CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Realizing Consistency-or-Die: Verifiable Consistency for Key Logs

Master's thesis in Computer Science and Engineering

PEDRO FLORINDO

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Realizing Consistency-or-Die: Verifiable Consistency for Key Logs

PEDRO FLORINDO

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Realizing Consistency-or-Die: Verifiable Consistency for Key Logs

PEDRO FLORINDO

Supervisor: Elena Pagnin, Computer Science and Engineering
Examiner: Magnus Almgren, Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Realizing Consistency-or-Die: Verifiable Consistency for Key Logs

PEDRO FLORINDO
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Public key cryptography has become an important backbone for end-to-end encrypted communication, commonly used in the majority of the most popular messaging applications, such as WhatsApp and Signal. For this purpose, these applications utilize a centralized key log, which users can query to receive the public key of their desired recipient. However, this can open the possibility of a split-view attack, in which two users receive different information on what keys are registered.

To prevent these attacks, the key log must be consistent, meaning there needs to be a way to confirm that all legitimate users receive the same information. While there are already some methods that try to enforce consistency, they either rely on users trusting third parties, or are unscalable to billions of users.

Consistency-or-Die is a novel solution which utilizes the large user base together with verifiable randomness to generate endorsements from an ever-changing fraction of users, which can then be used by participants to check consistency. While this approach is promising, it still has not been tested experimentally, and has some theoretical gaps which require additional work, such as the maximum permitted fraction of malicious users along with specifications of how the random seed generation occurs.

This thesis presents a concrete design and implementation of the protocol, capable of being tested, and addresses some remaining theoretical challenges. It presents the necessary background required to understand Consistency-or-Die, before explaining the protocol itself, followed by the implementation specifications, design choices and expected execution. Then, it proves that the maximum number of malicious users CoD can efficiently handle is one third of the entire population, and discuss how realistic this scenario is. Furthermore, it analyzes the requirements of seed generation, multiple approaches to generate a random seed and their respective security considerations. Finally, a discussion on the obtained results and future areas of this protocol that could also be improved as well as possible future applications is presented.

Keywords: Key Transparency, Consistency, Split-View Attack, Verifiable Key Directories

Acknowledgements

I would like to deeply thank my supervisor, Elena Pagnin, as well as her PhD student, Lucia Lavagnino, for their invaluable assistance, feedback, and suggestions throughout this thesis. Without their support, I would not have been able to complete it.

I would also like to thank my family for everything they have done to give me this opportunity, and all the help they gave me along the way.

Pedro Florindo, Gothenburg, 2025-07-05

Contents

List of Figures

List of Tables

1

Introduction

1.1 Motivation

Public key cryptography has become extremely popular as a method of securing online communications. It is key for multiple different cryptographic protocols, with two specific applications being Public Key Encryption (PKE) and Public Key Signatures. PKE allows users to encrypt their messages and guarantees that only the target recipient can successfully decrypt and understand them, while anyone else who accesses the encrypted message is unable to retrieve its contents. Figure 1.1 shows how Public Key Encryption secures communication between two users.

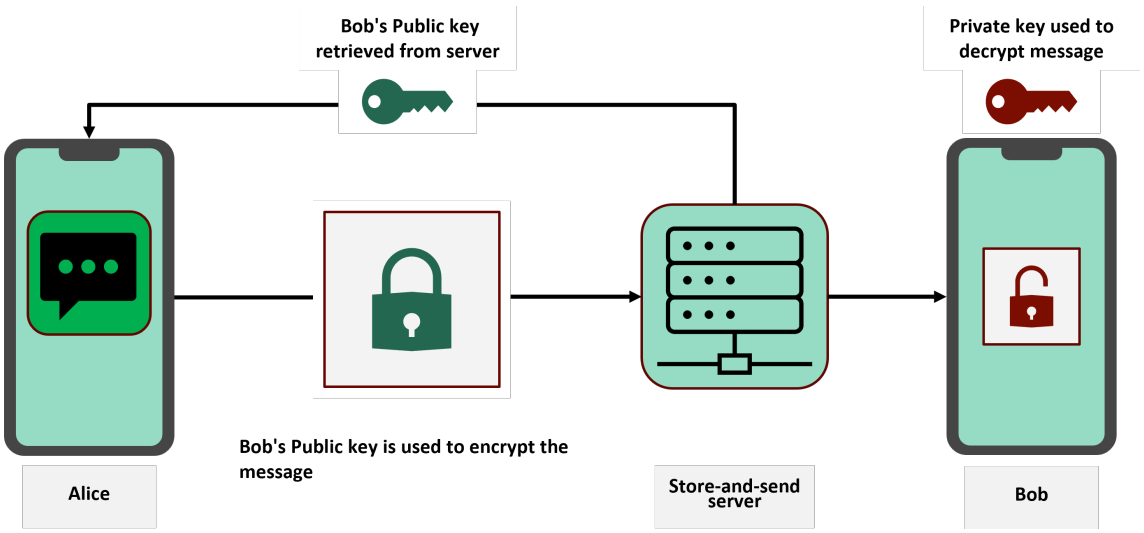


Figure 1.1: Public Key Encryption Diagram

However, users must know each other's public keys in order to encrypt their messages and verify signatures, which raises the problem of how users initially acquire each other's keys in such an application. A solution is to utilize a public key log, in which users can ask the service provider for either the full log or a particular key and receive a response with the necessary information and a proof of inclusion of the key in the log. Since all users will be directly connected to the central server, this can be seen as a network with a star topology, where instead of having any peer-to-peer connections, users only communicate directly with the central entity.

Key transparency logs [1] were introduced with this goal in mind. They are records that keep track of the history of public keys used in a system, which users can regularly access and update. In key transparency logs, two main properties are imperative: append-only, which states that any update from a log must be a mere extension of a previous log with only the addition of keys, with none being removed or altered, and consistency, which states that any two users who request access to the log at the same time must receive the same view. The first is crucial, as it prevents attackers from being able to delete or alter previous keys and lie about what information was present in the log at any given point in time. Meanwhile, the second is necessary to prevent split-view attacks from a corrupt log maintainer, preventing targeted attacks with the intention of deceiving users.

A split-view attack is a state of divergence in which two distinct users receive different views of a shared resource, with each one believing they see the correct information and assuming all other users possess the same record. A straightforward example is assuming two users, Alice and Bob, each with a secret and public key, and a compromised server that keeps track of a log containing all the public keys required for communication. When Alice and Bob request the current state of the log, in order to obtain each other's public keys, the corrupt log maintainer can distribute two different views to each, one in which Alice sees a forged public key in place of Bob's, and another in which Bob sees the forged key in place of Alice's. Then, when Alice tries to send a message to Bob, she will encrypt it with what she believes is Bob's public key, which in reality is the adversary's key, and then send the message. The server can intercept, decrypt, read, and possibly modify the message, before encrypting it with Bob's real public key and forwarding it. This is known as a man-in-the-middle attack [2] and is one of multiple possible attack vectors that are enabled after a successful split-view attack. Clearly, there needs to be a way to verify that users are in fact seeing the same view of the log at any given time.

Current solutions [1], [3], [4] for achieving consistency in key transparency logs are still lacking, with three main groups of proposals,namely gossip protocols, blockchains and external auditors, each having their own shortcomings. They are either not scalable up to the user base messaging applications need to handle, are too computationally intensive for the majority of users to efficiently run them, or only provide weak security guarantees that are vulnerable to powerful attackers.

Consistency-or-Die (CoD) [5] is a recent alternative proposal that introduces a novel concept of a hidden council of endorsers, which changes over time in a verifiable way and is responsible for endorsing the view it receives. These endorsements are then sent back to the channel maintainer, who later distributes them to all users. Users compare the endorsed views with the ones they originally received, and, if they obtain enough endorsements that match their own perceived state of the log, they can be confident that all other honest and consistent users share that same view. If not enough endorsements are received, the user cannot trust that they are in a consistent state and thus ceases participation in the protocol (i.e., dies). The number of match-

ing endorsements required for users to be confident in their state is called a *quorum* and can be derived using the total number of users and respective percentages of honest, malicious, and inactive users, as well as the desired level of bit security, i.e., the acceptable probability of failure in the protocol, and the chance that any given user is chosen as a member of the committee.

The four main steps of updating the log, sending it to users, receiving endorsements, and publishing them are the building blocks of a round of the protocol, called an epoch, which is how time is tracked in an execution of the protocol. If at the end of the epoch a user receives at least a quorum of endorsements that match the view it received, it can be confident that it is consistent with all other honest users who successfully completed the epoch.

Users are selected to be endorsers through a randomness seed along with a private identifier to check whether they should act as endorsers for the current epoch. This check also creates a proof of generation, which other users can use to confirm the endorsements they receive are legitimate. This randomness forces different users to take on this role in each epoch in an unpredictable, but verifiable manner, allowing for protection against attackers who target participants who acted as endorsers in previous epochs.

1.2 Purpose and Goals

While CoD is an extremely promising protocol in concept, it has yet to be tested experimentally, with there being no implementation made that demonstrates the procedure. This is what this thesis intends to address, by creating a proof-of-concept implementation, which enables practical testing of the protocol's potential efficiency. While the demonstration is not production-level, it serves the purpose of giving a basis for future applications, while also allowing for an initial analysis of CoD's scalability, highlighting what real-scale implementations might need to consider and improve.

Furthermore, there are some additional areas in CoD that need to be more clearly defined, such as the maximum fraction between malicious and honest users that CoD can handle efficiently, and an analysis of the security considerations that come with different variants of the function used to generate the randomness used in committee selection.

The objective of this thesis is to investigate and explore solutions to these missing pieces, with the goal of validating Consistency-or-Die so that it can be used as a solid foundation for ensuring consistency in key transparency logs, which has large-scale applications in some of the world's largest messaging platforms.

The objectives of this thesis are clear:

- Develop a proof-of-concept implementation for CoD following the concept laid out in the original paper [5], capable of simulating real-world scenarios the protocol needs to handle and test its efficiency in such cases.

- Simulate a split-view attack and its possible consequences in a system that does not offer consistency.

- Analyze and compare three different methods of generating a randomness seed, and the security and practical trade-offs each one brings, determining which is most suitable for a real-world implementation of CoD.

- Discover the maximum fraction between honest and malicious users for which CoD is still efficient, and how this ratio impacts scalability.

1.3 Scope and Limitations

Implementing a cryptographic protocol from scratch requires extreme care and consideration, especially one that requires multiple cryptographic primitives like CoD. For this reason, some core building blocks of the protocol come from pre-existing libraries, like OpenSSL. Pre-existing code such as verifiable random functions [6] and key evolving signatures [7] was also used.

While CoD is a protocol intended to be used in large scale messaging applications, it is not feasible to simulate real user connections to the scale the protocol is expected to handle in a proof-of-concept implementation. As is shown later, the implementation's speed scales linearly with the number of endorsements, which in turn scales with the number of users. For testing purposes, fewer users were used, each of whom sends multiple signatures in order to approximate the behavior that is expected for quorum sizes that would be used in cases with billions of users, allowing the extrapolation of the performance of the protocol for such scenarios. This was necessary as it is simply unfeasible to simulate billions of users for testing, but the experimental results acquired are still representative of a real execution.

Furthermore, it is expected that the majority of the clients of these applications are running in mobile devices, which have reduced computational power when compared to regular computers. While the current implementation does not support mobile devices, in order to test the efficiency in these scenarios, a weaker machine was used, restricted in terms of available computational power in order to try to approximate behavior for a mobile device.

1.4 Outline

The thesis has the following structure:

- Chapter 1 presents the motivation for the thesis, as well as its goals.
- Chapter 2 begins with the currently proposed solutions to the problem of key transparency logs and current techniques for key consistency.
- Chapter 3 gives an overview of the main cryptographic building blocks and principles CoD takes advantage of before diving into an explanation of the protocol itself.
- Chapter 4 details the implementation that was created, design choices made in it, as well as minor differences between it and the original specification of the protocol that were necessary for the realization of the project.
- Chapter 5 contains the theoretical analysis for the calculation of the maximum ratio between malicious and honest users the protocol can efficiently handle, followed by a discussion on possible approaches to generate the randomness seed required for endorsement selection, along with the risks and considerations each requires.
- Chapter 6 presents the results acquired from testing the implementation, and discusses what these results imply for the viability of CoD as a consistency protocol in large-scale deployments.
- Chapter 7 concludes the thesis with a discussion on the results of the implementation and its overall design, closing by presenting other possible research vectors for future work on this protocol.

2

Related Work

The need for transparency in key logs used for secure communication has been raised for almost a decade [1]. Transparency is achieved when two sub-properties hold simultaneously: Correctness and Consistency. Correctness, also referenced as the Append-Only property, implies that any log must be resistant to changes to already existing data in memory. This means that both unauthorized deletions of pre-existing keys and unauthorized alterations to them are forbidden and must be detectable.

On the other hand, consistency implies that all users agree on the new additions to the log, meaning all honest users can reliably confirm they share the same information as all others and can agree that it is the correct information.

In order to understand the necessity for better consistency protocols, this section gives an overview of current techniques for key transparency, both for achieving correctness and consistency, and distinctions between consistency, consensus, and Byzantine Fault Tolerance.

Currently, the Append-Only property has multiple solutions that have reached production level, which are applicable to multiple real-world scenarios. Research on guaranteeing consistency has not shown the same results. Here, I will explore the current solutions for both of these properties, as well as compare the consistency protocols currently proposed and in circulation with Consistency-or-Die, while stating their differences.

2.1 Verifiable Key Directories

This section gives an overview of the history of the proposals of solutions for key transparency, presenting their distinguishing features, considerations in terms of realism and privacy, and the chosen consistency methods of each. A comparison between each method can be seen in Table 2.1, comparing specific attributes of each, along with the choice of gossip between users and the use of external parties for consistency verification.

Feature	CONIKS	Parakeet	ELEKTRA	OPTIKS
VKD	✓	✓	✓	✓
MVKD	✗	✗	✓	✗
Non-membership proof	✗	✗	✗	✓
Compaction	✗	✓	✗	✗
Key rotation	✓	✓	✓	✓
Consistency	Gossip	Ext.Party	Ext.Party	Gossip

Table 2.1: Feature comparison of key transparency schemes. Features compared are the existence of a Verifiable Key Directory/Multi-Device Verifiable Key Directory, the capacity to present Non-Membership proofs for any given entry, Key Log Compaction, the ability for users to rotate their keys, and Consistency methods chosen, between Gossip and an external party of auditors/witnesses.

2.1.1 CONIKS

CONIKS [1] was the first suggestion for a solution to the problem of key transparency for public-key directories. Presented in 2015, it was inspired by the problem of certificate transparency and created to solve a similar problem for key registers.

It utilizes a Merkle Prefix Tree, where user inputs are stored to create an ordered structure that keeps track of public keys. Keys are stored at a specific index corresponding to each user, calculated using a Verifiable Random Function that outputs a unique index given the username of the key holder as input. Entry locations reveal nothing about the user's identity to outsiders, as only the VRF output is known to them, preserving privacy.

Every midnight, the server updates and signs the new hash tree, which represents the entire content of the log. This hash shows that the log is a mere extension of the previous one. To prove this, every new addition or modification of the log stores the string of the previous epoch's hash. This pins the server to the previous round, ensuring it did not cheat by changing or removing entries.

Clients can ask the server for the key associated with a given username, to which the server responds with the desired public key, the VRF proof (which shows the index corresponds uniquely to the queried username), the previous epoch's reference string, and the Merkle path associated with the index. The client can verify the proof and recreate the path to the desired key, confirming that it is correct.

Clients who wish to update or change their key can send a new key-addition request, signing it with their previous key to prove their identity. This overwrites the previous key's value, and the server tracks this change in the leaf's change log. For this update to occur, the server sends a proof of the addition to the log, which the client has a grace window of two weeks to confirm. Only after this confirmation is the

change registered.

To ensure consistency, users can act as monitors that use a gossip protocol [8] to share the epoch's reference string with each other. The dispersion of log views among users gives them greater confidence that everyone shares the same view, because a fake view will be detected and reported.

As it was the first streamlined proposal for key transparency, CONIKS was the foundational work for newer proposals, which have taken its ideas and built upon them, either by increasing protocol efficiency, allowing multiple keys for a single user with multiple devices, or employing different consistency protocols.

2.1.2 Parakeet

Parakeet [9] is a production-grade implementation of a key transparency system, practical and aimed at being able to handle a real-world scale user base. It utilizes a username-to-key mapping in order to store users' keys and allows for their lookup by users who want to be able to communicate with others. It introduces an efficient VKD implementation along with a mechanism used to mitigate its ever-growing size: Compaction.

Parakeet's main focus is being efficient in large-scale scenarios, presenting an extremely scalable VKD, capable of storing billions of user keys. Its advancements in the method of storing keys make it much more efficient than proposals that were made before it.

Each user can perform several requests, either to add a new key, replace their key with a new one, or revoke a previously claimed key, in case they were compromised or wish to change keys. Unlike other VKD proposals, Parakeet creates an extension proof by guaranteeing that the new log is an extension of the previous ones in terms of the username-key mappings, instead of guaranteeing absolutely no changes or deletions from the log. This is due to the fact that a key replacement event would occupy the same position in the history of the log as the key addition, since they share the same username, which is taken advantage of by compaction in order to reduce memory.

Parakeet utilizes an ordered Zero-Knowledge Set [10], which allows users to request membership proofs for keys, guaranteeing their presence or absence from the corresponding data structure. Its efficiency comes from the fact that Parakeet uses a Merkle Patricia tree [11], a tree structure designed to efficiently store and look up keys. When a user requests the proof of a key, if the event is a rotation, they access the previous key in order to verify the signature of the rotation request, as long as

the previous key is in the retention window.

In comparison to previous VKD solutions, Parakeet does not store all previous states of the log, instead storing only the mutable tree which is published at every round. Previous states are assumed to be correct, as the protocol assumes that they were verified and found to be correct when published, as only then would the protocol continue and publish a new state.

Parakeet also introduces the concept of compaction, a way to reduce the memory size of the tree considerably. Every time a user rotates a key, the previous key's entry no longer affects the root node's hash, as there is no direct connection from it anymore, and so the protocol marks it for deletion. This happens due to the fact that different key actions for the same user will share the same index, as this is created using a Verifiable Random Function, the user's username, and the server's secret key, meaning two operations for the same username will occupy the same index in the tree.

After a certain number of server rounds, titled the retention window, the server deletes all marked keys, massively reducing the storage capacity used by the protocol. Since the deleted actions were no longer tied to the root node, these deletions do not affect the overall extension property of the protocol, and invalid modifications can still be detected when verifying each entry.

As a consistency method, Parakeet uses predetermined witnesses, who sign the new view of the log whenever it is published. In order for users to be confident they are consistent with the remaining users, they need enough witnesses, $2f + 1$, with f being a predetermined fault tolerance parameter, to sign a view that matches their own in order to ensure probable consistency. There are a total of $3f + 1$ witnesses, which implies that if there are $2f + 1$ witnesses for a single view, there cannot be enough unique endorsements for another view.

2.1.3 ELEKTRA

ELEKTRA [3] is a newly proposed solution for key transparency, introducing the concept of a Multi-Device Verifiable Key Directory (MVKDs), further bolstering privacy and usability guarantees offered by a standard Verifiable Key Directory.

The goal of this proposal is the introduction of a verifiable key directory that is able to handle users who have multiple trusted devices, such as a personal computer and their phone at the same time, a scenario that is very common for its target applications of messaging platforms.

Each user has a unique identifier in the form of their username, which serves as a way to keep track of their different devices. Users can manually add any number of keys, corresponding to each of their platforms, and revoke previous ones. The addition of new keys requires identification using a previously stored key, with the exception of the addition of a new user to the directory.

This approach guarantees that users can store multiple keys simultaneously, while also preventing attackers from trying to create a fake key in the name of an existing user. Entries are verifiable by users, allowing each participant to manually confirm that the additions to their section of the log match what they expect.

MVKDs work by storing users' keys in key chains, which are logical sequences of statements that define what operations are meant to be performed with any given key. They allow for the addition of new starting keys, the addition of keys for new devices, the revocation of any of the user's keys, which is useful in the case of a compromised device, and an additional command that binds extra information to any one given key. A user's total key history can be acquired by reading the entire key chain in order, obtaining a full record of what keys were added and revoked. This allows recipients to confirm that the key used to sign any given message that is claimed to come from a specific user does in fact belong to that user and has not been revoked.

In addition, ELEKTRA stores users' key chains using a Rotatable Zero Knowledge Set [12], a unique method of storing information that allows additional privacy. It works by having the server keep a private Verifiable Random Function, which it uses on each new addition to the log in order to determine the corresponding index of a Merkle tree where the new input should be stored. This allows for a replicable lookup system that stores users' keys in a verifiable pattern, while also preventing users' information from being leaked simply by reading the tree, as the user's identity is not shown directly in each node.

In the case the server is compromised and the Verifiable Random Function's secret key is leaked, the protocol also offers a procedure known as PCSUpdate, which stands for Post-Compromise Security update, which lets the server update its VRF by changing its key and offers a zero-knowledge proof that each new index matches the previous hidden labels, causing an update in the tree.

ELEKTRA presents three crucial properties: *Completeness*, *Extraction-Based Soundness*, and *Privacy*. *Completeness* states that if the server and users are all responding correctly, every proof can be verified and there is only a negligible chance of random failure. *Extraction-Based Soundness* guarantees that auditors can detect any inconsistent view states from the same epoch. Finally, *Privacy* ensures that the minimum number of information possible is revealed at any time, meaning proofs do not reveal usernames or the targets of queries.

As a consistency mechanism, ELEKTRA suggests using an external group of auditors. Users can poll the auditors for confirmation that the extensions of the log are correct and that all users are downloading the same information. Log states are checked periodically in different epochs, with auditors confirming logs are only extensions of previous epochs without any modifications or removals.

2.1.4 OPTIKS

OPTIKS [4] is another proposed solution for the problem of Key Transparency, focusing on the efficiency and scalability problems of having a log with billions of additions, while matching previous solutions' security guarantees.

Similarly to other approaches, OPTIKS uses a single mutable Merkle tree used to store user inputs and updates, but a big difference is in the way the index needed to store the key is calculated. Instead of simply checking the output of a Verifiable Random Function with the username as input, OPTIKS concatenates both the username and the current version of the key to determine the index.

This approach allows for the creation of a non-membership proof, where a user can verify whether there is an update to a given key by checking if such an update exists in the tree. When Alice asks for Bob's key, the server sends both the proof of membership for the current key, as well as a proof of non-membership for a possible updated key. This assures Alice that she is receiving the newest stored key from Bob.

The non-membership proof allows for key additions and substitution without changing the history of the log, allowing the Correctness property to hold while still letting users change keys. Whenever a new state is to be published, the server stores the sequence of nodes needed for verification for Correctness, and sends them along with the new root and a signature as proof of Correctness.

Users rotate keys, the same as in CONIKS, using a previous key to sign a new addition, which is then stored in an update log, only being added to the tree after the user sends confirmation, having an adjustable grace period to send this confirmation. In order to increase throughput, OPTIKS separates the server responsible for updating keys from query servers, meant to allow for faster lookup, and lookups are still available if the main commit server crashes or some query servers stop responding.

OPTIKS presents the same privacy considerations as the other solutions mentioned, as observers without knowledge of the secret Verifiable Random Function key cannot know where each username is stored in each index. However, unlike ELEKTRA, it does not present a method to replace the VRF key in case it is compromised,

presenting a possible privacy risk if the key is leaked.

For consistency, OPTIKS suggests using an external monitor, who watches the distributed state hashes and reports if it finds a mismatch of roots for the same epoch. In addition, it also permits an "auditor-free" mode, where clients can self-check consistency by gossiping their state log with each other. Information spreads this way, leading to the discovery of possible inconsistencies.

2.2 Consistency Mechanisms

Consistency is one of the key focal points of Key Transparency, as it is required for users to communicate confidently with each other. This section discusses the currently used methods to achieve consistency, their flaws, and differences between them. Furthermore, it also compares them with the proposed solution of Consistency-or-Die, as can be seen in Table 2.2. This table compares each method in terms of its efficiency for large user bases, the existence of only in bound communication, where they use an endorser system, their untargetability and whetter there are any additional monetary costs associated with the given solution.

Strategy	**Efficient**	**In Bound**	**Endorser**	**Untargetable**	**Add. Costs**
Gossip	✓	✗	✗	✗	✗
Consensus	✗	✓	✗	✓	✓
Fixed-Committee	✓	✗	✓	✗	✗
CoD	✓	✓	✓	✓	✗

Table 2.2: Comparison of Strategies for Key Consistency

2.2.1 Gossip

Gossip protocols [8], [13] are used in order to ensure consistency, which relies on direct communication between users of a specific protocol.

The idea behind gossip protocols comes from real world communication in a word-of-mouth fashion. Users receive a central piece of information from a given source, and they distribute their respective information directly to other users. Subsequently, participants compare the information they received from the central party with the data obtained through other participants, and check for inconsistencies or discrepancies. If the verification is correct, the user can now publish that both they and another user agreed on the state received, increasing the confidence for future users that this is the correct view.

Users who detect a mismatch can then claim that either another user lied about what information they received, or that the central server gave wrong information

to one or more participants, indicating that it has been compromised.

While gossip has extremely enticing properties such as a strength in numbers that comes from using the entire user base as verifiers, and the fact it does not need to put trust in one central party, instead distributing it through the population, it still has flaws that hamper its real world utility.

First, for each communication made with another user, an out-of-band channel is required, due to the fact that the users would need each other's public keys to communicate, the very thing they are trying to verify the consistency of. Due to the consistency guarantee being larger with an increased number of direct communications, this raises the question on how users could uphold so many out-of-band channels. An isolated user who cannot communicate with many others is also a prime choice for targeted attacks, as malicious users working with a compromised server can distribute fake states that match the server's fake log, creating an undetectable attack due to the lack of honest users.

Furthermore, as information takes time to travel from user to user, the gossiping protocol can take time to detect a split-view attack, as a user may only receive enough verifications from other users to detect an inconsistency much later after receiving the VKD's state.

2.2.2 Consensus

Consensus is the consistency mechanism enforced by blockchain and has been proposed as a possible solution to the problem of consistency in key transparency by combining blockchain functionality with a verifiable key directory [14].

EthIKS [15] is a proposal that aims to combine a CONIKS Verifiable Key Directory together with the consistency that can be obtained from the Ethereum blockchain. The goal is to associate every published state of the VKD with a publicly available cryptographic block.

Each state change published by the VKD gets registered in an Ethereum block using the smart contract system, in which a transaction or change is stored in a verifiable way inside a cryptographic block. This feature from Ethereum allows for multiple applications, protocols, or business deals to be recorded and confirmed, with ETHiks using this to store confirmation of an updated log and its respective history.

In order to do this, the system takes advantage of Ethereum's proof-of-stake architecture, which forces attackers who try to create splits in the block history to have extremely large monetary backing, which they risk losing if a split is detected and

traced back to them, ensuring continuity.

EthIKS ensures guaranteed continuity and a single output for each state update, as only the first publication will be added as a correct continuation of the chain. Users can then request proof of their keys and stored keys of other users in order to ensure no unwanted alterations were made.

However, this solution brings with it the normally associated costs of using blockchain, requiring users to not only trust the legitimacy of full nodes but also to account for the costs associated with transaction fees for registration of a contract.

2.2.3 Fixed Committee

Finally, the concept of an external group of witnesses has also been suggested [9]. These are trusted participants in the protocol who download updated views from the service provider. The log maintainer sends a commitment of the current log to the predetermined set of trusted auditors, who then independently sign the registry. Users then poll the committee of trusted witnesses in order to obtain their corresponding endorsements for the view each one received. If enough endorsements match a specific view of the log, users can be confident that the endorsed view is correct.

While this method is more scalable than the previous two, due to the fact that it avoids blockchain's heavy computational load and gossip's out-of-bound channels from user to user, there are some flaws in this strategy. The committee size needs to be small, as finding a large number of trusted auditors is extremely difficult, as all users need to agree to trust the auditors. Furthermore, an increase in the number of auditors also leads to an increase in the number of signatures required by users, and a corresponding increase of peer-to-peer communications between users and witnesses.

At the same time, a small committee that is known to all users means an attacker with significant computational power can attempt to target these known entities, allowing the attackers to utilize more resources against a small number of targets.

Despite its flaws, witness auditing is currently being used in real deployments of key transparency protocols as the current best solution to the problem of achieving consistency, present in practical solutions such as Parakeet and SEEMless.

2.2.4 CoD

CoD presents an alternative efficient method of ensuring consistency, that utilizes an ever-changing group of endorsers, selected randomly from all users of the protocol, ensuring untargetability. It ensures consistency through the computational difficulty

and statistical unlikeliness required for the protocol to failed. A deeper analysis of the protocol and it's behavior is presented in the following section of the thesis.

3

Background

This chapter gives an overview of what a split-view attack is, the main type of vulnerability CoD is intended to prevent. Following this, it reviews the main cryptographic building blocks that CoD utilizes, before explaining the protocol in detail.

3.1 Split-View Attack

A Split-View Attack occurs when two participants try to access a single public resource, and both receive different information, with each one believing they received the same information. This kind of attack exploits the lack of consistency in a communication protocol and works as a starting point for other threats, such as man-in-the-middle attacks in which a third party can decrypt and falsify messages between two parties. An example of a split-view attack against a key log that offers no consistency guarantees can be seen in Figure 3.1.

When Alice and Bob both request access to the current log, the malicious Channel Maintainer (CM) sends two distinct views of the log. Neither Alice nor Bob know that the other's view differs from their own, and they are unable to confirm this without talking to each other, leading to a state where both believe they are sharing the same information. If Alice tries to communicate with Bob, she will encrypt her messages using what she believes to be Bob's public key, which is actually a key chosen by the CM, responsible for storing, updating, and publishing the log. The CM can then intercept the message and change it, before encrypting it again with Bob's public key, and sending it to him. Bob will then decrypt his message correctly and will not detect the change.

This type of attack is possible because there is no Consistency mechanism in place that allows users to ensure the state of the log they received is the same as what other users received, leading to an undetected attack. This is what CoD and other procedures designed to ensure Consistency try to prevent, assuring that all users share the same information.

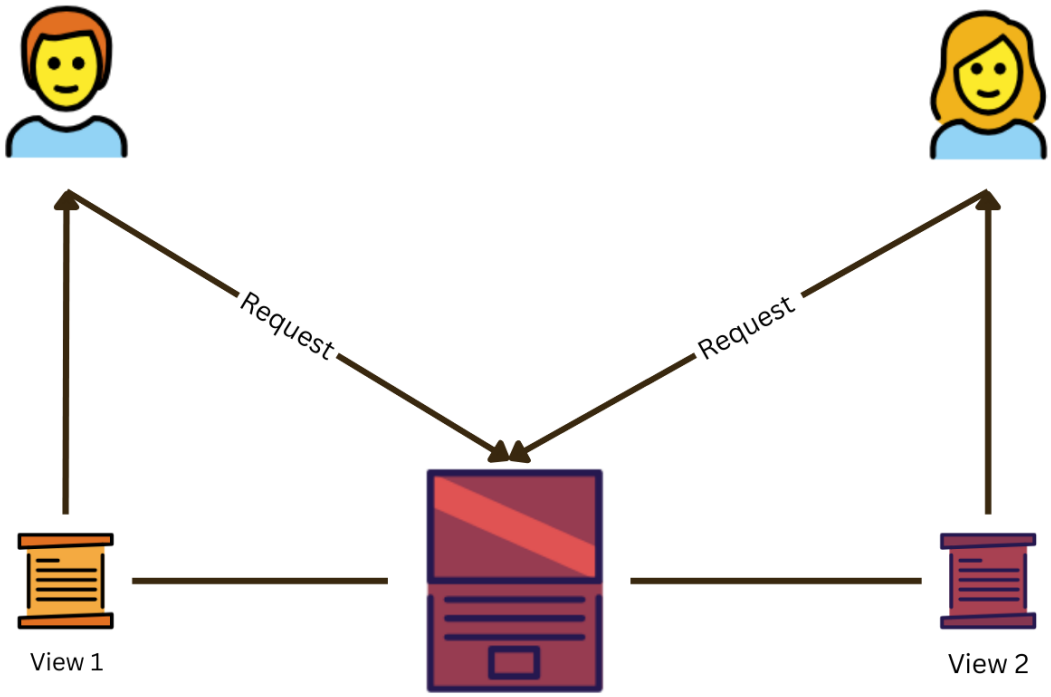


Figure 3.1: Example of a split-view Attack

3.2 Cryptographic primitives

Before diving into the specifics of how CoD works, it is important to understand the underlying primitives that it takes advantage of in order to create a secure protocol that is robust against various types of attacks and can achieve reliable and verifiable consistency for all legitimate parties involved.

3.2.1 Elliptic Curve Cryptography

Elliptic Curve Cryptography is very commonly used as a fundamental basis for generation of public and private key pairs. Elliptic curves are functions defined by an equation of the type $y^2 = x^3 + ax + b$, whose properties enable operations defined entirely within their domain such as point addition and multiplication, which are fundamental for the construction of cryptographic building blocks. Curves used in cryptography also possess a large cyclic subgroup, meaning that the curve has at least one point, called the generator, for which all other points belonging to this group are multiples of, which is the basis for the construction of cryptographic schemes.

The theoretical foundation for using these curves in cryptography lies in the hardness of the Elliptic Curve Discrete Logarithm Problem, which states that given a generator of the curve, and a multiple of the generator by a corresponding scalar, it is computationally infeasible to calculate the scalar value. Due to this property,

a scheme for generating key pairs can be constructed, by first setting the generator point, g, to be used for a corresponding curve, choosing a scalar value which will be the private key, $sk = x$, and "multiplying" the generator the corresponding number of times to generate the multiple which is used as the public key, $pk = g^x$.

Key pair generation using elliptic curve cryptography is extremely efficient, as it has small key sizes and fast computations, while not sacrificing security considerations, and so has become very common. It does, however, suffer from the fact that it is not post-quantum secure, due to being broken by Shor's algorithm [16], which is able to efficiently solve the Elliptic Curve Discrete Logarithm Problem by discovering the secret key in polynomial time, given the generator and public key. The increased desire for post-quantum secure algorithms in order to keep security guarantees in a possible era of efficient quantum computers has lead to other alternatives being researched, but currently elliptic curve cryptography still has widespread usage.

3.2.2 Verifiable Random Function

Verifiable Random Functions (VRFs) [6] are cryptographic functions that utilize a given input, typically named the seed, and a secret key to generate a pseudorandom value (output) and a proof of this computation. Any user with the corresponding public key, seed, claimed output and the generated proof can then run a verification process that confirms whether the given value matches the result of the VRF for the given input and public key. Formally, a VRF requires three components:

- **KeyGen**$(\lambda) \rightarrow (sk, pk)$ This function receives the target bit security level of the keys and generates a key pair consisting of a secret key sk and its corresponding public key pk.

- **Prove**(sk, s) $\rightarrow (v, \pi)$ This function receives the previously created secret key sk and the seed of randomness s, and outputs the pseudorandom value v and the proof of computation π.

- **Verify**(pk, s, v, π) $\rightarrow$ True/False This function receives a public key pk, seed s, value v, and proof π, and returns True if π verifies that v was created using **Prove**(sk, s), with sk being the secret key that matches the given public key, else it returns False.

There are three properties VRFs need to exhibit to be considered secure and be usable in cryptographic schemes: *Uniqueness*, *Provability*, and *Pseudorandomness* [17]. The first property states that for any given s and sk, there is only one unique proof π for which **Verify**(pk, s, v, π) returns true, which guarantees determinism from the VRF. The second states that any proof that is correctly generated from the **Prove** algorithm should always pass verification, i.e, the output of the **Verify** function must always be **True**. Finally, *Pseudorandomness* states that any computationally

bound attacker should not be able to distinguish a generated output from true randomness with better than negligible odds, which depend on the target bit security λ.

It is also crucial that users who successfully run a verification of the generated number do not obtain any information on the value of the secret key. For this, its important that the VRF proofs are Non-interactive Zero Knowledge (NIZK) proofs [18], meaning they allow the prover to show the generation was done correctly from the secret key, without leaking information that might allow for its reconstruction.

VRF's ability to allow for randomized values that can be proven is extremely useful for a variety of different protocols with a variety of applications such as leader election, lottery systems and committee selection. However, it is crucial that the seed used by users be the same, in order for values to be correctly generated and subsequently verified. VRFs play a fundamental role in CoD as the key mechanism behind the election of the endorsement committee, giving each participant an equal chance of being selected and a way to enable them to prove this selection.

3.2.3 Verifiable Key Directory

Verifiable Key Directories (VKDs) are centralized systems that store the public keys of users in a given system, while also offering an interface to allow for user's to lookup and verify the existence of a given key in the log. The maintainer of the log keeps a history of all keys added in pairs, associating a key with a corresponding identifier and allows the creation of inclusion proofs that guarantee that any given key is in fact present in the log. It also has the important property of being append-only, meaning that any change that happens to the log must be an extension, which only happens with the addition of a new key. Previously logged keys cannot be changed, and the VKD also offers a method of verifying this property. Broadly, a VKD can be defined by the following functions:

- **Insert**($\mathsf{st_i}$, k, id) $\rightarrow$ $\mathsf{st_{i+1}}$ This function receives the current state $\mathsf{st_i}$, a key k, and an identifier id, and updates the VKD by inserting a new entry composed of the identifier and key. It outputs an updated state st_{i+1}.

- **ProveMembership**(st_i, k) $\rightarrow$ π_{mem} This function receives a VKD state st_i and a key k, and returns a proof π_{mem} that k belongs for state number i. If the specified key is not present in the log, it returns False.

- **ProveExtension**(st_i, st_j) $\rightarrow$ π_{ext} This function receives two VKD states st_i and st_j for $i < j$, and outputs a proof π_{ext} that shows that st_j is a valid extension of st_i. If st_j is not an extension of st_i, it returns False instead.

- **VerifyMembership**(st_i, k, id, π_{mem}) $\rightarrow$ $\mathsf{True}/\mathsf{False}$ This function receives a

state st_i, a key k, an identifier id, and a membership proof π_{mem}, and returns True if the proof verifies that k is mapped to id in the VKD corresponding to st_i. Otherwise, returns False.

- **VerifyExtension**$(\mathsf{st}_i, \mathsf{st}_j, \pi_{\mathsf{ext}}) \rightarrow \mathsf{True}/\mathsf{False}$ This function receives two states st_i and st_j and a correctness proof π_{ext}, and returns True if st_j is a valid append-only extension of st_i. Otherwise, returns False.

A solid option for implementing VKDs are Merkle trees, a data structure in which all new additions are nodes who are added at the bottom of a tree as a new leaf. Leaves are paired and hashed together, creating a node for a higher layer whose value depends on its two children nodes. This process is repeated sequentially through all nodes in all layers, creating the root node, which depends on the value of all subsequent nodes.

Merkle trees allow for efficient inclusion proof generation by providing a path from the leaf node to the root, requiring only the hashes necessary to reach it, scaling logarithmically with the number of nodes. They are also very efficient for proving the append-only property, as the addition of new leaves does not affect the pre-existing structure allowing for quick verification by checking if the new log is merely an extension of the previous one. Combining the ease of verification of key inclusion and log correctness associated with VKDs with a consistency protocol is the foundation of key transparency, and is required for users to be able to trust the state of the directory they receive.

3.2.4 Key Evolving Signature

Signature schemes are commonly used to allow two parties to communicate with each other while being able to verify that messages received were sent by the expected correspondent. To do this, users sign a message with their secret key and create a digest, which can then be reversed using the corresponding public key to obtain the message that was signed. This then provides a guarantee that the received message came from the correct user, as no other user's secret key would be able to reverse the digest and obtain the original message.

However, this kind of signing scheme suffers because if a user's secret key is leaked in any way, then attackers can falsify signatures for previous and future messages. If this simple signature scheme were used in Consistency-or-Die, an attacker that is adaptive and rushing (meaning it can actively target users after they have revealed themselves as endorsers for an epoch) could wait to see which users claim to be endorsers, corrupt them, and sign any given view using their secret key.

To prevent this, CoD instead uses a Key Evolving Signature (KES) for signing. In this scheme, whenever a user signs a message with the secret key, it is evolved in

a deterministic and irreversible manner, up to a limited number of evolutions. The signed message then contains an index representing which iteration of the secret key was used to sign it, which users can then compare with the expected index to confirm the secret key that was used has been evolved the correct number of times. Formally, a KES can be defined by having the following four algorithms:

- **KeyGen**(λ, Max) $\rightarrow (sk, pk)$ This function receives the target bit security level λ and generates an evolving secret key sk, which can be updated up to Max times, along with its corresponding public key pk.

- **KeyUpdate**(sk, i) $\rightarrow sk_{i+1}$ This function receives a secret key sk and its current index i, and returns the secret key at the following index, $\mathsf{sk}_{\mathsf{i}+1}$, or fails if the key is updated more than Max times or if it is updated using an incorrect index.

- **Sign**(sk, i, msg) $\rightarrow \sigma$ This function receives a secret key sk, its current index i, and the message msg to be signed, and returns its signature σ, or fails if the index i does not match the given secret key.

- **Verify**(pk, i, msg, σ) $\rightarrow$ True/False This function receives a public key pk, an index i, the message msg that was signed, and its signature σ, and returns **True** if the signature is valid for pk at index i, or **False** otherwise.

In CoD, the presence of a Key Evolving Scheme prohibits adaptive and rushing attackers from simply waiting to see which users are endorsers and corrupting them, since when a user reveals itself as an endorser, they already signed their current view of the epoch, which implies that their secret key has evolved. If an attacker then tried to sign a message with the updated key, when a user verifies the signature, its index would not match the index the user is expecting, thus leading to a failed verification. However, this could imply the need for every user to keep track of the expected index for every other user, which could imply that users might be restricted by memory. The solution to this, is to force all users to also sign an empty message, if they are not selected as endorsers. This then causes all users to be in the exact same index, meaning they do not store individual indexes for all other users. As a drawback, this approach causes users to reset their keys at a predictable interval, and it is possible to calculate when a specific user would create a new key pair after exhausting all evolutions of the previous one, which could lead to attackers replacing a user's expired key before they can do it themselves.

3.3 Consistency-or-Die protocol

Now I will go over the Consistency-or-Die protocol, as it is described in its original paper [5].

Consistency-or-Die is a protocol for reliable and secure verifications on the consistency of a given key log, stored by a third party, called the Channel Maintainer (CM), who is untrusted by users. Users have no guarantee of whether the CM is malicious but can consistently verify whether the view of the log they receive for any given epoch matches what other consistent users see. To do this, some users are randomly chosen as endorsers and publish endorsements of their views to the CM, who then distributes them to all users. Users can then verify the legitimacy of the endorsements, and if a user receives at least the minimum number of endorsements required to feel confident their view is legitimate, referred to as a quorum of endorsements, they will continue to participate in the protocol (Consistency). If they do not, them user aborts participation (Die), since they cannot be certain there was not a split-view attack or whether or not the CM is functioning properly.

3.3.1 Participants

The protocol recognizes two types of participants: the Channel Maintainer, responsible for keeping track of the current log, as well as responding to requests from users, and Users, who occasionally upload data to the CM and request information about the log's current state and its endorsements, in order to verify if the view they received is consistent with other participants' view. Users can be of one of three types:

- **Honest**, meaning they follow the protocol exactly as defined without trying to take advantage of it, which typically make up the largest portion of users;
- **Inactive**, meaning that either some or all queries or updates to the server are being suppressed or the user simply is not responsive (be it due to a legitimate or malicious motive);
- **Malicious**, in the sense they are trying to manipulate the algorithm in order to try to launch a successful split-view attack

Malicious users are allowed to exhibit any arbitrary behavior and collaborate with other malicious users and/or a corrupt CM to try to break the consistency property of the protocol. Their final goal is to achieve a quorum of endorsements for at least two distinct views, and allowing the successful launch of a split-view attack, breaking consistency. Due to their ability to collaborate with the CM in the case of it also being compromised, they are allowed to sign multiple different views, leading to their ability to, in the case of being selected as endorsers, send multiple endorsements in a single epoch.

Let H, I, and M be the total number of honest, inactive, and malicious users in the system, respectively, with N being the total number of users, $N = M + H + I$. An

analysis of the maximum valus for M that still allows for consistency to be efficiently confirmed is presented further on in the thesis.

Users possess two different key pairs, VRF and KES, each with a different purpose. The VRF key pair, $kp_{VRF} = \{sk_{VRF}, pk_{VRF}\}$, is used in the generation and verification of a random value. During each epoch of the protocol, all users run a VRF, which is used to see whether that member was chosen as an endorser for the current epoch, by comparing the result to a threshold. This threshold intends to limit the number of users selected to endorse, in order to ensure both statistical correctness, but also to reduce the work participants are forced to do in each epoch. In the interest of computational and time efficiency the goal is to restrict the number of chosen endorsers to a small percentage of the total population, while still guaranteeing statistical certainty as to the security of the results.

The KES key pair $kp_{KES} = \{sk_{KES}, pk_{KES}\}$ is used by endorsers to sign the view they received, and by all users to be able to verify these signatures. After each signature the secret key is evolved, changing in order to ensure forward security. Whenever a secret key has been evolved to its limit, it becomes exhausted and can no longer be used while safely ensuring forward security. This then implies the need to generate a new KES key pair, and add the new public key to the log.

Each key pairs' public key needs to be added to the log whenever a new one is generated, in order to ensure all other users can have access to them to be able to run the previously mentioned verification algorithms. These constant additions are what define the new view of the log for each epoch, and what users want to be able to agree on after achieving consistency.

3.3.2 Trusted Setup

Before running the first epoch of a new instance of a Consistency-or-Die protocol, initial users and the CM need to agree on the starting parameters in order to guarantee everyone starts with the same information. This procedure is called the Trusted Setup, and users are required to trust the CM for this step of the protocol. Here, all initial users' keys are added to the log, and the corresponding parameters for the initial epoch are generated, which serves as a trusted starting point for all users to begin participating in the protocol.

Each epoch of CoD has some parameters that are vital to the correct execution of the protocol. In detail, each epoch's parameters are stored in $epar_e$, the parameters used in epoch e, and includes the total number of users, the bit security that is being considered, the seed that is created (or used) in the following epoch (see Section 5.3 for further analysis), the current digest at the end of the first phase of the epoch, the threshold that corresponds to the likelihood of any individual user being chosen as an endorser, the quorum size, and the ratios of honest/malicious/inactive users

i.e. $epar_e = (\lambda_e, N_e, \text{seed}_e, \text{dig}_e, T_e, k_e, fH, fM, fI)$. In the trusted setup, all initial users' keys are added to the log, and $epar_0$ is created. Users then request access to these starting parameters and are then ready to start participating in the protocol.

3.3.3 Phases

CoD functions in a series of epochs, each divided into 4 non-overlapping phases. At the end of the last phase, users either abort (meaning they cannot be confident in the consistency of their view) or continue to the next epoch if they have strong enough evidence that the system is still valid. Epochs are incremental and consist of the following phases:

- **Collection**, in which users can send new keys to the CM to be added to the log. This is typically done by either new users who are joining the protocol, or old users who wish to update their keys. At the end of this phase, the CM constructs the view for this epoch, View_e, which includes a log of all keys, the proof that this log is an append-only extension of the previous logs, and a list of the parameters necessary for the next epoch, $epar_e$, which include the total number of users N, along with the fraction of the number of honest, malicious, and inactive users, fH, fM, fI, the quorum size for this epoch, k, a threshold that determines to the chance a user is selected as an endorser, T, and the seed to be used in the next epoch, $seed_e$.

- **Distribution**, in which users send a request to the CM, asking for the view of the current epoch, along with the proof it is an append-only extension of the previous epoch's log. Users then check the legitimacy of the view by confirming the proof and verifying the seed was correctly generated. If the verifications fail, the user aborts.

- **Certification**, where users run the VRF with input $(sk_{\text{VRF}}, seed_{e-1})$ and compare the obtained result with the threshold obtained in the parameters of the previous setup T. If the value passes the verification, the user is chosen as an endorser, creates an endorsement and sends it along with the proof of VRF generation to the CM, who then stores it in a list of endorsements.

- **Verification**, where users receive the list of endorsements and proofs, verify that each VRF was calculated correctly and passes the threshold previously given by the CM, as well as confirm the signature is correct. Then, they compare the endorsed view with the one they received in the Download phase. If all proofs are correct, and the views are the same, the user increases the number of matching valid endorsements by one. If the user receives enough endorsements to reach the quorum, it can be confident its view matches the other consistent users and remains active for the next epoch of the protocol.

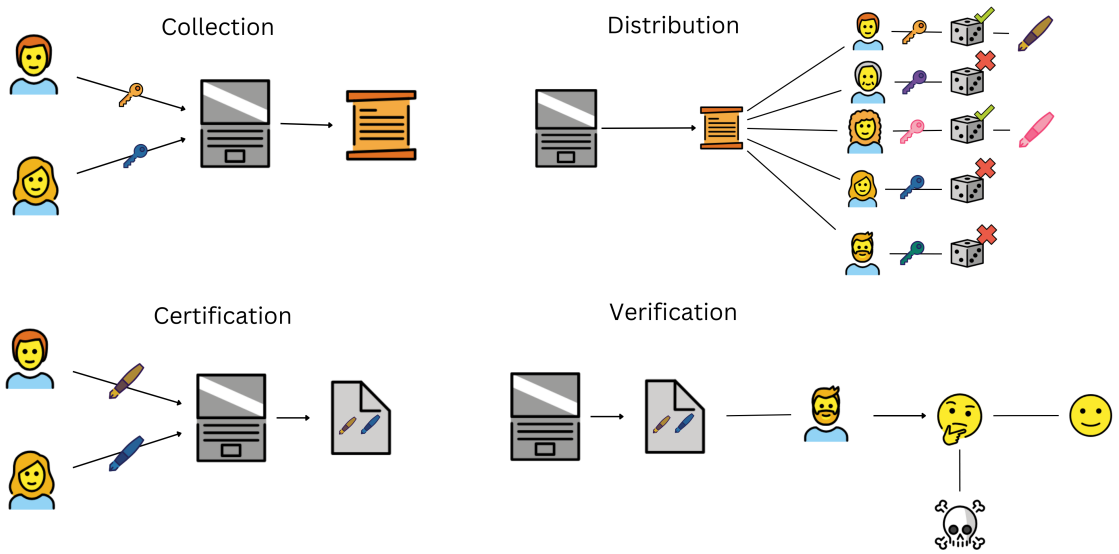


Figure 3.2: Diagram of the phases of CoD

Otherwise, it ceases communication and aborts.

A layout of how these phases connect with each other can be seen in Figure 3.2.

Consistency-or-Die hinges its security on the fact that it has an immense number of users, but only a few are selected to participate as endorsers,and these are only revealed after having already endorsed. This prevents attackers from targeting users who will be endorsers, as they have no way of knowing who specifically will be in the endorsing group, while also preventing the generation of enough fake endorsements to create quorums for two distinct views, as these endorsements would need to pass all verifications and there would need to be a considerable fraction of malicious users who are selected in order to accomplish this.

Thus, any user who completes an epoch of the protocol successfully, can be confident they are consistent with all other users who also participated and correctly finished the epoch, and can repeat this process for future ones.

It is important to note that CoD focuses on allowing users to detect inconsistencies, but does not present a mechanism to tolerate them. If any proof fails or not enough endorsements are received, users abort communication, meaning a corrupt CM can easily abort communication from all users by, for example, simply sending an empty endorsement list in the **Verification** phase. CoD focuses on preventing split-view attacks, which have more pressing consequences than the denial-of-service attack possible by crashing users as mentioned.

4

Methods

This chapter will focus on implementation details and specifics for the implementation aspect of the thesis, together with an explanation of how the results were measured and collected. Finally, it also presents the analysis required for the calculation of the maximum threshold of malicious users compatible with CoD's security features. The implementation is open source [19].

4.1 Design Choices

4.1.1 Programming Language

The finalized implementation was made entirely in C, taking advantage of the extreme efficiency in both execution time and memory manipulation that this language has in comparison with other possible choices, such as Python. This was done to better simulate a real-world deployment of the protocol, as it would need to be as efficient as possible to handle the number of users that are expected to participate.

Furthermore, C already has available some key libraries for this type of project, such as the OpenSSL library, which contains multiple cryptographic foundations, such as hashing, which are vital for the implementation of a consistency protocol. While other languages such as Rust and Go are also possible choices for a realization of this protocol, the availability of these libraries, along with other available implementations of fundamentals such as KES and VRFs, makes C a solid choice for a production-level implementation of CoD.

4.1.2 Libraries and External Code

Simulating user and server behavior for this type of project requires various considerations in both the specifications of the protocol itself and the communication between these two parties. For these purposes, multiple libraries were used that are tried and tested as to their efficiency and reliability in their respective functionalities. For cryptographic foundations such as hashing and elliptic curve mathematics (which are crucial to implementations of private and public key pairs), the OpenSSL library was chosen. This library is foundational, as it not only implements open-sourced versions of important transportation layer protocols (such as its namesake

SSL) but also implements secure hashing algorithms.

For VRF, a modified version of the code originally utilized in the CoD paper for random number generation was used [20], updated to the standards of OpenSSL 3.0. This code utilizes elliptic cryptography to generate private and public key pairs, which are used along with a randomness seed to generate a random value. This generation also includes the creation of a NIZK proof, which users can use, along with a public key and seed, to confirm whether the output of the VRF was correct.

The KES chosen was XMSS [7], which allows users to create a key pair specifically used for signature purposes. While the public key generated by XMSS is never changed, every time a user signs a message, the secret key is automatically updated, creating forward security. XMSS achieves this by generating multiple ephemeral one-time keys upon the creation of the key pair, with each one being utilized to sign a single message. The public XMSS code allows for multiple different possible parameters for the key pair generation, each bringing different tradeoffs between security and efficiency, be it in terms of computations required or memory consumption.

Forward Security is crucial in order to prevent against adaptive attackers, as it protects previously signed messages and prevents the forging of messages that could be claimed to have been created earlier. Since the key evolves after every signature, messages signed before a user was compromised cannot be manipulated, and so an adaptive attacker who can choose to corrupt a target user still would not be able to fake previous entries.

Furthermore, for the communication between users and CM, there needs to be a way to serialize information in order to safely transmit and reconstruct data without losing any bytes, which could corrupt messages. For this, ProtoBuf was used [21]. Protobuf is an open-source mechanism that automatically creates the code necessary for the serialization of data structures. Its used to automatically serialize and unserialize the different message structures that are used in our protocol, such as the view generated by the server, personalized user endorsements along with their necessary proofs, and the corresponding list of endorsements, facilitating communication between the server and users.

Together, these libraries and code packages lay the foundation necessary to create the implementation of CoD, with the remaining core specifications of the code being built upon these publicly available primitives.

4.1.3 Access to Keys

In CoD, users need to have access to all other participants' public keys. This is not only the overall goal of a key log, but also the focus point of the entire protocol.

However, it is extremely costly for users to download the entire log every epoch update, as with billions of users, the log will have to store billions of keys. The simple solution to this problem is for users to check the consistency of the log using the root hash of the log instead of its entirety, as this string is generated depending on each of the stored keys in an irreversible and unique way, which means that, for the purpose of checking that the log is a mere extension, the root hash suffices.

However, this still leaves the question of how users should have access to the keys in order to check the validity of proofs that are generated by endorsers. A straightforward solution to this would be for the endorsement to also provide the id of the endorser, with users then being able to ask the server for proofs of inclusion of that id's keys. This would mean users would have tangible proof that the key they should use to validate a proof exists in the log and belongs to the correct endorser.

However, the current implementation does not allow for this approach, due to using a hash-chain structure to store the keys, which, while also creating a root hash that allows for consistency verification and append-only guarantees, does not allow for inclusion proofs. As a solution, the server sends the public keys users need for verification along with the endorsements when a request to run the Verification phase occurs. While this is not realistic, as we do not want users to have to trust any of the CM's input, this solution works as a simplification of the protocol, as long as real implementations keep in mind the necessity for a better system to store keys that allow for the generation of inclusion proofs.

Deployed implementations should instead strive to use a Merkle Tree approach for their VKD instead of a Hash-Chain, in order to allow for inclusion proofs and simple lookup. The original plan for the implementation was to implement a Merkle Tree, but in order to stay on schedule for the remaining work, a Hash-Chain was implemented instead, where each entry is hashed with the previous ones in order to create a root hash capable of correctness proofs, but not of inclusion proofs.

4.1.4 Security Level

Utilizing the aforementioned security primitives brings with it many choices that affect the security level the application is targeting, while also impacting the computational power necessary, both for the users and for the server.

XMSS's one-time signature keys utilize Winternitz One-Time Signature Scheme (WOTS) [22], which hashes a message a fixed number of times, determined by a value called the Winternitz parameter. This parameter determines the balance between signature size and signature and validation speed, with a small value leading to reduced signature size but more hash operations required to reach the final signature, and a large value leading to faster operations, but a larger signature size.

The large signature size previously mentioned is a consequence of XMSS using WOTS, which, while an extremely strong protocol in terms of security, due to requiring multiple hashes per message chunk and size of signature, ends up creating large messages which are slow to verify. This is due to the Winternitz parameter, which determines how many hashes are necessary per message and the corresponding size of the signature. The code is using a Winternitz parameter of 16, meaning each message digest of originally 256 bits is decomposed into base-16 representation, creating 64 hashes. An increase in the value of the parameter leads to faster verification, but larger signature size, while a decrease leads to slower computations due to the need of more hashing, but reduced signature size. The chosen value of 16 serves as a solid middle ground, which tries to balance both of these metrics without sacrificing the other.

Another key parameter is tree size, which determines how many one-time keys are generated in the KeyGen algorithm for XMSS. This affects how long users can participate in the protocol before having to send new keys, affecting the key generation speed and key storage. The implementation uses trees of height 16, meaning 2^{16} different one-time keys. Assuming epoch's of 1 hour, these 65536 instances only force users to update keys once every 7.5 years. This then leads to less updates in the logs, resulting in less overhead in correctness proofs. Recently, WhatsApp has suggested reducing its epoch duration to 5 minutes. With this time frame, users would instead change their key around once every 227 days (roughly twice a year).

The amount of time elapsed between key invalidation, that comes from the exhaustion of all one-time keys due to continuous signatures, is an important security consideration, as if a user is compromised during any point of the protocol, an attacker can predict all remaining one-time keys it possesses. This lasts until the user exhausts them and generates a new key pair, forcing an attacker to attempt to corrupt the device again. However, too short of an interval leads to users needing to constantly update their keys, which is inconvenient for them and for the server, which needs to store all new additions and generate proofs for their inclusion in new epochs, leading to larger correctness proofs and slower verifications.

XMSS is using SHA-256 as its hash function, an extremely common choice offering up to 128 bits of post quantum security and 256 bits of classical security. As such, XMSS security level for this execution is the same, which is important to know for considerations on the entire protocols security level.

VRF also allows for customization of the parameters used, which allows for balance tradeoffs between speed and security. At its core, VRFs utilize elliptic curve cryptography to ensure soundness. Currently, the implementation uses the P-256 curve, which is one of the standardized curves recommended by the American National Institute of Standards and Technology [23], offering up to 128 bit security level for

classic cryptography. While calculations with this curve are not post-quantum secure, it is still a widely used curve in modern applications, and so using it here is a good tradeoff between efficiency and security. It also utilizes SHA-256 as a hashing function, which is secure, as was previously mentioned.

Due to the protocol's use of multiple cryptographic primitives, each with its own security level, and additional considerations in quorum formation, the effective security level of the entire system is determined by its weakest component. In terms of the primitives used, currently VRF limits the security offered by the implementation to 128 bits, due to using the P-256 curve.

This is sufficient for the experiments made in the thesis, but in order to increase the security of the implementation, P-521 could be used instead, offering 256 bits of classical security [23], at the cost of larger keys and slower VRF generation and verification.

4.2 Entities

Here I explain the specifics of how each participant behaves in the implementation, along with details on how each action is being performed specifically in the code. In this implementation, both the users and the CM have their own programs, which communicate with one another using sockets, the foundation of communication in server-client applications.

4.2.1 Message Structure

In order for the CM and users to be able to recognize each other's messages they need to have a predefined structure that each can process. For this, all messages, both users and the CM's, begin with specific headers, identifying the type of request made.

The CM needs to be able to receive and handle any of the possible user requests, be them to add an XMSS or VRF public key to the log, request the current view of the log, add an endorsement or request the endorsement list, and then to call the appropriate handler function.

On the other hand, the clients need to be able to receive the answers to their requests such as the positions of the keys added, the view of the log and the endorsement list. The possible messages between users and the server can be found in Tables 4.1 and 4.2.

Furthermore, messages that have a payload attached, such as sending an endorse-

ment or the endorsement list, explicitly state the size of the payload to be received following the header, which then gets read and unserialized into the corresponding structure required. The functions used to serialize and unserialize the binary messages sent are automatically created by Protobuff when defining the structures used.

Functionality	**Phase**	**Header**	**Payload**
Request Initial Parameters	-	TrustedSetup	No
Add XMSS Key	Collection	addEntryXMSS	Yes
Add VRF Key	Collection	addEntryVRF	Yes
Request View	Distribution	download	No
Send Endorsement	Certification	EAK	Yes
Request Endorsement List	Verification	cc	No

Table 4.1: User Messages

Functionality	**Functionality**	**Header**	**Payload**
Send Initial Parameters	-	TS	Yes
Send XMSS Key Position	Collection	xmssEntry	Yes
Send VRF Key Position	Collection	vrfEntry	Yes
Send View	Distribution	VIEW	Yes
Send Endorsement List	Verification	CC	Yes

Table 4.2: Server Messages

4.2.2 Users

The user program begins by generating both the VRF and XMSS key pairs. As previously shown, both of these are fundamental for future steps of the prototype, and so the user stores them before sending the corresponding public keys to the server. This is so that other users can access them, so they can validate the proofs the user might create if they are chosen as members of the committee.

Afterwards, it receives the position of the log in which their keys were stored, which is then passed to the server when they get chosen as endorsers, so it can add the correct keys along with the endorsement and its corresponding proofs to the list of endorsements.

After initiating communication with the server, users have access to a terminal that allows them to perform their role in each of the main phases of CoD. Each one is only possible if the server is in the corresponding phase, with the users receiving an error if they try to participate in a phase that the server is not in.

After the initial desired number of users are connected to the server, they must all run the Trusted Setup procedure by inputting "ts". This then makes the user send a request to the server, asking for the initial parameters required for the first epoch, such as the number of users registered and its honest and malicious ratios, the necessary quorum size, the initial randomness seed, the randomness threshold, the number of initial entries in the log, and the root hash of the log. After this, the user is ready to participate in the first epoch of the protocol.

During the first phase, Collection, users are allowed to send keys to populate the log. For this, users can type the command "nk", which will add a copy of their public XMSS key to the log. In a real scenario, users would instead generate a new key pair each time they want to add a new public key, but this simplified simulation of key addition was made to facilitate the population of the log for testing purposes.

After the server creates the view of the epoch by advancing to the second phase, Distribution, users can then request access to it, which will be the view that they will compare future endorsements against. First, users check the correctness proof that is generated alongside the view, in order to confirm the updated log is indeed an extended version of the previous one, without any subtraction or change to previous entries. To do this, users receive the keys added in this epoch, and the new root hash. Users chain the new keys with the last root hash they confirmed was consistent, and verify if the final hash matches the one claimed by the server. If so, then the verification passes and users continue, and if they detect an inconsistency, since the new root does not match the root the server should have provided, they abort all future participation.

Immediately afterwards, users run a VRF using their corresponding secret key and the seed acquired from the previous epoch, or the trusted setup, depending on the epoch of the protocol. This then generates a random BIGNUM, the internal representation of an arbitrary-precision integer, larger than standard integers or doubles can handle, along with a NIZK proof that links this output to the seed and key used. The generated value is compared to the BIGNUM that corresponds with the previously received threshold. If the generated value is smaller, then the verification passes, and the user is selected to be an endorser.

Users who are selected to be an endorser then sign the view they received with their private XMSS key, creating a signature that implies that the user who created this signature believes this to be the correct view for the current epoch.

Users then send this signature to the CM in the Certification phase, along with the NIZK proof that was generated during the VRF, and the positions the users claim their public keys are located in. In the original description of the protocol, users only run the VRF in order to check their inclusion in the committee in the Certification phase, while in this implementation it is ran immediately after the successful

download of the state of the log. This was simply a design choice made so users can immediately send the endorsement when the Certification phase begins, as they do not need to wait to generate the endorsement.

Finally, during the Verification phase, users request access to the endorsement list created by the server, in order to try to verify consistency for this epoch. Users check each endorsement's XMSS signature, to ensure the message was signed by the correct individual, before checking the validity of the VRF. This implies checking the NIZK proof to confirm the number was generated correctly, and then comparing against the threshold, to ensure the user was selected as an endorser. If these checks pass, and the view that was endorsed matches the user's local view, the user increments the number of correct endorsements by one and compares this against the quorum size of the epoch, which was previously acquired in the previous epoch, or in the trusted setup if this is the first round of the protocol. If enough correct endorsements have been received, the user is assured consistency, and updates their parameters with the ones downloaded in this epoch's views, being ready to repeat the protocol for the next epoch. It also creates the seed for the new epoch, by hashing the list of endorsements received together and storing the resulting output. If not enough valid endorsements were received, the user cannot be sure of its consistency and ceases all participation instead.

In addition to this standard functionality, users also have a performance testing mode, initiated by turning the "PERFORMANCETEST" variable to 1. This changes the behavior of the users in a few specific ways, that are implemented in order to facilitate simulating user behavior for scenarios with a larger user base, in order to get efficiency estimates for these scenarios. The key changes come in the Certification phase, in which a user who tries to send an endorsement can send an arbitrarily large number of endorsements to the CM, all of which are identical, and in the Verification phase, in which the user now checks the validity of all endorsements. After this, the user prints the amount of time that was required for the verification of the validity of the endorsements.

4.2.3 Channel Maintainer

Unlike clients, who will only have to communicate with one other party, the CM needs to be able to handle requests from as many users as possible at any time. Furthermore, the server might need to send extremely large messages in the case of a large quorum size. In this case, it may not be able to send the entire list of endorsements in one write. As a solution to both of these problems, the infrastructure of the structure's messaging capabilities utilizes Epoll, which is a scalable input and output infrastructure that allows for the handling of multiple requests at the same time, while also handling partial write calls to buffers, which will be completed when enough resources are available. A scalable infrastructure such as this one is crucial for any protocol that requires a central figure to handle requests for millions to billions of users, like CoD was designed to handle. While this implementation

cannot handle that number of users simply due to design and hardware constraints, it gives insight into what might be needed for a production-level deployment of CoD.

On startup, the server creates two threads, one responsible for handling input through the terminal, that allows for the manual advancement of the current phase of the server, and the thread responsible for dealing with user requests. It is important that all functions that alter memory structures such as the log or endorsement lists have mutexes in order to prevent multiple users from affecting it at the same time, creating race conditions which could result in unpredictable behavior for the server. All communication is made using sockets, with the server keeping a list of active ones and accepting new connections.

The CM is responsible for handling the previously mentioned user requests, depending on whether the requests match the current phase the server is in. Currently, the server automatically advances the phase every two minutes, with it being possible to advance the phase manually with a "next" command. When the server changes from the Collection to the Distribution phase, it creates the view and parameters for the current epoch, which it then allows users to download. This also implies the creation of the correctness proof, which serves to users as proof that the new root hash is in fact merely an extension of the previous one, with only new entries being added to the log, without any modifications or exclusions. Finally, when beginning a new epoch, the server generates the seed for the next epoch in the same way as the users, by hashing together the list of all endorsements in order to create an unpredictable seed that offers no advantage to a possible attacker. It also cleans the list of endorsements and its view of the epoch, wiping the slate to prepare for the next epoch.

It is important to note that whenever the server creates the parameters for an epoch, there are multiple assumptions being made. First, is the Honest, Malicious and Inactive fractions, which the server has no way of knowing. As such only representative values are chosen. A real implementation can try to monitor its user base in order to try to reach some better approximations, but these values can be chosen in order to select the level of security the protocol tries to handle, by setting fixed ratios for which theoretical analysis can be made to prove their efficiency and correctness. For threshold and quorum size, the values used were constantly changing in testing in order to try different setups of the program with a varying number of users, and as such are also placeholders. The number of users is acquired from the number of connections being upheld by the server, with users trusting this value is correct as they cannot check the real number of users in the network due to the star topology of the communication network.

The CM stores keys received in the Collection phase in a hash-chain-structured key log. Any addition will change the root hash for the log, and so the server keeps track of the number of keys added in each epoch. After changing to the Distribution phase,

it uses this to know which keys need to be added to the correctness proof, which is then combined with the parameters of the epoch to create the view that users will download. After receiving a key, the server responds to users with the position in the log where the key was added. This enables users to later validate if an endorsement is coming from a valid endorser, without any need for communication with one another. This is necessary as in this implementation users do not have access to the key log itself, and download only the hash that represents its current state.

In the Certification phase, the CM receives the user's signed endorsements, which contain not only the claimed view of a committee member and its VRF proof, but also the positions in the log of the public keys that correspond to this user. The CM then creates a structure that includes the endorsement and the keys, and stores them in a list that contains all endorsements, which are then sent to users in the final phase of the protocol.

In order to better simulate how CoD protects against split-view attacks, the implementation also includes a "behavior" setting that simulates a compromised server. Whenever the behavior variable is set to true, the view distributed to users contains an additional key that is not present in the real log. Furthermore, endorsements received are kept in a separate list, which allows the server to store endorsements for both the legitimate and dishonest root hashes, which are then distributed to users in any way the server sees fit. By manipulating the size of the quorum and the probability of any user being chosen as a committee member, this feature shows how CoD protects against split view attacks, since not enough endorsements will be generated for both views to be accepted, meaning at most one group of users will claim they are consistent, and the other will abort.

This code can also be used to easily simulate a split view attack on a system with no consistency guarantees. To do so, all that is needed is for two users to join the protocol and obtain the initial parameters through the trusted setup, and then in between the distribution of the view for both of them, switch the behavior of the server, leading to the distribution of two views and ignoring the remaining phases of CoD. With no consistency verification in place, this suffices to launch the attack with users just blindly trusting that the server is correct.

5

Analysis

This chapter includes a security analysis of CoD, explaining how the relationship between Honest and Malicious users affects security considerations. Following this, I present the experimental results acquired from the implementation and discuss their implications for the efficiency of a possible implementation of Consistency-or-Die at a global level in real world applications.

5.1 Security guarantees

As was previously mentioned, CoD ensures statistical consistency guarantees by using the immense user base to generate a quorum of endorsements for the correct view, which was defined as the necessary number of endorsements users need in order to be assured all consistent users received the same view.

Before diving into the guarantees CoD provides, it is important to understand what the protocol requires for an epoch to be correctly handled by users, and guarantee the intended output. In order for users to reach a consistency guarantee at the end of each epoch, they need to have received and validated, at minimum, one quorum of signatures that match the view they originally received in the Distribution phase. This in turn implies that enough users must be chosen as endorsers per epoch to create the quorum of endorsements.

Due to the fact that each users' chances of being selected as an endorser are independent of others', i.e., one user being selected or not as an endorser has no effect on other user's selection, and that all users share the same chance of being selected, the total number of users who are selected follows a Binomial distribution $A \sim B(N, p)$, with N being the total number of users and p the probability that any given user is chosen as an endorser. As was previously mentioned, the total number of users present in a CoD protocol can be defined as the sum of all Honest, Malicious and Inactive users, i.e, $N = H + M + I$.

This allows us to approximate the total number of endorsers chosen given the chance of selection and total number of users, allowing for statistical analysis. Using these same principles, it is possible to also consider the number of Honest users selected

as endorsers, and number of Malicious users selected as endorsers as Binomial distributions, respectively $X \sim B(H, p)$ and $Z \sim B(M, p)$.

In order for the protocol to be able to function continuously given correct execution, it is crucial that enough Honest users are selected as endorsers in each epoch to create at least a quorum of endorsements. This means that for any given epoch, the chance that the number of honest endorsers generated is smaller than the quorum size must be negligible. This can be expressed as $Pr(X < k) < 2^{-b}$, with b being the target bit security level, which determines what is considered negligible.

However, this condition alone is not enough for CoD to protect against split-view attacks. In order to achieve that degree of protection, there must be enough legitimate endorsements for a given view in order to continue the protocol, but there cannot be enough endorsements for another, distinct view to be validated, as this would allow a corrupt CM to choose which endorsements to send in order to make users pass the consistency check for each view. A corrupt CM has the best chance possible to create such a scenario by creating only two views and giving each one to a different half of the Honest population. This then allows malicious users who are chosen as endorsers to endorse both views, and maximizes the number of endorsements acquired for each view from honest users. The number of endorsers for each view in this scenario follows the binomial distribution of $Y \sim B(M + H/2, P)$, as all malicious endorsers endorse both views, but honest endorsers would only endorse the first one they receive as this is the expected behavior. In order to prevent split view attacks in these scenarios, the probability that there is at least a quorum of endorsers selected from this distribution must be negligible. As such, it is necessary that $Pr(Y \geq k) < 2^{-b}$ in order to be confident that a split view attack is not possible.

These distributions can be plotted, allowing for a clearer representation of their impact in the protocol, as can be seen in Figure 5.1. This graph represents the distribution of X and Y for a user base of 2^{30}, approximately 1.1 billion users, with a 60/20 honest/malicious ratio. At any given point, the y value represents the probability of exactly that many endorsers being selected.

The orange curve represents Y and the blue curve represents X, with each value on the X axis representing the different number of endorsers selected, and the corresponding value on the y axis representing the probability that exactly that many endorsers are selected. The probability that there are not enough Honest users to obtain a valid quorum, $Pr(X < k) < 2^{-b}$, can be visualized in the graph by selecting a single point, and calculating the area under the blue curve to the left of the given point. Similarly, the probability that a split-view attack is successful, $Y \sim B(M + H/2, P)$, can be seen by selecting a point and calculating the area under the orange curve to the right of that point.

As such, the smallest quorum size that can hold both of these proprieties is the first

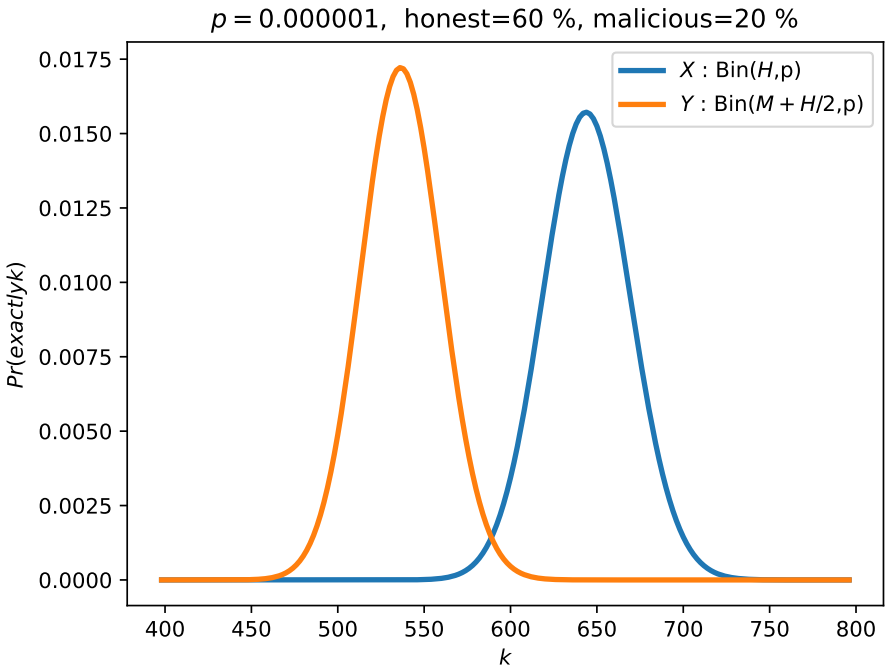


Figure 5.1: Binomial Distributions for a user base of 2^{30} (approximately a billion) users, with 60% being honest and 20% malicious, with a probability of a user being selected as an endorser being 10^{-6}.

point for which the area to the right of it that belongs to the orange curve must amount to be negligible, while the left area of the blue curve must also be negligible. For Figure 5.1, this point is around 600, which then serves as a solid choice for quorum size given these parameters.

5.2 Maximum Malicious Ratio

Here I present my finding for the maximum number of malicious users CoD can hope to defend against efficiently.

For efficiency reasons, CoD's users want to minimize the number of endorsements they need to verify in order to be confident of consistency. As such, it is important that the quorum size is the minimum amount necessary for the confirmation of consistency, to prevent unnecessary calculations.

Given the previous analysis, it is possible to accurately determine this point by selecting the first point in the graph for which both necessary conditions are met. However, this requires that such a point does exist, which depends on the Honest and Malicious User's ratios.

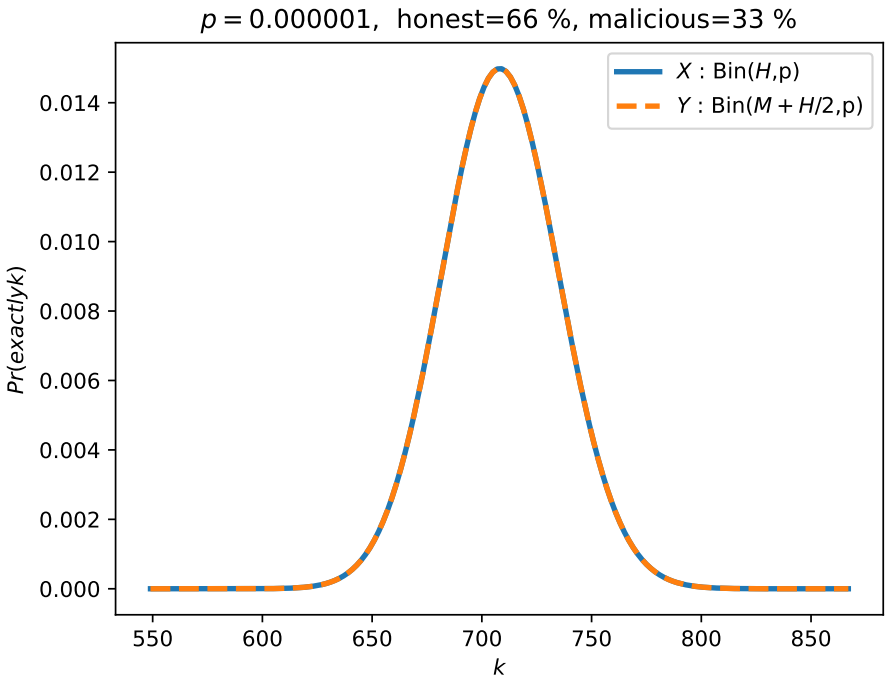


Figure 5.2: Binomial Distributions for a user base of 2^{30} users, with 66% being honest and 33% malicious, with a probability of selection of 10^{-6}.

With the increase of the malicious user ratio, the two curves become closer to each other, requiring a larger quorum value to achieve the necessary conditions, given a fixed value for the probability of any given user being selected, and the total number of users.

When the number of malicious users reaches 1/3 of the total active user base, i.e, $M = 1/3N$ and $H = 2/3N$, assuming no inactive users (something that while unrealistic, can be taken into account when defining the parameters of the protocol by simply scaling the user base and honest and malicious ratios accordingly), a problem arises. Since now $M = H/2$, the dual condition of $Pr(Y \geq k) < 2^{-b}$ and $Pr(X < k) < 2^{-b}$ becomes impossible to uphold, as Y and X are now equal, since $Y \sim B(M + H/2, p) = X \sim B(H, p)$, and thus it is contradictory for the same distribution to be both larger or equal and simultaneously lesser than the quorum value. This then implies that for 1/3 of malicious users or higher, there is no efficient quorum size selection that can guarantee both a negligible probability for a split-view attack, and a negligible chance of a quorum of endorsements not being reached by the honest users.

This scenario can be seen in Figure 5.2, which shows the distribution curves for a user base of 2^{30}, with a 66/33 honest to malicious ratio. As is clear to see, the two curves completely overlap, meaning there does not exist a single point in which the two conditions can possibly hold simultaneously. This shows that for any ratio equal to or smaller than H/M = 2, there is no point in which a quorum size can

be efficiently chosen, which shows the limit of how many malicious users can be tolerated in the system.

As it is unrealistic to expect users to be able to monitor the honesty of all other users, or to trust the claims the CM gives about the total malicious ratio, these parameters are best suited to be chosen as deterministic values in Trusted Setup, which are maintained for future executions, acting as a maximum the system can handle. The discovery of a maximum threshold allows implementers to know how far these limitations reach theoretically and adjust their chosen ratios accordingly.

5.3 Analysis of Seed Generation

As mentioned in the definition of the CoD protocol, the SeedGen function is used by users to generate the randomness seed that is used to verify whether or not the user should act as an endorser in any given epoch, used to create the proof the user was selected as an endorser, and to verify the proofs other endorsers generated, in order to guarantee that they are acting honestly and not trying to generate endorsements when they were not selected. Thus, it is clear that users who are consistent with one another should generate the same seed, as if not, users will not be able to accurately verify whether or not proofs other participants generate are correct.

Having all users share the same seed raises the question of how they should acquire or derive the seed while still being unpredictable, so that attackers are not allowed to predict a seed for multiple rounds in the future, generate keys until they find a quorum of keys that pass the VRF, and use it to launch a split-view attack that is guaranteed to succeed.

Clearly, the choice of algorithm for seedGen affects the possible security of CoD drastically. Here I present some possible implementations of SeedGen, as well as the necessary considerations that come with them, along with how realistic they are to implement and if they should be used for a protocol such as CoD.

5.3.1 Randomness beacon

Online distributed randomness beacons are publicly verifiable sources of randomness, where users can ask for the randomness value associated with any given round, and receive a value, along with a common reference string used in order to verify the given value. For our protocol, this offers both a source of true randomness while also being easily distributed by all users, as they can obtain the value simply by polling the provider with the additional advantage that is un-grindable, as an attacker who may have the seed for a given epoch cannot predict what the seed is for a future epoch. However, this approach requires trusting a central third party, the randomness distributor. This goes against CoD's primary goal, to create a key consistency procedure for which users do not trust a third party. If the distributor were to be

compromised, it could manipulate the seed generated in order to try to have more malicious users pass verification, with users still believing the key they received was correctly used.

The CoD implementation currently uses the Drand randomness beacon [24] in order to generate an initial seed, but changes to another method of seed generation in future epochs. In a real deployment of CoD, since users are already forced to trust the CM in the first epoch due to the Trusted Setup procedure, it is acceptable for the CM to give any seed, so using a randomness beacon is a valid starting point.

5.3.2 Fluctuating verifiable information

Publically available information that is easily verifiable and that fluctuates unpredictably can also be used as a randomness source to create a seed in a way that all users can agree. Examples of this can be the newest addition in a publicly verifiable blockchain, or the result of a given stock index at the start of the current epoch. These values, are both unpredictable and verifiable since users can simply lookup the newest values from trustworthy sources.

However, they also come with their drawbacks, which make them hard to adapt into seed generation for CoD. In order for CoD to be secure, the seed must change in a verifiable way in every single one of its epochs. Having consecutive epochs with the same seed is extremely unsafe, as the same users would be chosen as endorsers, and an attacker can check which users were endorsers in the previous epoch, and specifically target those users in order to launch a split view attack.

This is the problem with this approach, as there is no guarantee as to when or if an update is published at any given time. For Blockchain, the newest update depends on whether or not a user has been able to successfully mine a block in the interval between any two given epochs, which is not guaranteed. This is called block time, and currently Bitcoin's block time is of approximately 10 minutes, slightly larger than WhatsApp's target epoch duration. For stock values, due to the fact that the market opens and closes at the same time worldwide, there are periods of time in which the market is closed, meaning no stock fluctuations occur, which would cause identical seeds for consecutive epochs, thus breaking the protocol.

5.3.3 Endorsement generated seed

Another approach for seed generation is to utilize the already existing randomness that comes from the endorsement committee, to introduce randomness to the next epoch's seed. To do this, users who successfully complete the Verification phase of any given epoch, hash together all endorsements received, and use this hash as the basis for the seed used in the following epoch. This then ensures that the seed generated is not manipulated by an adversary, as for a user to have achieved consistency,

they must have received a quorum of endorsements, which implies that some of the endorsements were legitimate, since, as shown in the analysis of the security of the protocol, the number of endorsements that are generated by malicious user's and half of honest users are not sufficient to reach a quorum.

This in turn implies an attacker cannot predict a future seed, as they have extremely limited control over the final hash due to the endorsements that originate from honest users. However, as the endorsement list is finalized in the end of the Certification phase, a corrupt CM has half of an epoch's duration to grind possible keys, as they have until the end of the following epoch's Collection phase to add any key they might have found which would pass the VRF check with the endorsement based seed. Given the current time estimates for the duration of an epoch, of about five to ten minutes [25], this is not a meaningful enough amount of time for an attacker to generate enough successful keys to threaten the security of the protocol. Furthermore, keys added from this possible grinding lose any advantage they may have in the epoch following their inclusion in the log, as the new seed will have been generated in an unpredictable manner.

A corrupt CM could try to send differing endorsement lists, in order to generate multiple seeds. This is due to the fact that for any given epoch, the CM can receive more endorsements than what is needed for a quorum, as the number of expected endorsers and the number of users selected as endorsers in practice is not necessarily the same. This could then allow the server to handpick different endorsement lists to different users, who will pass the consistency check and generate distinct seeds. This creates a similar situation to a split-view attack, but with the key difference that users still agreed the view they received was the same, with only the seed being different. In the following epoch this inconsistency will always be detected, as users with different seeds will refuse each others VRF proofs, which in the worst case scenario, will make users abort due to not receiving enough valid endorsements.

This approach for seed generation is the one that is implemented in the proof of concept, as it was the one I found to be the most realistic, since it does not force users to trust a third party, while still offering randomness and allowing users to verify that others generated the same seed.

6

Results

6.1 Testing Methodology

The main result of this thesis is the proof-of-concept implementation of the CoD [5] consistency protocol, which serves to demonstrate the proposal's intended functionality, and allows extrapolation of possible implementation specifics and conclusions on what is necessary for large-scale deployment. Furthermore, the implementation allows for experimental testing in order to obtain initial metrics for how efficient this proof of concept is, in terms of communication overhead and computations done by users. While these results are only representative of this implementation, which is not optimized, they still allow an extrapolation of what a refined implementation can achieve, further bolstering the case that CoD is an efficient and secure protocol that scales to extremely large user bases and can be adapted for the largest messaging applications currently being used worldwide.

In order to estimate the viability of the protocol in terms of efficiency, there are two main metrics that are important to measure and consider: execution time, meaning the amount of time participants are expected to be doing computations, and memory consumption and overhead, meaning the amount of memory that users need to have to be able to run the protocol and the information that is sent in communication between the user and server.

It is crucial that the protocol is capable of being handled by various types of users with varying levels of hardware. In order to simulate these differences in performance, I tested the protocol on two different machines:

- Machine I: Dell Inc. OptiPlex 7060, 32GB Memory, Processor Intel Core i7-8700 x12, for faster experiments;

- Machine II: Dell Inc. Precision 5520, 16GB Memory, Processor Intel Core i7-7820HQ x8, capped to 4GB memory and 2 cores, with capabilities similar to those of an iPhone 11.

To measure the efficiency of the protocol, it is important to first understand which phases of CoD are the most computationally intensive, and where users are expected to spend most of their execution time. Due to the nature of CoD, the Verification phase is the slowest, as the amount of work it takes grows quickly with the number of users when compared to other phases. In order to justify this assessment, experiments were ran to measure the execution time of key procedures of each of the remaining phases, which can be seen in Table 6.1, which were acquired using the aforementioned Machine I.

Phase	**Procedure**	**Time (s)**
Collection	Key Generation	154.092
Distribution	View Download and Committee Verification	0.013
Certification	Generation of Endorsement	0.0205

Table 6.1: Execution times for key procedures of the Collection, Download, and Endorsement Phases

It is important to understand the significance of each of these values and how often these procedures are executed for each user. Key generation is the main step in the addition of keys in the Collection phase, being run once by users every time they want to generate a new key pair to add to the log, typically due to their previous one being expired or compromised. While this procedure is slow, it is ran very rarely due the current implementation using secret XMSS keys with height 16, this means that, assuming one signature every epoch, the average execution time for key generation per epoch is 0.00234 seconds meaning that it has very little impact in average execution times.

In contrast, every user needs to download the current view and check whether or not they are committee members every single epoch, meaning this section is ran much more commonly. However, it also presents an extremely small computational workload, similarly to the generation of the endorsement, which only the users who were chosen to endorse need to do. These checks do not scale with the number of users or of endorsements, as they are independent and so, have approximately constant execution times.

While the verification that the view is an extension of the previous one scales with the number of new keys added to the log in any given epoch, these can be balanced by the channel maintainer and distributed throughout multiple epochs by withholding additions to any given epoch in order to avoid overloading users.

2^b	H/M (%)	Quorum Size	List Size (MB)	CheckCon Time (s)	
				Machine (I)	Machine (II)
128	75/5	2,127	6.94	3.49	7.73
128	70/10	3,509	11.44	5.77	11.62
128	65/15	6,951	22.66	12.25	24.02
128	60/20	20,537	66.95	32.11	70.41
256	75/5	4,329	14.11	6.65	14.86
256	70/10	7,141	23.28	10.94	25.71
256	65/15	14,145	46.11	23.57	45.72
256	60/20	41,816	136.32	69.80	155.91

Table 6.2: Bottleneck processing (endorsement lists) at 128-bit and 256-bit security levels on OptiPlex (I) and Precision (II) machines, for 2^{30} users.

Execution times associated with verifying endorsement lists of various sizes were tested in order to compare efficiency considerations for various use cases. Each experiment depended on the expected quorum size for a setting with a billion users and a varying honest/malicious ratio, and target levels of bit security. The execution times presented are the averages of multiple executions, while the endorsement list size is fixed, as it does not change between experiments with the same number of endorsements. The results of these experiments can be found in Table 6.2. All results assume an inactive fraction of 20%. The chosen ratios of honest to malicious users were chosen arbitrarily, as possible security breakpoints that a deployed solution of CoD could choose to consider as the maximum expected number of malicious users. A comparison between execution time for the Verification phase for both target machines can be seen in the Figure 6.1.

6.2 Analysis of results

These results allow for multiple vital observations when discussing the viability of the protocol. First, it is important to note that both execution time and memory space occupied by the endorsement list both scale linearly with the quorum size, suggesting sustainable scalability. Furthermore, we can see that, even for the worst case of 256 bit security with a malicious ratio of 60/20 with a billion users, users take, on average, two and a half minutes (155 seconds) to verify the entire endorsement list, with it occupying 136 megabytes. Despite this scenario being extremely demanding, these results show that the protocol is still efficient for users in middling machines, and can be executed in a reasonable amount of time.

For comparison, other data logs are currently using epochs of approximately 7 hours

such as Cloudflare's Nimbus2025 [26], while WhatsApp is currently using epochs of around 5-10 minutes [25]. With these epoch intervals in mind, we can conclude that, for intervals like Cloudflare's, CoD is an extremely appealing choice, as it takes a very small fraction of the entire epoch duration, meaning users will always be able to complete their computations. For WhatsApp's smaller period, the results of this implementation also suggest that CoD is an applicable choice. Even though the consistency check can take up to around half of the entire epoch, in the conservative case of 5 minute epochs, there are multiple factors that point to CoD still being applicable in large scale deployments.

First, XMSS verification is slow, and a production level implementation could be using a faster KES, which would dramatically increase the execution of the Verification phase, as the overwhelming majority of this phase is checking the XMSS signature. This is also a good solution to the large overhead costs visible in the results, as the large signature size is a current drawback of the XMSS implementation that was used for the proof of concept. An aggregate KES, which combines all signatures into one could drastically reduce this cost and the memory required to run the consistency check.

Furthermore, it is not necessary for phases to be split equally during the execution of an epoch. It is possible that the 5 minute epochs can be adjusted to allow for leeway in the verification step, by shortening the allotted time for the remaining phases. As long as users know how long each phase lasts and when each epoch starts, smaller durations for different phases are not an issue for users, as long as they are still reasonably large enough so slower users can still complete all required computations in time for the following phase.

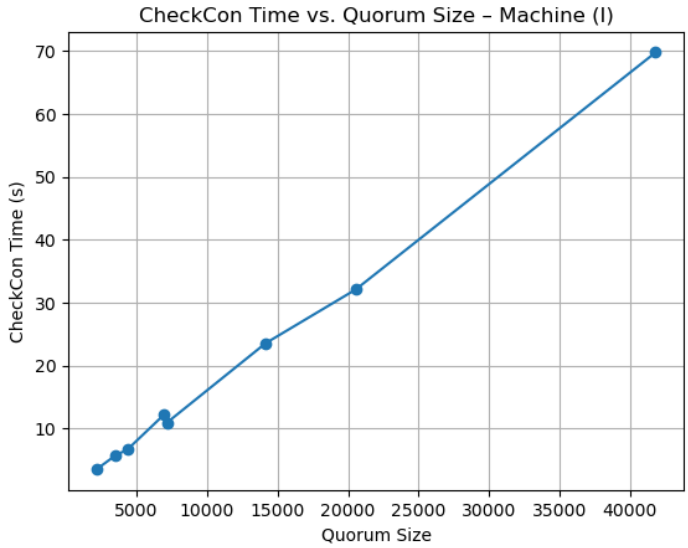
Finally, it is important to restate that this implementation is not completely optimized, and its results are only meant to be representative. Since even these results still point to the protocol being efficient enough for widespread usage, it is easy to imagine that a refined implementation would present better results and thus be viable for the current largest messaging applications.

It is also important to note the large size associated with these endorsement lists of 136 megabytes. This is much larger than standard messages or file attachments users download, which is very unsatisfactory for a protocol that would want to send these messages every 5 minutes, depending on epoch duration. This large size comes from the XMSS signatures, each being about 3000 bytes large, something that does not scale well into tens of thousands of endorsements. As previously mentioned, using an aggregate KES which combines all signatures into a single signature, which only passes if all individual original signatures were correct, is a solid option to reduce this message size.

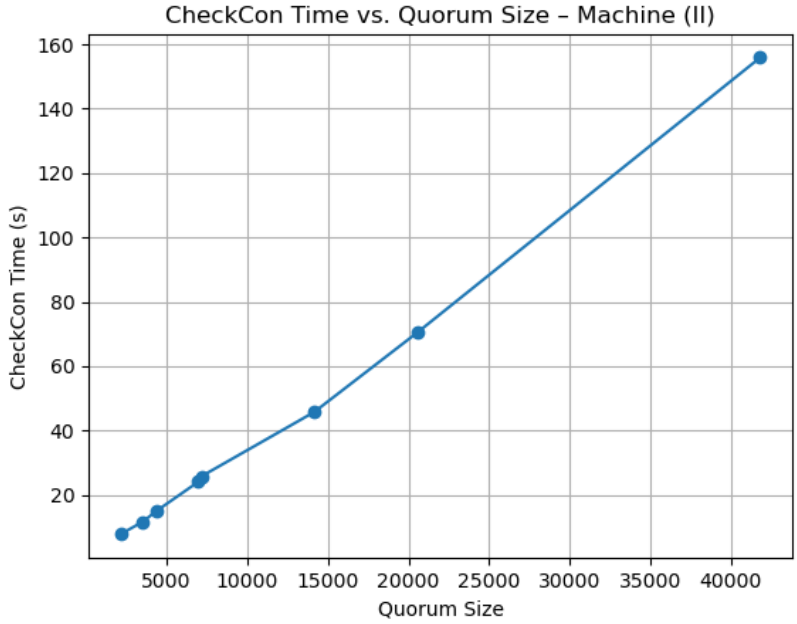
Considering different parameter choices in chosen KES, such as the Winternitz pa-

rameter used in signature schemes that use Winternitz One Time Keys, like XMSS, is another possible way to reduce message size, with the important caveat that this can also affect verification speed. A good balance between these two and the target security level is crucial for the protocol to be practical, and so they must both be considered when choosing the KES to use, as well as its primitives.

WhatsApp's Maximum size for audio and video messages is 16MB [27], which serves as an indicator for maximum size users could be expected to download and so a good benchmark for what CoD should try to have as a maximum size limit for the endorsement lists. Using this limitation, we can see that only the most optimistic option of 75/5 in regards to Honest/Malicious limit for 256 bits of security level, while for 128 bits, 70/10 is also still within this bound, showing that it is still possible to achieve some security guarantees using this unoptimized XMSS KES.



(a) Execution time vs. quorum size Machine (I)



(b) Execution time vs. quorum size Machine (II)

Figure 6.1: Experimental results for Check Con on Both test Machines

7

Discussion

This chapter gives an overview of the thesis, restating the motivation for the work and achievements made, while also discussing future research vectors for Consistency-or-Die and its possible applications.

7.1 Future work

This thesis serves as a supporting work to the original Consistency-or-Die proposal by Pagnin et al. [5], presenting solid evidence of CoD's efficiency and security guarantees both at a theoretical and practical levels, demonstrating that the protocol can be used as a solution to the problem of guaranteeing consistency in Key Transparency logs. However, there are still some aspects of the protocol that need further exploration or definition in order to make an effective production-level deployment complete and easy to implement.

First, currently the protocol does not mention how the server should react when a new user tries to join, and what requirements the new user needs in order to be sure the protocol is running correctly. It is important to note that a user who wants to join after the initial epoch would need more information than the common reference string generated in the Trusted Setup. This is because the joining user will lack the knowledge of the current randomness seed, which is crucial for endorsement generation/verification.

A solution to this problem could be for users to always trust the server on their first use, with the server providing the current view of the epoch when users first want to join the protocol, but this solution goes against CoD's objective of users only needing to trust the server in the original setup of the protocol.

Users could also ask others for the view they are currently seeing, and agglomerate a collection of views to see which is the most popular, and trust that to be correct, sending a message to the server that they have joined the protocol along with the view they are seeing. The server could then verify the view the user is claiming matches its view and accept the new user if so, or refuse them otherwise. This ap-

proach requires users to have a peer-to-peer connection with the users from whom they are requesting a view, or an out-of-band channel instead, and is still vulnerable to attackers who collude with a corrupt server to feed an incorrect view to users, which they cannot verify. Trust on first use is currently utilized widely for various protocols as an initial way to allow users to join. While it is not perfect for a protocol that wants to place minimal trust in the central party such as CoD, trust on first use is still acceptable and the simplest answer, but future research on a solution to allow users to join the protocol with as little trust in the CM is needed.

Allowing users to join and leave midway through also brings another problem with the fact that the total number of users would change in between epochs. If there are no considerations for how much the number of users can change, a compromised server can manipulate the number of users it presents when participants request a view in order to break the protocol. By claiming there to be a small number of users, the associated quorum size will be much smaller, which could then allow an attacker to launch a split-view attack, as users will need fewer endorsements. On the other hand, claiming there are more participants than there really are makes users need more endorsements than what will likely be generated, causing users to abort due to not being able to reach a quorum of endorsements. A mechanism to either control how much the total number of users can vary each epoch or to allow users to verify that the number of users claimed by the server is correct is crucial in order to prevent these attacks, allowing for new users to join the protocol without affecting security guarantees, but is something the current CoD proposal does not present, and a key area to focus on.

On the implementation side, research is needed on ways to reduce the size associated with the endorsement list, by using a more optimized KES, such as RapidXMSS [28], or an aggregate KES [29] in order to reduce endorsement sizes and thus lower the associated memory consumption for users. Another solution could be to allow users to partially download endorsement lists, and after verifying that segment request a new one. This would in turn force the server to be able to handle more requests, and keep track of which sections it has sent to which users, causing more overhead and possible communication slowdown. The results from the proof of concept implementation currently point to the endorsement list size being the critical viability bottleneck, forcing users to download extremely large message sizes, something that is not acceptable by current standards.

Another important area for future work is tied to research into making it post-quantum secure. With the increased study of quantum computers and the revolutionary computations that can be made with them, multiple cryptographic protocols that were considered secure were found to have flaws that quantum computers could exploit in order to successfully break them. As such, research is being done in order to find cryptographic protocols that are post-quantum secure, meaning that even with a quantum computer, an attacker would not have a meaningful chance of beating the protocol. A deep investigation into the protocol at the theoretical level would

be required to test whether or not it is post-quantum secure, as well as taking into account any implementation specifics that may be unsafe against such an attacker. Elliptic curve cryptography, which is currently used in the VRF generation of the protocol, has already been proven to be unsafe against quantum computers, due to the existence of Shor's Algorithm, which breaks the Discrete Logarithm problem, whose difficulty is the main security guarantee of elliptic curve cryptography. Future research focusing on post-quantum verifiable random functions would greatly help in CoD's future as a secure cryptographic scheme against post-quantum adversaries.

Finally, research is currently being done to discover a simple formula that allows for the calculation of the quorum size and corresponding probability of a user being chosen as endorser needed to achieve a target security level, given the total number of users and malicious and honest user ratios. This formula is crucial to allow versatility in CoD, allowing users to dynamically change the number of endorsements they need to be confident that all consistent users share the same information. Due to the fact new users are always joining messaging applications, the ability to change the quorum size to match the new user base in each epoch is extremely important in order to guarantee protection against split-view attacks in expanding systems.

7.2 Achieving Key Transparency and Implications

As mentioned earlier, key log consistency is only half of what is required in order to achieve a full Key Transparency log, the real goal and final target of this research. In order to achieve this, it is crucial that a protocol such as CoD is paired with a well-tested and efficient Verifiable Key Directory in order to achieve complete Key Transparency. This assures both the append-only nature associated with VKDs, as well as their correct handling of new additions to the log, along with the verification that all users correctly receive the same view of the log at any given epoch associated with CoD, ensuring functional and efficient Key Transparency. Already existing logs such as Parakeet [9] and CONIKS [1] are great candidates that can be adapted in order to accommodate the CoD infrastructure, creating a robust Key Transparency log solution that is efficient and scalable and can be used by multiple messaging applications and other communication systems, preventing split-view attacks and malicious alterations to the log in the case of a corrupt server, which lets users be confident in the reliability of the application being used.

This thesis shows the importance of there being a reliable, measurable and efficient method for ensuring consistency, which CoD fulfills, in tandem with Verifiable Key Directories to create functional Key Transparency. Efficient and easily implementable Key Transparency has multiple real-world applications, with the clearest use cases being messaging applications, as I have mentioned throughout the thesis. However, this protocol can also serve as a blueprint to allow for more widespread applications of consistency over logs or data structures, not necessarily related to keys. A method that allows multiple participants to guarantee consistency for any

singular data structure can be built on top of CoD, allowing users to endorse the current view of data and then verify others' endorsements. This then extends the protocols use cases immensely, such as scenarios where users wish to retrieve data from a server, but want to be sure all other users received the same data. This is useful for software distribution services, as an example, where consistency is crucial in order to prevent targeted malware downloads.

7.3 Ethics and Sustainability

As with all other works in computer science, it is important to reflect on the possible impact the widespread use of a protocol such as Consistency-or-Die can have in the environment, as well as possible ethical considerations that come with it.

In terms of ethical considerations, CoD leads to increased confidence in private conversations, which allows users to feel secure from third parties listening in on their messages. For regular honest individuals this is a positive achievement, but secure communications are also something dishonest individuals can use to share illegal content or create secure channels for organized crime. While this risk is outweighed by the positives that come with widespread consistency and privacy, they are still present and as such must be considered.

CoD is a protocol that is meant to increase privacy and confidentiality, two of the central topics of ACM's Code of Ethics and Professional Conduct [30], and as such future deployments of the protocol should keep these principles in mind by assuring that users only reveal the information that is strictly necessary for participation. To do this, it is imperative that the underlying structure used as a VKD focuses on privacy, following in the footsteps of ELEKTRA [3] and OPTIKS [4].

When it comes to sustainability, as was shown earlier in the thesis, CoD is an efficient protocol, that is not associated with considerable additional costs in terms of computational power and execution time. As such, it is possible to conclude that its widespread use would not have a large impact on the environment, contributing very little to energy consumption in phones and other devices which are running the protocol. While the central server responsible for storing and distributing the VKD requires more resources than typical users, these kinds of servers are already common practice for the largest messaging applications.

7.4 Conclusion

This thesis presents a proof of concept implementation of the Consistency-or-Die protocol by Pagnin et al. [5], capable of simulating the intended execution of the

program at a smaller scale. This implementation was then tested and analyzed in order to measure the computational workload and memory storage required to operate, in multiple different scenarios meant to simulate real-world execution. The results point to CoD being an efficient protocol in practice, capable of being implemented in the current largest messaging applications used worldwide. Furthermore, it also presents an analysis on how the protocol cannot efficiently handle a scenario where a third or more of all users are malicious, presenting a theoretical explanation for this due to the approximation of the distributions of honest users, and the distribution of malicious users along with half of honest users, which is the distribution tied to successfully launching a split-view attack. In addition, it also covered the importance of the SeedGen algorithm that is used in CoD, while presenting multiple possible options, their upsides and flaws, before focusing on the endorsement-based seed generation that is currently being used in the proof of concept implementation. Finally, a discussion about further research that can be done with CoD is shown, along with a list of fields where transparency logs might be applicable, as well as other works that can be combined with it in order to create functional and complete Key Transparency.

Bibliography

[1] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: Bringing key transparency to end users," in *24th USENIX Security Symposium (USENIX Security '15)*, Washington, D.C., 2015, pp. 383–398. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara.

[2] A. Mallik, "Man-in-the-middle-attack: Understanding in simple words," *Cyberspace: Jurnal Pendidikan Teknologi Informasi*, vol. 2, no. 2, pp. 109–134, 2019.

[3] J. Len, M. Chase, E. Ghosh, D. Jost, B. Kesavan, and A. Marcedone, "Elektra: Efficient lightweight multi-device key transparency," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2915–2929.

[4] 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.

[5] J. Brorsson, E. Pagnin, B. David, and P. S. Wagner, "Consistency-or-Die: Consistency for key transparency," *Cryptology ePrint Archive*, 2024.

[6] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *Proceedings of the 40th Annual Symposium on the Foundations of Computer Science (FOCS '99)*, IEEE Computer Society Press, 1999, pp. 120–130. DOI: 10.1109/SFFCS.1999.814584. [Online]. Available: http://nrs.harvard.edu/urn-3:HUL.InstRepos:5028196.

[7] A. Huelsing, D. Butin, S.-L. Gazdag, J. Rijneveld, and A. Mohaisen, *XMSS: eXtended Merkle Signature Scheme*, RFC 8391, May 2018. DOI: 10.17487/RFC8391. [Online]. Available: https://www.rfc-editor.org/info/rfc8391.

[8] K. Birman, "The promise, and limitations, of gossip protocols," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 5, pp. 8–13, 2007.

[9] H. Malvai, L. Kokoris-Kogias, and A. Sonnino, "Parakeet: Practical key transparency for end-to-end encrypted messaging," *Cryptology ePrint Archive*, 2023.

[10] S. Micali, M. Rabin, and J. Kilian, "Zero-knowledge sets," in *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, IEEE, 2003, pp. 80–91.

[11] H. S. de Ocáriz Borde, "An overview of trees in blockchain technology: Merkle trees and merkle patricia tries," *University of Cambridge: Cambridge, UK*, 2022.

[12] B. Chen and Y. Dodis, "Rotatable zero knowledge sets: Post compromise secure auditable dictionaries with application to key transparency," in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2022, pp. 547–580.

[13] 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.

[14] 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)*, 2018, pp. 105–114. DOI: 10.1109/ICDMW.2018.00022.

[15] J. Bonneau, "Ethiks: Using ethereum to audit a coniks key transparency log," vol. 9604, Feb. 2016, pp. 95–105, ISBN: 978-3-662-53356-7. DOI: 10.1007/978-3-662-53357-4_7.

[16] T. Monz, D. Nigg, and E. A. Martinez, "Realization of a scalable Shor algorithm," *Science*, vol. 351, no. 6277, pp. 1068–1070, 2016.

[17] Y. Dodis and A. Yampolskiy, "A verifiable random function with short proofs and keys," in *Public Key Cryptography - PKC 2005*, S. Vaudenay, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 416–431, ISBN: 978-3-540-30580-4.

[18] C. Rackoff and D. R. Simon, "Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack," in *Annual international cryptology conference*, Springer, 1991, pp. 433–444.

[19] Anonymous. "CoD implementation." (2025), [Online]. Available: https://anonymous.4open.science/r/CoD-0313/README.md.

[20] J. B, *SpeedTestCoD*, https://github.com/joakimb/SpeedTestCoD, Accessed: 2025-05-03, 2024.

[21] S. Popi, D. Pezer, B. Mrazovac, and N. Tesli, "Performance evaluation of using protocol buffers in the internet of things communication," in *2016 International Conference on Smart Systems and Technologies (SST)*, 2016, pp. 261–265.

[22] J. Buchmann, E. Dahmen, S. Ereth, A. Hülsing, and M. Rückert, "On the security of the winternitz one-time signature scheme," *International Journal of Applied Cryptography*, vol. 3, no. 1, pp. 84–96, 2013.

[23] L. Chen, D. Moody, K. Randall, A. Regenscheid, and A. Robinson, *Recommendations for discrete logarithm-based cryptography: Elliptic curve domain parameters*, en, 2023-02-02 05:02:00 2023. DOI: https://doi.org/10.6028/NIST.SP.800-186. [Online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=935198.

[24] I. Rhee, A. Warrier, J. Min, and L. Xu, "Drand: Distributed randomized TDMA scheduling for wireless ad-hoc networks," in *Proceedings of the 7th ACM international symposium on Mobile ad hoc networking and computing*, 2006, pp. 190–201.

[25] Security Cryptography Whatever. "Whatsapp key transparency with Jasleen Malvai and Kevin Lewi." https://securitycryptographywhatever.com/

2023/05/06/whatsapp-key-transparency/. (May 6, 2023), (visited on 04/21/2025).

[26] Cloudflare Certificate Transparency. "Merkle town (nimbus2025 log)." (), [Online]. Available: https://ct.cloudflare.com/logs/nimbus2025 (visited on 04/21/2025).

[27] Amazon Web Services, *Supported media file types and sizes in whatsapp*, https://docs.aws.amazon.com/social-messaging/latest/userguide/supported-media-types.html, Accessed: 2025-04-29, 2025.

[28] J. W. Bos, A. Hülsing, J. Renes, and C. van Vredendaal, "Rapidly verifiable XMSS signatures," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 137–168, 2021.

[29] B. David, R. Dowsley, A. Konring, and M. Larangeira, "MUSEN: Aggregatable key-evolving verifiable random functions and applications," in *International Conference on Security and Cryptography for Networks*, Springer, 2024, pp. 317–337.

[30] Association for Computing Machinery, *ACM Code of Ethics and Professional Conduct*, https://www.acm.org/code-of-ethics, Adopted by the ACM Council on June 22, 2018, 2018.