Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

TU UB WIEN Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

TU WIEN FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Randomness for Blockchains

Generating Publicly-Verifiable and Bias-Resistant Randomness in Decentralized Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Philipp Schindler, BSc
Matrikelnummer 1128993

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Mitwirkung: Dipl.-Ing. Aljosha Judmayer

Wien, 6. Oktober 2017

Philipp Schindler	Edgar Weippl

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Randomness for Blockchains

Generating Publicly-Verifiable and Bias-Resistant Randomness in Decentralized Systems

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Philipp Schindler, BSc
Registration Number 1128993

to the Faculty of Informatics

at the TU Wien

Advisor: Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Assistance: Dipl.-Ing. Aljosha Judmayer

Vienna, 6th October, 2017

Philipp Schindler　　Edgar Weippl

Erklärung zur Verfassung der Arbeit

Philipp Schindler, BSc
Quellenstraße 109/20, 1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 6. Oktober 2017

Philipp Schindler

Acknowledgements

A thank you goes to my parents, who enabled me to study the subject of my passion and encouraged me throughout the years of my study. I also want to acknowledge the support received from my friends and, in particular, my girlfriend. Martha, I really loved the time discussing the fundamentals of my research with you, even the complicated things.

I would like to thank all my colleges at SBA Research, which supported me from the time I joined. Specifically, I express my acknowledgement to Georg Merzdovnik for discussing and providing valuable feedback on the thesis and to Alexei Zamyatin for his backing throughout the time of the thesis and the spark for the collaboration with SBA Research.

Last but not least, I want to express my deep thank you to Nicolas Stifter and my co-advisor Aljosha Judmayer. Thank you Nicolas for supporting me during protocol design and development. I really enjoyed our discussions on all the topics involved. And thank you Aljosha for your outstanding assistance throughout the work on this thesis. Without the guidance of both of you this accomplishment would not have been possible. Thank you.

Kurzfassung

S. Nakamoto präsentierte 2008 eine Peer-to-Peer Version von elektronischem Geld: *Bitcoin*. Dieses System ermöglicht den direkten Zahlungsverkehr zwischen verschiedenen Personen und Organisationen, ohne Finanzdienstleister oder andere zu-vertrauende Dritte als Intermediäre einsetzen zu müssen. Im Zuge dessen entwickelte er die erste praktische Lösung für das Problem der Konsensfindung innerhalb eines dynamischen Netzwerks von potentiell anonymen Knoten, ohne die Notwendigkeit diese zuvor festzulegen. Dieses Ergebnis wird auf Basis des Konzepts von *Proof-of-Work* erzielt, das auf Grund der hohen Anforderungen für die benötigten Berechnungen zu einem enormen Energieverbrauch führt.

Unter Verwendung des alternativen Prinzips von *Proof-of-Stake* versuchen neue Protokolle Nakamoto's Ansatz weiterzuentwickeln. Eine grundlegende Voraussetzung für die Sicherheit dieser Protokolle ist eine vertrauenswürdige (d. h. *öffentlich-verifizierbare* und *manipulationssichere*) Quelle von Zufallszahlen. Deren Erzeugung stellt ein komplexes Problem dar, da diese in einem dezentralen Netzwerk unter dem potentiellen Einfluss von Angreifern durchgeführt wird. Kürzlich veröffentlichte Forschungsergebnisse und Projekte aus der Wirtschaft beschäftigen sich mit diesem Problem und stellen sogenannte *Random Beacon Protokolle* vor, welche die erforderlichen Zufallszahlen in regelmäßigen Intervallen generieren.

Diese Diplomarbeit beschäftigt sich intensiv mit den Herausforderungen der Entwicklung von Random Beacon Protokollen und liefert den ersten detaillierten Vergleich. Es wird gezeigt, dass *Publicly-Verfiable Secret Sharing* (PVSS) in vielen dieser Ansätze als gemeinsame Komponente dient. Weiters präsentiert diese Arbeit ein neu entwickeltes Protokoll, das ebenfalls PVSS verwendet und die Skalierbarkeit im Vergleich zu den bereits existierenden deutlich verbessert. Da dieser neue Ansatz nur eine PVSS-Instanz pro Runde benötigt, verringert sich der Kommunikationsaufwand von $\mathcal{O}(n^3)$ auf $\mathcal{O}(n^2)$. Diese Verbesserung wird erzielt, ohne auf wichtige Protokolleigenschaften, wie öffentliche Verifizierbarkeit, Manipulationssicherheit oder Nichtvorhersagbarkeit, verzichten zu müssen. Darüber hinaus erfolgt eine Optimierung der erarbeiteten Lösung durch die Entwicklung einer Protokollerweiterung, die die Interaktion zwischen den Knoten weiter reduziert und einen nahezu optimalen Kommunikationsaufwand von $\mathcal{O}(nc)$ erreicht. Dennoch stellt das erweiterte Protokoll mit sehr großer Wahrscheinlichkeit sicher, dass Zufallszahlen kontinuierlich erzeugt werden können und diese weder manipulierbar noch vorhersagbar sind.

Abstract

In 2008, S. Nakamoto presented *Bitcoin* as a peer-to-peer version of electronic cash – a system allowing direct payments between participants without the need for a financial institution or trusted third party. Thereby, he proposed the first practical solution for the problem of reaching consensus in a dynamic set of potentially anonymous participants without a prior agreement on this set. Bitcoin achieves this advancement at the cost of high computational requirements for *Proof-of-Work*, leading to vast amounts of electricity being consumed.

Recently, new protocols, using *Proof-of-Stake* as a fundamental principle, have tried to improve upon Nakamoto's solution. These protocols require a trustworthy source of randomness, i.e. *publicly-verifiable* and *bias-resistant* randomness, to maintain desirable security guarantees. However, obtaining trustworthy randomness, in a highly decentralized network and under potentially adversarial conditions, is by itself a challenging task. Recent academic research as well as projects from the industry try to address this problem by designing *random beacon protocols*, which produce the required random values in regular intervals.

In this thesis, we highlight the design challenges of random beacon protocols as well as provide the first in-depth review and comparison of state-of-the-art protocols. We identify *public-verifiable secret sharing* (PVSS) as a common building block and develop a new protocol using this cryptographic primitive. Our PVSS-based random beacon protocol greatly improves upon the scalability of existing approaches. Communication complexity is reduced from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$, as our solution only requires a single PVSS instance per round. Our protocol achieves this advancement while ensuring important protocol characteristics such as public-verifiability, bias-resistance and unpredictability. Additionally, we present an optimized solution as a protocol extension, which further reduces the interaction between network nodes and achieves a near optimal communication complexity of $\mathcal{O}(nc)$. This improvement is accomplished while still retaining liveness, bias-resistance and unpredictability of the random beacon values with very high probability.

Contents

Kurzfassung xi

Abstract xiii

Contents xv

1 Introduction 1
1.1 Proof-of-Work . . . 1
1.2 Proof-of-Stake . . . 2
1.3 Randomness . . . 2
1.4 Aim of the Work . . . 4
1.5 Structure . . . 4

2 Research Issues and Approach 5
2.1 Research Issues . . . 6
2.2 Methodology and Approach . . . 7

3 Background 9
3.1 Blockchain Fundamentals . . . 10
3.2 Bitcoin . . . 12
3.2.1 Proof-of-Work . . . 13
3.3 Ethereum . . . 15
3.4 Cryptographic Hash Functions . . . 16
3.4.1 Properties of Cryptographic Hash Functions . . . 17
3.4.2 The Random Oracle Model . . . 18
3.5 Digital Signature Schemes . . . 19
3.5.1 RSA Signatures . . . 19
3.6 Verifiable Random Functions (VRF) . . . 21
3.7 Secret Sharing . . . 23
3.7.1 Shamir's Secret Sharing . . . 24
3.7.2 Concrete example of Shamir's Secret Sharing . . . 26
3.8 Verifiable Secret Sharing . . . 27
3.9 Publicly-Verifiable Secret Sharing . . . 28
3.9.1 Schoenmakers' PVSS . . . 28

4 State-of-the-Art **33**
4.1 Basic Constructions . . . 34
4.1.1 Commitment Schemes . . . 34
4.1.2 Commitments Schemes with Security Deposits . . . 35
4.1.3 Proof-of-Work . . . 36
4.1.4 Iterated Proof-of-Work . . . 38
4.2 Dfinity . . . 39
4.2.1 Threat and Communication Model . . . 39
4.2.2 Unique Threshold Signatures . . . 39
4.2.3 Signature Uniqueness . . . 39
4.2.4 Signature Aggregation . . . 40
4.2.5 Dfinity's Random Beacon . . . 41
4.2.6 Distributed Key Generation . . . 41
4.2.7 Properties . . . 42
4.3 Algorand . . . 43
4.3.1 Threat and Communication Model . . . 43
4.3.2 Algorand's Random Beacon . . . 44
4.3.3 Verifier Selection . . . 44
4.3.4 Leader Selection . . . 45
4.3.5 Properties . . . 45
4.4 Ouroboros . . . 47
4.4.1 PVSS Fundamentals . . . 47
4.4.2 Threat and Communication Model . . . 47
4.4.3 Ouroboros' Random Beacon . . . 48
4.4.4 Properties . . . 49
4.5 RandShare, RandHound and RandHerd . . . 52
4.5.1 Threat and Communication Model . . . 52
4.5.2 RandShare . . . 52
4.5.3 RandHound . . . 53
4.5.4 RandHerd . . . 55
4.6 Scrape . . . 56
4.6.1 Share Distribution . . . 57
4.6.2 Share Verification . . . 57
4.6.3 Use Cases . . . 58

5 Hashchain-based Random Beacon **59**
5.1 Scenario and Threat-Model . . . 59
5.2 Hashchains . . . 59
5.3 Setup Phase . . . 62
5.4 Operation Phase . . . 62
5.4.1 Leader Selection . . . 63
5.4.2 Revealing Preimages . . . 63
5.4.3 Verification of Preimages . . . 63

5.4.4 Obtaining the Next Random Beacon Value . . . 64
5.5 Summary . . . 65
5.6 Evaluation . . . 66
5.6.1 Computational Complexity . . . 66
5.6.2 Communication Complexity . . . 66
5.6.3 Predictability . . . 66
5.6.4 Bias-Resistance . . . 69
5.6.5 Predictability under Active Manipulation . . . 72
5.6.6 Forks . . . 73
5.7 Seed Selection . . . 73

6 PVSS-based Random Beacon **75**
6.1 Threat and Communication Model . . . 75
6.2 Protocol Overview . . . 76
6.3 Setup Phase . . . 76
6.4 Operation Phase . . . 77
6.4.1 Leader Selection . . . 77
6.4.2 Random Beacon Blocks . . . 78
6.4.3 Generating Leader Blocks . . . 79
6.4.4 Generating Recovered Blocks . . . 80
6.4.5 Potential Leaders . . . 81
6.5 Verification . . . 82
6.5.1 Verification of Recovered Blocks . . . 83
6.5.2 Verification of Random Beacon Values . . . 83
6.5.3 Verification of PVSS Commitments . . . 84
6.6 Evaluation . . . 84
6.6.1 Computational Complexity . . . 84
6.6.2 Communication Complexity . . . 85
6.6.3 Availability / Liveness . . . 86
6.6.4 Public-Verifiablility . . . 86
6.6.5 Unpredictability . . . 86
6.6.6 Bias-Resistance . . . 87
6.6.7 Forks . . . 88

7 Protocol Extensions **89**
7.1 Quorum Share Distribution . . . 89
7.1.1 Construction Overview . . . 90
7.1.2 Quorum Selection . . . 90
7.1.3 Availability / Liveness . . . 91
7.1.4 Unpredictability . . . 93
7.1.5 Bias-Resistance . . . 95
7.1.6 Quorum Parameters . . . 95
7.1.7 Optimizing Quorum Parameters . . . 96
7.1.8 Communication Complexity . . . 99

7.2 Chained PVSS Commitments . . . 100
7.2.1 Motivation . . . 100
7.2.2 Setup . . . 101
7.2.3 Operation . . . 102
7.2.4 Verification . . . 102
7.2.5 Obtaining Random Beacon Values . . . 102

8 Discussion **105**
8.1 Comparison Overview . . . 105
8.2 Comparison of PVSS-based Random Beacon Protocols . . . 108
8.2.1 Existing PVSS-based Random Beacon Protocols . . . 108
8.2.2 Advantages of Existing PVSS-Beacon Protocols . . . 109
8.2.3 Disadvantages of Existing PVSS-Beacon Protocols . . . 109
8.2.4 Comparison to our PVSS-Beacon Protocol . . . 109
8.3 Our PVSS-based Random Beacon Protocol in Relation to other Existing Approaches . . . 112
8.4 Future work . . . 114
8.4.1 Communication Model . . . 114
8.4.2 Construction of a Full Fledged Distributed Ledger . . . 114
8.4.3 Integration into Existing Blockchains . . . 115

9 Conclusion **117**

Bibliography **119**

List of Figures **126**

List of Tables **128**

List of Algorithms **129**

A Appendix **131**
A.1 Full example of Shamir's Secret Sharing . . . 131
A.2 Python Implementation of Schoenmakers' PVSS . . . 133
A.3 Attack on the Use of a Fixed Initial Seed . . . 142

CHAPTER 1

Introduction

With the introduction of Bitcoin [1] in 2008, S. Nakamoto laid out the foundation for a decentralized peer-to-peer currency system. The underlying *blockchain* technology opened a wide range of possible applications far beyond the use as a currency [2]. As of August 2017, the website coinmmarketcap.com [3] lists over a thousand related projects.

Consensus algorithms are fundamental parts of Bitcoin and related technologies to synchronize the nodes in such decentralized networks. In traditional distributed systems, the consensus algorithm often only considers the failure of nodes (crash fault tolerance), whereas Bitcoin's consensus algorithm is able to withstand a much stronger failure model, where so-called byzantine nodes actively try to manipulate the network. Byzantine fault tolerant algorithms have previously become infamous for their limited scalability [4]. Nakamoto was first to address those issues by introducing a new consensus algorithm, which relies on the concept of *Proof-of-Work* — both revolutionary and heavily criticized.

1.1 Proof-of-Work

Proof-of-Work was initially considered as a protection mechanism against junk mail by Dwork and Naor in 1992 [5]. In the context of Bitcoin, Proof-of-Work depends on continuously performed computations in the decentralized network [1]. Network participants, called miners, solve computationally intensive cryptographic puzzles as part of transaction processing. It is a miner's task to verify transactions according to the protocol rules and group sets of unconfirmed transactions into blocks. These blocks include a reference to a previous block, forming a chain of blocks, which serves as a history of transactions – a *distributed ledger* or *blockchain.* As soon as a miner finds a solution to a puzzle, it gains the right to publish the corresponding block, and receives an economic reward for the work done. The validity of blocks, including the puzzle solution, can independently be verified by other participants in the network.

It is actually the chaining of Proofs-of-Work, which establishes an ordering of events, that together with the "longest chain rule"[1] leads to eventual agreement on the blockchain. Proof-of-Work in this case replaces the necessity for a prior agreement on the set of participants and, hence, allows for *dynamic* membership and also potential anonymity.

The protocol is constructed in a way that automatically regulates the difficulty of the puzzles. Considering the example of Bitcoin, this ensures that a new solution is found approximately every 10 minutes. As miners are economically rewarded for finding puzzles solutions, the increasing value of cryptocurrencies leads to a rise in mining activity. As a result, the involved parties invest vast amounts of money in special hardware equipment to solve the underlying cryptographic puzzles. Consequently, the difficulty of the puzzles and the electric power consumed increases to very high levels. As of August 2017, approximately $7 \cdot 10^{18}$ computations are performed every second for Bitcoin alone [6, 7]. In 2014, a study [8] already estimated the power consumption of Bitcoin mining matching the national energy requirements of countries like Ireland. Bitcoin's current estimated annual electricity consumption exceeded 16 TWh in August 2017 [9].

1.2 Proof-of-Stake

To address the issue of very high resource demands in Proof-of-Work based blockchains as well as other concerns such as scalability, latency, throughput and centralization risks [10], alternatives to Proof-of-Work are proposed. In this thesis, we focus on one of the most promising ones called *Proof-of-Stake*, and refer the reader to, for example, [11] for a more general discussion. Proof-of-Stake aims to establish similar security guarantees as Proof-of-Work, but in comparison uses only a negligible amount of computational resources.

As a key difference, Proof-of-Stake does not rely on solving cryptographic puzzles as part of its consensus algorithm. Instead of consuming electricity as a physical resource, virtual resources in form of digital coins are used. Leaders, which produce new blocks in Proof-of-Stake based systems, are then *randomly* selected based on the amount of coins they stake. Unfortunately, obtaining and agreeing on the involved randomness is a difficult problem by itself. In fact, Kiayias et al. identified leader election as a fundamental problem of Proof-of-Stake based protocols, as any introduced entropy is subject to potential manipulation by an adversary [12].

1.3 Randomness

This finding leads to the main topic of this thesis: *Generating Publicly-Verifiable and Bias-Resistant Randomness in Decentralized Systems.*[2] In 2016 and 2017, a substantial amount of research towards solving open problems in the area has been published. In

[1] The longest chain rules describes the fact that miners should always extend the longest chain of blocks, i.e. the chain containing the most Proof-of-Work.

[2] (verifiable) randomness for short

chapter 4, we present important proposals in the space including Ouroboros, Dfinity, Algorand, Scrape and RandHerd in detail. Following M. Rabin, who introduced the term *beacon* in [13], we refer to these approaches as *random beacon protocols*. Dfinity [14] builds a distributed verifiable random function as the key ingredient for reaching consensus among the network nodes. To the best of our knowledge, they are the first to utilize BLS signatures for that purpose. BLS signatures, introduced by D. Boneh, B. Lynn and H. Shacham, provide signature uniqueness as well as support for signature aggregation [15, 16]. Dfinity combines both of these key properties to obtain a random beacon protocol. Algorand [17], an alternative approached proposed by J. Chen and S. Micali, builds a decentralized ledger by combining (i) a randomness beacon based on unique signatures and hash functions with (ii) a new byzantine agreement protocol. Other protocols such as Ouroboros [12] and RandHerd [18] use a combination of publicly-verifiable secret sharing (PVSS) and other cryptographic primitives to obtain verifiable randomness in a decentralized environment. Similar to Dfinity and Algorand, the agreed randomness is then used as the basis for the respective consensus algorithm. Scrape [19] introduces an optimized variant of Schoenmakers' secret sharing protocol [20], an important building block for the Ouroboros and RandHerd protocols.

This research already establishes the key role of verifiable randomness as a fundamental building block for the construction of Proof-of-Stake based blockchains. The approaches aim to mitigate some of the most critical problems of Proof-of-Work based blockchains as well as the limited scalability of traditional byzantine fault tolerant protocol.

In addition to the scenario of leader selection and establishing consensus in Proof-of-Stake based distributed ledgers, publicly-verifiable randomness is useful in a variety of other scenarios. One prominent example is the provision of randomness in so-called *Smart Contracts*. Simply speaking, Smart Contracts are similar to normal computer programs, executed by the nodes in a decentralized network. Due to the deterministic nature of the execution environment, using randomness in Smart Contracts is currently not possible directly [21]. Instead, Smart Contracts, which require unpredictable, bias-resistant randomness, need to rely on (i) insecure sources of randomness such as the hash of block headers, which can be manipulated by miners, or (ii) trusted third parties such as the NIST random beacon service [21, 22]. Here, a decentralized publicly-verifiable random beacon could be more efficient and provide better security against manipulation than existing approaches.

Other use case scenarios include gambling and lottery services, publicly-auditable selections (e.g. soccer World cup draws) or the verifiable assignment of a limited number of resources (e.g. places in a kindergarten or the selection of one of two equally qualified job applicants). Syta et al. [18] list additional use cases for publicly-verifiable randomness including Tor hidden services, generation of elliptic curve parameters, byzantine consensus and electronic voting. Further use case scenarios and applications are outlined, for example, by Bonneau et al. in [23].

1.4 Aim of the Work

Generating public-verifiable randomness in decentralized systems is a difficult problem. Solutions have to solve complex design challenges in order to achieve desirable properties such as unpredictability, bias-resistance, verifiability and liveness and yet be scalable. While there are already proposed approaches to address open questions, very little research on comparison and evaluation of these new proposals has been performed. We identify additional desirable protocol characteristics and provide an overview of existing approaches. Further, we analyse and compare these protocols and show key advantages, drawbacks and open questions.

Based on the gathered insights obtained by this study, we present an alternative solution and possible protocol extensions in chapters 6 and 7. We focus on the properties of scalability and bias-resistance and make use of well-established cryptographic primitives, such as cryptographic hash functions, digital signatures and publicly-verifiable secret sharing, for our protocol. In addition, we introduce a simplified protocol to highlight the underlying concepts.

1.5 Structure

The remainder of the thesis is structured as follows:

Chapter 2 describes the research issues covered in this thesis as well as the methodological approach.

We further give background information on blockchain-based systems, using the examples of Bitcoin and Ethereum, discuss the concepts of Proof-of-Work and Proof-of-Stake, and introduce useful cryptographic primitives required for generating publicly-verifiable randomness in chapter 3.

Chapter 4 illustrates the difficulty of the problem by presenting basic approaches, which fail to provide a solution in the general case. Additionally, state-of-the-art proposals for obtaining randomness in decentralized systems are outlined and discussed in this chapter.

To highlight the main idea of our proposed random beacon protocol, we present a simplified construction based on the concept of hashchains in chapter 5.

In chapter 6, our main contribution – a new random beacon protocol based on publicly-verifiable secret sharing – is presented and evaluated.

We enhance our protocol by describing protocol extensions in chapter 7.

In chapter 8, our solution is discussed and compared to existing state-of-the-art protocols. We further outline open questions and give suggestions for future research.

Finally, we summarize and conclude this thesis in chapter 9.

CHAPTER 2

Research Issues and Approach

The generation of high quality randomness in a local and trusted environment is a well understood problem already considered by, for example, M. Blum and S. Micali in the 1980s [24]. In this case, many practical solutions exist. However, when considering decentralized systems, in particular in the context of blockchains, the problem of generating public-verifiable and bias-resistant randomness poses several challenging requirements.

To show the fundamental differences between a trusted local environment and a potentially untrusted, decentralized environment, we give the following illustrative example:

- *trusted / local environment*: Alice rolls a dice and the result shows a six. Neglecting the fact that physical dice have imperfections, Alice can be sure that the outcome she rolled was indeed selected uniformly at random from all possible outcomes.

- *untrusted / decentralized environment*: Alice is on the phone with a stranger. Alice rolls a die, the result shows a six and Alice tells the stranger that she rolled a six. How can the stranger ever be sure that Alice indeed rolled a six?

In the second case, the stranger could, of course, just trust Alice – accepting all the potential problems that might arise. The much more interesting case is however how to verify that she indeed rolled a six.

The problem, as well as a similar example using coin tossing, was already considered by M. Blum back in the 1980s [25]. Due to the rise of blockchain-based technology, the problem of generating verifiable randomness in decentralized systems was, however, recently been reconsidered in academic work [12, 17, 18, 19] as well as practical solutions [14].

2.1 Research Issues

The constructions for generating verifiable randomness in decentralized systems are often described as part of a larger system and address the common problem in quite different ways. The protocols rely on different threat and communication models, use other cryptographic primitives and / or achieve distinct properties. Therefore, the evaluation and comparison of existing protocols is a difficult task. In this thesis, we address the issue of systematizing the current progress on the problem of generating verifiable randomness in decentralized environments, thereby giving an in-depth comparison of existing proposals as well as developing new approaches towards an optimal solution.

In the ideal case, solutions manage to meet all desirable protocol characteristics listed below. To which extent all of theses properties can be achieved at the same time is still an open problem. Different desirable properties for random beacon protocols in decentralized environments have been identified by Bonneau et al. [23] and Syta et al. [18]:

(1) **Availability / Liveness:** Any single participant, or a colluding adversary (e.g. a set of malicious participants controlling up to 33% of the nodes in the network) should not be able to prevent progress.

(2) **Unpredictability:** Participants and attackers should not be able to predict or precompute future random beacon values.

(3) **Bias-Resistance:** Any single participant, or colluding adversary, should not be able to influence future random beacon values.

(4) **Public-Verifiability:** Third parties, i.e. participants which are not involved in producing the sequence of random numbers, should also be able to verify generated values. As soon as a new random beacon value becomes available, all parties can verify the correctness of the new value based on public information only.

As a result of our extensive study on the current state-of-the-art, we further suggest that practical solutions should achieve good

(5) **Scalability** in terms of the number of supported participants as well as

(6) **Efficiency** considering the number of messages, the amount of data transmitted and the required computational resources.

Additionally, the **security of underlying cryptographic primitives** and **implementation complexity** are important properties of random beacon protocols.

2.2 Methodology and Approach

In order to address the stated research issues, the following four part methodological approach is used:

- **Literature review:** This includes an extensive examination of the current state-of-the-art. To aid the comparison and discussion of potential problems, we give an overview of those protocols. An additional focus is placed on important underlying concepts, which are and can be used as building blocks for generating verifiable randomness in decentralized systems. These concepts include cryptographic hash functions, digital signatures and (publicly-verifiable) secret sharing.
- **Requirement analysis and evaluation of existing approaches:** Based on the insights gained during the literature review, key requirements as well as desirable properties for random beacon protocols are identified. Existing approaches are evaluated in respect to those requirements. We further examine to which extent the desired characteristics are met by those protocols.
- **Development of protocol improvements:** As the third part, we design protocol improvements. We use the insights gained during the literature review to address potential problems of current approaches. Further, we develop an improved solution for a decentralized random beacon protocol using a combination of existing and well-established cryptographic primitives.
- **Evaluation and comparison:** The proposed solution is evaluated, its key properties are identified and the design is compared to existing approaches. For the evaluation, desirable protocol properties as outlined in, for example, [18] and [23] as well as additional characteristics identified by the requirement analysis are considered.

CHAPTER 3

Background

To describe the context of this work, we provide an overview of blockchain-based technologies, as well as their key ideas and properties in section 3.1. We further outline the two most established blockchain-based systems, i.e. Bitcoin and Ethereum, in sections 3.2 and 3.3.

In addition, this chapter discusses cryptographic primitives, which are suitable as building blocks for random beacon protocols and are required for the understanding of state-of-the-art protocols. Section 3.4 describes cryptographic hash functions, one of the most used primitives in modern blockchains and random beacon protocols. We introduce the general principles and important properties of cryptographic hash functions as well as the Random Oracle Model – a mathematical abstraction of cryptographic hash functions.

We consider digital signatures schemes in section 3.5, introducing RSA as the most well known example. We explain and elaborate on the uniqueness property of digital signatures, which is useful for the construction of verifiable random functions (VRFs) in section 3.6. Further, unique signature schemes are required for certain random beacon protocols, e.g. for

- Dfinity [14] (see section 4.2) and
- Algorand [17] (see section 4.3).

In sections 3.7, 3.8 and 3.9, we discuss the topic of (publicly-verifiable) secret sharing. We first introduce Shamir's original secret sharing scheme and its properties and discuss why the additional properties, which are offered by verifiable secret sharing schemes (VSS) and publicly-verifiable secret sharing schemes (PVSS), are important for the construction of random beacon protocols.

Lastly, in section 3.9.1, we describe Schoenmakers' PVSS [20], which is used as a basis for the following PVSS-based random beacon protocols:

- Ouroboros [12] (see section 4.4)
- RandHound / RandHerd [18] (see section 4.5)
- Scrape [19] (see section 4.6)

Schoenmakers' PVSS also serves as a fundamental building block for our proposed random beacon protocol, described in chapter 6.

3.1 Blockchain Fundamentals

Since the introduction of Bitcoin in 2008, the popularity and general awareness of so-called cryptographic currencies steadily increased. As of August 2017, coinmarketcap.com lists more than a thousand different systems with a total market capitalization of around 161 billion USD. The largest ten of those account for around 90% of the total market capitalization [3]. The underlying technology for these cryptographic currencies is most commonly referred to as *blockchain* or *distributed ledger technology*.

Although the dominant use case for blockchains today is digital currency, the technology is not limited to this scope. Ethereum, one of the largest systems in use today, is a noticeable counterexample. It serves as an application platform based on the concept of Smart Contracts in extension to providing a payment system. In addition, M. Swan, for example, tries to demonstrate potential applications of blockchain-based technology in various segments including government, health, science, literacy, publishing, economic development, art and culture [26]. Other use case scenarios, such as public notary services, proof of existance, decentralized storage as well as applications in the music industry and field of IoT, are described by Crosby et al. [27].

What is a blockchain?
A blockchain is a term that is arguably associated with "things that are kind of like Bitcoin".

Or, more precisely, the term blockchain describes decentralized networks which store an ever growing list of records. These records, also called transactions, are grouped into blocks, which are then linked together to a data structure, which is ambiguously also called blockchain.

In the literature, various different definitions can be found. In the following, we give two descriptions identified by Judmayer et al. in [28]:

> A blockchain, according to the Princeton Definition [29], is defined as a linked list data structure, that uses hash sums over its elements as pointers to the respective elements.

> Colloquially the term *blockchain* refers to the category of distributed systems that are built using blockchain/cryptographic currency technologies, e.g., hash chains, asymmetric cryptography, game theory, etc.

While there exist a variety of different implementations for blockchains, they share various characteristics such as integrity, decentralization, the absence of trusted third parties and immutability.

Integrity A combination of cryptographic primitives serves as the core of blockchains and ensures the integrity of recorded datasets. Two important building blocks, namely digital signatures and cryptographic hash functions, are described in sections 3.4 and 3.5.

Cryptographic hash functions are used to establish the links between blocks. When a new block is append to the blockchain, the new block includes the hash of the previous block. This ensures that any manipulation of previously recorded data can be detected.

Digital signatures play an important role in proving the validity of each transaction inside a block. Considering a simple payment scenario where one party wants to transfer funds to another one, the first party has to authorize such a transaction by digitally signing it. When a transaction is added into a block, the corresponding signature(s)[1] is/are verified. The transaction is only considered valid if it contains valid signatures for the funds it wants to spend.

Decentralization Blockchains operate in a decentralized environment. There is no central point of failure or authority. Based on the concrete setting, two types can be identified:

(1) permissionless / public blockchains

(2) permissioned / private / consortium blockchains

In a public blockchain system, anyone is free to join the network as a participant. In this case, there is no requirement for registration. Major blockchain systems such as Bitcoin, Ethereum and many others fall into this category. In permissioned blockchains, such as Hyperledger Fabric, participants are required to register themselves. In the case of Hyperledger Fabric, they acquire identity credentials called enrollment certificates through certificate authorities [30]. In the permissioned setting, blockchain operators are in charge of defining access control rules for different types of participants.

[1] Depending on the number of inputs for a specific transaction, the inclusion of more than one valid signature is required.

Decentralized Systems vs. Distributed Systems To highlight the differences between decentralized and distributed systems we give the definitions of both terms, as stated by Troncoso et al. below [31]:

> **Distributed system:** A system with multiple components that have their behavior coordinated via message passing. These components are usually spatially separated and communicate using a network, and may be managed by a single root of trust or authority.
>
> **Decentralized system:** A distributed system in which multiple authorities control different components and no single authority is fully trusted by all others.

No trusted third party Traditional distributed databases share data among nodes in order to increase, for example, capacity, throughput, availability or latency. The nodes in such a system are often controlled by a single trusted entity like a cloud service provider. Blockchains, in contrast, do not impose such trust assumptions. Nodes are operated by different participants, who do not trust each other. Instead, nodes independently verify that the other nodes behave according to specified protocol rules and assume that a majority of participants act honestly.

Immutability In traditional databases, anyone with (physical) access to the data can change records. In contrast, blockchains ensure that after some data is recorded, no malicious participant can manipulate the stored data at a later point in time (given that the general assumptions required by the system model, for example honest majority of mining power, hold). This property is especially enforced by Proof-of-Work based blockchains. In section 3.2, we introduce how Bitcoin achieves this property.

3.2 Bitcoin

At its core, Bitcoin [1] is based on a peer-to-peer system, which allows Bitcoin clients to transfer cryptographic money / tokens (i.e. Bitcoins) between each other. All of these transfers, so-called transactions, are persisted in a distributed ledger called the blockchain. Each client can download the full history of transactions and independently verify that the ledger is indeed constructed according to the Bitcoin protocol rules.

A typical transaction includes source and destination addresses and is created, signed and sent to the Bitcoin network by the payer / sender. The requirement for transaction signing via the owners private key ensures that only the owner of the coins can spend them. The notion of a bank account applied to Bitcoin is somewhat misleading. There is no central authority calculating and updating a balance for its customers. Instead, the amount of Bitcoins an individual owns is directly inferred by the sum of tokens in unspent transactions for which the user controls the corresponding private keys.

Bitcoin needs to address the so-called *double-spending problem*. The problem of double-spending is a direct consequence of the ability to copy digital information. To illustrate it, we consider the following scenario: Alice constructs and signs a transaction to send one Bitcoin to Bob. In addition, Alice also creates a transaction to send the same Bitcoin to Carol. Both transactions get published and Alice was able to spend her Bitcoin twice. When using a centralized system, the attempt to spend the same Bitcoins twice can be detected at the time the second transaction is transmitted to the central service. A possible countermeasure is to simply ignore the second transaction. However, in a highly decentralized system like Bitcoin, there is no synchronized time between the network nodes. As a consequence, nodes might disagree on which transaction happened first and thus should be processed. This is true even though each individual node can decide on a set of valid transactions. Finding a mechanism for reaching consensus, i.e. which transactions should be included in which order, in the entire distributed system is the difficult part. The first large scale solution using the concept of *Proof-of-Work* was famously introduced by S. Nakamoto in the original Bitcoin paper [1].

3.2.1 Proof-of-Work

In order to illustrate how the described consensus problem [32] is resolved in Bitcoin, a more detailed look into the process of adding transaction to the blockchain is required. This process is called mining. Bitcoin nodes (i.e. miners) collect transactions, verify them (e.g. they check that the referenced tokens are not already spent in their view of the blockchain and that the given signatures are valid) and add them to a pool of unconfirmed transactions. The unconfirmed transactions are combined with additional information, such as the hash of the previous block header and a nonce value, to form the next block of the blockchain.

Each block includes the hash of the predecessor block header, which effectively creates a linked chain. As a consequence, a single change in past block would automatically require a change in all following blocks in order for them to stay valid.

As a further protocol requirement, the hash value, interpreted as 256 bit number, of a proposed block needs to be less than a specific target value. The difficulty in finding such low hash values is a result of the security properties of the used hash function SHA256, which is considered secure [33]. Finding such a hash value is not feasible in an efficient manner. All a miner can do to find a low enough hash value is to perform a brute force search by varying the block's data, e.g. the nonce field. Depending on the used nonce, the resulting hash value changes in an unpredictable way. Eventually, a chosen nonce produces a low enough hash for a block to be published and accepted by the Bitcoin network.

The following example further illustrates the difficulty of finding such hash values. We consider the actual hash of block header #482625 mined on Aug. 30, 2017:

```
000000000000000000d61c4f32930ada2259b55679078fb65f8692e1c7e0c8f0
```

Finding such a hash value takes all the miners in the Bitcoin network together 10 minutes on average. The target value (difficulty) is adjusted periodically to adapt to the generally increasing computing power of the Bitcoin mining network. As of Aug 2017, the combined mining power of the Bitcoin network is about $7 \cdot 10^{18}$ hashes/second (7,000,000 TH/s) [6, 7]. While previously mining on CPUs and GPUs could be profitable, nowadays specialised mining hardware, so-called ASICs, are used.

While it is difficult to find a nonce producing such a small hash value, once one is found, it can be easily verified that the published block indeed meets the difficulty target. Furthermore, this verification provides evidence that the miner of the block actually used a high amount of computational resources in order to find it. The hash serves as a *Proof-of-Work*.

Bitcoin addresses the distributed consensus problem in a new, innovative manner. By using Proof-of-Work, Bitcoin works in a byzantine setting with a dynamic set of potentially anonymous participants. Each node can independently and without reliance on trust of a central authority verify all transactions and issued Bitcoins. In fact, *"much of the trust in Bitcoin comes from the fact that it requires no trust at all"* [34].

Establishing consensus via Proof-of-Work In the following, we describe how Bitcoin's Proof-of-Work approach can be used to establish consensus across the whole Bitcoin network. Miners individually gather transactions, group them together into blocks and start searching for a nonce value in order to match the current difficulty target of the network. It is important to note that blocks created by two miners are most likely different as they include a different set of transactions, or the transaction ordering does not match. But according to the Bitcoin protocol, only one block can become the next block of the blockchain. Each miner has an incentive that it mines this block, because a reward (currently 12.5 BTC) and transaction fees are paid to the creator of this block. Consequently, the miners are in a race against each other. Each one is trying to find a low enough hash value. As soon as a miner finds such a hash, it publishes the block as fast as possible. Other miners receive the new block, verify it and the race starts again with the new block added to the end of the blockchain. Conflicts, i.e. two miners finding blocks at approximately the same time, are eventually resolved by the *longest chain rule*. This means that the branch with the most Proof-of-Work in its history is considered to be the correct one. While it is temporally possible that there is more than one branch (called a fork), as soon as the next blocks are found a single longest chain forms and honest nodes will switch to this chain. This can be considered as a majority vote by computational power.

3.3 Ethereum

Ethereum [35, 36] is another major implementation of a permissonless blockchain. As of August 2017, it is the second largest in terms of market capitalization after Bitcoin. Ethereum shares many concepts with its counterpart Bitcoin, for example, both record a history of transaction by producing and linking blocks, both operate in the open / permissionless setting and both use Proof-of-Work as part of their consensus algorithm. Ethereum allows for peer-to-peer payments without a trusted third party just like Bitcoin.

Despite the similarities, there are also various aspects, which distinguish both blockchains. While Bitcoin with its limited scripting ability has set its focus on financial transactions, Ethereum provides a platform which allows anyone to build their own applications. In this sense, Ethereum is a more general purpose blockchain. Here, *more general purpose* does not necessarily mean better. There are complex trade-offs involved. For example, the increased flexibility is directly associated with a higher complexity and a larger attack surface.

Ether The cryptographic currency of the Ethereum platform is called Ether. In the simplest case, Ether can be sent from one account[2] in the Ethereum network to another account, very similar to the process, in which Bitcoins are transferred between Bitcoin addresses. Just like Bitcoins can be divided into smaller units (i.e. mBTC, Satoshi), Ether can also be divided into smaller parts called Wei – 1 Ether being equal to 1,000,000,000,000,000,000 Wei [37]. This divisibility enables payments of very small amounts such as the payment of computational resources in Ethereum itself.

Smart Contracts & the Ethereum Virtual Machine In Bitcoin, users can only interact with the system in a very limited way. They can submit transaction, which will eventually be processed and validated by miners according the consensus rules. A Bitcoin transaction, at a technical level, is actually a program in a non-turing complete, stack-based scripting language. Although there a variety of possible operations in Bitcoin's scripting language there are a number of limitations, i.e.:

(1) No loops

(2) No constructs for recursion

(3) No state

Ethereum addresses this shortcomings by introducing a more powerful execution environment: the Ethereum Virtual Machine (EVM). The EVM is used to process so-called *Smart Contracts* in the Ethereum network. Smart Contracts can be viewed as computer programs running on the Ethereum platform. Such programs are typically expressed in a higher level programming language such as Solidity. A noticeable difference to Bitcoin

[2] Here we refer to simple account, i.e. account without attached contract code.

is that the allowed programs form a turning complete language, i.e. allow arbitrary computations, in particular loops, calls to other subroutines as well as data store and read operations. The EVM is run by each node in the Ethereum network to verify and process transactions. Therefore, each node has to process all the instructions of the Ethereum blockchain.

Deterministic Execution The EVM has to process all Smart Contracts in a deterministic manner, because otherwise different network nodes cannot easily agree on a single execution. This is a restriction in case non-manipulable and unpredictable randomness is required for the execution of particular Smart Contracts. *Random beacon protocols* as described in chapters 4 and 6 might be used to obtain the required randomness in a verifiable manner, which does not contradict the deterministic execution environment of the EVM.

3.4 Cryptographic Hash Functions

Cryptographic hash functions are important primitives in the field of modern cryptography [38]. They have been shown to be very useful to solve various security related challenges in the field of telecommunication and computer networks [39]. A major application of cryptographic hash functions is public key cryptography. Here, in a first step, they are used to condense the input to a fixed size. The result is then used to construct a digital signature to ensure authenticity of the whole message [40].

More recently, cryptographic hash functions are used as fundamental building block of cryptographic currencies. In [1], S. Nakamoto describes Proof-of-Work, which uses cryptographic hash functions, as part of a consensus mechanism for decentralized systems

In this thesis, we focus on *cryptographic* hash functions. Hence, when we omit the term *cryptographic* and just use the term *hash function*, we always refer to *cryptographic hash functions*.

A hash function H takes some finite input bitstring M and efficiently produces an output value $H(M)$. The output $H(M)$ is called the hash of M. Since H operates deterministically and without side effects, there is exactly one unique hash per message. Typical lengths for the hash are, for example, 160, 256 or 512 bits. The length of the output only depends on the specific algorithm chosen, but not on the length of the input message M.

3.4.1 Properties of Cryptographic Hash Functions

Consider the following two examples of the hash function *SHA3-256*:

```
SHA3-256("The quick brown fox jumps over the lazy dog")
 => 69070dda01975c8c120c3aada1b282394e7f032fa9cf32f4cb2259a0897dfc04

SHA3-256("The quick brown fox jumps over the lazy dog.")
 => a80f839cd4f83f6c3dafc87feae470045e4eb0d366397d5c6ce34ba1739f734d
```

This examples demonstrates the fundamental properties of a secure hash function. Intuitively, the output of a hash function (here given as a hexadecimal value) appears to be random and unrelated to the input. Small modifications on the input (e.g. adding a dot at the end of the message in the example) change the hash drastically. In fact each bit of the output gets flipped with a probability of approximately 50% [41].

More formally, we consider the following properties of a hash function H on input x, x' and outputs y, y' as given in [38][3]:

(1) *preimage resistance* — for essentially all pre-specified outputs, it is computationally infeasible to find any input which hashes to that output, i.e., to find any preimage x' such that $H(x') = y$ when given any y for which a corresponding input is not known.

(2) *2nd-preimage resistance* — it is computationally infeasible to find any second input which has the same output as any specified input, i.e., given x, to find a 2nd-preimage $x' \neq x$ such that $H(x) = H(x')$.

(3) *collision resistance* — it is computationally infeasible to find any two distinct inputs x, x', which hash to the same output, i.e., such that $H(x) = H(x')$. (Note that here there is free choice of both inputs.)

A hash function with collision resistance always satisfies the property of 2nd-preimage resistance as well. Preimage resistance is not guaranteed in this case. However, in practice this implication is almost always true [38].

Collision resistance in practice By definition, the input of a hash function is of arbitrary (finite) length and the output is of fixed length, e.g. 256 bits. Therfore, there must exist collisions for such hash functions, i.e. there are indeed messages M and M' such that $M \neq M'$ but $H(M) = H(M')$. However, finding such values is computationally infeasible and, hence, extremely unlikely for secure hash functions.

Due to the so-called birthday attack, a collision on an ideal hash function with 256 bit output length can be found on average using about 2^{128} hash invocations. In comparison,

[3] Here, we changed the original notation of the hash function $h(\cdot)$ to $H(\cdot)$ for consistency.

the arguably largest effort for calculating SHA256 hashes is mining in the cryptocurrency Bitcoin. In April 2017, the estimated number of hashes calculated every second reached around $7 \cdot 10^{18} \approx 1.5 \cdot 2^{62}$ [6]. Assuming this computation power of the Bitcoin network as a constant, the expected time for finding a collision with this amount of computational resources would still be approximately $2.2 \cdot 10^{11}$ years (220,000 million years).

3.4.2 The Random Oracle Model

When describing protocols which internally use cryptographic hash functions, their security is often evaluated in the so-called Random Oracle Model (ROM). It was introduced by Bellare and Rogaway in 1995 and serves as a *"bridge between cryptographic theory and cryptographic practice"* as they stated in [42].

Some criticism, e.g. the existence of provably secure signatures in the ROM, which are insecure in practice, was stated by, for example, Canetti et al. in [43]. However, Koblitz and Menezes in [44] argue *"that there is no evidence that the need for the random oracle assumption in a proof indicates the presence of a real world security weakness in the corresponding protocol"*. The discussed state-of-the-art protocols as well as our constructions rely on the ROM abstraction and the security properties of their realizations (e.g. SHA-3).

Hash functions as Random Oracles We refer to [42] for a formal definition. Intuitively, in the ROM, hash functions are modelled as random oracles. This means that we do not consider the actual implementation of the hash function, but rather use an abstraction. Querying a random oracle with a message M is the abstraction for evaluating a hash function on the message M.

The abstraction works as follows: The random oracle initially stores an empty set $\mathcal{D}$ of tuples (a, b). When queried with a message M, it checks if $(M, x) \in \mathcal{D}$ for some element x. Based on the result of the check, there are two options:

(1) $\exists x, (M, x) \in \mathcal{D}$: In this case, x is returned as response to the query.

(2) $\neg\exists x, (M, x) \in \mathcal{D}$: Otherwise, some value y is randomly selected. Then, the tuple (M, y) is added to the set $\mathcal{D}$ and y is returned as response to the query.

Notice that the construction above satisfies all properties of secure hash functions as given in the previous section. Therefore, this abstraction can be used in protocols, which require secure hash functions. To obtain a practical implementation, the random oracle is then replaced by a cryptographically secure hash function. Currently considered suitable candidates are SHA-3 variants such as SHA3-256, for example recommended by NIST [45].

3.5 Digital Signature Schemes

Digital signatures are primitives in the field of public key cryptography, or asymmetric cryptography. Consider the case when two parties, Alice and Bob want to communicate with each other by exchanging messages. A digital signature $\sigma = sign_{sk_A}(M)$ of a message M, sent from Alice to Bob, then ensures that Alice was indeed the sender of M. Only Alice can produce a valid signature since the signature is dependent on sk_A, the secret/private key only known to Alice. After sending the message M and the signature σ to Bob, Bob can plausibly convince a third party of the fact that Alice did sign M, whereas Alice cannot deny the fact that she signed the message. The digital signature also ensures that modifications of the message M can be detected at the time of signature verification.

A digital signature scheme consists of three efficient (polynomial-time) algorithms [46]:

(1) Signing algorithm $sign_{sk}(M)$: Given a message M and a secret key sk, this algorithm produces a digital signature. The signing algorithm might use additional randomness, thus the algorithm might be non-deterministic.

(2) Verification algorithm $verify_{pk}(\sigma, M)$: Given a candidate digital signature σ of a message M and the public key pk of the message signer, this algorithm checks whether σ is a valid signature of M, i.e. it checks if $\sigma = sign_{sk}(M)$ holds.

(3) Key generation algorithm $keygen()$: This probabilistic algorithm creates a secret/public keypair (sk, pk). The private/secret key sk is used for signing and is kept private. The public key pk is required for verification typically distributed via a public key infrastructure (PKI).

3.5.1 RSA Signatures

In the following, we illustrate the RSA digital signature scheme. The original scheme was described by Rivest, Shamir and Adleman in [47].

Key generation algorithm: $keygen()$
Choose two large prime numbers p and q.
Compute $n = p \cdot q$.
Compute $\phi(n) = (p-1) \cdot (q-1)$.
Pick an integer $1 < e < \phi(n)$ such that e and $\phi(n)$ are relatively prime.[4]
Compute $d = e^{-1} \pmod{\phi(n)}$ as the multiplicative inverse of e.[5]
The public key pk and private/secret sk are then given by: $pk = (n, e)$ and $sk = (n, d)$.

[4] $gcd(e, \phi(n)) = 1$ must hold – i.e. greatest number dividing both e and $\phi(n)$ is required to be 1.
[5] I.e. compute d such that $d \cdot e \equiv 1 \pmod{\phi(n)}$ holds using the extended euclidean algorithm.

Signing algorithm: $sign_{sk}(M)$
Using the private key $sk = (n, d)$, the signature σ is obtained by computing $\sigma \equiv M^d \pmod{n}$. To allow messages of arbitrary length, full domain hashing using a hash function H with output range $\{0, 1, ..., n-1\}$ might be used. In this case, σ is given by $\sigma \equiv H(M)^d \pmod{n}$.

Verification algorithm: $verify_{pk}(\sigma, M)$
Using the public key $pk = (n, e)$, the verifier computes $M' \equiv \sigma^e \pmod{n}$.
Then, the verification algorithm returns *VALID* if, and only if, the condition $M = M'$ holds (or $H(M) = H(M')$ in the case of arbitrary length messages).

Security considerations The security of this scheme is based on the (unproven but widely studied) assumption that factoring the RSA modulus n from the public key into its prime factors p and q is intractable for large primes p and q. In [48], NIST recommends at least 2048 bits for the size of the RSA modulus n.[6]

We stress that this illustration is not aimed to provide a guideline for building a secure system in practice. For additional information on secure RSA based signatures, the reader might consider [49] and [50].

Uniqueness of RSA signatures As we further elaborate in section 3.6, unique signature schemes are important building blocks for verifiable random functions (VRFs). Intuitively, uniqueness here means that the verification algorithm only excepts a single signature per message. A formal definition is given in [51]. Notice that RSA as described above produces signatures deterministically. There is however no guaranty for uniqueness. Consider the following counterexample: $n = 13031$, $e = 3$, $M = 360$, $\sigma_1 = 1977$, $\sigma_2 = 3471$, $\sigma_3 = 8368$. A verifier knowing the public key (n, e) verifies the signatures by checking the following conditions:

$$360 \equiv 1977^3 \pmod{13031}$$
$$360 \equiv 3471^3 \pmod{13031}$$
$$360 \equiv 8368^3 \pmod{13031}$$

Notice that all three conditions hold. Therefore, the counter example show, that there is indeed a single message with different valid signatures, which contradicts the definition of uniqueness.

The problem in this case is that e and $\phi(n)$ are not relatively prime. This property is required by the key generation algorithm of RSA. However, a verifier cannot check this condition and, therefore, cannot be sure that the public key was indeed constructed correctly. For this purpose, a verifier would need to know that $\phi(n) = 12792$. This requires a factorization of $n = 13031 = 157 \cdot 83$. While this calculation is easy for the

[6] The website https://www.keylength.com provides a comparison of various key size recommendations by different parties.

small numbers in the example, it is infeasible in practice as large values of p and q are used. The reason for this is that there are no known efficient algorithms for integer factorization (on classical computers).

A solution to this problem is that the verifier requires e to be a prime number with $e > n$. Choosing e in this way ensures that e and $\phi(n)$ are relative prime [51].

Applications Upon the use case of ensuring authentication, non-repudiation and integrity of messages, digital signatures are useful in a variety of applications. Secure internet communication via TLS/SSL uses digital signatures to ensure the validity of TLS/SSL certificates [52]. Bitcoin uses digital signatures to ensure that funds can only be spent by their rightful owners [1]. More recently, digital signatures have also been used as a building block to generate randomness in Dfinity and Algorand:

- Dfinity [14] builds a verifiable random function (VRF, see section 3.6) based on Boneh-Lynn-Shacham (BLS) signatures [15]. The described protocol allows a group of participants to agree on a random number, which is then used as a common coin in an asynchronous BFT consensus protocol [14].
- Chen et al. [17] describes the Algorand protocol (short for algorithmic randomness), which heavily uses digital signatures and cryptographic hash functions to generate verifiable randomness and use it to perform leader and verifier selection. They require a digital signature scheme with unique signatures in the protocol [17] but do not specify which signature scheme to use. As we discuss in more detail in section 3.6, signature schemes typically do not have this property. BLS signatures are one noticeable exception [15].

3.6 Verifiable Random Functions (VRF)

In this section, we describe verifiable random functions (VRF). Intuitively, the idea behind a VRF is that Alice can ask Bob to compute a function f_s on some input x. Only B is able to compute f_s as its result is dependent on some secret value s, which only Bob knows. The result $v = f_s(x)$ has the property of being unique and computationally indistinguishable from a truly random string v' of equal length. Alice wants to be sure that Bob indeed provided the unique correct result of the computation [53]. VRFs are introduced by Micali, Rabin and Vadhan in [53] as a natural extension of pseudo random functions (PRFs).

Pseudo Random Functions (PRFs) Micali et al. define a PRF $f_s(x)$ as an efficiently computable function from input bitstrings of length a to output bitstrings of length b. The value s denotes a seed. The fundamental property of the PRF is that an observer cannot distinguish the outputs of the PRF from truly random strings of the same length. This property holds based on the following assumptions [53]:

(1) The observer does not know the seed s.

(2) The observer is computationally bounded.

These assumptions do not imply that the description of the PRF has to be kept secret. Keeping the seed s private is sufficient. In practice, PRFs are constructed using cryptographic primitives such as block ciphers, or hash functions.

Verifiable Random Functions (VRFs) VRFs address the issue of unverifiability of PRFs. Consider the case where a party computing $f_s(x_1), f_s(x_2), ..., f_s(x_n)$ claims the corresponding outputs are $o_1, o_2, ..., o_n$. Without knowledge of s, an observer (by definition) cannot verify that applying f_s to x_i indeed yields the corresponding output o_i. As soon as s gets published, future output values are not indistinguishable from truly random strings anymore. They get fully predictable and can be efficiently computed by any party.

To obtain verifiability without compromising the unpredictability property of future outputs, a party knowing the seed s publishes $v = f_s(x)$ together with a proof $proof_x$. This proof allows verification of the fact that $v = f_s(x)$ indeed holds without revealing s. It is crucial that a party knowing s can only construct a valid proof for a unique v for every x [53]. For the proof itself, there is no uniqueness requirement.

Early solutions proposed interactive zero knowledge proofs [53]. However, Micali et al. in [53] and other authors described solutions [51, 54, 55], which do not require interaction for the verification process.

VRFs via Digital Signatures To construct a VRF with non-interactive proofs, one might consider to use digital signatures (see section 3.5) as an underlying primitive. Directly defining $f_s(x) = sign_{sk}(M)$ based on the signing function $sign_{sk}(M)$ of a digital signature scheme is insufficient in general. The core property needed to construct a VRF from a digital signature scheme is uniqueness of the signatures. The corresponding verification algorithm must only accept a single output value for each input. Unfortunately, these properties typically do not hold. Popular signatures schemes such as DSA, ECDSA [56] or RSA-PSS [49] operate non-deterministically and can, therefore, produce multiple different signatures for a single message M. Even when using a specification for a deterministic signing operation $sign_{sk}(M)$, for example as given in [57], this does not result in a VRF. The signer can still construct multiple valid signatures by not following the scheme's deterministic approach for generating certain parameters (for example by picking a non-random nonce value).

A signature scheme satisfying the uniqueness property can been seen as verifiable unpredictable function (VUF) [53]. Micali et al. further proof a way to convert a VUF to a VRF. Full VRF schemes, based on the following different number theoretic assumptions, have been proposed:

(1) RSA Hardness Assumption [53]

(2) (Decisional) Diffie Hellman Separation [51]

(3) Decisional Bilinear Diffie-Hellman Inversion [54]

(4) Decision Linear Assumption [55]

We do not provide additional details on these schemes as our construction in chapter 6 uses a different approach based on publicly-verifiable secret sharing (see section 3.9). In practical protocols such as Dfinity [58] (see section 4.2) and Algorand [17] (see section 4.3), VRFs are constructed by a combination of cryptographic primitives such as unique digital signatures and cryptographic hash functions.

3.7 Secret Sharing

Secret Sharing was independently introduced by Shamir [59] and Blakley [60] in 1979. The main idea of secret sharing is to distribute a secret S among a certain number of participants. Each one of those n participants receives a part of the secret, called a share. Shares can later be combined by collaborating participants to reconstruct the original secret. The number of collaborating participants needed for recovering the secret successfully is referred to as t. We consistently denote this scenario by a (t, n) secret sharing scheme. Any group of t (or more) out of n participants can recover S from their shares.

The secret S is typically some highly sensitive digital information, for example an encryption key. However, we use a 6-digit code vault combination as our secret for an illustrative example. Consider the following scenario for a $(3, 3)$ secret sharing scheme: A father wants to share the combination $S = 913821$ among his three children. The children should be able to open the vault together in case the father dies. For this purpose, the father tells each children a different share from the set $\{91xxxx, xx38xx, xxxx21\}$. We now analyse this example using the basic goals / criteria for a secret sharing scheme given by Shamir [59]:

(1) Knowledge of at least t shares makes S easily computable.

(2) Knowledge of fewer than t shares leaves S completely undetermined, i.e. all possible values for S are equally likely.

Clearly the children are able to recover the vault combination by concatenating their digits. Thus, criteria (1) is met. Criteria (2) is violated as each share leaks two digits of information about the secret S. Intuitively, a child with, for example, the share $91xxxx$ knows that S must start with 91, hence the first digits are determined. As a consequence, the search space is reduced.

For the case, where $t = n$ holds, a secure secret sharing scheme which fulfills both criteria can be easily obtained. Consider that the father instead calculates the shares $\{S_i \mid 1 \leq i \leq n\}$ using the following method:

(1) $S_i \in_R [0, 999999] \quad 1 \leq i \leq n-1$

(2) $S_n = (S - \sum_{i=1}^{n-1} S_i) \pmod{1000000}$

Here, $\in_R U$ denotes a randomly chosen element from a set U. This intuitively means that the father randomly selects shares from $[0, 999999]$ such that their sum is equal to $S \pmod{1000000}$. An example for the shares is the set $\{811591, 527645, 574585\}$. Criteria (1) is met as the reconstruction of S can easily be obtained by summing the shares:

$$S \equiv 811591 + 527645 + 574585 \pmod{1000000}$$
$$S = 913821$$

In contrast to the previous method, this one does not leak any information about S as long as less than three children collaborate. Thus criteria (2) is also met.

The given example can be generalized for any $n \geqslant 1$ resulting in a simple and efficient secret sharing scheme for the special case $t = n$. Addition and subtraction might be replaced by logical-exclusive-or.

By distributing the original secret S to all participants, one trivially obtains a $(1, n)$ secret sharing scheme. For a solution, which also works in the general case $1 \leq t \leq n$, Shamir's original secret sharing scheme is introduced in the following section.

3.7.1 Shamir's Secret Sharing

Shamir's secret sharing protocol [59] is based on *polynomial interpolation*. The key idea behind this scheme is the fact that given t points (x_1, y_1), (x_2, y_2), ..., (x_t, y_t) with different x-coordinates, there is a unique polynomial $p(x)$ of degree $t-1$ going through all of the points.

To perform (t, n) secret sharing, the dealer (i.e. the party who wants to share a secret S) constructs a polynomial of degree $t-1$ [59]:

$$p(x) = \alpha_0 + \alpha_1 x + ...\alpha_{t-1} x^{t-1}$$

The coefficients $\alpha_i \mid 0 \leq i \leq t-1$ are selected uniformly at random from $\mathbb{Z}_p = \{0, 1, 2, ..., p-1\}$. Here p is a prime number selected such that $p > S$ and $p > n$ holds.[7] The coefficient α_0 is defined as S, i.e. $\alpha_0 = S$. By construction, $\alpha_0 = S = p(0)$ holds.

[7] Note that without loss of generality the secret S is a number or can be encoded as a number [59].

The dealer then computes a share for each participant by evaluating the polynomial $p(x)$ for different x-values.

$$S_i = p(i) \quad 1 \leq i \leq n$$

The evaluation is performed using modular arithmetic over the finite field $\mathbb{Z}_p$. Thus, the dealer and participants need to agree on the prime number p as protocol parameter. Notice that each pair (i, S_i) represents a point on the polynomial.

The reconstruction of the secret S is then accomplished by Lagrange interpolation using t different shares. Without loss of generality, let $i_1, i_2, ..., i_t$ denote the indices of the shares used for reconstruction. Then, reconstruction of S can be accomplished by calculating [61]:

$$S = \sum_{j=1}^{t} S_{i_j} \prod_{\substack{k=1 \\ j \neq k}}^{t} i_k \cdot (i_k - i_j)^{-1}$$

All calculation has to be done using modular arithmetic over p. Thus, x^{-1} denotes the multiplicative inverse of x, i.e. $x \cdot x^{-1} \equiv 1 \pmod{p}$. Since p is prime and $(i_k - i_j)$ is always non-zero, this inverse always exists. This calculation can be done in polynomial time, therefore criteria (1) for secret sharing schemes is met.

Regarding criteria (2), we consider a coalition of $t-1$ participants and corresponding share indices $i_1, i_2, ..., i_{t-1}$. Such a coalition should not be able to obtain (partial) information about the secret. This is the case for Shamir's protocol, it is *information theoretically secure* [62]. For each of the candidate secrets $S' \in \mathbb{Z}_p$, exactly one polynomial $p'(x)$, which satisfies the conditions $p'(0) = S'$ and $p'(i_j) = S_{i_j} \ \forall j \mid 1 \leq j \leq t-1$, can be found. The resulting candidate polynomials are all equally likely and, thus, do not leak information about S [59]. This statement in shown more formally in [61].

3.7.2 Concrete example of Shamir's Secret Sharing

We now provide an example for a $(3, 5)$ Shamir's secret sharing protocol. The full example, including all computation steps, is given in appendix A.1.

Let $S = \alpha_0 = 10$.

Select prime $p = 53$, thus $\mathbb{Z}_p = \mathbb{Z}_{53} = \{0, 1, 2, ..., 52\}$.

Choose coefficients $\alpha_1 = 17$ and $\alpha_2 = 44$.

This yields the polynomial $p(x) = 10 + 17x + 44x^2$.

Calculate the shares:

$$\begin{aligned}
S_1 &\equiv p(1) && \pmod{53} \\
S_1 &\equiv 10 + 17 \cdot 1 + 44 \cdot 1^2 && \pmod{53} \\
S_1 &\equiv 10 + 17 + 44 && \pmod{53} \\
S_1 &\equiv 71 && \pmod{53} \\
S_1 &= 18
\end{aligned}$$

Similarly we get $S_2 = 8$, $S_3 = 33$, $S_4 = 40$ and $S_5 = 29$.

We select a set of t shares for reconstruction: $\{S_1, S_4, S_5\}$

The indices for the shares are then given by $i_1 = 1$, $i_2 = 4$ and $i_3 = 5$.

The reconstruction can be computed as follows:

$$\begin{aligned}
S &\equiv \sum_{j=1}^{t} S_{i_j} \prod_{\substack{k=1 \\ j \neq k}}^{t} i_k \cdot (i_k - i_j)^{-1} && \pmod{53} \\
S &\equiv S_{i_1} \prod_{\substack{k=1 \\ 1 \neq k}}^{3} i_k \cdot (i_k - i_1)^{-1} + S_{i_2} \prod_{\substack{k=1 \\ 2 \neq k}}^{3} i_k \cdot (i_k - i_2)^{-1} + S_{i_3} \cdot \prod_{\substack{k=1 \\ 3 \neq k}}^{3} i_k \cdot (i_k - i_3)^{-1} && \pmod{53} \\
&\dots \\
S &\equiv S_{i_1} \prod_{\substack{k=1 \\ 1 \neq k}}^{3} i_k \cdot (i_k - i_1)^{-1} + 4 + 29 && \pmod{53} \\
S &\equiv S_{i_1} \cdot \big((i_2 \cdot (i_2 - i_1)^{-1}) \cdot (i_3 \cdot (i_3 - i_1)^{-1})\big) + 4 + 29 && \pmod{53} \\
S &\equiv S_1 \cdot \big((4 \cdot (4 - 1)^{-1}) \cdot (5 \cdot (5 - 1)^{-1})\big) + 4 + 29 && \pmod{53} \\
S &\equiv 18 \cdot \big((4 \cdot (3)^{-1}) \cdot (5 \cdot (4)^{-1})\big) + 4 + 29 && \pmod{53} \\
S &\equiv 18 \cdot \big((4 \cdot 18) \cdot (5 \cdot 40)\big) + 4 + 29 && \pmod{53} \\
S &\equiv 259200 + 4 + 29 && \pmod{53} \\
S &\equiv 30 + 4 + 29 \\
S &= 10
\end{aligned}$$

3.8 Verifiable Secret Sharing

Shamir's secret sharing protocol as described in section 3.7.1 relies on the following crucial assumption: The participants assume that they are given correct shares. This is reasonable for the example given (e.g. a father wants to share a secret to his children). However, this limits the ability to apply this scheme in e.g. fault tolerant (or even trust-less) distributed systems, because this assumption does not hold in such cases [63]. Therefore, extending the abilities of secret sharing to this broader use case is natural and leads to the notion of verifiable secret sharing (VSS), which was introduced by Chor, Goldwasser, Micali and Awerbuch in 1985 [63].

Protecting against malicious dealers Shamir Secret Sharing assumes an honest dealer (i.e. honest share distributors), while VSS is designed to resolve the problem of faulty dealers. More precisely, when using a verifiable secret sharing scheme, each participant can verify that his own share was correctly created by the dealer. During reconstruction, VSS additionally provides protection against malicious participants.

In 1987, Feldman introduced an efficient VSS scheme based on Shamir's secret sharing protocol [64]. Feldman's scheme greatly reduces the amount off communication needed compared to the scheme presented by Chor et al. In addition, Feldman's VSS was the first to support non-interactive share verification. A participant who has received his share does not need to exchange further messages with the dealer in order to verify his share [64].

To illustrate the problems VSS protects against, consider the reconstruction phase of a (t, n) Shamir's secret sharing protocol. Assume that participants $P_1, P_2, ..., P_{t-1}$ already pooled their shares. Then, participant P_t can select an invalid share to influence the outcome of the reconstruction to its liking, given that P_t knows the pooled shares. Given that no more then t participants pool their share for reconstruction, this manipulation remains undetected. Even worse, in case of a dispute during reconstruction (e.g. different groups of t participants get different results), a malicious participant can just claim that the dealer provided the invalid share. In Shamir's case, this is totally plausible and there is no way to prove that a malicious participant indeed tried to manipulate the reconstruction. We explain some modern techniques to resolve these challenges, as well as the additional challenge of share verification for other third parties in the following section.

3.9 Publicly-Verifiable Secret Sharing

In this section, we present an even stronger notation of secret sharing, namely publicly-verifiable secret sharing (PVSS). This type of secret sharing scheme is the one used for state-of-the-art random beacon protocols as well as for our constructions. First, we summarize the properties expected from a (t, n) PVSS scheme with a dealer and n participants:

(1) Knowledge of at least t shares makes S easily computable [59].

(2) Knowledge of fewer than t shares leaves S completely undetermined, i.e. all possible values for S are equally likely [59].

(3) A malicious dealer, sending incorrect shares to some or all participants, should be detected [20].

(4) Malicious participants, providing invalid shares during reconstruction, should be detected [20].

(5) The verification process of distributed shares and shares submitted for reconstruction should be non-interactive [64].

The additional property required for a **P**VSS is then given by:

(6) Any third party (not necessary a participant) can verify the validity of the distributed shares and the shares used for reconstruction [20]. I.e. (5) should be publicly-verifiable.

There is a variety of different approaches for PVSS which meet those criteria. Stadler first described the notion of PVSS and introduced a PVSS scheme based on ElGamal's cryptosystem [65]. Since then, a variety of schemes based on different number theoretic assumptions have been proposed. Shil et al. in [66] give an overview and a comparison of these schemes.

3.9.1 Schoenmakers' PVSS

For our random beacon protocol, we consider Schoenmakers' PVSS [20], as this one is the simplest PVSS to our knowledge and is already used in a variety of state-of-the-art protocols such as Ouroboros [12], RandHound / RandHerd [18] and Scrape [19]. In the following, we describe Schoenmakers' PVSS, which we use as a fundamental building block for our protocol. This section describes the scheme itself, whereas chapter 6 describes how the scheme is applied in our solution. Formal correctness and security proofs for the scheme are provided in [20].

Schoenmakers' PVSS has as variety of properties which suit our use case:

- Common assumptions: The scheme is based on standard assumptions only, the Diffie-Hellman assumption (DDH), its computational variant (CDH) and the existence of secure cryptographic hash functions.
- Simplicity: The use of Schoenmakers' Special PVSS scheme, which provides a solution of sharing random secrets only, is sufficient for our use case.
- Low communication costs: The scheme works with a minimal number of messages. The dealer is required to broadcast a single message to the participants for share distribution. Participants are required to broadcast a single message for share reconstruction.

Compared to Shamir's scheme, Schoenmakers' PVSS additionally requires number theoretical assumptions: namely the existence of groups for which the discrete logarithm problem is intractable [20].

Publicly known parameters Following the notation from [20], n participants P_1, P_2 ,..., P_n and the dealer agree on a group G_q with two generators g and G. The group G_q is a group of prime order q, in which the discrete logarithm problem is hard. The generators g and G are independent, i.e. no party knows the discrete logarithm of g in respect to G. Further, the public keys $y_i = G^{x_i} \mid 1 \leq i \leq n$ are publicly known, where each $x_i \in_R \mathbb{Z}_q^\times$ denotes the corresponding private key of participant P_i.

Share distribution The following protocol describes the process of sharing a random value $S \in G_q$. Hence, we restrict ourselves to the description of the Special PVSS scheme introduced in section 3 of [20].

(1) Like in Shamir's secret sharing protocol (see section 3.7.1), the dealer constructs a polynomial of degree $t-1$.

$$p(x) = \sum_{j=0}^{t-1} \alpha_j x^j = \alpha_0 + \alpha_1 x + ... + \alpha_{t-1} x^{t-1}$$

The coefficients $\alpha_i \mid 0 \leq i \leq t-1$ are selected randomly from $\mathbb{Z}_q$. In Shamir Secret Sharing, we defined α_0 such that $S = \alpha_0$ holds (see section 3.7.1). However, here we define $s = \alpha_0$ and obtain S by computing $S = G^s$. Since the discrete logarithm is hard in G_q, we cannot obtain s from G^s and, thus, cannot share arbitrary secrets from G_q but only random ones. In our case, this poses no restriction, but Schoenmakers proposed a straight forward extension, which can be used to share arbitrary secrets [20].

(2) The shares are then computed by evaluating the polynomial $p(i)$ for each participant P_i as previously described for Shamir's secret sharing protocol. However, only the encrypted shares Y_i are published.

$$Y_i = y_i^{p(i)} \mid 1 \leq i \leq n$$

Share correctness proof In order to allow any third party to verify that the encrypted shares Y_i have been computed correctly, the dealer publishes a non-interactive zero-knowledge proof (NIZK proof) alongside the shares. The proof consists of three parts:

(1) The commitments C_j for the coefficients of the secret polynomial.

$$C_j = g^{\alpha_j} \mid 0 \leq j \leq t-1$$

(2) A common challenge $c \in \mathbb{Z}_q$, computed using a cryptographic hash function $H(\cdot)$.

$$c = H(\langle X_1, X_2, ..., X_n\rangle, \langle Y_1, Y_2, ..., Y_n\rangle, \langle a_{1,1}, a_{1,2}, ..., a_{1,n}\rangle, \langle a_{2,1}, a_{2,2}, ..., a_{2,n}\rangle)$$

The values $X_i \in G_q$, $Y_i \in G_q$, $a_{1,i} \in G_q$ and $a_{2,i} \in G_q$ for $1 \leq i \leq n$ are defined as follows:

$$\begin{aligned} X_i &= g^{p(i)} \\ Y_i &= y_i^{p(i)} \\ w_i &\in_R \mathbb{Z}_q \\ a_{1,i} &= g^{w_i} \\ a_{2,i} &= y_i^{w_i} \end{aligned}$$

(3) The n responses $r_i \mid 1 \leq i \leq n$ computed as:

$$r_i = w_i - p(i) \cdot c \quad \mid 1 \leq i \leq n$$

Notice that part (2) and (3) are the result of applying the non-interactive version of the protocol $DLEQ(g, X_i, y_i, Y_i)$ once for each participant, using a common challenge c.

DLEQ The (non-interactive version of the) $DLEQ(g_1, h_1, g_2, h_2)$ protocol is used in the PVSS scheme as a subprotocol to prove that two discrete logarithms in G_q are equal. More precisely, execution of the protocol shows that $log_{g_1} h_1 = log_{g_2} h_2$ for generators g_1, h_1, g_2, $h_2 \in G_q$. The protocol is described by Schoenmakers as follows: [20]

> It consists of the following steps, where [only] the prover knows α such that $h_1 = g_1^\alpha$ and $h_2 = g_2^\alpha$:
>
> 1. The prover sends $a_1 = g_1^w$ and $a_2 = g_2^w$ to the verifier, with $w \in_R \mathbb{Z}_q$.
> 2. The verifier send a random challenge $c \in_R \mathbb{Z}_q$ to the prover.
> 3. The prover responds with $r = w - \alpha c \pmod q$.
> 4. The verifier checks that $a_1 = g_1^r h_1^c$ and $a_2 = g_2^r h_2^c$.

This protocol requires interaction between the verifier and the prover, as the verifier has to provide the challenge c. To apply the protocol in a non-interactive way (i.e. only one message from the prover to the verifier is required), c is computed as a cryptographic hash of h_1, h_2, a_1, a_2. When running multiple instances of the protocol $DLEQ$ in parallel, a single common challenge can be used [20].

To apply the protocol $DLEQ$ to get parts (2) and (3) of the share correctness proof as stated above, $DLEQ(g, X_i, y_i, Y_i)$ is run for each participant P_i. Notice that the dealer computed $X_i = g^{p(i)}$, $Y_i = y_i^{p(i)}$ and, thus, knows $\alpha_i = p(i)$ which is required to run the protocol.

Share verification The dealer publishes the encrypted shares Y_i, the commitments C_j to the coefficients of the secret polynomial $p(\cdot)$, the common challenge c and the n responses r_i, i.e. the n-tuple $\langle \{Y_1, Y_2, ..., Y_n\}, \{C_0, C_1, ..., C_{t-1}\}, c, \{r_1, r_2, ..., r_n\} \rangle$. Given these values, any third party can verify the correctness of the encrypted shares without the need to decrypt the shares using the following three step approach:

(1) Compute $X_i = g^{p(i)} \mid 1 \leq i \leq n$ by calculating:

$$X_i = \prod_{j=0}^{t-1} (C_j)^{i^j}$$

Which is possible since:

$$X_i = g^{p(i)} = g^{\sum_{j=0}^{t-1} \alpha_j \cdot i^j} = \prod_{j=0}^{t-1} g^{\alpha_j \cdot i^j} = \prod_{j=0}^{t-1} (g^{\alpha_j})^{i^j} = \prod_{j=0}^{t-1} (C_j)^{i^j}$$

(2) Compute $a_{2i} = g^{w_i}$ and $a_{2i} = y_i^{w_i}$ for $1 \leq i \leq n$ using:

$$a_{1i} = g^{r_i} \cdot X_i^c$$
$$a_{2i} = y_i^{r_i} \cdot Y_i^c$$

Which yields correct values for a_{1i} and a_{2i} if and only if the published proof is valid since:

$$\begin{aligned} a_{1i} &= g^{w_i} & a_{2i} &= y_i^{w_i} \\ a_{1i} &= g^{w_i - p(i)\cdot c} \cdot g^{p(i)\cdot c} & a_{2i} &= y_i^{w_i - p(i)\cdot c} \cdot y_i^{p(i)\cdot c} \\ a_{1i} &= g^{w_i - p(i)\cdot c} \cdot \left(g^{p(i)}\right)^c & a_{2i} &= y_i^{w_i - p(i)\cdot c} \cdot \left(y_i^{p(i)}\right)^c \\ a_{1i} &= g^{r_i} \cdot X_i^c & a_{2i} &= y_i^{r_i} \cdot Y_i^c \end{aligned}$$

(3) Verify the correctness of the shares by computing:

$$c' = H\big(\langle X_1, X_2, ..., X_n\rangle, \langle Y_1, Y_2, ..., Y_n\rangle, \langle a_{1,1}, a_{1,2}, ..., a_{1,n}\rangle, \langle a_{2,1}, a_{2,2}, ..., a_{2,n}\rangle\big)$$

The shares are valid if, and only if, the condition $c = c'$ holds.

Share decryption For reconstruction of the shared secret S, at least t participants need to decrypt their shares Y_i. The decrypted share is defined by $S_i = G^{p(i)}$ and obtained using $S_i = Y_i^{x_i^{-1}}$, where x_i^{-1} denotes the multiplicative inverse of the private key x_i in $\mathbb{Z}_q^\times$.

Share decryption correctness proof In order to convince other participants of the fact that the share S_i is a valid decryption of Y_i, a participant P_i proves the correctness using the non-interactive version of the *DLEQ* subprotocol: $DLEQ(G, y_i, S_i, Y_i)$.

This shows that a participant P_i knows an α such that $G^\alpha = y_i$ and $S_i^\alpha = Y_i$. More intuitively, this shows that P_i knows the private key corresponding to y_i and that S_i is a decryption of Y_i using this private key.

Notice that $\alpha = x_i$ indeed is a solution for $G^\alpha = y_i$ and $S_i^\alpha = Y_i$:

$$\begin{aligned}
G^\alpha &= G^{x_i} = y_i \\
S_i^\alpha &= S_i^{x_i} = \left(G^{p(i)}\right)^{x_i} = G^{x_i \cdot p(i)} = (G^{x_i})^{p(i)} = y_i^{p(i)} = Y_i
\end{aligned}$$

The result of the non-interactive *DLEQ* protocol, i.e. the challenge $c = H(y_i, Y_i, a_1, a_2)$ and the corresponding response r, are provided alongside the decryption S_i of Y_i.

Share decryption verification This process is very similar to the verification of the encrypted shares Y_i published by the dealer. Any third party can validate the correctness of the decryption of a share S_i using the verification step from $DLEQ(G, y_i, S_i, Y_i)$. First, $a_1 = G^r \cdot y_i^c$ and $a_2 = S_i^r \cdot Y_i^c$ are computed. Then, the results are verified by comparing the hash $c' = H(y_i, Y_i, a_1, a_2)$ with c. The decryption is valid if, and only if, $c = c'$ holds.

Share reconstruction The reconstruction of the shared secret $S = G^s$ is accomplished by Lagrange interpolation as previously described for Shamir Secret Sharing in section 3.7.1. Let, without loss of generality, denote i_1, i_2, ..., i_t the indices of the validated shares, which should be used for reconstruction. Then S can be calculated using the following formula:

$$S = \sum_{j=1}^{t} S_{i_j} \prod_{\substack{k=1 \\ j \neq k}}^{t} i_k \cdot (i_k - i_j)^{-1}$$

Here, $(i_k - i_j)^{-1}$ denotes the multiplicative inverse of $(i_k - i_j)$ in $\mathbb{Z}_q^\times$.

CHAPTER 4

State-of-the-Art

In this chapter, we present different state-of-the-art approaches for generating publicly-verifiable and bias-resistant randomness in decentralized environments. We assume general understanding of the cryptographic primitives described in chapter 3.

As an introduction into the topic, we explain how randomness might be obtained by using commitment schemes in section 4.1. We further introduce how economic incentives can be combined with commitment schemes and illustrate resulting problems. The process of using Proof-of-Work as a source of randomness and the potential issues of this approach are also addressed in this section.

In section 4.2, we describe a state-of-the-art approach for generating randomness used in the Dfinity blockchain [14]. We focus on the construction of the random beacon protocol and describe the properties of the underlying cryptographic primitives used.

Section 4.3 introduces J. Chen's and S. Micali's approach for producing randomness as part of the Algorand protocol [17].

We finish the chapter with presenting PVSS-based approaches, namely Ouroboros [12], RandShare / RandHound / RandHerd [18] and Scrape [19], in sections 4.4, 4.5 and 4.6. In contrast to the other approaches, Scrape introduces an optimization of the underlying PVSS protocol, which can directly be applied to all protocols using Schoenmakers' PVSS. This, in particular, includes our proposed protocol as described in chapter 6.

4.1 Basic Constructions

In the following subsections, we illustrate some common approaches for generating random numbers in the context of blockchains. These approaches are considered as an introduction to the topic. While the approaches are quite useful in specific scenarios, they are not suitable as a general purpose protocol for generating verifiable randomness in decentralized systems. We highlight the individual problems in the corresponding subsections.

4.1.1 Commitment Schemes

A classical approach for generated random numbers in a two-party setting is called a commitment scheme. Using a cryptographic hash function $H(\cdot)$, the construction is straightforward and illustrated by the following example:

(1) Alice picks x as a 256 bit random integer.

(2) Bob picks a random 256 bit integer y.

(3) Alice sends the corresponding commitment $Com(x) \leftarrow H(x)$ to Bob.

(4) Bob sends y to Alice.

(5) Alice sends x to Bob

(6) Bob verifies that the condition $Com(x) = H(x)$ indeed holds.

(7) Both obtain the random number $x \oplus y$.

In order to ensure that none of the parties can manipulate the result $x \oplus y$, the order of the messages sent is important. In particular, Alice has to send the commitment $Com(x)$ before receiving y and Bob has to send y before receiving x. In case both participants follow the protocol, and the underlying hash function is secure, the result is indeed a randomly chosen 256 bit integer. Neither, Alice nor Bob can bias the result. This simple approach of commit and reveal can be extended for multiple participants.

Still, a critical problem with the approach remains: Alice learns the resulting random number as soon as she receives y. Bob, however, does not know the result at this point in time and, in fact, never gets to know the result if Alice chooses to abort the protocol at this stage – i.e. in case Alice does not send x to Bob. The problem is that Alice can decide, based on full knowledge of the resulting random number, whether or not the result suits her. Depending on this judgment, she can decide to send or withhold the value of x. In the multi-party setting, the problems gets worse as the protocol relies on the fact that all participants reveal their commitments.

4.1.2 Commitments Schemes with Security Deposits

To reduce the issues of simple commitment schemes as described in the previous section, a possible option is to introduce economic incentives. For this purpose, we consider the above approach, but use a Smart Contract (e.g. in the Ethereum blockchain) to verify the behavior of the participants and penalize malicious actors:

(1) Alice picks a random 256 bit integer x.

(2) Bob picks a random 256 bit integer y.

(3) Alice sends the corresponding commitment $Com(x) \leftarrow H(x)$ and a security deposit to the Smart Contract.

(4) Bob sends y to the Smart Contract.

(5) Alice send x to the Smart Contract.

(6) The Smart Contract verifies that the condition $Com(x) = H(x)$ indeed holds. Depending on the verification result either

    (1) the security deposit is returned back to Alice and both parties obtain the $x \oplus y$ or

    (2) Alice loses the security deposit and Bob does not get to know $x \oplus y$.

Additionally, the Smart Contract can also enforce that Alice has to provide x during a specific timeframe, e.g. within 1000 blocks after the Smart Contract receives y. The timeframe needs to be of considerable size to ensure Alice gets the transaction into the blockchain in time. In particular, this is important during periods where the blockchain is congested.[1]

Depending on the use for the resulting random number, in particular the associated economic value, the rational decision for Alice is to reveal or withhold the value of x. The underlying problem for Bob remains: there is no technical way to force Alice to reveal the value of x. Furthermore, the use of the Smart Contract poses additional challenges for Alice: Bob or others actors might try to prevent Alice from sending x to the Smart Contract. Miners could choose to not include the corresponding transaction into the blockchain – in fact, they could get paid by Bob to do so. From Alice's perspective, her security deposit might be at risk, even if she acts honestly and according to the described protocol.

[1] As blockchains such as Ethereum impose a limit on the number of transactions executed within a certain time frame, the processing of transactions might be delayed during periods of high transaction volume.

4.1.3 Proof-of-Work

Proof-of-Work blockchains, for example Bitcoin, rely on miners for transaction validation. After the verification of individual transactions, miners group them together into blocks. The header of a block typically includes the root of the transaction merkle tree as well as the hash of the previous block. This block header is repetitively hashed using different nonce values to find a block hash, which is lower than a certain difficulty target.

This process makes the blockhashes very difficult to (i) predict and (ii) manipulate by non-miners. Furthermore, a new random value is automatically part of each produced block. Therefore, such a protocol has very good liveness characteristics – it only stalls, if the underlying blockchain stalls.

Predictability and Bias-Resistance

Prediction as well as biasing the blockhashes used as random beacon values is possible for miners. However, they only have limited (and costly) options to predict or manipulate the next beacon value. As soon as a miner finds a block, it can either

- publish the block (immediately or at a later point in time) or
- withhold the block.

This decision directly influences the value of the random beacon. In the first case, its value is most likely[2] based on the miner's block, while the random beacon value is derived from a different block (potentially produced by a different miner) in the second case.

Incentives

So the key question here is: What are miners incentives for being honest vs. being dishonest – e.g. when do they publish or withhold?

Honest miners publish blocks as soon as the corresponding blockhashes are found. This allows other miners to build on top and maximizes the probability that the honest miner's block is part of the longest chain. As a result, the probability that the block reward and fees are actually paid to the particular miner are maximized.

However, there are also scenarios in which the decision to discard a block – and, thus, lose the block reward – is the rational one. One scenario, where this is the case, could be that the random beacon is used by a gambling service, and the miner itself has placed bets in this service. In this case, a rational miner would discard his block under the following two premises: (i) it looses his bet in case it publishes and (ii) the bet is higher than the block reward and fees.

[2]The are various influence factor such as network propagation delay.

In case the random beacon is used for leader selection, miners might selectively discard blocks to influence the selection of the next leader – i.e. only publish blocks, which lead to a selected leader controlled by the colluding parties.

Withholding and publishing at a later time might also be a rational decision. For illustration, consider a gambling service or lottery. The miner who withholds a block, already knows the next value of the random beacon and, thus, can potentially place bets on the profitable outcomes.

As soon as a miner finds a block, it could also try to find a different one, which yields a more suitable random beacon value. In case the miner is not able to do so, it can publish his found block immediately after another block is known – thus creating chances to still gain the associate block reward.

Use case scenarios

Besides the mentioned attack vectors, the use of a Proof-of-Work block hash as basis for a randomness beacon is quite appealing. Some reasons to consider are:

(1) easy implementation (implicit)

(2) verifiability

(3) unpredictability by non-miners

(4) good liveness characteristics

In cases where the amount at stake is less than the block reward, rational miners behave honest. The properties of the cryptographic hash function then ensure equally distributed[3] values and bias-resistance for the random beacon. The approach might also be extended to cases where a significantly higher amount is at stake. A possible extension is described in section 4.1.4.

Major drawbacks of the approach are for example the high computational resources required, as well as complex verification, which in principle requires the verification of the underlying blockchain itself.

[3] It is required to remove the leading zeros from the Proof-of-Work blockhash.

4.1.4 Iterated Proof-of-Work

In the previous section, we examined the properties of reusing the blockhashes of Proof-of-Work blockchains as a random beacon. In this section, we describe an extension to address the issues of that approach. The use of blockhashes from future blocks on its own is not sufficient, because a manipulation by miners is possible. When a miner finds a block, it can make an informed decision on whether or not to publish the block and, thus, influence the next random number directly.

While we cannot provide a way to force the miner to release the block, we can take away his ability to make a informed decision. At the cost of high computational resources, this yields a protocol producing very strong verifiable randomness, achieving near optimal bias-resistance and unpredictability.

Due to the high computational demands, the approach is solely suitable for scenarios where a random beacon value is required only a small number of times. The generation of initial protocol parameters might be an appealing use case.

The approach works as follows: The participants, who want to agree on a random beacon value, agree on some block number x of a future block B_x for example in the Bitcoin blockchain. As soon as this block is actually mined, the corresponding random beacon value R_x is derived from the block's hash $H(B_x)$ via the construction of a hashchain, i.e. by iteratively applying a hash function $H(\cdot)$ to $H(B_x)$:

$$
\begin{aligned}
R_x &= H(H(H(\dots H(B_x)\dots))) \\
R_x &= H^{\Delta t \cdot r}(H(B_x))
\end{aligned}
$$

Depending on the values of Δt and r, the above calculation is very time intensive and can only be computed *sequentially*. The parameter Δt specifies the targeted computational delay in seconds, whereas r is the rate of sequential hash operations per second.

As our tests show, $r \approx 1,000,000$ for an non-optimized Python implementation running on a notebook with an 2.3 GHz Intel® Core™ i5-5300 CPU. Thus, in practice $r \gg 1,000,000$.

While r is a performance assumption, Δt as a protocol parameter is selected much higher than the Bitcoin's block interval. This ensures that a miner M_A cannot calculate R_x before publishing a found block. If it tries, other (honest) miners would find and publish their blocks in the meantime. As a result, M_A's block is never included in the main branch of the blockchain and hence not used as the base block for obtaining R_x.

The value Δt serves as security parameter. For uses cases like protocol bootstrapping, even a value of one day ($\Delta t = 86400$) is suitable as the high amount of sequential computation involved to obtain and verify R_x has to be done only once.

4.2 Dfinity

A different approach for generating randomness in decentralized environments is developed as part of the Dfinity project [58]. The randomness generated is directly used at the core of Dfinity's consensus algorithm. Dfinity aims to provide a decentralized cloud platform, based on various techniques related to blockchain. The concept is somewhat similar to Ethereum and focuses on achieving improved performance, scalability and capacity.

Even though independent related work on Dfinity is very limited as of June 2017, we believe that their approach for generating verifiable randomness is a valuable contribution. Therefore, we provide a critical review of the Dfinity protocol with a particular focus on the construction of their random beacon protocol.

4.2.1 Threat and Communication Model

We can only describe Dfinity's threat model based on the fault tolerance example given in [14]. To the best of our knowledge, their exists no detailed description at the time of writing. Dfinity considers a threat model in which at most 30% of the network nodes at most fail or behave byzantine. Communication uses authenticated messages with pre-shared public keys. Further, an asynchronous BFT protocol is used as a subprotocol, but no additional details on the assumption in regard to synchrony of the overall protocol are given.

4.2.2 Unique Threshold Signatures

The concept of Dfinity is based on a cryptographic primitive called *unique threshold signatures*. We have introduced the underlying concepts of digital signatures schemes in section 3.5.

In Dfinity so-called BLS-signatures are used. BLS signatures, short for Boneh, Lynn and Shacham, are introduced in [15]. For details on how the signature scheme works, we refer the interested reader to their paper. In this thesis, will limited ourselves to the properties of particular interest: (i) uniqueness and (ii) the ability to perform signature aggregation. Both properties are essential for Dfinity's random beacon protocol and are described in the following sections 4.2.3 and 4.2.4.

4.2.3 Signature Uniqueness

A digital signature scheme is called unique if, and only if, the verification algorithm accepts exactly one signature per message. Arguably, the simplest unique signatures are based on RSA, which is described in section 3.5.1. Choosing the public exponent e as a prime number with $e > n$, where $n = p \cdot q$ is the modulus, ensures uniqueness [51]. For additional details as well as a counterexample for RSA signatures without this additional property, we refer the reader to sections 3.5.1 and 3.6.

BLS signatures, as used in Dfinity's protocol, are one of a few signatures schemes, which inherently provide the property of uniqueness. As we outlined in sections 3.5 and 3.6, deterministic signature schemes do not automatically imply uniqueness. BLS Signatures are quite new as they have only been introduced in 2003. Therefore, standardization, parameter recommendations as well as public libraries are coming up short compared to more traditional digital signatures schemes like DSA and its variants.

4.2.4 Signature Aggregation

The possibility for signature aggregation in a threshold signature scheme is closely related to secret sharing as introduced in section 3.7. In a (t, n) secret sharing scheme, a dealer distributes shares of a secret value s to n participants in such a way that any coalition of at least t participants can recover the secret s. A group of less than t participants cannot obtain any information about s.

In comparison, any group of at least t members in a (t, n) threshold signature scheme can construct a valid signature on some message M, whereas a coalition of fewer participants cannot do so. Such a threshold signature scheme consists of multiple parts:

(1) **Setup:** There are basically two options for setting up a threshold signature scheme:

   a) Key Distribution: A trusted dealer computes the group's private/public keypair, calculates a private key for each participant and sends these keys to the corresponding participant via a secure channel.

   b) Distributed Key Generation (DKG): The participants run a multi-party protocol to generate their private keys and the groups public key without a trusted dealer. During the DKG protocol, the group's private key is never available to a single participant.

(2) **Signing:** Computing a partial signature, also called a signature share, for a particular message.

(3) **Signature Aggregation:** Combining signatures shares from t participants to produce a signature for a particular message. This process is typically accomplished by Lagrange Interpolation, as described in section 3.7 for the purpose of secret sharing.

(4) **Signature verification:** Verifying an (aggregated) signature using the group's public key.

The theoretical basis describing how BLS can be used as threshold signature scheme are outlined in e.g. [16]. A working implementation of the BLS threshold signature scheme, which is used by Dfinity, is developed by S. Mitsunari and available at https://github.com/herumi/bls. It includes a command line interface which allows users to perform the above steps. Unfortunately, key generation is only supported via a trusted

dealer in this library. Additionally, there is a library for pairing-based cryptography including available at https://crypto.stanford.edu/pbc/.

4.2.5 Dfinity's Random Beacon

In this section, we explain how Dfinity uses unique threshold signatures to construct a random beacon protocol. The description is based on [14].

We assume that participants in the network register their public key on the underlying blockchain. In parallel to blockchain progression, participants form groups, run a distributed key generation protocol (DKG) and register the group's public key on the blockchain if the DKG was successful. Participants cannot freely select other nodes to form a group but are rather assigned to a specific group by the value of the random beacon itself.

Group selection At each block height h, one of the groups, which have registered their public key on the blockchain, is responsible for producing the next value of the random beacon. The responsible group is determined as follows:

$$G^{h+1} = \mathcal{G}[\sigma^h \mod |\mathcal{G}|]$$

Here, G^{h+1} represents the group responsible at block height $h+1$, whereas σ^h is the value of the random beacon at height h and $\mathcal{G}$ is the set of all registered groups.

Randomness generation Given a group G^h, the value σ^h of the random beacon at round h is obtained by aggregation of the signature shares of its members $\{\sigma_p^h \mid p \in G^h\}$. To obtain and then distribute σ_p^h, a group member p signs the previous value of the random beacon σ^{h-1} using his private key.

Due to the properties of the BLS signature scheme, each σ_p^h and, thus, also the aggregation is unique. The aggregated signature can be verified using the groups public key. This is an important property, which not only ensures a deterministic and verifiable sequence of random beacon values but also allows for off-chain signature aggregation. In addition, the size of the aggregated signature is constant and, thus, not dependent on the number of participants or the size of the group producing the signature.

4.2.6 Distributed Key Generation

Distributed key generation (DKG) is a key requirement for Dfinity's threshold signature scheme. DKG allows a group to agree on a public/private keypair for the BLS signature scheme in such a way that the private key is not exposed to any participant. This by itself is a hard problem, which Dfinity wants to solve via the use of Joint-Feldman Verifiable Secret Sharing [14]. As of June 2017, we were unable to find additional information on their approach for DKG or on a working prototype.

4.2.7 Properties

In the following, we provide an assessment of Dfinity's random beacon protocol in regard to important protocol characteristics such as: availability / liveness, unpredictability, bias-resistance and verifiability.

Availability / Liveness A critical factor in Dfinity construction is the ability to produce a valid threshold signature at each height of the chain. Whether or not this is possible depends on various factors – the most important being:

- Total number of nodes
- Number of faulty/byzantine nodes
- Threshold group size
- Signature aggregation threshold

The probability of failing to construct a valid threshold signature at a single height is approximately 10^{-17} [14]. In the example stated, at most 3000 of 10000 process are faulty, the group size is 400 and each set of 201 (non-faulty) group members is able to produce the threshold signatures. The failure probability is obtained via a hyper-geometric distribution and is therefore, simply speaking, the answer to the following question: Given 10000 nodes (3000 faulty ones, 7000 correct ones), what is the probability selecting at least 201 correct nodes when uniformly at random picking 400 nodes?

For, for such a claim to be true, is very important that the underlying assumption indeed hold – e.g. are the nodes actually *picked at random*. In this case, we thus need to ask whether or not an attacker can influence in which groups his nodes are placed. As described in [14], nodes are assigned to groups randomly based on the random beacon value itself. As important details are missing in the available information, the self-reference, i.e. using the random beacon to assign nodes to groups producing the random beacon, is very hard to assess. An attacker might influence the selection by carefully coordinating the time when it registers nodes or the public keys it uses. Whenever an attacker is able to register a group's public key onto the chain for which it controls half of the group members, it can prevent chain progression. A suitable countermeasure might be to "lock" registered nodes for some period of time and assign them to groups only after that period has elapsed.

Unpredictability and Bias-Resistance While Dfinity describes their randomness as unpredictable and unmanipulable [14] similar concerns as described for availability arise. In case the attacker is able to influence the nodes assignment to groups, unpredictability and bias-resistance is not ensured. Again, using the information available, a detailed assessment is not possible.

Verifiability The chain of random beacon values can be verified by a third party, which knows the registered group's public keys. Based on previous values of the random beacon, a verifier can determine which group was selected to construct the next random beacon.

4.3 Algorand

Algorand, as described by J. Chen and S. Micali in [17], is a proposal for a public distributed ledger. It aims to address problems of previous designs such as high computational costs (Proof-of-Work) or blockchain forks. The authors describe Algorand as follows: [17]

> *Algorand* is a truly democratic and efficient way to implement a public ledger. Unlike prior implementations based on Proof-of-Work, it requires a negligible amount of computation, and generates a transaction history that will not fork with overwhelmingly high probability.

Algorand, in its entirety, cannot be covered as part of these thesis due to its extent and complexity. Nevertheless, we introduce the key concepts of Algorand, focusing on two particular areas of interest:

- How is verifiable randomness generated in Algorand?
- Given such a randomness beacon, how can it be used as a building block for leader and verifier selection in a decentralized system?

4.3.1 Threat and Communication Model

Algorand's threat model is described in detail by Chen et al. in section 2.6 [17]. In the following, we summarize the key assumptions:

(1) The byzantine adversary controls less than one half of the total stake (money) in the system. This assumption is similar to controlling at most 50% of the computational power in e.g. Bitcoin or a scenario where an attackers controls at most f out of $n = 2f + 1$ nodes.

(2) The adversary is computationally bounded.

An additional distinguishing fact from other protocols is that the adversary is highly dynamic. At each point in time it can immediately corrupt any nodes it likes with the only restriction being that upper bound (1) is not exceeded.

The communication model assumes that all messages are delivered to all reachable nodes within some time bound. This time bound is depended on the network reachability and message size. Message integrity and authentication is ensured by digital signatures.

4.3.2 Algorand's Random Beacon

Algorand, at its core, requires unpredictable randomness for leader and verifier selection. This randomness is generated by making use of the properties of cryptographic hash functions and digital signatures. For the digital signature scheme, the uniqueness property, as discussed in sections 3.5 and 3.6, is required. In [17], the authors do not give details about which concrete *unique* digital signature scheme is used.

Following the notation from the original description in we use:

- Q^r for the value of the random beacon at round r
- l^r for the leader at round r
- $SIG_p(M)$ for the digital signature of the message M signed by participant p
- $H(\cdot)$ for a 256-bit cryptographic hash function

Q^0 is the initial random number and part of the system description. Given Q^{r-1}, there are two possible ways to derive the value of the random beacon for the next round Q^r:

(1) The leader l_r exists and reveals $SIG_{l_r}(Q^{r-1})$ together with his leadership credential during some predefined time interval. Then $Q^r = H(SIG_{l_r}(Q^{r-1}), r-1)$

(2) Otherwise $Q^r = H(Q_{r-1}, r-1)$

The leadership credential, i.e. a proof that l_r is in fact a potential leader for round r, is explained in detail in next section 4.3.4.

Key for this construction is the uniqueness of the signature scheme. This ensure that a (malicious) participant has only very limited options to influence the value of the random beacon. A malicious leader can only choose whether or not to reveal $SIG_{l_r}(Q^{r-1})$.

4.3.3 Verifier Selection

Algorand uses a byzantine agreement protocol to verify proposed blocks, which should be added to the blockchain. Running such a protocol among all participants in the network is not feasible as soon as the number of participants gets large. To circumvent this problem, Algorand selects a much smaller set of participants which perform the verification. In order to ensure that an attacker cannot control more than a third of the verifiers, although the number of verifiers is much smaller than the number of nodes an attacker might control, it is important that this selection is performed randomly.

In Algorand, the verifier set is highly dynamic. It changes after each round and even during individual steps in a single round. Using such a dynamic verifier set allows Algorand to deal with a powerful adversary model. For additional details, we refer the reader to [17].

Whether or not a player i has the role of a verifier during a step s in round r is determined based on the following condition:

$$.H(SIG_i(r, s, Q^{r-1})) \leq p$$

Here, $.H(x)$ denotes the hash value of x represented as a 256-bit binary number in the interval $[0, 1]$. Q^{r-1} is the value of the random beacon from the previous round and p is the probability of a participant being selected.

An important property of the above approach is that other players do not know which nodes are verifiers in a particular round. Attackers cannot know this fact either. Only participant i itself can compute $SIG_i(\cdot)$, and is therefore able to determine whether or not it is in the set of verifiers. In addition, a participant i can convince other participants that it has the verifier role by publishing his verifier credential $SIG_i(r, s, Q^{r-1})$.

4.3.4 Leader Selection

Leader selection is very similar to verifier selection. A participant i is called *potential* leader if $.H(SIG_i(r, 1, Q^{r-1})) \leq 1/n$, where n denotes the number of participants. It is possible that there are no leaders for a round r. This issue is addressed by automatically determining the next value of the random beacon based on the previous value. There is no distinction between the case of a leaderless round or a round with a non-responding leader.

When there are potential leaders, the one with the lowest value for $.H(SIG_i(r, 1, Q^{r-1}))$ is the round's leader. All potential leaders perform the following tasks:

(1) Propose a new block, which should be added to the blockchain.

(2) Sign the previous value of the random beacon to obtain the next value.

(3) Publish a proof of leadership, i.e. $SIG_i(r, 1, Q^{r-1})$

As the protocol does not ensure that there is only one potential leader, it is the verifiers duty to agree on one of the proposed blocks in case of multiple proposals.

4.3.5 Properties

In the following, we assess Algorand's random beacon in regard to important protocol characteristics.

Availability / Liveness Protocol liveness is ensured with high probability. The acceptable failure probability F of the protocol is a protocol parameter. Typical values given by the authors are $F = 10^{-12}$ and $F = 10^{-18}$ [17]. However, it is possible that there is no potential leader responding in a particular round. In this case, progress is nevertheless ensured, because the participants running the BFT protocol agree on this fact and a new leader is selected based on the next quantity $Q^r = H(Q_{r-1}, r-1)$.

Unpredictability Unpredictability of the random beacon values is ensured eventually. In case a sequence of malicious leaders is selected, those leaders can predict future values of the random beacon. However, the probability of such a selection decreases exponentially in the length of the sequence. As such, Algorand achieves unpredictability very similar to the probabilistic bounds given in our protocol (see section 6.6 and figure 6.3).

Bias-Resistance Bias-resistance for the random beacon values is *not* established. In case a malicious nodes is selected as a potential leader for some round, this node can always choose to (i) publish or (ii) withhold his leadership credential. Consequently, different values for the random beacon are obtained. In case two or more malicious participants fullfil the leadership requirement (see section 4.3.4), the adversary can further pick which leadership credential to publish.

Our evaluation is supported by the authors themselves, who explicitly discuss the scenario of malicious leaders [17]:

> When l^r is malicious, however, Q^r is no longer univocally defined from Q^{r-1} and l^r. There are at least two separate values for Q^r. [...] what matter[s] is that l^r has two choice[s] for Q^r, and thus it can double his chances to have another malicious user as the next leader. The options for Q^r may even be more numerous for the Adversary who controls a malicious l^r. [...]

The authors further argue, without giving specific numbers, that the adversary's choices cannot significantly reduce the probability of the selection of honest users in the future. In section 5.6, we evaluate our illustrative protocol and show that the impact on the leader selection probability is non-negligible. While this approach for obtaining the random beacon values is not identical to Algorand, it suffers from the same problem: malicious leaders selectively withholding values.

Verifiability The produced random beacon values can be verified by a third party with access to the public keys of the network nodes. In each step, the random beacon value is either based on (i) the previous beacon value only or (ii) on the previous value combined with a leadership credential. In case (i), verification can be accomplished by re-computation of the involved hash function, whereas an additional signature verification is required in case (ii). There might be the requirement for additional verification in regard to the agreement reached by the BFT protocol.

4.4 Ouroboros

Kiayias et al. [12] described a multi-party protocol for generating verifiable randomness. Their protocol called Ouroboros, uses the obtained randomness for the leader selection process in the context of a Proof-of-Stake system. The authors cover the overall design of a secure Proof-of-Stake system, whereas we limit ourselves to the description of one of the contributions in the paper: the process of simulating a trusted random beacon.

The protocol for simulating a trusted random beacon is based on the concept of Publicly-Verifiable Secret Sharing (PVSS). A detailed introduction to secret sharing is given in section 3.7. The details for the PVSS protocol used here are given in section 3.9.

4.4.1 PVSS Fundamentals

A (t, n) PVSS scheme is a protocol, which allows a dealer to share information among n participants in such a way, that any subset of this participants with more than t members can reconstruct the shared information. The parameter t is called the threshold of the (t, n) PVSS scheme. Any group with less than t participants cannot obtain any information about the shared secret.

The dealer computes a share for each participant in the protocol. Each share is encrypted with the public key of the corresponding participant and, thus, might be distributed via a public communication channel. The PVSS protocol ensures that anyone with access to the shares (e.g. not only the participants) can verify their validity. This is achieved using non-interactive zero-knowledge (NIZK) proofs and is an essential property of the PVSS scheme as it removes the requirement for a trusted dealer. Participants and third parties can verify that the dealer behaves according to the protocol on their own.

In addition, the used Schoenmakers' PVSS [20] ensures that participants cannot manipulate the reconstruction process – they might submit invalid shares; other participants or third parties can, however, verify the submitted shares and detect a manipulation. As long as at least t valid shares are pooled for reconstruction, the process is successful.

4.4.2 Threat and Communication Model

Before introducing how PVSS can be used to simulating a trusted random beacon, we discuss the threat and communication model.

In Ouroboros, it is assumed that an adversary controls less than 50% of the stake in the system. This approach is similar to the traditional setting, in which an attacker is controlling at most f out of $n = 2f + 1$ nodes.

The communication model is a synchronous one, where time is split into discrete units called slots. A slot is associated with a block of the underlying distributed ledger [12]. Participants exchange / broadcast messages via this blockchain. It is assumed that broadcasted messages are received by other participants during the same slot they are sent [12].

4.4.3 Ouroboros' Random Beacon

In the following, we describe the construction for the random beacon protocol as part of the Ouroboros protocol based on [12]. The protocol is run by n participants $\{P_1, P_2, ..., P_n\}$ and is separated into three phases – commitment, reveal and recovery phase. The security of the construction is based on the assumption of an honest and participating majority.

Commit phase During the commit phase, each participant executes the share distribution process for a $(t, n-1)$ PVSS protocol in the role of the dealer. The threshold t may be set to $f+1$. This ensures that any (honest) majority is able to reconstruct the shared value during the reconstruction phase if necessary, while a colluding attacker cannot reconstruct the value without participation of honest participants.

For this purpose, each participant generates randomness for its private polynomial and computes the shares for the other participants. These shares together with the zero-knowledge correctness proof are published through the blockchain as communication mechanism.

Reveal phase After a fixed timeframe (e.g. 4000 slots as described in the original paper) the reveal phase starts, if a majority of participants provided valid commitments. Otherwise, the protocol stalls. Similar to the process used in simple hash based commitment schemes, Schoenmakers' PVSS allows to reveal the shared secret without requiring a reconstruction. A possibility is that each participant in the role of the dealer publishes the coefficient α_0 of the underlying PVSS scheme. The other participants can verify the correctness by checking that the condition $C_0 = g^{\alpha_0}$ holds. Additionally computation of the shared secret $S = G^{\alpha_0}$ is possible using the additional information of α_0.

Recover phase After the reveal phase (e.g. 4000 slots), the participants begin to recover any missing shared secrets. While this phase could be run simultaneously with the reveal phase, Kiayias et al. prefer to run them sequentially for efficiency [12]. Participants publish their decrypted shares and the corresponding share decryption proofs for all the valid commitments which have not been revealed. As soon as a majority of the participants have finished executing these steps, the shared secrets for all the valid previously-submitted commitments are known to the participants.

Combining the shared secrets After the three phases of the protocol have been finished, the underlying blockchain contains all the information required to compute the value of the random beacon and verify its correctness. No further message exchange is required. Without loss of generality, let $\{S_1, S_2, ..., S_m\} \mid t \leq m \leq n$ be the set of shared secrets so that for each secret S_i, a valid PVSS commitment (i.e. the shares and their zero-knowledge correctness proof) has been submitted during the commitment phase. The next value of the random beacon is then obtained in a deterministic fashion by combining the entropy from $\{S_1, S_2, ..., S_m\}$ e.g. by using a hash function. For this step,

it is irrelevant whether S_i is obtained by reconstruction or directly during the reveal phase.

4.4.4 Properties

In this section, we describe some of the key properties of Ouroboros' PVSS-based random beacon. When considering the protocol from a high level view, it is quite similar to hash-based commitment schemes as described in section 4.1.1. In both approaches, participants, in a first step, commit themselves to values, which are revealed at a later point in time (e.g. after all / a majority of the commitments are collected).

However, the key advantage of the PVSS-based protocol is the ability to force the revealment of the committed values. In a hash-based commitment scheme, the last party / the last colluding parties can choose whether or not to reveal their values based on the outcome of the random beacon. The outcome can be precomputed by malicious actors as soon as honest participants have revealed their values. This problem and its implications are described in further detail in section 4.1.1. Based on the honest participating majority assumption, withholding the committed values after publishing a commitment is not effective in the PVSS-based protocol, because honest nodes can always recover the shared secrets.

Availability / Liveness Availability is ensured in the given model. After a fixed time interval, a majority of honest participants has provided their secret shares for all other participants and the protocol progresses to the reveal phase. The reveal and recover phases succeed as the threshold for reconstruction is defined in a way that always allows an honest majority to recover the commit secret values.

Unpredictability and Bias-Resistance As there is an honest majority required to reach the recover phase of the protocol, at least one honest participant P_h has committed itself to a random number S_h. This random number is further part of the resulting values $\{S_1, S_2, ..., S_h, ..., S_m\}$. All the other values might come from a colluding attacker. However, at the time the attacker commits to its values, it cannot know S_h, because an honest majority is required to obtain it by reconstruction. Per definition, honest nodes do not participate in any recovery process during the commitment phase. Therefore, the value of the random beacon depends on some true random value S_h as well as on some other values, which might be manipulated by an adversary. At the time the adversary has to decide on his values, it, however, cannot know S_h and, thus, cannot predict the value of the random beacon. Therefore, S_h and the values of the adversary are independent sources of randomness. The combination of such sources is as strong as the strongest one of them [67]. Thus, Ouroboros achieves both unpredictability and bias-resistance.

Verifiability The protocol uses a blockchain as the underlying communication mechanism. All exchanged messages are thus written to the blockchain. The participant's public keys are also registered on the blockchain. Using this information, any third party,

which is not necessarily a participant, is able to verify the execution of the protocol and the value of the random beacon. This property follows directly from the used secret sharing share, described in section 3.9.

Communication complexity During the commitment phase, each participant has to distribute a single message via the blockchain. This message includes (i) the encrypted shares for each participant and (ii) the zero-knowledge correctness proof for the shares. Therefore, such a message has size $\mathcal{O}(n)$.

During the reveal phase, honest participants reveal their values, resulting in one message broadcast of constant size per participant.

Recovering a single non-revealed value requires one message broadcast of constant size from at least t participants. However, all participants might participate during the reconstruction. This leads to $\mathcal{O}(n)$ broadcasted messages per reconstruction. A participant, which provides shares for more than one recovery, might pool the corresponding messages to a single one of size $\mathcal{O}(t)$.

The use of an agreed broadcast channel somewhat hide the fact that the number of messages sent between all participants is actually $\mathcal{O}(n^2)$ assuming a naive broadcast implementation. Therefore, we summarize the actual number of messages sent in table 4.1. For the purpose of estimating the total amount of data transferred, we consider a 256-bit elliptic curve implementation of Schoenmakers' PVSS. This means we assume group elements and exponents can each be represented using 256 bits. As n describes the total number of participants but shares have to be distributed to all other participants except oneself, n instances of a $(t, n-1)$ PVSS scheme are run. We consider a typical $n = 3f + 1$ scenario, in which an adversary controls less than a third of the nodes and set $t = f + 1$ accordingly.

	commit	reveal	recover
number of messages to broadcast per sender	1	1	f
number of senders	n	$n - f$	$n - f$
number of receivers	$n - 1$	$n - 1$	$n - f - 1$
total number of messages (assuming naive broadcast)	$n \cdot (n-1)$	$(n-f) \cdot (n-1)$	$f \cdot (n-f) \cdot (n-f-1)$

Table 4.1: Communication requirements for Ouroboros' random beacon protocol

The messages sizes for a single participant per operation are given by:

(1) Commit: $(2(n-1)+t+1) \cdot 256$ bit

- Encrypted shares: $(n-1) \cdot 256$ bit
- Commitments to coefficients: $t \cdot 256$ bit
- NIZK proof (common challenge): 256 bit
- NIZK proof (responses): $(n-1) \cdot 256$ bit

(2) Reveal: 256 bit

(3) Recover: $3 \cdot 256$ bit

- Decrypted share: 256 bit
- NIZK-Proof (challenge): 256 bit
- NIZK-Proof (response): 256 bit

Combining the communication requirements from table 4.1 with the messages size given above, we can estimate the total amount of data transfer required by the protocol, which is illustrated in figure 4.1.

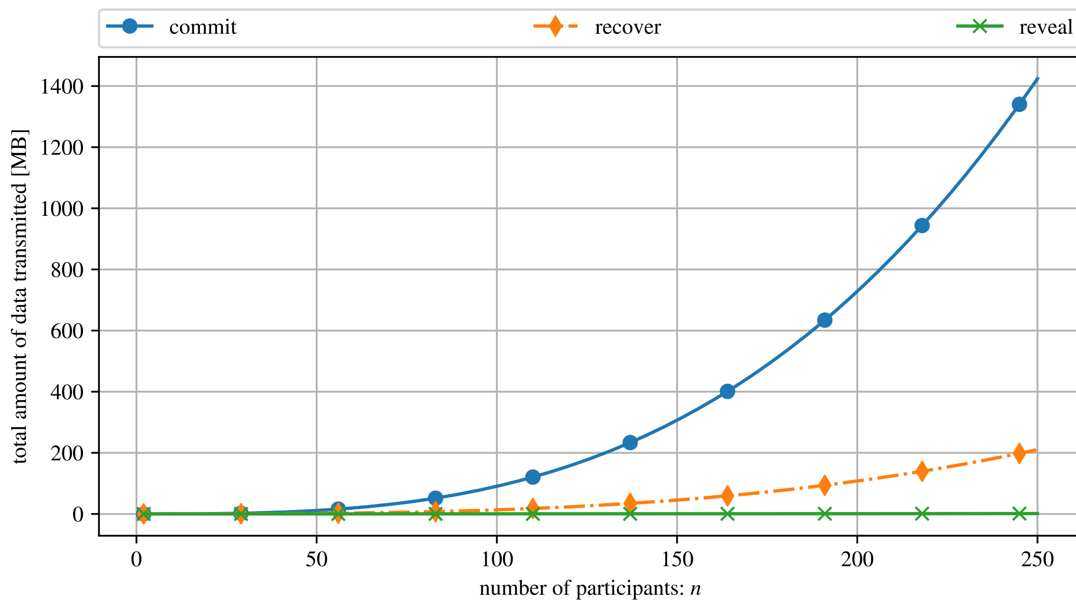

Figure 4.1: Total amount of communication required for a PVSS-based random beacon protocol, assuming an adversary controlling up to 33% of the nodes

4.5 RandShare, RandHound and RandHerd

RandShare [18], RandHound [18] and RandHerd [18] are three protocols for obtaining bias-resistant randomness in decentralized environments. The authors describe RandShare as a small-scale protocol only described for introduction, whereas RandHound and RandHerd are tailored towards large scale deployment and are considering scalability aspects. In RandHound, a client needs to contact a set of RandHound servers to obtain a new random beacon value while RandHerd uses RandHound in a setup process and then delivers a stream of random beacon values without a client interacting with the system.

The authors' evaluation of their prototypes shows good performance across hundreds of participants. In an exemplary scenario with 512 nodes, which are assigned into groups of 32, a client using RandHound can obtain a new random beacon value after 240 seconds from the RandHound servers [18]. After protocol setup, which takes about 260 seconds, RandHerd can output random beacon values at intervals of approximately 6 seconds [18].

Both RandHound and RandHerd fail to provide liveness in certain scenarios, where too many byzantine nodes are assigned to subgroups of nodes. The upper bound for the protocol failure probability varies depending on the group size and the number of used groups. For the above configuration, Syta et al. give a failure probability of 0.08% [18]. RandShare is not effected by the same problem.

4.5.1 Threat and Communication Model

For the protocols, a threat model, where a byzantine adversary controls at most f out of $n = 3f + 1$ nodes, is considered. Further, the authors assume asynchronous messaging, where each message is eventually delivered and authenticated message channels are used. Sent messages are signed and participants verify the signatures of all incoming messages using a copy of the preshared public key of the sender.

The authors only explicitly state the asynchronous model for the illustrative protocol RandShare. No details on the communication model are given for the more elaborate RandHound and RandHerd protocols. The following quote from section III. A. indicates that the communication model for RandHound is a (partially) synchronous one: [18]

> The client chooses a subset of server inputs from each group, omitting servers that *did not respond on time* or with proper values, thus fixing each group's secret and consequently the output of the protocol.

4.5.2 RandShare

Before introducing RandHound and RandHerd, the authors introduce a simpler protocol called RandShare [18]. RandShare is similar to the Ouroboros approach described in section 4.4.

Each participant acts as the dealer in an instance of PVSS, e.g. Schoenmakers' PVSS, and therefore shares a secret value with the other RandShare nodes. After the participants

have committed to their values, these values are revailed or recovered and then combined to obtain the random beacon value. To tolerate up to f byzantine nodes, honest participants do not reveal their committed values and do not participate in any reconstruction until they have decided on which shares to combine. For this purpose, a byzantine agreement protocol is run in combination to the process of share verification.

The successful completion of the protocol is only ensured after a so-called barrier point [18]. Whether or not this point is reached is depended on the outcome of the byzantine agreement protocol. If the participants agree on a set of at least $f + 1$ commitments, this set contains at least one commitment from an honest node. As all commitments are revealed or can be recovered by a collaboration of honest nodes, the resulting secrets can be combined, and consequently form an unbiased random beacon value as at least one honest node's secret is part of the combination.

4.5.3 RandHound

RandHound aims to address RandShare's scalability issues, which directly arise from the fact that RandShare uses a traditional byzantine agreement protocol among the set of all participants. In RandHound, secrets are not shared among all servers but only within defined subsets, reducing the communication and computational overhead from $\mathcal{O}(n^3)$ to $\mathcal{O}(nc^2)$ [18]. As a client / server protocol, it is the client's role to initiate communication with the RandHound server to obtain randomness. Figure 4.2 illustrates RandHound design:

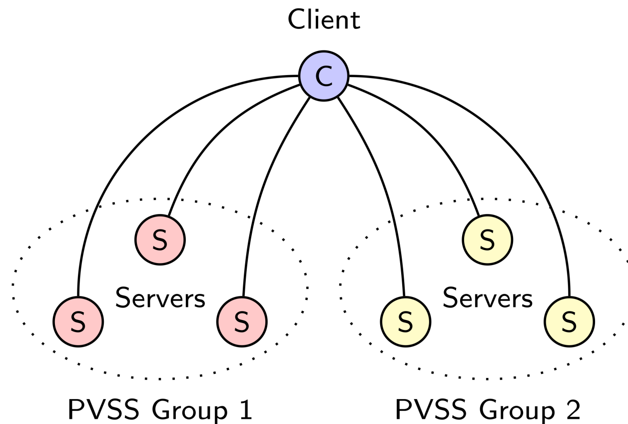

Figure 4.2: Overview of the RandHound design [18]

In the following, the steps required to obtain a new random beacon value using RandHound are given in more detail:

(1) The client constructs a session configuration, which includes a description of the intended purpose of the randomness and uniquely identifies the protocol run.

Further, the client randomly assigns servers to groups. In the exemplary scenario, 16 groups, each containing 32 servers, are used. The session configuration and the group assignment is sent to all servers.

(2) Upon receiving the session configuration, servers store it and can therefore detect malicious clients, who try to re-run the protocol multiple times until they receive a favorable outcome. If a server sees the configuration for the first time, it performs the share distribution process of a PVSS protocol, generating shares of a random secret for all other servers in their group. The shares are not distributed to the other servers but rather sent back to the client.

(3) The client randomly selects which server's secret should be used to obtain the value of the random beacon. For each of the groups, it has to select more than one third of all servers belonging to that group. In case two thirds of the nodes in any group do not respond, i.e. because they are controlled by the adversary, the protocol stalls.

(4) The above selection is sent to all servers, which acknowledge the client's choice.

(5) The client presents the acknowledgment to all servers, which verify the validity and respond with the decrypted shares.

(6) The client recovers the randomness for all selected servers and uses Lagrange interpolation to recover missing secrets. In case a single secret cannot be recovered, the protocol is aborted. Otherwise the client combines the individual secrets and constructs the random beacon value.

(7) The client presents the random beacon value as well as a transcript of the protocol run to convince a third party of the fact that the random beacon value was indeed obtained as described by the protocol.

Availability / Liveness An adversary might prevent honest clients to obtain a random beacon value. This can happen in particular when the selected grouping from step (1) contains groups, which contain less or equal to a third of honest servers. The protocol might stall in step (3) as malicious servers might not respond, or at step (5) in case byzantine servers do not follow the request of share decryption.

Unpredictability and Bias-Resistance In step (1), an adversarial client might choose a non-random grouping. Using this approach, the client cannot obtain biased randomness. The pigeonhole principle ensures that there is at least one group, where at most a third of the nodes behave byzantine. For this group, the client cannot know the resulting randomness before it commits to the outcome in step (4). As the client has to combine randomness from all groups including at least one group which provides an unbiased and unpredictable value, the resulting random beacon value is unbiased and unpredictable.

Malicious clients can, however, run the protocol multiple times. While (honest) servers check for repeated protocol execution using the same session configuration, a client can try different session configuration which are similar (e.g. they only differ slightly in their purpose string, timestamp and so forth). This can lead to a malicious client, which can choose one of multiple random beacon values as such minor differences might be hard to identify for third parties.

Verifiability The resulting random beacon value is third party verifiable. For this purpose the client, which obtained the random beacon values, provides the result, the session configuration and the transcript of the protocol run to the verifier, who can independently check the steps of the protocol execution.

4.5.4 RandHerd

RandHerd is the third protocol presented by Syta et al. in [18]. It is tailored towards the use case of providing a sequence of random beacon values at regular intervals. RandHerd is based on RandHound and improves its performance in case of repeated execution. The authors describe their protocol as follows: [18]

> RandHerd provides a continually-running decentralized service that can generate publicly-verifiable and unbiasable randomness on demand, at regular intervals, or both. RandHerd's goal is to reduce communication and computational overhead of the randomness generation further from RandHound's $\mathcal{O}(c^2 n)$ to $\mathcal{O}(c^2 \log n)$ given a group size c.

In the descriptions given in [18], the authors briefly mention that tree-structured communication and aggregation are used to reduce the complexity to $\mathcal{O}(c^2 \log n)$ **per server**. However, no details on the construction of this primitives and the implied reduction in communication complexity by the factor of $\frac{\log n}{n}$ are given. It is not clear in which scenario the stated complexity of $\mathcal{O}(c^2 \log n)$ indeed holds and what the communication complexity is in regard to the overall system.

In contrast to RandHound, RandHerd does not rely on a client to initiate the protocol. Instead, a RandHerd instance is given by its configuration, which consists of a collection of the servers' public keys and the instance's collective public key. RandHound is used for protocol setup in order to established a secure sharding of nodes into groups. After a successful setup, the random beacon values are obtained using an involved protocol based on the techniques of "threshold-based witness cosigning" [18]. In the following, we discuss important protocol properties, whereas for a detailed protocol description we refer the reader to the original paper.

Availability / Liveness A direct consequence of using RandHound for protocol setup is that RandHerd has at least the same system failure probability as RandHound. An additional problem for availability arises, as RandHerd, at various stages of the protocol,

requires a leader in order to make progress. In the defined threat model, the probability of selecting a malicious leader is approximately $1/3$, thus highly likely. In the scenario of a malicious leader, availability is ensured by selecting a new leader using a BFT view change protocol. Here, the authors give no additional details on the concrete protocol and its communication overhead. In section C. Security Properties for the RandHerd description, availability is only claimed in case of an honest leader [18]:

> Given an honest leader, the protocol successfully completes and produces the final random output Z with high probability.

Unpredictability and Bias-Resistance The protocol achieves unpredictability and bias-resistance in the presence of a byzantine adversary, which is controlling up to f out of $3f + 1$ nodes. Both properties are ensured by the pigeonhole principle in a similar way that RandHound accomplishes them.

Verifiability The produced random beacon outputs can be verified by a third party, which checks the corresponding signature using the RandHerd's instance collective public key. The verifiability of the protocol setup is directly based on the properties of RandHound.

4.6 Scrape

The construction of the Scrape random beacon protocol, as introduced in [19], is very similar to the Ouroboros protocol outlined in detail in section 4.4. Therefore, we limit ourselves to the key contributions of the authors: a variant of Schoenmakers' PVSS protocol.

The proposed PVSS improves the computation complexity for share verification to $\mathcal{O}(n)$, compared to Schoenmakers' variant which requires $\mathcal{O}(nt)$ exponentiations [19]. In particular, when considering the fact that Ouroboros, Scrape and other PVSS-based protocols like RandShare use n instances of the PVSS protocol simultaneously, the optimization is an important contribution to achieve better scalability.

Scrape's PVSS is based on the idea that secret sharing is equivalent to encoding the secret with the corresponding Reed Solomon error correcting code [19]. We do not provide the background information on Reed Solomon error correction codes here, but neverthelesss present the required changes to Schoenmakers' PVSS. In particular, we consider the description of the π_{DDH} protocol from [19]. Wherever possible, we follow the original notation provided by Schoenmakers.

4.6.1 Share Distribution

During share distribution, the dealer previously computed the commitments $C_0, C_1, ..., C_{t-1}$ using the coefficients $\alpha_0, \alpha_1, ..., \alpha_{t-1}$ of the underlying polynomial $p(\cdot)$:

$$C_j = g^{\alpha_j} \mid 0 \leq j \leq t-1$$

In the modified protocol, the commitments are computed based on the values of the polynomial instead:

$$C_i = g^{p(i)} \mid 1 \leq j \leq n-1$$

Consequently, the NIZK share correctness proof is adapted. The parallel composition $DLEQ(g, C_i, y_i, Y_i) = \langle c, r_1, r_2, ..., r_n \rangle$ of the DLEQ subprotocol (illustrated in section 3.9.1) is used to prove correctness of the encrypted shares. Notice the correspondence $C_i = X_i$ between the new construction of the values C_i and Schoenmakers' definition of X_i.

4.6.2 Share Verification

The share verification previously relied on computing the values of X_i using $C_0, C_1, ..., C_{t-1}$:

$$X_i = \prod_{j=0}^{t-1} (C_j)^{i^j} \mid 1 \leq i \leq n$$

The computation involves $\mathcal{O}(nt)$ exponentiations and is therefore inefficient for large sets of participants. This calculation is not necessary in the modified PVSS. Instead, the NIZK proof can be verified directly, as all required values, in particular $C_i = X_i \mid 1 \leq i \leq n$, are already provided.

After verifying the NIZK proof, the verifier has to perform an additional check to ensure the commitments are valid. For this purpose the verifier samples a random codeword of the dual code, corresponding to the instance of the secret sharing scheme, and checks whether the inner product with the share vector is 1 [19]. This means it performs the following steps:

(1) Sample a random codeword $\langle c_1^\perp, c_2^\perp, ..., c_n^\perp \rangle$ from the dual code with $c_i^\perp = \lambda_i f(i)$. The definition of the Lagrange coefficients λ_i and the polynomial $f(\cdot)$ are given as follows:

$$f(x) = \sum_{i=0}^{n-t-1} \beta_i x^i \quad \text{with } \beta_i \in_R \mathbb{Z}_q$$

$$\lambda_i = \prod_{j \neq i} \frac{j}{j-i}$$

(2) Compute the inner product with the shares vector $\langle p(1), p(2), ..., p(n)\rangle$, and check whether the result is 1. The verification is only successful if the following condition holds:

$$\prod_{i=1}^{n} C_i^{c_i^{\perp}} = g^{\sum_{i=1}^{n} p(i) c_i^{\perp}} = 1$$

The interested reader can follow the above procedure in detail in [19]. The authors outline the approach as well as the underlying principles in regard to coding theory and Reed Solomon error correction codes.

4.6.3 Use Cases

The described modification to Schoenmakers' PVSS can directly be applied to PVSS-based random beacon protocols using the same underlying PVSS protocol. In particular, the optimization can be used in

(1) the Ouroboros protocol (see section 4.4),

(2) the RandShare / RandHound / RandHerd protocols (see section 4.5),

(3) our proposed solution (see chapter 6) as well as

(4) our protocol extension Quorum Share Distribution (see section 7.1).

The concept is orthogonal to the mentioned protocols and directly reduces computation complexity by a factor t during share verification.

CHAPTER 5

Hashchain-based Random Beacon

In this section, we introduce a concept for generating publicly-verifiable randomness in decentralized systems. First, we describe a simplified random beacon based on *hashchains*. In chapter 6, we then rely on these underlying concepts to introduce a random beacon protocol based on PVSS.

5.1 Scenario and Threat-Model

We assume a permissioned blockchain setting, i.e. there is a fixed set of participants. For simplicity, we further assume that these participants are known at the start of the protocol and do not change over time. The communication model is a synchronous one, in which message propagation time is fixed by some time bound. We further assume pre-shared public keys, which are used for authenticated communication channels between the participants.

5.2 Hashchains

Our construction is based on a cryptographic primitive called a hashchain. Hu et al. describes them as follows [68]:

> Lamport first proposed to use one-way chains for one-time password authentication [69]. Subsequently, researchers proposed one-way chains as a basic building block for digital cash, for extending the lifetime of digital certificates, for constructing one-time signatures, for packet authentication, etc. As one-way chains are very efficient to verify, they recently became increasingly popular for designing security protocols [...]

Given some (secret) seed value s, a hashchain is computed by repeatedly applying a cryptographic hash function $H(\cdot)$. In each step, the result of the previous step is taken as the argument for the hash function, starting with s as initial argument. We introduce the following notation, where v_j represents the value of the hashchain after j steps:

$$
\begin{aligned}
v_0 &= s && = H^0(s) \\
v_1 &= H(s) && = H^1(s) \\
v_2 &= H(H(s)) && = H^2(s) \\
&\dots \\
v_j &= H(v_{j-1}) && = H^j(s) \\
&\dots \\
v_d &= \dots && = H^d(s)
\end{aligned}
$$

Performance As modern hash functions such as SHA-2 or SHA-3 are quite fast, computing hashchains with a significant length is possible. The calculation of the value of the hashchain after 10^6 iterations takes less than one second.[1]

One-way characteristic As secure cryptographic hash functions are resistant against preimage attacks, going back a step in the hashchain, i.e. computing the value v_{j-1} given v_j, is infeasible in practice. Thus, given some value v_j of a hashchain, it is only possible to calculate v_{j+1} efficiently, but not v_{j-1}.

Commitment and Verification Publishing a value v_d commits a participant to all the previous values $\{v_j \mid 0 \leq j < d\}$ in his hashchain. This means that if a participant reveals some intermediate value v_j at a later point in time, other participants can check if this value is indeed part of the hashchain, given the value of v_d. The verification procedure is given in algorithm 5.1.

[1] A corresponding test was performed using the SHA2-256 algorithm from Python's hashlib package running on a 2.3 GHz Intel® Core™ i5-5300.

Algorithm 5.1: Verification of Hashchain Values

Input:

- a commitment $v_d = H^d(s)$ for a hashchain
- a candidate value v for this hashchain

Output:

- VALID if $v \in \{s, H(s), H^2(s), ..., H^d(s)\}$,
- INVALID otherwise

```
1 t ← v
2 for j ← 1 to d do
3     t ← H(t)
4     if t = v_d then
5         return VALID
6 return INVALID
```

This algorithm has linear runtime $\mathcal{O}(d)$ in the worst case. For the special case of verifying if v is preimage of the head v_d of the chain, one can simply compute $H(v)$ and compare the result to v_d. In this case, verification in constant time is possible. For our protocol, we exclusively use this special case (see section 5.4.3). Revealing the last unrevealed value, i.e. v_{d-1} at the first step, v_{d-2} at the second step, and so forth, is of particular interest because:

(1) When revealing v_j, a participant is still committed to all values $v_k \mid k < j$.

(2) Verification can be done in constant time $\mathcal{O}(1)$.

On the contrary, revealing v_k also reveals the commitments for all $v_j \mid j > k$, as this values can trivially be computed by (iteratively) applying the hash function $H(\cdot)$ to v_k.

5.3 Setup Phase

Based on the characteristics of hashchains as introduced in the previous section, we now introduce a random beacon protocol. We distinguish two phases: setup and operation. During the setup phase, each participant P_i performs the following steps:

(1) Generate a 256-bit random number s_i only known to P_i. This number is used as the secret seed for the participant's hashchain.

(2) Compute the value $c_i = H^d(s_i)$ where d is some large positive integer, e.g. $d = 10^9$. We refer to c_i as the head of the hashchain and to d as the length of the hashchain.

(3) Publish the value c_i.

We assume that all participants agree on the set of values $\{c_1, c_2, ...c_n\}$ included in the initial block.

After the above setup steps are performed, all participants have to further agree on a random value R_0, used as a starting point (seed) for the random beacon during the operation phase. How R_0 is selected has some important implications. Different approaches are outlined in more detail in section 5.7. In the following, we assume the ideal case of R_0 being selected from the set $\{0, 1, \ldots, 2^{256} - 1\}$ uniformly at random. The seed R_0 becomes public knowledge only after the participants have agreed on the commitments during the setup phase.

5.4 Operation Phase

After bootstrapping the random beacon in the setup phase, normal operation starts. Based on an initial 256-bit value R_0 and the inputs from the participants, the future values of the random beacon R_1, R_2, R_3 and so forth are derived.

We describe the protocol in a synchronous communication model. The operation phase proceeds in rounds. The value of the random beacon R_1 is the result of round 1, R_2 is the result of round 2 etc. In each round x, the following steps are performed:

(1) The leader L_x of round x is selected.

(2) The leader reveals the last preimage of his hashchain.

(3) All other participants verify the revealed preimage.

(4) Random beacon value R_x is derived from R_{x-1} and the revealed preimage.

In the following subsections, we describe these steps in detail.

5.4.1 Leader Selection

In each round $x \geq 1$, the value of R_{x-1} of the previous round is used to deterministically select a unique leader L_x for this round. We denote the fact that P_i is the leader of round x as $L_x^{(i)}$, and define L_x directly based on a uniformly random selection among all participants:

$$L_x = P_i \iff R_{x-1} \equiv i \pmod{n}$$

The set of potential leaders $\mathcal{L}_x$ for round x is equal to the set of all participants $\mathcal{P}$, i.e. the following condition holds:

$$\forall x \geq 1, \mathcal{L}_x = \mathcal{P}$$

This is a difference to the PVSS protocol described in chapter 6, where we restrict the set of potential leaders.

5.4.2 Revealing Preimages

During the setup phase, each participant P_j has committed itself to a hashchain $\langle H^0(s_j), H^1(s_j), ..., H^d(s_j)\rangle$ via publishing $c_j = H^d(s_j)$. In particular, $L_x^{(i)}$ has done so via publishing the commitment c_i. In any round x, the round's leader $L_x^{(i)}$ is in charge of revealing the latest unrevealed preimage of c_i. He has to publish $H^{d-\gamma_i}(s_i)$, where $\gamma_i \geq 1$ denotes how many times it has already been selected as a leader (up to and including round x). As before, d denotes the length of the hashchain.

Here, we do not consider the case of $\gamma_i > d$. In this case, $L_x^{(i)}$ is not able to calculate $H^{d-\gamma_i}(s_i)$. A finite amount of steps for the underlying hashchains, i.e. setting the protocol parameter d to 10^9, is enough to ensure that this case is never encountered for any practical purpose. As our tests showed, computing a hashchain of such length during the setup phase is doable in less than 20 minutes on common hardware.[2] It is very unlikely that a participant ever encounters the situation, where no more preimages are available. Consider the following example, in which the number of participants $n = 1000$, $d = 10^9$ and a new random beacon value should be produced every 10 seconds. Here, the expected duration for a participant P_j to be unable to compute $H^{d-\gamma_j}(s_j)$ when asked to do so is given by:

$$\begin{aligned} n \cdot d \cdot 10 \; seconds &= 10^{13} \; seconds \\ &\approx 317098 \; years \end{aligned}$$

5.4.3 Verification of Preimages

Any other participant P_j can easily verify if $L_x^{(i)}$ has revealed a valid preimage. This verification only involves a single invocation of $H(\cdot)$ and, thus, can be computed in

[2] The test was performed using the SHA2-256 algorithm from Python's hashlib package running on a 2.3 GHz Intel® Core™ i5-5300.

constant time. This is possible, because all P_j keep track of the latest valid commitments for all participants. Each participant P_j stores a list of all hashchain heads $\langle h_1, h_2, ..., h_n \rangle$. The term hashchain head is used to refer to the last publicly-known valid value of a participants hashchain. The list $\langle h_1, h_2, ..., h_n \rangle$ is initialized with the commitments $\langle c_1, c_2, ..., c_n \rangle$.

When a leader $L_x^{(i)}$ reveals a candidate preimage ρ_i of his hashchain head, P_j checks the validity of ρ_i by verifying the condition $H(\rho_i) = h_i$. If the condition holds, ρ_i is accepted and h_i is updated accordingly $h_i \leftarrow \rho_i$. Otherwise, ρ_i is ignored and h_i is not updated.

5.4.4 Obtaining the Next Random Beacon Value

The value R_x of the random beacon in round x can be calculated by each participant based on R_{x-1} and the preimage ρ_i of the rounds leader $L_x^{(i)}$:

$$R_x = H(R_{x-1} \mid\mid \rho_i)$$

In case the current leader fails, the required preimage does not become available to the participants. This is also the case if the leader is controlled by an attacker and deliberately withholds the preimage. To prevent the protocol from stalling, we define the value of next random beacon based on the previous one, if no preimage becomes available a specific timeframe:

$$R_x = H(R_{x-1})$$

We stress that this approach only works in the synchronous network setting and potentially leads to biased randomness.

5.5 Summary

In the following, algorithm 5.2 summarizes the hashchain-based approach of building a random beacon from the perspective of a participant P_i. For $n = 1000$ participants and a hashchain depth $d = 10^9$, the protocols is guaranteed to run for $\Omega_{min} = 10^9$ rounds. As a randomly selected participant has to reveal his preimage at each round, the expected number of rounds Ω until a participants runs out of available preimages is close to $\Omega_{max} = n \cdot d = 10^{12}$ rounds.

Algorithm 5.2: A Hashchain-based Random Beacon

Input:

- initial random seed R_0
- the depth of the hashchains d
- a list of commitments $\langle c_1, c_2, ..., c_n \rangle$
- participants i's secret s_i

Output:

- random beacon values R_1, R_2, R_3 and so forth

```
1  ⟨γ_1, γ_2, ..., γ_n⟩ ← ⟨0, 0, ..., 0⟩
2  ⟨h_1, h_2, ..., h_n⟩ ← ⟨c_1, c_2, ..., c_n⟩
3  for x ← 1 to Ω do
4      l ← R_{x−1} mod n
5      L_x ← P_l
6      if L_x = P_i then
7          γ_i ← γ_i + 1
8          R_x ← H(R_{x−1} || H^{d−γ_i}(s_i))
9          broadcast H^{d−γ_i}(s_i)
10         yield return R_x
11     else
12         while ¬timeout do
13             if candidate preimage ρ_l from leader L_x received then
14                 if H(ρ_l) = h_j then
15                     h_j ← ρ_l
16                     R_x ← H(R_{x−1} || ρ_l)
17                     yield return R_x
18                     break
19         if R_x is still undefined then
20             R_x = H(R_{x−1})
```

5.6 Evaluation

In this section, we analyse the hashchain-based random beacon. We discuss various protocol aspects such as computation complexity, message complexity, as well as characteristics of the resulting random beacon values like unpredictability and bias-resistance.

5.6.1 Computational Complexity

We distinguish the computational costs, which have to be performed only once during the setup phase, and the computation, which has to be performed at each round.

Setup During the setup process, each participants has to construct a hashchain of considerable length. This computation has to be performed only once and can be done in less than 20 minutes on common hardware, for the hashchain length $d = 10^9$.

Operation – round's leader During normal operation, the round's leader $L_x^{(i)}$ has to compute the preimage ρ_i of his current hashchain head h_i. A naive approach is to compute ρ_i based on the s_i directly, resulting in a complexity of $\mathcal{O}(d)$.

To improve the performance, participants store intermediate value of their hashchain during the setup process. When storing the values $s_i, H^{1000000}(s_i), H^{2000000}(i), \dots, H^d(s_i)$, P_i has to perform at most 999999 hashing operations to calculate any given intermediate value of his hashchain. Such a calculation can be done in less than 1 second.

For $d = 10^9$, this approach requires only $1000 \cdot 256$ bit $= 3.125$ KB of storage. However, participants might also store additional preimages, i.e. the preimages which need to be revealed in the near future. This further reduces the computations which have to be performed during operation.

Operation – verifier The computational effort required for verification is negligible. A published preimage can be very easily verified by any participants. Only a single invocation of the hash function $H(\cdot)$ is required. Thus, the computational complexity for verification is given by $\mathcal{O}(1)$.

5.6.2 Communication Complexity

During each round x, the round's leader $L_x^{(i)}$ has to broadcast a message containing his next to reveal preimage ρ_i to all other participants. This results in $\mathcal{O}(n)$ messages. For the verification process and for deriving the next value of the random beacon, no further communication is required.

5.6.3 Predictability

By construction, the random beacon values are predictable to some extent. Clearly, a round's leader $L_x^{(i)}$ knows the preimage ρ_i required to calculate R_x. He can perform this

calculation before publishing ρ_i and, thus, knows R_x before it becomes available to the other participants.

In the following, we analyse how predictable the random beacon values indeed are. For this analysis, we consider a set of n participants $\mathcal{P} = \{P_1, P_2, ..., P_n\}$. An attacker controls a subset $\mathcal{A} \subset \mathcal{P}$ of at most f nodes, i.e. $|\mathcal{A}| = f$. We assume byzantine behavior for all nodes in $\mathcal{A}$, i.e any node $A_i \in \mathcal{A}$ might arbitrarily deviate from the protocol. Further, we do not distinguish between failed/offline nodes and attacker controlled nodes. The parameter f includes failed/offline nodes.

We define the probability that the attacker can predict a future random beacon based value on the parameter δ, which describes the number of steps to be predicted:

$$P(\{\text{Given } R_x, R_{x+\delta} \text{ is predictable}\})$$

Obviously, for $\delta = 0$ we have:

$$P(\{\text{Given } R_x, R_{x+0} \text{ is predictable}\}) = 1$$

Prediction is possible in case a node controlled by the adversary is selected as leader. Therefore for $\delta = 1$ we obtain the probability:

$$P(\{\text{Given } R_x, R_{x+1} \text{ is predictable}\}) = P(\{L_{x+1} \in \mathcal{A}\})$$

Similarly, for $\delta = 2$ we have:

$$P(\{\text{Given } R_x, R_{x+2} \text{ is predictable}\}) = P(\{L_{x+1} \in \mathcal{A}\} \wedge \{L_{x+2} \in \mathcal{A}\})$$

In the following, we assume that the attacker cannot influence the random beacon values. The evaluation considering active manipulation of random beacon values is given in section 5.6.5. Considering the assumption holds, the probabilities $\forall w \geq 1$ are given by:

$$P(\{L_w \in \mathcal{A}\}) = {}^{f}\!/_{n}$$

As the events $\{L_{x+1} \in \mathcal{A}\}$, $\{L_{x+2} \in \mathcal{A}\}$, $\dots$, $\{L_{x+\delta} \in \mathcal{A}\}$ are independent, we can obtain the probability of prediction as follows:

$$\begin{aligned} P(\{\text{Given } R_x, R_{x+\delta} &\text{ is predictable}\}) \\ &= P(\{L_{x+1} \in \mathcal{A}\}) \cdot P(\{L_{x+2} \in \mathcal{A}\}) \cdot ... \cdot P(\{L_{x+\delta} \in \mathcal{A}\}) \\ &= \prod_{\theta=1}^{\delta} P(\{L_{x+\theta} \in \mathcal{A}\}) \\ &= \prod_{\theta=1}^{\delta} \frac{f}{n} \\ &= \left(\frac{f}{n}\right)^{\delta} \end{aligned}$$

Figure 5.1 shows that the probability of an attacker, who controls f out of n nodes, predicting δ future random beacon values decreases exponentially for increasing values of δ and any value of $f < 1$.

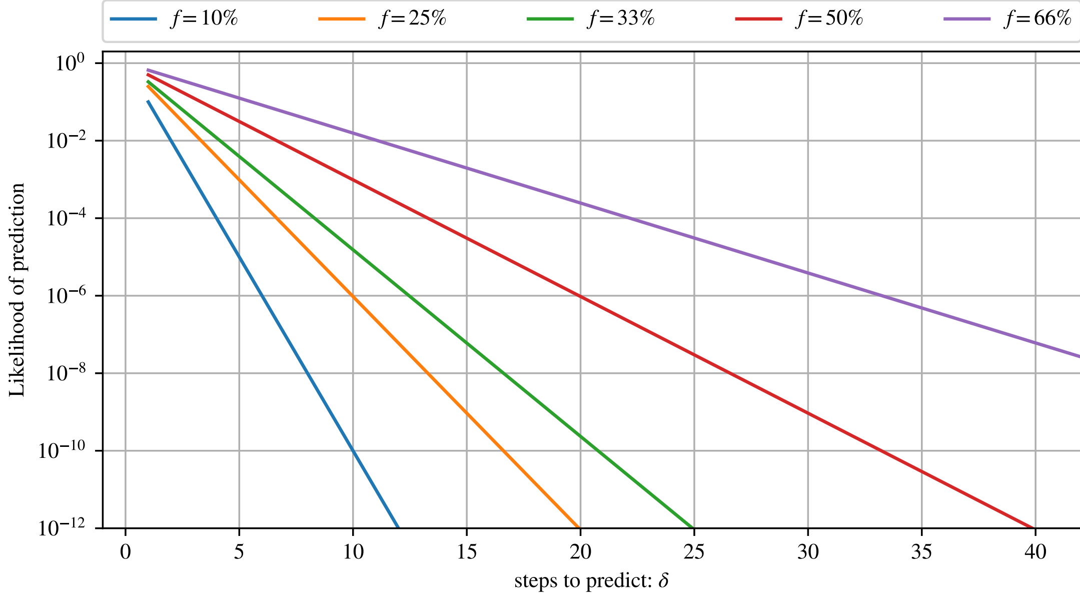

Figure 5.1: Probability of an attacker being able to predict a specific value of the hashchain-based random beacon δ rounds before it is released

However, an attacker might not try to predict the random beacon value starting at a specific round x but might try predicting δ random beacon values over a period of time, i.e. starting the prediction at rounds $x, x+1, x+2..., x+\tau$. Considering such a scenario, we are interested in the probability of a single successful prediction over different periods of time. Assuming the independence of the events in regard to different starting points we calculate:

$$P\left(\{\exists y \in \{x, x+1, ..., x+\tau\} \mid \text{given } R_y, R_{y+\delta} \text{ is predictable}\}\right) = 1 - \left(1 - \left(\frac{f}{n}\right)^{\delta}\right)^{\tau}$$

In the following, we analyse the probability that an attacker is able to predict the value of the random beacon δ rounds into the future at least once over the period of 1) one day, 2) one week, 3) one month and 4) one year. We assume that a new random beacon value is produced every 10 seconds, leading to $\tau_1 = 8640, \tau_2 = 60480, \tau_3 = 259200, \tau_4 = 3153600$. The results, still showing an exponentiation decrease in the probability of prediction, are given in figure 5.2.

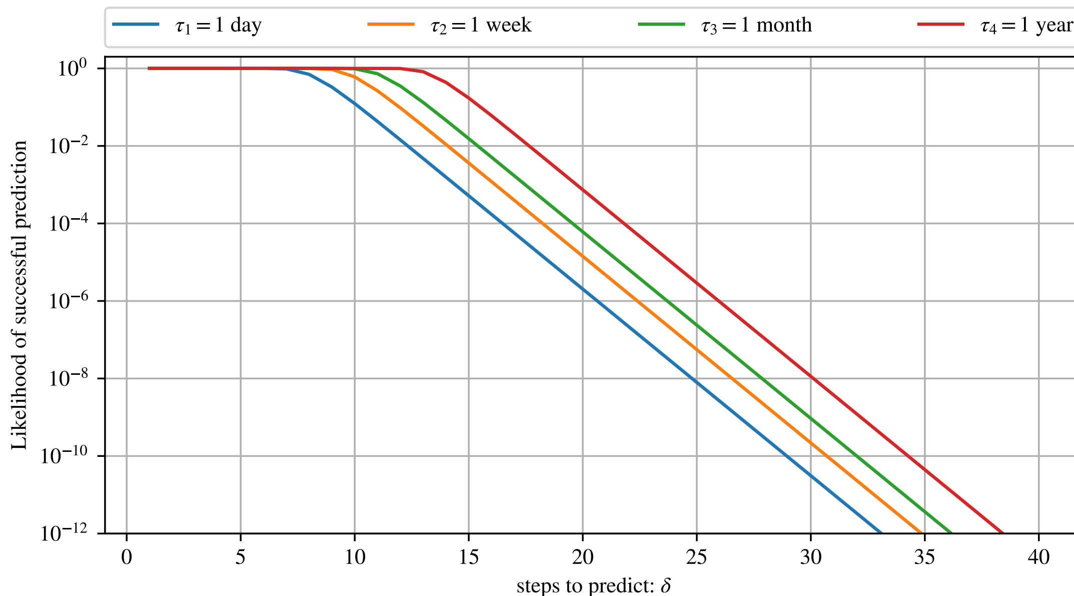

Figure 5.2: Probability of an 33% attacker being able to predict at least one of the hashchain-based random beacon values δ rounds before it is released, considering a continuous prediction effort over different time periods

5.6.4 Bias-Resistance

In the previous section, we analyzed how a colluding passive attacker can predict future values of the random beacon. Thus, we considered a passive attacker. In this section, we consider an active attacker, who not only tries to predict future values of the random beacon, but also actively takes actions to manipulate the beacon values in his favour.

As we discussed in section 5.4.2, a malicious participant selected as a leader $L_x^{(a)}$ for some round x can always choose to:

(1) publish the next preimage ρ_a or

(2) withhold the value ρ_a.

We stress that it, however, cannot introduce arbitrary values into the system after the setup phase due to the properties of the hashchains used.

Depending on his choice, R_x takes one of the values from $\{H(R_{x-1} \mid\mid \rho_a), H(R_{x-1})\}$. Therefore, at each round x, in which a participant $A_i \in \mathcal{A}$ is selected as leader, the attacker can pick one out of two random values.

A simple attack strategy might be to maximize the probability of being selected as leader again in the next round. We consider four cases at round x, where the leader L_x of the particular round is controlled by the attacker.

Let

$$i_1 = H(R_{x-1} \mid\mid \rho_a) \pmod{n}$$
$$i_2 = H(R_{x-1}) \pmod{n}$$

denote the two possible indices of the leader of the next round $x+1$. Then, the four cases are given by

(1) $P_{i_1} \in \mathcal{A} \wedge P_{i_2} \in \mathcal{A}$

(2) $P_{i_1} \in \mathcal{A} \wedge P_{i_2} \notin \mathcal{A}$

(3) $P_{i_1} \notin \mathcal{A} \wedge P_{i_2} \in \mathcal{A}$

(4) $P_{i_1} \notin \mathcal{A} \wedge P_{i_2} \notin \mathcal{A}$

The first three cases are favorable for the attacker. Only the fourth is unfavorable. Combining this cases, we calculate the probability that another attacker controlled node is selected as the next leader as follows:

$$\begin{aligned} P(\{L_{x+1} \in \mathcal{A}\} \mid \{L_x \in \mathcal{A}\}) &= 1 - P(\{P_{i_1} \notin \mathcal{A}\} \wedge \{P_{i_2} \notin \mathcal{A}\}) \\ &= 1 - \left(1 - \frac{f}{n}\right) \cdot \left(1 - \frac{f}{n}\right) \\ &= 1 - \left(1 - \frac{f}{n}\right)^2 \end{aligned}$$

We get a lower bound on the probability that an attacker node is selected as a leader $P(\{L_x \in \mathcal{A}\})$ by iteratively applying the above formula using $P(\{L_1 \in \mathcal{A}\}) = f/n$ as starting point.

The process can also be described by the Markov chain given in figure 5.3 with $\alpha = f/n$ and $\beta = 1 - (1 - f/n)^2$. In this case, we look at the probability of being in the attacker state S_A. Figure 5.4 shows that resulting probabilities quickly converge to values significantly larger than f/n within a few rounds, whereas table 5.1 gives the resulting probabilities in comparison to the expected ones.

As $L_x \in \mathcal{A}$ directly implies the option to bias the resulting random beacon value R_x of the respective round, figure 5.4 also gives the probability that an attacker controlling f out of n nodes can manipulate a specific R_x.

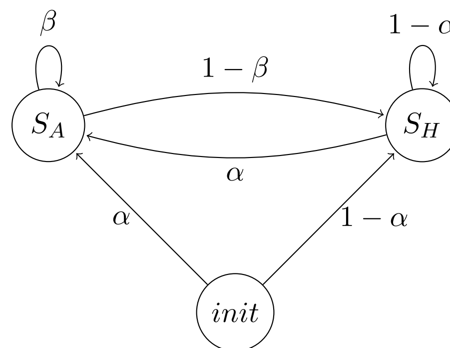

Figure 5.3: Markov chain, illustrating leader selection bias in the hashchain-based random beacon protocol

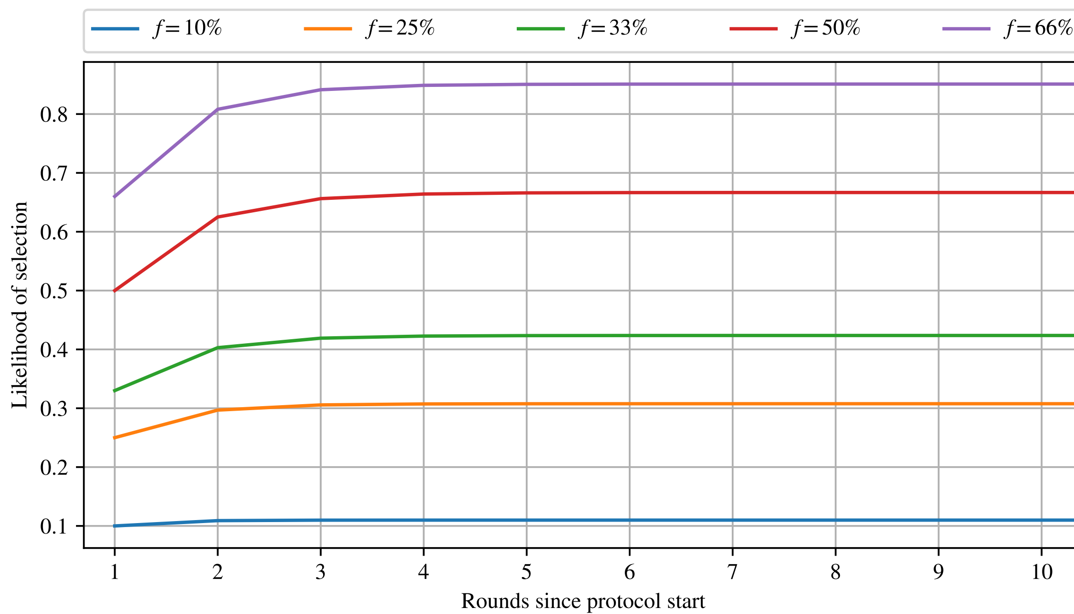

Figure 5.4: Influence of leader selection bias in the hashchain-based random beacon protocol over time

fraction of attacker nodes: f/n	expected $P(\{L_x \in \mathcal{A}\})$	actual $P(\{L_x \in \mathcal{A}\})$
0.1	0.1	0.1099
0.25	0.25	0.3077
0.33	0.33	0.4237
0.5	0.5	0.6667
0.66	0.66	0.8510

Table 5.1: Comparison of the expected leader selection probability and the actual leader selection probability under active manipulation for the hashchain-based random beacon protocol

5.6.5 Predictability under Active Manipulation

In section 5.6.3, we assumed that the attacker does not / cannot influence the random beacon values. As we have shown in section 5.6.4, an attacker can manipulate the random beacon values to a certain degree and can consequently increase (i) the probability of being selected as a leader and (ii) the likelihood of predicting future random beacon values as shown in figure 5.4 and table 5.1. Figures 5.5 and 5.6 show the influence of the random beacon on predictability comparing passive and manipulating attackers.

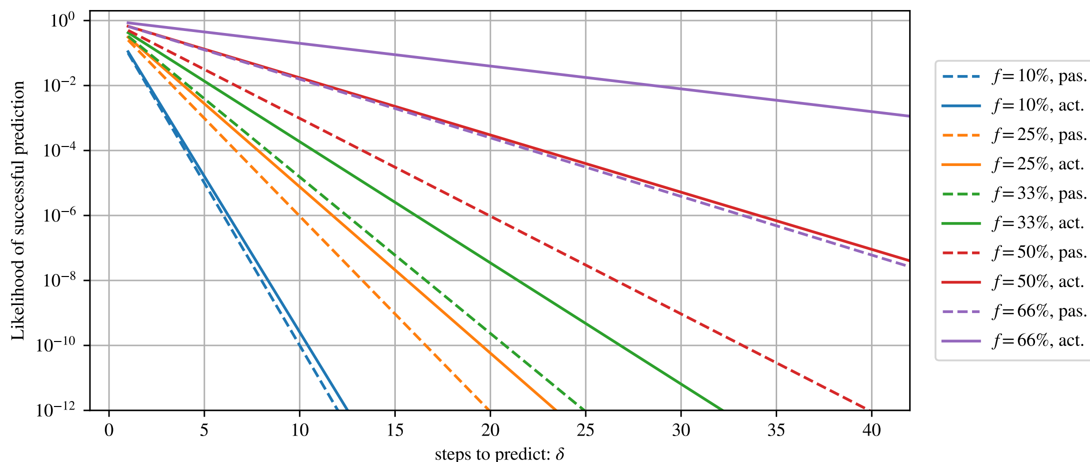

Figure 5.5: Probabilities of passive and manipulating attackers being able to predict specific values of the hashchain-based random beacon δ steps before they are released

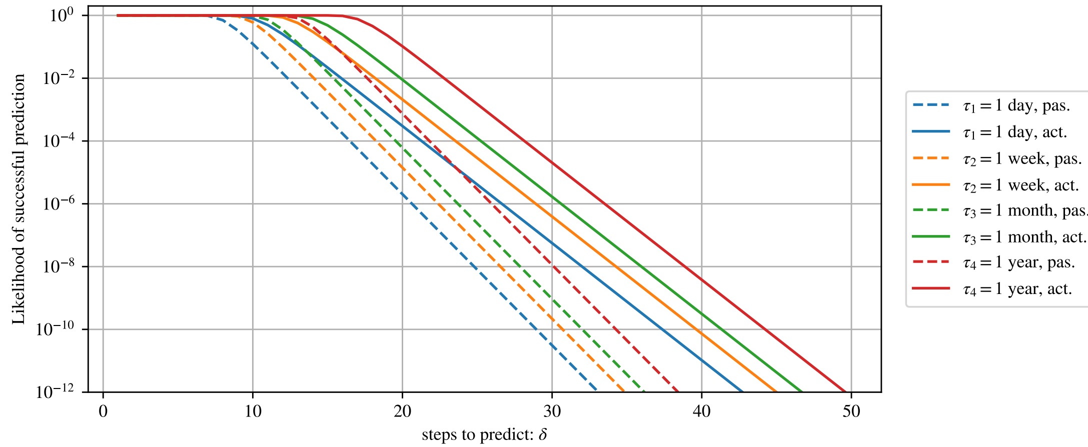

Figure 5.6: Probability of an 33% actively manipulating attacker being able to predict at least one of the hashchain-based random beacon values δ rounds before it is released, considering continuous prediction effort over different time periods

5.6.6 Forks

We only defined the hashchain-based random beacon protocol in a setting of synchronous communication. Therefore, forks cannot arise. As soon as we weaken this property, an attacker might publish his valid preimage ρ_a in a round, in which it controls the leader $L_x^{(a)}$ just before the end of the respective time slot. Due to network latency, parts of the network receive ρ_a and compute $R_x = H(R_{x-1} \mid\mid \rho_a)$, while other nodes do not receive ρ_a in time and thus compute $R_x = H(R_{x-1})$. This consequently produces a fork in the chain of random values. There is no protocol mechanism included that allows to resolve such a situation. Therefore, the hashchain-based random beacon protocol without modifications is not suitable for a setting of asynchronous communication.

5.7 Seed Selection

In the following, we describe and evaluate three different approaches for seed selection and assess their practicability. The outcome of the seed selection process is a seed R_0, which is required after protocol setup. We consider the following approaches:

(1) Select a seed based on a mathematical constant such as π, i.e. define R_0 to be equal to the first 256 bits of the fractional part of π:

```
243f6a88 85a308d3 13198a2e 03707344
a4093822 299f31d0 082efa98 ec4e6c89
```

(2) Deterministically derive a random number from the set of commits $\{c_1, c_2, ...c_n\}$.

(3) Deterministically derive a random number from a publicly available source of randomness such as a future Bitcoin block hash.

Fixed initial seed Approach (1) has the advantage of being very simple with the drawback that an attacker can influence the first values of the random beacon. The attacker P_a could send in his commitment c_a at last. Using the knowledge from the other commitments, the attacker can try a high (but computationally bounded) number of different secrets s_a to obtain favorable / biased value for c_a.

As described in detail in the next 5.4.1, the random beacon values $\{R_0, R_1, R_2, ...\}$ are used for leader selection. By carefully choosing the value c_a with specific properties, an attacker can ensure that it is selected as the leader for the first rounds. This gives the attacker the advantage of knowing the first values of the random beacon before other participants can obtain the values.

Even worse, a colluding attacker, i.e. an attacker who controls multiple participants $P_{a_1}, P_{a_2}, ..., P_{a_t}$, can pick the corresponding commitments $c_{a_1}, c_{a_2}, ..., c_{a_n}$ in a way that it can predict the values $R_1, R_2, ..., R_t$ with minor computational effort. A proof of concept implementation for the attack is available in the appendix A.3.

Seed derived from the commitments Another quite simple alternative, which eliminates some drawbacks of approach (1), is deriving R_0 from the set of commitments $\{c_1, c_2, ..., c_n\}$. Here, n denotes the total number of participants. Let $\langle c_{1'}, c_{2'}, ..., c_{n'} \rangle$ denote the list obtained by sorting $\{c_1, c_2, ..., c_n\}$. Then, R_0 is derived as follows:

$$R_0 = H(c_{1'} \mid\mid c_{2'} \mid\mid ... \mid\mid c_{n'})$$

Similar to approach (1), an attacker can try different values for his commitments. A colluding attacker can select the commitments independent of each other. He is, therefore, able to precompute a chain of random beacon values of at least t elements, where t denotes the number of attackers. This is an important difference to approach (2). Here, changing a single commitment directly influences the seed R_0 and, thus, all future random beacon values. Consequently the attack on approach (1) (see appendix A.3) does not work in this case.

Still, an attacker can try different values for his commitment(s). By investing an *exponential* amount of computational resources, it can however only precompute future values of the random beacon a *linear* number of steps.

Seed derived from a publicly available source of randomness To address the (already very limited) issues of the approach (2), one can choose to derive R_0 from a verifiable public source of randomness. We already described an approach, which generates unbiased randomness with overwhelming probability in section 4.1.4. Although the approach is very slow and resource intensive, it is valuable in the context of seed selection, as the computation has to be performed only once in this case.

CHAPTER 6

PVSS-based Random Beacon

In the previous section, we introduced a random beacon protocol based on hashchains. We discussed the problem that an attacker can selectively reveal and withhold his preimages to bias the generated random values. PVSS, as introduced in section 3.9, can be used to address this issue. In the following, we describe a random beacon protocol, where participants commit to secrets using Schoenmakers' PVSS [20] instead of publishing hashchain heads. The distribution of PVSS shares to the other participants in the system then ensures that previously committed secrets are always revealed by a collaborating set of honest participants. We assume an understanding of the principles of PVSS as well as of the specifics of Schoenmakers' PVSS protocol in this section. The required details on PVSS and Schoenmakers' approach are given in section 3.9.

6.1 Threat and Communication Model

We assume a fixed set of N participants denoted as $\mathcal{P} = \{P_1, P_2, ..., P_N\}$. Of these participants, a set $\mathcal{A} = \{A_1, A_2, ..., A_f\} \subset \mathcal{P}$ of at most f members are considered byzantine and may arbitrarily deviate from the protocol. For the sake of explanation, we further introduce the notation of a dealer $D \in \mathcal{P}$, and define $\mathcal{P}^{\times} = \mathcal{P} \setminus \{D\} = \{P_1, P_2, ..., P_n\}$, where $\mathcal{P}^{\times}$ is used to refer to all nodes receiving PVSS shares from D. Hence, we have $N = n + 1$. Throughout the PVSS-based random beacon protocol, we will use instances of (t, n) PVSS schemes, where the reconstruction threshold $t = f + 1$ is selected in such a way that a colluding attacker, controlling the set of nodes $\mathcal{A}$, can never perform a reconstruction without participation of honest nodes since $t > |\mathcal{A}|$ holds. The network model is synchronous and fully connected. Each participant has authenticated message channels with all other participants and messages are delivered within some time bound.

6.2 Protocol Overview

In the following, we give an overview of our construction for our PVSS-based random beacon protocol. Our protocol provides a fresh randomness value in discrete steps called rounds. The first value R_0 is agreed upon the setup process. In all further rounds, the randomness of the previous round is used to select a leader, whose task it is to reveal a value previously committed to. This value and the value of the random beacon of the previous round get combined using a hash function to form the next random beacon value.

We address the problem of withholding the committed values of a standard commit/reveal approach, which uses cryptographic hashes. For that purpose, we require participants to publish valid PVSS shares as part of all commitments they make. Under normal operation the asked participants reveal their commitments. In case they fail or maliciously withhold their values, the protocol achieves liveness by reconstructing the required values via PVSS.

A key difference to existing PVSS-based protocols is that in our construction only a single PVSS instance is run at each round. Consequently, less messages need to be exchanged to produce a new random beacon value. Nevertheless, our construction ensures that the produced randomness cannot be biased and is unpredictable after $f + 1$ rounds. Our protocol further provides probabilistic guarantees for unpredictability of random beacon values before waiting $f + 1$ rounds.

6.3 Setup Phase

We assume a trusted setup, in which the participants' public keys, as well as a single PVSS commitment for each participant are included in initial dataset B_0. This data is agreed upon by all participants.

PVSS commitment To generate his initial PVSS commitment, every participant, in the following denoted by D, selects a randomly chosen secret s. Participant D then follows the process of share generation and proving share correctness of the PVSS protocol in the role of the dealer, generating shares for all other participants in $\mathcal{P}^{\times}$. Following Schoenmakers' protocol, each dealer D does not actually share the value s but rather G^s. The encrypted shares together with the non-interactive zero-knowledge (NIZK) proof of share correctness form the PVSS commitment. We denote the PVSS commitment from participant D to $\mathcal{P}^{\times}$ as $Com(G^s)$:

$$Com(G^s) = \{\langle y_1^{p(1)}, y_2^{p(2)}, ..., y_n^{p(n)} \rangle, \langle g^s, g^{\alpha_1}, g^{\alpha_2}, ..., g^{\alpha_{t-1}} \rangle, \text{NIZK-proof}\}$$

Here, G and g denote generators of the underlying group used in the PVSS protocol. The polynomial $p(\cdot)$ with coefficients $\langle s, \alpha_1, \alpha_2, ..., \alpha_{t-1} \rangle$ is used for sharing s. The variables y_i are used to refer to the public keys of all participants $P_i \in \mathcal{P}^{\times}$.

Initial random beacon value We additionally assume an agreed random seed R_0 as the first value of our random beacon. In section 5.7, we already described approaches on how such a value might be obtained. We further assume that R_0 is unbiased, selected uniformly at random and becomes public knowledge only after each participant has provided his initial PVSS commitment.

6.4 Operation Phase

The protocol proceeds in rounds. In each round x, a new random beacon value R_x is produced. To give a general overview, this process consists of the following steps:

(1) Uniquely determining the rounds leader $L_x^{(i)}$

(2) Obtaining the committed value G^{s_i} corresponding to the PVSS commitment $Com(G^{s_i})$ from $L_x^{(i)}$ via one of the following two options:

  a) The leader $L_x^{(i)}$ publishes the committed value, together with a new PVSS commitment, which $L_x^{(i)}$ constructs as the dealer.

  b) The committed value is reconstructed by $t+1$ participants. This happens if the leader fails or a malicious leader does not publish the required data on purpose.

(3) Combining the random value from the previous round R_{x-1} and the newly obtained value G^{s_i} to form R_x.

6.4.1 Leader Selection

In each round $x \geq 1$, the randomness of the previous round R_{x-1} determines the unique leader $L_x \in \mathcal{P}$ of round x. We denote the fact that participant P_i is leader of round x as $L_x^{(i)}$.

The random number R_{x-1} is used to simulate a uniformly random selection of the round's leader, drawn from a set of potential leaders $\mathcal{L}_x \subset \mathcal{P}$. For the definition of this set, we refer to the later section 6.4.5, because the definition requires the notion of random beacon blocks, which have not yet been introduced.

Let $\langle l_0, l_1 ..., l_{|\mathcal{L}_x|-1} \rangle$ denote the ordered[1] list of all participants from $\mathcal{L}_x$. Then the round's leader L_x is defined as:

$$L_x = l_i \iff R_{x-1} \equiv i \pmod{|\mathcal{L}_x|}$$

Notice that in our protocol, we do not *elect* a rounds leader but rather *select* one. The round's leader is derived in a deterministic fashion.

[1] The ordering might be obtained based on the participants public keys.

6.4.2 Random Beacon Blocks

After a round's leader $L_x^{(i)}$ is selected, it is his responsibility to produce and broadcast a set of data, a so-called random beacon block B_x. We distinguish between three types of random beacon blocks:

(1) the initial block B_0, constructed during the setup phase,

(2) leader blocks, denoted by B_x (or $B_x^{(i)}$, if we want to describe the fact that participant P_i is the leader of round x producing this block), and

(3) recovered blocks, denoted by b_x.

Random beacon blocks are chained in order to produce a verifiable stream of random beacon values. Figure 6.1 gives an overview of the construction:

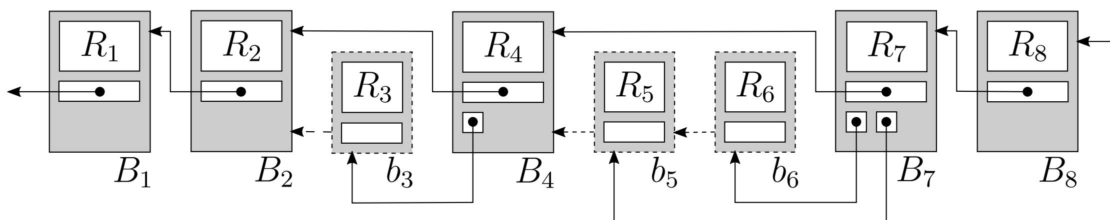

Figure 6.1: Overview of the PVSS-based random beacon

Leader blocks A leader block B_x is a data structure, which contains (at least) the following parts:

(1) The round's index: x

(2) The round's random beacon value: R_x

(3) The hash of the previous leader block: $H(B_{x-u-1})$
Here $u \geq 0$ denotes the number of recovered blocks since the last leader block.

(4) A list of exactly u hashes of recovered blocks since the last leader block: $\langle H(b_{x-u}), H(b_{x-u+1}, ..., H(b_{x-1}) \rangle$.

(5) The value s_i, corresponding to the leader's last PVSS commitment $Com(G^{s_i})$.

(6) The leader's new PVSS commitment: $Com(G^{\tilde{s_i}})$

(7) The set of new PVSS commitments C_x from other participants, which have previously been recovered and consequently failed to provide a PVSS commitment. The exact definition of C_x is given in section 6.4.5.

(8) The blocks signature: $\sigma_{B_x} = sign_{sk_i}(B_x)$
Here, sk_i denotes the private key used by the leader $L_x^{(i)}$ to sign the block.

This data structure can be expanded to also include, for example, proofs of malicious behavior (see sections 6.4.3 and 6.4.5).

Recovered blocks In case a round's leader $L_x^{(i)}$ is offline or does not provide a valid leader block on purpose, any coalition of at least t participants is able to construct a recovered block b_x for this round. A recovered block b_x contains the following parts:

(1) The round's index: x

(2) The round's random beacon value: R_x

(3) A list of t decrypted shares corresponding to the leader's last PVSS commitment $Com(G^{s_i})$

(4) A list of t NIZK correctness proofs for share decryption

Block hashes We define the hash of a block depending on the block type. For a leader block B_x, we compute $H(B_x)$ based on all the data in B_x, i.e. the round's index x, the random beacon value R_x, ..., the signature σ_{B_x}. For recovered blocks b_x, we define $H(b_x) = H(\langle x, R_x \rangle)$. By doing so we deliberately omit the decrypted shares and correctness proofs, or references to previous blocks when computing a recovered block's hash. Therefore, reconstruction by any set of t participants leads to the same block hash. We stress that block hashes are only used for referring to previous blocks, but do not influence the random beacon values in any way.

6.4.3 Generating Leader Blocks

Depending on whether the leader is malicious or not, i.e. $L_x \in \mathcal{A}$ or $L_x \notin \mathcal{A}$, the leader might deviate from the protocol. An honest leader $L_x^{(i)} \notin \mathcal{A}$ performs the following steps to produce leader block B_x:

(1) If $u > 0$, verify the last u recovered blocks $b_{x-u}, b_{x-u+1}, ..., b_{x-1}$.

(2) Verify the last leader block B_{x-u-1}.

(3) Calculate $R_x = H(R_{x-1} \mid\mid G^{s_i})$

(4) Select a new random value $\tilde{s_i}$ and generate a new PVSS commitment $Com(G^{\tilde{s_i}})$.

(5) Construct the new block B_x and sign the block with the long term private key sk_i.

Honest participants only construct and sign a single block B_x per round x. This block is referring to a single valid leader block B_{x-u-1}, and potentially to a single recovered block per round since B_{x-u-1}.

It is however possible that B_x can be constructed in different ways. Such a situation can arise in the following cases:

(1) A malicious leader $L_{x-u-1} \in \mathcal{A}$ distributes various alternatives, but otherwise valid leader blocks e.g. B_{x-u-1}, B'_{x-u-1}, B''_{x-u-1}.

(2) Assume that B_v is the last valid leader block known by $L_x^{(i)}$, and $L_x^{(i)}$ knows a valid recovered block b_v for the same round v. Then, $L_x^{(i)}$ has to decide either to include a reference to B_v or to b_v.[2]

In case, there are multiple choices for the previous leader B_{x-u-1}, there are various solutions to resolve the conflict. The simplest solution is an honest leader $L_x^{(i)}$ picking one of the candidates at random. As an alternative, $L_x^{(i)}$ could refer to all candidate blocks and, therefore, provide a proof of malicious behavior. As a consequence, the malicious actor, i.e. the participant who published B_{x-u-1} and B'_{x-u-1}, could be removed from the set of potential leaders for all future rounds.

6.4.4 Generating Recovered Blocks

In case a round's leader $L_x^{(i)}$ fails or does not provide the leader block B_x on purpose, the recovered block b_x is constructed. Any subset of at least t (honest) participants is able to produce a recovered block b_x collectively. While there might be multiple different recovered blocks for the same round x, the included randomness is equal and not dependent on the specific subset of participants, who have generated the block.

From the perspective of any single participant $P_j \neq L_x^{(i)}$, the process takes one of two paths depending on whether P_j receives a leader block B_x before a timeout occurs. If such a block arrives at P_i in time, P_i does not participate in producing b_x but rather distributes B_x instead. The timeout is selected in such a way that leaders have enough time to generate and distribute B_x during normal operation. In case P_i does not receive a leader block B_x in time, it participates in the construction of b_x, which consists of the following steps:

(1) Broadcast the decrypted secret share $G^{p(j)}$ corresponding to the leader's last PVSS commitment $Com(G^{s_i})$ together with the share's NIZK correctness proof.

(2) Wait until $t-1$ valid decrypted secret shares $G^{p(k)} \mid k \neq j$, broadcasted by other participants, arrive.

[2] Strictly speaking this situation cannot occur in the assumed synchronous communication setting, but is important when using considering partially-synchronous or asynchronous communication.

(3) Reconstruct the shared secret G^{s_i} via Lagrange interpolation using the set of shares $\{G^{p(j)}, G^{p(k_1)}, G^{p(k_2)}, ..., G^{p(k_{t-1})}\}$.

(4) Calculate $R_x = H(R_{x-1} \mid\mid G^{s_i})$.

(5) Construct b_x based on the values of x, R_x, the set of shares and their NIZK correctness proofs.

6.4.5 Potential Leaders

In this section, we give the full definition concerning participants which might be selected as leader for some round x, i.e. the definition of the set of potential leaders $\mathcal{L}_x$.

We restrict the set $\mathcal{L}_x$ to exclude participants $\{L_{x-1}, L_{x-2}, ..., L_{x-f}\}$, which have been leaders in one of the previous f rounds. If a participant is selected as a leader for some round x, it cannot be selected during the rounds $x+1, x+2, ..., x+f$. This ensures that in any period of $f+1$ rounds at least one honest leader is selected.

Further, we can only select a participant P_i as a leader for round x, if P_i has previously submitted an unused PVSS commitment. Under no circumstances is a node selected as leader, which has not provided a PVSS commitment. Under normal operation, honest nodes always submit such PVSS commitments as part of their leader blocks. However, in case P_i fails and the corresponding secret was reconstructed, the new PVSS commitment $Com(G^{\tilde{s_i}})$ of P_i is missing. As soon as P_i is online again, the node broadcasts the missing commitment, which is included in some later block B_y. After f rounds, P_i can again be selected as leader, i.e. $y + f < x$ must hold. Using this idea, we define the set of available commitments $\mathcal{C}_x$ at round x as follows:

(1) Initially, $\mathcal{C}_0$ is the set of all commitments provided during the setup phase.

(2) In every round $x \geq 1$, in which some node P_i is selected as leader, the corresponding commitment $Com(G^{s_i})$ is revealed and, therefore, excluded. In case a leader block B_x is available, P_i's new commitment $Com(G^{\tilde{s_i}})$ is included. Additionally, leader blocks provide a set of new commitments C_x from other participants P_j, which failed to produce a leader block previously:

$$C_x = \{Com(G^{\tilde{s_{j_1}}}), Com(G^{\tilde{s_{j_2}}}), ... \mid Com(G^{s_j}) \notin \mathcal{C}_{x-1}\}$$

The set C_x can only include new commitments $Com(G^{\tilde{s_j}})$ from participants P_j which do not already have a commitment $Com(G^{s_j})$ in $\mathcal{C}_{x-1}$.

Using C_x, we obtain the definition of the set of available commitments $\mathcal{C}_x$ for rounds $x > 0$ as follows:

$$\mathcal{C}_x = \begin{cases} (\mathcal{C}_{x-1} \setminus \{Com(G^{s_i})\}) \cup \{Com(G^{\tilde{s_i}})\} \cup C_x & \text{leader block } B_x \text{ is available} \\ \mathcal{C}_{x-1} \setminus \{Com(G^{s_i})\} & \text{otherwise} \end{cases}$$

We give the full definition of the set of potential leaders $\mathcal{L}_x$ based on the conditions required. A participant P_i is part of $\mathcal{L}_x$ if, and only if, all of the following conditions hold:

(1) The participant P_i was not leader during the last f rounds:

$$P_i \notin \{L_{x-1}, L_{x-2}, ..., L_{x-f}\}$$

(2) The participant P_i has provided a new PVSS commitment in time:

$$Com(G^{s_i}) \in \mathcal{C}_{x-f-1}$$

Therefore, $\mathcal{L}_x$ is given by:

$$\mathcal{L}_x = \{P_i \in \mathcal{P} \mid Com(G^{s_i}) \in \mathcal{C}_{x-f-1}\} \setminus \{L_{x-1}, L_{x-2}, ..., L_{x-f}\}$$

An additional condition applies if we want to consider excluding participants, which have provably shown malicious behavior:

(3) The produced block B_x does not include a (recursive) reference to any block, which proves malicious behavior of participant P_i.

6.5 Verification

The verification process can be performed by any participant $P_i \in \mathcal{P}$ or any other third party with access to the public keys and commitments given in the initial block B_0. By definition, we consider the block B_0 as valid.

In the following, we describe the verification procedure for a leader block B_x. We denote the last known valid leader block, referred to by B_x, as B_{x-u-1}. The number $u \geq 0$ describes the number of recovered blocks $b_{x-u}, b_{x-u+1}, ..., b_{x-1}$ given in B_x. The following list summarizes all steps required to verify B_x:

(1) Check if B_x includes the correct round index x.

(2) Check if B_x contains the hash of the valid previous leader block B_{x-u-1} with round index $x - u - 1$.
(subsection 6.5.1)

(3) Check if B_x contains a list of exactly u hashes $\langle H(b_{x-u}), H(b_{x-u+1}, ..., H(b_{x-1})\rangle$ of valid previous recovered blocks $b_{x-u}, b_{x-u+1}, ..., b_{x-1}$.
(subsection 6.5.1)

(4) Check if B_x contains a valid random beacon value R_x.
(subsection 6.5.2)

(5) Check if B_x includes a valid value s_i, corresponding to the last commitment $Com(G^{s_i})$ of $L_x^{(i)}$.
(subsection 6.5.2)

(6) Check if B_x includes a new valid PVSS commitment $Com(G^{s'_i})$ of $L_x^{(i)}$
(subsection 6.5.3)

(7) Check if the blocks signature $\sigma_{B_{x-1}}$ is valid and produced by the round's leader $L_x^{(i)}$. The required public key pk_i used for signature verification is given as part of protocol setup.

6.5.1 Verification of Recovered Blocks

In case a leader $L_x^{(i)}$ failed to produce a leader block B_x, the proposed recovered block b_x can be verified, given a valid key block B_{x-1}. Using B_{x-1}, we can derive the leader $L_x^{(i)}$ and the leader's last unrevealed commitment $Com(G^{s_i})$. A verifier can then check whether the NIZK share decryption proofs given in b_x are indeed valid in regard to the last PVSS commitment $Com(G^{s_i})$ of the rounds leader $L_x^{(i)}$. If any of the proofs are invalid, b_x is invalid and the verification process terminates.

Otherwise, a verifier can obtain the value G^{s_i} via Lagrange interpolation, following the reconstruction protocol of the underlying PVSS scheme, as given in section 3.9. Based on G^{s_i} and R_{x-1}, a verifier can compute $R_x = H(R_{x-1} \mid\mid G^{s_i})$ by itself and compare the result to the value given in b_x.

Based on the validity of B_{x-1} and b_x, a verifier can check the validity of the next recovered block b_{x+1} using the same verification procedure, which can iteratively be applied to show the validity of the recovered blocks b_{x+2}, ..., b_{x+u}.

6.5.2 Verification of Random Beacon Values

In the following, we give the details for the verification of random beacon values. We recall how R_x is obtained: $R_x = H(R_{x-1} \mid\mid G^{s_i})$, where $Com(G^{s_i})$ refers to the PVSS commitment of the round's leader $L_x^{(i)}$. The value R_x is verified by recalculating R_x based on G^{s_i} and the already verified value of R_{x-1}.

If s_i is not included in B_x, the verification fails and terminates. Otherwise, s_i is checked for correctness by evaluating whether the condition $g^{s_i} = C_0$ holds. Notice that the value C_0 was previously submitted as part of the corresponding PVSS commitment $Com(G^{s_i})$. If $g^{s_i} \neq C_0$, the block B_x is invalid and the verification process terminates. Otherwise, s_i is valid and G^{s_i} can be obtained directly.

Based on G^{s_i}, a verifier can recalculate R_x and compare the result with the value given in B_x. The verification algorithm rejects B_x if the values of R_x do not match.

6.5.3 Verification of PVSS Commitments

To verify whether the new PVSS commitment $Com(G^{s'_i})$ provided by $L_x^{(i)}$ in the leader block B_x is valid, a verifier performs the steps for verification of the underlying PVSS scheme. The steps for Schoenmakers' PVSS are described in detail in section 3.9.

6.6 Evaluation

In this section, we analyse our PVSS-based random beacon protocol. We evaluate and describe the characteristic properties our protocol achieves and outline computation complexity, communication costs, as well as liveness, unpredictability and bias-resistance guarantees.

6.6.1 Computational Complexity

Following the same evaluation approach as introduced in section 5.6.1, we distinguish the computational costs performed only once during the setup phase and the computational costs performed at each round. The exponentiations / point multiplications in the underlying group G_q of Schoenmakers' PVSS scheme are the computationally most expensive part.

Setup During the setup process, each participant has to provide a PVSS commitment via Schoenmakers' PVSS. In this case, the number of exponentiations, which have to be performed during share distribution, are bounded by $\mathcal{O}(n)$.

Verification of a PVSS commitment can be accomplished in $\mathcal{O}(n \cdot t)$ time. However, each participant has to verify n commitments from all other participants. This leads to verification costs of $\mathcal{O}(n^2 \cdot t)$ for each participant. During the setup phase, the amount of time required for verification is not critical. For larger n however, the complexity can be reduced to $\mathcal{O}(n^2)$ by using Scrape's optimization [19] of Schoenmakers' PVSS, as described in section 4.6.

Operation – round's leader During the normal operation of the protocol, the round's leader has to provide a new PVSS commitment. Similar to the setup phase, this can be accomplished using $\mathcal{O}(n)$ exponentiations.

Operation – verification The leader's PVSS commitment has to be verified. For this purpose each participant needs to perform $\mathcal{O}(n \cdot t)$ computations. Again using Scrape's optimization, this number can be reduced to $\mathcal{O}(n)$.

Operation – reconstruction In case a leader fails, his last shared secret has to be recovered. The computations required for reconstruction are bounded by $\mathcal{O}(t)$.

6.6.2 Communication Complexity

During normal operation in each round x, the round's leader broadcasts a single message: the leader block B_x. In case a leader fails or withholds B_x on purpose, the other participants have to exchange $\mathcal{O}(n^2)$ messages to distribute the secret shares in order to produce the recovered block b_x. In both cases, communication complexity in comparison to existing PVSS-based protocol, which requires n PVSS-instances to produce a random beacon value, is reduced from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$.

Figure 6.2 shows the total amount of data transmitted in the network. We assume an adversary controlling up to one third of the nodes in the network and that Schoenmakers' PVSS is implemented via 256 bit elliptic curves. Additional details on the message size calculation and the communication complexity of existing PVSS-based random beacon protocols, are given in section 4.4.3.

For better comparison with the previous evaluation given in figure 4.1, we still use the categories *commit*, *recover* and *reveal* in figure 6.2. The categories commit and reveal combined correspond to communication required for distributing the leader blocks, whereas communication in regard to recovered blocks is covered by the category recover.

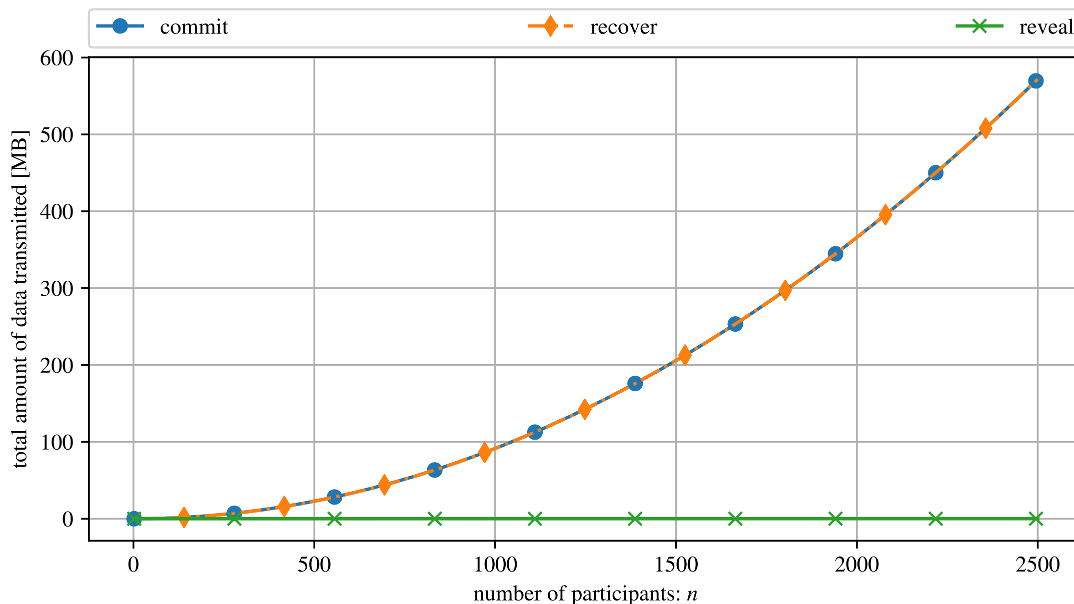

Figure 6.2: Total amount of communication required for our PVSS-based random beacon protocol, assuming an adversary controlling up to 33% of the nodes

A full comparison with existing protocols is given in figure 8.1.

6.6.3 Availability / Liveness

Liveness is ensured as the protocol makes progress in each round. Honest leaders always reveal their values in time, whereas malicious leaders, which withhold their committed values to effect liveness, can always be reconstructed by honest participants. In both cases, the protocol progresses.

6.6.4 Public-Verifiablility

With access to the initial setup data (e.g. R_0, the PVSS commitment and public keys of the participants), any third party can verify the chain of random beacon values R_1, R_2, R_3 and so forth. Values published by the round leaders can directly be checked against their respective PVSS commitments, whereas a shared secret obtained by reconstruction can be verified by checking the respective NIZK proofs for the shares of the secret. The detailed steps required for verification are given in section 6.5. As we discuss in section 6.6.7, forks in the chain of blocks are resolved before the randomness is effected. Consequently, there is one and only one valid value for the random beacon at each round x.

6.6.5 Unpredictability

Similar to the hashchain-based random beacon introduced in chapter 5, the PVSS-based random beacon is also predictable to some extent. Consider the following scenario, in which all participants know the broadcasted random beacon value R_x for some round x. Then, a sequence of malicious leaders $\{L_{x+1}, L_{x+2}, ..., L_{x+\delta}\} \subseteq \mathcal{A}$ is selected. Clearly, $R_{x+\delta}$ is known by the attacker at round x.

We define the corresponding probability as follows:

$$P(\{\text{Given } R_x, R_{x+\delta} \text{ is predictable}\})$$

Unpredictability for $\boldsymbol{\delta} > \mathbf{f}$ For any $\delta > f$, where f denotes the number of malicious participants, this probability is zero, because the leader selection algorithm prohibits that a participant can be selected twice as a leader during $f + 1$ rounds. As a consequence, at least one honest participants is leader during $f + 1$ rounds and, thus, $R_{x+\delta} \mid \delta > f$ is unpredictable per definition.

Probabilistic unpredictability guarantees for $\boldsymbol{\delta} \leq \mathbf{f}$ For $\delta \leq f$, we still have the probabilistic guarantees for unpredictability equivalent to the guarantees provided by the hashchain-based beacon:

$$P(\{\text{Given } R_x, R_{x+\delta} \text{ is predictable}\}) = \left(\frac{f}{N}\right)^{\delta}$$

The corresponding values are given in figure 6.3.

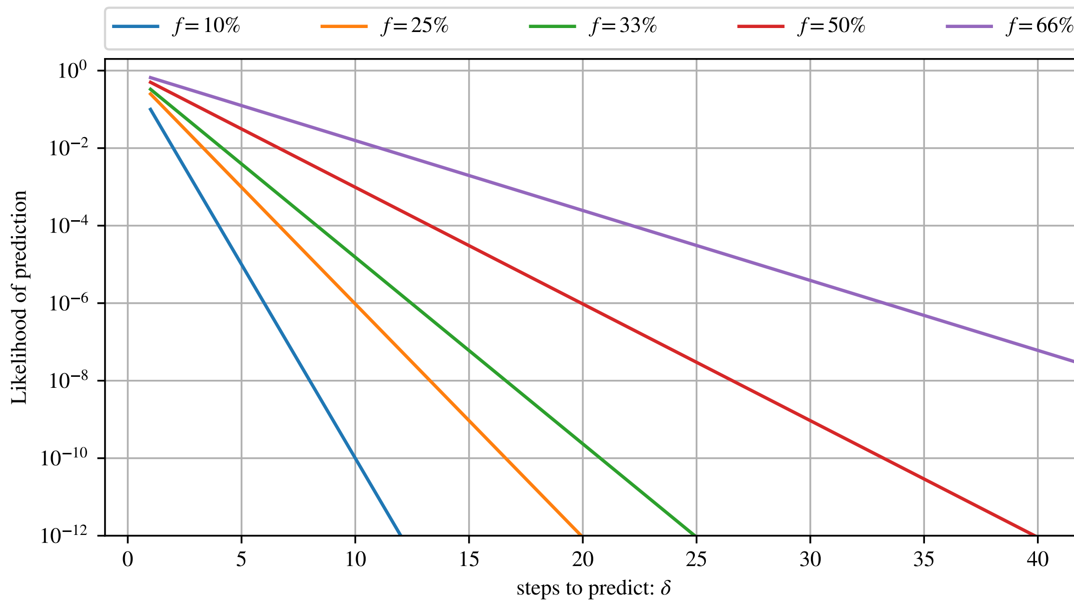

Figure 6.3: Probabilistic guarantees for PVSS-beacon unpredictability – the chart gives the probability of an attacker being able to predict a specific value of the PVSS-based random beacon value δ steps before it is released

6.6.6 Bias-Resistance

In contrast to the hashchain-based approach, the PVSS-based random beacon has very strong bias-resistance. The only points in time when an attacker could try to influence random beacon values is when one of the malicious nodes is selected as leader. Let $x+1$ denote one such round. In this case, the attacker might choose which and how many leader blocks it should publish at round $x+1$. Independent of his decision, the decision can only influence the random beacon value $f+1$ rounds after the decision was made. Round $x+f+2$ is the earliest point in time where L_x might potentially be selected as leader again. As discussed in the previous section, R_{x+f+1} is unpredictable by the attacker. Therefore, the attacker can only blindly decide which decision to take. He does not know in which way it influences R_{x+f+2} at the time $x+1$ the decision is made.

Given that an attacker cannot influence the random beacon values, it cannot influence the probability of being selected as leader. This follows directly from the process of leader selection, which only depends on the produced randomness. Similar to the figure 5.3 provided for the hashchain-based approach in chapter 5, figure 6.4 gives the Markov Chain for the state transitions between honest and malicious leader states. The probability α is given by $\alpha = {}^{f}/_{N}$.

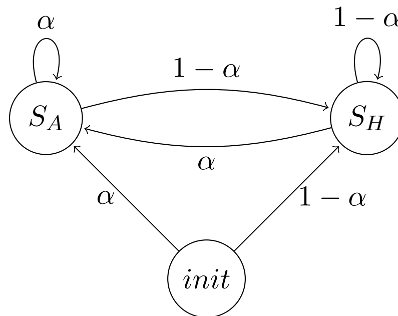

Figure 6.4: Markov chain, illustrating the unbiasability for leader selection in the PVSS-based random beacon protocol

6.6.7 Forks

Similar to blockchains like Bitcoin, we consider forks in the chain of random beacon blocks. A fork of the random beacon blocks in our case means that part of the nodes assume that a block B_x or b_x is the last random beacon block, while other nodes perceive B'_x or b'_x as the last block.

Honest leaders We distinguish two cases depending on whether or not the leader L_x is honest. In case L_x is honest, it timely publishes a single valid leader block B_x, which is received by all other nodes within some fixed time bound. Honest nodes never start to build a recovered block until the time bound is reached. Malicious nodes also fail to produce a valid recovered block, as the participation of at least one honest node is required to produce such a block. Consequently, there is one, and only one, candidate block B_x for round x. As B_x only includes a single reference to a block at height $x - 1$, eventual forks at round $x - 1$ get resolved.

Byzantine leaders A byzantine leader L_x can timely send different (valid) blocks B_x, B'_x or B''_x to the participants instead of broadcasting the same block B_x to all participants. The adversary might also send blocks just before the time bound ends. As a result, some nodes consider the received block, while others start to reconstruct. Honest nodes never participant in a reconstruction at round x if they have received a valid leader block B_x. They instead just forward the leader block. However, the reconstruction might still be successful and various participants could potentially have different opinions on the last block, e.g. B_x, B'_x, B''_x, b_x, b'_x or b''_x. The randomness R_x in all of the valid candidates is, however, equal by construction and the fork is automatically resolved as soon as an honest leader is selected. Therefore, for the selection of an honest leader, the same guarantees as for unpredictability hold. In the worst case, the fork is resolved after $f + 1$ rounds. During this at most f rounds where the state diverges, the randomness is nevertheless the same, as new commitments can only affect the randomness after at least $f + 1$ rounds.

CHAPTER 7

Protocol Extensions

The previously described PVSS-based random beacon protocol improves the amount of data, which need to be exchanged by the participants for producing a sequence of random beacon values to $\mathcal{O}(n^2)$. In our protocol, only *one* participant runs a *single* PVSS instance in each round. This is a notable difference to other PVSS-based protocols such as Ouroboros [12] or Scrape [19], where *each* participant runs the PVSS protocol as the dealer, resulting in a communication complexity of to $\mathcal{O}(n^3)$.

Although the amount of communication required could already be reduced by a factor of n, the resulting message complexity could still be too high for systems with a large number of participants. To further optimize our protocol for such a scenario, we describe two extensions of our protocol in the following sections. While *Quorum Share Distribution* given in section 7.1 explicitly focuses on improving scalability of the protocol, our second extension *Chained PVSS Commitments*, discussed in section 7.2, integrates advantages of our hashchain-based protocol into the PVSS protocol.

7.1 Quorum Share Distribution

To further optimize the protocol in terms of communication complexity, which is in particular important as the number of participants N increase, we introduce a concept called *quorum share distribution*. The aim of this concept is to reduce the number of participants receiving shares as well as the number of participants which need to collaborate for reconstruction. We directly describe the approach in the context of our PVSS-based random beacon protocol. However, the general concept – i.e. performing certain operations only in a randomly selected quorum instead of the set of all participants, while still preserving strong probabilistic guarantees – is applicable to a much wider set of applications. Other existing approaches Algorand [17], Dfinity [58] and RandHound / RandHerd [18] have also used this general idea. Their constructions however use quite different approaches.

7.1.1 Construction Overview

In the PVSS-based random beacon protocol described in chapter 6, the participants commit themselves to values which are revealed in the future. Each participant P_i publishes n shares for his secret value s_i as part of his PVSS commitment $Com(G^{s_i})$. That is one share for each other participant. In case a participant fails to reveal or maliciously withholds the shared secret, an honest group of at least t participants can reconstruct the shared value.

When using this protocol extension, each time a participant commits to a value, the corresponding PVSS shares are only generated for a subset of all participants. We use the term *quorum* to refer to such subsets, which are denoted by the letter $\mathcal{Q}$. The number of participants in a quorum $\mathcal{Q}$ is referred to as the quorum size q_s. The number of quorum member required for a successful reconstruction is denoted as the quorum threshold q_t. All secret sharing operations previously used a (t, n) PVSS scheme. One participant acts as the dealer, while the n other participants receive shares. In the extended protocol, a (q_t, q_s) PVSS scheme with different participants receiving shares at each invocation is run instead.

The quorum is dynamically selected based on the value of the random beacon. It changes every round. We are using $\mathcal{Q}_x$ to refer to the specific quorum of round x. The detailed process of selecting quorum members is given in section 7.1.2. We discuss how the extended protocol achieves liveness as well as unpredictability of the sequence of random beacon values in the sections 7.1.3 and 7.1.4. Both characteristics are ensured by carefully selecting the size q_s and reconstruction threshold q_t for the quorum. The selection process these parameters is specified in sections 7.1.6 and 7.1.7. We further evaluate the benefits the protocol extension provides for scalability in section 7.1.8.

7.1.2 Quorum Selection

The value of the random beacon itself is used to select a quorum for share distribution at each round. We describe the approach for the operation phase, but it can, in a similar manner, also be used during the initial setup process.

Consider the following scenario: The leader $L_x^{(i)}$ of round x publishes a valid leader block B_x. As part of his block, it is required to publish a new PVSS commitment $Com(G^{s'})$. To perform this task, it runs a PVSS protocol as the dealer generating shares for all participants in the quorum $\mathcal{Q}_x$. The round's quorum $\mathcal{Q}_x$ is derived from the round's random beacon value R_x as follows:

(1) Assign each other participant $P_j \neq L_x^{(i)}$ a score $score_{R_x}(P_j) = H(R_x \mid\mid pk_j)$. Here, the public key pk_j is used to uniquely identify a node P_j.

(2) Create a score-ordered list of those participants $\langle P_{j_1}, P_{j_2}, ..., P_{j_n} \rangle$ with $score_{R_x}(P_{j_1})$ being the smallest score.

(3) Select the lowest q_s scoring participants $\langle P_{j_1}, P_{j_2}, ..., P_{j_{q_s}} \rangle$ as $\mathcal{Q}_x$.

The above approach has complexity $\mathcal{O}(n \log n)$, as step (2) requires the sorting of list of length n. In case the number of participants is very high and the computational overhead becomes a problem, the approach given in algorithm 7.1 might be used instead. In this case, the computation complexity can be lowered to $\mathcal{O}(q_s)$.

Algorithm 7.1: Optimized Quorum Selection

Input:

- the canonical representation of the set of all participants: $\langle P_0, P_1, ..., P_n \rangle$
- the quorum size parameter q_s
- the round's random beacon value R_x
- the round's leader $L_x^{(i)}$

Output:

- the set of participants forming the quorum $\mathcal{Q}_x$

```
1 Q_x ← {}
2 k ← 0
3 while |Q_x| < q_s do
4     l ← H(R_x || k) (mod N)
5     if l ≠ i then
6         Q_x ← Q_x ∪ {P_l}
7     k ← k + 1
8 return Q_x
```

7.1.3 Availability / Liveness

Without the protocol extension, protocol progress was ensured as long as the number of honest participants is greater than the PVSS reconstruction threshold t. If we define $t = f + 1$ and $N = 3f + 1$, the set of participants, which need to collaborate during reconstruction, is potentially large. This leads to a high amount of data transfer, as the message complexity for reconstruction without the protocol extension is $\mathcal{O}(n^2)$. To lower this communication complexity, we replace the strong liveness guarantees of the original protocol with a probabilistic guarantee, which still ensures liveness with very high probability.

The protocol parameter $\mathbf{p_{max\text{-}stalling}}$ The maximum probability of stalling at some round is a given by the protocol parameter $p_{max\text{-}stalling}$. Following J. Chen and S. Micali, who considered 10^{-12} as acceptable failure probability for their Algorand protocol [17], we also set $p_{max\text{-}stalling} = 10^{-12}$. The expected time until the protocol stalls is then given

by:

$$\frac{1}{p_{max\text{-}stalling}} \cdot beacon\text{-}interval$$

This number is ≈ 317000 years, if a random beacon value is produced every 10 seconds.

Liveness and protocol stalls The protocol is called alive as long as it is able to produce new values for the random beacon. Otherwise, the protocols stalls. In each round, the protocol is alive if, and only if, one of the following conditions hold:

- The round's leader reveals a valid preimage as part of the corresponding leader block.
- A valid preimage is reconstructed by (honest) participants.

It stalls if, and only if, a (malicious) round leader withholds his preimage and, in addition, the other participants are not able to reconstruct.

Honest and byzantine leaders Honest (online) participants always distribute their leader block to all nodes when they are required to do so. Malicious participants might withhold their blocks, requiring a reconstruction by honest (online) participants. Following our definition of byzantine nodes, we assume all honest participants are online. Without loss of generality, offline nodes can be considered by increasing the fraction of malicious nodes. As before, we denote the set of f byzantine nodes by $\mathcal{A}$. Consequently, the probabilities that a leader L_x at some round x is honest or byzantine are given as follows:

$$P(\{L_x \notin \mathcal{A}\}) = \frac{N-f}{N}$$
$$P(\{L_x \in \mathcal{A}\}) = \frac{f}{N}$$

Ensuring liveness in case of byzantine leaders Whether or not L_x is byzantine, L_x had to provide a PVSS commitment at some previous round w in order to be selected as leader in round x. In case L_x is byzantine, protocol liveness is ensured if the members of the corresponding quorum $\mathcal{Q}_w$ are able to reconstruct the committed value. This is possible if the number of honest nodes in the quorum $\mathcal{Q}_w$ matches or exceeded the reconstruction threshold q_t, i.e. if the condition $|\mathcal{Q}_w \setminus \mathcal{A}| \geq q_t$ holds. As $\mathcal{Q}_w$ was randomly selected, we can use the hypergeometric distribution to obtain the probability that the honest participants in $\mathcal{Q}_w$ are able to perform the reconstruction:

$$P(\{reconstruction\ possible\}) = \sum_{q=q_t}^{q_s} h(q; N, N-f, q_s)$$

Here, $h(k; M, K, m)$ denotes the probability mass function of the hypergeometric distribution with population size M, K good elements in the population, sample size m and k observed good draws.

$$h(k; M, K, m) = \frac{\binom{M}{K}\binom{M-K}{m-k}}{\binom{M}{m}}$$

Likelihood of protocols stalls Combining the two scenarios of honest leaders, in which liveness is ensured by definition, and the case of byzantine leaders, where a reconstruction is required, we can obtain the probability of a liveness failure.

$$P(\{stalling\}) = 1 - P(\neg\{stalling\})$$
$$P(\{stalling\}) = 1 - \left(\frac{N-f}{N} + \frac{f}{N} \cdot \sum_{q=q_t}^{q_s} h(q; N, N-f, q_s)\right)$$

By appropriately selecting the parameter q_s and q_t, we ensure that this probability, denoted by $P(\{stalling\})$, is always lower than the maximum expected failure probability given by the parameter $p_{max\text{-}stalling}$.

7.1.4 Unpredictability

By design, our random beacon protocol ensures unpredictability over time. In particular, the protocol without the extension of Quorum Share Distribution achieves unpredictability of random beacon values after $f + 1$ rounds. In addition, the protocol provides a probabilistic unpredictability guarantee for random beacon values less than $f + 1$ rounds in the future. In the extended protocol, we only want to ensure this probabilistic guarantee.

Unpredictability of the original PVSS-based random beacon protocol reconsidered Assume the random beacon protocol is at some intermediary state in round x. A (colluding) attacker, who is selected as leader for the consecutive rounds $x+1, x+2, ..., x+\delta$, is able to precompute (i.e. predict) $R_{x+\delta}$, the value of the random beacon δ before it is released. Fortunately, the probability of prediction for the PVSS-based protocol, as described in section 6.6.5, decreases exponentially with δ. In a scenario in which at most f out of N participants act malicious, the probability is given by:

$$P(\{\text{Given } R_x, R_{x+\delta} \text{ is predictable}\}) = \left(\frac{f}{N}\right)^{\delta}$$

An upper bound $p_{max\text{-}predictable} = 10^{-12}$, given as protocol parameter, is reached after $\delta = 25$ rounds, assuming the attacker controls one third of the nodes. Figure 7.1 shows

the minimum number of rounds δ required for the probability $P(\{\text{Given } R_x, R_{x+\delta}\})$ to be lower than the specified bound $p_{max\text{-}predictable}$, considering different fractions of attacker nodes.

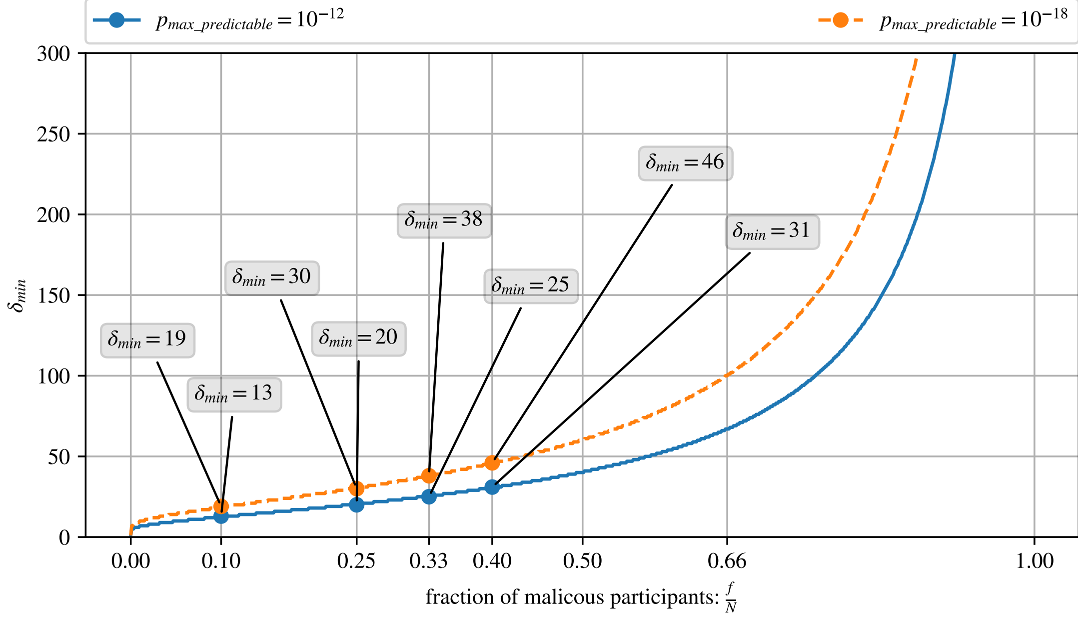

Figure 7.1: Minimal values for the quorum parameter δ_{min} to achieve the desired predictability upper bound of 10^{-12}

The values given in figure 7.1 are the minimum values, considering the original PVSS-based random beacon without the protocol extension. Therefore, we can only aim to find combinations of working quorum parameters q_s and q_t for $\delta^* \geq \delta_{min}$.

Unpredictability in the extended protocol An attacker is able to predict the values of the random beacon δ steps before they are released if, and only if, it can compute all intermediate steps without participation of honest nodes. He can compute a single step in case it is selected as a leader for the respective round, or if it is able to reconstruct the shared secret. This is an important difference to the protocol in chapter 6, as reconstruction was previously only possible with participation of honest nodes. Despite the change, the resulting probability still falls exponentially with increasing values of δ. It is calculated via the hypergeometric distrubution in a quite similar manner as previously described for the probability of protocol failures:

$$P(\{\text{Given } R_x, R_{x+\delta} \text{ is predictable}\}) =$$
$$P(\{predictable\}) = \left(\frac{f}{N} + \frac{N-f}{N} \cdot \sum_{q=q_t}^{q_s} h(q; N, f, q_s)\right)^{\delta}$$

As with protocol failure, the selection of the parameters q_s and q_t ensures that the condition $P(\{\text{Given } R_x, R_{x+\delta} \text{ is predictable}\}) \leq p_{max\text{-}predictable}$ holds.

7.1.5 Bias-Resistance

Bias resistance is ensured in a very similar way compared to the original protocol (see section 6.6.6). We however cannot provide bias-resistance with absolute certainty, as this property crucially relies on the unpredictability of the random beacon values. Without the protocol extension, both properties are a consequence of the leader election mechanism, which ensures that at least one honest node introduces a unpredictable value into the sequence of random beacon values after at least $f+1$ rounds. As any new introduced commitment can only influence random beacon values after f rounds but the value at this point cannot be predicted, bias-resistance is ensured.

For the extended protocol, we can ensure unpredictability with very high probability after δ^* rounds. To also ensure bias-resistance with near absolute certainty, we require that the number of rounds ζ, until the same participant can be selected as a leader again, is at least δ^*. For typical numbers of participants, the above property is already given, because ζ corresponds to f, i.e. $\zeta = f$ in the original description, and is much greater than δ^*. We define any value $\geq \delta^*$ as the protocol parameter ζ and obtain the corresponding probability of a bias-resistance failure similar to the probability of predictability given in section 7.1.4:

$$\begin{aligned}
&P(\{\text{bias-resistance failure}\}) \\
&\quad = P(\{\text{Given } R_x, R_{x+\zeta} \text{ is predictable}\} \wedge \{L_{x+1} = L_{x+\zeta+1}\}) \\
&\quad = \left(\frac{f}{N} + \frac{N-f}{N} \cdot \sum_{q=q_t}^{q_s} h(q; N, f, q_s)\right)^{\zeta} \cdot \frac{1}{N}
\end{aligned}$$

The set of potential leaders, given in section 6.4.5, is adapted to exclude the last ζ leaders instead of the last f leaders. For the above probability, the condition

$$P(\{\text{bias-resistance failure}\}) \leq p_{\textit{max-predictable}}$$

is automatically ensured by setting $\zeta \geq \delta^*$. Additional measures to protect against biasing random beacon values as well as to detect malicious behavior are presented in section 7.2.

7.1.6 Quorum Parameters

The parameters q_s and q_t are selected in such a way that actual protocol stalling and prediction probabilities, denoted as $P(\{\textit{stalling}\})$ and $P(\{\textit{predictable}\})$ for short, are lower than their given parameters $p_{\textit{max-stalling}}$ and $p_{\textit{max-predictable}}$. The definition for $P(\{\textit{stalling}\})$ and $P(\{\textit{predictable}\})$ are given in sections 7.1.3 and 7.1.4. Before describing how the actual values for the quorum parameter q_s and q_t are selected to ensure that maximum protocol stalling and predictability guarantees can be met, we summarize the parameters on which q_s and q_t depend:

(1) The total number of participants N.

(2) The upper bound for malicious participants f.

(3) The protocol stalling probability $p_{max\text{-}stalling} = 10^{-12}$.

(4) The protocol prediction probability $p_{max\text{-}predictable} = 10^{-12}$.

(5) The number of rounds $\delta^* \geq \delta_{min}$ for which the upper bound for predictability should hold:

$$P(\{\text{Given } R_x, R_{x+\delta^*} \text{ is predictable}\}) \leq p_{max\text{-}predictable}$$

Effects of changing the parameters $\mathbf{q_s}$ and $\mathbf{q_t}$ In the following, we introduce the effects of changing q_s and q_t in an intuitive manner:

(1) Increasing q_s, static q_t: As shares are distributed to more participants, $P(\{stalling\})$ decreases. $P(\{predictable\})$ rises as shares are potentially distributed to more (colluding) attackers.

(2) Static q_s, increasing q_t: The collaboration of more participants is required for reconstruction, therefore $P(\{stalling\})$ increases. However, a higher number of colluding attackers is also required to reconstruct, thus $P(\{predictable\})$ decreases.

7.1.7 Optimizing Quorum Parameters

In the following, we present the algorithm, which is used to derive suitable quorum parameters q_s and q_t. The algorithm aims to find a sweet spot between the two effects described above. Using a naive approach, the computation time required to evaluate the actual probabilities for larger values of N and f increases rapidly. Therefore, we present an optimized algorithm to find the best possible parameters based on the idea that if some values for q_s and q_t fulfill both the requirement for unpredictability as well as for protocol liveness, there is no need to check for any $q'_t > q_t$ for the same q_s:

$$\begin{aligned} & P(\{stalling\})[q_s, q_t] > p_{max\text{-}stalling} \\ \implies \forall q'_t > q_t : \; & P(\{stalling\})[q_s, q'_t] > p_{max\text{-}stalling} \end{aligned}$$

This reduces the number of parameters to check. Algorithm 7.2 gives the detailed approach on how the parameters are optimized. The formulas to calculate the probabilities $P(\{stalling\})$ and $P(\{predictable\})$ are given in sections 7.1.3 and 7.1.4 respectively.

Algorithm 7.2: Derivation of Optimal Quorum Parameters

Input:

- the total number of participants N
- the upper bound for the number of malicious participants f
- the target probabilities $p_{max\text{-}stalling}$ and $p_{max\text{-}predictable}$
- the target number of steps δ^* until $P(\{predictable\}) \leq p_{max\text{-}predictable}$ should hold

Output:

- the lowest possible quorum parameters q_s and q_t

```
1  q_s ← 1
2  q_t ← 1
3  while q_s ≤ N do
4      if P({stalling})[q_s, q_t] > p_max-stalling then
5          q_s ← q_s + 1
6      else if P({predictable})[q_s, q_t] > p_max-predictable then
7          q_t ← q_t + 1
8      else
9          return ⟨q_s, q_t⟩
10 return "No solution found!"
```

As above we use 10^{-12} for both of the parameters $p_{max\text{-}stalling}$ and $p_{max\text{-}predictable}$. To compare the resulting values for different fractions of malicious participants, we selected $\delta^* = 38$ for figure 7.2, as it is possible to achieve the upper bound for predictability with all considered fractions of attacker within 38 rounds.

In comparison, figure 7.3 shows the optimized quorum parameters using the different minimum value δ_{min}, depending on the amount of malicious participants. For $f = 10\%, f = 25\%, f = 33.3\%$ and $f = 40\%$ the corresponding values are: $\delta_{min} = 13, \delta_{min} = 20, \delta_{min} = 26$ and $\delta_{min} = 31$. The values for δ_{min} are based on figure 7.1.

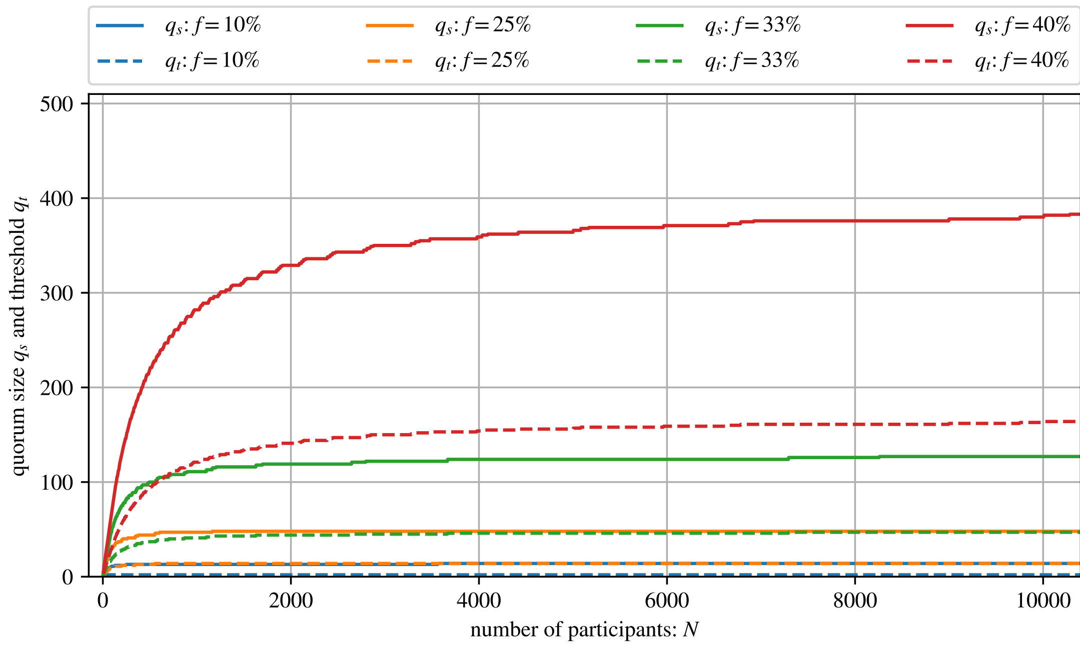

Figure 7.2: Optimized quorum parameters q_s and q_t for $\delta^* = 38$

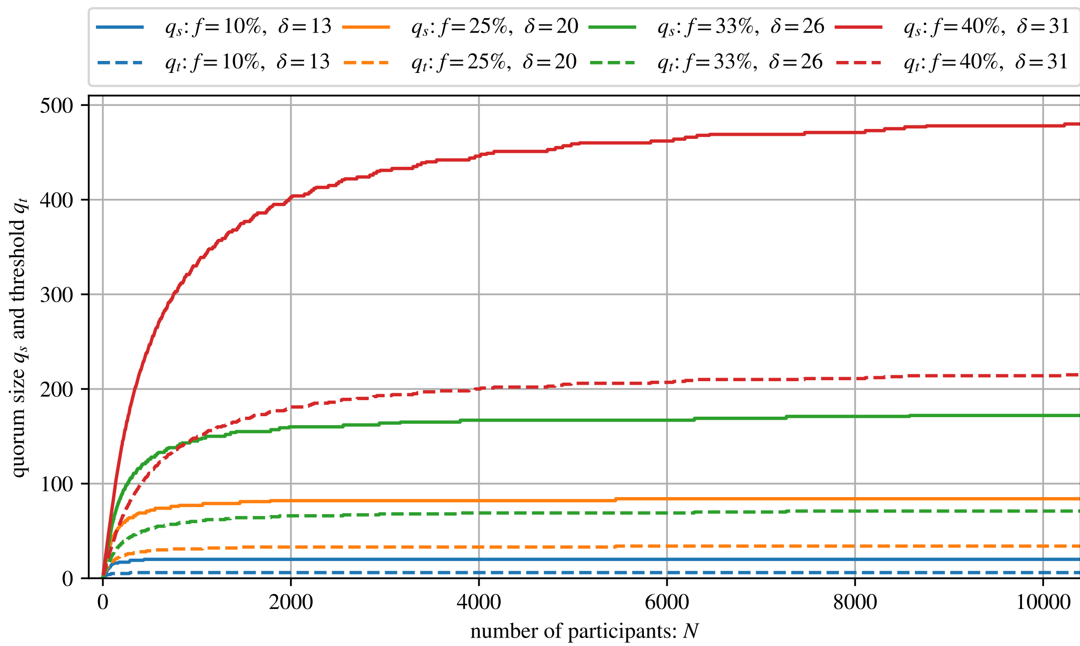

Figure 7.3: Optimized quorum parameters q_s and q_t for δ_{min}

7.1.8 Communication Complexity

Quorum share distribution further improves upon the scalability of our PVSS-based random beacon protocol. As a round's leader has to provide encrypted shares only for a subset of participants, the size of the message it has to broadcast reduces from $\mathcal{O}(n)$ to $\mathcal{O}(q_s)$. The message still has to be broadcasted, thus the complexity for share distribution is given by $\mathcal{O}(n \cdot q_s)$. As we have shown in section 7.1.7, the quorum size q_s is much smaller that n, when considering large values of n.

In case a leader fails, communication complexity for recovery is also reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \cdot q_s)$, as only quorum members (instead of all participants) have to broadcast a single message of constant size in order for all participants to recover the missing shared secret and obtain the next random beacon value.

In both cases, the overall communication complexity of the protocol is lowered from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \cdot q_s)$. Figure 7.4 shows the resulting amount of data transmission required. As before (see sections 4.4 and 6.6.2), we base our figure on message sizes obtained from a 256 bit implementation of Schoenmakers' PVSS. We assume an adversary controlling less than a third of the nodes in the network and use the optimized quorum parameters for $\delta^* = 38$ as given in figure 7.2.

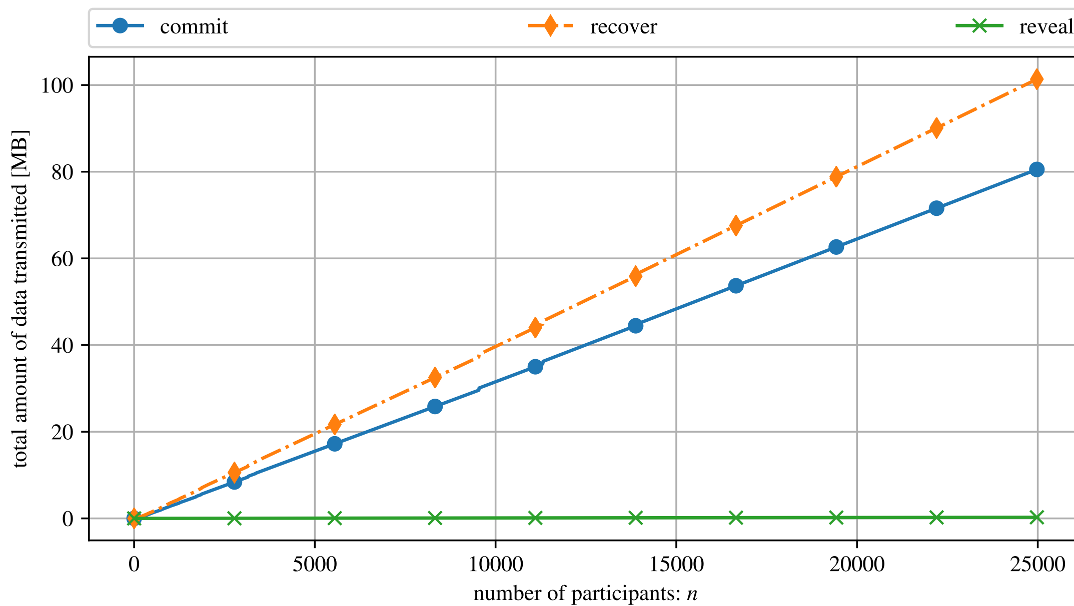

Figure 7.4: Total amount of communication required for our PVSS-based random beacon protocol with quorum share distribution, assuming an adversary controlling up to 33% of the nodes

7.2 Chained PVSS Commitments

In this section, we describe another extension to the PVSS-based random beacon protocol from chapter 6. This extension might be applied together or without the approach of quorum share distribution given in section 7.1.

As we will see in section 7.2.2, the protocol setups shows similarities to both the hashchain-based random beacon as well as the PVSS-based beacon. Its aim is to combine the advantages of both. In particular, the property of committing to all future values during the setup phase, as given in the hashchain-based beacon, is applied to the PVSS-based beacon.

7.2.1 Motivation

In section 6.6, we analyzed the possible options an attacker has to manipulate the random beacon values. The only points in time when a participant can try to manipulate future random beacon are rounds where the participant is leader. The leader L_x of a round x basically has the following options:

(1) Do nothing.

(2) Produce and broadcast an invalid leader block.

(3) Produce and broadcast a valid leader block B_x, including a new PVSS commitment $Com(G^s)$.

For cases one and two, there is no change. Both cases lead to the construction of a recovered block. In the second case, if a valid signature is provided for the block, the protocol might be extended to exclude the corresponding leader from the protocol.

For the third and most interesting case, the leader L_x has the choice to select a particular secret value s to construct different PVSS commitments $\{Com(G^{s_1}), Com(G^{s_2}), ...\}$ (see section 6.6.7 for an extended discussion). In the rare case an attacker (i) controls L_x, (ii) can predict the random beacon value up to round $x + \tau$ and (iii) controls $L_{x+\tau+1} = L_x$, he an manipulate the random beacon value $R_{x+\tau+1}$. This can be accomplished by precomputing $R_{x+\tau}$ and then trying different values for s, which are used for the commitment at round x. As a result, various alternatives for the value $R_{x+\tau+1}$ are obtained. We stress that this attack is not possible with in the attacker model presented in 5.1, because:

(1) The construction of the set of potential leaders ensures that the same participant cannot be leader at two distinct rounds that are less than $f + 1$ rounds apart:

$$\forall \tau < f, L_{x+\tau+1} \neq L_x$$

(2) The value $R_{x+\tau}$ of the random beacon at round $x+\tau$ for $\tau \geq f$ cannot be precomputed, since at least one honest participant's influence on $R_{x+\tau}$ is unknown to the attacker.

However, when applying the protocol extension from section 7.1, these guarantees only hold with very high probability. Further, we might consider different threat models. Instead of defining honest and malicious behavior, we could consider rational actors. In such a scenario, we want to minimize the attack surface as much as possible. With the approach of chained PVSS commitment, we are limiting the choices for the attacker in the described attack. We ensure that a participant publishing a new leader block B_x can only lead to one non-malicious outcome. The extended protocol ensures that only a single value for s is considered valid as the basis for the PVSS commitment $Com(G^s)$. Using any other value for s is detected as soon as the publisher is selected as leader again. In this case, the other participants are certain about the malicious behavior as it can be cryptographically proven.

7.2.2 Setup

The setup process shows similarities to the hashchain-based random beacon. In fact, this protocol extension can be seen as a combination of both the hashchain-based beacon and the PVSS-based beacon.

As described in section 6.3, the initial block includes the participants' public keys as well as a single PVSS commitment for each participant. Such an initial PVSS commitment, previously denoted as $Com(G^s)$, is now constructed differently. We use $Com(G^{v_d})$ to refer to the updated definition for the protocol extension, where v_d denotes the head of a chained data structure. Similar to a hashchain, this data structure is based on some secret seed s and is constructed as follows:

$$
\begin{aligned}
v_0 &= G^s \\
v_1 &= G^{H(v_0)} &&= G^{H(G^s)} \\
v_2 &= G^{H(v_1)} &&= G^{H(G^{H(G^s)})} \\
v_3 &= G^{H(v_2)} &&= G^{H(G^{H(G^{H(G^s)})})} \\
&\dots \\
v_d &= G^{H(v_{d-1})}
\end{aligned}
$$

As in chapter 5, we use d to refer to the length of this of chain of values. As before, the calculation is very fast and can be accomplished for large values of d. It is also not time critical as the process has to be performed only once during the setup phase of the protocol.

7.2.3 Operation

During the operation phase (see section 6.4), it is each participant's task to construct and broadcast a leader block every time they are selected as a leader. As part of this block, a participant also needs to publish a new PVSS commitment. In the extended protocol, a round's leader $L_x^{(i)}$ cannot publish a PVSS commitment of his choice. He rather has to provide the specific commitment $Com(G^{v_{d-\gamma}^{(i)}})$ where γ denotes the number of times $L_x^{(i)}$ has previously been selected as leader (excluding the current round). Here, the list $\langle v_1^{(i)}, v_2^{(i)}, ..., v_d^{(i)} \rangle$ denotes the chained data structure P_i constructed during the setup process.

7.2.4 Verification

A verifier has to check whether the shared secret $G^{v^{(i)}}$, to which a leader $L_x^{(i)}$ has committed itself, is actually part of the leader's chain of values $\langle v_1^{(i)}, v_2^{(i)}, ..., v_d^{(i)} \rangle$. In particular, the verifier has to check whether $G^{v^{(i)}}$ corresponds to the last unrevealed value in the chain.

This can be accomplished as soon as the shared value $G^{v^{(i)}}$ is revealed or as part of next leader block B_x or the next recovered block b_x. A verifier can then check if the published commitment is indeed part of the leader chains by calculating

$$G^{H(G^{v^{(i)}})}$$

and checking whether the result matches the last revealed value in the chain. The above condition is not verified when a participant P_i is selected as a leader for the first time ($\gamma = 0$). In this case, the first shared secret from P_i is used as the base to commit P_i to all the future shared secrets.

7.2.5 Obtaining Random Beacon Values

The values of the random beacon R_x have previously been calculated based on the value of the previous round R_{x-1} as well as the revealed or recovered shared secret for the round's leader. The definition was given by:

$$R_x = H(R_{x-1} \mid\mid G^{s_i})$$

Here, G^{s_i} denotes the shared secret of the leader's previous PVSS commitment.

In the extended protocol, we define the value of R_x dependent on the result of the verification:

$$R_x = \begin{cases} H(R_{x-1} \mid\mid G^{v^{(i)}}) & \text{if verification is successful} \\ H(R_{x-1}) & \text{otherwise} \end{cases}$$

In case the additional verification steps of the extended protocol fails, malicious behavior is proven since $L_x^{(i)}$ previously signed a leader block containing a PVSS commitment to some illegal value. This is also the case if $L_x^{(i)}$ does not publish the block. Using the NIZK proofs provided by the reconstructing participants, the validity of the reconstruction process, which then leads to an illegal value, is ensured.

CHAPTER 8

Discussion

In this section, we discuss the advantages and drawbacks of the presented state-of-the-art protocols as well as outline potential areas of improvements. We start our discussion by providing an overview of the different protocols and their characteristics in section 8.1. In sections 8.2 and 8.3, we provide additional explanations, revisit the key properties our solution provides and compare it to existing protocols. The discussion ends with section 8.4, where suggestions for future research in this area are presented.

8.1 Comparison Overview

In the following, we provide an overview of various properties of the discussed approaches for generating public-verifiable and bias-resistant randomness. For the sake of comparison we are also including the Proof-of-Work variants described in sections 4.1.3 and 4.1.4 as well as our illustrative hashchain-based protocol. Additional details for the assessment of the protocol properties are outlined in chapter 4 for Algorand, Dfinity, RandShare, RandHound, RandHerd, Ouroboros, Scrape, PoW and Iterated PoW, in chapter 5 for our hashchain-based protocol, in chapter 6 for our PVSS-based protocol and in chapter 7 for our extended PVSS-based protocol.

Regarding the comparison provided in table 8.1, we mark a property *prop* as uncertain using the notation ∼*prop* if we have not been able to fully access the property using the available information. For cells marked with **?** we cannot provide an adequate evaluation due to a lack of available information. The symbol ✓ is used to describe that a property is fulfilled, whereas ✗ refers to unfulfilled properties. Additionally, we use (✓) to indicate that a property is achieved with very high probability and / or over time. Further information on specific properties is indicated using the notation $prop_{(1)}, prop_{(2)}, ...$ and given after the comparison table. For the complexity evaluations, n refers to the number of participants in the network and c describes the size of the subset used in the specific protocol. Notice that c is different depending on the protocol.

	Communication model	Liveness / Failure probability	Communication complexity (overall system)	Unpredictability	Bias-Resistance	Computation complexity (per participant)	Verification complexity (per verifier)	Characteristic cyptographic primitive(s)
Algorand	syn.$_{(1)}$	10^{-12} $_{(3)}$	?	(✓)	✗$_{(10)}$	?	$\sim\mathcal{O}(1)$	unique signatures
Dfinity	?	10^{-17}	$\sim\mathcal{O}(cn)_{(5)}$	(✓)	?	?	$\sim\mathcal{O}(1)$	BLS
RandShare	asyn.	✗$_{(4)}$	$\mathcal{O}(n^3)$	✓	✓	$\mathcal{O}(n^3)_{(6)}$	$\mathcal{O}(n^3)_{(6)}$	PVSS
RandHound	~syn.$_{(2)}$	0.08%	$\sim\mathcal{O}(c^2n)$	✓	✓	$\sim\mathcal{O}(c^2n)_{(7)}$	$\sim\mathcal{O}(c^2n)_{(7)}$	PVSS & CoSi
RandHerd	~syn.$_{(2)}$	0.08%	?$_{(11)}$	✓	✓	$\mathcal{O}(c^2 \log n)$	$\mathcal{O}(c^2 \log n)$	PVSS & CoSi
Ouroboros	syn.	✓	$\mathcal{O}(n^3)$	✓	✓	$\mathcal{O}(n^3)_{(6)}$	$\mathcal{O}(n^3)_{(6)}$	PVSS
Scrape	syn.	✓	$\mathcal{O}(n^3)$	✓	✓	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	PVSS
PoW	syn.	✓	$\mathcal{O}(n)$	(✓)	✗	very high$_{(8)}$	$\mathcal{O}(1)$	hash function
PoW iterated	syn.	✓	$\mathcal{O}(n)$	✓	✓	very high$_{(8)}$	very high$_{(8)}$	hash function
Our Hashchain Protocol	syn	✓	$\mathcal{O}(n)$	(✓)	✗	$\mathcal{O}(1)$	$\mathcal{O}(1)$	hash function
Our PVSS Protocol	syn.	✓	$\mathcal{O}(n^2)$	(✓)$_{(9)}$	✓	$\mathcal{O}(n^2)_{(6)}$	$\mathcal{O}(n^2)_{(6)}$	PVSS
Our Ext. PVSS Protocol	syn.	10^{-12} $_{(3)}$	$\mathcal{O}(cn)$	(✓)	(✓)	$\mathcal{O}(c^2)_{(7)}$	$\mathcal{O}(c^2)_{(7)}$	PVSS

Table 8.1: Comparison of approaches for generating public-verifiable randomness

In the following, we provide additional details in regard to the assessment provided in table 8.1:

(1) In Algorand, a custom communication model is specified in great detail. Although synchrony is assumed to some extent by using time bounds, other protocols have stronger synchrony requirements.

(2) The authors of the RandShare, RandHound and RandHerd protocols explicitly state asynchronous communication only for their RandShare protocol. For RandHound and RandHerd, an indication for the requirement of synchronous communication is given. See section 4.5 for an extended discussion.

(3) The exact probability is configurable as a protocol parameter. The given value represents a suggestion by the by the respective authors.

(4) Liveness in the asynchronous communication model is only achieved after a barrier point. Whether or not this point is reached depends on the outcome of a byzantine agreement protocol, which RandShare uses as a subprotocol (see section 4.5). Assuming synchronous communication instead, liveness is ensured.

(5) Due to a lack of information, we can only estimate the communication complexity. Assuming that the only communication strictly necessary to produce the random beacon values is the broadcast of partial signatures, which each member of the correct group has to perform, the complexity $\mathcal{O}(cn)$ can be derived.

(6) Using the optimization of Schoenmakers' PVSS proposed by the authors of the Scrape protocol, the complexity can be further reduced by a factor of n.

(7) Again using Scrape's optimization, the complexity can be reduced. Since the PVSS protocol is executed among a subset of participants, a reduction by a factor of c is possible.

(8) The complexity is not dependent on the number of participants and hence is actually $\mathcal{O}(1)$ in terms of participants. However, as PoW is inherently computation intensive, the notation of $\mathcal{O}(1)$ would be misleading in comparison to other schemes.

(9) Our protocol reaches unpredictability with absolute certainty after $f + 1$ rounds. Before that point, the protocol can only provide unpredictability with increasingly high probability (see section 6.6.5).

(10) For Algorand, bias-resistance is not achieved because the corresponding leader selection algorithm does not ensure leader uniqueness. Further, malicious leaders can selectively withhold values to bias the produced randomness.

(11) According to our interpretation, RandHerd's communication complexity $\mathcal{O}(c^2 \log n)$ is stated per server only. Therefore, this value is not comparable to the other approaches, which consider the communication complexity of the overall system.

8.2 Comparison of PVSS-based Random Beacon Protocols

We follow our discussion with the comparison of our existing PVSS-based random beacon protocols to our PVSS-based solution. During our evaluation of the PVSS-based random beacon protocols in Ouroboros, RandShare and Scrape, we discovered that those protocols have a very similar overall structure. In the following section, we highlight these similarities, advantages and disadvantages and afterwards compare them to our approach.

8.2.1 Existing PVSS-based Random Beacon Protocols

In Ouroboros and Scrape, the use of *publicly-verifiable* secret sharing is explicitly stated. The authors of the RandShare protocol do not only explicitly mention *verifable* secret sharing for RandShare, but also describe the primitive of *publicly-verifiable* secret sharing and use it for the protocols RandHound and RandHerd. Ignoring this minor distinction for RandShare, the overall protocol structure for all three protocols is as follows:

(1) **Share distribution:** Every participant runs an instance of PVSS to distribute encrypted shares of a secret value to all other participants. The set of encrypted shares a participant distributes is called the participant's commitment.

(2) **Share verification:** Each participant verifies the encrypted shares it received using the included NIZK proofs.

(3) **Share agreement:** The participants need to agree on a set of at least $f + 1$ commitments. The use of $f + 1$ or more commitments ensures that the secret of at least one honest participant is used. In the synchronous communication setting, this problem can be resolved easily. Ouroboros and Scrape assume a common agreed broadcast channel. In the asynchronous setting considered by RandShare, a BFT agreement protocol is run.

(4) **Share revealment:** Honest participants (in the agreed set) disclose their shared secret to all other participants. Correctness of the secrets can be verified against the corresponding participant's commitment.

(5) **Share recovery:** Missing secrets for all participants in the agreed set get recovered. This is always possible as the number of honest nodes is by assumption greater than the PVSS reconstruction threshold.

(6) **Randomness recovery:** All revealed / recovered secrets in the agreed set are combined, for example by using a cryptographic hash function.

In the following, we summarize the advantages and issues of the three protocols.

8.2.2 Advantages of Existing PVSS-Beacon Protocols

Given the underlying assumptions hold, the protocols achieve **optimal unpredictability** and **optimal bias-resistance**. Further, the results are **publicly-verifiable** and the protocols use **only established cryptographic primitives**.

8.2.3 Disadvantages of Existing PVSS-Beacon Protocols

Major drawbacks of the approaches are the high communication complexity, as a result of running n instances of a PVSS protocol for producing a single random beacon value. Assuming peer-to-peer communication, each participant needs to send messages to all other participants, resulting in $\mathcal{O}(n^2)$ messages. For the commitment, the message size is given by $\mathcal{O}(n)$, which leads to an amount of $\mathcal{O}(n^3)$ data transmitted. Consequently, these protocols have **bad scalability** as the number of participants increases. In case asynchronous communication is assumed (RandShare), the presented protocol only produces an output after a barrier point is reached (see section 4.5.2 for details). Consequently, liveness is not ensured under asynchrony.

8.2.4 Comparison to our PVSS-Beacon Protocol

The aim of our protocol is to improve upon the bad scalability properties of existing approaches, while still retaining their advantages. We do not change the underlying assumption in regard to the communication model and use the same established cryptographic primitives for our construction. If not explicitly stated otherwise, our protocol provides the same guarantees as existing approaches. We consider the properties of our protocol as described and evaluated in chapter 6, as well as our protocol in combination with the developed protocol extension *quorum share distribution* from section 7.1, which we refer to by the term *extended protocol*.

Bias-resistance and Unpredictability in our Protocol Still, the properties of optimal bias-resistance is preserved and unpredictability is ensured with absolute certainty after $f+1$ rounds, where f specifies the number of malicious nodes. In addition to this unpredictability guarantee our protocol provides a probabilistic one, which ensures that the probability of predicting future random beacon values decreases exponentially over time. Independent of the number of of nodes, unpredictability with very high probability, is ensured after 25 rounds, considering an adversary controlling 33% of all nodes. Additional details are given in section 6.6.5.

Bias-Resistance and Unpredictability in our Extended Protocol As a tradeoff for better message complexity, the strict unpredictability guarantee after $f+1$ cannot be achieved when applying the quorum share distribution protocol extension. Still unpredictability is ensured with very high probability. The corresponding failure probability is defined using the protocol parameter $p_{\textit{max-predictable}} = 10^{-12}$. As bias-resistance is ensured by unpredictability in our protocol, the probabilistic guarantees also apply to bias-resistance. For both properties, we outline and evaluate the resulting guarantees in more detail in section 7.1.

Availability / Liveness in our Extended Protocol An additional tradeoff of quorum share distribution is that availability can only be ensured with very high probability. In comparison, existing protocols and our protocol without the extension always provide availability (given the underlying assumptions hold). However, we also ensure liveness in the extended approach based on the protocol parameter $p_{max\text{-}stalling} = 10^{-12}$, which defines the probability of a protocol stall. For the given value, the expected duration until a protocol stall happens under adversarial conditions is ≈ 317000 years.

Communication Complexity in our Protocol For our PVSS-based random protocol described in detail in chapter 6, a noticeable difference is share distribution. In each round, only one participant, i.e. the round's leader, distributes PVSS shares to the other participants, whereas each participant has to perform this task in existing schemes. As a consequence, the communication complexity for the system reduces from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$.

Communication Complexity in our Extended Protocol Using the described protocol extension from section 7.1, communication complexity is further reduced. In each round, the leader has to broadcast his PVSS commitment of size $\mathcal{O}(q_s)$. This results in a complexity of $\mathcal{O}(n \cdot q_s)$ for share distribution. Here, q_s refers to the quorum size. As shown in figures 7.2 and 7.3, q_s quickly converges and can thus be considered constant for large numbers of participants. In case of an honest leader, a new random beacon is obtained by revealing the previously committed value, hence the optimal communication complexity of a single broadcast $\mathcal{O}(n)$ is achieved. Even in the worst case, i.e. the quorum has to collaborate to recover the shared secret, the complexity is bounded by $\mathcal{O}(n \cdot q_s)$ for the whole system.

In the following, figure 8.1 shows a comparison of the amount of data transfer required in the overall network to produce a new random beacon value. The figure is based on the evaluation of the communication complexity of existing PVSS-based protocols given in section 4.4.3, the evaluation of our protocol (see section 6.6) and the evaluation of the extended protocol (given in section 7.1.8).

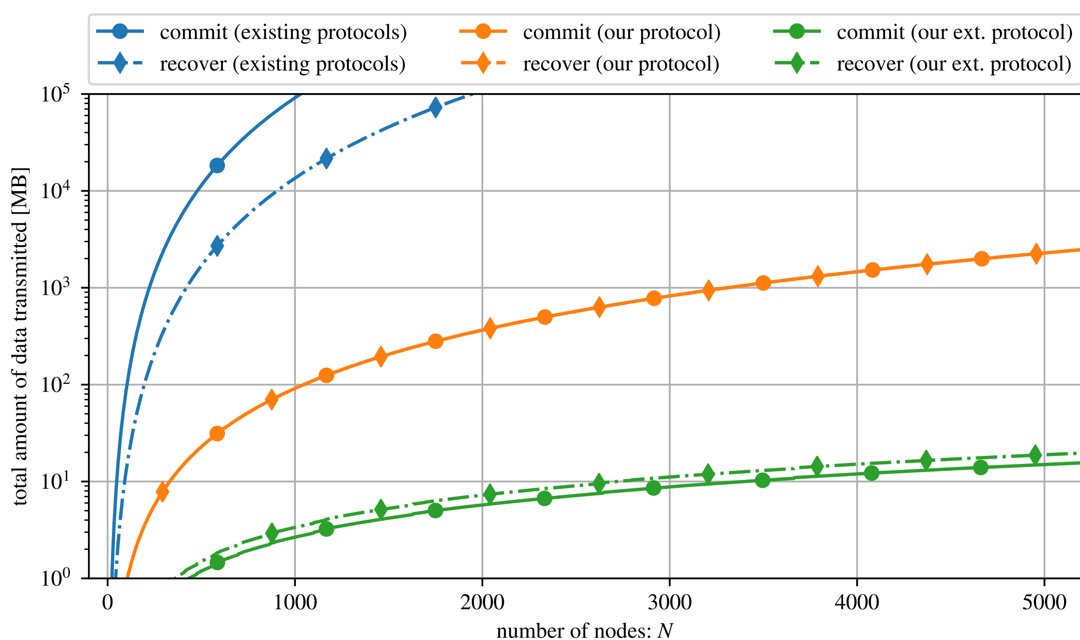

Figure 8.1: Comparison of communication complexity of existing PVSS-based protocols, with our protocol and our extended protocol

8.3 Our PVSS-based Random Beacon Protocol in Relation to other Existing Approaches

In the following, we compare our solution to existing approaches, namely Dfinity [58], Algorand [17], RandHound [18] and RandHerd [18]. In contrast to our proposal, those existing approaches are not using PVSS (Dfinity, Algorand) at all or use PVSS in a quite different construction (RandHound and RandHerd).

Probabilistic Liveness Guarantees All four protocols share probabilistic liveness guarantees with our extended protocol. In the following, table 8.2 compares the characteristics given by the authors. For Dfinity, RandHound and RandHerd, we give the failure probabilities for the typical scenarios outlined, while the resulting failure probabilities are protocol parameters for Algorand and our proposal, and thus can be defined according to the specific use case.

	Protocol failure probability
Dfinity	10^{-17}
Algorand	10^{-12}
RandHound	0.08%
RandHerd	0.08%
Our Protocol	10^{-12}

Table 8.2: Comparison of protocol failure guarantees

Existing PVSS protocols as well as our protocol without the extension of quorum share distribution provide liveness without relaying on probabilistic guarantees.

Bias-Resistance Full bias-resistance is only provided by RandHound, RandHerd as well as our protocol. Our extended protocol ensures bias-resistance with very high probability. For Dfinity, bias-resistance (as well as liveness) crucially depends on the attackers ability to manipulate the assignment of nodes into groups. As described in more detail in section 4.2.7, assessment of this property is hard due to lack of available information. To the best of our understanding, we believe that Dfinity's protocol ensures bias-resistance. For Algorand, bias-resistance is not given. A malicious leader is able to pick between two (and potentially more) potential random beacon values for a single round. A more detailed evaluation on why Algorand by design fails to provide bias-resistance is given in section 4.3.5.

Unpredictability While all protocols provide unpredictable randomness, Dfinity, RandHound, RandHerd and existing PVSS-based protocols manage to ensure unpredictability of the next random beacon value. Our protocol provides unpredictability with absolute certainty after $f + 1$ rounds, and gives probabilistic guarantees otherwise. Algorand provides probabilistic guarantees similar to our extended protocol.

Verifiability All considered protocols allow for public verification, i.e. verification from third parties, which are not necessary participants in the system. However, the systems can be distinguished by the computational effort required for verification. Algorand, Dfinity, RandHerd and our illustrative hashchain-based protocol have an advantage, as they only require a minor amount of computation. For PVSS-based protocols, the verification effort depends on the group size and can be significantly lowered by using Scrape's [19] optimization of Schoenmakers' PVSS, described in section 4.6.

Cryptographic primitives Existing PVSS-based protocols as well as RandHound, RandHerd and our proposed protocol rely on Schoenmakers' PVSS, digital signatures and cryptographic hash functions as cryptographic primitives. RandHound and RandHerd additionally rely on Schnorr Threshold Signatures. Algorand does not require PVSS but a unique signature scheme. The authors however do not give additional details on which digital signature scheme to use. Pairing-based digital signatures, such as the BLS signatures, provide signature uniqueness and, thus, might serves as a candidate for instantiation. Additionally, BLS signatures are used as the fundamental building block for Dfinity's random beacon protocol.

Scalability To provide better scalability, Dfinity, Algorand, RandHound, RandShare as well as our extended protocol try to reduce the number of participants which need to perform certain operations. In Dfinity, the participants are assigned to groups, which then collectively produce a new random beacon value by computing a threshold signature. Algorand randomly selects a smaller set of verifiers, which (instead of the total set of participants) run a byzantine agreement protocol to confirm a claimed block. RandHound and RandHerd split the set of participants into smaller disjunct groups. Each participant then runs a PVSS only with the other member of its group. The pigeonhole principle then ensures that the result, which is combined using values from all groups, is not manipulable and can be considered random. RandHerd requires a round of RandHound during setup and improves upon the performance of RandHound for additional invocations.

In our extended protocol, a round's leader distributes his commitment to a randomly selected group of participants. This set, referred to by the term quorum, ensures that the committed value is revealed if the leader fails at a later point in time or behaves byzantine. In the typical case of an honest leader, the leader itself reveals his committed value leading to optimal performance in this scenario, requiring only a broadcast of a single message of constant size.

8.4 Future work

During our extensive study of existing protocols as well as throughout the design of our protocols, we gained deep insights on the topic of verifiable randomness generation in decentralized systems. In this section, we want to discuss our findings in regard to open questions and potential areas for further improvements.

8.4.1 Communication Model

With the exception of RandShare protocol, which is not scalable due to the high communication complexity, and the Dfinify project, where no detailed information on the communication model is available, all analysed protocols rely on the assumption of synchronous communication to some extent. In a synchronous communication model, messages between honest nodes are delivered within some fixed time bound. Protocols like Ouroboros, Scrape or our illustrative hashchain-based random beacon protocol crucially rely on the assumed time bounds. Other protocols, in particular Algorand, for which the authors give a very detailed description of the underlying communication model, considerably weaken the synchrony assumptions and only require probabilistic bounds on the delay and transmission guarantees of messages between honest participants.

Still, underlying timing assumptions remain and are potential subject for violation. For example, Miller et al. [70], as part of their work on an asynchronous BFT protocols, argues in favor of asynchronous protocols, which do not rely on timing assumption of the underlying communication network. In particular, they answer the question of *"why weakly synchronous BFT protocols can fail (or suffer from performance degradation) when network conditions are adversarial (or unpredictable)"* [70], and motivate the use of asynchronous protocols in blockchain-based systems. Consequently, we argue that the extension / modification of existing random beacon protocols to support fully asynchronous communication is an important area for future research.

8.4.2 Construction of a Full Fledged Distributed Ledger

As described in sections 4.2, 4.3 and 4.4, existing state-of-the-art protocols already use verifiable randomness as a fundamental building block for (Proof-of-Stake based) blockchains. However, none of the protocols have been deployed in a large scale setting. Other designs, such as the improvements by Cascudo et al. [19] in Scrape, the RandHerd protocol by Syta et al. [18], as well as our protocol, are described in a standalone setting and have not yet been used to build full fledged distributed ledgers. Therefore, we strongly advocate additional research in this area, which should not only focus on further improvements of the random beacon protocols themselves, but also on the overall construction and integration with distributed ledgers.

8.4.3 Integration into Existing Blockchains

As already outlined in section 1.3, publicly-verifiable bias-resistance randomness is particularly important for Smart Contracts, for example in the Ethereum blockchain. In this case, the access to a trustworthy source of randomness is severely restricted due to the use of a deterministic execution environment.

Integration of a random beacon protocol, for example the inclusion of the successive random beacon values in the block header, would allow Smart Contracts to access randomness in a trust-less manner without the need to change the guarantees a deterministic execution environment provides. Consequently, the problem of using insecure sources of randomness (e.g. the blockhash or the reliance on external random oracles) could be resolved.

CHAPTER 9

Conclusion

The introduction of Proof-of-Work, as the underlying technology for establishing consensus in decentralized systems like Bitcoin, enabled scalability to thousands of miners. The inherent problem of very high resource consumption, as a direct consequence of solving the computational puzzles involved, nevertheless remains. To address this issue, various alternatives including Proof-of-Stake have been proposed. Proof-of-Stake, by construction, does not require a high amount of computational resources. Instead, virtual resources (i.e. cryptographic money) are used. As, for example, identified by Kiayias et al. [12], *publicly-verifiable and bias-resistant randomness* is required as underlying building block for blockchains based on Proof-of-Stake. Various other use cases for trustworthy randomness, including lottery services, publicly-auditable selections, generation of cryptographic parameters and traditional byzantine consensus, have been identified [18, 21, 23].

In this thesis, we have been the first to provide an extensive review and comparison of state-of-the-art protocols in this domain. We analyzed both *random beacon protocols* (Scrape, RandShare, RandHound, RandHerd), which are built for the purpose of generated verifiable randomness, as well as protocols, which require verifiable randomness as part of their overall construction (Dfinity, Algorand, Ouroboros). We identified publicly-verifiable secret sharing (PVSS), in particular Schoenmakers' PVSS, as a common building block for the randomness generation in Scrape, RandShare, RandHound, RandHerd and Ouroboros. In addition to traditional primitives like cryptographic hash functions and digital signatures, we showed how Algorand and Dfinity employ unique signatures and pairing-based cryptography (BLS signatures) to obtain verifiable randomness.

In our evaluation, we illustrate that Algorand's randomness is not suitable for scenarios where bias-resistance is required. Dfinity, a project which relies on distributed key generation for their random beacon, has not published detailed information on how this crucial component should be implemented. Using the scenario given by the authors of RandHound and RandHerd, the protocol failure probability of their protocols shows a disadvantage compared to existing approaches like Algorand and our proposal, which

allows the definition of an acceptable failure probability of 10^{-12} or 10^{-18} as protocol parameter. Our comparison further outlines that PVSS-based random beacon protocols, which require each participant to exchange shares with all other participants (RandShare, Ouroboros, Scrape), provide good properties in regard to the quality of the produced randomness, but fail to scale to a large number of participants.

This finding leads to another major contribution of this thesis, a new PVSS-based random beacon protocol with greater focus on scalability. Our protocol improves the communication complexity of existing approaches from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$, while still providing the key properties of availability, bias-resistance, unpredictability and public-verifiability.

We advance our protocol using a concept called *quorum share distribution*. The extended protocol still achieves the desired protocols characteristics and gives strong probabilistic guarantees with negligible and parameterizable failure probability. Quorum share distribution further lowers the required amount of exchanged data to $\mathcal{O}(n \cdot q_s)$. We show that q_s, referring to the size of the quorum, is effectively a small constant, considering a large number of participants. In addition, our protocol performs exceptionally well in the common case, where an honest leader is selected. In this case, the leader is only required to broadcast a single message, leading to a new agreed random beacon value among all the network participants.

In this thesis, we highlighted the importance of non-manipulable, publicly-verifiable randomness in decentralized systems. While basic approaches fail to provide a solution in the general case, state-of-the-art approaches set new standards in this field. We presented a new random beacon protocol improving towards an optimal solution by eliminating the scalability disadvantages of existing PVSS-based protocols. Still, further research is required to evaluate existing approaches, mitigate potential weaknesses and extend the approach to the setting of asynchronous communication. To conclude this thesis, we follow Donald Knuth's famous words:

Random numbers should not be generated with a method chosen at random.

Bibliography

[1] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. Accessed: 2017-20-08. URL: https://bitcoin.org/bitcoin.pdf.

[2] Karl Wüst and Arthur Gervais. Do you need a Blockchain?, 2017. Accessed: 2017-08-20. URL: http://eprint.iacr.org/2017/375.pdf.

[3] CoinMarketCap. CryptoCurrency Market Capitalizations, 2017. Accessed: 2017-08-20. URL: https://coinmarketcap.com/.

[4] Marko Vukolić. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. In *International Workshop on Open Problems in Network Security*, pages 112–125. Springer, 2015. Accessed: 2017-08-20. URL: http://vukolic.com/iNetSec_2015.pdf.

[5] Cynthia Dwork and Moni Naor. Pricing via Processing or Combatting Junk Mail. In *Annual International Cryptology Conference*, pages 139–147. Springer, 1992. Accessed: 2017-10-02. URL: https://link.springer.com/content/pdf/10.1007/3-540-48071-4_10.pdf.

[6] Blockchain Luxembourg S.A. Hash Rate, 2017. Accessed: 2017-08-20. URL: https://blockchain.info/en/charts/hash-rate.

[7] BitcoinWisdom.com. Bitcoin Difficulty and Hashrate, 2017. Accessed: 2017-08-20. URL: https://bitcoinwisdom.com/bitcoin/difficulty.

[8] KJ O'Dwyer and D Malone. Bitcoin Mining and its Energy Footprint. In *IET Conference Proceedings*. The Institution of Engineering & Technology, 2014. Accessed: 2017-08-20. URL: http://eprints.maynoothuniversity.ie/6009/1/DM-Bitcoin.pdf.

[9] Digiconomist. Bitcoin Energy Consumption Index, 2017. Accessed: 2017-08-29. URL: https://digiconomist.net/bitcoin-energy-consumption.

[10] Arthur Gervais, Ghassan Karame, Srdjan Capkun, and Vedran Capkun. Is Bitcoin a Decentralized Currency? *IEEE security & privacy*, 12(3):54–60, 2014. Accessed: 2017-08-21. URL: http://www.syssec.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/system-security-group-dam/research/publications/pub2014/spmagazine_gervais.pdf.

[11] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without Proof of Work. In *International Conference on Financial Cryptography and Data Security*, pages 142–157. Springer, 2016. Accessed: 2017-08-21. URL: https://arxiv.org/pdf/1406.5694.pdf.

[12] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017. Accessed: 2017-08-20. URL: https://pdfs.semanticscholar.org/4e98/d936d7693682a7378488bd651a6e48cf8fa3.pdf.

[13] Michael O Rabin. Transaction protection by beacons. *Journal of Computer and System Sciences*, 27(2):256–267, 1983. Accessed: 2017-10-02. URL: http://www.sciencedirect.com/science/article/pii/0022000083900429#.

[14] Dfinity Stiftung. Threshold Relay: How to Achieve Near-Instant Finality in Public Blockchains using a VRF, 2017. Accessed: 2017-08-20. URL: https://dfinity.network/library/threshold-relay-blockchain-stanford.pdf.

[15] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. *Advances in Cryptology—ASIACRYPT 2001*, pages 514–532, 2001. Accessed: 2017-08-20. URL: https://pdfs.semanticscholar.org/4cb3/9e436271c2a5a529082509b24132bc8ca42f.pdf.

[16] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In *Eurocrypt*, volume 2656, pages 416–432. Springer, 2003. Accessed: 2017-08-20. URL: http://cseweb.ucsd.edu/~hovav/dist/sigexts.pdf.

[17] Jing Chen and Silvio Micali. Algorand. *arXiv preprint arXiv:1607.01341*, 2017. Accessed: 2017-08-20. URL: https://arxiv.org/pdf/1607.01341.

[18] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable Bias-Resistant Distributed Randomness. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 444–460. IEEE, 2017. Accessed: 2017-08-20. URL: https://eprint.iacr.org/2016/1067.pdf.

[19] Ignacio Cascudo and Bernardo David. SCRAPE: Scalable Randomness Attested by Public Entities, 2017. Accessed: 2017-08-20. URL: http://eprint.iacr.org/2017/216.pdf.

[20] Berry Schoenmakers. A Simple Publicly Verifiable Secret Sharing Scheme and its Application to Electronic Voting. In *Annual International Cryptology Conference*, pages 148–164. Springer, 1999. Accessed: 2017-08-20. URL: https://pdfs.semanticscholar.org/54cb/6c71da3c2b0efd1557b9d03f7c0eea6700eb.pdf.

[21] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts. In *International Conference on Principles of Security and Trust*, pages 164–186. Springer, 2017. Accessed: 2017-08-30. URL: https://eprint.iacr.org/2016/1007.pdf.

[22] Benedikt Bunz, Steven Goldfeder, and Joseph Bonneau. Proofs-of-delay and randomness beacons in Ethereum. In *S&B '17: Proceedings of the 1st IEEE Security & Privacy on the Blockchain Workshop*, April 2017. Accessed: 2017-08-21. URL: http://stevengoldfeder.com/papers/BGB17-IEEESB-proof_of_delay_ethereum.pdf.

[23] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. On Bitcoin as a public randomness source. *IACR Cryptology ePrint Archive*, 2015:1015, 2015. Accessed: 2017-08-22. URL: https://eprint.iacr.org/2015/1015.pdf.

[24] Manuel Blum and Silvio Micali. How to Generate Cryptographically Strong Sequences of Pseudorandom Bits. *SIAM journal on Computing*, 13(4):850–864, 1984. Accessed: 2017-10-03. URL: https://pdfs.semanticscholar.org/3e9c/5f6f48d9ef426655dc799e9b287d754e86c1.pdf.

[25] Manuel Blum. Coin Flipping by Telephone A Protocol for Solving Impossible Problems. *ACM SIGACT News*, 15(1):23–27, 1983. Accessed: 2017-08-30. URL: http://users.cms.caltech.edu/~vidick/teaching/101_crypto/Blum81_CoinFlipping.pdf.

[26] Melanie Swan. *Blockchain: Blueprint for a New Economy.* " O'Reilly Media, Inc.", 2015. Accessed: 2017-08-30. URL: http://w2.blockchain-tec.net/blockchain/blockchain-by-melanie-swan.pdf.

[27] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, and Vignesh Kalyanaraman. BlockChain Technology: Beyond bitcoin. *Applied Innovation*, 2:6–10, 2016. Accessed: 2017-08-30. URL: http://scet.berkeley.edu/wp-content/uploads/AIR-2016-Blockchain.pdf.

[28] Aljosha Judmayer, Nicholas Stifter, Katharina Krombholz, and Edgar Weippl. Blocks and Chains: Introduction to Bitcoin, Cryptocurrencies, and Their Consensus Mechanisms. *Synthesis Lectures on Information Security, Privacy, & Trust*, 9(1):1–123, 2017.

[29] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction.* Princeton University Press, 2016.

[30] Christian Cachin. Architecture of the Hyperledger Blockchain Fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, 2016. Accessed: 2017-08-29. URL: https://www.zurich.ibm.com/dccl/papers/cachin_dccl.pdf.

[31] Carmela Troncoso, George Danezis, Marios Isaakidis, and Harry Halpin. Systematizing decentralization and privacy: Lessons from 15 years of research and

deployments. *arXiv preprint arXiv:1704.08065*, 2017. Accessed: 2017-10-03. URL: https://arxiv.org/pdf/1704.08065.pdf.

[32] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980. Accessed: 2017-10-03. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/Reaching-Agreement-in-the-Presence-of-Faults.pdf.

[33] Quynh Dang. NIST Special Publication 800-107 Revision 1 Recommendation for Applications Using Approved Hash Algorithms, 2012. Accessed: 2017-08-30. URL: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-107r1.pdf.

[34] Bitcoin Project. Bitcoin FAQ: Why do people trust Bitcoin?, 2017. Accessed: 2017-08-30. URL: https://bitcoin.org/en/faq#what-are-the-advantages-of-bitcoin.

[35] Ethereum Foundation. Ethereum Project, 2017. Accessed: 2017-08-30. URL: https://ethereum.org.

[36] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Project Yellow Paper*, 151, 2017. Accessed: 2017-08-30. URL: http://yellowpaper.io.

[37] Ethereum Foundation. Ether – Etherium Homestead Documentation, 2017. Accessed: 2017-08-17. URL: http://ethdocs.org/en/latest/ether.html.

[38] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996. Accessed: 2017-08-30. URL: http://labit501.upct.es/~fburrull/docencia/SeguridadEnRedes/teoria/bibliography/HandbookOfAppliedCryptography_AMenezes.pdf.

[39] Bart Preneel. Cryptographic Hash Functions. *Transactions on Emerging Telecommunications Technologies*, 5(4):431–448, 1994. Accessed: 2017-08-30. URL: https://securewww.esat.kuleuven.be/cosic/publications/article-281.pdf.

[40] Shahram Bakhtiari, Reihaneh Safavi-Naini, Josef Pieprzyk, et al. Cryptographic hash functions: A Survey. *Centre for Computer Security Research, Department of Computer Science, University of Wollongong, Australie*, 1995. Accessed: 2017-08-30. URL: https://pdfs.semanticscholar.org/0774/8b0e0a9c601169929a427a327a19ba478101.pdf.

[41] Baris Coskun and Nasir Memon. Confusion/Diffusion Capabilities of Some Robust Hash Functions. In *Information Sciences and Systems, 2006 40th Annual Conference on*, pages 1188–1193. IEEE, 2006. Accessed: 2017-08-30. URL: https://isis.poly.edu/memon/pdf/2006_confusion.pdf.

[42] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *Proceedings of the 1st ACM conference on Computer*

and communications security, pages 62–73. ACM, 1993. Accessed: 2017-08-30. URL: https://pdfs.semanticscholar.org/0ac4/0e64b2bf4a090ad76c6a5e54033f262ae4c2.pdf.

[43] Ran Canetti, Oded Goldreich, and Shai Halevi. The Random Oracle Methodology, Revisited. *Journal of the ACM (JACM)*, 51(4):557–594, 2004. Accessed: 2017-08-30. URL: https://arxiv.org/pdf/cs/0010019.

[44] Neal Koblitz and Alfred J Menezes. The Random Oracle Model: A Twenty-Year Retrospective. *Designs, Codes and Cryptography*, 77(2-3):587–610, 2015. Accessed: 2017-08-30. URL: http://eprint.iacr.org/2015/140.pdf.

[45] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. FIPS Publication 202, National Institute of Standards and Technology, U.S. Department of Commerce, August 2015. Accessed: 2017-08-30. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[46] Mihir Bellare and Phillip Rogaway. Introduction to Modern Cryptography. *Ucsd Cse*, 207:207, 2005. Accessed: 2017-08-30. URL: http://digidownload.libero.it/persiahp/crittografia/2005_Introduction_to_Modern_Cryptography.pdf.

[47] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. Accessed: 2017-08-30. URL: http://www.dtic.mil/get-tr-doc/pdf?AD=ADA606588.

[48] Elaine Barker and Allen Roginsky. Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths. *NIST Special Publication*, 800:131A, 2011. Accessed: 2017-08-30. URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf.

[49] Jakob Jonsson, Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. PKCS# 1: RSA Cryptography Specifications Version 2.2. *RFC8017*, 2016. Accessed: 2017-09-06. URL: https://tools.ietf.org/html/rfc8017.

[50] Mihir Bellare and Phillip Rogaway. The Exact Security of Digital Signatures – How to Sign with RSA and Rabin. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 399–416. Springer, 1996. Accessed: 2017-09-06. URL: https://link.springer.com/content/pdf/10.1007/3-540-68339-9_34.pdf.

[51] Anna Lysyanskaya. Unique Signatures and Verifiable Random Functions from the DH-DDH Deparation. In *Annual International Cryptology Conference*, pages 597–612. Springer, 2002. Accessed: 2017-08-30. URL: http://cs.brown.edu/research/pubs/pdfs/2002/Lysyanskaya-2002-USV.pdf.

[52] Tim Dierks and Eric Rescorla. Rfc 5246: The transport layer security (tls) protocol. *The Internet Engineering Task Force*, 2008. Accessed: 2017-08-30. URL: https://tools.ietf.org/html/rfc5246.

[53] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable Random Functions. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 120–130. IEEE, 1999. Accessed: 2017-08-30. URL: https://dash.harvard.edu/bitstream/handle/1/5028196/Vadhan_VerifRandomFunction.pdf?sequence=2.

[54] Yevgeniy Dodis and Aleksandr Yampolskiy. A Verifiable Random Function With Short Proofs and Keys. In *International Workshop on Public Key Cryptography*, pages 416–431. Springer, 2005. Accessed: 2017-08-30. URL: https://eprint.iacr.org/2004/310.pdf.

[55] Dennis Hofheinz and Tibor Jager. Verifiable Random Functions from Standard Assumptions. In *Theory of Cryptography Conference*, pages 336–362. Springer, 2016. Accessed: 2017-08-30. URL: https://eprint.iacr.org/2015/1048.pdf.

[56] Cameron F Kerry and Patrick D Gallagher. Digital Signature Standard (DSS). *FIPS PUB*, pages 186–4, 2013. Accessed: 2017-08-30. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[57] Thomas Pornin. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). 2013. Accessed: 2017-08-30. URL: https://tools.ietf.org/html/rfc6979.html.

[58] Dfinity Stiftung. Dfinity | The Decentralized Cloud, 2017. Accessed: 2017-08-20. URL: https://dfinity.network.

[59] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979. Accessed: 2017-08-20. URL: http://www.dtic.mil/get-tr-doc/pdf?AD=ADA069397.

[60] George Robert Blakley. Safeguarding cryptographic keys. *Proc. of the National Computer Conference1979*, 48:313–317, 1979.

[61] Amos Beimel. Secret-Sharing Schemes: A Survey. In *International Conference on Coding and Cryptology*, pages 11–46. Springer, 2011. Accessed: 2017-08-30. URL: https://pdfs.semanticscholar.org/1e53/3e711f6c5905143f8a500843a6fcab150fc0.pdf.

[62] Changlu Lin, Lein Harn, and Dingfeng Ye. Information-theoretically Secure Strong Verifiable Secret Sharing. In *SECRYPT*, pages 233–238, 2009. Accessed: 2017-08-30. URL: http://h.web.umkc.edu/harnl/papers/C1.pdf.

[63] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *Foundations of Computer Science, 1985., 26th Annual Symposium on*, pages 383–395. IEEE, 1985.

[64] Paul Feldman. A Practical Scheme for Non-interactive Verifiable Secret Sharing. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 427–438. IEEE, 1987. Accessed: 2017-08-30. URL: https://www.cs.umd.edu/~gasarch/TOPICS/secretsharing/feldmanVSS.pdf.

[65] Markus Stadler. Publicly Verifiable Secret Sharing. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 190–199. Springer, 1996. Accessed: 2017-08-30. URL: https://pdfs.semanticscholar.org/9e7c/439f791b22173470c9bbff5f1f306af58e6d.pdf.

[66] Assia Ben Shil, Kaouther Blibech, Riadh Robbana, and Wafa Neji. A New PVSS Scheme with a Simple Encryption Function. *arXiv preprint arXiv:1307.8209*, 2013. Accessed: 2017-08-30. URL: https://arxiv.org/pdf/1307.8209.pdf.

[67] Jan Varho. Is it possible to get better randomness by using multiple PRNGs?, 2014. Accessed: 2017-08-16. URL: https://crypto.stackexchange.com/questions/17897/is-it-possible-to-get-better-randomness-by-using-multiple-prngs.

[68] Yih-Chun Hu, Markus Jakobsson, and Adrian Perrig. Efficient constructions for one-way hash chains. In *International Conference on Applied Cryptography and Network Security*, pages 423–441. Springer, 2005. Accessed: 2017-08-30. URL: http://repository.cmu.edu/cgi/viewcontent.cgi?article=3215&context=compsci.

[69] Leslie Lamport. Password Authentication with Insecure Communication. *Communications of the ACM*, 24(11):770–772, 1981. Accessed: 2017-09-10. URL: http://merlot.usc.edu/cs530-s07/papers/Lamport81a.pdf.

[70] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42. ACM, 2016. Accessed: 2017-09-06. URL: https://eprint.iacr.org/2016/199.pdf.

List of Figures

4.1 Total amount of communication required for a PVSS-based random beacon protocol, assuming an adversary controlling up to 33% of the nodes 51
4.2 Overview of the RandHound design [18] 53

5.1 Probability of an attacker being able to predict a specific value of the hashchain-based random beacon δ rounds before it is released 68
5.2 Probability of an 33% attacker being able to predict at least one of the hashchain-based random beacon values δ rounds before it is released, considering a continuous prediction effort over different time periods 69
5.3 Markov chain, illustrating leader selection bias in the hashchain-based random beacon protocol 71
5.4 Influence of leader selection bias in the hashchain-based random beacon protocol over time 71
5.5 Probabilities of passive and manipulating attackers being able to predict specific values of the hashchain-based random beacon δ steps before they are released 72
5.6 Probability of an 33% actively manipulating attacker being able to predict at least one of the hashchain-based random beacon values δ rounds before it is released, considering continuous prediction effort over different time periods 73

6.1 Overview of the PVSS-based random beacon 78
6.2 Total amount of communication required for our PVSS-based random beacon protocol, assuming an adversary controlling up to 33% of the nodes . . . 85
6.3 Probabilistic guarantees for PVSS-beacon unpredictability – the chart gives the probability of an attacker being able to predict a specific value of the PVSS-based random beacon value δ steps before it is released 87
6.4 Markov chain, illustrating the unbiasability for leader selection in the PVSS-based random beacon protocol 88

7.1 Minimal values for the quorum parameter δ_{min} to achieve the desired predictability upper bound of 10^{-12} 94
7.2 Optimized quorum parameters q_s and q_t for $\delta^* = 38$ 98
7.3 Optimized quorum parameters q_s and q_t for δ_{min} 98

7.4 Total amount of communication required for our PVSS-based random beacon protocol with quorum share distribution, assuming an adversary controlling up to 33% of the nodes 99

8.1 Comparison of communication complexity of existing PVSS-based protocols, with our protocol and our extended protocol 111

List of Tables

4.1 Communication requirements for Ouroboros' random beacon protocol . . . 50

5.1 Comparison of the expected leader selection probability and the actual leader selection probability under active manipulation for the hashchain-based random beacon protocol . . . 72

8.1 Comparison of approaches for generating public-verifiable randomness . . . 106
8.2 Comparison of protocol failure guarantees . . . 112

List of Algorithms

5.1 Verification of Hashchain Values 61
5.2 A Hashchain-based Random Beacon 65

7.1 Optimized Quorum Selection 91
7.2 Derivation of Optimal Quorum Parameters 97

APPENDIX A

Appendix

A.1 Full example of Shamir's Secret Sharing

In the following, we provide an example for a $(3, 5)$ Shamir Secret Sharing, including details on all the computation steps involved. We assume the a dealer whats to share a secret number S between 0 and 52, e.g. 10, among $n = 5$ participants, such that any collaboration of $t = 3$ participants, e.g. participants 1, 4 and 5, can reconstruct the secret number.

Let $S = \alpha_0 = 10$ be the secret number to be shared.

The prime number $p = 53$ defines the set of potentials values to be shared, thus $\mathbb{Z}_p = \mathbb{Z}_{53} = \{0, 1, 2, ..., 52\}$. All operations are performed over the set $\mathbb{Z}_p$.

Choose $t - 1 = 2$ coefficients $\alpha_1 = 17$ and $\alpha_2 = 44$ randomly from $\mathbb{Z}_p$.

The coefficients and the secret number form the polynomial $p(x) = 10 + 17x + 44x^2$.

Calculate the shares: $S_i \equiv p(i) \pmod{p}$

$$
\begin{aligned}
S_1 &\equiv p(1) && \pmod{53} \\
S_1 &\equiv 10 + 17 \cdot 1 + 44 \cdot 1^2 && \pmod{53} \\
S_1 &\equiv 10 + 17 + 44 && \pmod{53} \\
S_1 &\equiv 71 && \pmod{53} \\
S_1 &= 18
\end{aligned}
$$

Similarly we get $S_2 = 8$, $S_3 = 33$, $S_4 = 40$ and $S_5 = 29$.

We select a set of t shares for reconstruction: $\{S_1, S_4, S_5\}$

The indices for the shares are then given by $i_1 = 1$, $i_2 = 4$ and $i_3 = 5$.

Then the reconstruction is given by evaluating the formula for Lagrange interpolation:

$$S \equiv \sum_{j=1}^{t} S_{i_j} \prod_{\substack{k=1 \\ j \neq k}}^{t} i_k \cdot (i_k - i_j)^{-1} \pmod{53}$$

$$S \equiv S_{i_1} \prod_{\substack{k=1 \\ 1 \neq k}}^{3} i_k \cdot (i_k - i_1)^{-1} + S_{i_2} \prod_{\substack{k=1 \\ 2 \neq k}}^{3} i_k \cdot (i_k - i_2)^{-1} + S_{i_3} \prod_{\substack{k=1 \\ 3 \neq k}}^{3} i_k \cdot (i_k - i_3)^{-1} \pmod{53}$$

$$S \equiv S_{i_1} \prod_{\substack{k=1 \\ 1 \neq k}}^{3} i_k \cdot (i_k - i_1)^{-1} + ... \pmod{53}$$

$$S \equiv S_{i_1}\big((i_2 \cdot (i_2 - i_1)^{-1}) \cdot (i_3 \cdot (i_3 - i_1)^{-1})\big) + ... \pmod{53}$$

$$S \equiv S_1[(4 \cdot (4 - 1)^{-1}) \cdot (5 \cdot (5 - 1)^{-1})\big) + ... \pmod{53}$$

$$S \equiv 18\big((4 \cdot (3)^{-1}) \cdot (5 \cdot (4)^{-1})\big) + ... \pmod{53}$$

$$S \equiv 18\big((4 \cdot 18) \cdot (5 \cdot 40)\big) + ... \pmod{53}$$

$$S \equiv 259200 + ... \pmod{53}$$

$$S \equiv 30 + ... \pmod{53}$$

$$S \equiv 30 + S_{i_2} \prod_{\substack{k=1 \\ 2 \neq k}}^{3} i_k \cdot (i_k - i_2)^{-1} + ... \pmod{53}$$

$$S \equiv 30 + S_{i_2}\big((i_1 \cdot (i_1 - i_2)^{-1}) \cdot (i_3 \cdot (i_3 - i_2)^{-1})\big) + ... \pmod{53}$$

$$S \equiv 30 + S_4\big((1 \cdot (1 - 4)^{-1}) \cdot (5 \cdot (5 - 4)^{-1})\big) + ... \pmod{53}$$

$$S \equiv 30 + 40\big((1 \cdot (-3)^{-1}) \cdot (5 \cdot 1^{-1})\big) + ... \pmod{53}$$

$$S \equiv 30 + 40\big((1 \cdot 50^{-1}) \cdot (5 \cdot 1^{-1})\big) + ... \pmod{53}$$

$$S \equiv 30 + 40\big((1 \cdot 35) \cdot (5 \cdot 1)\big) + ... \pmod{53}$$

$$S \equiv 30 + 7000 + ... \pmod{53}$$

$$S \equiv 30 + 4 + ... \pmod{53}$$

$$S \equiv 30 + 4 + S_{i_3} \prod_{\substack{k=1 \\ 3 \neq k}}^{3} i_k \cdot (i_k - i_3)^{-1} \pmod{53}$$

$$S \equiv 30 + 4 + S_{i_3}\big((i_1 \cdot (i_1 - i_3)^{-1}) \cdot (i_2 \cdot (i_2 - i_3)^{-1})\big) \pmod{53}$$

$$S \equiv 30 + 4 + S_5\big((1 \cdot (1 - 5)^{-1}) \cdot (4 \cdot (4 - 5)^{-1})\big) \pmod{53}$$

$$S \equiv 30 + 4 + 29\big((1 \cdot (-4)^{-1}) \cdot (4 \cdot (-1)^{-1})\big) \pmod{53}$$

$$S \equiv 30 + 4 + 29\big((1 \cdot 49^{-1}) \cdot (4 \cdot 52^{-1})\big) \pmod{53}$$

$$S \equiv 30 + 4 + 29\big((1 \cdot 13) \cdot (4 \cdot 52)\big) \pmod{53}$$

$$S \equiv 30 + 4 + 78416 \pmod{53}$$

$$S \equiv 30 + 4 + 29 \pmod{53}$$

$$S = 10$$

A.2 Python Implementation of Schoenmakers' PVSS

In the following, we provide a implementation developed during our study of Schoenmakers' PVSS protocol. We tested our implementation using Pyhton 3.6. There are no dependencies on external libraries.

File: `pvss.py`

```
import hmac
from hashlib import sha512
from collections import namedtuple
from random import SystemRandom

from Zp import *

random = SystemRandom() # should return cryptographic random numbers
# update to use Python 3.6 new random module secrets

GroupParameters = namedtuple('GroupParameters', ['p', 'q', 'g', 'G'])
KeyPair = namedtuple('KeyPair', ['private_key', 'public_key'])


class PVSS:

   def __init__(self, t, n, group=None, keypair=None, coefficients=None):
      """
      Creates an instance of a (t, n) public-verifiable secret sharing scheme.
      :param t: number of required shares for recovering the secret
      :param n: total number of participants in the scheme (not counting the dealer)
      :param group: a group of prime order q in which the discrete logarithm problem is hard,
          independent generators g, G of the group (no party knows the discrete log of g with
          respect to G) these are publicly known parameters
      :param keypair: optional, set a private/public keypair used for share decryption proof
      :param coefficients: optional, list of coefficients of length t used for secret polynomial
      """
      if group is None:
         group = Zp_1367
      self.t = t
      self.n = n
      self.q = group.size()
      self.Gq = group
      self.Zq = Zp(self.q)
      self.g, self.G = group.generators()
      if keypair is not None:
         self.keypair = KeyPair(*keypair)
      self.coefficients = coefficients

   def init_secret(self, secret=None):
      """
      Sets the coefficients of the secret polynomial to random values from Zq.
      :param secret: optional, if specified the first coefficient is set to secret, otherwise a
          random value is chosen
      :return: G**p(0) == G**secret, note this is different to secret itself
      """
      self.coefficients = [self.random_element() for _ in range(self.t)]
      if secret is not None:
         self.coefficients[0] = self.Zq(secret)
      return self.G ** self.evaluate_polynomial(0)

   def share(self, public_keys):
      """
      Creates n shares the secret G**p(0) and a non-interactive zero-knowledge correctness proof
          for the shares.
      The shares can be validated by any third party.
      :param public_keys: a list of the public keys of the n participants
      :return: tuple of the encrypted shares and the correctness proof
      """
      t, n, g = self.t, self.n, self.g
      # creates share for each participant, encrypt with the provided public keys
      encrypted_shares = [pub ** self.evaluate_polynomial(i + 1) for i, pub in
```

```
            enumerate(public_keys)]

    # generate a publicly-verifiable, zero-knowledge, non-interactive proof for the validity of
        the shares
    commitments = [g ** c for c in self.coefficients]
    challenge, responses = self.discrete_log_equality_parallel_composition(
        g1=[g] * n,                                                    # use same generator g
        h1=[prod([commitments[j] ** ((i + 1) ** j) for j in range(t)]) for i in range(n)], #
            h1[i] = g**p(i+1)
        g2=public_keys,
        h2=encrypted_shares,
        alpha=[self.evaluate_polynomial(i + 1) for i in range(n)]
    )

    proof = (commitments, challenge, responses)
    return encrypted_shares, proof

def decrypt(self, encrypted_share):
    """
    Decrypts a provided shares using the stored keypair and proofs the correctness of the
        decryption.
    :param encrypted_share: the encrypted share to decrypt
    :return: a tuple of the decrypted share and the correctness proof
    """
    if self.keypair is None:
        assert False, 'No keypair set'
    share = encrypted_share ** (1 / self.keypair.private_key)
    proof = self.discrete_log_equality(
        g1=self.G,
        h1=self.keypair.public_key,
        g2=share,
        h2=encrypted_share,
        alpha=self.keypair.private_key
    )
    return share, proof

def recover(self, shares):
    """
    Recovers the shared secret from exactly t shares.
    Note that the provided shares shares have to be encrypted and verified before recovering.
    :param shares: a list of n values;
        each value is either a verified and decrypted share or None (if the shareholder is not
            participating)
    :return: the shared secret G**p(0)
    """
    indices_one_based = [i + 1 for i in range(self.n) if not shares[i] is None]
    assert (len(indices_one_based) >= self.t)
    return prod([shares[i - 1] ** self.lagrange_coefficient(i, indices_one_based) for i in
        indices_one_based])

def verify_shares(self, encrypted_shares, proof, public_keys):
    """
    Verifies that a dealer has provided correct shares.
    :param encrypted_shares: list of n encrypted shares
    :param proof: correctness proof
    :param public_keys: public keys of the participants of the secret sharing scheme
    :return: True if the verification was successful, False otherwise
    """
    t, n, g = self.t, self.n, self.g
    commitments, challenge, responses = proof

    if len(encrypted_shares) != self.n: return False
    if len(commitments) != self.t: return False
    if len(responses) != self.n: return False

    try:
        return self.verify_discrete_log_equality_parallel_composition(
            g1=[g] * n,
            h1=[prod([commitments[j] ** ((i + 1) ** j) for j in range(t)]) for i in range(n)], #
                h1[i] = g**p(i+1)
            g2=public_keys,
            h2=encrypted_shares,
            challenge=challenge,
```

```
            responses=responses
        )
    except:
        return False

def verify_share_decryption(self, share, encrypted_share, proof, public_key):
    """
    Verifies that a participant has correctly decrypted a share.
    :param share: the decrypted share to be verified
    :param encrypted_share: the publicly known encrypted share
    :param proof: the correctness proof for the decryption
    :param public_key: the public key of the participant who provided the share
    :return: True if the verification was successful, False otherwise
    """
    challenge, response = proof
    return self.verify_discrete_log_equality(
        g1=self.G,
        h1=public_key,
        g2=share,
        h2=encrypted_share,
        challenge=challenge,
        response=response
    )

def generate_keypair(self, return_private_key=False):
    """
    Generates a random keypair for the participant.
    :param return_private_key: if True, also the private key is returned
    :return: the public key generated, or a tuple of private and public key
    """
    private_key = self.random_element(exclude_zero=True) # indeed breaks if private key is zero
    public_key = self.G ** private_key
    self.keypair = KeyPair(private_key, public_key)
    if return_private_key:
        return self.keypair
    return public_key

def discrete_log_equality(self, g1, h1, g2, h2, alpha):
    """
    Zero-knowledge, non-interactive proof of the fact that the proofer knows alpha such that:
    h1 = g1**alpha and h2 = g2**alpha
    :return: tuple of a challenge and a response which are used to verify the proof
    """
    w = self.random_element()
    a1 = g1 ** w
    a2 = g2 ** w
    challenge = self.compute_hash(h1, h2, a1, a2)
    response = w - alpha * challenge
    return challenge, response

def discrete_log_equality_parallel_composition(self, g1, h1, g2, h2, alpha):
    """
    See discrete_log_equality.
    Runs discrete_log_equality in parallel n times using a common challenge.
    :return: tuple of a common challenge and n responses which are used to verify the proof
    """
    n = self.n
    w = [self.random_element() for _ in range(n)]
    a1 = [g1[i] ** w[i] for i in range(n)]
    a2 = [g2[i] ** w[i] for i in range(n)]
    challenge = self.compute_hash(h1, h2, a1, a2)
    responses = [w[i] - alpha[i] * challenge for i in range(n)]
    return challenge, responses

def verify_discrete_log_equality(self, g1, h1, g2, h2, challenge, response):
    """
    Verifies a discrete log equality proof.
    :return: True if the verification was successful, False otherwise
    """
    a1 = g1 ** response * h1 ** challenge
    a2 = g2 ** response * h2 ** challenge
    challenge_verification = self.compute_hash(h1, h2, a1, a2)
    return challenge == challenge_verification
```

```
    def verify_discrete_log_equality_parallel_composition(self, g1, h1, g2, h2, challenge,
         responses):
        """
        Verifies a parallel discrete log equality proof with a common challenge.
        :return: True if the verification was successful, False otherwise
        """
        n = self.n
        a1 = [g1[i] ** responses[i] * h1[i] ** challenge for i in range(n)]
        a2 = [g2[i] ** responses[i] * h2[i] ** challenge for i in range(n)]
        challenge_verification = self.compute_hash(h1, h2, a1, a2)
        return challenge == challenge_verification

    def evaluate_polynomial(self, x):
        """
        Evaluates the secret polynomial (of degree t-1) at x.
        The coefficients for the polynomial are randomly generated by init_secret.
        :param x: evaluation point for the polynomial, typically a 1-based index of the participant
        :return: f(x) mod p
        """
        return sum(self.coefficients[j] * (x ** j) for j in range(self.t))

    def random_element(self, exclude_zero=False):
        """
        Returns a random element from Zq which can be used for cryptographic purposes.
        The result is used as an exponent for generators of the Group Gp
        :param exclude_zero: if set, the function returns only non-zero elements
        :return: the random element generated
        """
        r = random.randint(1 if exclude_zero else 0, self.q - 1)
        return self.Zq(r)

    def lagrange_coefficient(self, i, share_indices):
        """
        Calculates the lagrange coefficient used for recovering the shared secret.
        :param i: 1-based index of the current share
        :param share_indices: 1-based indices of all t shares used for recovering
        :return: the lagrange coefficient l_i = prod(j/(j-i), j!=i)
        """
        a, b = self.Zq(1), self.Zq(1)
        for j in share_indices:
            if j != i:
                a *= j
                b *= j - i
        return a / b

    def compute_hash(self, *data):
        """
        Computes a cryptographic hash.
        To get a evenly distributed element of Zq multiple HMAC with keys 0, 1, ... get concatenated
        :return: hash(data) from Zq
        """
        msg = str(data).encode()
        # each element of Zp should be equally likely, therefore the hash need to be extended
        required_hash_concatenations = int(math.ceil((math.log(self.q) + 256) / 512))
        h = ''
        for i in range(required_hash_concatenations):
            h += hmac.new(key=str(i).encode(), msg=msg, digestmod=sha512).hexdigest()
        return self.Zq(int(h, 16))


def prod(values):
    """
    Calculated the product of the given values from Gq.
    (This corresponds to point addition in elliptic curves)
    :param values: generator of elements from Gq
    :return: the product of the given elements
    """
    p = values[0]
    for v in values[1:]:
        p *= v
    return p
```

```
def is_element_of_group(x, p, q):
   return 0 < x < p and pow(x, q, p) == 1


def pvss_example(t=3, n=5):
   print("\n%d out of %d pvss example\n" % (t, n))

   dealer = PVSS(t, n)
   participants = [PVSS(t, n) for _ in range(n)]
   anyone = PVSS(t, n)

   public_keys = [p.generate_keypair(return_private_key=False) for p in participants]
   private_keys = [p.keypair.private_key for p in participants]
   print('private keys: ' + str(private_keys))
   print('public keys: %s\n' % str(public_keys))

   secret = dealer.init_secret()
   print('secret: %s\n' % str(secret))

   shares = [dealer.G ** dealer.evaluate_polynomial(i + 1) for i in range(n)]
   encrypted_shares, proof = dealer.share(public_keys)
   print('Encrypted shares: ' + str(encrypted_shares))

   decrypted_shares = [participants[i].decrypt(encrypted_shares[i])[0] for i in range(n)]
   decrypted_share_proofs = [participants[i].decrypt(encrypted_shares[i])[1] for i in range(n)]
   print('Decrypted shares: %s\n' % str(decrypted_shares))

   shares_for_recover = decrypted_shares[:]
   for not_participating_index in random.sample(range(n), n-t):
      shares_for_recover[not_participating_index] = None

   recovered_secret = anyone.recover(shares_for_recover)
   print('Shares for recover: %s\n' % str(shares_for_recover))
   print('Recovered secret: %s\n' % str(recovered_secret))

   verification_result = anyone.verify_shares(encrypted_shares, proof, public_keys)
   print('Share validity proof: ' + str(proof))
   print('Verification result: %s\n' % str(verification_result))

   verification_results_decryption = [anyone.verify_share_decryption(
      decrypted_shares[i], encrypted_shares[i], decrypted_share_proofs[i], public_keys[i])
      for i in range(n)]
   print('Decrypted share proofs: %s' % str(decrypted_share_proofs))
   print('Verification results for decryption: %s\n' % str(verification_results_decryption))

   assert(shares == decrypted_shares)
   assert(secret == recovered_secret)
   assert(verification_result is True)
   assert(verification_results_decryption == [True] * n)
```

File: `Zp.py`

```
import math
from Zn import Zn


def Zp(p):
   return lambda value: Zn(value, p)


def define_group(p):
   q = (p - 1) // 2
   group = Zp(p)
   group.generators = lambda: [group(2), group(3)]
   group.size = lambda: q
   group.bits = math.ceil(math.log(p, 2))
   return group
```

```
# the smallest possible groups for testing, p is a safe prime, 2 and 3 are generators

Zp_23 = define_group(23)
Zp_47 = define_group(47)
Zp_167 = define_group(167)
Zp_263 = define_group(263)
Zp_359 = define_group(359)
Zp_383 = define_group(383)
Zp_479 = define_group(479)
Zp_503 = define_group(503)
Zp_719 = define_group(719)
Zp_839 = define_group(839)
Zp_863 = define_group(863)
Zp_887 = define_group(887)
Zp_983 = define_group(983)
Zp_1319 = define_group(1319)
Zp_1367 = define_group(1367)

# groups from 128, 256, up to 8192 bits based on safe primes calculated from pi
# also here 2 and 3 are independent generators of the group

Zp_128 = define_group(int(
   'c90fdaa22168c234c4c6628b80dc1daf', 16))

Zp_256 = define_group(int(
   'c90fdaa22168c234c4c6628b80dc1cd129024e088a67cc74020bbea63b15031f', 16))

Zp_512 = define_group(int(
   'c90fdaa22168c234c4c6628b80dc1cd129024e088a67cc74020bbea63b139b22'
   '514a08798e3404ddef9519b3cd3a431b302b0a6df25f14374fe1356d6d535a77', 16))

Zp_1024 = define_group(int(
   'c90fdaa22168c234c4c6628b80dc1cd129024e088a67cc74020bbea63b139b22'
   '514a08798e3404ddef9519b3cd3a431b302b0a6df25f14374fe1356d6d51c245'
   'e485b576625e7ec6f44c42e9a637ed6b0bff5cb6f406b7edee386bfb5a899fa5'
   'ae9f24117c4b1fe649286651ece45b3dc2007cb8a163bf0598da48361c7e4caf', 16))

Zp_1536 = define_group(int(
   'c90fdaa22168c234c4c6628b80dc1cd129024e088a67cc74020bbea63b139b22'
   '514a08798e3404ddef9519b3cd3a431b302b0a6df25f14374fe1356d6d51c245'
   'e485b576625e7ec6f44c42e9a637ed6b0bff5cb6f406b7edee386bfb5a899fa5'
   'ae9f24117c4b1fe649286651ece45b3dc2007cb8a163bf0598da48361c55d39a'
   '69163fa8fd24cf5f83655d23dca3ad961c62f356208552bb9ed529077096966d'
   '670c354e4abc9804f1746c08ca18217c32905e462e36ce3be39e772c18331077', 16))

Zp_2048 = define_group(int(
   'c90fdaa22168c234c4c6628b80dc1cd129024e088a67cc74020bbea63b139b22'
   '514a08798e3404ddef9519b3cd3a431b302b0a6df25f14374fe1356d6d51c245'
   'e485b576625e7ec6f44c42e9a637ed6b0bff5cb6f406b7edee386bfb5a899fa5'
   'ae9f24117c4b1fe649286651ece45b3dc2007cb8a163bf0598da48361c55d39a'
   '69163fa8fd24cf5f83655d23dca3ad961c62f356208552bb9ed529077096966d'
   '670c354e4abc9804f1746c08ca18217c32905e462e36ce3be39e772c180e8603'
   '9b2783a2ec07a28fb5c55df06f4c52c9de2bcbf6955817183995497cea956ae5'
   '15d2261898fa051015728e5a8aaac42dad33170d04507a33a85521abdf53ee2f', 16))

Zp_3072 = define_group(int(
   'c90fdaa22168c234c4c6628b80dc1cd129024e088a67cc74020bbea63b139b22'
   '514a08798e3404ddef9519b3cd3a431b302b0a6df25f14374fe1356d6d51c245'
   'e485b576625e7ec6f44c42e9a637ed6b0bff5cb6f406b7edee386bfb5a899fa5'
   'ae9f24117c4b1fe649286651ece45b3dc2007cb8a163bf0598da48361c55d39a'
   '69163fa8fd24cf5f83655d23dca3ad961c62f356208552bb9ed529077096966d'
   '670c354e4abc9804f1746c08ca18217c32905e462e36ce3be39e772c180e8603'
   '9b2783a2ec07a28fb5c55df06f4c52c9de2bcbf6955817183995497cea956ae5'
   '15d2261898fa051015728e5a8aaac42dad33170d04507a33a85521abdf1cba64'
   'ecfb850458dbef0a8aea71575d060c7db3970f85a6e1e4c7abf5ae8cdb0933d7'
   '1e8c94e04a25619dcee3d2261ad2ee6bf12ffa06d98a0864d87602733ec86a64'
   '521f2b18177b200cbbe117577a615d6c770988c0bad946e208e24fa074e5ab31'
   '43db5bfce0fd108e4b82d120a92108011a723c12a787e6d788719a10bdefa25f', 16))

Zp_4096 = define_group(int(
   'c90fdaa22168c234c4c6628b80dc1cd129024e088a67cc74020bbea63b139b22'
   '514a08798e3404ddef9519b3cd3a431b302b0a6df25f14374fe1356d6d51c245'
   'e485b576625e7ec6f44c42e9a637ed6b0bff5cb6f406b7edee386bfb5a899fa5'
```

```
    'ae9f24117c4b1fe649286651ece45b3dc2007cb8a163bf0598da48361c55d39a'
    '69163fa8fd24cf5f83655d23dca3ad961c62f356208552bb9ed529077096966d'
    '670c354e4abc9804f1746c08ca18217c32905e462e36ce3be39e772c180e8603'
    '9b2783a2ec07a28fb5c55df06f4c52c9de2bcbf6955817183995497cea956ae5'
    '15d2261898fa051015728e5a8aaac42dad33170d04507a33a85521abdf1cba64'
    'ecfb850458dbef0a8aea71575d060c7db3970f85a6e1e4c7abf5ae8cdb0933d7'
    '1e8c94e04a25619dcee3d2261ad2ee6bf12ffa06d98a0864d87602733ec86a64'
    '521f2b18177b200cbbe117577a615d6c770988c0bad946e208e24fa074e5ab31'
    '43db5bfce0fd108e4b82d120a92108011a723c12a787e6d788719a10bdba5b26'
    '99c327186af4e23c1a946834b6150bda2583e9ca2ad44ce8dbbbc2db04de8ef9'
    '2e8efc141fbecaa6287c59474e6bc05d99b2964fa090c3a2233ba186515be7ed'
    '1f612970cee2d7afb81bdd762170481cd0069127d5b05aa993b4ea988d8fddc1'
    '86ffb7dc90a6c08f4df435c93402849236c3fab4d27c7026c1d4dcb2637aa8ff', 16))

Zp_6144 = define_group(int(
    'c90fdaa22168c234c4c6628b80dc1cd129024e088a67cc74020bbea63b139b22'
    '514a08798e3404ddef9519b3cd3a431b302b0a6df25f14374fe1356d6d51c245'
    'e485b576625e7ec6f44c42e9a637ed6b0bff5cb6f406b7edee386bfb5a899fa5'
    'ae9f24117c4b1fe649286651ece45b3dc2007cb8a163bf0598da48361c55d39a'
    '69163fa8fd24cf5f83655d23dca3ad961c62f356208552bb9ed529077096966d'
    '670c354e4abc9804f1746c08ca18217c32905e462e36ce3be39e772c180e8603'
    '9b2783a2ec07a28fb5c55df06f4c52c9de2bcbf6955817183995497cea956ae5'
    '15d2261898fa051015728e5a8aaac42dad33170d04507a33a85521abdf1cba64'
    'ecfb850458dbef0a8aea71575d060c7db3970f85a6e1e4c7abf5ae8cdb0933d7'
    '1e8c94e04a25619dcee3d2261ad2ee6bf12ffa06d98a0864d87602733ec86a64'
    '521f2b18177b200cbbe117577a615d6c770988c0bad946e208e24fa074e5ab31'
    '43db5bfce0fd108e4b82d120a92108011a723c12a787e6d788719a10bdba5b26'
    '99c327186af4e23c1a946834b6150bda2583e9ca2ad44ce8dbbbc2db04de8ef9'
    '2e8efc141fbecaa6287c59474e6bc05d99b2964fa090c3a2233ba186515be7ed'
    '1f612970cee2d7afb81bdd762170481cd0069127d5b05aa993b4ea988d8fddc1'
    '86ffb7dc90a6c08f4df435c93402849236c3fab4d27c7026c1d4dcb2602646de'
    'c9751e763dba37bdf8ff9406ad9e530ee5db382f413001aeb06a53ed9027d831'
    '179727b0865a8918da3edbebcf9b14ed44ce6cbaced4bb1bdb7f1447e6cc254b'
    '332051512bd7af426fb8f401378cd2bf5983ca01c64b92ecf032ea15d1721d03'
    'f482d7ce6e74fef6d55e702f46980c82b5a84031900b1c9e59e7c97fbec7e8f3'
    '23a97a7e36cc88be0f1d45b7ff585ac54bd407b22b4154aacc8f6d7ebf48e1d8'
    '14cc5ed20f8037e0a79715eef29be32806a1d58bb7c5da76f550aa3d8a1fbff0'
    'eb19ccb1a313d55cda56c9ec2ef29632387fe8d76e3c0468043e8f663f4860ee'
    '12bf2d5b0b7474d6e694f91e6dbe115974a3926f12fee5e438777cb6ac52db87', 16))

Zp_8192 = define_group(int(
    'c90fdaa22168c234c4c6628b80dc1cd129024e088a67cc74020bbea63b139b22'
    '514a08798e3404ddef9519b3cd3a431b302b0a6df25f14374fe1356d6d51c245'
    'e485b576625e7ec6f44c42e9a637ed6b0bff5cb6f406b7edee386bfb5a899fa5'
    'ae9f24117c4b1fe649286651ece45b3dc2007cb8a163bf0598da48361c55d39a'
    '69163fa8fd24cf5f83655d23dca3ad961c62f356208552bb9ed529077096966d'
    '670c354e4abc9804f1746c08ca18217c32905e462e36ce3be39e772c180e8603'
    '9b2783a2ec07a28fb5c55df06f4c52c9de2bcbf6955817183995497cea956ae5'
    '15d2261898fa051015728e5a8aaac42dad33170d04507a33a85521abdf1cba64'
    'ecfb850458dbef0a8aea71575d060c7db3970f85a6e1e4c7abf5ae8cdb0933d7'
    '1e8c94e04a25619dcee3d2261ad2ee6bf12ffa06d98a0864d87602733ec86a64'
    '521f2b18177b200cbbe117577a615d6c770988c0bad946e208e24fa074e5ab31'
    '43db5bfce0fd108e4b82d120a92108011a723c12a787e6d788719a10bdba5b26'
    '99c327186af4e23c1a946834b6150bda2583e9ca2ad44ce8dbbbc2db04de8ef9'
    '2e8efc141fbecaa6287c59474e6bc05d99b2964fa090c3a2233ba186515be7ed'
    '1f612970cee2d7afb81bdd762170481cd0069127d5b05aa993b4ea988d8fddc1'
    '86ffb7dc90a6c08f4df435c93402849236c3fab4d27c7026c1d4dcb2602646de'
    'c9751e763dba37bdf8ff9406ad9e530ee5db382f413001aeb06a53ed9027d831'
    '179727b0865a8918da3edbebcf9b14ed44ce6cbaced4bb1bdb7f1447e6cc254b'
    '332051512bd7af426fb8f401378cd2bf5983ca01c64b92ecf032ea15d1721d03'
    'f482d7ce6e74fef6d55e702f46980c82b5a84031900b1c9e59e7c97fbec7e8f3'
    '23a97a7e36cc88be0f1d45b7ff585ac54bd407b22b4154aacc8f6d7ebf48e1d8'
    '14cc5ed20f8037e0a79715eef29be32806a1d58bb7c5da76f550aa3d8a1fbff0'
    'eb19ccb1a313d55cda56c9ec2ef29632387fe8d76e3c0468043e8f663f4860ee'
    '12bf2d5b0b7474d6e694f91e6dbe115974a3926f12fee5e438777cb6a932df8c'
    'd8bec4d073b931ba3bc832b68d9dd300741fa7bf8afc47ed2576f6936ba42466'
    '3aab639c5ae4f5683423b4742bf1c978238f16cbe39d652de3fdb8befc848ad9'
    '22222e04a4037c0713eb57a81a23f0c73473fc646cea306b4bcbc8862f8385dd'
    'fa9d4b7fa2c087e879683303ed5bdd3a062b3cf5b3a278a66d2a13f83f44f82d'
    'df310ee074ab6a364597e899a0255dc164f31cc50846851df9ab48195ded7ea1'
    'b1d510bd7ee74d73faf36bc31ecfa268359046f4eb879f924009438b481c6cd7'
    '889a002ed5ee382bc9190da6fc026e479558e4475677e9aa9e3050e2765694df'
    'c81f56e880b96e7160c980dd98a573ea4472065a139cd2906cd1cb72a081a97f', 16))
```

File: Zn.py

```
class Zn(object):
    """
    wrapper class for easy arithmetic over integers mod n
    """

    def __init__(self, value, mod):
        self.value = value % mod
        self.mod = mod

    def __int__(self):
        return self.value

    def __add__(self, other):
        return Zn(self.value + int(other), self.mod)

    def __radd__(self, other):
        return Zn(self.value + int(other), self.mod)

    def __sub__(self, other):
        return Zn(self.value - int(other), self.mod)

    def __rsub__(self, other):
        return Zn(self.value - int(other), self.mod)

    def __mul__(self, other):
        return Zn(self.value * int(other), self.mod)

    def __rmul__(self, other):
        return Zn(self.value * int(other), self.mod)

    def __div__(self, other):
        if isinstance(other, Zn):
            return self * other.inverse()
        return self * Zn(other, self.mod).inverse()

    def __truediv__(self, other):
        if isinstance(other, Zn):
            return self * other.inverse()
        return self * Zn(other, self.mod).inverse()

    def __rdiv__(self, other):
        return other * self.inverse()

    def __rtruediv__(self, other):
        return other * self.inverse()

    def __pow__(self, power, mod=None):
        if mod is None:
            mod = self.mod
        if isinstance(power, Zn):
            power = power.value
        else:
            power = int(power)
        return Zn(pow(self.value, power, mod), mod)

    def __iadd__(self, other):
        self.value = (self.value + int(other)) % self.mod
        return self

    def __isub__(self, other):
        self.value = (self.value - int(other)) % self.mod
        return self

    def __imul__(self, other):
        self.value = (self.value * int(other)) % self.mod
        return self
```

```
    def inverse(self):
        """
        only supported if mod is a prime (other cases are not needed)
        """
        return self ** (self.mod - 2)

    def __repr__(self):
        return str(self.value)
        # return 'Zn(%d, mod=%d)' % (self.value, self.mod)

    def __str__(self):
        return str(self.value)

    def __eq__(self, other):
        if isinstance(other, Zn):
            return self.value == other.value
        return self.value == int(other) % self.mod

    def __ne__(self, other):
        if isinstance(other, Zn):
            return self.value != other.value
        return self.value != int(other) % self.mod
```

A.3 Attack on the Use of a Fixed Initial Seed

In the following, we provide a proof of concept implementation for the attack in regard to the use of a predefined seed in the hashchain-based random beacon protocol, as described in section 5.7. We use Python 3.6 for our implementation. Using the attack we show that an attacker controlling t out of n nodes can precompute at least t random beacon values after protocol start.

```
import hashlib
import secrets

NUM_PARTICIPANTS = 100
NUM_ATTACKERS = 30
COMMITMENT_DEPTH = 100
NUM_RANDOM_BEACONS = 50

# initialize R_0 with fractional part of pi
R = [0x243f6a8885a308d313198a2e03707344a4093822299f31d0082efa98ec4e6c89]


def H(x):
   return int.from_bytes(
      hashlib.sha256(x.to_bytes(32, 'big')).digest(),
      'big'
   )


def random_secret():
   return secrets.randbits(256)


def iterated_hash(value, iterations):
   for i in range(iterations):
      value = H(value)
   return value


def get_preimage(seed, image, max_iterations=1000):
   current = seed
   for _ in range(max_iterations):
      if H(current) == image:
         return current
      current = H(current)
   return None


def attack(C_honest):
   print("ATTACK")

   def find_commitment(lowerbound, upperbound):
      while True:
         s = random_secret()
         c = iterated_hash(s, COMMITMENT_DEPTH)
         if lowerbound < c < upperbound:
            return s, c

   # set of all commitments
   C = C_honest[:]

   # secrets for all attacker commitments
   S = dict()

   # add all but one (initial) attacker commitment to C
   for i in range(NUM_ATTACKERS - 1):
      s = random_secret()
      c = iterated_hash(s, COMMITMENT_DEPTH)
      S[c] = s
      C.append(c)
   C.sort()
```

```
    # add attacker commitment at index Pi % NUM_PARTICIPANTS
    # this ensures that the first leader an attacker
    idx_R0 = R[0] % NUM_PARTICIPANTS
    s_start, c_start = find_commitment(C[idx_R0 - 1], C[idx_R0])
    S[c_start] = s_start
    C.insert(idx_R0, c_start)

    # build a chain in such a way that each attacker is the chain once
    # a colluding attacker can predict as many random values as there are attackers
    attacker_idx = [i for i, c in enumerate(C) if c in S]
    available_idx = attacker_idx[:]
    available_idx.remove(idx_R0)
    current_idx = idx_R0
    R_current = R[0]

    while available_idx:
        print(NUM_ATTACKERS - len(available_idx), available_idx, current_idx)
        while True:
            c = C[current_idx]
            R_next = H(R_current ^ get_preimage(S[c], c))
            next_idx = R_next % NUM_PARTICIPANTS
            if next_idx in available_idx:
                # next leader is again a attack node
                available_idx.remove(next_idx)
                R_current, current_idx = R_next, next_idx
                break
            else:
                # try again using a modified commitment for the same attack idx
                lowerbound, upperbound = 0, 2**256-1
                if current_idx > 0: lowerbound = C[current_idx - 1]
                if current_idx + 1 < NUM_PARTICIPANTS: upperbound = C[current_idx + 1]
                sn, cn = find_commitment(lowerbound, upperbound)
                C[current_idx] = cn
                S[cn] = sn
                del S[c]

    print(available_idx, current_idx)
    return S


# list of all secrets, initialized with the commitments of the honest participants
_secrets = [random_secret() for _ in range(NUM_PARTICIPANTS - NUM_ATTACKERS)]
# _secrets = [random_secret() for _ in range(NUM_PARTICIPANTS)]

# transform secrets into dict: dict[commitment] => secret
_secrets = {iterated_hash(s, COMMITMENT_DEPTH): s for s in _secrets}

commitments = list(_secrets) # list of commitments of all participants
commitments_attacker = [] # only of the attacker

# add the attacker commitments
for commitment, secret in attack(commitments).items():
    commitments.append(commitment)
    commitments_attacker.append(commitment)
    _secrets[commitment] = secret


commitments.sort()
attacker_idx = [i for i, c in enumerate(commitments) if c in commitments_attacker]

print()
print("RANDOM BEACON OUTPUT")
print(f'  0: {R[-1]:064x}; initial seed')
for i in range(1, NUM_RANDOM_BEACONS):
    idx = R[-1] % NUM_PARTICIPANTS
    c = commitments[idx]
    preimage = get_preimage(_secrets[c], c)
    R.append(H(R[-1] ^ preimage))
    print(f'{i:>3}: {R[-1]:064x}; calculated by an attacker: {idx in attacker_idx}')

    commitments[idx] = preimage
    _secrets[preimage] = _secrets[c]
    del _secrets[c]
```