Consistency-or-Die: Consistency for Key Transparency

Joakim Brorsson*§, Elena Pagnin†, Bernardo David‡ and Paul Stankovski Wagner§
*Hyker
†Chalmers University of Technology
‡IT University of Copenhagen
§Lund University

***Abstract*—This paper proposes a new consistency protocol that protects a key transparency log against split-view attacks and – contrary to all previous work – does not to rely on small committees of known external auditors, or out-of-band channels, or blockchains (full broadcast systems).**

Our approach is to use a mechanism for cryptographically selecting a small committee of random and initially undisclosed users, which are then tasked to endorse the current view of the log. The name of our protocol, Consistency-or-Die (CoD), reflects that users are guaranteed to know if they are in a consistent state or not, and upon spotting an inconsistency in the key transparency log, users stop using this resource and become inactive (die). CoD relies on well-established cryptographic building blocks, such as verifiable random functions and key-evolving signatures, for which lightweight constructions exist. We provide a novel statistical analysis for identifying optimal *quorum* sizes (minimal number of endorsers for a view) for various security levels and percentages of malicious users.

Our experiments support that CoD is practical and can run in the background on mid-tier smart phones, for large-scale systems with billions of users.

1. Introduction

A backbone to securing the web is the existence of a trustworthy Certificate Authority (CA) infrastructure. The main role of a web CA is to issue digital certificates to validate the authenticity of websites. These certificates help establish secure connections between a user's browser and a website, ensuring that sensitive information is encrypted and transmitted securely. Additionally, CAs are expected to verify the identity of website owners and help prevent fraudulent activities. Unfortunately, we have witnessed several incidents [1] where incorrect or malicious certificates have successfully been used. For example, Symantec was caught wrongfully issuing a certificate for google.com [2], and DigiNotar was fully compromised by an unknown attacker that issued over 500 fake certificates [3], which were then used for spying on Iranian citizens. These incidents are due to placing *too much trust* in the CA infrastructure.

To remedy this situation, transparency logs for the web, called Certificate Transparency logs (CT) [4], have been deployed to ensure correct serving of TLS certificates. Many browsers currently mandate their use and thus reduce the trust that needs to be placed in CAs by making them transparent.

Intuitively, the transparency in transparency logs comes from publicly recording all certificates issued by a CA. This allows for greater visibility and accountability in the issuance of certificates, helping to detect any unauthorized or fraudulent certificates. More specifically, transparency logs are label-value data structures which are publicly verifiable to be *append-only*, allowing no data to be deleted or altered, and *consistent*, *i.e.*, serving all users the same view of the current state of the data structure. These two properties ensure that the same value (certificate) is delivered in response to all queries for a specific label.

Recently, transparency logs have been applied to serving public keys for end-to-end encryption messaging apps [5], [6], [7], [8], [9], and deployed in mainstream apps such as WhatsApp [10], iMessage [11] and Zoom [12]. In this case, the technique is called Key Transparency (KT) logs.

Even though CT and KT both aim to detect potential attacks connected to cryptographic material, they have a few core differences. While CT logs are designed for detecting and preventing issuance of fraudulent TLS certificates, KT logs focus on providing a secure and transparent way to manage and distribute cryptographic keys.

1.1. State-of-the-Art and Current Issues

Key Transparency research consists of protocols for ensuring *append-only*, and protocols for ensuring *consistency*. Protocols for ensuring *append-only* [5], [6], [7], [8], [9], *e.g.* Verifiable Key Directories (VKD) [7], [8], constitute the largest body of work, and these protocols have in recent years reached production level maturity. However, protocols for *consistency* have received much less attention, and current methods [8], [13], [14], [15], [16] are unsatisfactory as they provide weak consistency guarantees.

In light of this, it is imperative to develop better consistency protocols. A KT log with weak consistency guarantees cannot protect against split-view attacks (which break the consistency property) and is thus of limited value for ensuring that the correct public keys are served. The need is urgent. KT logs with no or weak consistency guarantees have already been deployed in high profile systems (see

Section 1.1.2). Meanwhile a number of split-view incidents in CT logs have been detected [17], [18], [19]. Such split-views risk going *undetected* in current proposals for KT.

1.1.1. Academic Proposals. There are a three known methods for consistency in KT gossip protocols, blockchains and designated witnesses. We here give an overview of these approaches, and provide further details in Appendix C.

Gossip protocols [20], [21], [22] achieve consistency by having users gossip over the state of the log using Out-Of-Band (OOB) channels, where the service provider cannot suppress messages. Gossiping essentially consists of exchanging messages that endorse (or oppose) a certain view. While this approach works for CT, it is problematic for KT systems, since OOB channels are unworkable in real-world KT scenarios (matching QR codes in person does not scale well to a user base of, say, a billion users).

Blockchains have been suggested [13], [14], [23] as an alternative to gossip protocols. By using a blockchain, consistency comes for free since each record in a blockchain is cryptographically linked to previous records, in a sequential and immutable manner. However, relying on this technology implicitly implies accepting the costs and assumptions that come with implementing it. We consider the costs, *e.g.*, end users running full blockchain nodes, unrealistic in most mainstream scenarios such as KT for large scale instant messaging apps. Indeed, none of the existing KT deployments have chosen to use blockchains for consistency.

External Committees of Consistency Auditors [8], [24] is the most recent alternative for achieving efficient and scalable consistency in KT. The idea is to appoint an external committee of auditors, where each committee member endorses its view, and where at last two thirds of the members are assumed to be honest. A user can be assured of consistency of its own view if there is a large enough set of committee members (other users) who agree with this view. Security demands that such a committee be large and untargetable, otherwise an adversary could simply corrupt all committee members. However, for efficiency and practical reasons, such a committee should at the same time be small. First of all, because users' CPU and network overhead grow with the committee size. Secondly, because finding a large number of trustworthy external parties can be difficult in practice. Finding the one-size that fits all is challenging, and the current suggestion (10-50 committee members [8]) falls short on meaningfully safeguarding potential high-impact targets such as WhatsApp, iMessage or Zoom.

In summary, neither gossip, nor blockchains, nor external committees of consistency auditors are satisfying solutions to ensure consistency for key transparency.

We note that concurrent work [16] explores another direction of ensuring consistency for KT which does not use external parties for auditing consistency. This is achieved by weakening the consistency guarantees of KT, so that the protocol only ensures that split-views are detected by *either* the party who queries for a key, *or* the key owner.

1.1.2. Practical Deployments. The drawbacks of state-of-the-art academic proposals for Key Transparency consistency impact the security of existing KT systems. Indeed, none of the systems we consider currently provide a satisfactory consistency guarantee.

The consistency guarantees in *iMessage* come from gossip protocols [11], which due to the lack of proper OOB channels give weak security guarantees. *Zoom* state that they "will partner with independent external auditors" [12] for ensuring the consistency guarantee, *i.e.*, they envision a future use of designated witnesses but lack a solution at present. *WhatsApp*'s approach is to use an S3 bucket with a 5-year retention period [10]. This means that consistency *depends on a central third party*, which is clearly undesirable since KT is designed to avoid trust in third parties.

1.2. A Novel Approach to KT Consistency

As mentioned in Section 1.1.1, consistency comes trivially if one is willing to accept the costs and assumptions associated with blockchains [25], [26], or more generally broadcast systems [27].[1] This is because by design such systems enjoy, among others, two distinctive features;

(1) *Consistency*: each party can verify that they have a view of an ordered list of data that is identical to the view of all other parties,
(2) *Guaranteed Output Delivery (GOD)*: each party has guaranteed access to any update of the list at specific time intervals.

For KT, feature (1) suffices and (2) is superfluous. Hence one could attempt to dissect a blockchain as a broadcast system, dispose of any machinery related to (2), and extract only what provides (1), in the hope of reducing costs and assumptions. Unfortunately, consistency and GOD are not easily separable – GOD enables consistency.

The next natural attempt is to design a novel mechanism that only provides consistency, and may not have guaranteed output delivery. This is precisely the aim of gossip protocols and protocols relying on external committees, with the shortcomings for KT discussed in Section 1.1.1.

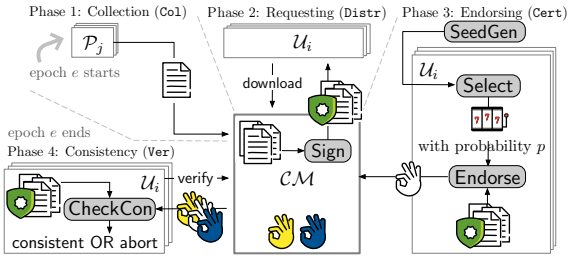


Figure 1: Core components and phases of CoD.

1. Although blockchains are not formally broadcast systems, they do implicitly provide similar guarantees, and are often used as broadcast channels in academic literature.

Overview of our Techniques. To guarantee consistency, we employ a mechanism similar to external committees where each member endorses its view, but with two significant novelties. The first novelty is that the committee is neither external nor fixed. Instead, for each epoch, a fresh set of *initially undisclosed* users is *randomly* selected from all users in the system. This allows for distributing trust across the entire set of users rather than trusting in a small external committee, which a powerful adversary could easily corrupt. The second novelty is that the data which constitutes the view, *e.g.* a commitment to the current state by the identity provider, needs to be delivered *before the identities of the committee members become known*. This makes users tasked with endorsing a view *untargetable* for an adversary.

Figure 1 illustrates the core mechanisms of our approach. Time in the protocol proceeds in consecutive epochs. Each epoch is divided into four non-overlapping phases. In the first phase (`Col`), the channel maintainer $\mathcal{CM}$ collects data from publishing parties $\mathcal{P}_j$. In the second phase (`Distr`), $\mathcal{CM}$ organizes the collected data into what we call *the view* for the current epoch, and distributes the signed view as a reply to download requests from users $\mathcal{U}_i$. In the third phase (`Cert`), users independently run a mechanism for randomly selecting a committee of endorsers that will certify the view of the epoch in a way that is cryptographically sound and does not reveal the identity of committee members until they publish an endorsement to the $\mathcal{CM}$. We realize this in a similar way as in [26], [28], [29]: users get a common seed and evaluate a Verifiable Random Function (VRF) (see Definition 2.1) on the seed – using their own secret VRF key. If the VRF output is below a threshold value, they endorse the view they received in the previous phase by signing it and sending it to $\mathcal{CM}$ along with the VRF proof for public audit of their selection as the current epoch's endorsers. This mechanism is essential in our security analysis. In the fourth and final phase (`Ver`), each user verifies the consistency of the view received in the second phase. This entails downloading a set of endorsements from the $\mathcal{CM}$, and verifying that it contains a *quorum*. In brief, the quorum is a number identifying the minimum number of independent endorsements needed to ensure consistency of one's view. This entails checking; (1) the VRF outputs (to ensure cryptographically sound sampling of the endorsers), (2) the endorsements (that should support the user's view), and (3) that there is at least a quorum of endorsements. If the quorum check fails, consistency is not guaranteed; thus the user aborts and ceases to participate in the protocol for all future epochs, *i.e.*, it "dies". We provide a new statistical analysis (see Section 5) of optimal quorum size for our setting that improves on the state of the art for selecting anonymous committees [30]. Techniques from [31] do not apply, as a corrupted $\mathcal{CM}$ arbitrarily controls communication (including dropping messages).

In contrast to gossip protocols, our approach enables immediate detection of split-view attacks by the user itself, provides strong detection guarantees where *all* split-views are detected, and is compatible with the in-band communication pattern where an adversarial central party is present in KT. Compared to external committee consistency protocols, it guarantees consistency assuming honest majority among all users (*i.e.*, in billions of users in the case of WhatsApp) rather than among a small set of targetable external auditors.

1.3. Contributions

Motivated by the lack of satisfactory solutions for providing consistency in KT, we propose a novel protocol that enables each user to autonomously detect split-view attacks, and in such a case immediately abort the protocol (die). We name our protocol Consistency-or-Die (CoD). CoD provides consistency without assuming broadcast, OOB channels, or a trusted committee of external auditors; it only relies on well-established, efficiently implemented cryptographic tools, and a novel statistical analysis that justifies parameters choices that make CoD attractive for large scale systems. In detail:

We *formalize the notion of a consistency protocol for KT* (Section 3), essentially consistent broadcast [32] (*i.e.*, complete and consistent but without guaranteed output delivery) where all communication is done via a central party.

We present CoD (Section 4 and Protocol 1), an *efficient realization of a consistency protocol* that combines in a black box way: an unforgeable signature scheme, a forward-secure signature scheme, a verifiable random function with unpredictability under malicious key generation, a randomness beacon, and a verifiable key directory (under monitoring). Our realization relies on a one-time trusted setup to produce a common reference string (in line with the standard KT trust model).

We discuss how to efficiently instantiate CoD using existing cryptographic schemes. Our efficiency estimates support that CoD is *practical* and can run in the background on mid-tier smart phones, for large scale systems with billions of users (Section 4.4).

We provide an *extensive mathematical analysis* of how to set the quorum in our CoD protocol (Section 5). Our analysis leverages results from combinatorics that have not been used in previous work [30], [31] and reveals a concrete mechanism to identify optimal *quorum* sizes. The quorum is a key concept leveraged by our construction and essentially sets the (minimal) number of endorsements required for considering one's view to be consistent with the view of the majority of honest endorsers.

As a minor contribution, we show to apply our mathematical analysis to reliable broadcast systems (*with* guaranteed output delivery), and draw connections between split-view attacks in settings without GOD, and honest majority committees assuming reliable broadcast (Section 5.1).

2. Preliminaries

VRFs A Verifiable Random Function [33] (VRF) allows a prover to, given a seed, output a value which is verifiably random. It is defined as in Definition 2.1.

Definition 2.1 (Verifiable Random Function)**.** *A VRF consists of the following procedures:*

■ $\mathsf{KeyGen}(\lambda) \rightarrow (\mathsf{sk}, \mathsf{pk})$*: on input a security parameter* λ *outputs a secret key* sk *and a public key* pk*.*

■ $\mathsf{Prove}(\mathsf{sk}, \mathsf{s}) \rightarrow (r, \pi_r)$*: on input a secret key* sk *and a seed* s*, outputs a value* r *and a proof* π_r*.*

■ $\mathsf{Verify}(\mathsf{pk}, \mathsf{s}, r, \pi_r) \rightarrow 0/1$*: on input a public key* pk*, a seed* s*, a value* r*, and a proof* π_r*, outputs 1 if the proof verifies, and 0 otherwise.*

A VRF is secure if it has *uniqueness*, *i.e.*, for each (pk, π_r) there is a single r which will verify, it has *provability*, *i.e.* a correctly produced r and π_r will always verify, and it has *pseudorandomness*, *i.e.* a computationally bounded adversary has a negligible advantage in distinguishing r from randomness. For formal definitions, see [33].

The above properties which are the common ones required for a VRF however do not account for malicious key generation, which can impact the output of the VRF and skew the distribution of r. Since wee must account for this, we require a VRF with *unpredictability under malicious key generation*. Informally, this guarantees that regardless of how the public key was generated, the output of the VRF must be random. For formal definitions, see [28].

Verifiable Key Directories (VKD) A VKD [7], [8] is a centrally maintained verifiably append-only label-value directory, intended for secure lookup of public keys. The syntax of a full-fledged VKD is out of scope for our paper, it suffices here to note that a VKD outputs a digest dig_e of the state of a directory Dir at time e. A VKD provides completeness and soundness, which informally means the following security properties.

Membership: Relative to a digest dig_e, a VKD ensures that proofs of membership for a value v will only verify if $v \in \mathsf{Dir}$ at time e.

Append-only: Relative to 2 consecutive digests dig_{e-1} and dig_e, a VKD ensures proofs of append-only will only verify if all changes to Dir between time $e-1$ and time e were append only (*i.e.* no deletes).

Consistency (relative): Relative to a digest dig_e, a VKD ensures that any query for a specific label a has a unique value which satisfies proofs of consistency.

We emphasize that the consistency guarantee of a VKD is *relative* to a digest. A separate protocol for consistent distribution of digests is needed, as discussed in Section 1.1).

We refer to [7], [8] for a definition of VKD syntax and for formal definitions of security.

Key Evolving Signatures and Forward Security A key evolving signature scheme (KES) [34] is a signature scheme which allows the signer to evolve the secret key each time it signs a message.

Definition 2.2 (Key Evolving Signatures)**.** *A KES scheme consists of the following procedures:*

■ $\mathsf{KeyGen}(\lambda, \mathsf{Max}) \rightarrow (\mathsf{sk}, \mathsf{pk})$*: on input a security parameter* λ *and a maximum number of allowed signatures for a secret key* Max*, outputs a secret key* sk *and a public key* pk*.*

■ $\mathsf{KeyUpdate}(\mathsf{sk}, i) \rightarrow \mathsf{sk}_{i+1}$*/fail: on input a secret key* sk *and an index* i *outputs the evolved signing key* sk_{i+1} *if* sk *is a valid secret key for index* i *and* $i < \mathsf{Max}$*, or fail otherwise.*

■ $\mathsf{Sign}(\mathsf{sk}, m, i) \rightarrow \sigma_i/$*fail: on input a secret key* sk*, a message* m*, and an index* i*, outputs a signature* σ_i *if* sk *is a valid secret key for index* i *and* $i < \mathsf{Max}$*, or fail otherwise.*

■ $\mathsf{Verify}(\mathsf{pk}, m, \sigma_i, i) \rightarrow 0/1$*: on input a public key* pk*, a message* m*, a signature* σ_i*, and an index* i*, outputs 1 if* σ_i *is a valid signature on* m *under* pk *for index* i*, and 0 otherwise.*

We require a KES existentially unforgeable under adaptive chosen message attacks (EUF-CMA). We also require that the KES has forward security (FSSIG), which intuitively means that the an (adaptive) adversary which is given the current time secret key of a user sk_i, cannot forge a signature for keys which have been evolved (any $\mathsf{sk}_j, j < i$). For a full definition of EUF-CMA and FSSIG for KES, see [34].

3. Introducing CoD

CoD is a lightweight version of a broadcast channel that can be realized in a concretely efficient way, and without relying on complex blockchain ecosystems. Efficiency comes by replacing the distributed system (typical in blockchains) with an *untrusted* central party (typical in transparency logs), and by relaxing some properties

1) CoD achieves the same *consistency* guarantees as broadcast,

2) contrasting broadcast, CoD makes no guarantees on message delivery, *i.e.*, it provides neither censorship resistance nor guaranteed output delivery (as is the case with transparency logs with communication via a central party),

3) instead, CoD provides *abort* upon discovery of an inconsistent view (honest parties that discover an inconsistency will abort and become inactive).

3.1. Model

3.1.1. Entities, Roles, and States. A CoD system consists of the following entities (input/output processes):

■ A channel maintainer: denoted $\mathcal{CM}$, which maintains a list of published data, and answers queries for the data.

■ Users: denoted $\mathcal{U}_i$, which download data from the $\mathcal{CM}$. The set of users is allowed to evolve over time.

■ External parties: covering specific roles.

There are two roles that some of the entities may take:

■ `publisher`: any number of users and external parties may take this role. This entitles the party to send data to $\mathcal{CM}$. In our CoD protocol, we will assume a special publisher (external entity) called the Identity Provider (IdP).

■ `endorser`: the protocol assigns the role of `endorser` to a small subset of the users. Each such user is tasked with endorsing the view they obtain from the $\mathcal{CM}$ and thus help audit view consistency across users. At any point in time honest users are in one of two possible states: `consistent`: meaning that their view equals the view of

all other active honest users who are in a `consistent` state; or `abort`: meaning that the user discovered a potential split-view attack.

3.1.2. Infrastructure. The entities in the system communicate over a star-topology network, with $\mathcal{CM}$ acting as the central node, while users and external parties are edge nodes. No direct user-to-user channels are assumed. The $\mathcal{CM}$ is assumed to be online (practically) all the time. The CoD system can tolerate a varying fraction of users being offline.

3.1.3. Time, Epochs and Phases. CoD does not include any notion of time *per se*, as natively it is an asynchronous system. However, we do need a notion of time to decide whether a message is simply delayed or completely suppressed. We do so by using *epochs* as an abstract time-unit, and collect and distribute data in epoch-batches.

Epochs progress in an incremental way. Each epoch is divided into four phases: *data collection, data distribution, data certification*, and *data verification*. The syntax of a CoD protocol follows these four phases. First, during the data collection phase `Col`, publishers submit data to the channel maintainer $\mathcal{CM}$ which collects these messages into a list View_e. Second, during the data distribution phase `Distr`, $\mathcal{CM}$ distributes View_e to users. Third, during the data certification phase `Cert`, a special subset of users act as endorsers by sending an endorsement of their view, which $\mathcal{CM}$ collects into End_e. Fourth and finally, during the data verification phase `Ver`, all users query for End_e and verify their view against the endorsements in End_e, so that all honest parties in the consistent state have the same view.

3.1.4. Adversary. We consider an adversary $\mathcal{A}$ that is malicious, adaptive, and rushing; $\mathcal{A}$ is modelled as a probabilistic polynomial-time algorithm. The adversary has full control over the communication network, and can corrupt a fraction f_M of active users in the system, the $\mathcal{CM}$, and any external entity acting as a publisher (e.g., the IdP). In particular, $\mathcal{A}$ intercepts all protocol messages and can arbitrarily suppress messages to and from $\mathcal{CM}$. To better emulate reality, our model assumes a fraction f_I of inactive users at each epoch. Inactive users are honest users which are unresponsive due to being blocked by the adversary or simply being offline. They do not count as malicious.

3.2. Trusted Setup

A protocol that implements a CoD channel relies on a trusted setup (which can be bootstrapped, as in our CoD construction, see Protocol 1). Since the system relies on a central untrusted party (the channel maintainer $\mathcal{CM}$), and it does not assume peer-to-peer connection among users, there needs to be a way to kick-start the system and tell all entities involved who the users are, what (initial) key material they have, etc. The trusted setup is meant to output a CRS (a public common reference string) that provides such information, and it takes care of:

1) **Registering and generating keys** for all parties in the system. This entails: associating to each party an entity type, a public and unique identifier ID_i, and generating the necessary key material for that entity, e.g., key pairs $(\mathsf{sk}_i, \mathsf{pk}_i)$ for relevant building blocks in the protocol. Notably, the *trusted* setup ensures only the party itself knows sk_i.
2) Including the list of **entity identifiers and corresponding public keys** in the CRS. In detail, the CRS contains a description of $\mathcal{PK} = \{(\mathsf{ID}_{\mathcal{CM}}, \mathsf{pk}_{\mathcal{CM}}), (\mathsf{ID}_1, \mathsf{pk}_1), (\mathsf{ID}_2, \mathsf{pk}_2), \ldots\}$.
3) Including a **security parameter** λ, and the **total number of users** in the system N, in the CRS.
4) Optionally, include any **additional auxiliary material** `aux` to kickstart the protocol in the CRS. In the case of our CoD protocol, this entails a random seed s_0 (to bootstrap `endorser` selection), a digest dig_0 of $\mathcal{PK}$, a pair of integers needed for security (i.e., T, k), public parameters of the building blocks, and any additional information needed to run the procedures.

3.3. Syntax

Definition 3.1 provides the syntax of a CoD protocol. To reflect the nature of most centralized systems, all interactive procedures are initiated by a user or publisher. We denote interactive procedures as:

$$Name : \left\langle \begin{array}{c} Initator(input) \\ Interlocutor(input) \end{array} \right\rangle \mapsto [Entity : local\ output].$$

Definition 3.1 (CoD)**.** *A* CoD *protocol for a given* CRS *output by a trusted setup is defined by the following procedures (the* CRS *is implicitly available to all entities):*

$$\mathsf{Send} : \left\langle \begin{array}{c} \mathcal{P}(\mathsf{data}, e) \\ \mathcal{CM}(\mathsf{sk}_{\mathcal{CM}}, \mathsf{View}_{e'}, e') \end{array} \right\rangle \mapsto [\mathcal{CM} : (\mathsf{View}_{e'}, \mathsf{Tkn}_{e'})]$$

The Send *procedure is interactive and can be run multiple times during phase* `Col`*. It is initiated by a publisher $\mathcal{P}$, who sends a* $(\textsc{send}, \mathsf{data})$ *request to $\mathcal{CM}$. $\mathcal{CM}$ includes the received data into the view* $\mathsf{View}_{e'}$*, and returns* OK *to $\mathcal{P}$. Further, $\mathcal{CM}$ generates a (potentially empty) token* $\mathsf{Tkn}_{e'}$ *for the data view, for later use.*

$$\mathsf{Download} : \left\langle \begin{array}{c} \mathcal{U}(e) \\ \mathcal{CM}(\mathsf{View}_{e'}, \mathsf{Tkn}_{e'}, e') \end{array} \right\rangle \mapsto [\mathcal{U} : (\mathsf{View}_{e'}, \mathsf{Tkn}_{e'})]$$

The Download *procedure is interactive and is initiated by every active user, who during the* `Distr` *phase sends a download (*DWN*) request to $\mathcal{CM}$. $\mathcal{CM}$ responds by sending* $(\mathsf{View}_{e'}, \mathsf{Tkn}_{e'})$ *to $\mathcal{U}$.*

$$\mathsf{Endorse} : \left\langle \begin{array}{c} \mathcal{U}(\mathsf{sk}, \mathsf{View}_e, \mathsf{Tkn}_e, e) \\ \mathcal{CM}(\mathsf{End}_{e'}, e') \end{array} \right\rangle \mapsto [\mathcal{CM} : \mathsf{End}_{e'}]$$

The Endorse *procedure is interactive and is initiated by every active user in the system once during the* `Cert` *phase. The user uses its secret key* sk*, and its epoch counter to check if it should act as endorser. If so, it sends an endorsement of its view to $\mathcal{CM}$. $\mathcal{CM}$ includes the endorsement into the* $\mathsf{End}_{e'}$ *list.*

$$\mathsf{CheckCon} : \left\langle \begin{array}{c} \mathcal{U}(\mathsf{View}_e, \mathsf{Tkn}_e, e) \\ \mathcal{CM}(\mathsf{End}_{e'}, e')\rangle \end{array} \right\rangle \mapsto [\mathcal{U} : (\mathsf{epar}_e, \mathcal{U}.\mathsf{state})]$$

The CheckCon *procedure is interactive and is initiated by every active user in the system, who during the* `Ver` *phase*

sends a (VERIFY, e) *request to* $\mathcal{CM}$*.* $\mathcal{CM}$ *replies with the list of endorsements for epoch* e*,* End_e*.* $\mathcal{U}$ *checks the validity of the endorsements using its local data* $\mathsf{View}_e, \mathsf{Tkn}_e$*. If the check fails,* $\mathcal{U}$ *sets it state to* `abort` *and stops participating in the protocol for all future epochs. Else,* $\mathcal{U}$ *considers* View_e *valid, updates its epoch parameters and remains in* `consistent` *state.*

3.4. Properties

CoD enjoys *completeness* and *consistency*.

Completeness ensures that *any user engaged in the protocol that is in a* `consistent` *state at the beginning of an epoch, will be in a* `consistent` *state at the end of the same epoch with overwhelming probability.* This property needs to hold for every epoch, assuming all parties behave honestly, that each procedure is executed in the designated phase, and there is no temporal overlap between phases.

Definition 3.2 (completeness)**.** *For a given* CRS*, a* CoD *protocol is said to be complete if: for every epoch* $e \geq 1$*, let* λ *be the security parameter and* N *be the total number of users; let* $(\mathsf{View}, \mathsf{Tkn})$ *denote* $\mathcal{U}$*'s output in* Download *during epoch* e*; let* End *denote* $\mathcal{CM}$*'s output from the final* Endorse *call during the* `Cert` *phase in epoch* e*. Then*

$$\Pr\left[\mathsf{CheckCon}\left\langle \begin{array}{c} \mathcal{U}(\mathsf{View}, \mathsf{Tkn}, e) \\ \mathcal{CM}(\mathsf{End}, e) \end{array} \right\rangle \mapsto \texttt{abort}\right] < negl(\lambda, N).$$

Consistency guarantees that there exists a *unique* view which is shared among all honest users in the `consistent` state. We formalize this security property as follows:

Definition 3.3 (Consistency)**.** *Let* CRS *denote the common reference string output by the trusted setup of the* CoD *protocol. Let* λ *be the security parameter and* N *be the total number of users. A* CoD *protocol for* CRS *is said to provide consistency in the presence of at most* $I = f_I N$ *inactive users, and* $M = f_M(1 - f_I)N$ *corrupt parties if for every epoch* $e \geq 1$*, and for any pair of honest* $\mathcal{U} \neq \widehat{\mathcal{U}}$ *that are in* `consistent` *state at the beginning of epoch* e*, the probability that the events* $E_1 ... E_4$ *described below all happen given that the* $\mathcal{A}$ *generated* $(\mathsf{View}, \mathsf{Tkn}, \mathsf{End}), (\widehat{\mathsf{View}}, \widehat{\mathsf{Tkn}}, \widehat{\mathsf{End}})$ *such that* $\mathsf{View} \neq \widehat{\mathsf{View}}$*, is negligible.*

Formally, for every epoch e *let*

$E_1 = \mathcal{U}$*'s* Download *output is* $(\mathsf{View}, \mathsf{Tkn})$

$E_2 = \widehat{\mathcal{U}}$*'s* Download *output is* $(\widehat{\mathsf{View}}, \widehat{\mathsf{Tkn}})$

$E_3 = \mathcal{U}$*'s* CheckCon *output is* $(\mathsf{epar}_e, \texttt{consistent})$

$E_4 = \widehat{\mathcal{U}}$*'s* CheckCon *output is* $(\widehat{\mathsf{epar}}_e, \texttt{consistent})$

Then

$$\Pr\left[\bigwedge_{i=1}^{4} E_i \,\middle|\, \begin{array}{c} \mathsf{View} \neq \widehat{\mathsf{View}} \\ \wedge \\ \left(\begin{array}{c} (\mathsf{View}, \mathsf{Tkn}, \mathsf{End}) \\ (\widehat{\mathsf{View}}, \widehat{\mathsf{Tkn}}, \widehat{\mathsf{End}}) \end{array} \right) \leftarrow \mathcal{A} \end{array}\right] < negl(\lambda, N). \tag{1}$$

This property needs to hold for every epoch, under the assumptions that each procedure is executed solely in the designated phase, there is no temporal overlap between phases, and that the building blocks used in CoD are secure against the adversary described in 3.1.4.

4. Constructing CoD

We now construct a secure CoD instance from black box use of a few building blocks. While our construction and security analysis are generic, we later show how to efficiently and concretely instantiate each building block.

Our construction employs a trusted setup for generating public parameters of *the first epoch*, containing a random nonce for committee election and an initial list of the parties' public keys. The construction then updates these parameters in a *non-trusted* manner *each subsequent epoch*.

In this section, we assume we have a protocol that generates random nonces. We also assume generic parameters for committee sizes and for the number of committee members in a quorum who must endorse a view to guarantee that at least one honest party endorsed it. Later on, we address how random nonces can be generated via external randomness beacons or via VRFs by leveraging the CoD protocol itself. Moreover, we provide a careful analysis of committee and quorum sizes, which is one of our contributions.

4.1. Building Blocks

Our building blocks, introduced in Section 2, are the following: an EUF-CMA signature scheme Sig, a VRF with unpredictability under malicious key generation, a Key Evolving Signature (KES) scheme with forward security and a VKD. The VRF will be used as in [28] to randomly select a committee of anonymous parties. The KESscheme will be used also as in [28] to achieve security against rushing adaptive adversaries via forward security, preventing such adversaries from signing alternative versions of committee members' messages as soon as they are sent by corrupting those members and using their signing keys. The VKD is used to monitor an updated list of public keys assigned to each party for append-onlyness as the execution proceeds. In addition, we rely on a common source of randomness called SeedGen. Next we give more details on the VKD and the SeedGen.

4.1.1. IdP, VKD Operation and related Assumptions. We assume that there is an external party acting as an identity provider IdP, which is responsible for maintaining the set of users and their public keys. We assume that all users enroll a *single* keypair with the IdP (*i.e.* we do not deal with Sybil attacks and instead put our faith into heuristics such as verification of phone numbers etc.). In the same line, we assume that the number of users in the system N is known.

The $\mathcal{CM}$ maintains a VKD. In each epoch, the IdP sends the keys of new users to the $\mathcal{CM}$, which includes the keys in the VKD. For brevity, and to keep focus on how to ensure consistency, we have not detailed the VKD operations in the protocol. That is, the following is implicit in the protocol: The $\mathcal{CM}$ is expected to (1) expose the interface for

the VKD and respond to queries for public keys, and (2) to send the VKD digest (including an append-only proof) over the CoD channel each epoch during the `Col` phase. Further, all queries to the $\mathcal{CM}$ for public keys, and the verification of membership proofs of keys against the VKD digest, are implicit in the protocol.

With the above omittances, for the protocol to be formally complete, we have included a list $\mathcal{PK}$ of all public keys in the system. We remark however that in practice this list is not distributed to all parties (since it is quite large). Instead public keys are in practice obtained from the VKD whenever they are needed. This is possible since endorsers do not use public keys when endorsing and thus will not reveal themselves to the $\mathcal{CM}$.

4.1.2. SeedGen. We also rely on black box use of a function $\mathsf{s}_e \leftarrow \mathsf{SeedGen}(e)$ which takes an epoch counter e as input and outputs the seed s_e of the epoch. A SeedGen is correct if all honest parties in the `consistent` state receive an identical output, except with a negligible probability. It is secure if it is *unpredictable*, meaning that an adversary has a negligible advantage in predicting its output s_e before the time $e-\delta$ (e being the current epoch and δ being a number of epochs or phases). The period between $e-\delta$ and e is referred to as the leaky period. There are multiple ways to realize such a function, which we detail in Section 4.3.

4.2. The CoD Protocol

Our CoD protocol is presented in Protocol 1. In each epoch e, the protocol takes as implicit input to all procedures the epoch parameters epar_{e-1} (obtained from the trusted setup executed as Algorithm 1 or from the view of the previous epoch).

4.2.1. Trusted Setup (Description of Algorithm 1). The trusted setup algorithm outputs values as described in Section 3.2. In detail, it generates a common reference string (CRS) containing public parameters for all building blocks, and the initial set of public keys for all users registered in the system and the channel maintainer. In addition, it gets the initial VKD digest from the $\mathcal{CM}$, it computes the appropriate threshold value T (for selecting endorsers), and the quorum size k for the system accounting for a fraction of f_M corrupted parties, and f_I inactive users. Finally, it generates a random seed s_0. In practice, s_0 can computed through a heuristic such as using a hash of a current stock market valuation, or obtained by running a fair coin tossing protocol.

4.2.2. Our CoD Protocol (Description of Protocol 1). In line with the design of messaging apps, all procedures are initiated by users or publishers who send requests or upload data to the channel maintainer $\mathcal{CM}$. Each time a user initiates a procedure it starts with checking whether it is still in a `consistent` state, or if it has detected a split-view and entered the `abort` state and shall thus not participate in the protocol any further.

The Send procedure takes place during the `Col` phase and can be initiated by a publisher who wishes to upload data for publication. The $\mathcal{CM}$ stores any such data it receives into an initially empty View for the current epoch (line

---

Protocol 1 – CoD Protocol

Entities: $\mathcal{CM}$ channel maintainer; $\mathcal{P}_j$ publishers (including an external IdP); $\mathcal{U}_i$ users.
Notation: $(\mathsf{sk.X}_i, \mathsf{pk.X}_i)$ is $\mathcal{U}_i$'s key pair for primitive X, sk contains all $(\mathsf{sk.X}_i, \mathsf{pk.X}_i)$.
Trusted Setup: $\mathsf{epar}_0 = \mathsf{CRS}$, as output by Algorithm 1.
Initialization: At the beginning of epoch $e = 1$, for all i set $\mathcal{U}_i$.state to `consistent`. At the beginning of each epoch $e \geq 1$, set $\mathsf{End}_e \leftarrow \varnothing$, $\mathsf{View}_e \leftarrow \varnothing$, $\mathsf{Tkn}_e \leftarrow \varnothing$.

---

1. Send: $\langle \mathcal{P}_j(\mathsf{data_j}, \mathsf{e}); \mathcal{CM}(\mathsf{sk}_{\mathcal{CM}}, \mathsf{View_e}, \mathsf{e}) \rangle$ `Col`

1: $\mathcal{P}_j$ sends $(\text{SEND}, \mathsf{data}_j)$ to $\mathcal{CM}$
2: **if** $(\mathcal{P}_j = IdP)$, $\mathcal{CM}$ does the following
3: Interpret data_j as a set of public keys $\mathcal{PK}$, add the updated keys to a VKD, and generate a digest dig_e
4: Let $N = |\mathcal{PK}| - 1$, derive (T, k) from (N, f_M, f_I)
5: Let $\mathsf{s}_e \leftarrow \mathsf{SeedGen}(e)$
6: Set $\mathsf{epar}_e = (\mathcal{PK}, \lambda, N, (\mathsf{s}_e, \mathsf{dig}_e, T, k))$
7: Let $\mathsf{View}_e \leftarrow (\mathsf{View}_e || \mathsf{epar}_e)$
8: Sign $\mathsf{Tkn}_e \leftarrow \mathsf{Sig.Sign}(\mathsf{sk.Sig}_{\mathcal{CM}}, (\mathsf{View}_e, e))$
9: $\mathcal{CM}$ **outputs** $(\mathsf{View}_e, \mathsf{Tkn}_e)$ to itself (move to next phase)
10: **else**
11: $\mathcal{CM}$ lets $\mathsf{View}_e \leftarrow (\mathsf{View}_e || \mathsf{data}_j)$, and sends OK to $\mathcal{P}_j$

---

2. Download: $\langle \mathcal{U}_i(e); \mathcal{CM}(\mathsf{View}_e, \mathsf{Tkn}_e, e) \rangle$ `Distr`

1: $\mathcal{U}_i$ sends (DWN) to $\mathcal{CM}$, who answers (VIEW,View_e,Tkn_e)
2: $\mathcal{U}_i$ **outputs** the received $\mathsf{View}'_e, \mathsf{Tkn}'_e$ to itself

---

3. Endorse: $\langle \mathcal{U}_i(\mathsf{sk}_i, \mathsf{View}'_e, \mathsf{Tkn}'_e, e); \mathcal{CM}(\mathsf{End}_e, e) \rangle$ `Cert`

1: $\mathcal{U}_i$ obtains (s_{e-1}, T) from epar_{e-1}
2: $\mathcal{U}_i$ runs $(y_i, \pi_i) \leftarrow \mathsf{VRF.Prove}_{\mathsf{sk.VRF}_i}(\mathsf{s}_{e-1})$
3: **if** $y_i \geq T$
4: $\mathcal{U}_i$ **ends** here (without aborting, moves to the next phase)
5: **else** $\mathcal{U}_i$ does the following ($\mathcal{U}_i$ has `endorser` role in epoch e)
6: **Verify that**: VKD updates are append-only w.r.t. $(\mathcal{PK}, \mathsf{dig}_e)$, **and** $\mathsf{s}_e = \mathsf{SeedGen}(e)$, **and** (T, k) are correctly derived from $(|\mathcal{PK}| - 1, f_M, f_I)$ and match the values given in epar_e (contained in View'_e). **If any verification fails**, $\mathcal{U}_i$ sets its state to `abort`, and **leaves** the protocol
7: Let $\sigma_i \leftarrow \mathsf{KES.Sign}(\mathsf{sk.KES}_i, (y_i, \pi_i, \mathsf{View}'_e, \mathsf{Tkn}'_e), e)$
8: Let $\mathsf{sk}'.\mathsf{KES}_i \leftarrow \mathsf{KES.KeyUpdate}(\mathsf{sk.KES}_i, e)$
9: Delete $\mathsf{sk.KES}_i$ and lets $\mathsf{sk.KES}_i \leftarrow \mathsf{sk}'.\mathsf{KES}_i$
10: $\mathcal{U}_i$ sends $(\text{ENDORSE}, (y_i, \pi_i, \sigma_i, e))$ to $\mathcal{CM}$
11: Upon receiving $(\text{ENDORSE}, (y'_i, \pi'_i, \sigma'_i, e))$, $\mathcal{CM}$ includes it in $\mathsf{End}_e \leftarrow \mathsf{End}_e \cup \{(y_i,' \pi'_i, \sigma'_i)\}$ and $\mathcal{CM}$ **outputs** End_e

---

4. CheckCon: $\langle \mathcal{U}_i(\mathsf{View}'_e, \mathsf{Tkn}'_e, e); \mathcal{CM}(\mathsf{End}_e, e) \rangle$ `Ver`

1: **if** $0 = \mathsf{Sig.Verify}(\mathsf{pk.Sig}_{\mathcal{CM}}, (\mathsf{View}'_e, e), \mathsf{Tkn}'_e)$
2: $\mathcal{U}_i$ sets its state to `abort`, and **leaves** the protocol
3: $\mathcal{U}_i$ sends (VERIFY, e) to $\mathcal{CM}$, who answers (VERIFY, End_e)
4: $\mathcal{U}_i$ lets $\mathsf{ctr} \leftarrow 0$, and obtains (T, k, s_{e-1}) from epar_{e-1}
5: **for** $(y_j, \pi_j, \sigma_j) \in \mathsf{End}_e$ **do** (Endorsement Validity Check)
6: **if** $\begin{pmatrix} \mathsf{VRF.Verify}(\mathsf{pk.VRF}_j, \mathsf{s}_{e-1}, y_j, \pi_j) \wedge \ (y_j < T) \wedge \\ \mathsf{KES.Verify}(\mathsf{pk.KES}_j, (y_j, \pi_j, \mathsf{View}'_e, \mathsf{Tkn}'_e), \sigma_j, e) \end{pmatrix}$
7: $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$
8: **if** $\mathsf{ctr} < k$ (Quorum Check)
9: $\mathcal{U}_i$ sets its state to `abort`, and **leaves** the protocol
10: $\mathcal{U}_i$ **outputs** $(\mathsf{epar}_e, \mathtt{consistent})$ and sets $e = e + 1$.

Algorithm 1 – Trusted Setup

Input: λ security parameter, f_M (f_I) max fraction of admissible corrupted parties (inactive users)
Output: $\mathsf{epar}_0 = \mathsf{CRS}$

1: Setup all building blocks: $pp_{\mathsf{Sig}} \leftarrow \mathsf{Sig.Setup}(\lambda)$, $pp_{\mathsf{KES}} \leftarrow \mathsf{KES.Setup}(\lambda)$, $pp_{\mathsf{VRF}} \leftarrow \mathsf{VRF.Setup}(\lambda)$
2: Construct $\mathcal{PK} = \{(\mathsf{ID}_{\mathcal{CM}}, \mathsf{pk}_{\mathcal{CM}}), (\mathsf{ID}_1, \mathsf{pk}_1), (\mathsf{ID}_2, \mathsf{pk}_2), \ldots\}$ and enroll all parties in $\mathcal{PK}$ in the VKD
3: Let $N = |\mathcal{PK}| - 1$, use (N, f_M, f_I) to derive (T, k)
4: Obtain dig_0 from $\mathcal{CM}$ (See how in Section 5)
5: Let $\mathsf{s}_0 \xleftarrow{\$} \{1, \ldots, 2^\lambda\}$
6: Let $\mathsf{CRS} = (\mathcal{PK}, \lambda, \mathsf{N}, (\mathsf{s}_0, \mathsf{dig}_0, \mathsf{T}, \mathsf{k}, \mathsf{pp}_{\mathsf{Sig}}, \mathsf{pp}_{\mathsf{KES}}, \mathsf{pp}_{\mathsf{VRF}}))$
7: **Output** $\mathsf{epar}_0 = \mathsf{CRS}$

11). In the special case where the information is a list of public keys $\mathcal{PK}$ from the IdP (line 2), $\mathcal{CM}$ creates a VKD digest (including the append-only proof) (line 3). $\mathcal{CM}$ also generates the next epoch parameters by deriving (T, k) w.r.t. $(|\mathcal{PK}|, f_M, f_I)$ (line 4), and computing a new seed $\mathsf{s}_e \leftarrow \mathsf{SeedGen}(e)$ (line 5). The way T, k are computed and the way they relate to one another is described in Section 5. In practice, they are computed from the code in [**?**] . We discuss several ways to instantiate SeedGen in Section 4.3. $\mathcal{CM}$ then sets the new parameters epar_e containing the updated set of public keys $\mathcal{PK}$, the VKD digest dig_e (for verifying that $\mathcal{PK}$ has been updated in an append-only manner), the new seed s_e, and a copy of the values which are to remain consistent over epochs (line 6). $\mathcal{CM}$ then adds epar_e to the view View_e (line 7), generates a token as a signature of the view in epoch e (line 8), and stops accepting updates for the view in this epoch (line 9), thereby completing the `Col` phase.

The rest of the procedures must be initiated by all honest active users in each epoch.

The Download procedure takes place during the `Distr` phase. Here, all active honest users request the data published during the current epoch from $\mathcal{CM}$. $\mathcal{CM}$ responds to each request with the data view and the token for the view. This token allows users – who are later selected as endorsers – to prove that their view was actually given by the $\mathcal{CM}$, and not falsified by a corrupt endorser. It also allows non-selected users to efficiently compare their views with endorsed views. Note that at this point the consistency of the published data is not yet audited and thus it should not be trusted.

The Endorse procedure takes place during the `Cert` phase. Here each user first obtains the seed and selection threshold of the epoch (line 1), uses the seed to compute a verifiably random value (line 2), and checks if the value is below the threshold for being selected as an endorser of this epoch (line 3). If it is not selected, the user ends the procedure for this epoch (line 4). If the user is selected, it verifies the new set of parameters epar_e (line 6). To this end, it verifies that the new set of public keys $\mathcal{PK}$ is a modification for the previous epoch's one by auditing the VKD append-onlyness with respect to $\mathcal{PK}$ and dig_e. This step is intentionally vague to admit different realizations for the VKD construction. A trivial, yet inefficient way, would be to verify that dig_e is the root of a hash tree with $\mathcal{PK}$ as leaves. If a discrepancy is detected, the user aborts and leaves the system; otherwise $\mathcal{PK}$ is considered verified and the algorithm derives from it the new user headcount N. A discussion on practically realizing the VKD is available in Section 4.4.2. The user also checks that T, k are derived correctly from N and the fractions of maximal admissible corrupt and inactive users. Furthermore, it checks that the seed for the next epoch is correctly generated. If all checks pass, the user creates an endorsement of their view (line 7), evolves the secret signing key (line 8), and deletes the old secret key (line 9). For simplicity, we here assume that users resuming from being inactive in the previous epoch evolves the secret signing key up to the current epoch. The user then sends the endorsement signature σ_i and proof of selection as an endorser (y_i, π_i) to the $\mathcal{CM}$ (line 10). When the $\mathcal{CM}$ obtains the message, it adds the endorsement to its list of received endorsements for this epoch and outputs the updated list (line 11).

The CheckCon procedure takes place during the `Ver` phase. Here each user first checks whether the $\mathcal{CM}$ signature on the view it received during Download is valid (line 1) and aborts if the check fails. To then verify whether this view is consistent, it requests the endorsements for this epoch from $\mathcal{CM}$, who responds with all endorsements it has received (line 3). The user then counts the number of endorsements which are both valid endorsements for *the users own view*, and are generated by a valid endorser of this epoch (lines 4-7). If the number of such endorsements falls short of a quorum (line 8), the user has detected a split-view or a failure in obtaining sufficiently many valid endorsements. In either of those cases, the user sets its state to `abort` (line 9) and ceases to participate for all future epochs. Otherwise, the user computes the next epoch parameters and outputs its consistent state for the view of this epoch (line 10).

4.2.3. Security Analysis. The following theorem states that our protocol is secure as long as the cryptographic building blocks are secure. This means that CoD is complete (Definition Definition 3.2) and provides consistency (Definition Definition 3.3), given that the VRF has uniqueness, provability and has pseudorandom outputs that are unpredictable even under malicious key generation, the signatures Sig and KES are existentially unforgeable under chosen message attack, and KES is also forward secure, the VKD is complete and sound, and SeedGen has outptus that are upredictable.

Theorem 4.1. *Let* CoD *be the protocol described in Protocol 1, instantiated with a Key Evolving Signature scheme* KES *(as per Definition 2.2), a standard EUF-CMA secure signature scheme* Sig*, and a Verifiable Random Function* VRF *(as per Definition 2.1), a Verifiable Key Directory* VKD *(as discussed in Section 2) and a random seed generation protocol* SeedGen *(as discussed in Section 4.3). Let λ denote the security parameter, N the total number of active users at each epoch, and $f_I, f_M \in [0, 0.5]$ denote respectively the fraction of inactive users, and of parties*

corrupted by a rushing adaptive malicious adversary $\mathcal{A}$ (who may also corrupt $\mathcal{CM}$ and IdP). Then, for every epoch $e \in \{1, \ldots, E\}$ and $E = \mathsf{poly}(\lambda)$, Protocol 1 is a CoD *protocol (as per Definition 3.1) that is complete (as per Definitions 3.2) and provides consistency (as per Definition 3.3) in the* CRS *model, i.e., assuming the* CRS *(epar_0) is generated as in Algorithm 1.*

The complete and detailed proof is available in Appendix A, but we provide a proof overview here.

Completeness essentially reduces to the correctness of the building blocks and to appropriately setting the quorum size k. Our novel statistical analysis in Section 5.2 bounds the probability of not collecting enough endorsements by 2^{-b} for security parameter b, as illustrated in Figure 3b.

Consistency when $\mathcal{CM}$ is honest reduces to the unforgeability of Sig, indeed malicious parties (controlled by the adversary $\mathcal{A}$) cannot generate valid signatures for other views (line 1 in CheckCon).

Consistency against a malicious $\mathcal{CM}$ (and other adaptively corrupted parties) relies on multiple factors. The unpredictability of $\mathsf{SeedGen}(e)$ guarantees that $\mathcal{A}$ cannot predict or influence its output s_e, which in turn could bias VRF evaluations. The security of the VRF evaluated on an unpredictable s_e ensures that $\mathcal{A}$ cannot affect the cryptographic selection of endorsers. Unforgeability for KES implies that $\mathcal{A}$ cannot create signatures (hence endorsements) for different views in place of honest endorsers. However, upon sending an endorsement to the malicious $\mathcal{CM}$, $\mathcal{A}$ can choose to corrupt the (honestly selected) endorser. Forward security makes such targeted corruptions pointless, since endorsers update and securely erase current signing keys prior to sending out endorsements (line 8 in Endorse). Finally, our analysis in Section 5 guarantees that the quorum size k can be set so that more than k honest and fewer than k malicious endorsements will be collected with high enough probability. The statements '$\mathcal{A}$ *cannot*' above are meant in the cryptographic sense, *i.e.*, if $\mathcal{A}$ could, then there exists a reduction that breaks a property of a building block.

4.3. Realizing SeedGen

As defined in Section 4.1.2, a secure SeedGen needs to be unpredictable. Without an unpredictable seed, an adversary could target a (any) given future epoch and use the time until that epoch arrives to pre-compute key pairs that are selected by the VRF for this seed. This would undermine the quorum check of Protocol 1, since the adversary could then compute sufficiently many adversarial keys to form a quorum. Seed unpredictability prevents such adversarial key generation attacks by making sure that the adversary gets no (zero) time for pre-computation. Thus, keys in the VKD can only be trusted for VRF selection based on the seed s_e if the keys were registered in the VKD before the leaky period for SeedGen's computation of s_e began. That is, in a secure CoD *the seed that selects an endorser must not be older than an endorser's public key*. With this in mind, we now give realizations of SeedGen under different assumptions, first by assuming GOD in a pre-existing external system, then internally realized without GOD but with simplifying assumptions, and finally in the general case assuming neither GOD nor other simplifications.

4.3.1. Using an External Randomness Beacon *with* GOD. One way of realizing SeedGen is to use an external randomness beacon [35] which provides a fresh random nonce in each epoch. Such protocols allow for any party to publicly verify that each output has been generated correctly, ensuring that the output is unbiased and unpredictable. In order to achieve such strong properties, randomness beacons generally require standard broadcast with GOD, which we want to avoid. However, notice that randomness beacons are readily available in a number of Proof-of-Stake based blockchain protocols that intrinsically execute them as subprotocols, *e.g.* [36]. Hence, instead of executing a randomness beacon protocol, parties can rely on an existing beacon and leverage its public verifiability properties to validate its random outputs relative to each epoch.

4.3.2. Using an Internal Beacon Without GOD (With Simplifying Assumptions). Let us now describe how to realize SeedGen via the bounded bias beacon of Ouroboros Praos [28], without relying on GOD for security. We refer to this realization as $\mathsf{SeedGen}_{\mathsf{Praos}}(e)$. In the model of [28], the adversary is allowed to reset the beacon (resample randomness) a bounded number of times during the leaky period which is a specific time interval before epoch e. Such a beacon can be realized internally in CoD by having every endorser in the CoD protocol provide not only an endorsement signature σ_i and a VRF output/proof pair (y_i, π_i), but also an extra VRF output/proof pair (s_i, π'_i). When computing $\mathsf{SeedGen}_{\mathsf{Praos}}(e)$, the seed s_e is derived by verifying all pairs (s_i, π'_i) from epoch $e-1$, and computing $\mathsf{s}_e = H(s_1|\ldots|s_n)$ from valid VRF outputs $s_1, \ldots, s_n$, where H is modeled as a random oracle.

Provided that the VRF key pairs are fixed before the randomness generation starts, we can follow the analysis from [28]. If we make the simplifying assumption that keys are fully fixed (no updated or added keys throughout the protocol), then the adversary can only introduce a bounded amount of bias in s_e by selectively adding or removing pairs s_i, π'_i from the view of honest users (*i.e.* refusing to deliver pairs generated by honest parties or refusing to generate pairs that should have been generated by corrupted parties). In other words, seed generation in epoch e is derived from VRF outputs from epoch $e-1$. Thus s_e is unpredictable if *at least one* of the s_i's used to compute the SeedGen was delivered by an honest party, which can be statistically guaranteed with high probability by choosing an appropriate committee size.

4.3.3. General Construction (Without Simplifying Assumptions). We now provide a realization $\mathsf{SeedGen}_{\mathsf{Gen}}$ that neither relies on GOD nor assumes fixed keys.

In the above realization, $\mathsf{SeedGen}_{\mathsf{Praos}}$, the seed s_e is computed from the VRF outputs $s_1, \ldots, s_n$ from epoch

$e-1$, with proofs $\pi'_1, \ldots, \pi'_n$ that are verified against the seed s_{e-1}. The seed s_{e-1} has, in turn, been constructed from VRF outputs from epoch $e-2$, with proofs that are verified against the seed s_{e-2}, and so on. When keys are allowed to be updated, this chaining of seeds becomes a problem for users which resume after being offline. Resuming users can not rely on the unpredictability of such a chain of seeds to guard against adversarial key generation, because they were asleep when the seeds were fresh. Thus, resuming users do not know how old the seeds in the chain are – they can even be older than when these users were last online. Securely evaluating $\mathsf{SeedGen}_{\mathsf{Praos}}(e)$ requires that keys used in the evaluation are fixed when randomness generation starts. Randomness generation in $\mathsf{SeedGen}_{\mathsf{Praos}}(e)$ can start as soon as the seed is known. However, without any guarantees for seed freshness, there is no way to set a point in time for when to fix keys.

A heuristic for meeting the key fixation requirement in our case is for a resuming user evaluating $\mathsf{SeedGen}_{\mathsf{Praos}}(e)$ to make sure that the seed used for evaluating the s_i's is not older than when the user was last online (and ignore any keys updated after that time). In order to solve this, we can define $\mathsf{SeedGen}_{\mathsf{Gen}}(e)$ as $H(\mathsf{s}_{e-1}, \mathsf{SeedGen}_{\mathsf{Praos}}(e))$ and have a user compute s_{e-1} as follows. The user obtains s_k and $\mathsf{SeedGen}_{\mathsf{Praos}}(k)$ from the $\mathcal{CM}$ for all epochs $e-j < k < e$ for which the user was sleeping (two hashes per offline epoch). Then, it reconstructs the hash chain defined by $\mathsf{SeedGen}_{\mathsf{Gen}}(e)$ from s_{e-j} and $\mathsf{SeedGen}_{\mathsf{Praos}}(e-j)$ (when it was last online) up to s_{e-1}. Since the user knows that s_{e-j} was unpredictable for epoch $e-j$, and due to the preimage resistance of H, then s_{e-1} is by construction more recent than epoch $e-j$. No keys registered before epoch $e-j$ could thus have been adversarially generated to be selected from s_{e-1}.

4.4. Efficiency Estimates

4.4.1. CoD Efficiency. Protocol 1 employs only lightweight cryptographic tools. The heaviest computational tasks for users is in CoD.CheckCon, where they verify selection proofs and endorsements (lines 4-7 in CheckCon), since for these tasks the CPU overhead scales linearly with the size of the quorum. We here provide an efficiency estimation of these parts when concretely instantiating selection proofs from a VRF, and endorsements as XMSS forward secure signatures. Our main use case for CoD is consistency in key transparency, and thus the protocol should be efficient enough to run on mobile phones.

We have implemented the VRF of [28] in C using OpenSSL 1.1.1w. Internally this VRF depends on a hash, which we instantiate as SHA256, and on group operations, which we instantiate over the P256 curve. We compiled for iOS, and executed tests of VRF on the mid-tier phone iPhone 12 with iOS 17.3.1 installed. CPU time was measured using the mach_absolute_time function in mach_time.h. This timer provides high resolution measurements (nanoseconds) of CPU ticks. The code of the implementation and tests is available at [37]. The results are that a VRF evaluation takes 0.00145 seconds, *i.e.*, 1.45 seconds per 1,000 endorsers.

We have no direct measurements of a KES scheme on an iPhone 12. For a rough performance estimate we take the measurements in [38, Table IV] for the XMSS KES instantiated with SHA-256 and evaluated on an 9-year old Intel Core i7-6700K (which, although comparing mobile and desktop CPUs is tricky, can be said to be roughly on par with the CPU in iPhone 12 in single core tasks). For 128-bit security, signature verification time is 0.3ms (0.3 seconds per 1,000 endorsers). We defer concrete examples of execution times for realistic parameters for CoD to Section 5.3, where we take into account security against grinding attacks.

As we note in Appendix B, efficiency can be further improved by exploring the use of aggregate signatures and VRFs in CoD.

4.4.2. VKD Efficiency. While the specifics of how to audit a VKD for append-onlyness is not in the scope of this paper and should be considered an orthogonal problem, the Endorse procedure in Protocol 1 depends on endorsers verifying VKD append-onlyness. We therefore here account for the overhead of this verification. We note though that using a VKD for key transparency requires verifying the append-onlyness of the VKD whether CoD is used or not, and thus the existence of overhead due to auditing the VKD for append-onlyness is not a consequence of CoD.

If the VKD is implemented in the style of CONIKS, where append-onlyness follows from each user monitoring its own key, the overhead is small. CONIKS [5] reports the user network overhead to be 736 bytes per epoch. The user CPU overhead is also very small, consisting only of verifying a signature and computing $\mathcal{O}(\log_2(N))$ hashes.

For a VKD in the style of SEEMless, where auditing append-onlyness consists of verifying tree paths for inserted keys, cost are a bit larger. SEEMless [7] reports a verifier CPU overhead of 0.22s and a network overhead of maximum 4.24 MB for a VKD with 10 million users, where 1000 users are added and 1000 keys are updated during an epoch. A nuance here is that in CoD, the overhead of monitoring append-onlyness falls on endorsers on mobile phones, whereas SEEMless expects auditors to be external parties with potentially more powerful hardware. Further, the monitoring costs of SEEMless-style VKDs grow with the number of keys inserted in each epoch, and the overhead for endorsers will increase accordingly. We emphasize again however that the efficiency of verifying VKD append-onlyness is an orthogonal problem, and that straightforward improvements such as reducing the amount of work per endorser by dividing the verification between the endorsers, or reducing proof sizes with zk-SNARKs [39], are possible.

5. How to set the Quorum Size

We now provide a new and improved statistical analysis, which is motivated by the fact that the stochastic nature of the sortition process makes it impossible for a system user to know the size of the endorsement committee. That is, in

any epoch, the actual number of users that are selected as endorsers in that epoch is not a number that is revealed or can be made available to the system users. This makes it more involved to define security guarantees for an honest majority of endorsers or split-view defense, further motivating our quorum concept. Compared to the analyses in, e.g., [30], [31], our statistical analysis is also more flexible regarding how to model users who do not provide outputs on time, which may be considered actively malicious or simply inactive (crashed) depending on the situation.

The security analysis of our CoD protocol (Protocol 1) relies on having an opportune quorum size k that identifies the minimal number of endorsements – signatures for the same view generated by distinct users – which together guarantee, with high probability, the existence of a unique view across all users who are in a `consistent` state.

Our goal here is to develop a well-defined mechanism to compute k for any given CoD system given the following three parameters; (1) the probability of being selected as an endorser (denoted by p), (2) the total number of honest active users (denoted by H), and (3) the total number of malicious active users (denoted by M).

Let $0 < p < 1$ denote the probability that any given user is selected for jury duty on the hidden committee of endorsers. Note that all users in the system are selected cryptographically at random in a publicly verifiable way, and independently from other users. This is done using the function VRF.Prove (employed for Endorse in CoD), which returns as part of its output a pseudo-random value y. If $y < T$, this indicates that a user is selected as an endorser. Intuitively this selection process simulates a coin flip that returns "heads" (output 1) with probability p. The selection of a user as endorser is therefore accurately modeled as a Bernoulli random variable with parameter p. Since the user selection trials are independent of one another, the overall sortition process runs across the whole population of N users is distributed as a Binomial random variable $B(N, p)$ with $p = \frac{T}{|y|}$, where T is the threshold available in the epoch parameters and $|y|$ denotes the size of the random value output by VRF.Prove.

For the novelty in our analysis, we adapt a partition–and–cut methodology in which we first partition users into different classes, and then we exploit statistical properties of these classes to find a cutoff point that in some sense maximizes the differences between those classes of users.

We partition N, the total number of users in the system, into three disjoint sets, identified by the fractions of inactive f_I, and malicious f_M users);

- I inactive (non-responsive) users, I has size $f_I \cdot N$,
- M malicious users who are active in this epoch, M has size at most $f_M \cdot (1 - f_I) \cdot N$,
- H honest active users, H has size at least $(1 - f_M) \cdot (1 - f_I) \cdot N$.

Note that $N = H + M + I$ with $H, M, I \geq 0$, and the number of active users is $N_A = H + M = (1 - f_I) \cdot N$.

We remark that that for the I inactive users we do not care *why* they are inactive. They can be actively suppressed by an adversary or they may simply be off-line and temporarily not interacting in the system. This partitioning approach enables us to (conceptually) separate actively malicious users from users that are inactive in a given epoch (which can typically be a non-negligible fraction of the population when considering large scale, world-wide adopted systems) to provide more accurate security estimates.

At each epoch, we expect to have $H + M$ active users running endorser selection. We recall a known fact about Binomial distributions, which follows directly from Chu–Vandermonde identity: $B(H + M, p) = B(H, p) + B(M, p)$. This means that we can analyse the selection of active malicious users and of active honest users independently. In particular, $B(M, p)$ identifies a well-defined discrete probability distribution that is different, in shape, from $B(H, p)$ if $M \neq H$. In short, our analysis is based on this shape difference.

For the sake of understanding our analysis, it is easier to first consider the simplified case of selecting a committee that, with high probability, has an honest majority. This is what more intuitively corresponds to a quorum in the traditional setting where we have a channel *with* GOD. We present this in Section 5.1. This part in itself also serves as a new and more accurate analysis of the setting in [30].

In Section 5.2 we will then make our analysis more involved for the CoD channel quorum case to provide high probability guarantees that the system is additionally protected against split-view attacks, which is harder to accomplish since it requires more than just an honest majority in the endorsement committee.

5.1. Warmup: Quorum in Broadcast *with* GOD – Protection Relying on an Honest Majority

In GOD channels, everyone that speaks will be heard, so a quorum in this setting is a committee size that, with high probability, yields an honest majority.

Let us denote $Z \sim B(M, p)$ and $Y \sim B(H, p)$, so that Z and Y, respectively, correspond to the number of malicious and honest users that have been selected as endorsers. While the endorsement committee additionally consists of a number $W \sim B(I, p)$ of silent users, they are not active in the endorsement process and need not be considered further in this analysis.

The reader may note that the exact values taken by the random variables Z and Y at any given epoch are not available to the system users, so they do not know the size of the actual endorsement committee. Our analysis therefore focuses on finding a value $k \in \{1, \ldots, M\}$ such that both $\Pr(Z \geq k)$ and $\Pr(Y \leq k)$ are negligible. In other words, in our practical application we need a cutoff value k (that we call quorum) which guarantees that, with high probability, the following two bad events do *not* occur;

- A) committee has k or more malicious endorsers, and
- B) committee has k or fewer honest endorsers.

Intuitively, we are thus comparing the right tail area of $B(M, p)$ with the left tail area of $B(H, p)$, and require both of them to be small. If such a cutoff value k exists, then

the act of verifying k valid signatures on one and the same view provides a provable guarantee that more honest than malicious users endorse the view.

To see that such a k always exists, first note that the probability p is a parameter that directly determines the expected size of the endorsement committee. Given any fixed probability p, let the value $k' \in \{1, \ldots, M\}$ denote the smallest value i for which $\Pr(Z \geq i) \leq \Pr(Y \leq i)$. It is clear that k' exists and is uniquely defined since $M < H$. Note that for k' we have $\Pr(Z \geq k') \approx \Pr(Y \leq k')$, and $b' = -\log(\Pr(Y \leq k'))$ can be interpreted as the bit security level of the probability p, as $2^{-b'}$ is the probability of endorsement committee failure in terms of the bad events A or B above.

In order to find a suitable cutoff value k that is optimal in this metric for any given bit security level b, we take the cutoff value k' corresponding to the minimal probability p (which determines the minimal expected committee size) with bit security level $b' \geq b$.

These computations are indeed practical, even for our extreme case target of systems with $N = 1,000,000,000 \approx 2^{30}$ users. We provide concrete values for realistic scenarios in Section 5.2.1, and we also provide code that can be used to compute these values [40].

5.2. Quorum in CoD – Protecting Against split-view Attacks

Protecting against split-view attacks is tightly coupled to computing the term $negl_{\mathsf{Qrm}}(N, f_M, f_I)$ in the proof of Theorem 4.1 in the case of a malicious $\mathcal{CM}$. In this case, the adversary's highest chance to succeed in a split-view attack is to divide the honest users into two disjoint sets of equal size. This translates to having half of the honest endorsers endorsing the same view, regardless of which view the malicious users choose to endorse. Statistically, we thus need to compare $X \sim B(M + \frac{H}{2}, p)$ to $Y \sim B(H, p)$. The intuitive reasoning is identical to the explanation in Section 5.1, except that the peaks of the probability distributions are now closer together to reflect that safe-guarding against split-view attacks is harder than ensuring an honest majority. However, the procedure for finding an optimal cutoff value k for any given bit security b is precisely the same, and the act of verifying k valid signatures on one and the same view now provides a provable guarantee of the consistency property of the KT log. And this k value is precisely our quorum size that we set out to determine, which completes our main goal for this part.

The probability distributions we are discussing closely resemble the ones plotted in Figure 2. The optimal cutoff value k' corresponds to a vertical line that separates the two bumps. The bit security level b intuitively corresponds to the $\log$ of the tail areas beyond the cutoff value k'. Figure 2 further illustrates how the probability distributions change shape as we modify the probability p, which governs the expected endorsement committee size.

From our statistical analysis it follows that the limit of efficiency is approached as the number of malicious users M approaches $\frac{H}{2}$. When $M \geq \frac{H}{2}$, there exists no hope of efficiently protecting the system against split-view attacks, but for reasonable $M < \frac{H}{2}$, we can actually compute concrete values for relevant parameter sets.

5.2.1. Some Optimal Quorum Size Computations. For some realistic parameter sets, consider a large scale application with one billion (1,000,000,000) users, where we allow for $f_I = 20\%$ of the users to be silent/non-responsive. In Figure 3 we present optimal quorum sizes for parameter sets with varying degrees of honesty and maliciousness in the remaining population.

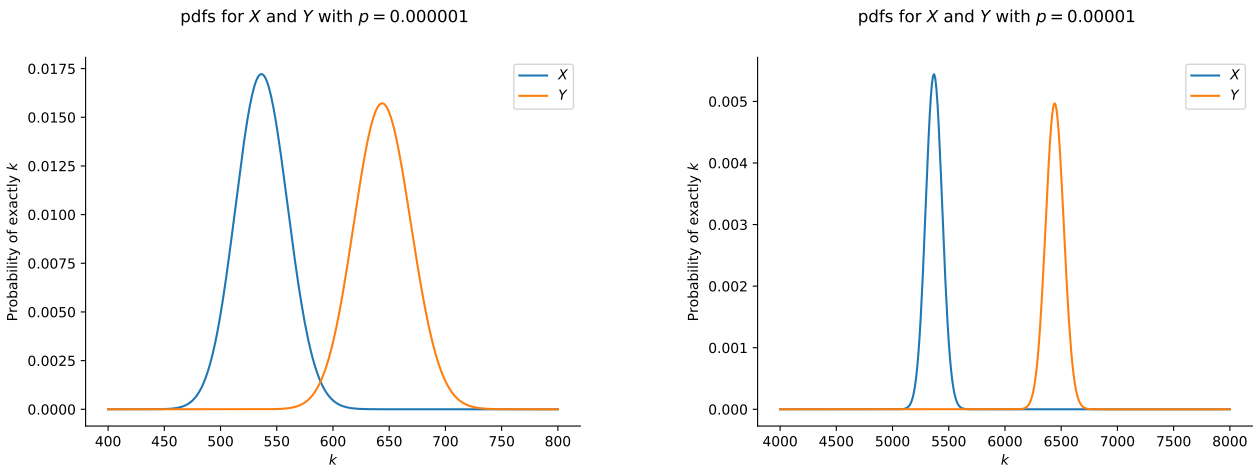
5.3. About Grinding

We handle adversarial seed tampering for the cryptographic sortition similarly to [30]. This approach gives the adversary power to choose the seed. However, since epochs have time limitation, and the adversary is computationally bounded, only up to $C = 2^c$ seed resets are allowed. In our analysis, this corresponds to having C samplings from the random variable X. Since each sampling (new seed) corresponds to an independent event, the overall probability of succeeding in finding a seed that yields a larger committee is $\sum_{i=1}^{C} \Pr(X \geq k) = C \cdot \Pr(X \geq k)$. To make this latter probability negligible, it is enough to select a k for which $\Pr(X \geq k) = 2^{-(b+c)}$. In other words, using our methodology, it is possible to explicitly account for grinding by adjusting the metric to weight the respective tail areas of $B(M + \frac{H}{2}, p)$ and $B(H, p)$ accordingly.

5.3.1. Considerations on Grinding and Epoch Duration. For a conservative choice that targets $b' = 128$ bit security level and grinding tolerance $C = 2^c = 2^{128}$, the quorum is given in the $b = b' + c = 256$ row in Figure 3b. In a system of a billion users, the *worst case* scenario with 20% malicious users (200 million corruptions) and 20% offline users we get a quorum of $41,816$. According to the compute times discussed in Section 4.4, the bulk of consistency check verifications takes about 70 seconds for each epoch.

In practice it is possible to eliminate the grinding attack vector with minimal efficiency penalties by adjusting the grinding tolerance to the epoch duration, similarly to how Bitcoin adjusts the block generation speed. Intuitively, the shorter the epoch, the smaller the grinding tolerance needed to maintain security. However, too short epochs place high CPU demands on users due to frequent consistency checks. A realistic setting of parameters for a system with $N = 10^9$ users is a bit security level $b' = 128$, epoch duration of 7 hours, that is on par with existing CT logs[2], a grinding factor of $C = 2^c = 2^{60}$ (using the beacon described in Section 4.3.2), $f_I = 20\%$ inactive users, $f_M = 5\%$ malicious users (corresponding to $40,000,000$ users, roughly the population of Canada). Interpolating the numbers in Figure 3b for the $75\% - 5\%$ column, these parameters yield a quorum

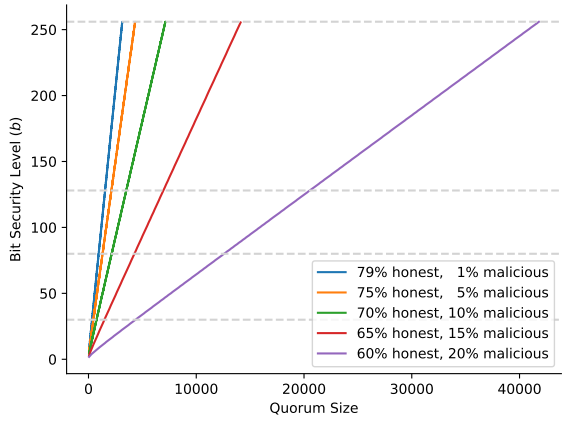
2. For example, the epoch duration of Cloudflare's Nimbus2024 log is 7 hours [41]. Other logs have similar epoch durations.



(a) Smaller Committee Size

(b) Larger Committee Size

Figure 2: Plots of the probability distribution functions $X \sim B(M + \frac{H}{2}, p)$ and $Y \sim B(H, p)$ for a population of $N = 10^9$ with 20% inactive users, 20% (active) malicious users and 60% (active) honest users, with $p = 10^{-6}$ (left) and 10^{-5} (right).



(a) Plotted

b	Quorum Size					
	79%	75%	70%	65%	60%	Honest
	1%	5%	10%	15%	20%	Malicious
	20%	20%	20%	20%	20%	Inactive
256	3,133	4,329	7,141	14,145	41,816	
128	1,541	2,127	3,509	6,951	20,537	
80	944	1,306	2,152	4,260	12,584	
30	329	454	750	1,481	4,365	

(b) Tabulated

Figure 3: Plots and concrete values for $negl_{\mathsf{Qrm}}(N, f_M, f_I)$, identifying suitable quorum sizes for $N = 10^9, f_I = 20\%$ and varying degrees of honesty in the remaining (active) population.

size of $k = 3,236$. According to the performance estimates in Section 4.4, this translates to each user spending roughly 6 seconds every 7 hours for running the bulk consistency check of our CoD protocol.

These crude estimates illustrate that our techniques are not limiting in practice, not even for large scale applications. More optimized approaches can give even better performance guarantees in practice, for example, as we suggested in Section 5.3, by recomputing the optimal quorum sizes using different metrics that explicitly include a grinding factor that is suitable for the target use-case application. And as we mention in Appendix B, future work leveraging aggregate signatures and VRFs can improve the efficiency of our scheme further.

6. Privacy & User Churn

Privacy Some VKDs, *e.g.*, CONIKS [5], SEEMless [7] and Parakeet [8] (but not all, *e.g.*, Merkle^2 [6]) provide a notion of privacy which limits information leaks such as the number of registered keys and information about other keys than the one which was queried for during lookups. However, these protocols do leak, *and must leak*, information about what keys are registered in the system. This is since their main purpose is to provide append-only guarantees for key lookup systems. However, the identities of users and their public keys are *public information* in key lookup systems. Any privacy regarding the identities of registered users is thus necessarily weak. While Protocol 1 does not directly undermine many of the privacy guarantees of a

VKD, CoD relies on an certain fraction of honesty among the users. Thus, to boost confidence in this assumption, it is natural for the identities in the system to be public. Since this is the normal case for a key lookup system, and since privacy guarantees of VKD schemes are necessarily weak in the above sense, we argue that publishing the list of identities of users is sensible since membership of a system is in fact already public information.

Silent User Churn While users explicitly leaving the system is not a problem in Protocol 1, some users might silently stop participating in the protocol without notification. For this to not affect the liveliness of the protocol (by overrunning the capacity for offline parties), such users need to be removed from the system.

Acknowledgements

Joakim Brorsson was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Bernardo David was supported by the Independent Research Fund Denmark (IRFD) grant numbers 9040-00399B and 0165-00079B. Paul Stankovski Wagner was supported by the Swedish Foundation for Strategic Research grant RIT17-0035, and the Swedish Research Council grant 2019-04166.

References

[1] sslmate, "Timeline of certificate authority failures," https://sslmate.com/resources/certificate_authority_failures, 2024, [Accessed 30-06-2024].

[2] B. Post, "Improved digital certificate security," https://security.googleblog.com/2015/09/improved-digital-certificate-security.html, 2015, [Accessed 12-02-2024].

[3] T. H. Hoogstraaten, D. Niggebrugge, D. Heppener, F. Groenewegen, J. Wettinck, K. Strooy, P. Arends, P. Pols, R. Kouprie, S. Moorrees *et al.*, "Black tulip," Tech. Rep.(Fox-IT BV, 2012), Tech. Rep., 2012.

[4] B. Laurie, A. Langley, E. Kasper, E. Messeri, and R. Stradling, "Certificate Transparency Version 2.0," RFC 9162, Dec. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9162

[5] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: Bringing key transparency to end users," in *USENIX Security 2015*, J. Jung and T. Holz, Eds. USENIX Association, Aug. 2015, pp. 383–398.

[6] Y. Hu, K. Hooshmand, H. Kalidhindi, S. J. Yang, and R. A. Popa, "Merkle^2: A low-latency transparency log system," in *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021, pp. 285–303.

[7] M. Chase, A. Deshpande, E. Ghosh, and H. Malvai, "SEEMless: Secure end-to-end encrypted messaging with less trust," in *ACM CCS 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM Press, Nov. 2019, pp. 1639–1656.

[8] H. Malvai, L. Kokoris-Kogias, A. Sonnino, E. Ghosh, E. Oztürk, K. Lewi, and S. Lawlor, "Parakeet: Practical key transparency for end-to-end encrypted messaging," *Cryptology ePrint Archive*, 2023.

[9] J. Len, M. Chase, E. Ghosh, K. Laine, and R. C. Moreno, "{OPTIKS}: An optimized key transparency system," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4355–4372.

[10] Meta, "Whatsapp key transparency overview," 2023, [Accessed 12-02-2024].

[11] Apple, "Advancing imessage security: imessage contact key verification," 2023, [Accessed 12-02-2024].

[12] J. Blum, S. Booth, B. Chen, O. Gal, M. Krohn, J. Len, K. Lyons, A. Marcedone, M. Maxim, M. E. Mou *et al.*, "Zoom cryptography whitepaper," 2022.

[13] J. Bonneau, "Ethiks: Using ethereum to audit a coniks key transparency log," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 95–105.

[14] A. Tomescu and S. Devadas, "Catena: Efficient non-equivocation via bitcoin," in *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2017, pp. 393–409.

[15] B. McMillion, "Key Transparency Architecture," Internet Engineering Task Force, Internet-Draft draft-ietf-keytrans-architecture-00, Jan. 2024, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-keytrans-architecture/00/

[16] E. Ghosh and M. Chase, "Weak consistency mode in key transparency: Optiks," *Cryptology ePrint Archive*, 2024.

[17] F. Post, "Trust Asia 2021 has produced inconsistent STHs — groups.google.com," https://groups.google.com/a/chromium.org/g/ct-policy/c/VJaSg717m9g, 2020, [Accessed 01-02-2024].

[18] ——, "Upcoming CT Log Removal: Izenpe — groups.google.com," https://groups.google.com/a/chromium.org/g/ct-policy/c/qOorKuhL1vA, 2016, [Accessed 01-02-2024].

[19] ——, "Upcoming Log Removal: Venafi CT Log Server — groups.google.com," https://groups.google.com/a/chromium.org/g/ct-policy/c/KMAcNT3asTQ, 2017, [Accessed 01-02-2024].

[20] L. Chuat, P. Szalachowski, A. Perrig, B. Laurie, and E. Messeri, "Efficient gossip protocols for verifying the consistency of certificate logs," in *2015 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2015, pp. 415–423.

[21] R. Dahlberg, T. Pulls, J. Vestin, T. Høiland-Jørgensen, and A. Kassler, "Aggregation-based certificate transparency gossip," *arXiv preprint arXiv:1806.08817*, 2018.

[22] L. Nordberg, D. K. Gillmor, and T. Ritter, "Gossiping in CT," Internet Engineering Task Force, Internet-Draft draft-ietf-trans-gossip-05, Jan. 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-trans-gossip/05/

[23] L. Dykcik, L. Chuat, P. Szalachowski, and A. Perrig, "Blockpki: An automated, resilient, and transparent public-key infrastructure," in *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 2018, pp. 105–114.

[24] A. Dirksen, D. Klein, R. Michael, T. Stehr, K. Rieck, and M. Johns, "Logpicker: Strengthening certificate transparency against covert adversaries." *Proc. Priv. Enhancing Technol.*, vol. 2021, no. 4, pp. 184–202, 2021.

[25] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized business review*, 2008.

[26] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 51–68.

[27] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.

[28] B. David, P. Gazi, A. Kiayias, and A. Russell, "Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain," in *EUROCRYPT 2018, Part II*, ser. LNCS, J. B. Nielsen and V. Rijmen, Eds., vol. 10821. Springer, Heidelberg, Apr. / May 2018, pp. 66–98.

[29] P. Daian, R. Pass, and E. Shi, "Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake," in *FC 2019*, ser. LNCS, I. Goldberg and T. Moore, Eds., vol. 11598. Springer, Heidelberg, Feb. 2019, pp. 23–41.

[30] F. Benhamouda, C. Gentry, S. Gorbunov, S. Halevi, H. Krawczyk, C. Lin, T. Rabin, and L. Reyzin, "Can a public blockchain keep a secret?" in *TCC 2020, Part I*, ser. LNCS, R. Pass and K. Pietrzak, Eds., vol. 12550. Springer, Heidelberg, Nov. 2020, pp. 260–290.

[31] B. David, B. Magri, C. Matt, J. B. Nielsen, and D. Tschudi, "GearBox: Optimal-size shard committees by leveraging the safety-liveness dichotomy," in *ACM CCS 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM Press, Nov. 2022, pp. 683–696.

[32] G. Bracha, "An o(\mathop lg n) expected rounds randomized byzantine generals protocol," in *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, R. Sedgewick, Ed. ACM, 1985, pp. 316–326. [Online]. Available: https://doi.org/10.1145/22145.22180

[33] Y. Dodis and A. Yampolskiy, "A verifiable random function with short proofs and keys," in *PKC 2005*, ser. LNCS, S. Vaudenay, Ed., vol. 3386. Springer, Heidelberg, Jan. 2005, pp. 416–431.

[34] J. A. Buchmann, E. Dahmen, and A. Hülsing, "XMSS - A practical forward secure signature scheme based on minimal security assumptions," in *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*, B.-Y. Yang, Ed. Springer, Heidelberg, Nov. / Dec. 2011, pp. 117–129.

[35] K. Choi, A. Manoj, and J. Bonneau, "SoK: Distributed randomness beacons," in *2023 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2023, pp. 75–92.

[36] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *CRYPTO 2017, Part I*, ser. LNCS, J. Katz and H. Shacham, Eds., vol. 10401. Springer, Heidelberg, Aug. 2017, pp. 357–388.

[37] Anonymous. (2024) Code for implementation estimates. [Online]. Available: https://anonymous.4open.science/r/SpeedTestCoD-0286/README.md

[38] A. K. D. De Oliveira, R. Cabral *et al.*, "High performance of hash-based signature schemes," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 3, 2017.

[39] J. Groth, "On the size of pairing-based non-interactive arguments," in *EUROCRYPT 2016, Part II*, ser. LNCS, M. Fischlin and J.-S. Coron, Eds., vol. 9666. Springer, Heidelberg, May 2016, pp. 305–326.

[40] Anonymous. (2024) Code for statistical analysis. [Online]. Available: https://anonymous.4open.science/r/quorum-stats-1742/README.md

[41] Cloudflare. (2024) Nimbus2024. [Accessed 30-06-2024]. [Online]. Available: https://ct.cloudflare.com/logs/nimbus2024

[42] D. Ma and G. Tsudik, "Extended abstract: Forward-secure sequential aggregate authentication," in *2007 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2007, pp. 86–91.

[43] B. David, R. Dowsley, A. Konring, and M. Larangeira, "MUSEN: Aggregatable key-evolving verifiable random functions and applications," Cryptology ePrint Archive, Paper 2024/628, 2024, https://eprint.iacr.org/2024/628 (To Appear in SCN'24). [Online]. Available: https://eprint.iacr.org/2024/628

[44] G. Malavolta, "Key-homomorphic and aggregate verifiable random functions," Cryptology ePrint Archive, Paper 2024/643, 2024, https://eprint.iacr.org/2024/643. [Online]. Available: https://eprint.iacr.org/2024/643

[45] N. Fleischhacker, M. Hall-Andersen, M. Simkin, and B. Wagner, "Jackpot: Non-interactive aggregatable lotteries," Cryptology ePrint Archive, Paper 2023/1570, 2023, https://eprint.iacr.org/2023/1570. [Online]. Available: https://eprint.iacr.org/2023/1570

[46] M. Oxford, D. Parker, and M. Ryan, "Quantitative verification of certificate transparency gossip protocols," in *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2020, pp. 1–9.

[47] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping authorities" honest or bust" with decentralized witness cosigning," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 526–545.

[48] F. Benhamouda, T. Lepoint, J. Loss, M. Orrù, and M. Raykova, "On the (in)security of ROS," *Journal of Cryptology*, vol. 35, no. 4, p. 25, Oct. 2022.

Appendix A.
Security Analysis of the CoD Protocol (Proof of Theorem 4.1)

Proof (Completeness). An honest $\mathcal{CM}$ distributes to all parties the same view View_e with the same honestly generated signature Tkn_e (line 1 of Download, and 8 of Send in Protocol 1). Hence, all honest parties output the same, consistent view during Download. Further, since the VKD and SeedGen are correct, all checks for the updated parameters by endorsers (line 6 of Endorse) will verify for honestly computed values. In the Ver phase, the endorsement validity check (line 6 of CheckCon) passes for all (honest) endorsements, since KES and VRF are correct. Recall that the quorum parameter k is set so that $\mathcal{CM}$ will collect at least k endorsements of the same view, except with negligible probability (more details on this in Section 5). Using the notation of Section 5, the quorum check (line 8 of CheckCon) will pass with probability $\Pr(B(N,p) < k)$, where p denotes the probability of a user being selected as `endorser`. Denote by $negl_{\mathsf{X}}$ the probability that primitive X is not perfectly correct, then the probability that, in epoch $e = 1$, any honest user outputs `abort` during the execution of CheckCon is equal to

$$negl_{\mathsf{KES}}(\lambda) + negl_{\mathsf{SeedGen}}(\lambda) + negl_{\mathsf{Sig}}(\lambda) + negl_{\mathsf{VRF}}(\lambda) + \Pr(B(N,p) < k)$$

which is negligible for suitable choices of λ and k (see Figure 3b). This proves our CoD protocol complete. □

Proof (Consistency). This proof proceeds by induction on the epoch counter $e \geq 1$, and is presented in two main steps. The first step, the base case, proves that CoD is consistent for epoch $e = 1$. The second step, the induction step, proves CoD is consistent for any $e \geq 1$, by assuming that CoD is consistent up to the generation of epar_{e-1}.

STEP 1 (BASE CASE) Let $\mathcal{U}$ and $\widehat{\mathcal{U}}$ denote two distinct honest users that are in a consistent state at the beginning of the first epoch $(e = 1)$, i.e., $\mathcal{U}.\mathsf{state} = \texttt{consistent} = \widehat{\mathcal{U}}.\mathsf{state}$. Recall that epar_0 is generated honestly (CoD works in the CRS model, i.e., with the trusted setup Algorithm 1). To prove consistency for $e = 1$ as per Definition 3.3, we must show that the inequality described in Equation (1) in Definition 3.3 holds. We do this using the definition of conditional probability $\Pr[\mathsf{A}|\mathsf{B}] = \Pr[\mathsf{A} \cap \mathsf{B}]/\Pr(\mathsf{B})$, and showing that $\Pr[\mathsf{A} \cap \mathsf{B}]$ is negligible. For us $\mathsf{A} = E_1 \cap E_2 \cap E_3 \cap E_4$ (the events defined in Definition 3.3), while B is generated by the adversary (and thus has probability at least 1 over all possible outputs of $\mathcal{A}$, which is computationally bounded to polynomial many outputs).

We distinguish two cases depending on whether or not $\mathcal{A}$ corrupts the channel maintainer.

The case of an *honest* $\mathcal{CM}$: If $\mathcal{CM}$ is honest, its input to Download is identical when interacting with $\mathcal{U}$ and $\widehat{\mathcal{U}}$. Formally,

$$\mathsf{Download}\left\langle \begin{array}{c} \mathcal{U}(e) \\ \mathcal{CM}(\mathsf{View}_{\mathcal{CM}}, \mathsf{Tkn}_{\mathcal{CM}}, e) \end{array} \right\rangle \mapsto [\mathcal{U} : (\mathsf{View}_e, \mathsf{Tkn}_e)]$$

$$\mathsf{Download}\left\langle \begin{array}{c} \widehat{\mathcal{U}}(e) \\ \mathcal{CM}(\mathsf{View}_{\mathcal{CM}}, \mathsf{Tkn}_{\mathcal{CM}}, e) \end{array} \right\rangle \mapsto [\widehat{\mathcal{U}} : (\widehat{\mathsf{View}}_e, \widehat{\mathsf{Tkn}}_e)]$$

Notice that $\mathsf{View}_{\mathcal{CM}}$ may be different from the received View_e and $\widehat{\mathsf{View}}_e$ (and similarly for $\mathsf{Tkn}_{\mathcal{CM}}$). By the probability in (1) (see Definition 3.3), $\mathsf{View}_e \neq \widehat{\mathsf{View}}_e$, which implies $\mathsf{View}_e \neq \mathsf{View}_{\mathcal{CM}}$ (w.l.o.g). This means that $\mathcal{CM}$'s message $(\text{VIEW}, \mathsf{View}_{\mathcal{CM}}, \mathsf{Tkn}_{\mathcal{CM}})$ must have been suppressed by the adversary at the network level (between line 1 and 2 in Download) for at least one of the users, and replaced with an adversarial message. We argue that in this case $\Pr[E_1 \wedge E_2 \wedge E_3 \wedge E_4 \wedge \mathsf{B}] < \Pr[E_3 \cap \mathsf{B}]$ is negligible and upperbounded by the (negligible) probability of producing a forgery against the signature scheme Sig, i.e., $\Pr[E_3] < \Pr[1 = \mathsf{Exp}^{\text{EUF-CMA}}_{\mathsf{Sig}}(\mathcal{A})] = negl_{\mathsf{Sig}}(\lambda)$. Indeed, for event E_3 to happen, we have $\mathcal{U}.\mathsf{state} = \texttt{consistent}$ at the end of CheckCon, hence the signature check for $\mathsf{Tkn}_{\mathcal{A}}$ (line 1 in CheckCon) does not fail. Since $\mathcal{CM}$ never signed View_e, the message signature pair $(\mathsf{View}_e, \mathsf{Tkn}_{\mathcal{A}})$ constitutes a forgery against Sig. This violates the assumption that Sig is EUF-CMA.

The case of a *corrupt* $\mathcal{CM}$: By the probability in (1) (see Definition 3.3), $\mathsf{View}_e \neq \widehat{\mathsf{View}}_e$ and CheckCon must output $(\cdot, \texttt{consistent})$ for both $\mathcal{U}$ and $\widehat{\mathcal{U}}$. For the latter to happen, *i.e.* $(E_3 \wedge E_4)$, the checks in lines 1 ($\mathcal{CM}$ signature verification), 6 (endorsement validity), and 8 (quorum) must all pass for both users. In what follows we argue that $\Pr[E_3 \wedge E_4]$ is negligible and upperbounded by $negl_{\mathsf{KES}}(\lambda) + negl_{\mathsf{VRF}}(\lambda) + negl_{\mathsf{Qrm}}(N, f_M, f_I)$, where $negl_{\mathsf{KES}}(\lambda)$ denotes the probability that $\mathcal{A}$ generates a successful forgery against the KES scheme or breaks the forward security of KES, $negl_{\mathsf{VRF}}(\lambda)$ denotes the probability that $\mathcal{A}$ breaks the security of the VRF, and $negl_{\mathsf{Qrm}}(N, f_M, f_I)$ is the probability of selecting an endorsement committee without honest majority. Concrete estimates of $negl_{\mathsf{Qrm}}(N, f_M, f_I)$ are given in Figure 3b for $N = 2^{30}$, varying values of $f_M, f_I < 0.2$ and varying security levels in the interval $[2^{-256}, 2^{-30}]$.

Since $\mathcal{CM}$ is corrupt, $\mathcal{A}$ can generate valid signatures for any view, hence, for simplicity, we can assume that the signature check on line 1 of CheckCon always passes for both $\mathcal{U}$ and $\widehat{\mathcal{U}}$.

We first discuss the endorsement validity check at line 6. For this check to pass it must be the case that $\mathcal{A}$ provides two sets of endorsements $\mathsf{End} \neq \widehat{\mathsf{End}}$ that verify for different views $\mathsf{View} \neq \widehat{\mathsf{View}}$. Lines 1 and 2 in Endorse, and line 4 in Verify ensure that all endorsements are for the same seed s_{e-1}. In the first epoch, s_0 is part of the trusted CRS. The *uniqueness* property of the VRF ensures that one seed will lead to a unique set of endorsers (the same for both End and $\widehat{\mathsf{End}}$). The *pseudorandomness* property of the VRF ensures $\mathcal{A}$ cannot predict if a user will act as endorsers for s_0, unless the user is already corrupt, except with a negligible probability $negl_{\mathsf{VRF}}(\lambda)$. The *forward security* property of the KES after a key is evolved and securely erased ensures that even though $\mathcal{A}$ learns the identities of each honest endorser (at line 11 of Endorse), even if it targets them for corruption, it does not learn the KES secret key needed to generate endorsements for this epoch. Thus, in the worst case scenario, the adversary has corrupted a fraction of f_M active users, *for which $\mathcal{A}$ holds the* KES *secret key of this epoch*. Note that for all corrupt users acting as endorsers, $\mathcal{A}$ can produce correct KES signatures for any view. This is not the case for honest endorsers, in which case, generating a valid KES signature for the other view gives a forgery against the KES scheme, which happens with negligible probability $negl_{\mathsf{KES}}(\lambda)$ since KES is *unforgeable* and *forward secure* given secure erasures.

Next, we discuss the the quorum check at line 8. For this check to pass $\mathcal{A}$ must have submitted k many valid endorsements for both View and $\widehat{\mathsf{View}}$. The quorum value k can be set so that the probability that this happens is as low as desirable. Suitable values of k are displayed in Figure 3b, e.g., for a bit security level $b = 128$ (i.e., the quorum check fails with probability 2^{-128}), $k = 20,537$ is a suitable quorum value for $N = 2^{30}$, $f_M = f_I = 0.2$. These figures are backed by the mathematical arguments based on combinatorics and probability theory detailed in Section 5.

Intuitively, the VRF's security properties guarantee that $\mathcal{A}$ cannot bias the committee of endorsers to contain much more than an expected number of corrupt endorsers; at the same time, the KES's security properties guarantee that $\mathcal{A}$ cannot forge signatures for honest endorsers. Therefore, as long as the number of expected honest endorsers is *comfortably larger* than the number of malicious ones, it is possible to identify a value k for which the probability of obtaining *at least k honest endorsers is high*, and the probability of obtaining *at most k malicious endorsers is low*. This is sufficient in settings with GOD, where all honest endorsements will reach all users (see Section 5.1). In CoD, however, (honest) endorsements may be suppressed without notice. To achieve consistency (and deter split-view attacks) without relying on GOD, the quorum value k must be set differently (see Section 5.2). Denote by k_M the expected number of malicious endorsers randomly selected by the cryptographic sortition process (lines 1 to 3 in Endorse). Then, k is set as before, but additionally ensuring that the probability that $k_M + (k/2) > k$ *is low*. This is because $\mathcal{A}$ can generate valid endorsements for the legitimate k_M malicious endorsers, in addition it can distribute View to half of the user (among which there are, in expectation, half of the endorsers) and $\widehat{\mathsf{View}}$ to the remaining half of the users. Hence, $\mathcal{A}$ can easily get $k_M + (k/2)$ endorsements for both views. Therefore consistency is guaranteed as long as the quorum is safely larger than $k_M + (k/2)$, since in this case either $\mathcal{U}$ or $\widehat{\mathcal{U}}$ will not collect enough endorsements to pass the quorum check and end up in $\texttt{abort}$. More formally, if $\mathcal{A}$ generates $(\mathsf{View}, \mathsf{Tkn}, \mathsf{End})$ and $(\widehat{\mathsf{View}}, \widehat{\mathsf{Tkn}}, \widehat{\mathsf{End}})$ with $\mathsf{View} \neq \widehat{\mathsf{View}}$ and that pass the endorsement validity check,

then $\Pr[(|\mathsf{End}| \geq k) \wedge (|\widehat{\mathsf{End}}| \geq k)] \leq negl_{\mathsf{Qrm}}(N, f_M, f_I)$. Figure 3 displays selected values for the quorum function $negl_{\mathsf{Qrm}}(N, f_M, f_I)$.

Putting all pieces together (in the notation of Definition 3.3), $\Pr[E_1 \wedge E_2] = 1$ since we consider an active malicious adversary that controls the communication network. Let $(\mathsf{View}, \mathsf{Tkn}, \mathsf{End})$ and $(\widehat{\mathsf{View}}, \widehat{\mathsf{Tkn}}, \widehat{\mathsf{End}})$ denote $\mathcal{A}$'s split-view attack output (against the consistency of our CoD protocol), then we have

$$\Pr\left[E_3 \ \wedge \ E_4 | \, \mathsf{View} \neq \widehat{\mathsf{View}}\right] \leq$$
$$\frac{\Pr\left[(E_3 \ \wedge \ E_4) \wedge \ (\mathsf{View} \neq \widehat{\mathsf{View}})\right]}{\Pr[\mathsf{View} \neq \widehat{\mathsf{View}}]} <$$
$$negl_{\mathsf{KES}}(\lambda) + negl_{\mathsf{VRF}}(\lambda) + negl_{\mathsf{Qrm}}(N, f_M, f_I),$$

which is negligible for suitable parameters choices.

This proves that any two users who are in a consistent state at the end of epoch $e = 1$, must hold the same View (consistency). In particular, *this implies that all honest users in a consistent state will obtain the same set of parameters for the next epoch*. Since epar_1 generated by $\mathcal{CM}$ is verified by the endorsers before they endorse View_1 and we have a quorum of endorsements on View_1, we are also guaranteed that epar_1 is valid and consistent with epar_0 and View_1.

STEP 2 (INDUCTION) The inductive step assumes that CoD is consistent up to the generation of epar_{e-1}, and aims to prove that CoD is consistent in generating epar_e.

Since the generation of epar_{e-1} is consistent across honest active users, it replaces the parameters from the CRS in the consistency proof of the base step, which concludes arguing that honest users will obtain the same (consistent) set of parameters for the next epoch, epar_e. We must however ensure that epar_e fulfills the same requirements as the initial CRS output by the trusted setup. The fact that epar_e is obtained consistently by all honest users and generated honestly by $\mathcal{CM}$ due to the existence of a quorum of endorsements on View_e is one fundamental property. We have that epar_e must guarantee that the set of public keys $\mathcal{PK}$ is correct for the set of users in the system, and the new seed s_e is *random*. Recall that we have assumed (in Section 4.1.1) that each user (including newly added users) has registered a single key to the IdP/VKD, thus if $\mathcal{PK}$ at epoch e is an append-only update of the previous epoch's $\mathcal{PK}$, the set of public keys is correct. This property follows from the soundness and append-only properties of the VKD. We discuss how to securely instantiate SeedGen in Section 4.3. □

Appendix B. Improving Efficiency via Aggregate Signatures and VRFs (or lotteries)

Our CoD scheme has computational, communication and storage costs linear in the quorum size: each endorsement (signature and VRF) needs to be stored, downloaded and verified individually. This can be improved by instantiating CoD with forward secure aggregate signatures [42] and aggregate VRFs [43], [44], which would yield considerable improvements in computational, communication and storage complexity. Aggregate signatures allow for a constant size representation of the signatures on the VKD digest by each committee member. On the other hand, aggregate VRFs allow for a constant size representation of the VRF proofs provided by each committee member, albeit still requiring an individual representation of each VRF output. Hence, instead of requiring storing/downloading/verifying n signatures and n pairs of VRF output/proofs, an endorsement would require storing one group element for the aggregate signature, storing 1 group element for the aggregate VRF proof and n strings of security parameter length, which can be verified by computing $2n$ bilinear pairings and $2n$ hash functions evaluations. Considering the VRF-based randomness generation scheme described in Section 4.3.3, one can take advantage of the MUSEN [43] scheme to further reduce complexity by using MUSEN both as a VRF and as a forward secure signature. In this setting, an endorsement by n parties requires storing only n group elements and verifying an endorsement requires only n bilinear pairings and n hash function computations.

An alternative solution is employing a different scheme for electiing random anonymous committee based on aggregate lotteries [45], which allow for a constant size representation of the proofs that certain parties have been elected for a committee. Following this approach, the complexity of storing an endorsement becomes constant, requiring one group element for the forward secure aggregate signature and 80 bytes for the lottery proof. Verifying the forward secure aggregate signature still requires computing n bilinear pairings and n hash functions and verifying the aggregate lottery proof requires computing $2n$ modular exponentiations.

Appendix C. Literature Review of Consistency Protocols

Consistency Via Blockchains Blockchains [25], [26] securely handle financial transactions without involving any central party, and have broadcast-like properties. Thus, one can store data such as digests of a VKD/PAHD on a blockchain to obtain consistency guarantees. Since blockchains come with extended security assumptions and performance penalties, one can argue that adding them only to use them for consistency is unnecessarily expensive.

Dykcik et al. [23], presents a straightforward scheme which incorporates a transparency log into a smart contract on a blockchain. Tomescu and Devadas in their Cathena system [14] design a scheme which leverages Bitcoin to provide consistency. Performance improvements over a straightforward blockchain for the entire log is presented. Bonneau in EthIKS [13] proposes to embed CONIKS data structures in an Ethereum contract, allowing to piggyback on Ethereum's consensus protocol for security guarantees.

Consistency Via Gossip Protocols Gossiping over OOB channels are the suggested method of achieving consistency

by Certificate Transparency [4] and in a recent internet draft for Key Transparency [15]. When using a gossip protocol for consistency, the log maintainer provides a digest of the log state at recurring intervals to users. Users then exchange these digests over OOB channels and compare them. Inconsistencies between these digests can thus be detected during comparison, and constitutes cryptographic proof of misbehavior which can be be reported. Comparing digests from the log maintainer is done on a voluntary basis.

This approach, while avoiding blockchains and its drawbacks, only give statistical probabilities for detecting split-views (which are quite low as discussed below) rather than formal guarantees about consistency. Further, gossiping is done retroactively, meaning that split-view attacks will only be detected after they have occurred. Gossiping proactively would not be practical due to the "soft" guarantees and long waiting times. While this makes undetected split-view attacks harder, it does *not* rule them out, and users thus do not have guarantees that a public key is safe to use.

A further drawback of gossip protocols in the context of key transparency, is that they rely upon using OOB channels. If OOB channels are not used, an adversary controlling the network can suppress messages from targeted users to evade detection of an attack (or even alter them through Meddler-in-the-Middle attacks as we discuss below). This is problematic since OOB channels are not available at scale where KT protocols are deployed.

Chuat et al. [20] propose the first gossip protocols for Certificate Transparency and present results for how signed views are distributed using a simulation based on real Internet traffic traces at a 0.1% gossiping rate (the fraction of parties volunteering to gossip). In their results, after 24 hours, 11% of the users holds a signed view signed during this period. No results for the probability and speed of detection of log inconsistencies are presented.

The Chuat et al. gossip protocols are also evaluated in [46] by Oxford et al. Assuming a gossiping rate of 100%, they measure data dissemination, probability of split-view detection and rate of such detections. In their analysis, after 20 gossiping rounds the latest view is disseminated to all users, and a split-view attack is detected with 40% probability. They also show that if one makes the extra assumption of server-to-server gossip, these numbers can be improved.

Dahlberg et al. [21] explore how the network infrastructure such as routers and switches can provide gossiping as a service. Nordberg et al. [22] proposes an Internet-Draft where gossiping takes place between a server and a client, so that if a meddler-in-the-middle attack occurs, it will be detected once the attack ends and the client has established contact directly with the correct server.

We note that Apple's approach to gossiping in iMessage [11], where they piggyback gossip data on end-to-end encrypted messages over in-band channels (*i.e.*, via Apple), does not solve this issue of in-band gossiping. Even though the gossip data is now encrypted, the log maintainer (Apple) can execute a split-view attack where it replaces public keys given to the victim, and act as a Meddler-in-the-Middle which alters gossip data. Such an attack can be sustained indefinitely (assuming it does not break append-only guarantees). To the best of the authors knowledge, Apple has not provided a security analysis for this approach.

Consistency Via External Committees of Consistency Auditors To avoid both the cost of blockchains and weak guarantees of gossip protocols, one can instead opt to use a set of external consistency auditors. These external auditors are centrally selected as a static set of parties. During the protocol execution, the log maintainer sends a log digest to all consistency auditors in each epoch. The auditors then sign these. So, instead of comparing commitments directly as in gossiping, users can compare their view with the signed views of the consistency auditors to see if it corresponds with a majority of the auditors. This approach provides both pro-active (immediate) detection of misbehaviour as well as formal guarantees, as long as the set of auditors has a sufficiently large fraction of honest parties.

The drawback however, is that it sacrifices the distributed nature of using blockchains or gossiping. Consistency guarantees in blockchains and gossip protocols are rooted in the honesty of a very large set of users (all miners/stakers or all KT users). It is unlikely that even a powerful adversary can corrupt the majority of such large sets of users. The same can not be said about a small and well known set of consistency auditors. Current proposals suggest committee sizes of roughly 50 [8] auditors. A powerful adversary (say, *e.g.*, a state sponsored adversary) can realistically corrupt most, if not all, of the auditors in a set of this size.

Parakeet by Malvai et al. [8], presents a simple protocol for a fixed set of external auditors which all sign their view, (up to 50 auditors are simulated in the paper). In order to prove that there is no split-view attack, the identity provider needs to obtain a threshold of two thirds of auditor signatures agreeing on a single view. Dirksen et al. [24] let a set of Certificate Transparency logs act as consistency auditors for each other by pitting them against each other, where for each cert, it is included in one log only and the other logs are expected to audit this log.

While increasing the number of external auditors would somewhat increase the resilience against split-view attacks, it would still not achieve the distribution of blockchains or gossip protocols, and thus not live up to the same level of resilience. Further, scaling up the number of auditors is non-trivial. First, it is a problem in practice to find a large set of auditors which all users trust to have an honest majority. Second, it poses an efficiency problem for users which have to validate authenticity and correctness of the auditors' statements. For example, the overhead of Parakeet's consistency protocol scales linearly with the number of auditors.

We note that Syta et al. [47] proposed a protocol for auditor cosigning with sublinear verification overhead, where auditors interact with each other to combine their signatures into a single multisignature. This protocol has however since been broken [48], and even if it were secure it would only scale to thousands of auditors, compared to millions or billions in blockchains or gossiping.

Weaker Versions of Consistency The work in [16]

provides a protocol which does not use external parties for auditing consistency. This is achieved by weakening the consistency guarantees of KT, so that the protocol only ensures that split-views are detected by *either* the party who queries for a key, *or* the key owner.