(\mathcal{A}) - \frac{Q^2}{p}\right)/Q_k$$

by accounting for the abort events happening when queries to $\mathsf{H}_{\mathrm{non}}$ are made and when the forgery is submitted by $\mathcal{A}$. Since $\mathcal{C}$ succeeds in breaking the OMDL assumption as long as Fork outputs two valid outputs *and* it doesn't abort, overall

$$\begin{aligned}\mathbf{Adv}^{\mathsf{OMDL}}(\mathcal{C}) &\geq \mathsf{frk} - Q^2/p \\ &\geq \mathsf{acc}(\mathcal{B}) \cdot \left(\frac{\mathsf{acc}(\mathcal{B})}{Q} - \frac{1}{p}\right) - \frac{Q^2}{p}.\end{aligned}$$

Finally, we remark that the above lower bound is > 0 and can easily be made noticeable since p's size is determined directly by the security parameter.

Type-(2) Forgery If the adversary $\mathcal{A}$ causes a Type-(2) forgery, we construct the following reduction to OMDL. The reduction goes through multiple layers of wrappers analogous to Type-(1) forgery. We will only explain the differences below.

Description of $\mathcal{B}$

- Initialization: $\mathcal{B}$ performs the same initialization operations as before. Additionally, it samples uniform $s \in [1, Q_k] \setminus \{t\}$, indicating one of the honest public keys whose owner is responsible for detecting Type-(2) forgery. We call it a *detector key.*
- $\mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\mathrm{ch}}, \mathsf{OKeyReg}, \mathsf{OSignOff}$: Queries to these oracles are answered as before. Additionally, if $\mathcal{A}$ makes the s-th query with empty aux to $\mathsf{OKeyReg}$, $\mathcal{B}$ remembers an honestly generated key pk' as the detector key.
- $\mathsf{OSignOn}$: Whenever $\mathcal{A}$ queries $\mathsf{OSignOn}$ with an valid sid, m^*, $L^* = (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$, $\mathsf{offs}^* = (R^*_{1,1}, R^*_{1,2}, \ldots, R^*_{n,1}, R^*_{n,2})$, $\sigma_{i^*-1} = (R^*, \tilde{z}^*)$ with an index i^* corresponding to one of the honest keys, $\mathcal{B}$ performs the same operations as before, except that before sending a response to $\mathcal{A}$, it carries out the following checks. $\mathcal{B}$ verifies the so-far aggregation, i.e., check $g^{\tilde{z}^*} = \tilde{R}^* \cdot \prod_{i=1}^{i^*-1} \mathsf{pk}_i^{c^*}$, where $\tilde{R}^* = (\prod_{i=1}^{i^*-1} R^*_{i,1})(\prod_{i=1}^{i^*-1} R^*_{i,2})^{v^*}$, $v^* = \mathsf{H}_{\mathrm{non}}(m^*, \mathsf{offs}^*, L^*)$, and $c^* = \mathsf{H}_{\mathrm{ch}}(L^*, m^*, R^*)$. If the check passes and inputs are well-formed, $\mathcal{B}$ proceeds as follows.
  - If there exists some $j^* < i^*$ such that pk_{j^*} has never signed (m^*, L^*) while the check passes (i.e., the flag win would be set in the $\mathsf{OMS\text{-}UF\text{-}CMA}$

game), then $\mathcal{B}$ proceeds as follows. *First, if* $\mathsf{pk}_{i^*} \neq \mathsf{pk}'$ *then* $\mathcal{B}$ *aborts. Second, if for all* $j < i^*$ $\mathsf{pk}_j \neq \mathsf{pk}^*$ *or* pk^* *has already signed* (m^*, L^*) *then* $\mathcal{B}$ *aborts. Since the challenge* pk^* *is uniformly embedded in one of* Q_k *honest keys and the detector key* pk' *is uniformly embedded in one of the remaining* $Q_k - 1$ *honest keys, respectively, the probability that* $\mathcal{B}$ *does* ***not*** *abort here is at least* $1/Q_k^2$. If $\mathcal{B}$ does **not** abort (i.e., $\mathsf{pk}_{i^*} = \mathsf{pk}'$ **and** $\exists j^* < i^*$ such that $\mathsf{pk}^* = \mathsf{pk}_{j^*} \in L^*$ and pk_{j^*} has never signed (m^*, L^*)), $\mathcal{B}$ sets the following variables. Let $\mathsf{ctr_ch}^*$ and $\mathsf{ctr_non}^*$ be the values of $\mathsf{ctr_ch}$ and $\mathsf{ctr_non}$ at the moment $\mathrm{HT}_{\mathrm{ch}}[m^*, R^*, L^*]$ is defined, respectively. That is, $\mathrm{HT}_{\mathrm{ch}}[m^*, R^*, L^*] = c_{\mathsf{ctr_ch}^*}$ and $v_1, \ldots, v_{\mathsf{ctr_non}^*}$ have been used by the time $\mathrm{HT}_{\mathrm{ch}}[m^*, R^*, L^*]$ is set. Let K^* be the list of secret keys corresponding to the public keys in $L^* \setminus \{\mathsf{pk}_{j^*}\}$. Then $\mathcal{B}$ halts by outputting $(\mathsf{ctr_ch}^*, \mathsf{ctr_non}^*, \mathsf{out})$, where $\mathsf{out} = (i^*, j^*, L^*, K^*, \tilde{R}^*, c^*, \tilde{z}^*)$.
- Otherwise, $\mathcal{B}$ proceeds as in Type-(1) forgery.

Description of $\mathcal{C}$

– $\mathcal{C}$ runs Fork as before. After obtaining $(b, \mathsf{out}, \widehat{\mathsf{out}})$ from Fork, if $b = 0$ $\mathcal{C}$ aborts. If $b = 1$, then $\mathcal{C}$ parses

$$\mathsf{out} = (\mathsf{ctr_ch}^*, \mathsf{ctr_non}^*, (i^*, j^*, L^*, K^*, \tilde{R}^*, c^*, \tilde{z}^*))$$

and

$$\widehat{\mathsf{out}} = (\mathsf{ctr_ch}^*, \mathsf{ctr_non}^*, (i^*, j^*, L^*, K^*, \tilde{R}^*, \widehat{c}^*, \widehat{\tilde{z}}^*)),$$

from Fork.

We argue that $\mathsf{ctr_ch}^*, \mathsf{ctr_non}^*, i^*, j^*, L^*, K^*, \tilde{R}^*$ from two outputs are guaranteed to be identical whenever $b = 1$, $\mathsf{ctr_ch}^*$ is by definition identical due to the success condition of Fork. In that case, $\mathsf{ctr_non}^*$ must also be identical due to the way $\mathcal{B}$ defines it. Since $\mathsf{ctr_ch}^*$ is identical and $\mathcal{B}$'s behavior is identical until the $\mathsf{ctr_ch}^*$-th query is made to H_{ch}, *it is guaranteed* that the corresponding input (m^*, R^*, L^*) to H_{ch} is the *same* in the two executions as in the analysis of Type-(1) forgeries. Since L^* is identical, K^* is also identical. Thanks to the abort condition of $\mathcal{B}$ when handling $\mathsf{OSignOn}$ queries, once L^* is fixed *it is guaranteed* that i^* and j^* are the same in the two executions.

The crucial difference with the analysis of Type-(1) is that the partial aggregation of $R_{i,j}$ for $i = 1, \ldots, i^* - 1$, denoted by $\tilde{R}^*$, is *not* included in the input of H_{ch}. Still, we can make sure that $\tilde{R}^*$ is identical in the two executions. Thanks to the abort event happening within simulation of $\mathsf{H}_{\mathrm{non}}$, *it is guaranteed* that the corresponding $(m^*, \mathsf{offs}^* = (R^*_{1,1}, R^*_{1,2}, \ldots, R^*_{n,1}, R^*_{n,2}), L^*)$ has been queried to $\mathsf{H}_{\mathrm{non}}$ (which then determines the value of v^*) *before* H_{ch} is queried with (m^*, R^*, L^*). Thus, $\tilde{R}^* = (\prod_{i=1}^{i^*-1} R^*_{i,1}) \cdot (\prod_{i=1}^{i^*-1} R^*_{i,2})^{v^*}$ does not change after forking.

Now, due to the verification condition, if $b = 1$ we have that

$$g^{\tilde{z}^*} = \tilde{R}^* \cdot \left(\prod_{i=1}^{i^*-1} \mathsf{pk}_i\right)^{c_{\text{ctr_ch}^*}}$$
$$g^{\widehat{\tilde{z}}^*} = \tilde{R}^* \cdot \left(\prod_{i=1}^{i^*-1} \mathsf{pk}_i\right)^{\widehat{c}_{\text{ctr_ch}^*}} .$$

- $\mathcal{C}$ extracts the secret key for the challenge public key pk^* as follows.

$$\mathsf{sk}^* = \left((z^* - \widehat{z}^*) \cdot (c_{\mathsf{ctr_ch}^*} - \widehat{c}_{\mathsf{ctr_ch}^*})^{-1} - \sum_{i=1, i \neq j^*}^{i^*-1} \mathsf{sk}_i\right)$$

  where co-signers' secret keys sk_i for $i \neq j^*$ are indeed known thanks to the key registration requirement.
- Now that $\mathcal{C}$ knows DLog of pk, what's left is computing DLogs of $(U_1, \ldots, U_{2Q_s})$. This can be done as in Type-(1) forgery with the same additive loss of Q^2/p.

By the forking lemma (Lemma E.2) we have that

$$\mathsf{acc}(\mathcal{B}) = \left(\mathbf{Adv}_{\mathsf{OMS}}^{\mathsf{OMS\text{-}UF\text{-}CMA}}(\mathcal{A}) - \frac{Q^2}{p}\right) / Q_k^2$$

by accounting for the abort events happening when queries to H_{non} are made and when a forged aggregate so-far is submitted to $\mathsf{OSignOn}$. The rest of the analysis is identical to Type-(1) forgery. Rearranging the terms, we obtain the advantage bound of the theorem statement □

E.3 Three OMS Variants

In what follows we describe three flavors of OMSs and show how to tweak the plain construction in Figure 8 to meet different features.

Out-and-Back topologies We present a variation of our OMS construction that runs in a out-and-back fashion (instead of round-Robin) and without broadcast. This is relevant in settings where signers are placed in such a way that the signer in position i can communicate with signers in positions $i + 1 \leq n$ and $i - 1 > 0$, but no loop is possible between signer n and signer 1. $\mathsf{Setup}(1^\tau)$, $\mathsf{KeyGen}()$, and $\mathsf{Vrfy}(L, m, \sigma)$ are the same as in Figure 8. $\mathsf{SignOff}$ is essentially as in Figure 8, except that it takes in input $\mathsf{aux} = (L, \mathsf{off}'_{i-1})$ where $\mathsf{off}'_{i-1} = (\mathsf{off}_j)_{j=1}^{i-1}$ and returns the output off'_i that collects the offline contributions of all singers so-far. Signer in position i passes the list L together off'_i to signer $i + 1 \leq n$, and the output of SignOn to signer $i - 1 > 0$ (out-and-back). Signer in position n generates $\sigma_0 = (1_{\mathbb{G}}, 0)$ and $\mathsf{offs} = (\mathsf{off}_1, \ldots, \mathsf{off}_n)$ and initiates the online phase. SignOn is very similar to the one in Figure 8, the only difference is that the index of aggregated items (computed in lines 4, 7) are starting from n down to i (instead of from 1, up to i). Similarly, line 11 runs for $i = n$.

Determining signers order on-the-fly See Figure 25. Up to now, we considered settings where L, the list of ordered signers, was known to all parties before the two-phase interactive signing procedure. Now we propose an OMS scheme that can work in settings where L is determined on-the-fly during the offline phase. Here we assume the round-Robin communication model, so the signers' order is preserved in the offline and online phases.

Key Aggregation in the Plain Public-Key Model See Figure 26. The construction closely follows MuSig2, except that an aggregated key pk varies depending on the order of keys in the list L and SignOn additionally validates an aggregate so-far as in our basic OMS construction. Essentially, each signer needs to derive joint aggregate public keys by taking the random linear combination of all public keys, where coefficients are derived through the random oracle $\mathsf{H}_{\mathrm{agg}}$. To generalize the security proof of Theorem E.4 to prove security of the present variant in the PPK model, one should set $\ell = 4$ as in [58]. Accordingly, the reduction must invoke the forking lemma twice, first at an aggregation coefficient a_i for honest party's pk and second at challenge c. This will lead to a quartic reduction loss similar to [58], which, howerver, could be circumvent by accepting stronger assumptions such as the algebraic group model [41].

$\mathsf{Setup}(1^\tau), \mathsf{KeyGen}(), \mathsf{SignOn}$ and $\mathsf{Vrfy}(L, m, \sigma)$ are the same as in Figure 8.

$\mathsf{SignOff}_{\mathtt{st}}(\mathsf{sk}, \mathsf{aux} = (L', \mathsf{offs}'))$:

1: $i := |L'| + 1$ ▷ Identify signer's position
2: $\mathtt{st} := (i, L')$ ▷ Store position and so-far signers' chain
3: **for** $j \in [1, \ell]$ **do**
4: $\quad r_{i,j} \xleftarrow{\$} \mathbb{Z}_p$
5: $\quad R_{i,j} := g^{r_{i,j}}$
6: $\quad \mathtt{st}_i := \mathtt{st}_i | r_{i,j} | R_{i,j}$
7: $\mathsf{offs} := \mathsf{offs}' | (R_{i,1}, \ldots, R_{i,\ell})$
8: $L := L' | \mathsf{pk}$
9: **return** (L, offs)

Fig. 25: OMS construction for determining signers' order on-the-fly

KeyGen() and SignOff are the same as in Figure 8. $\mathsf{Vrfy}(\mathsf{pk}, m, \sigma)$ is identical to the usual Schnorr verification algorithm. Setup and SignOn are very similar to the ones in Figure 8, we highlight the lines that differ. KAgg is the additionalkey aggregation algorithm that combines an ordered list of keys into a single pk that looks like a usual Schnorr verification key.

$\mathsf{Setup}(1^\tau)$:

1: $(\mathbb{G}, p, g) \leftarrow \mathsf{GroupGen}(1^\tau)$
2: $(n, \ell) \leftarrow \mathsf{poly}(\tau)$
3: $\mathsf{H}_{\text{non}}, \mathsf{H}_{\text{ch}}, \mathsf{H}_{\text{agg}} : \{0,1\}^* \rightarrow \mathbb{Z}_p^*$
4: $\sigma_0 := (1_{\mathbb{G}}, 0) \in \mathbb{G} \times \mathbb{Z}_p$
5: **return** $\mathsf{pp} := (\mathbb{G}, g, p, n, \ell, \mathsf{H}, \sigma_0)$

$\mathsf{KAgg}(L, k)$:

1: **Parse** $L = (\mathsf{pk}_1, \dots, \mathsf{pk}_n) \in \mathbb{G}^n$
2: **for** $i \in [1, n]$ **do**
3: $\quad a_i = \mathsf{H}_{\text{agg}}(L, \mathsf{pk}_i)$
4: $\mathsf{pk} = \prod_{i=1}^{k} \mathsf{pk}_i^{a_i}$
5: **return** pk

$\mathsf{Vrfy}(\mathsf{pk}, m, \sigma)$:

1: **Parse** $\sigma = (R, z) \in \mathbb{G} \times \mathbb{Z}_p$
2: $c \leftarrow \mathsf{H}_{\text{ch}}(m, R, \mathsf{pk}) \in \mathbb{Z}_p$
3: **if** $g^z = R \cdot \mathsf{pk}^c$ **then**
4: $\quad$ **return** 1
5: **return** 0

$\mathsf{SignOn}_{\mathtt{st}_i}(\mathsf{sk}_i, m, L, \mathsf{offs}, \sigma_{i-1})$:

▷ Lines 1-10: processing independent of m, σ_{i-1}
1: **Parse** $\mathtt{st}_i = (r_{i,j} | R'_{i,j})_{j=1}^{\ell}$
2: **Parse** $\mathsf{offs} = ((R_{1,j})_{j=1}^{\ell}, \dots, (R_{n,j})_{j=1}^{\ell})$
3: **if** $R'_{i,j} \neq R_{i,j}$ **then return** ε
4: **for** $j \in [1, \ell]$ **do**
5: $\quad R_j := \prod_{k=1}^{\ell} R_{k,j}$
6: $\quad \tilde{R}_j := \prod_{k=1}^{i-1} R_{k,j}$
7: $\quad$ **if** $R_j = 1_{\mathbb{G}}$ or $\tilde{R}_j = 1_{\mathbb{G}}$ **then return** ε
8: $\mathsf{pk} := \mathsf{KAgg}(L, n)$
9: $\widetilde{\mathsf{pk}} := \mathsf{KAgg}(L, i-1)$
10: $a_i := \mathsf{H}_{\text{agg}}(L, \mathsf{pk}_i)$
▷ Lines 11-22: processing that depends on m, σ_{i-1}
11: **Parse** $\sigma_{i-1} = (R, \tilde{z})$
12: $v \leftarrow \mathsf{H}_{\text{non}}(m, \mathsf{offs}, \mathsf{pk})$
13: **if** $i = 1$ **then**
14: $\quad R := \prod_{j=1}^{\ell} R_j^{v^{j-1}}$
15: $\quad c \leftarrow \mathsf{H}_{\text{ch}}(m, R, \mathsf{pk})$
16: **else** ▷ Check so-far aggregation
17: $\quad$ **if** $R \neq \prod_{j=1}^{\ell} R_j^{v^{j-1}}$ **then return** ε
18: $\quad \tilde{R} := \prod_{j=1}^{\ell} \tilde{R}_j^{v^{j-1}}$
19: $\quad c \leftarrow \mathsf{H}_{\text{ch}}(m, R, \mathsf{pk})$
20: $\quad$ **if** $g^{\tilde{z}} \neq \tilde{R} \cdot \widetilde{\mathsf{pk}}^c$ **then return** $\perp^*$
21: $z_i := c \cdot a_i \cdot \mathsf{sk}_i + \sum_{j=1}^{\ell} v^{j-1} \cdot r_{i,j}$
22: $z := \tilde{z} + z_i$ **return** $\sigma_i := (R, z)$

Fig. 26: OMS construction supporting key aggregation and security in the PPK model

F Multi-Signatures in the UC Framework

In this section, we formalize variants of multi-signatures in a bottom-up manner. First, we present $\mathcal{F}_{\mathsf{MS}}$ as generalization of the $\mathcal{F}_{\mathsf{Sig}}$ functionality. Then we extend it to $\mathcal{F}_{\mathsf{IMS}}$ to capture interactive multi-signatures with preprocessing. Finally, we modify it to $\mathcal{F}_{\mathsf{OMS}}$ to model interactive ordered multi-signatures with preprocessing.

F.1 Ideal Functionality for Multi-Signatures

As a warm-up, we present an ideal functionality $\mathcal{F}_{\mathsf{MS}}$ for (non-interactive) multi-signatures. An ideal functionality for stake-based threshold multi-signatures exists in the literature [31]. Our functionality is much simpler because it aims to capture existing schemes in the standard plain-public key model. It can be seen as a generalization of $\mathcal{F}_{\mathsf{Sig}}$ from [25] with following differences:

- $\mathcal{F}_{\mathsf{Sig}}$ is defined for a single designated signer S and only accepts a key registration query with sid encoding a signer identity, i.e., $\mathsf{sid} = (\mathsf{sid}', S)$ for some sid', whereas $\mathcal{F}_{\mathsf{MS}}$ is defined w.r.t. n signers $\mathcal{P}_1, \ldots, \mathcal{P}_n$ and records an individual key for every signer.
- $\mathcal{F}_{\mathsf{MS}}$'s Sign command takes a key list L together with message m as input. To capture security in the plain public key model, the functionality does not validate keys of co-signers. This allows the environment to insert maliciously created self-chosen keys into L.
- $\mathcal{F}_{\mathsf{MS}}$ additionally has the Aggregate command, which allows any party to aggregate n partial signatures into a single combined signature σ. Note that this command does not perform validity check of individual parial signature.
- $\mathcal{F}_{\mathsf{MS}}$ verifies a multi-signature analogously to $\mathcal{F}_{\mathsf{Sig}}$, but its procedures are more involved due to multiple signers. Essentially, it guarantees completes by checking that Aggregate command was invoked on some $\sigma_1, \ldots, \sigma_\ell$ which have been explicitly created via Sign command on input L and m. To model unforgeability, if there exists some uncorrupt party $\mathcal{P}_i$ that has never agreed to sign m with L, then it rejects a signature. Otherwise, $\mathcal{F}_{\mathsf{MS}}$ asks an ideal adversary to determine the result of verification.

By extending the result of [25] in a straightforward manner, one can show that non-interactive multi-signature schemes meeting the standard UF-CMA security in the plain public-key model (e.g. BLS-based scheme of [16]) UC-realizes $\mathcal{F}_{\mathsf{MS}}$. However, $\mathcal{F}_{\mathsf{MS}}$ does not model many recent schemes requiring interaction between parties. $\mathcal{F}_{\mathsf{IMS}}$ (Figure 28) is an extended functionality with a setup phase where parties register themselves and their intended co-signers (in a way similar to the ideal functionality for threshold signatures [29]). This phase is necessary for interactive schemes where each party requires the knowledge of co-signers' party ID. It also turns out useful for modeling some existing schemes that can preprocess the first round of interaction without knowing a message to be signed. Then once the functionality received a message m to be signed, it hands over a

partial signature σ_i to $\mathcal{P}_i$ as long as all parties encoded in ssid have completed the setup phase.

Finally, we present $\mathcal{F}_{\mathsf{OMS}}$ (Figure 29) to model *ordered* interactive multi-signatures. Unlike $\mathcal{F}_{\mathsf{MS}}$ and $\mathcal{F}_{\mathsf{IMS}}$, this functionality verifies that all n keys are registered and is usable for fixed ordering of exactly n keys, which is sufficient for our application. This restriction can be dropped in case the plain-public key model is desired. Note that $\mathcal{F}_{\mathsf{OMS}}$ has not aggregation interface since OMS assumes every party in a sequence to perform partial aggregaton as it contributes to signing. Order of online signing is guaranteed by making sure that all prior signers in the list have recorded partial signatures for the same (ssid, L, m).

F.2 UC Security of Interactive Multi-Signatures

Game-based Security of Interactive Multi-Signatures We recall the syntax of offline-online interactive multi-signatures and the game-based security notion.

Definition F.1 (Interactive Multi-Signature Scheme (IMS)). *We define interactive multi-signature* IMS *as a tuple of algorithms*[10]

$$\mathsf{IMS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{SignOff}, \mathsf{SignOn}, \mathsf{Agg}, \mathsf{Vrfy})$$

with the following input-output behavior. To formally handle the interactive signing, SignOff *and* SignOn *are stateful algorithms that share a common state* $\mathtt{st}$.

$\mathsf{Setup}(1^\tau)$**:** *on input the security parameter τ, this algorithm outputs a handle of public parameters* pp. *Throughout, we assume that* pp *is given as implicit input to all other algorithms.*

$\mathsf{KeyGen}()$**:** *on input the public parameters, the key generation algorithm outputs a key pair* $(\mathsf{pk}, \mathsf{sk})$.

$\mathsf{SignOff}_{\mathtt{st}}(\mathsf{sk}, \mathsf{aux})$**:** *on input a secret key* sk, *and some (possibly empty) auxiliary information* aux*; the offline signing algorithm outputs an offline token* off. *This is a stateful algorithm, and updates* $\mathtt{st}$ *at every execution. We note that this algorithm is agnostic of the message m to be signed, and runs independently of m.*

[10] We remark that IMSs as defined in [58] have an additional signing function that post-processes the aggregated online tokens to generate a signature. We assume this function to be trivial, meaning that it merely outputs the input aggregated online tokens as a signature (as MuSig2 does) and therefore can be omitted. Another minor modification from [58] is that Vrfy explicitly takes a public key list L instead of a single aggregated key. This does not impact the security claim: since in the EUF-CMA game of [58] the verifier is guaranteed to receive an output of key aggregation anyway, we can assume this operation happens inside Vrfy. Finally, we rename the algorithms involved in the interactive signing procedure to follow the pattern we introduced for OMS.

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

The functionality interacts with n signers in $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, a verifier $\mathcal{V}$, and a simulator $\mathcal{S}$. The adversary can corrupt $c < n$ signers, denoted by the set $\mathcal{C} \subsetneq \mathcal{P}$. We assume the functionality only accepts a key set L that has cardinality at most n and does not contain duplicate keys.

Key generation Upon receiving input $(\text{KeyGen}, \mathsf{sid})$ from a signer $\mathcal{P}_i \in \mathcal{P}$:

1. If there exists a record $(\text{keyrec}, \mathcal{P}_i, *)$, then abort. ▷ Prevent overwriting
2. Send $(\text{KeyGen}, \mathsf{sid}, \mathcal{P}_i)$ to $\mathcal{S}$ and wait for $(\text{KeyConf}, \mathsf{sid}, \mathsf{pk}_i)$ from $\mathcal{S}$.
3. If there exists a record $(\text{sigrec}, L, *, *, *)$ such that $\mathsf{pk}_i \in L$, then abort. ▷ Enforce distinct public keys
4. Create record $(\text{keyrec}, \mathcal{P}_i, \mathsf{pk}_i)$ and output $(\text{KeyConf}, \mathsf{sid}, \mathsf{pk}_i)$ to $\mathcal{P}_i$.

Signature Generation Upon receiving input $(\text{Sign}, \mathsf{sid}, L, m)$ from any $\mathcal{P}_i \in \mathcal{P}$:

1. If there exists **no** record $(\text{keyrec}, \mathcal{P}_i, \mathsf{pk}_i)$, then abort.
2. Retrieve $(\text{keyrec}, \mathcal{P}_i, \mathsf{pk}_i)$. If $\mathsf{pk}_i \notin L$, then abort.
3. Send $(\text{Sign}, \mathsf{sid}, \mathcal{P}_i, L, m)$ to $\mathcal{S}$, and wait for $(\text{Signature}, \mathsf{sid}, \sigma_i)$ from $\mathcal{S}$.
4. Create a record $(\text{psigrec}, \mathcal{P}_i, L, m, \sigma_i)$.
5. Output $(\text{Signature}, \mathsf{sid}, \sigma_i)$ to $\mathcal{P}_i$.

Signature Aggregation Upon receiving input $(\text{Aggregate}, \mathsf{sid}, L, m, \sigma_1, \ldots$ $\ldots, \sigma_\ell)$ from any party:

1. If $|L| \neq \ell$, then abort.
2. Send $(\text{Aggregate}, \mathsf{sid}, L, m, \{\sigma_1, \ldots, \sigma_\ell\})$ to $\mathcal{S}$, wait for $(\text{Aggregated}, \mathsf{sid}, \sigma)$ from $\mathcal{S}$.
3. Create a record $(\text{aggrec}, L, m, \{\sigma_1, \ldots, \sigma_\ell\}, \sigma)$ and output $(\text{Aggregated}, \mathsf{sid}, \sigma)$.

Signature Verification Upon receiving input $(\text{Verify}, \mathsf{sid}, L, m, \sigma)$ from $\mathcal{V}$, send $(\text{Verify}, \mathsf{sid}, L, m, \sigma)$ to $\mathcal{S}$. On receiving $(\text{Verified}, \mathsf{sid}, b')$ from $\mathcal{S}$:

1. If there exists a record $(\text{sigrec}, L, m, \sigma, b)$, set $f := b$. ▷ Consistency
2. Let $L' \subseteq L$ be the **maximal** subset of L such that for all $\mathsf{pk}' \in L'$ there exists a record $(\text{keyrec}, \mathcal{P}_i, \mathsf{pk}_i)$ such that $\mathsf{pk}_i = \mathsf{pk}'$. Let $\mathcal{P}' \subseteq \mathcal{P}$ the set of parties corresponding to L' and $I' \subseteq [n]$ be the set of index such that for all $i \in I'$ $\mathcal{P}_i \in \mathcal{P}'$:
   - If (i) $L' = L$, (ii) for all $\mathcal{P}_i \in \mathcal{P}'$, there exists a record $(\text{psigrec}, \mathcal{P}_i, L, m, \sigma_i)$ for some σ_i, and (iii) there exists a record $(\text{aggrec}, L, m, \{\sigma_i\}_{i \in I'}, \sigma)$, then set $f := 1$ ▷ Completeness
   - Else, if for some $\mathcal{P}_i \in \mathcal{P}' \cap (\mathcal{P} \setminus \mathcal{C})$, there exists **no** record $(\text{psigrec}, \mathcal{P}_i, L, m, *)$, then set $f := 0$. ▷ Unforgeability
   - Else, set $f := b'$.
3. Create a record $(\text{sigrec}, L, m, \sigma, f)$ and output $(\text{Verified}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Fig. 27: MS ideal functionality in the plain public-key model

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

The functionality interacts with n signers in $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, a verifier $\mathcal{V}$, and a simulator $\mathcal{S}$. The adversary can corrupt $c < n$ signers, denoted by the set $\mathcal{C} \subsetneq \mathcal{P}$. We assume the functionality only accepts a key set L that has cardinality at most n and doe not contain duplicate keys. We distinguish different offline phases by using unique ssids encoding co-signers' identities.

Key generation Same as $\mathcal{F}_{\mathsf{MS}}$.

Signature Setup (Offline Phase) Upon receiving $(\textsc{SignOff}, \mathsf{sid}, \mathsf{ssid}, L)$ from $\mathcal{P}_i \in \mathcal{P}$ or $(\textsc{SignOff}, \mathsf{sid}, \mathsf{ssid}, L, \mathcal{P}_i)$ from $\mathcal{S}$:

1. Check that $\mathsf{ssid} = (\mathsf{ssid}', \{\mathcal{P}_j\}_{j \in I})$ for some ssid', $I \subseteq [n]$, $|L| = |I|$, and $i \in I$. If not, then abort. ▷ Check valid signer identities are encoded in ssid
2. If the input is from $\mathcal{P}_i \in \mathcal{P} \setminus \mathcal{C}$:
   - Retrieve $(\textsc{keyrec}, \mathcal{P}_i, \mathsf{pk}_i)$ or abort if no such record exists.
   - If $\mathsf{pk}_i \notin L$, then abort.
   - If there exists record $(\textsc{joined}, \mathsf{ssid}, *, \mathcal{P}_i)$, then abort ▷ Honest parties will only use ssid once.
   - Else, send $(\textsc{Join}, \mathsf{sid}, \mathsf{ssid}, L, \mathsf{pk}_i)$ to $\mathcal{S}$ and create record $(\textsc{joined}, \mathsf{ssid}, L, \mathcal{P}_i)$.
3. If the input is from $\mathcal{S}$:
   - If $\mathcal{P}_i \notin \mathcal{C}$, then abort.
   - Else, create record $(\textsc{joined}, \mathsf{ssid}, L, \mathcal{P}_i)$ ▷ Ideal adversary notifies a corrupt party $\mathcal{P}_i$ has joined.
4. If for all $j \in I$, there exists record $(\textsc{joined}, \mathsf{ssid}, L, \mathcal{P}_j)$ for the same (ssid, L), create record $(\textsc{sigsetup}, \mathsf{ssid}, L)$. ▷ All parties have completed the setup for L for this ssid.
5. Output $(\textsc{SignedOff}, \mathsf{sid}, \mathsf{ssid})$ to $\mathcal{P}_i$.

Signature Generation (Online Phase) Upon receiving $(\textsc{SignOn}, \mathsf{sid}, \mathsf{ssid}, L, m)$ from any $\mathcal{P}_i \in \mathcal{P}$:

1. If **no** record $(\textsc{sigsetup}, \mathsf{ssid}, L)$ exists, return $(\textsc{fail}, \mathsf{sid}, \mathsf{ssid})$. ▷ Setup for ssid must be completed before signing and setup must be fresh.
2. If $\mathcal{P}_i \in \mathcal{P} \setminus \mathcal{C}$ has sent $(\textsc{Signature}, \mathsf{sid}, \mathsf{ssid}, *)$ before, return $(\textsc{fail}, \mathsf{sid}, \mathsf{ssid})$. ▷ Honest parties only issue one partial signature per ssid.
3. Send $(\textsc{Sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_i, L, m)$ to $\mathcal{S}$, and wait for $(\textsc{Signature}, \mathsf{sid}, \mathsf{ssid}, \sigma_i)$ from $\mathcal{S}$.
4. Create record $(\textsc{psigrec}, \mathsf{ssid}, \mathcal{P}_i, L, m, \sigma_i)$.
5. Output $(\textsc{Signature}, \mathsf{sid}, \mathsf{ssid}, \sigma_i)$ to $\mathcal{P}_i$.

Signature Aggregation Same as $\mathcal{F}_{\mathsf{MS}}$.

Signature Verification Upon receiving input $(\textsc{Verify}, \mathsf{sid}, L, m, \sigma)$ from $\mathcal{V}$:

1. If there exists a record $(\textsc{sigrec}, L, m, \sigma, b)$, set $f := b$. ▷ Consistency
2. Let $L' \subseteq L$ be the **maximal** subset of L such that for all $\mathsf{pk}' \in L'$ there exists a record $(\textsc{keyrec}, \mathcal{P}_i, \mathsf{pk}_i)$ such that $\mathsf{pk}_i = \mathsf{pk}'$. Let $\mathcal{P}' \subseteq \mathcal{P}$ the set of parties corresponding to L' and $I' \subseteq [n]$ be the set of index such that for all $i \in I'$ $\mathcal{P}_i \in \mathcal{P}'$:
   - If (i) $L' = L$, (ii) for all $\mathcal{P}_i \in \mathcal{P}'$, there exists a record $(\textsc{psigrec}, \mathsf{ssid}, \mathcal{P}_i, L, m, \sigma_i)$ for the same ssid and for some σ_i, and (iii) there exists a record $(\textsc{aggrec}, L, m, \{\sigma_i\}_{i \in I'}, \sigma)$, then set $f := 1$ ▷ Completeness
   - Else, if for some $\mathcal{P}_i \in \mathcal{P}' \cap (\mathcal{P} \setminus \mathcal{C})$, there exists **no** record $(\textsc{psigrec}, *, \mathcal{P}_i, L, m, *)$, then set $f := 0$. ▷ Unforgeability
   - Else, set $f := b'$.
3. Create a record $(\textsc{sigrec}, L, m, \sigma, f)$ and output $(\textsc{Verified}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Fig. 28: IMS ideal functionality in the plain public-key model. Differences from $\mathcal{F}_{\mathsf{MS}}$ are in this font.

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

The functionality interacts with n signers in $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, a verifier $\mathcal{V}$, and a simulator $\mathcal{S}$. The adversary can corrupt $c < n$ signers, denoted by the set $\mathcal{C} \subsetneq \mathcal{P}$. We assume the functionality only accepts a key set L that has cardinality at most n and doe not contain duplicate keys. We distinguish different offline phases by using unique ssids encoding co-signers' identities. In what follows, L denotes the list of signers' public keys, ordered according to the index of the corresponding signer, i.e., let pk_i denote the public key of signer $\mathcal{P}_i$, then $L = (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$. Signature setup, signature generation, and verification only accept input containing L such that its all n public keys are registered in the functionality.

Key generation Same as in $\mathcal{F}_{\mathsf{MS}}$ upon receiving input from $\mathcal{P}_i \notin \mathcal{C}$. Moreover, upon receiving $(\text{KeyGen}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pk}_i)$ from $\mathcal{S}$ for $\mathcal{P}_i \in \mathcal{C}$, create record $(\text{keyrec}, \mathcal{P}_i, \mathsf{pk}_i)$.

Signature Setup (Offline Phase) Same as $\mathcal{F}_{\mathsf{IMS}}$.

Signature Generation (Online Phase) Upon receiving input $(\text{Sign}, \mathsf{sid}, \mathsf{ssid}, L, m, \sigma_{i-1})$ from any $\mathcal{P}_i \in \mathcal{P}$:

1. If **no** record $(\text{sigsetup}, \mathsf{ssid}, L)$ exists, return $(\text{fail}, \mathsf{sid}, \mathsf{ssid})$. ▷ Setup for ssid must be completed before signing and setup must be fresh.
2. If $\mathcal{P}_i \in \mathcal{P} \setminus \mathcal{C}$ has sent $(\text{Signature}, \mathsf{sid}, \mathsf{ssid}, *)$ before, return $(\text{fail}, \mathsf{sid}, \mathsf{ssid})$. ▷ Honest parties only issue one partial signature per ssid.
3. If $1 < i \leq n$ **and no** record $(\text{psigrec}, \mathsf{ssid}, \mathcal{P}_j, L, m, *)$ exists for some $1 \leq j < i$ such that $\mathcal{P}_j \in \mathcal{P} \setminus \mathcal{C}$, then return $(\text{Fail}, \mathsf{sid}, L, m)$. ▷ Honest signers prior to i must have already signed
4. Send $(\text{Sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}_i, L, m, \sigma_{i-1})$ to $\mathcal{S}$, and wait for $(\text{Signature}, \mathsf{sid}, \mathsf{ssid}, \sigma_i)$ from $\mathcal{S}$.
5. If $\mathcal{P}_i \in \mathcal{P} \setminus C$ and $\sigma_i \in \{\perp^*, \varepsilon\}$, then return $(\text{Fail}, \mathsf{sid}, L, m)$
6. Create record $(\text{psigrec}, \mathsf{ssid}, \mathcal{P}_i, L, m, \sigma_i)$.
7. Output $(\text{Signature}, \mathsf{sid}, \mathsf{ssid}, \sigma_i)$ to $\mathcal{P}_i$.

Signature Verification Upon receiving input $(\text{Verify}, \mathsf{sid}, L, m, \sigma)$ from $\mathcal{V}$:

1. If there exists a record $(\text{sigrec}, L, m, \sigma, b)$, set $f := b$. ▷ Consistency
2. Let $L' \subseteq L$ be the **maximal** subset of L such that for all $\mathsf{pk}' \in L'$ there exists a record $(\text{keyrec}, \mathcal{P}_i, \mathsf{pk}_i)$ such that $\mathsf{pk}_i = \mathsf{pk}'$. Let $\mathcal{P}' \subseteq \mathcal{P}$ the set of parties corresponding to L' and $I' \subseteq [n]$ be the set of index such that for all $i \in I'$ $\mathcal{P}_i \in \mathcal{P}'$:
   - If (i) $L' = L$, (ii) for all $\mathcal{P}_i \in \mathcal{P}'$, there exists a record $(\text{psigrec}, \mathsf{ssid}, \mathcal{P}_i, L, m, \sigma_i)$ for the same ssid and for some σ_i, and (iii) $\sigma_n = \sigma$, then set $f := 1$ ▷ Completeness
   - Else, if for some $\mathcal{P}_i \in \mathcal{P}' \cap (\mathcal{P} \setminus \mathcal{C})$, there exists **no** record $(\text{psigrec}, *, \mathcal{P}_i, L, m, *)$, then set $f := 0$. ▷ Unforgeability
   - Else, set $f := b'$.
3. Create a record $(\text{sigrec}, L, m, \sigma, f)$ and output $(\text{Verified}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Fig. 29: OMS ideal functionality $\mathcal{F}_{\mathsf{OMS}}$. Differences from $\mathcal{F}_{\mathsf{IMS}}$ are in this font.

$\mathsf{SignOn_{st}}(\mathsf{sk}, m, \mathsf{offs}, L)$**:** *on input a secret key* sk*, message* m*, offline tokens* $\mathsf{offs} = (\mathsf{off}_1, \ldots, \mathsf{off}_n)$*, and a list of public keys* $L = (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$*, the online signing algorithm outputs an online signing token* σ*, and updates its internal state* $\mathtt{st}$*.*

$\mathsf{Agg}(\sigma_1, \ldots, \sigma_n)$**:** *on input* n *partial online tokens, the online aggregation algorithm outputs a signature* σ*. This algorithm is deterministic.*

$\mathsf{Vrfy}(L, m, \sigma)$**:** *on input a list of public keys* L*, a message* m*, and a signature* σ*, the verification algorithm outputs* 1 *(accept) or* 0 *(reject).*

Following [58], we recall the game-based security notion tailored to two-round offline-online IMS, except that we assume that aggregation of offline tokens is locally performed by each party (which is in fact the setting considered captured by a more widely used security notion of [11]).

Definition F.2 (IMS Security). *An interactive multi signature* IMS *is said to be secure if for any probabilistic polynomial time adversary* $\mathcal{A}$ *and* $\forall \tau$ *it holds that:*

$$\mathbf{Adv}^{\mathsf{IMS\text{-}UF\text{-}CMA}}_{\mathsf{IMS}}(\mathcal{A}, \tau) := \Pr[1 \leftarrow \mathsf{IMS\text{-}UF\text{-}CMA}(\mathcal{A}, \tau)] \leq \mathsf{negl}(\tau)$$

where IMS-UF-CMA *is the security-game for unforgeability under chosen-message attack of interactive multi-signatures defined in Figure 30.*

Security Model for IMS

Game $\mathsf{IMS\text{-}UF\text{-}CMA}(\mathcal{A}, \tau)$

```
1:  T := ∅; Q := ∅
2:  pp ← Setup(1^τ)
3:  (pk, sk) ← KeyGen(pp)
4:  O := {OSignOff, OSignOn }        ▷ If pp
    contain hash functions, RO is in O
5:  (L*, m*, σ*) ← A^O(pp, pk)
6:  if pk ∉ L* then
7:    return 0
8:  if ∃ sid* : Q[sid*] = (L*, m*) then
9:    return 0
10: return Vrfy(L*, m*, σ*)
```

Oracles

$\mathsf{OSignOff}(\mathsf{sid}, \mathsf{aux})$

```
1: if T[sid] ≠ ⊥ then
2:   return session exists
3: off ← SignOff_st(sk, aux)
4: T[sid] := st
5: return off
```

$\mathsf{OSignOn}(\mathsf{sid}, L, m, \mathsf{offs})$

```
1: if pk ∉ L then return honest
   signer missing
2: if T[sid] = ⊥ ∨ Q[sid] ≠ ⊥ then
3:   return invalid session
4: st := T[sid]
5: σ ← SignOn_st(sk, m, offs, L)
6: if σ ≠ ⊥ then
7:   Q[sid] := (L, m)
8: return σ
```

Fig. 30: Game-based security model for IMS (IMS-UF-CMA). For each new session, $\mathtt{st}$ is initially assumed to be empty.

Security Proof In Fig. 31 we define π_{IMS} as a direct instantiation of IMS in the UC model. Note that, as observed in the analysis of standard signature schemes in the UC framework [23], the proof would go through even if $\mathcal{G}_{\mathsf{rpoRO}}$ is replaced with a *strict* variant of the global random oracle, since the simulator merely relays random oracle queries made by an algorithm of IMS to the global functionality.

Theorem F.3. *Let* IMS *be an interactive multi-signature scheme (Definition F.1) that is* IMS-UF-CMA *secure in the random oracle model (Definition F.2) and let $n \in \mathsf{poly}(\lambda)$. Then the protocol π_{IMS} (Figure 31) UC-realizes $\mathcal{F}_{\mathsf{IMS}}$ in the $\mathcal{G}_{\mathsf{rpoRO}}$-hybrid model.*

Proof. In Figure 32 we describe a PPT simulator $\mathcal{S}_{\mathsf{IMS}}$ that has black-box access to $\mathcal{A}$, and simulates $\mathcal{F}_{\mathsf{IMS}}$ in a way that is indistinguishable to any efficient environment. The IMS scheme used in the proof follows the syntax given in Def. F.1.

The proof closely follows that of [23] for a single-user signature scheme. Let us go over each interface.

- KeyGen: The only difference is when $\mathcal{F}_{\mathsf{IMS}}$ aborts after receiving pk_i from $\mathcal{S}_{\mathsf{IMS}}$. This is negligible assuming sufficiently high min-entropy of honestly generated pk.
- SignOff, SignOn, Aggregate: Since $\mathcal{S}_{\mathsf{IMS}}$ runs $\mathsf{SignOff}, \mathsf{SignOn}$, Agg as in π_{IMS}, there is no difference.
- Verify: We consider the following cases:
  - If there is **no** corruption **and** for input (L, m) SignOn is completed with partial signatures $(\sigma_1, \ldots, \sigma_\ell)$ **and** Aggregate is completed by an honest party with input $(L, m, \sigma_1, \ldots, \sigma_\ell)$: By the correctness of IMS, there is no difference in the view of $\mathcal{Z}$.
  - If for input (L, m) all honest parties $\mathcal{P}_i \notin \mathcal{C}$ complete SignOn with partial signature σ_i: In this case, $\mathcal{F}_{\mathsf{IMS}}$ asks $\mathcal{S}_{\mathsf{IMS}}$ to simulate a decision bit. Since $\mathcal{S}_{\mathsf{IMS}}$ also runs Vrfy as in π_{IMS}, there is no difference in the view of $\mathcal{Z}$.
  - If for input (L, m) some honest party $\mathcal{P}_i \notin \mathcal{C}$ has **not** completed SignOn: In this case, there is a potential discrepancy in the view of $\mathcal{Z}$ since $\mathcal{F}_{\mathsf{IMS}}$ always outputs a decision bit 0 while π_{IMS} returns the output of $\mathsf{Vrfy}(L, m, \sigma)$. However, if $\mathcal{Z}$ manages to come up with (L, m, σ) causing Vrfy to return 1, this implies $\mathcal{Z}$ created a valid forgery in the IMS-UF-CMA game. That is, using such $\mathcal{Z}$ as a subroutine one can construct a reduction $\mathcal{B}$ breaking IMS-UF-CMA. On receiving a challenge public key pk, $\mathcal{B}$ with access to $\mathsf{OSignOff}, \mathsf{OSignOn}$ and the random oracle H plays the role of $\mathcal{F}_{\mathsf{IMS}}$, $\mathcal{S}_{\mathsf{IMS}}$, and $\mathcal{G}_{\mathsf{rpoRO}}$ and proceeds as follows.
    - Use pk as a public key for one of the uncorrupted parties. Let $i^* \in [n]$ be the index of that party. Generate key pairs for all the other honest parties as $\mathcal{S}_{\mathsf{IMS}}$ would.
    - Whenever $\mathcal{Z}$ makes a query to $\mathcal{G}_{\mathsf{rpoRO}}$, relay queries and responses to and from H.

* Whenever $\mathcal{S}_{\mathsf{IMS}}$ is asked to run JOIN command for party i^*, query $\mathsf{OSignOff}$ with ssid to receive off_{i^*} and forwards it to a copy of the adversary $\mathcal{A}$. Otherwise, JOIN command is handled as in $\mathcal{S}_{\mathsf{IMS}}$.
* Whenever $\mathcal{S}_{\mathsf{IMS}}$ is asked to run SIGN command for party i^*, query $\mathsf{OSignOn}$ with input $(\mathsf{ssid}, L, m, \mathsf{offs})$ to receive σ_{i^*}. Otherwise, SIGN command is handled as in $\mathcal{S}_{\mathsf{IMS}}$.
* If $\mathcal{Z}$ outputs (L, m, σ) causing $\mathsf{Vrfy}(L, m, \sigma) = 1$ while $\mathsf{pk}_{i^*} \in L$ and (L, m) has never been queried to $\mathsf{OSignOn}$, forward this tuple to the $\mathsf{IMS\text{-}UF\text{-}CMA}$ game.

In this way, the reduction $\mathcal{B}$ succeeds in winning the $\mathsf{IMS\text{-}UF\text{-}CMA}$ game as long as the party index i^* is correctly guessed. Hence, with a multiplicative factor of loss $1/n$, the advantage of $\mathcal{B}$ is non-negligible if $\mathcal{Z}$ finds inconsistency of verification in real and ideal executions with non-negligible probability.

□

Protocol π_{IMS}

The protocol is parameterized by $\mathsf{IMS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{SignOff}, \mathsf{SignOn}, \mathsf{Agg}, \mathsf{Vrfy})$ and $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\tau)$ and executed by n signers $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and a verifier $\mathcal{V}$. Whenever an algorithm of IMS queries to a random oracle, π_{IMS} relays queries to and responses from $\mathcal{G}_{\mathsf{rpoRO}}$.

Key generation Upon $(\textsc{KeyGen}, \mathsf{sid})$ a signer $\mathcal{P}_i \in \mathcal{P}$ proceeds as follows:

1. If there exists a record $(\textsc{KeyRec}, \mathsf{sid}, *)$, then abort.
2. Run $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\tau)$.
3. Create record $(\textsc{keyrec}, \mathsf{sk}_i, \mathsf{pk}_i)$ and output $(\textsc{KeyConf}, \mathsf{sid}, \mathsf{pk}_i)$.

Signature Setup (Offline Phase) Upon $(\textsc{SignOff}, \mathsf{sid}, \mathsf{ssid}, L)$, a signer $\mathcal{P}_i \in \mathcal{P}$ proceeds as follows:

1. Check that $\mathsf{ssid} = (\mathsf{ssid}', \{\mathcal{P}_j\}_{j \in I})$ for some ssid', $I \subseteq [n]$, $|L| = |I|$, and $i \in I$. If not, then abort.
2. If there exists **no** record $(\textsc{keyrec}, *, *)$, then abort.
3. Retrieve $(\textsc{keyrec}, \mathsf{sk}_i, \mathsf{pk}_i)$. If $\mathsf{pk}_i \notin L$, then abort.
4. If there exists record $(\textsc{joined}, \mathsf{ssid}, *)$, then abort.
5. Run $\mathsf{off}_i \leftarrow \mathsf{SignOff}_{\mathtt{st}_i}(\mathsf{sk}_i, \mathsf{aux} = L)$, broadcast $(\mathsf{ssid}, L, \mathcal{P}_i, \mathsf{off}_i)$, and create record $(\textsc{joined}, \mathsf{ssid}, L)$.
6. On receiving $(\mathsf{ssid}, \mathcal{P}_j, \mathsf{pk}_j, \mathsf{off}_j)$ for all $j \in I$ such that $j \neq i$, create a local record $(\textsc{sigsetup}, \mathsf{ssid}, L, \mathsf{offs} = \{\mathsf{off}_j\}_{j \in I}, \mathtt{st}_i)$.
7. Output $(\textsc{SignedOff}, \mathsf{sid}, \mathsf{ssid})$.

Signature Generation (Online Phase) Upon receiving $(\textsc{SignOn}, \mathsf{sid}, \mathsf{ssid}, L, m)$, a signer $\mathcal{P}_i \in \mathcal{P}$ proceeds as follows:

1. If there exists **no** record $(\textsc{sigsetup}, \mathsf{ssid}, L, *, *)$, then return $(\textsc{Fail}, \mathsf{sid}, \mathsf{ssid})$.
2. If $\mathcal{P}_i$ has output $(\textsc{Signature}, \mathsf{sid}, \mathsf{ssid}, *)$ before, return $(\textsc{Fail}, \mathsf{sid}, \mathsf{ssid})$.
3. Retrieve record $(\textsc{sigsetup}, \mathsf{ssid}, L, \mathsf{offs}, \mathtt{st}_i)$
4. Run $\sigma_i \leftarrow \mathsf{SignOn}_{\mathtt{st}_i}(\mathsf{sk}_i, m, \mathsf{offs}, L)$.
5. Output $(\textsc{Signature}, \mathsf{sid}, \mathsf{ssid}, \sigma_i)$.

Signature Aggregation Upon receiving $(\textsc{Aggregate}, \mathsf{sid}, L, m, \sigma_1, \ldots, \sigma_n)$, a party $\mathcal{P}_i \in \mathcal{P}$ runs $\sigma \leftarrow \mathsf{Agg}(\sigma_1, \ldots, \sigma_n)$ and outputs $(\textsc{Agg}, \mathsf{sid}, \sigma)$.

Signature Verification Upon receiving input $(\textsc{Verify}, \mathsf{sid}, L, m, \sigma)$, $\mathcal{V}$ runs $b \leftarrow \mathsf{Vrfy}(L, m, \sigma)$ and outputs $(\textsc{Verified}, \mathsf{sid}, b)$.

Fig. 31: IMS protocol

Simulator $\mathcal{S}_{\mathsf{IMS}}$

$\mathcal{S}_{\mathsf{IMS}}$ interacts with an internal copy of $\mathcal{A}$, towards which it simulates the honest parties in $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and relays all the random oracle calls requested by $\mathsf{SignOff}$, SignOn, and Vrfy externally to the global random oracles functionality $\mathcal{G}_{\mathsf{rpoRO}}$.

KeyGen: On receiving $(\textsc{KeyGen}, \mathsf{sid}, \mathcal{P}_i)$ from $\mathcal{F}_{\mathsf{IMS}}$, run KeyGen to obtain the key pair $(\mathsf{sk}_i, \mathsf{pk}_i)$. Return $(\textsc{KeyConf}, \mathsf{sid}, \mathsf{pk}_i)$ to $\mathcal{F}_{\mathsf{IMS}}$.

Sign Setup: On receiving $(\textsc{Join}, \mathsf{sid}, \mathsf{ssid}, L, \mathsf{pk}_i)$ from $\mathcal{F}_{\mathsf{IMS}}$:

1. Store $(\textsc{Join}, \mathsf{sid}, \mathsf{ssid}, L, \mathsf{pk}_i)$.
2. Run $\mathsf{off}_i \leftarrow \mathsf{SignOff}_{\mathtt{st}_i}(\mathsf{sk}_i, L)$ and create record $(\mathsf{ssid}, L, \mathcal{P}_i, \mathsf{off}_i, \mathtt{st}_i)$. $\triangleright$ $\mathtt{st}_i$ is needed for online signing
3. Send $(\mathsf{ssid}, L, \mathcal{P}_i, \mathsf{off}_i)$ to $\mathcal{A}$.

On receiving $(\mathsf{ssid}, L, \mathcal{P}_i, \mathsf{off}_i)$ from $\mathcal{A}$ for corrupted $\mathcal{P}_i$, create record $(\mathsf{ssid}, L, \mathcal{P}_i, \mathsf{off}_i)$ and output $(\textsc{SignOff}, \mathsf{sid}, L, \mathcal{P}_i)$ to $\mathcal{F}_{\mathsf{IMS}}$. Once all $\mathsf{offs} := \{\mathsf{off}_j\}_{j \in I}$ are recorded for the same (ssid, L), create record $(\mathsf{ssid}, L, \mathsf{offs})$.

Sign Gen: On receiving $(\textsc{Sign}, \mathsf{sid}, \mathsf{ssid}, \mathsf{pk}_i, L, m)$ from $\mathcal{F}_{\mathsf{IMS}}$, retrieve the corresponding stored $\mathtt{st}_i$ and offs, then run $\sigma_i \leftarrow \mathsf{SignOn}_{\mathtt{st}_i}(\mathsf{sk}_i, m, \mathsf{offs}, L)$, then return $(\textsc{Signature}, \mathsf{sid}, \mathsf{ssid}, \sigma_i)$ to $\mathcal{F}_{\mathsf{IMS}}$.

Sign Agg: On receiving $(\textsc{aggregate}, \mathsf{sid}, L, m, \sigma_1, \ldots, \sigma_n)$ from $\mathcal{F}_{\mathsf{IMS}}$, run $\sigma = \mathsf{Agg}(\sigma_1, \ldots, \sigma_n)$, return $(\textsc{Aggregated}, \mathsf{sid}, \sigma)$ to $\mathcal{F}_{\mathsf{IMS}}$.

Verify: On receiving $(\textsc{Verify}, \mathsf{sid}, L, \mathsf{ssid}, m, \sigma)$, run $b \leftarrow \mathsf{Vrfy}(L, m, \sigma)$ and return $(\textsc{Verified}, \mathsf{sid}, b)$.

Fig. 32: Simulator $\mathcal{S}_{\mathsf{IMS}}$ interacting with $\mathcal{F}_{\mathsf{IMS}}$.

F.3 UC Security of Interactive Ordered Multi-Signature Scheme

In Fig. 33 we define π_{OMS} as a direct instantiation of OMS. To realize a protocol in the key registration model, we assume parties have access to $\mathcal{F}_{\mathsf{vReg}}$.

Protocol π_{OMS}

The protocol is parameterized by $\mathsf{OMS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{SignOff}, \mathsf{SignOn}, \mathsf{Vrfy})$ and $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\tau)$ executed by n signers $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$. Whenever an algorithm of OMS queries to a random oracle, π_{OMS} relays queries to and responses from $\mathcal{G}_{\mathsf{rpoRO}}$. We assume that pp includes a description of σ_0 and initializates the internal states $\mathtt{st}_i$ to empty strings. Signature setup, signature generation, and verification interfaces validate the input $L = (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ by retrieving a key pk'_i for $\mathcal{P}_i$ from $\mathcal{F}_{\mathsf{vReg}}$ for each $i \in [n]$ and checking $\mathsf{pk}_i = \mathsf{pk}'_i$. If validation fails, the functionality returns $(\textsc{Fail}, \mathsf{sid})$.

Key generation Upon $(\textsc{KeyGen}, \mathsf{sid})$ a signer $\mathcal{P}_i \in \mathcal{P}$ proceeds as follows:

1. If there exists a record $(\textsc{KeyRec}, \mathsf{sid}, *)$, then abort.
2. Run $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\tau; \rho)$.
3. Send $(\textsc{register}, \mathsf{sid}, \mathsf{pk}_i, \mathsf{sk}_i, \rho)$ to $\mathcal{F}_{\mathsf{vReg}}$
4. Create record $(\textsc{keyrec}, \mathsf{sk}_i, \mathsf{pk}_i)$ and output $(\textsc{KeyConf}, \mathsf{sid}, \mathsf{pk}_i)$.

Signature Setup (Offline Phase) Same as π_{IMS}.

Signature Generation (Online Phase) Upon receiving $(\textsc{SignOn}, \mathsf{sid}, \mathsf{ssid}, L, m, \sigma_{i-1})$ from a party $\mathcal{P}_i \in \mathcal{P}$:

1. If there exists **no** record $(\textsc{sigsetup}, \mathsf{ssid}, L, *, *)$, then return $(\textsc{fail}, \mathsf{sid}, \mathsf{ssid})$.
2. If $\mathcal{P}_i$ has output $(\textsc{Signature}, \mathsf{sid}, \mathsf{ssid}, *, *)$ before, return $(\textsc{fail}, \mathsf{sid}, \mathsf{ssid})$.
3. Retrieve record $(\textsc{sigsetup}, \mathsf{ssid}, L, \mathsf{offs}, \mathsf{pk}_i, \mathtt{st}_i)$
4. Run $\sigma_i \leftarrow \mathsf{SignOn}_{\mathtt{st}_i}(\mathsf{sk}_i, m, L, \mathsf{offs}, \sigma_{i-1})$.
5. If $\sigma_i \in \{\perp^*, \varepsilon\}$ then return $(\textsc{fail}, \mathsf{sid}, \mathsf{ssid})$.
6. Else, output $(\textsc{Signature}, \mathsf{sid}, \mathsf{ssid}, \sigma_i)$.

Signature Verification Same as π_{IMS}.

Fig. 33: OMS protocol. Differences from π_{IMS} are in in this font.

Theorem F.4. *Let* OMS *be an ordered multi-signature scheme (Definition 5.1) that is* $\mathsf{OMS\text{-}UF\text{-}CMA}$ *secure in the random oracle model (Definition 5.2). Then the protocol* π_{OMS} *(Figure 33) UC-realizes* $\mathcal{F}_{\mathsf{OMS}}$ *(Figure 29) in the* $(\mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{vReg}})$*-hybrid model against a static active adversary corrupting a majority of parties in* $\mathcal{P}$.

Proof. In Figure 35 we describe a PPT simulator $\mathcal{S}_{\mathsf{OMS}}$ that has black-box access to $\mathcal{A}$, and simulates π_{OMS} in a way that is indistinguishable to any efficient environment $\mathcal{Z}$. The OMS scheme used in the proof follows the syntax given in Definition 5.1. The proof closely follows that of Theorem F.3 for an unordered interactive multi-signature scheme. Let us go over each interface.

- $\textsc{KeyGen}$: The only difference is when $\mathcal{F}_{\mathsf{OMS}}$ aborts after receiving pk_i from $\mathcal{S}_{\mathsf{OMS}}$. This is negligible assuming sufficiently high min-entropy of honestly generated pk.

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

$\mathcal{F}_{\mathsf{vReg}}$ is parametrized by some key generation algorithm KeyGen (e.g., signature key pair generation) and interacts with a set of parties $\mathcal{P}$ and an ideal adversary $\mathcal{S}$ as well as a global clock $\mathcal{G}_{\mathsf{Clock}}$ as follows:

Key Registration: Upon receiving a message $(\textsc{register}, \mathsf{sid}, \mathsf{pk}, \mathsf{sk}, \mathsf{aux})$ from a party $\mathcal{P}_i \in \mathcal{P}$:

1. Send $(\textsc{Read})$ to $\mathcal{G}_{\mathsf{Clock}}$, waiting for response $(\textsc{Read}, \nu)$.
2. Use aux to run $(\mathsf{pk}', \mathsf{sk}') \leftarrow \mathsf{KeyGen}()$. If $(\mathsf{pk}', \mathsf{sk}') \neq (\mathsf{pk}, \mathsf{sk})$, abort.
3. Send $(\textsc{Registering}, \mathsf{sid}, \mathsf{pk}, \mathcal{P}_i, \nu)$ to $\mathcal{S}$. Upon receiving $(\mathsf{sid}, ok, \mathcal{P}_i)$ from $\mathcal{S}$, and if this is the first message from $\mathcal{P}_i$, then record the tuple $(\mathcal{P}_i, \mathsf{pk}, \nu)$.

Key Retrieval: Upon receiving a message $(\textsc{Retrieve}, \mathsf{sid}, \mathcal{P}_j)$ from a party $\mathcal{P}_i \in \mathcal{P}$, send message $(\textsc{Retrieve}, \mathsf{sid}, \mathcal{P}_j)$ to $\mathcal{S}$ and wait for it to return a message $(\textsc{Retrieve}, \mathsf{sid}, ok)$. Then, if there is a recorded tuple $(\mathcal{P}_j, \mathsf{pk}, \nu)$ output $(\textsc{Retrieve}, \mathsf{sid}, \mathcal{P}_j, \mathsf{pk}, \nu)$ to $\mathcal{P}_i$. Otherwise, if there is no recorded tuple, return $(\textsc{Retrieve}, \mathsf{sid}, \mathcal{P}_j, \perp)$.

Fig. 34: Functionality $\mathcal{F}_{\mathsf{vReg}}$ for Verified Key Registration, modeling the key registration oracle of [14]. Differences with $\mathcal{F}_{\mathsf{Reg}}$ are highlighted in this font. The functionality essentially corresponds to PKI with the knowledge of secret key (KOSK) requirement [13] or the key verification [6, 37] which can be realized using efficient non-interactive proof of knowledge [64].

- SignOff: Since $\mathcal{S}_{\mathsf{OMS}}$ runs $\mathsf{SignOff}$ as in π_{OMS}, there is no difference.
- SignOn: If $\mathcal{F}_{\mathsf{OMS}}$ aborts with output $(\textsc{Fail}, \mathsf{sid}, \mathsf{ssid})$ after detecting a bad sequence, then there is a potential discrepancy in the view of $\mathcal{Z}$ since π_{OMS} may proceed with output $(\textsc{Signature}, \mathsf{sid}, \mathsf{ssid}, \sigma_i)$. However, if $\mathcal{Z}$ manages to come up with an input (L, m, σ^*_{i-1}) that causes the honest execution of SignOn to output $\sigma_i \notin \{\perp^*, \varepsilon\}$, this implies $\mathcal{Z}$ created a forged so-far-aggregated signature that sets the flag win to 1 in the $\mathsf{OMS\text{-}UF\text{-}CMA}$ game (i.e. detection of bad order). That is, using such $\mathcal{Z}$ as a subroutine, one can construct a reduction $\mathcal{B}$ breaking $\mathsf{OMS\text{-}UF\text{-}CMA}$. This is done in a manner analogous to the last case of Verify.
- Verify: We consider the following cases:
  - If there is **no** corruption **and** for input (L, m) SignOn is completed with partial signatures $(\sigma_1, \ldots, \sigma_n)$: By the correctness of OMS, there is no difference in the view of $\mathcal{Z}$.
  - If for input (L, m) all honest parties $\mathcal{P}_i \notin \mathcal{C}$ complete SignOn with partial signature σ_i: In this case, $\mathcal{F}_{\mathsf{OMS}}$ asks $\mathcal{S}_{\mathsf{OMS}}$ to simulate a decision bit. Since $\mathcal{S}_{\mathsf{OMS}}$ also runs Vrfy as in π_{OMS}, there is no difference in the view of $\mathcal{Z}$.
  - If for input (L, m) some honest party $\mathcal{P}_i \notin \mathcal{C}$ has **not** completed SignOn: In this case, there is a potential discrepancy in the view of $\mathcal{Z}$ since $\mathcal{F}_{\mathsf{OMS}}$ always outputs a decision bit 0 while π_{OMS} returns the output of $\mathsf{Vrfy}(L, m, \sigma)$. However, if $\mathcal{Z}$ manages to come up with (L, m, σ) causing Vrfy to return 1, this implies $\mathcal{Z}$ created a valid forgery in the $\mathsf{OMS\text{-}UF\text{-}CMA}$

Simulator $\mathcal{S}_{\mathsf{OMS}}$

$\mathcal{S}_{\mathsf{OMS}}$ interacts with an internal copy of $\mathcal{A}$, towards which it simulates the honest parties in $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and relays all the random oracle calls requests from SignOff, SignOn, and Vrfy to the external global random oracle functionality $\mathcal{G}_{\mathsf{rpoRO}}$. Whenever $\mathcal{A}$ queries $\mathcal{F}_{\mathsf{vReg}}$, $\mathcal{S}_{\mathsf{OMS}}$ executes the code of $\mathcal{F}_{\mathsf{vReg}}$ to simulate its response.

KeyGen: On receiving $(\textsc{KeyGen}, \mathsf{sid}, \mathcal{P}_i)$ from $\mathcal{F}_{\mathsf{OMS}}$, run KeyGen to obtain the key pair $(\mathsf{sk}_i, \mathsf{pk}_i)$. Return $(\textsc{KeyConf}, \mathsf{sid}, \mathsf{pk}_i)$ to $\mathcal{F}_{\mathsf{OMS}}$. Upon receiving a key registration request from corrupt $\mathcal{P}_i$, if the check passes, send $(\textsc{KeyGen}, \mathsf{sid}, \mathcal{P}_i, \mathsf{pk}_i)$ to $\mathcal{F}_{\mathsf{OMS}}$.
Sign Setup: Same as $\mathcal{S}_{\mathsf{IMS}}$.
Sign Gen: On receiving $(\textsc{Sign}, \mathsf{sid}, \mathsf{ssid}, \mathsf{pk}_i, L, m, \sigma_{i-1})$ from $\mathcal{F}_{\mathsf{OMS}}$, retrieve the corresponding stored $\mathtt{st}_i$ and offs, then run $\sigma_i \leftarrow \mathsf{SignOn}_{\mathtt{st}_i}(\mathsf{sk}_i, m, \mathsf{offs}, L, , \sigma_{i-1})$, then return $(\textsc{Signature}, \mathsf{sid}, \mathsf{ssid}, \sigma_i)$ to $\mathcal{F}_{\mathsf{OMS}}$.
Verify: Same as $\mathcal{S}_{\mathsf{IMS}}$.

Fig. 35: Simulator $\mathcal{S}_{\mathsf{OMS}}$ interacting with $\mathcal{F}_{\mathsf{OMS}}$.

game. That is, using such $\mathcal{Z}$ as a subroutine one can construct a reduction $\mathcal{B}$ breaking OMS-UF-CMA. On receiving public parameters pp, $\mathcal{B}$ with access to OKeyReg, OSignOff, OSignOn and the random oracle H, plays the role of $\mathcal{F}_{\mathsf{OMS}}$, $\mathcal{S}_{\mathsf{OMS}}$, and $\mathcal{G}_{\mathsf{rpoRO}}$ and proceeds as follows.

* Upon receiving a key generation request for $\mathcal{P}_i$: If $\mathcal{P}_i$ is honest, then query OKeyReg with $uid = i$ and $\mathsf{aux} = \varepsilon$ to retrieve a key pk_i. If $\mathcal{P}_i$ is corrupted and queries $\mathcal{F}_{\mathsf{vReg}}$ with $(\textsc{register}, \mathsf{sid}, \mathsf{pk}_i, \mathsf{sk}_i, \mathsf{aux})$, then query OKeyReg with $uid = i$ and aux to register a corrupted key.
* Whenever $\mathcal{Z}$ makes a query to $\mathcal{G}_{\mathsf{rpoRO}}$, relay queries and responses to and responses from H.
* Whenever $\mathcal{S}_{\mathsf{OMS}}$ is prompt with a Join command for party $\mathcal{P}_i \notin \mathcal{C}$, query OSignOff with ssid to receive off_i and forwards it to a copy of the adversary $\mathcal{A}$. Otherwise, Join command is handled as in $\mathcal{S}_{\mathsf{OMS}}$.
* Whenever $\mathcal{S}_{\mathsf{OMS}}$ is asked to run Sign command for party $\mathcal{P}_i \notin \mathcal{C}$, query OSignOn with input $(\mathsf{ssid}, L, m, \mathsf{offs}, \sigma_{i-1})$ to receive σ_i. Otherwise, Sign command is handled as in $\mathcal{S}_{\mathsf{OMS}}$.
* If $\mathcal{Z}$ outputs (L, m, σ) causing $\mathsf{Vrfy}(L, m, \sigma) = 1$ while $\exists \mathsf{pk}_{i^*} \in L$ such that $\mathcal{P}_{i^*} \notin \mathcal{C}$ and (L, m) has never been queried to OSignOn, forward this tuple to the IMS-UF-CMA game.

In this way, the reduction $\mathcal{B}$ succeeds in winning the OMS-UF-CMA game. Hence, the advantage of $\mathcal{B}$ is non-negligible if $\mathcal{Z}$ finds inconsistency of verification in real and ideal executions with non-negligible probability. □

F.4 Integrating $\mathcal{F}_{\mathsf{OMS}}$ into $\pi_{\mathsf{Multi-SCD}}$

We now provide a variant of $\pi_{\mathsf{Multi-SCD}}$ that realizes $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ using $\mathcal{F}_{\mathsf{OMS}}$. It can be found in Figure 36. Our protocol is a simplified version where the parties run

the preprocessing for the next signature during the current signature round. Of course, this can be preprocessed during idle time instead.

Theorem F.5. *The protocol* $\pi_{\mathsf{Multi-SCD}_{\mathsf{OMS}}}$ *UC-securely implements* $\mathcal{F}_{\mathsf{SCD}}^{\mathsf{f}_\Delta}$ *in the* $\mathcal{G}_{\mathsf{Clock}}, \mathcal{F}_{\mathsf{OMS}}, \mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_\Delta}$*-hybrid model with security against any adversary actively corrupting up to* $k = n - 1$ *parties with permissible delay function given by* delays.

The proof for the theorem follows almost verbatim from the proof of Theorem 4.4 in Appendix B.4. While there, an honest party $\mathcal{P}_i$ would only sign if isP is true and all previous parties signed the same message m, t, it now always calls $\mathcal{F}_{\mathsf{OMS}}$ to create a signature if isP is true. But $\mathcal{F}_{\mathsf{OMS}}$ only creates a signature if honest parties before $\mathcal{P}_i$ have signed m, t in correct order as specified by L, in which case the message m, t must have traveled between them as required as they will always use the respective delay channels for communication which guarantees delay. It is immediate that the preprocessing by calls to **Signature Setup (Offline Phase)** always succeeds if parties act honestly. An adversary may not participate in the offline phase for some ssid, but in that case the signing for this ssid will simply fail.

Protocol $\pi_{\mathsf{Multi-SCD}}$

This protocol is executed by a sender $\mathcal{P}_1$, a set of intermediate parties $\mathcal{P}_2, \dots, \mathcal{P}_{n-1}$ and a receiver $\mathcal{P}_n$, as well as a set of verifiers $\mathcal{V}$ interacting with each other and with $\mathcal{G}_{\mathsf{Clock}}, \mathcal{F}_{\mathsf{OMS}}$. Each pair $\mathcal{P}_i, \mathcal{P}_{i+1}$ is connected by $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,i}}$. Let $L = (\mathcal{P}_1, \dots, \mathcal{P}_n)$ In every step, the activated party sends $(\textsc{Read})$ to $\mathcal{G}_{\mathsf{Clock}}$ to obtain $(\textsc{Read}, \bar{t})$. Each party has two counters $\mathsf{ssid}_{\mathtt{off}}$ and $\mathsf{ssid}_{\mathtt{on}}$ initially 0.

Setup: Upon first activation, each $\mathcal{P}_i$ proceeds as follows:

1. Send $(\textsc{KeyGen}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{OMS}}$ and obtain $(\textsc{KeyGen}, \mathsf{sid}, \mathsf{pk}_i)$. Then send $(\textsc{Ready}, \mathsf{sid}, i)$ to each other party.
2. Upon having received $(\textsc{Ready}, \mathsf{sid}, j)$ from every other $\mathcal{P}_j$ send $(\textsc{SignOff}, \mathsf{sid}, 0, L)$ to $\mathcal{F}_{\mathsf{OMS}}$. If the functionality outputs $(\textsc{Fail}, \mathsf{sid}, 0)$ then abort. Otherwise wait for the message $(\textsc{SignedOff}, \mathsf{sid}, 0)$. Upon receiving it, increase $\mathsf{ssid}_{\mathtt{off}}$ by 1.

Send: Upon receiving first input $(\textsc{Send}, \mathsf{sid}, m)$ for $\bar{t}$, $\mathcal{P}_1$ proceeds as follows:

1. Check if $\mathsf{ssid}_{\mathtt{off}} > \mathsf{ssid}_{\mathtt{on}}$. Otherwise, abort.
2. Send $(\textsc{Sign}, \mathsf{sid}, \mathsf{ssid}_{\mathtt{on}}, L, (m, \bar{t}), \varepsilon)$ to $\mathcal{F}_{\mathsf{OMS}}$. If $\mathcal{F}_{\mathsf{OMS}}$ outputs $(\textsc{Fail}, \mathsf{sid})$ for this $\mathsf{ssid}_{\mathtt{on}}$ then abort. Otherwise, upon receiving $(\textsc{Signature}, \mathsf{sid}, (\mathsf{ssid}_{\mathtt{on}}, \sigma_1))$ send $(\textsc{Send}, \mathsf{sid}, (m, \bar{t}, \sigma_1, \mathsf{ssid}_{\mathtt{on}}))$ to $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,1}}$ and increase $\mathsf{ssid}_{\mathtt{on}}$ by 1.
3. Send $(\textsc{SignOff}, \mathsf{sid}, \mathsf{ssid}_{\mathtt{off}}, L)$ to $\mathcal{F}_{\mathsf{OMS}}$. If $\mathcal{F}_{\mathsf{OMS}}$ outputs the message $(\textsc{SignedOff}, \mathsf{sid}, \mathsf{ssid}_{\mathtt{off}})$. Upon receiving it, increase $\mathsf{ssid}_{\mathtt{off}}$ by 1.

Receive: Upon receiving $(\textsc{Rec}, \mathsf{sid})$, $\mathcal{P}_n$ sends $(\textsc{Rec}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,n-1}}$ and proceeds as follows for the first $(\textsc{Sent}, \mathsf{sid}, (m, t, \sigma_{n-1}, \mathsf{ssid}'), t')$ received from $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,n-1}}$:

1. Check if $\mathsf{ssid}_{\mathtt{off}} > \mathsf{ssid}'$, otherwise abort.
2. Check if $\mathsf{isP}(t, \mathsf{f}_{\Delta,1}, \dots, \mathsf{f}_{\Delta,n-2}, t')$, otherwise output $(\textsc{NoProof}, \mathsf{sid})$.
3. Send $(\textsc{Sign}, \mathsf{sid}, \mathsf{ssid}', L, (m, \bar{t}), \sigma_{n-1})$ to $\mathcal{F}_{\mathsf{OMS}}$. If $\mathcal{F}_{\mathsf{OMS}}$ outputs $(\textsc{Fail}, \mathsf{sid})$ for this ssid' then output $(\textsc{NoProof}, \mathsf{sid})$. Otherwise, upon receiving $(\textsc{Signature}, \mathsf{sid}, (\mathsf{ssid}', \sigma_n)$ output $(\textsc{Sent}, \mathsf{sid}, m, t, \bar{t} - t, \sigma_n)$.
4. Send $(\textsc{SignOff}, \mathsf{sid}, \mathsf{ssid}_{\mathtt{off}}, L)$ to $\mathcal{F}_{\mathsf{OMS}}$. If $\mathcal{F}_{\mathsf{OMS}}$ outputs the message $(\textsc{SignedOff}, \mathsf{sid}, \mathsf{ssid}_{\mathtt{off}})$ then increase $\mathsf{ssid}_{\mathtt{off}}$ by 1.

Verify: Upon receiving $(\textsc{verify}, \mathsf{sid}, m, t, \Delta, \pi_{\mathsf{1o}})$, $\mathcal{V}_i \in \mathcal{V}$ proceeds as follows:

1. Check that $t + \Delta \geq \bar{t}$ and $\mathsf{isP}(t, \mathsf{f}_{\Delta,1}, \dots, \mathsf{f}_{\Delta,n}, t + \Delta)$ is true.
2. Send $(\textsc{Verify}, \mathsf{sid}, L, (m, t), \pi_{\mathsf{1o}})$ to $\mathcal{F}_{\mathsf{OMS}}$, obtaining $(\textsc{Verified}, \mathsf{sid}, f)$. Check that $f = 1$.
3. If all checks pass set $b = 1$, else $b = 0$. Output $(\textsc{verified}, \mathsf{sid}, m, t, \Delta, \pi_{\mathsf{1o}}, b)$.

Tick: Proceed as follows and then send $(\textsc{Update})$ to $\mathcal{G}_{\mathsf{Clock}}$.

1. Each $\mathcal{P}_i \in \{\mathcal{P}_2, \dots, \mathcal{P}_{n-1}\}$ sends $(\textsc{Rec}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,i-1}}$. If $\mathcal{P}_i$ obtains $(\textsc{Rec}, \mathsf{sid}, (m, t, \sigma_{i-1}, \mathsf{ssid}'), t_{i-1})$ the first time for t then it proceeds with the remaining steps.
2. Check if $\mathsf{ssid}_{\mathtt{off}} > \mathsf{ssid}'$, otherwise abort.
3. Check if $\mathsf{isP}(t, \mathsf{f}_{\Delta,1}, \dots, \mathsf{f}_{\Delta,i-2}, t_{i-1})$ is true. If so, then end $(\textsc{Sign}, \mathsf{sid}, \mathsf{ssid}', L, (m, \bar{t}), \sigma_{i-1})$ to $\mathcal{F}_{\mathsf{OMS}}$. If $\mathcal{F}_{\mathsf{OMS}}$ outputs $(\textsc{Fail}, \mathsf{sid})$ for this ssid' then abort. Otherwise, upon receiving $(\textsc{Signature}, \mathsf{sid}, (\mathsf{ssid}', \sigma_i)$ send $(\textsc{Send}, \mathsf{sid}, (m, t, \sigma_i, \mathsf{ssid}'))$ to $\mathcal{F}_{\mathsf{mdmt}}^{\mathsf{f}_{\Delta,i}}$.
4. Send $(\textsc{SignOff}, \mathsf{sid}, \mathsf{ssid}_{\mathtt{off}}, L)$ to $\mathcal{F}_{\mathsf{OMS}}$. If $\mathcal{F}_{\mathsf{OMS}}$ outputs the message $(\textsc{SignedOff}, \mathsf{sid}, \mathsf{ssid}_{\mathtt{off}})$ then increase $\mathsf{ssid}_{\mathtt{off}}$ by 1.

Fig. 36: Protocol $\pi_{\mathsf{Multi-SCD}_{\mathsf{OMS}}}$ composing two π_{SCD} instances into one.