

VIETNAM NATIONAL UNIVERSITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



CAPSTONE PROJECT REPORT

RESEARCH AND DEVELOPMENT OF A
DECENTRALIZED RANDOM NUMBER
GENERATION PROTOCOL FOR
BLOCKCHAIN-BASED APPLICATION

Major: COMPUTER SCIENCE

Committee: Committee 8 Computer Science

Instructors: Dr. Nguyen An Khuong

Mr. Tran Anh Dung

Orochi Network

Dr. Tang Khai Hanh

Dr. Dang Tuan Thuong

HHU

Reviewer: Tran Huy

Student: Pham Nhat Minh

1910346

Ho Chi Minh city, September 2023

KHOA: KH & KT Máy tính
BỘ MÔN: KHMT

NHIỆM VỤ LUẬN VĂN/ ĐỒ ÁN TỐT NGHIỆP
Chủ ý: Sinh viên phải dẫn tờ này vào trang nhất của bản thuyết trình

HỌ VÀ TÊN: PHẠM NHẬT MINH
NGÀNH: KHMT

MSSV: 1910346
LỚP:

1. Đầu đề luận văn/ đồ án tốt nghiệp: “Nghiên Cứu và Phát Triển Giao Thức Sinh Số Ngẫu Nhiên Phi Tập Trung”

2. Nhiệm vụ (yêu cầu về nội dung và số liệu ban đầu):

- Tìm hiểu sâu và toàn diện về các về các giao thức sinh số ngẫu nhiên RNG và DRNG đã có.
- Tìm hiểu sâu và toàn diện các kiến thức về cơ sở toán, nền tảng về mật mã học và công nghệ blockchain.
- Thông qua các kiến thức tìm hiểu và nghiên cứu được, đề xuất định nghĩa tường minh, chặt chẽ cho các tính chất cần thiết của một DRNG và
- Đề xuất và xây dựng một giao thức DRNG thỏa mãn định nghĩa và tính chất đã được nêu.
- Chứng minh chặt chẽ giao thức DRNG đề xuất thỏa mãn định nghĩa và tính chất đã được nêu.
- Hiện thực thử nghiệm giao thức DRNG và kiểm tra kết quả bằng các bộ test chuẩn của NIST

3. Ngày giao nhiệm vụ: .../.../2023

4. Ngày hoàn thành nhiệm vụ: 30/07/2023

5. Họ tên giảng viên hướng dẫn:

Phản hưởng dẫn:

- 1) Nguyễn An Khương (CSE-HCMUT),
- 2) Trần Anh Dũng (Orochi Network),
- 3) Tăng Khải Hạng (NTU Singapore),
- 4) Đặng Tuấn Thương (University Düsseldorf)

HD chính
HD phụ
HD phụ
HD phụ

Nội dung và yêu cầu LVTN/ ĐATN đã được thông qua Bộ môn.

Ngày tháng năm

CHỦ NHIỆM BỘ MÔN
(Ký và ghi rõ họ tên)

GIẢNG VIÊN HƯỚNG DẪN CHÍNH
(Ký và ghi rõ họ tên)



Nguyễn An Khương

PHẦN DÀNH CHO KHOA, BỘ MÔN:

Người duyệt (chấm sơ bộ):

Đơn vị:

Ngày bao vệ:

Điểm tổng kết:

Nơi lưu trữ LVTN/ĐATN:

Ngày 29 tháng 09 năm 2023

PHIẾU ĐÁNH GIÁ LUẬN VĂN/ ĐỒ ÁN TỐT NGHIỆP
(Dành cho người hướng dẫn)

1. Họ và tên SV: **Phạm Nhật Minh** MSSV: **1910346** Ngành (chuyên ngành): **KHMT**
2. Đề tài: **"RESEARCH AND DEVELOPMENT OF A DECENTRALIZED RANDOM
NUMBER GENERATION PROTOCOL FOR BLOCKCHAIN-BASED APPLICATION"**

3. Họ tên người hướng dẫn: **Nguyễn An Khương, Trần Anh Dũng;
Tăng Khải Hạnh và Đặng Tuấn Thương**

4. Tổng quát về bản thuyết minh:

Số trang: **174**

Số bảng số liệu

Số tài liệu tham khảo.

Hiện vật (sản phẩm)

Số chương: **07**

Số hình vẽ:

Phần mềm tính toán.

5. Những ưu điểm chính của ĐATN: Nhật Minh là một sinh viên có năng lực toán học vượt trội, có tinh thần làm việc độc lập, và khả năng nghiên cứu xuất sắc. Trong quá trình làm ĐACN và ĐATN này, Nhật Minh đã nghiên cứu rất chuyên sâu về mật mã học hiện đại (có liên quan đến bài toán sinh số ngẫu nhiên), và công nghệ blockchain để thu được nhiều kết quả vượt xa quy mô của một ĐATN thông thường, với độ khó rất cao. Trong đó có thể kể đến việc tác giả đã lần đầu tiên đưa ra một định nghĩa toàn diện về DRNG, bao gồm cho cả các biến thể tương tác và không tương tác. Điều này mở rộng hơn so với công trình trước của Galindo và cộng sự, trong đó chủ yếu tập trung vào các DRNG không tương tác. Đồng thời Nhật Minh cũng đề xuất được một hệ thống DRNG chuyên dành cho các ứng dụng dựa trên công nghệ blockchain bằng cách sử dụng giao thức DVRF với nhiều cải tiến nhằm tăng tính hiệu quả của giao thức DVRF ban đầu của Galindo và cộng sự. Một phần kết quả trong ĐATN này đã được chấp nhận để công bố trong "Springer's Lecture Notes in Computer Science (LNCS, volume 14376)" và các kết quả còn lại sẽ được chuẩn bị thành những bản thảo gửi đăng trong các tạp chí phù hợp trong tương lai gần.

6. Những thiếu sót chính của ĐATN: ...

7. Đề nghị: Được bảo vệ ☒ Bỏ sung thêm để bảo vệ ☐ Không được bảo vệ ☐

8. Các câu hỏi SV phải trả lời trước Hội đồng:

a. Hãy phân tích kỹ nhu cầu về tính "publicly verifiable" của các số ngẫu nhiên được sinh ra trong các ứng dụng blockchain?


b. Hãy so sánh kỹ cách tiếp cận bằng VRF so với VDF trong việc sinh số ngẫu nhiên phi tập trung?

c. Hãy bình luận về nhu cầu có một định nghĩa toàn diện một cách chính thức cho các DRGN?

9. Đánh giá chung (bằng chữ: Xuất sắc, Giỏi, Khá, TB): **Xuất sắc** Điểm :

10/10

Ký tên (ghi rõ họ tên)



Nguyễn An Khương

PHIẾU ĐÁNH GIÁ LUẬN VĂN/ ĐỒ ÁN TỐT NGHIỆP
(Dành cho người phân biện)

1. Họ và tên SV: Phạm Nhật Minh
MSSV: 1910346

Ngành (chuyên ngành): Khoa học Máy tính

2. Đề tài: Nghiên Cứu và Phát Triển Giao Thức Sinh Số Ngẫu Nhiên Phi Tập Trung
3. Họ tên người phân biện: Trần Huy

4. Tổng quát về bản thuyết minh:

Số trang: 174

Số bảng số liệu: 16

Số tài liệu tham khảo: 79

Hiện vật (sản phẩm)

Số chương: 7

Số hình vẽ: 4

Phần mềm tính toán:

5. Những ưu điểm chính của LV/ ĐATN:

Luận văn xây dựng một giao thức sinh số ngẫu nhiên phi tập trung (Decentralized Random Number Generator - DRNG). Thông qua tiến hành literature review của các giao thức trước đây, luận văn xác định giao thức DVRF để hiện thực dựa trên tiêu chí độ phức tạp và các tính chất của một DRNG.

Giao thức DVRF được luận văn hiện thực tường minh thông qua 3 bước đề xuất của việc sinh số ngẫu nhiên: DRNGSetup, DRNGGen và DRNGVerify. Giao thức được demo thông qua việc xây dựng một Smart Contract.

Giao thức đã hiện thực cũng được đánh giá thông qua bộ dữ liệu NIST nhằm đo đặc độ hiệu quả của giao thức và đạt được kết quả rất tốt.

6. Những thiếu sót chính của LV/ĐATN:

Demo của luận văn chưa cho thấy được tính ứng dụng.

Việc đánh giá thông qua bộ test NIST chưa được trình bày rõ ràng và so sánh với các giao thức khác. Vì vậy, chưa thấy được ý nghĩa của việc xây dựng giao thức DVRF.

7. Đề nghị: Được bảo vệ ☒

Bổ sung thêm để bảo vệ ☐

Không được bảo vệ ☐

Các câu hỏi SV phải trả lời trước Hội đồng:

Sinh viên hãy so sánh giữa việc sinh số ngẫu nhiên phi tập trung và các phương pháp sinh số cổ điển. Đâu là ưu, nhược? Trường hợp nào thì ta thực sự cần sinh số ngẫu nhiên phi tập trung?

Luận văn của sinh viên lựa chọn giao thức DVRF dựa trên độ phức tạp (thời gian/chi phí) và các tính chất của một DRNG. Vậy tại sao khi đánh giá, sinh viên không so sánh với các giao thức khác về thời gian hay độ hiệu quả (dựa trên số liệu của các nghiên cứu khác khi thực nghiệm) mà lại chỉ so sánh với bộ NIST?

Đánh giá chung (bằng chữ: Xuất sắc, Giỏi, Khá, TB):

Điểm : 9.4/10

tên (ghi rõ họ tên)

10


Trần Huy

Declaration

We certify that this thesis “Research and Development of a Decentralized Random Number Generation Protocol for Blockchain-based Application”, is our research under the supervision of Dr. Nguyen An Khuong, Mr. Tran Anh Dung, Dr. Tang Khai Hanh and Dr. Dang Tuan Thuong derived from practical needs and our desire to study. We also declare that everything written in this report is the result of our research and has never been published before.

Acknowledgement

First and foremost, I would like to give my deepest gratitude to my supervisor, Dr. Nguyen An Khuong, who guided me on this project. Thanks to him, I was able to discover the field of cryptography. He proposed various great resources to help me learn maths and cryptography, especially the backgrounds for this project and guided me into the right way when I faced difficulties in almost all contents of my thesis. Not only did he teach me academic studies, but he also gave me a lot of advice when I faced troubles in my daily life.

I am forever indebted to my seniors, Mr. Truong Viet Dung; my mentors, Mr. Tang Khai Hanh, Mr. Dang Tuan Thuong, Mr. Tran Anh Dung and my friend, Nguyen Phu Nghia, who also taught me a lot of background in maths and cryptography. I had a great time discussing my studies with them, and they also gave me various to help me improve my social and life skills, which helped me become a more responsible man in life.

I would like to give my sincere thanks to all members and friends of the Bach Khoa Information Security Club (BKISC), the club that I was a part of during the last year of my university life. The club was a great place to make friends and learn various topics in cyber-security. We also had a lot of fun solving and discussing many CTF problems. It has been my pleasure to meet these people and be friends with them.

My sincere thank is also extended to all of my friends I met at university, especially Le Gia Huy, Huynh Thanh Dat and Le Hoang Anh, who helped me a lot in studying many subjects at university and listened to me when I was having so much stress, especially during my first three years. Without them, I would have been doomed to fail many subjects.

My sincere thank is also extended to my other senior, Tran Dinh Vinh Thuy. He is a very good friend of mine in high school and university, who also guided me through many subjects. He also introduced me to my supervisor, who started my career in the field of cryptography.

My thanks also go to Mrs Duong Thi Xuan An, Mr Le Phuc Lu and all my friends in high

school, who taught and helped me a lot in learning mathematics and provided me a lot of knowledge before I came to university. Thanks to Mrs Duong Thi Xuan An, even though I am not very good at geometry, I still learned a lot of interesting facts from it.

Last but not least, I would like to thank my parents for their love and support during my journey until now.

Abstract

The scientific interest in the area of **decentralized random number generator** (DRNG) protocols has been thriving recently. This interest is driven by the success of disruptive technologies introduced by modern cryptography, such as cryptocurrencies, blockchain technologies, and decentralized finances, where there is an enormous need for a public, reliable, trusted, verifiable, and distributed source of randomness for these applications. However, not all DRNGs are perfect, and many systems employ older DRNG constructions for generating randomness due to their simplicity, and these DRNGs face problems in either security or efficiency. Newer DRNGs, on the other hand, are not widely known due to the lack of comprehensive literature survey for providing information about them, and thus they are not widely used for applications, such as blockchain.

In this thesis, we study the fundamental concepts of decentralized random number generators and conduct an intensive survey of their constructions. More specifically, we classify them by cryptographic primitive and present the advantages and disadvantages for each one. In the practical context, we specify a protocol with the following properties: Pseudorandomness, Unbiasability, Liveness, and Public Verifiability, which are the required properties for blockchain-based application. We then propose a DRNG for blockchain-based applications that can address almost all limitations of other current systems. Our proposal also aligns with the spirit of blockchain: decentralization, allowing many participants to join and contribute to the random generation process.

Contents

Declaration	4
Acknowledgement	4
Abstract	6
Tables List	i
Figure List	1
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	5
1.3 Challenges and Approaches	5
1.4 Structure of the Thesis	6
2 Background	8
2.1 Notation	8
2.2 Algebra	9
2.2.1 Group Theory	9
2.2.2 Field Theory	11
2.2.3 Fast Fourier Transform	12
2.3 Elliptic Curves	15
2.4 Cryptography	18
2.4.1 Negligible Function	18
2.4.2 System Models	18

2.4.2.1	Adversarial Model	18
2.4.2.2	Communication Model	19
2.4.3	Cryptographic Hardness Assumptions	20
2.4.4	Security Reduction	21
2.4.5	Cryptographic Hash Function	23
2.4.6	Zero-Knowledge Proof	24
2.4.6.1	Zero-Knowledge Proof	24
2.4.6.2	NIZK for Discrete Log Relation	26
2.4.7	Secret Sharing Scheme	28
2.4.7.1	Shamir Secret Sharing	29
2.4.7.2	Verifiable Secret Sharing	31
2.4.7.3	Publicly Verifiable Secret Sharing	38
2.4.8	Random Oracle Model	42
2.4.9	Homomorphic Encryption	43
2.4.9.1	Encryption Scheme	43
2.4.9.2	Homomorphic Encryption	44
2.4.10	Threshold Signature	46
2.4.10.1	Signature Scheme	46
2.4.10.2	Threshold Signature Scheme	47
2.4.11	Distributed Key Generation	48
2.4.12	Verifiable Random Function	50
2.4.13	Verifiable Delay Function	52
2.5	Blockchain	54
2.5.1	Blockchain Definition	54
2.5.2	Blockchain Platforms	55
2.5.3	Smart Contract	55
2.5.3.1	History of Smart Contracts	55
2.5.3.2	How Smart Contract Works	56
2.5.3.3	Application of Smart Contracts	57
3	A Comprehensive Survey of Existing DRNGs	58
3.1	Decentralized Random Number Generator	58
3.1.1	Formal Definition	59

3.1.2	Security Properties	60
3.2	DRNG Classifications	63
3.2.1	DRNG from Hashing	65
3.2.2	DRNG from Publicly Verifiable Secret Sharing	68
3.2.2.1	Constructions without Leaders	69
3.2.2.2	Construction with Leaders	76
3.2.3	DRNG from Threshold Signature	85
3.2.4	DRNG from Verifiable Random Function	88
3.2.5	DRNG from Homomorphic Encryption	92
3.2.6	DRNG from Verifiable Delay Function	97
3.2.6.1	Constructions without Trapdoors	98
3.2.6.2	Construction with Trapdoors	101
3.3	Discussion	105
3.3.1	Advantages and Disadvantages of DRNG Approaches	105
3.3.2	Complexity Analysis	107
3.3.3	Setup Assumptions	109
3.3.4	Adversarial and Network Model	109
3.4	Summary of DRNGs	110
4	Distributed Verifiable Random Function Protocol	115
4.1	Why DVRF Protocol	115
4.2	Protocol Overview	117
4.3	Distributed Key Generation of Gennaro	118
4.4	Verifiable Random Function Based on Elliptic Curves	122
4.5	Construction	125
4.5.1	Intitution	126
4.5.2	Actual Construction	127
4.5.3	Optimization	132
4.6	Security Proofs	136
4.6.1	Pseudo-randomness	136
4.6.2	Unbiasability	143
4.6.3	Liveness	144
4.6.4	Public Verifiability	144

4.7	Complexity Analysis	145
5	Our Proposed DRNG System for Blockchain-based Applications	147
5.1	Proposed System Architecture	148
5.2	Proposed System Workflow	149
6	Experiments	152
6.1	Programming Language	152
6.2	Elliptic Curve Dependencies	153
6.2.1	Library Choice	153
6.2.2	Scalar and Point Structure	153
6.3	Verifiable Random Function Based on Elliptic Curves	154
6.3.1	Structure	154
6.3.2	Main Functions	155
6.4	Implementation Flow	158
6.5	Evaluation and Result	159
6.5.0.1	The NIST Test Suite	160
6.5.0.2	The Interpretation of Empirical Results	161
7	Conclusion	163
7.1	Conclusion	163
7.2	Future Work	164
	Bibliography	166

List of Tables

3.1	A summary of existing DRNG approaches	106
3.3	A comparison of existing DRNGs.	114
6.1	NIST Test Suite.	161
6.2	NIST Test Result.	162

List of Figures

2.1	Intuition performing polynomial multiplication using FFT [CLRS09]	13
2.2	Intuition of an elliptic curve addition	16
2.3	Experiment $\text{ExpINDCPA}_{\text{Enc}}^A(1^\lambda)$	44
2.4	Experiment $\text{ExpForge}_{\text{Sign}}^A(1^\lambda)$:	47
2.5	Oracle $\mathcal{O}_{\text{Sign}}(M)$	47
2.6	Experiment $\text{ExpSec}_{\text{DKG}}^A(1^\lambda)$:	49
2.7	Experiment $\text{ExpRand}_{\text{VRF}}^A(1^\lambda, b)$:	51
2.8	Oracle $\mathcal{O}_{\text{VRF}}(X)$	52
2.9	Experiment $\text{ExpSeq}_{\text{VDF}}^A(1^\lambda)$	53
3.1	A visualization of a DRNG syntax	60
3.2	Experiment $\text{ExpRand}_{\text{DRNG}}^A(1^\lambda, b)$.	61
3.3	Oracle $\mathcal{O}_{\text{GetRandomness}}(.)$.	61
3.4	Experiment $\text{ExpBias}_{\text{DRNG}}^A(1^\lambda)$	62
3.5	An overview of existing DRNG constructions	64
3.6	The relation between DRNGs	65
4.1	The workflow of DRNGSetup	129
4.2	The workflow of DRNGGen	130
4.3	The workflow of DRNGVerify	130
5.1	System Architecture.	148
5.2	Random number request flow for blockchain-based applications	151
6.1	ECVRF Example	158
6.2	The public key of participants	158

6.3	Partial ECVRF Output	159
6.4	Final DRNG Output	159

Chapter 1

Introduction

1.1 Motivation

The Problem of Randomness Generation. Randomness are values chosen that do not follow an intelligible pattern or combination. In our daily life, randomness plays an important role in various applications, ranging from game analysis, fair distribution to simulation, modeling, statistical sampling, security, and cryptography. Let us take a look at two examples below.

- **Simulation.** In many scientific and engineering fields, computer simulations of real phenomena are commonly used. Many phenomena are affected by unpredictable processes in the real world, such as radio noise, weather, radiation... Random numbers are required to simulate these processes to give the best accurate result of the phenomena.
- **Cryptography.** Many cryptosystems require a randomness source for their scheme. Traditional schemes like **one-time pad** require their key to be random, otherwise, the adversary will be able to decrypt the ciphertext. Randomness is also used to prove the security of **ciphertext indistinguishability** (or IND-CPA for short) of many encryption schemes. To see the importance of ciphertext indistinguishability, consider the case of **e-voting**, where everyone votes for two parties, namely party 0 and party 1. Everyone encrypts their voting choice so that people do not know their vote. With the lack of sufficient randomness, ciphertext indistinguishability is not satisfied, and an adversary can distinguish between the two encryptions. Thus he can easily figure out

the e-voting result.

While random numbers find widespread application in various aspects of our daily lives, generating randomness is a different story. Computers are designed to provide predictable, logical outputs based on given inputs. They are not designed to produce the random samples necessary for all of the mentioned applications above. Consequently, substantial efforts have been dedicated to the development of methodologies for generating random numbers. One method is to use **true random number generator** (TRNG), which are machines that measure non-deterministic sources, e.g., radio noise, thermal noise, lava lamp, mouse movements, ... and use such data to produce random values. Due to the inherently unpredictable nature of these sources, they represent a viable method for generating randomness. However, TRNGs suffer from two following problems:

- TRNGs are slow to generate randomness and require specialized hardware.
- Values generated by TRNGs must be stored to test for errors. Storing them will exhaust computer memories.

When generating random numbers with TRNGs becomes inefficient, many protocols use digital algorithms to produce deterministic values indistinguishable from randomly chosen ones. These generated values are referred to as **pseudo-random** numbers, and the algorithms responsible for their generation are known as **pseudo-random number generators** (PRNGs). PRNGs typically rely on an input known as a **seed**, which entirely dictates the entire generation process. PRNGs produce outputs much faster than TRNGs due to having lower costs, but the provided outputs have excellent statistical properties similar to a TRNG, hence they are widely used. From the mathematical and statistical points of view, [KL14] gave the formal definition of a PRNG as follows.

Definition 1.1.1 (Pseudo-random Number Generator). A **pseudo-random number generator** [KL14] is a deterministic polynomial-time algorithm G that satisfies the two following conditions:

1. **Expansion.** There exists a function $\ell : \mathbb{N} \rightarrow \mathbb{N}$ such that $\ell(n) > n$ for all $n \in \mathbb{N}$ and $|G(s)| = \ell(s)$ for all $s \in \{0, 1\}^*$.
2. **Pseudo-randomness.** For all adversaries \mathcal{A} with polynomial run time in n , there exists

a negligible function negl such that for all large n , the following probability holds

$$\left| \Pr \left[\mathcal{A}(s) = 1 \mid s \xleftarrow{\$} \{0, 1\}^{\ell(n)} \right] - \Pr \left[\mathcal{A}(s) = 1 \mid s \xleftarrow{\$} G(\{0, 1\}^n) \right] \right| \leq \text{negl}(n).$$

Because PRNGs are deterministic and have a seed as an input, meaning that on the bright side, we can easily test if the output sequence produced by PRNG is computed correctly using the seed alone. However, PRNGs are not without problems. The seed holder knows the whole random sequence and may use it to benefit himself. One notorious example is Dual-EC-DRBG (Dual Elliptic Curve Deterministic Random Bit Generator) of NSA. They have been found to have backdoors. In Dual-EC-DRBG, two inputs are used, but these inputs are not independent of each other. This allows NSA to know the whole output sequence of Dual-EC-DRBG. BSAFE, Cisco, and Microsoft libraries used Dual-EC-DRBG as their random number generator. Thus the information from these companies easily falls into the hands of NSA.¹ The case of Dual-EC-DRBG highlights the risk associated with entrusting the random number generation process to a single entity or party. To mitigate this risk, one proposed approach is to distribute the responsibility of random number generation among multiple parties, as will be discussed further in the following sections.

Randomness in Blockchain Technology. Nowadays, blockchain technology is used popularly because of significant properties such as security, transparency and equality. At the dawn of blockchain, **proof-of-work** (PoW) [Nak] was used as a basic consensus protocol that allow all participants to reach an agreement on a certain state of the blockchain, which selects a leader based on their hash rate (or power) through implementing a problem that all participants must solve in order to find a very specific nonce. This procedure is randomly determined using security assumptions of the hash function, which is generally considered a one-way process, meaning it can't be reversed back to its original form. However, proof-of-work requires vast amounts of energy due to its extensive use of electricity, computing power and time. This is why **proof-of-stake** (PoS) has become an alternative method to replace the costly proof-of-work consensus protocol. It allows users to “stake” their tokens in exchange for authenticating transactions into blocks. A person who holds a stake is chosen based on how much coins they own, instead of how much power they can generate. Randomness

¹<https://archive.nytimes.com/bits.blogs.nytimes.com/2013/09/10/government-announces-steps-to-restore-confidence-on-encryption-standards/>

plays an important role in PoS systems, since it's responsible for selecting the block validator (leader) by staking levels and ascertaining fairness in balancing decisions over the network. Additionally, this reduces risks related to double spending or nothing-at-stake attacks that might occur under other conditions. All these measures help protect and secure the blockchain network as a whole.

Decentralized Random Number Generators (DRNG). As we have seen, randomness is very important for blockchain systems. However, due to publicity and determinism, blockchain cannot generate a secure random number itself. Hence, the problem of generating randomness for blockchain applications has received attention from the community. Naturally, one cannot entrust the entire random number generation process to a single trusted party, as this contradicts the philosophy of a decentralized system in blockchain, and the risk of placing trust in a trusted party is also high, as mentioned previously. To generate randomness in blockchain, one idea is to create a protocol that allows many participants to take part in the random generating process. Such a protocol is a **decentralized random number generator (DRNG)**. In a DRNG, the role of generating randomness is split among all participants, this reduces the risk of a single party having access to the whole randomness-generating process. The idea of a DRNG protocol seems to be a good direction for generating randomness for blockchain systems. Nevertheless, DRNGs are not perfect and in fact there exists problems in most constructions. Some of these problems are as follows:

- In some protocol [Ran17, Nak, GHM⁺17, DGKR18], whenever a participant in the protocol finds that the resulting randomness does not benefit him, he will not submit his computed values, leading the protocol to abort. This is the so-called **withholding attack**.
- The communication and computation complexity of some DRNGs constructions [CD17, SJK⁺17, CD20] are too high, making these protocols inefficient when the number of participants is large (for example, larger than 1000).

From the discussions above, we have seen the importance of DRNG in blockchain systems and that DRNGs are not without problems and constructing a DRNG that achieve both security and efficiency remains an interesting challenge. However, existing blockchain system rely on older DRNG constructions [Nak, Ran17, GHM⁺17, DGKR18, KRDO17], which all faces the problems above. Newer DRNG constructions, on the other hand, are not well known due

to the lack of comprehensive survey to provide information on them. Therefore, it is essential to have a comprehensive survey and research on these DRNGs to help people understand the strengths and weakness of each construction, and thus people can specify the most suitable construction for applications such as blockchain. Hence, in this thesis, we deeply research existing DRNG constructions, then specify the most suitable one for blockchain and design a DRNG system for blockchain-based applications using the DRNG of our choice.

1.2 Objectives

This thesis aims to accomplish the following:

- Deeply research existing DRNG constructions. More specifically, formally define the syntax and security properties of DRNGs, then conduct a systematic literature review and analyse the strengths and weaknesses of existing DRNG constructions in detail.
- Based on the results of our survey, specify a DRNG construction tailored for blockchain-based applications and present it formally. The DRNG must satisfy the following security properties:
 1. **Pseudo-randomness.** The output of the protocol is indistinguishable from a randomly chosen value.
 2. **Unbiasability.** Any single participant or colluding adversary should not be able to affect future random beacon values for their own goal.
 3. **Liveness.** Any colluding adversary or adversaries cannot delay the progress of the protocol or cause the protocol to abort.
 4. **Public Verifiability.** Any third party should be able to verify the outputs of the protocol using public information.
- Finally, apply the aforementioned protocol to construct a decentralized random number generation system for blockchain-based applications.

1.3 Challenges and Approaches

While doing this research, there are some problems we have faced:

- **DRNG Approach.** There are many possible attempts build a DRNG for blockchain-based applications, yet most of them contain several limitations. Several protocols do not provide Unbiasability, more specifically, they are susceptible to withholding attacks, such as the case of RANDAO, Proof-of-work. Other protocols achieve full security properties but incur a huge communication or computation complexity, making them inefficient for a large number of participants.
- **Protocol Description.** Some protocols present their idea and concepts very clearly, but many protocols, namely, ALBATROSS [CD20] and Ouroboros Praos [DGKR18] have many complicated techniques that are rather complicated for undergraduate students. Studying the necessary materials to understand their protocol clearly takes a lot of time.

To address these challenges and design an effective DRNG system for blockchain-based applications, we have adopted the following sequential steps:

- Begin by thoroughly studying the fundamental concepts employed in existing DRNG constructions, including secret sharing, threshold signature, homomorphic encryption, verifiable random functions, and verifiable delay functions.
- Conduct a systematic literature review of existing DRNG constructions by presenting each construction in high level and analyse their security and efficiency.
- Based on the systematic literature review, specify the DRNG protocols that meet our requirements, and select the protocol with the best performance.
- Present in detail our specified protocol and propose a possible optimization to it, then formally prove its security.
- Propose a DRNG system utilizing our specified protocol for blockchain-based applications.
- Finally, conduct statistical tests of the designed DRNG to assess its security.

1.4 Structure of the Thesis

The rest of the thesis is organized as follows:

- In Chapter 2, we present the necessary background of mathematics, cryptography and blockchain. These backgrounds are essential for understanding the concept and approaches of existing DRNG protocols and how to formally prove the security of a cryptographic scheme/protocol.
- In Chapter 3, we conduct a systematic literature review of existing protocols to help understanding the current state existing DRNG protocols. We give a formal definition of DRNGs, then present each construction at a high level and analyze them. Finally, we summarize all of our analyses.
- In Chapter 4, we specify our most favored DRNG protocol and formally describes the protocol, including its security proof. We also present an optimization to reduce the computation cost of the protocol.
- In Chapter 5, we propose to use the DRNG protocol to build a specific random number generating system for blockchain-based application.
- In Chapter 6, we conduct experiments on our system to access its security. Finally, we show our evaluation result.
- In Chapter 7, we summarize all of the results achieved in the thesis and all the directions for future works we can think of.

Chapter 2

Background

In this chapter, we attempt to cover various background materials in mathematics, cryptography and blockchain. These are all important materials for understanding the concept of existing DRNG constructions as well as their security and efficiency analysis in Chapters 3 and 4. In addition, we review some basic concepts of blockchain and blockchain technologies.

2.1 Notation

In this section, we introduce the notations used in this thesis.

Vectors. A vector is denoted by lowercase bolded letters, for example $\mathbf{u}, \mathbf{v}, \dots$. The i -th component of a vector \mathbf{v} is denoted by \mathbf{v}_i . Sometimes we also use (v_1, v_2, \dots, v_n) or $(v_i)_{i=1}^n$ to denote a vector whose components are v_1, v_2, \dots, v_n .

Sets. We use letters such as \mathcal{A}, \mathcal{B} to denote sets. Sometimes we also use $\{s_1, s_2, \dots, s_n\}$ or $\{s_i\}_{i=1}^n$ to denote a set whose elements are s_1, s_2, \dots, s_n . Finally, if $\mathcal{A} = \{a_1, a_2, \dots, a_t\}$ we use $\mathcal{A}_{[m,n]}$ to denote the set $\{a_m, a_{m+1}, \dots, a_{n-1}\}$, where $m < n \leq t$.

Matrices. A matrix is denoted by uppercase bolded letters, for example, $\mathbf{A}, \mathbf{B}, \dots$. The i -th row of the matrix is denoted by \mathbf{A}_i , the i -th column of the matrix is denoted by $\mathbf{A}^{(i)}$ and the element in the i -th row and j -th column is denoted by \mathbf{A}_{ij} .

Mathematics. We denote $f(x)$ to be a polynomial in x . We denote \mathbb{G} to be a group and p to be its order when \mathbb{G} is a finite group. Finally, we denote $\lambda_{i,\mathcal{V}}$ to be the Lagrange coefficient with respect to \mathcal{V} . The definition of $\lambda_{i,\mathcal{V}}$ will be introduced later in the thesis.

Protocols. We denote n to be the number of participants in a protocol and P_1, P_2, \dots, P_n to denote the participants. We denote t to be the maximal number of dishonest participants. Some protocols are **epoch-based**, in this case, we denote r to be the current epoch of the protocol. Finally, we use $\text{Protocol}(x)\langle A(x_A), B(x_B) \rangle$ to formally denote an interactive protocol between A and B with common input x , where x_A is the secret input of A and x_B is the secret input of B .

Input and Output. We usually denote X to be an input and Y to be an output of a function. Later, we denote Ω to be the output of a DRNG protocol. When the input or output of a DRNG depends on the current epoch r , then we denote them by X_r, Ω_r . In addition, the participants in a protocol also provide their partial output, in this case, we denote the partial output of P_i to be Y_i , or Y_{ri} if the partial output depends on the current epoch r .

Algorithms. We write $Y \leftarrow \text{Alg}(X)$ to denote Y as the output of an algorithm Alg with the input X . When Alg is a sampling algorithm, we denote $Y \xleftarrow{\$} \text{Alg}(X)$. In addition, the notation $Y \xleftarrow{\$} \mathcal{Y}$ denotes that Y is chosen randomly from \mathcal{Y} .

Cryptography. We denote λ to be the security parameter. We often denote pp , pk , and sk to be the public parameter, public key, and secret key used in a cryptosystem respectively. We denote pk_i and sk_i to be the public and secret key of participant P_i in a protocol. We use H to denote a cryptographic hash function used in some cryptosystems. Finally, we denote \mathcal{A} to be an adversary who tries to break the security of a cryptosystem and \mathcal{S} is an adversary who tries to break a cryptographic assumption.

2.2 Algebra

In this section, we recall the basics of group theory, field theory, and fast Fourier transform.

2.2.1 Group Theory

Definition 2.2.1 (Groups). A **group** is a set \mathbb{G} , together with a binary operation \cdot that satisfies all of the following properties:

- **Closure.** If $a, b \in \mathbb{G}$, then $a \cdot b \in \mathbb{G}$.
- **Associativity.** For all $a, b, c \in \mathbb{G}$, we have: $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.

- **Identity.** There exists an element $e \in \mathbb{G}$ such that for all $a \in \mathbb{G}$ we have $a \cdot e = e \cdot a = a$.
- **Inverse.** For every $a \in \mathbb{G}$, there exists a unique element $b \in \mathbb{G}$ such that $a \cdot b = b \cdot a = e$.
The element b is called the **inverse** element of a and is denoted by a^{-1} .

Example 2.2.1. For $n \geq 2$, let $\mathbb{Z}_n = \{0, 1, \dots, n-1\}$, together with the addition modulo n . Then \mathbb{Z}_n associated with operation “+” forms a group, denoted by $(\mathbb{Z}_n, +)$. The identity element of the group is 0. For every $a \neq 0$, the inverse of a is $n - a$.

Definition 2.2.2 (Abelian Groups). A group \mathbb{G} is said to be **abelian** if it satisfies the additional property of **commutativity**: For all $a, b \in \mathbb{G}$ we have $a \cdot b = b \cdot a$.

Example 2.2.2. We can easily check that $(\mathbb{Z}, +)$ because for any $a, b \in \mathbb{Z}$, we have $a + b = b + a$.

Definition 2.2.3 (Cyclic Groups). Let (\mathbb{G}, \cdot) be a group. Then \mathbb{G} is a **cyclic group** if there is an element $g \in \mathbb{G}$ such that every element $a \in \mathbb{G}$ can be written in the form $g^k = \underbrace{g \cdot g \cdot \dots \cdot g}_{k \text{ copies of } g}$ for some integer k . Such an element g is a **generator** of the group.

Example 2.2.3. Let \mathbb{Z}_5^\star denote the set of non-zero integers modulo 5. We see that $1 = 2^0, 2 = 2^1, 3 = 2^3, 4 = 2^2$. This means $(\mathbb{Z}_5^\star, \cdot)$ is a cyclic group with generator 2.

Definition 2.2.4 (Homomorphisms). A **homomorphism** is a map f from a group (\mathbb{G}, \cdot) to a group $(\mathbb{G}', *)$ that satisfies the following property:

$$f(a \cdot b) = f(a) * f(b) \quad \forall a, b \in \mathbb{G}.$$

In particular, we have $f(e_{\mathbb{G}}) = e_{\mathbb{G}'}$, where $e_{\mathbb{G}}$ and $e_{\mathbb{G}'}$ denote the identity element of \mathbb{G} and \mathbb{G}' respectively.

Example 2.2.4. Consider the set \mathbb{Z}_4 and \mathbb{Z}_5^\star defined in Examples 2.2.1 and 2.2.3 respectively. Then $f : \mathbb{Z}_4 \rightarrow \mathbb{Z}_5^\star$ given by $f(x) = 2^x \pmod{5}$ is a homomorphism. In fact, for any $a, b \in \mathbb{Z}_4$, we have

$$f(a) \cdot f(b) = 2^a \cdot 2^b \pmod{5} = 2^{a+b} \pmod{5}.$$

On the other hand, consider the set \mathbb{Z}_9 and \mathbb{Z}_5^\star . Then $g : \mathbb{Z}_9 \rightarrow \mathbb{Z}_5^\star$ given by $g(x) = 2^x \pmod{5}$ is NOT a homomorphism. This is because

$$g(0) = g(4 + 5) \pmod{9} = g(4) \cdot g(5) = 2^4 \pmod{5} = 16 \pmod{5} = 1 \neq 2^9 \pmod{5} = 512 \pmod{5} = 2.$$

Definition 2.2.5 (Isomorphisms). A homomorphism $f : \mathbb{G} \rightarrow \mathbb{G}'$ is an **isomorphism** if it is a bijection from \mathbb{G} to \mathbb{G}' .

Example 2.2.5. Consider the set \mathbb{Z}_4 and \mathbb{Z}_5^\star and $f : \mathbb{Z}_4 \rightarrow \mathbb{Z}_5^\star$ given by $f(x) = 2^x \pmod{5}$. We have $f(0) = 1$, $f(1) = 2$, $f(2) = 4$, $f(3) = 3$, hence f is a bijection from \mathbb{Z}_4 to \mathbb{Z}_5^\star . From Example 2.2.4, f is a homomorphism. Thus, it is obvious that f is an isomorphism.

Definition 2.2.6 (Orders). The **order** of a finite group is the number of its elements.

Example 2.2.6. The order of $(\mathbb{Z}_{23}, +)$ is 23.

2.2.2 Field Theory

Definition 2.2.7 (Fields). A set \mathbb{F} , together with addition operation “+” and multiplication operation “ \cdot ”, is a **field**, denoted by \mathbb{F} or $(\mathbb{F}, +, \cdot)$ if \mathbb{F} satisfies the following properties:

- **Addition.** $(\mathbb{F}, +)$ is an abelian group with the identity 0.
- **Multiplication.** $(\mathbb{F} \setminus \{0\}, \cdot)$ is an abelian group with the identity $1 \neq 0$.
- **Distributivity.** For all $a, b, c \in \mathbb{F}$, we have $a \cdot (b + c) = a \cdot b + a \cdot c$.

Example 2.2.7. Consider \mathbb{Z}_{23} . It is a field with the addition operation “+” and multiplication operation “ \cdot ”. The additive and the multiplicative identities $(\mathbb{Z}_{23}, +, \cdot)$ is 0 and 1, respectively.

Definition 2.2.8 (Algebraically Closed Fields). Let \mathbb{F} be a field and $\mathbb{F}[x]$ the set of polynomials of one variable with coefficients in \mathbb{F} . The field \mathbb{F} is said to be **algebraically closed** if and only if every polynomial $f \in \mathbb{F}[x]$ has all roots in \mathbb{F} . In other words, \mathbb{F} can be factorized into the product of linear polynomials in $\mathbb{F}[x]$.

Example 2.2.8 (See [FR97]). The set of complex numbers \mathbb{C} is an algebraically closed field. By Fundamental Theorem of Algebra, every polynomial of degree n in $\mathbb{C}[x]$ has all n roots (including multiplicities) in \mathbb{C} .

Definition 2.2.9 (Field Extensions). Let \mathbb{F} be a field, a **subfield** \mathbb{K} of \mathbb{F} is a subset $\mathbb{K} \subseteq \mathbb{F}$ such that \mathbb{K} is also a field with addition and multiplication inherited from \mathbb{F} . We also say that \mathbb{F} is a **field extension** of \mathbb{K} .

Example 2.2.9. *The field of complex numbers \mathbb{C} is a field extension of the field of real numbers \mathbb{R} , and \mathbb{R} is a field extension of the field of rational numbers \mathbb{Q} .*

Definition 2.2.10 (Algebraic Extensions). A field \mathbb{F} is said to be an **algebraic extension** of a field \mathbb{K} if every element of \mathbb{F} is algebraic over \mathbb{K} , i.e., every element of \mathbb{F} is the root of some nonzero polynomial with coefficients in \mathbb{K} .

Example 2.2.10. *The field of complex numbers \mathbb{C} is an algebraic extension of \mathbb{R} . It can be seen that every element z in \mathbb{C} has the form $z = a \cdot i + b$ for some $a, b \in \mathbb{R}$, hence z is a root of the polynomial $(x - b)^2 + a^2$, whose coefficients are real numbers.*

Definition 2.2.11 (Algebraic Closures). A field $\overline{\mathbb{F}}$ is an **algebraic closure** of \mathbb{F} if $\overline{\mathbb{F}}$ is an algebraic extension of \mathbb{F} and $\overline{\mathbb{F}}$ is algebraically closed.

Example 2.2.11. *The field of complex numbers \mathbb{C} is an algebraic closure of \mathbb{R} . From Example 2.2.10 we can see that \mathbb{C} is a field extension of \mathbb{R} . In addition, from Examples 2.2.8, it holds that \mathbb{C} is also an algebraically closed field.*

2.2.3 Fast Fourier Transform

One of the primary applications of fast Fourier transform is polynomial multiplication. Suppose we have two polynomials $A(x) = \sum_{i=0}^n a_i x^i$ and $B(x) = \sum_{i=0}^n b_i x^i$. Suppose we would like to calculate the coefficients of $C(x) = A(x)B(x)$. By using ordinary multiplication, i.e., calculating each c_i from $\{a_j\}_{j=0}^n$ and $\{b_k\}_{k=0}^n$, this requires $\Omega(n^2)$ steps. Naturally, this raises the question: can we compute the coefficients of $C(x)$ in linear time? Thankfully, the FFT algorithm provides a solution, enabling the calculation within $\mathcal{O}(n \log n)$ steps. At its core, given a polynomial $A(x)$ of degree n and a n -th root of unity ω_n , FFT allows us to calculate $A(\omega_n^i)$ for $i \in \{0, 1, \dots, n-1\}$ in $\mathcal{O}(n \log n)$ steps. Employing the FFT as a sub-algorithm, Figure 2.1 outlines the process of multiplying two polynomials $A(x)$ and $B(x)$ into $C(x) = A(x)B(x)$ within $\mathcal{O}(n \log n)$ steps.

Below we review some basics of complex n -th root of unity and the idea of FFT. Most of the materials in this subsection are drawn from [CLRS09].

Definition 2.2.12 (n -th root of unity). A **complex n -th root of unity** is a complex number ω such that $\omega^n = 1$, where n is a positive integer.

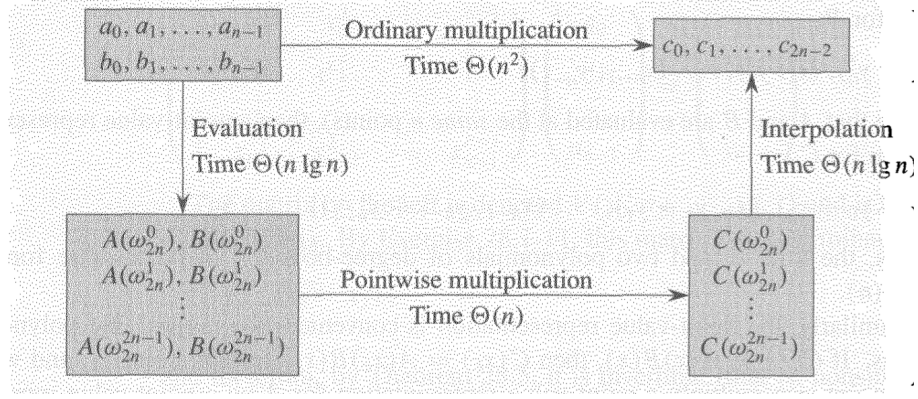


Figure 2.1: *Intuition performing polynomial multiplication using FFT* [CLRS09]

There are exactly n complex n th roots of unity: $\omega_n^j = e^{2ij\pi/n}$ for $j \in \{0, 1, \dots, n-1\}$ where i is the imaginary unit. The formula can be interpreted using the definition of the exponential of a complex number $e^{2ia} = \cos(a) + i \cdot \sin(a)$. The value $\omega_n = e^{2i\pi/n}$ is called the **principal n -th root of unity**.

Example 2.2.12. *There are exactly three complex 3rd roots of unity: 1 , $\frac{1}{2} + \frac{\sqrt{3}}{2}i$ and $\frac{1}{2} - \frac{\sqrt{3}}{2}i$.*

It can be seen that, the n th roots of unity $1, \omega_n, \omega_n^2, \dots, \omega_n^{n-1}$ forms a group under multiplication. The group structure has the same group structure as $(\mathbb{Z}_n, +)$.

FFT Algorithm. For a polynomial $f(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1}$ of degree at most $n-1$, where n is a power of 2, we now present Fast Fourier Transform (FFT), an algorithm that allows us to calculate $f(\omega_n), f(\omega_n^2), \dots, f(\omega_n^n)$ within $\mathcal{O}(n \log n)$ steps, where ω_n is the n -th root of unity defined above.

The FFT algorithm employs a divide and conquer method, using even-index and odd-index coefficients of $f(x)$ to separately define two new polynomials $f_0(x)$ and $f_1(x)$ of degree $n/2$:

$$f_0(x) = a_0 + a_2x + \dots + a_{n-2}x^{n/2-1} \text{ and } f_1(x) = a_1 + a_3x + \dots + a_{n-1}x^{n/2-1}.$$

It follows that $f(x) = f_0(x^2) + xf_1(x^2)$. Hence to calculate $f(\omega_n^i)$ for $i \in \{0, 1, \dots, n-1\}$ it suffices to calculate $f_0(w_n^i)$ and $f_1(w_n^i)$ for $i \in \{0, 1, \dots, n/2-1\}$, where $w_n = \omega_n^2$ is the principal $n/2$ -th roots of unity. These sub-problems have exactly the same form as the original

problem, but half the size. After finishing these sub-problems, we can calculate

$$\begin{cases} f(\omega_n^i) = f_0(w_n^i) + \omega_n^i f_1(w_n^i), \\ f(\omega_n^{i+n/2}) = f_0(w_n^i) - \omega_n^i f_1(w_n^i) \end{cases}$$

for all $i \in \{0, 1, \dots, n/2 - 1\}$ using the fact that $\omega_n^{i+n/2} = -\omega_n^i$. With this in mind, the FFT algorithm can be done recursively as follows.

Algorithm 1 FFT(\mathbf{f}, ω_n)

Input: A vector of coefficients $\mathbf{f} = (a_0, a_1, \dots, a_{n-1})$ of the polynomial $f(x) = \sum_{i=0}^{n-1} a_i x^i$, where n is a power of 2.

Output: A vector of values $\mathbf{y} = (f(1), f(\omega_n), f(\omega_n^2), \dots, f(\omega_n^{n-1}))$.

1. If $|\mathbf{f}| = 1$ return \mathbf{f} .
 2. Let $\mathbf{f}^0 = (a_0, a_2, \dots, a_{n-2})$ and $\mathbf{f}^1 = (a_1, a_3, \dots, a_{n-1})$.
 3. Recursively compute $\mathbf{y}^0 = \text{FFT}(\mathbf{f}^0, \omega_n^2)$ and $\mathbf{y}^1 = \text{FFT}(\mathbf{f}^1, \omega_n^2)$.
 4. Initialize $w = 1$.
 5. For $i = 0, \dots, n/2 - 1$ do:
 - Compute $\mathbf{y}_i := \mathbf{y}_i^0 + w \cdot \mathbf{y}_i^1$ and $\mathbf{y}_{i+n/2} := \mathbf{y}_i^0 - w \cdot \mathbf{y}_i^1$.
 - Set $w := w \cdot \omega_n$.
 6. Return $\mathbf{y} = (\mathbf{y}_0, \mathbf{y}_1, \dots, \mathbf{y}_{n-1})$.
-

To determine the time of the FFT algorithms above, we denote $T(n)$ as the upper bound of the number of steps when executing the FFT algorithm. We note that exclusive of the recursive calls, each step of the algorithm takes times $\mathcal{O}(n)$. Hence the recurrence for the running time is $T(n) = 2T(n/2) + \mathcal{O}(n) = \mathcal{O}(n \log n)$. Thus we conclude that, for a polynomial f of degree n , the FFT algorithm terminates after $\mathcal{O}(n \log n)$ steps.

Now that we have calculated $f(\omega_n), f(\omega_n^2), \dots, f(\omega_n^n)$ within $\mathcal{O}(n \log n)$ steps given the coefficients of f . Hence, it is natural to ask if we could do the inverse: Given the values $f(\omega_n), f(\omega_n^2), \dots, f(\omega_n^n)$, can we calculate the coefficients of f within $\mathcal{O}(n \log n)$ steps? Thankfully, the answer is also positive. and such algorithm is called **inverse FFT**. To see how inverse FFT works, we note that for each i $f(\omega_n^i) = \sum_{j=0}^{n-1} a_j \cdot \omega_n^{ij}$. For each k , by

multiplying ω_n^{-ki} with $f(\omega_n^i)$, it holds that $\omega_n^{-ki} f(\omega_n^i) = a_k + \sum_{j=0, j \neq k}^{n-1} a_j \cdot \omega_n^{i(j-k)}$. Hence, by summing all the values $\omega_n^{-ki} f(\omega_n^i)$, it holds that

$$\begin{aligned} \sum_{i=0}^{n-1} f(\omega_n^i) \omega_n^{-ki} &= n a_k + \sum_{i=0}^{n-1} \left(\sum_{j=0, j \neq k}^{n-1} a_j \omega_n^{i(j-k)} \right) \\ &= n a_k + \sum_{j=0, j \neq k}^{n-1} a_j \frac{\omega_n^{n(j-k)} - 1}{\omega_n^{(j-k)} - 1} \\ &= n \cdot a_k. \end{aligned}$$

Thus, the values a_k is given by the formula

$$a_k = \frac{1}{n} \sum_{i=0}^{n-1} f(\omega_n^i) \omega_n^{-ki}.$$

Consequently, by switching the roles of a_i and $f(\omega^i)$ in **Algorithm 1**, the coefficients a_i can be calculated in time $\mathcal{O}(n \log n)$ as well. Hence, by employing FFT and inverse FFT, we could perform polynomial multiplication in $\mathcal{O}(n \log n)$ steps, as shown in Figure 2.1.

2.3 Elliptic Curves

In this section, we review some basics of elliptic curves defined on finite fields. Most of the materials in this section are drawn from [Was03].

Definition 2.3.1 (Elliptic Curves). Let \mathbb{F} be a field where $\text{char}(\mathbb{F}) \neq 2, 3$. An **elliptic curve** E over \mathbb{F} is defined as follows.

$$E(\mathbb{F}) = \{y^2 = x^3 + ax + b : (x, y) \in \mathbb{F}^2\} \cup \{\mathcal{O}\}$$

where $a, b \in \mathbb{F}$ satisfying $4a^3 + 27b^2 \neq 0$ and \mathcal{O} is called the **point at infinity**.

In cryptography, since $\text{char}(\mathbb{F}) \neq 2, 3$, we only consider \mathbb{F} to be the field \mathbb{F}_p of integers modulo p for a prime $p > 3$.

We present the process of adding two points in an elliptic curve. Given two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, the sum of P and Q is a point R that can be defined as follows:

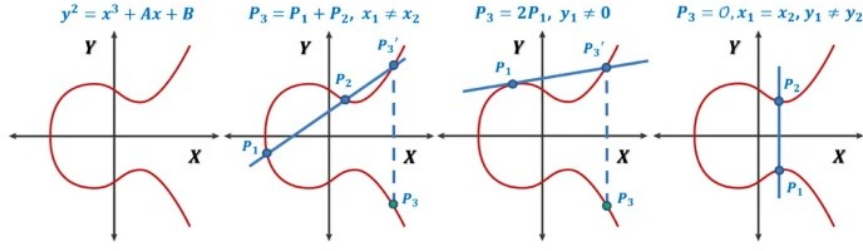


Figure 2.2: *Intuition of an elliptic curve addition [AAEHR22].*

- Let $y = sx + d$ be the equation of the line that intersects P and Q .
- When $x_P = x_Q$ and $y_P \neq y_Q$, we have $R = \mathcal{O}$.
- When $x_P = x_Q$ and $y_P = y_Q$, the slope s of the line is equal to

$$s = \frac{3x_P^2 + a}{2y_P}.$$

In this case, we have $x_R = s^2 - 2x_P$ and $y_R = s \cdot (x_P - x_R) - y_P$.

- Otherwise, the line has the slope

$$s = \frac{y_P - y_Q}{x_P - x_Q}.$$

In this case, we have $x_R = s^2 - x_P - x_Q$ and $y_R = s \cdot (x_P - x_R) - y_P$.

Example 2.3.1. *Consider the curve $E : y^2 = x^3 + 7$ over the field \mathbb{F}_{23} . It can be seen that two points $P = (8, 17)$ and $Q = (16, 3)$ are two points on the curve.*

- *The point $R = P + Q$ is computed as follows:*

1. *Since $8 \not\equiv 16 \pmod{23}$, we compute $s = 4$ since $(17 - 3)(8 - 16)^{-1} \equiv 4 \pmod{23}$.*
2. *Compute $x_R = 15$ since $4^2 - 8 - 16 \equiv 15 \pmod{23}$.*
3. *Compute $y_R = 1$ since $4(8 - 15) - 17 \equiv 1 \pmod{23}$.*
4. *The point $R = (15, 1)$ is the sum of P and Q .*

- *The point $S = 2 \cdot P$ is computed as follows:*

1. *Compute $s = 7$ since $(3 \cdot 8^2 + 0)(2 \cdot 17)^{-1} \equiv 7 \pmod{23}$.*

2. Compute $x_S = 10$ since $7^2 - 2.8 \equiv 10 \pmod{23}$.

3. Compute $y_S = 15$ since $7(8 - 10) - 17 \equiv 15 \pmod{23}$.

4. The point $S = (10, 15)$ is the sum of P and P .

- The point $-P$ is simply equal to $(15, 17)$.

It can be verified that $(E(\mathbb{F}_p), +)$ forms a group. The elements in $E(\mathbb{F}_p)$ satisfies the following properties:

- **Commutativity.** For all $P, Q \in E(\mathbb{F}_p)$, we have $P + Q = Q + P$.
- **Associativity.** For all $P, Q, R \in E(\mathbb{F}_p)$, we have $(P + Q) + R = P + (Q + R)$.
- **Identity.** The infinity point \mathcal{O} is the identity element of the group. We have, $P + \mathcal{O} = \mathcal{O} + P = P$ for any $P \in E(\mathbb{F}_p)$.
- **Inverse.** For all $P \in E(\mathbb{F}_p)$, there is a unique point $-P$ such that $P + (-P) = \mathcal{O}$. The point $-P$ is actually the symmetric point of P at the x -axis.

By the above description of the group law, we are now ready to discuss the elliptic curve discrete logarithm problem.

For a point P , we can define scalar multiplication by n as follows.

$$n \cdot P = \underbrace{P + P + \dots + P}_{n \text{ copies of } P}$$

When n is negative, then we have $n \cdot P = -n(-P)$.

Example 2.3.2. Consider the curve $E : y^2 = x^3 + 7$ over the field \mathbb{F}_{23} and let $P = (8, 17)$ be a point on the curve. We have $4 \cdot P = P + P + P + P = (11, 21)$.

The **discrete logarithm** problem is assumed to be hard on the group of an elliptic curve. It is defined as follows.

Definition 2.3.2 (Elliptic Curve Discrete Logarithm Problem (ECDLP)). Let E be an elliptic curve over \mathbb{F}_p . Given two points P and Q with $n \cdot P = Q$ for some integer n , find n .

Elliptic curves are widely used in cryptography due to the ECDLP problem and the requirement of smaller key size while providing similar security guarantees as RSA. For example, a 256-bit elliptic curve public key provides comparable security to a 3072-bit RSA public key [DM17].

2.4 Cryptography

In this section, we recall the basics of cryptography, which are important materials for understanding Chapters 3 and 4.

2.4.1 Negligible Function

For the rest of the thesis, we often use the word “negligible function” when defining security properties for many cryptographic primitives. We shall formally define this term.

Definition 2.4.1 (Negligible Function). A function $\text{negl} : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ is said to be **negligible** if for any polynomial $f(x)$, $\text{negl}(x) < 1/f(x)$ for all sufficient large $x \in \mathbb{R}^+$. Otherwise, negl is said to be **non-negligible**.

Example 2.4.1. *The function $\text{negl}(x) = 1/2^x$ is negligible, while the function $\text{negl}(x) = 1/x^2$ is non-negligible.*

Sometimes, we also use the phrase “negligible in λ ” in some expressions $\text{Exp}(\lambda)$ to denote that there exists a negligible function negl such that $\text{Exp}(\lambda) \leq \text{negl}(\lambda)$.

2.4.2 System Models

In this section, we recall some of the model assumptions that will be used in the security and complexity analysis of a protocol in Chapter 3 and 4.

2.4.2.1 Adversarial Model

An **adversary** is an algorithm that attempts to break a scheme, or a protocol or solve a hard problem. We focus on adversaries who try to break the security of a protocol. In a protocol, an adversary can choose to **corrupt** a specific number of participants, hence learning their internal states and secrets. The three ways in which the adversary can deviate from a protocol are omitting a message (i.e., withholding attack), sending invalid messages, and colluding to coordinate an attack based on the internal state shared among corrupted participants [CMB23]. By corruption strategy, adversaries can be divided into three types [Lin20]:

- **Static Adversary.** A static adversary must specify a set of participants and corrupt them before the start of the protocol. Honest participants remain honest throughout, and corrupted participants remain corrupted. We note that in this model, once a participant is corrupted, it remains corrupted from that point on.
- **Adaptive Adversary.** An adaptive adversary can choose any participant and corrupt it during the run of the protocol. In particular, it can be based on the information it learns during the protocol.
- **Proactive Adversary.** This model considers the possibility that participants are corrupted for a certain period of time only. Thus, honest participants may become corrupted throughout the computation (like in the adaptive adversarial model), but corrupted participants may also become honest.

In most constructions, people will focus on static or adaptive adversaries since proving security against proactive adversaries is hard. An example of a protocol that is secure against a static adversary but not secure against an adaptive adversary is as follows.

Example 2.4.2. *Consider n participants, and consider a protocol that begins by securely choosing a random subset of \sqrt{n} the participants who then carry out the computation for the rest. Assume that the adversary is limited to corrupting $n/2$ participants. Then, except with negligible probability, there will be at least one honest participant in the chosen \sqrt{n} . Thus, the protocol is secure against static adversaries. However, an adaptive adversary can wait until the \sqrt{n} participants are chosen and then adaptively corrupt all of them. Since it only corrupts $\sqrt{n} \leq n/2$ participants, this is allowed. Clearly, such an adversary completely breaks the protocol since it controls all the participants who carry out the actual computation.*

The adversarial model will be one of the aspects for analyzing the security of DRNG systems in Chapters 3 and 4.

2.4.2.2 Communication Model

In distributed computing, participants communicate with each other through a network by sending messages. However, in a network, there exist various problems related to communication, such as network delays, message transmission times, and clock drifts between different nodes. To capture this problem, protocols introduce the message **delay parameter**

Δ and categorize the communication model based on Δ . Below we informally review the concepts of the model of communication. Most of the materials here are drawn from [DLS88]. Recall that in distributed computing, participants may send/receive messages from other participants by performing one of these instructions:

- **Send**(M, P): Send a message m to P .
- **Receive**(P): Receive a set of messages sent to P .

Let I be an interval of time. We say that the communication bound in Δ if a message m is sent to P by **Send**(M, P) at time s_1 in I , and when P executes **Receive**(P) at time $s_2 = s_1 + \Delta$ then M must be delivered to P at time s_2 or earlier. This says intuitively that Δ is an upper bound on message transmission time in the interval I . Based on the condition of Δ , we divide the communication model into three categories:

1. **When Δ is known.** The value Δ is known for some fixed finite $\Delta \in (1, \infty)$.
2. **When Δ is unknown.** For every run R , there exists a Δ that holds in $(1, \infty)$.
3. **When Δ holds eventually.** The value Δ holds eventually for some fixed $\Delta \in (1, \infty)$.

If 1 holds, then we say that the communication is **synchronous**. If 2 or 3 holds, then the communication is **partial synchronous**. Otherwise, if there is no such Δ , i.e., the communication is unbounded, then the communication is **asynchronous**. The communication model will be one of the aspects for analyzing the security of DRNG systems in Chapter 3.

2.4.3 Cryptographic Hardness Assumptions

In the realm of cryptography, there are various problems that are assumed to be hard to solve for a computationally bounded algorithm. Many cryptosystem constructions are based on these hard problems, and these problems also serve as the pillar of the security proof of the corresponding cryptosystem. These problems can be categorized into two main types: **computational problems** and **decisional problems**. In computational problems, an algorithm has to output a specific string that matches a value in the problem. In decisional problems, an algorithm needs to decide on a value whether it belongs to a specific set or a random set. In this section, we focus on two assumptions related to a general group: The **Computational Diffie-Hellman Assumption** (CDH) and **Decisional**

Diffie-Hellman Assumption (DDH) [DH76]. Each of these assumptions represents a category of the aforementioned problems.

Let λ be a security parameter p be a prime with $p = \Omega(2^\lambda)$. Consider a cyclic group \mathbb{G} with order p and generator g . The two problems can be formulated below.

Computational Diffie-Hellman Assumption (CDH). The CDH assumption state that, for every adversary \mathcal{A} with polynomial running time in λ , it holds that

$$\text{ADV}_{\text{CDH}}(\mathcal{A}) = \Pr \left[g^{ab} \leftarrow \mathcal{A}(g^a, g^b) \mid a, b \xleftarrow{\$} \mathbb{Z}_p \right]$$

is negligible in λ . Informally, for uniform $a, b \in \mathbb{Z}_p$, given g^a, g^b , it is computationally hard to calculate g^{ab} , except for negligible probability.

Decisional Diffie-Hellman Assumption (DDH). The DDH assumption state that, for every adversary \mathcal{A} with polynomial running time in λ , it holds that

$$\text{ADV}_{\text{DDH}}(\mathcal{A}) = \left| \Pr \left[1 \leftarrow \mathcal{A}(g^a, g^b, g^{ab}) \mid a, b \xleftarrow{\$} \mathbb{Z}_p \right] - \Pr \left[1 \leftarrow \mathcal{A}(g^a, g^b, g^c) \mid a, b, c \xleftarrow{\$} \mathbb{Z}_p \right] \right|$$

is negligible in λ . Informally, the assumption state that for $a, b, c \in \mathbb{Z}_p$ and given g^a, g^b , it is hard to distinguish between g^{ab} and g^c .

2.4.4 Security Reduction

In cryptography, when proposing a protocol or a scheme, one would like to formally prove the proposed protocol or scheme to be secure against adversaries. Recall that in Subsection 2.4.3 many cryptographic systems are based on problems that are assumed to be hard. The idea for proving security is that: If there exists such an algorithm breaking the security of a proposed protocol or scheme, then there also exists an algorithm that can solve a certain hard problem. Such an algorithm is called a **reduction algorithm**. Below we briefly present some basic components in defining and proving the security of a protocol or a scheme. This allows us to get a better insight into formally proving the security of a protocol or a scheme. Most of the materials in this section are from [GSM18].

Experiment. An experiment is an interaction between an adversary and a scheme (or

a hard problem). The adversary can see all the public parameters (e.g. a public key in an encryption scheme), and some of the outputs corresponding to a set of inputs, and try to exploit the secret parameter. In a scheme, we usually use an experiment to define certain security properties of a scheme.

Reduction Algorithm. A reduction algorithm is an algorithm that uses another algorithm that breaks a proposed scheme to solve a hard problem. In other words, assume \mathcal{A} to be an algorithm that breaks the security scheme by winning an experiment, then \mathcal{S} is a reduction algorithm that uses \mathcal{A} to solve a hard problem. The main idea to prove the security of a scheme or protocol is that assuming that no computationally bounded algorithm can solve a certain hard problem, such a reduction algorithm will contradict the assumption, hence such an adversary \mathcal{A} does not exist. In some cases, \mathcal{S} is also known as the **simulator**.

Real View. A real view is the view of an adversary \mathcal{A} when interacting within the corresponding experiment. The view of an adversary consists of all of the internal states of corrupted participants and public transcripts of honest participants. An adversary then uses his view to break the security of the scheme or protocol.

Simulated View. A simulated view is what an adversary creates to take the solution of another adversary. The simulated view of the adversary \mathcal{A} made by \mathcal{S} should be indistinguishable from the real view. In this way, \mathcal{A} will proceed with the same attack as it does when interacting with the real scheme or protocol.

To give a better understanding of the idea, below we give an example of using the reduction technique to formally prove the IND-CPA security property of the ElGamal cryptosystem.

Example 2.4.3. *We present a quick demo of using the reduction technique to prove the IND-CPA security property of the ElGamal cryptosystem. In the ElGamal cryptosystem proposed by [Gam85], let \mathbb{G} be a cyclic group with order p and generator g . The setup and encryption algorithm **Setup** and **Enc** is as follows:*

Setup(1^λ) : *This algorithm sample $\mathbf{dk} \in \mathbb{Z}_p$ and let the decryption key to be \mathbf{dk} . The encryption key is $\mathbf{ek} = g^{\mathbf{dk}}$.*

Enc(M, \mathbf{ek}) : *On input a message M and encryption key \mathbf{ek} , it sample $s \in \mathbb{Z}_p$ and return the ciphertext $C = (g^s, m \cdot \mathbf{ek}^s)$.*

Dec(C, \mathbf{dk}) : *On input a ciphertext $C = (C_1, C_2)$ and a decryption key \mathbf{dk} , output $M = C_2(C_1^{\mathbf{dk}})^{-1}$. We see that $C_2(C_1^{\mathbf{dk}})^{-1} = M \cdot g^{\mathbf{dk} \cdot s} (g^{s \cdot \mathbf{dk}})^{-1} = M$.*

The IND-CPA security property of the cryptosystem is defined by using an **experiment** between a challenger \mathcal{C} and an adversary \mathcal{A} . The experiment $\text{ExpINDCPA}_{\text{Enc}}^{\mathcal{A}}(1^\lambda)$ for IND-CPA security will be defined in Subsubsection 2.4.9.1.

An adversary \mathcal{A} can break the IND-CPA security if there exists a non-negligible ϵ such that he can win $\text{ExpINDCPA}_{\text{Enc}}^{\mathcal{A}}(1^\lambda)$ with probability $1/2 + \epsilon$. Now suppose such an adversary \mathcal{A} exists, i.e., \mathcal{A} wins the IND-CPA experiment with probability $1/2 + \epsilon$, we create a **reduction algorithm** \mathcal{S} that breaks the DDH assumption with probability $1/2 + 1/2\epsilon$, which leads to contradiction, since we have assumed in Subsection 2.4.3 that \mathcal{S} cannot break the DDH assumption with non-negligible probability.

The adversary \mathcal{S} , on input $(X = g^x, Y = b^y, Z)$ has to decide whether $C = g^{xy}$ or $Z = g^z$ for some $z \xleftarrow{\$} \mathbb{Z}_p$. It proceeds to create a **simulated view** for \mathcal{A} as follows:

1. The adversary \mathcal{S} set $\text{ek} = X$ and gives ek to \mathcal{A} .
2. When \mathcal{A} outputs M_0, M_1 , \mathcal{S} picks a random $b \in \{0, 1\}$ and gives $C = (Y, m_b \cdot Z)$ to \mathcal{A} .
3. Finally, \mathcal{S} receives the guess bit b from \mathcal{A} and output b .

We see that, if $Z = g^{xy}$, then the simulated view of \mathcal{A} is the same as the real view of \mathcal{A} when interacting in the IND-CPA experiment, in this case, the probability that \mathcal{A} succeed is equal to $1/2 + \epsilon$. If $Z = g^z$ for $z \xleftarrow{\$} \mathbb{Z}_p$, then $(Y, m_b \cdot Z)$ is uniformly distributed, hence the probability that \mathcal{A} succeed in this case is $1/2$. Hence, the probability that \mathcal{S} success is equal to $1/2(1/2 + \epsilon) + 1/2 \cdot 1/2 = 1/2 + 1/2 \cdot \epsilon$, as desired.

2.4.5 Cryptographic Hash Function

A cryptographic hash function is a hash algorithm that uses a mathematical equation to validate and verify data. It is used across many industries but is most common in information security. Below recall the formal definition of a cryptographic hash function in [Gol01].

Definition 2.4.2 (Cryptographic hash function). A **cryptographic hash function** $H_\lambda : \{0, 1\}^* \rightarrow \{0, 1\}^\lambda$ takes an input $X \in \{0, 1\}^*$ and produces an output such that for all the following properties:

1. **Efficient Rvaluation.** There exists a polynomial-time algorithm in λ such that, given X , returns $H_\lambda(X)$.

2. **One-way.** The hash function is said to be one-way if given $y = H_\lambda(X)$, it is computationally infeasible to find x . More formally there exists a negligible function negl such that

$$\Pr \left[y = H_\lambda(X) \mid X \leftarrow \mathcal{A}(\lambda, y) \right] \leq \text{negl}(\lambda).$$

3. **Collision Resistance.** We say that the pair (X, X') forms a **collision** under the function H if $H(X) = H(X')$ but $X \neq X'$. We require that every adversary \mathcal{A} with probabilistic polynomial time in λ outputs a collision with negligible probability. Formally, for any adversary \mathcal{A} , there exists a negligible function negl such that

$$\Pr \left[H_\lambda(X) = H_\lambda(X') \wedge X \neq X' \mid (X, X') \leftarrow \mathcal{A}(\lambda) \right] \leq \text{negl}(\lambda).$$

Every cryptographic hash function is a hash function, but not every hash function is a cryptographic hash function. Cryptographic hash functions are designed to be more secure. They are designed so that finding a preimage and collision is impossible. Normal hash functions are designed to quickly access items in memory or compare items. This makes the length of cryptographic hash functions much longer, and they have 256 bit length. Commonly used cryptographic hash functions in the industry include SHA256, Keccak.

2.4.6 Zero-Knowledge Proof

2.4.6.1 Zero-Knowledge Proof

Zero-knowledge proofs (ZKP) were originally proposed by Goldwasser, Micali and Rackoff [GMR85]. Zero-knowledge proofs are protocols involving two participants: the verifier V and the prover P . When the protocol is executed, the prover convinces the verifier that they have a solution for a certain mathematical witness without disclosing any further data. The verifier has to make sure that the prover indeed has the necessary information. The following example helps us intuitively understand the concept of ZKP.

Example 2.4.4. Let N be a positive integer. Suppose P has a positive integer $y \in \mathbb{Z}_N^*$ claims that there exists an $x \in \mathbb{N}^*$ such that $x^2 \equiv y \pmod{N}$. The verifier V , who wishes to know whether P 's claim is true, asks the prover to jointly execute the following protocol:

1. The prover P chooses $r \in \mathbb{Z}_N^*$ and sends $s \equiv r^2 \pmod{N}$ to V .

2. The verifier V chooses a random bit $b \in \{0, 1\}$.
3. If $b = 0$, P sends $z \equiv r \pmod{N}$ to V , otherwise, P sends $z \equiv rx \pmod{N}$ to V .
4. The verifier V accepts if and only if $z^2 \equiv sy^b \pmod{N}$.

In the protocol, P might cheat by choosing s such that sy is a square \pmod{N} when $b = 1$ and s to be any square of an integer \pmod{N} when $b = 0$. However, s must be chosen before b is sent by V , hence a cheating prover does not know which s should he choose. Thus the probability that P can cheat is $1/2$. By repeating the protocol n times, the probability that P can cheat is $1/2^n$, hence V will be convinced that y is really a square in modulo N .

We now express everything above mathematically to help define the formal definition of ZKP. Mathematically speaking, let \mathcal{X} and \mathcal{W} be two sets and let $\mathcal{R} \subseteq \mathcal{X} \times \mathcal{W}$ be a **relation**. For a statement $x \in \mathcal{X} \in \{0, 1\}^*$, the prover has to prove that he knows a witness $w \in \mathcal{W} \in \{0, 1\}^*$ such that $(x, w) \in \mathcal{R}$ without revealing w , while the verifier V does not learn any additional information beyond the fact $(x, w) \in \mathcal{R}$. With this intuition, we now state the properties of a ZKP.

Definition 2.4.3. Let $\text{ZKP}(x) \langle P(w), V \rangle \in \{0, 1\}$ be a zero-knowledge protocol between P and V with statement x . Then $\text{ZKP}(x) \langle P(w), V \rangle$ satisfy the following properties:

- **Completeness.** If there exists w s.t $(x, w) \in \mathcal{R}$ then $\Pr[\text{ZKP}(x) \langle P(w), V \rangle = 1] = 1$.
- **Soundness.** If $(x, w) \notin \mathcal{R}$ for all $w \in \mathcal{W}$ then $\Pr [\text{ZKP}(x) \langle P(w), V \rangle = 1] \leq \text{negl}(|x|)$.
- **Honest Verifier Zero-Knowledge.** For all $(x, w) \in \mathcal{R}$ and adversaries \mathcal{A} with computational time bounded in $\text{poly}(|x|)$, there exists a simulator \mathcal{S} such that

$$\left| \Pr \left[\mathcal{A}(\text{tr}) = 1 \mid \text{tr} \leftarrow \text{View}(\text{ZKP}(x) \langle P(w), V \rangle) \right] - \Pr \left[\mathcal{A}(\text{tr}) = 1 \mid \text{tr} \leftarrow \mathcal{S}(x) \right] \right|$$

is negligible in $|x|$, where $\text{View}(\text{ZKP}(x) \langle P(w), V \rangle)$ denotes the public transcript made by P and V during the protocol. Informally, \mathcal{S} has to produce an accepting transcript tr by itself so that it is computationally indistinguishable from the actual transcript when P and V follow the protocol. Since tr can be simulated without using w , it follows that V cannot learn w from tr . Note that \mathcal{S} does not have to follow the steps of the protocol to create tr , as long as tr matches the actual transcript.

Example 2.4.5. We continue with Example 2.4.6 to see how can a simulator \mathcal{S} can create an accepting transcript without the witness x . The transcript tr in the protocol consists of the tuple (z, b, s) such that $z^2 \equiv y^b s \pmod{N}$. The simulator can create an accepting transcript tr by sampling $z \xleftarrow{\$} \mathbb{Z}_N^*$ and $b \in \{0, 1\}$, then compute $s \equiv z^2 / y^b \pmod{N}$. It can be proved that (z, b, s) is an accepting transcript and its distribution is identical to the actual transcript of the protocol executed by P and V . Note that in the protocol, P has to compute s first, then compute z . However, \mathcal{S} does not have to follow and thus compute z first, then compute s . It can do so as long as (z, b, s) matches the actual transcript at the end of the protocol.

Finally, we would like to note that ZKP protocols can be converted into a **non-interactive** version by applying Fiat-Shamir heuristic [FS86]. By non-interactive, we mean that the interaction between P and V consists of only one message from P to V . In this case, the non-interactive version of a ZKP protocol is abbreviated as NIZK.

2.4.6.2 NIZK for Discrete Log Relation

Now that we have understood the basic concepts of ZKP and its non-interactive version NIZK in Subsubsection 2.4.6.2, let us take a look at a ZKP for a specific relation \mathcal{R} . For a group \mathbb{G} with order p and two generators g and h , we consider the relation

$$\mathcal{R}_{\text{DLOG}} = \{(x = (X, Y), w = s) \mid X = g^s \wedge Y = h^s\}.$$

The Chaum-Pedersen protocol [CP92] allows us to prove the relations in $\mathcal{R}_{\text{DLOG}}$. Its non-interactive version consists of two algorithms **NIZKDLOGProve** and **NIZKDLOGVerify**, which are run by the P and V , respectively. These two algorithms will be employed in various protocol constructions later, for example, the PVSS scheme in Subsubsection 2.4.7.3 and the ECVRF construction in Subsection 4.4. We now describe the two algorithms below.

Public Parameter. Let \mathbb{G} be a cyclic group of order p where p is a prime and let g, h to be two of its generators. Let H be a cryptographic hash function that maps an arbitrary number of elements in \mathbb{G} into an integer. The two **NIZKDLOGProve** and **NIZKDLOGVerify**, which is run by P and V respectively as follows.

The prover P with g, h, X, Y , can prove that he knows s such that $X = g^s$ and $Y = h^s$ by running the NIZKDLOGProve algorithm. Its detail is seen at **Algorithm 2** below.

Algorithm 2 NIZKDLOGProve(g, X, h, Y, s)

Input: g, X, h, Y, s .

Output: $\pi = (c, z)$.

1. Choose a value $k \xleftarrow{\$} \mathbb{Z}_p$.
 2. Compute $c = \text{H}(g, h, X, Y, g^k, h^k)$.
 3. Compute $z \equiv k - c \cdot s \pmod{p}$.
 4. Output $\pi = (c, z)$.
-

The verifier V with public inputs g, h, X, Y, π , can verify whether P knows s by running the NIZKDLOGVerify algorithm. Its detail is seen at **Algorithm 3** below.

Algorithm 3 NIZKDLOGVerify(g, X, h, Y, π)

Input: g, h, X, Y, π

Output: $b \in \{0, 1\}$

1. Parse $\pi = (c, s)$.
 2. Compute $u = X^c g^s$ and $v = Y^c h^s$.
 3. Check if $c = \text{H}(g, h, X, Y, u, v)$.
-

Example 2.4.6. We give a quick demonstration of the two algorithms above.

\mathbb{G} : the group of elliptic curve $y^2 = x^3 + 7$ over \mathbb{F}_{43} . The order p of \mathbb{G} is 31.

Generator: $g = (7, 7), h = (38, 22)$.

Common inputs: $X = (42, 7), Y = (13, 22)$.

NIZKDLOGProve: The prover, with witness $s = 5$, process as follows:

1. Choose a random k in \mathbb{Z}_{31} , say $k = 11$.

2. Compute $g^k = (7, 7) \cdot 11 = (20, 3)$ and $h^k = (38, 22) \cdot 11 = (42, 36)$. Then, compute $c = H((7, 7), (38, 22), (42, 7), (13, 22), (20, 3), (42, 36))$. Since H acts as a black box, we can choose $c = 9$ for simplicity, and H always return 9 for inputs $(42, 7)$, $(13, 22)$, $(20, 3)$ and $(42, 36)$.
3. Compute $s = 11 - 9.5 \pmod{31} = 28$.
4. The proof is $\pi = (9, 28)$.

NIZKDLOGVerify: The verifier, who wish to know whether that P knows s , proceed as follows:

1. Compute $u = (42, 7) \cdot 9 + (7, 7) \cdot 28 = (20, 3)$.
2. Compute $v = (13, 22) \cdot 9 + (38, 22) \cdot 28 = (42, 36)$.
3. Finally, we see that $H((7, 7), (38, 22), (42, 7), (13, 22), (20, 3), (42, 36)) = 9$. Since both conditions are satisfied, the verifier is convinced that the prover knows s .

2.4.7 Secret Sharing Scheme

Secret sharing schemes were introduced by Shamir [Sha79]. Intuitively, a secret sharing scheme allows a person to share a secret s among n participants such that any **authorized** subset of participants can use all their shares to reconstruct the secret, while any other (non-authorized) subset learns nothing about the secret from their shares. Secret-sharing schemes are important tools in cryptography and they are used as a building box in many secure protocols, especially in multiparty computation, and threshold cryptography. As we will see later in Chapter 3 and 4, secret sharing scheme is an important tool for constructing many DRNGs.

Below we recall the formal definition of a secret sharing scheme and its security properties.

Definition 2.4.4 (Secret Sharing Scheme). A (t, n) -**secret sharing scheme** [BKP11] between a dealer D and participants P_1, P_2, \dots, P_n consists of an interactive protocol **Share** and an algorithm **Reconstruct** defined as follows.

1. $\{s_i\}_{i=1}^n \leftarrow \text{Share} \langle D(s), \{P_i\}_{i=1}^n \rangle$: This is an interactive protocol between D , who holds s and P_1, \dots, P_n . At the end of the protocol, each participant P_i receives a share s_i .
2. $s \leftarrow \text{Reconstruct}(\mathcal{V}, \{s_i\}_{i \in \mathcal{V}})$: This algorithm is run by participants to reconstruct the original secret s . On input any set $\mathcal{V} \subseteq \{1, 2, \dots, n\}$ with $|\mathcal{V}| \geq t + 1$ and shares $\{s_i\}_{i \in \mathcal{V}}$, this deterministic algorithms output a single value s .

We require a secret sharing scheme to satisfy the following properties:

- **Correctness.** If $\{s_i\}_{i=1}^n \leftarrow \text{Share} \langle D(s), \{P_i\}_{i=1}^n \rangle$ then for any $\mathcal{V} \subseteq \{1, \dots, n\}$ with $|\mathcal{V}| \geq t + 1$ it holds that $\text{Reconstruct}(\mathcal{V}, \{s_i\}_{i \in \mathcal{V}}) = s$.
- **Perfect/ Computational Secrecy.** For any $s \in \{0, 1\}^\lambda$ and any computationally unbounded/ computationally bounded adversary \mathcal{A} and for any sets $\mathcal{V} \subseteq \{1, \dots, n\}$ with $|\mathcal{V}| = t$, there exists a negligible function negl such that

$$\Pr \left[s \leftarrow \mathcal{A}(\mathcal{V}, \{s_i\}_{i \in \mathcal{V}}) \mid \{s_i\}_{i=1}^n \leftarrow \text{Share} \langle D(s), \{P_i\}_{i=1}^n \rangle \right] \leq \text{negl}(\lambda).$$

2.4.7.1 Shamir Secret Sharing

Shamir secret sharing scheme is the first secret sharing scheme proposed by Adi Shamir in 1979 [Sha79]. The main idea of the scheme is based on the Lagrange interpolation theorem, which is stated as follows.

Theorem 2.4.5. *Let \mathbb{F} be a field, and let $f(x) \in \mathbb{F}[x]$ be a polynomial with degree no more than t . Then for any set $\mathcal{V} \subseteq \mathbb{F}$ with $|\mathcal{V}| \geq t + 1$, given $|\mathcal{V}|$ distinct points $(i, f(i))_{i \in \mathcal{V}}$, we can uniquely determine $f(x)$ using the formula*

$$f(x) = \sum_{i \in \mathcal{V}} f(i) \cdot \prod_{j \in \mathcal{V}, j \neq i} \frac{x - j}{i - j}.$$

In particular, for any $f(x) \in \mathbb{F}[x]$ with degree no more than t and for any set $\mathcal{V} \subseteq \mathbb{F}$ with $|\mathcal{V}| \geq t + 1$ and $0 \notin \mathcal{V}$ we have the following equality

$$f(0) = \sum_{i \in \mathcal{V}} f(i) \cdot \lambda_{i, \mathcal{V}},$$

where the value $\lambda_{i, \mathcal{V}} = \prod_{j \in \mathcal{V}, j \neq i} \frac{j}{j - i}$ are the Lagrange coefficients w.r.t \mathcal{V} .

The Lagrange coefficient notion $\lambda_{i, \mathcal{V}}$ above will be used in the rest of the thesis. Now, let us back to the sharing scheme. At a high level, suppose the dealer wants to share a secret s among P_1, P_2, \dots, P_n . He then chooses a degree t polynomial $f(x)$ such that $f(0) = s$ and distributes the corresponding share $f(j)$ to participant P_j . Since the secret is the constant

term $s = f(0)$, it can be recovered given any $t + 1$ shares $\{f(j)\}_{j \in \mathcal{V}}$ by using Lagrange interpolation Theorem 2.4.5 above. Now that we have understood the concept of Shamir secret sharing scheme, we will describe the scheme in detail.

Protocol 4 Shamir's Secret Sharing Scheme

Share $\langle D(s), \{P_i\}_{i=1}^n \rangle$:

Let \mathbb{F} be a finite field, known to all participants and the dealer. The dealer randomly chooses a polynomial

$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_tx^t$$

such that $a_0 = s$. Then the dealer gives participant P_i the share $s_i = f(i)$.

Reconstruct $(\{s_i\}_{i \in \mathcal{V}})$:

From a set $\mathcal{V} \subseteq \{1, \dots, n\}$ with $|\mathcal{V}| \geq t + 1$ and secrets $\{s_i\}_{i \in \mathcal{V}}$, we can reconstruct s using Lagrange interpolation. More specifically,

$$s = f(0) = \sum_{i \in \mathcal{V}} f(i) \cdot \lambda_{i, \mathcal{V}} = \sum_{i \in \mathcal{V}} s_i \cdot \lambda_{i, \mathcal{V}}.$$

Example 2.4.7. *The following procedures are an example of a simple Shamir secret sharing scheme.*

Share(s) : *Let \mathbb{F}_{23} be the set of integers modulo 23. Suppose the dealer has a secret $s = 7$ and wants to split s among participants P_1, P_2, P_3 and P_4 . He then chooses the polynomial $f(x) = x^2 + 2x + 7$ and splits the secret to participants as follows.*

Participant	P_1	P_2	P_3	P_4
$s_i = f(i)$	$s_1 = 10$	$s_2 = 15$	$s_3 = 22$	$s_4 = 8$

Reconstruct $(\{s_i\}_{i=1}^3)$: *From any set of 3 shares, say s_1, s_2 and s_3 , they can reconstruct using Lagrange interpolation. In the case of s_1, s_2 and s_3 , we have $\mathcal{V} = \{1, 2, 3\}$ and*

$$s = 10(2 \cdot 3)(1 \cdot 2)^{-1} + 15(1 \cdot 3)((-1) \cdot 1)^{-1} + 22(1 \cdot 2)((-2)(-1))^{-1} \pmod{23} = 7.$$

As discussed above, from any $t + 1$ shares, we can recover s by using Lagrange interpolation. Moreover, given only t shares $(x_1, s_1), (x_2, s_2), \dots, (x_t, s_t)$, the value s_0 can be any element in

\mathbb{F} with equal probability, because for each element s_0 we can uniquely determine a polynomial $f(x)$ such that $f(x_i) = s_i \forall i \in \{1, \dots, t\}$ and $f(0) = s_0$, where the construction of $f(x)$ can be seen at Theorem 2.4.5. Hence, the knowledge of t shares does not give any information about s_0 . Thus Shamir secret sharing scheme achieves perfect secrecy.

The weakness of Shamir secret sharing scheme is that it is not verifiable. The participants can easily submit wrong values leading algorithm **Reconstruct** to return incorrect original secrets. In addition, the dealer may share values that do not lie in a polynomial of degree t .

Example 2.4.8. In Example 2.4.7, instead of broadcasting the correct share, participant P_3 can submit the value $s_3 = 21$, and the secret s will be calculated as

$$s = 10(2 \cdot 3)(1 \cdot 2)^{-1} + 15(1 \cdot 3)((-1) \cdot 1)^{-1} + 21(1 \cdot 2)((-2)(-1))^{-1} \pmod{23} = 6.$$

However, recall that the original secret is $s = 7$, thus the secret is constructed incorrectly without anyone realizing it.

Fortunately, the abovementioned problem of Shamir secret sharing can be solved using a **verifiable secret sharing**, which we shall describe in the next section.

2.4.7.2 Verifiable Secret Sharing

The security of Shamir secret sharing scheme assumes that the dealer and participants are honest. However, that is not always the case. participants can submit wrong values leading the reconstruct algorithm to output a wrong secret s^* . The dealer can also distribute n shares to n participants that do not lie in a polynomial of degree t . Hence, participants may need a procedure to verify the correctness of the shared values to prevent malicious behavior by the dealer as well as other participants. To solve this problem, a **verifiable secret sharing** scheme [CGMA85] is proposed. A verifiable secret sharing scheme (VSS) allows participants to verify whether other participants are lying about the contents of their shares, up to a reasonable probability of error, preventing participants from submitting wrong shares.

The syntax of a VSS scheme is a bit different from an ordinary secret sharing scheme, where a verification algorithm will be introduced to check the correctness of the shares. Now, the syntax of a VSS scheme can be formally presented as follows.

Definition 2.4.6 (Verifiable Secret Sharing Scheme). A (t, n) -**verifiable secret sharing**

scheme between a dealer D and participants P_1, P_2, \dots, P_n consists of an interactive protocol VSSShare and an two algorithms VSSShareVerify , VSSReconstruct defined as follows.

1. $(C, \{s_i\}_{i=1}^n) \leftarrow \text{VSSShare} \langle D(s), \{P_i\}_{i=1}^n \rangle$: This is an interactive protocol between D , who holds s and participants P_1, \dots, P_n . At the end of the protocol, a commitment C is publicly known to all participants. In addition, each participant P_i holds a secret share s_i that is only known to him.
2. $b \leftarrow \text{VSSShareVerify}(s_i, C)$: This is a deterministic algorithm run by each participant. On input a share s_i and a commitment C , it outputs a bit $b \in \{0, 1\}$.
3. $s/\perp \leftarrow \text{VSSReconstruct}(\mathcal{V}, \{s_i\}_{i \in \mathcal{V}}, C)$: This algorithm is run by participants to reconstruct the secret s . From a set \mathcal{V} and shares $\{s_i\}_{i \in \mathcal{V}}$ and a commitment value C , this deterministic algorithms output a single value s , or \perp .

VSS retains the Perfect/Computational Secrecy property of an ordinary secret sharing scheme. However, the Correctness property of VSS is modified a bit, which state that the VSSReconstruct returns the original secret s when D is honest instead of any D . In addition, VSS additionally includes the **Commitment** property. Below we will describe the modified Correctness and the Commitment properties of a VSS as follows:

- **Correctness.** If D is honest, then shares s_i it holds that $\text{VSSShareVerify}(s_i, C) = 1$, and for any $\mathcal{V} \subseteq \{1, \dots, n\}$ with $|\mathcal{V}| = t + 1$ it holds that $\text{VSSReconstruct}(\mathcal{V}, \{s_i\}_{i \in \mathcal{V}}) = s$.
- **Commitment.** If the dealer D is corrupt, then there exists a share s_i such that $\text{VSSShareVerify}(s_i, C) = 0$ and the VSSReconstruct algorithm outputs \perp with overwhelming probability.

The most commonly used VSS constructions are Feldman's [Fel87] and Pedersen's [Ped91] construction due to their simplicity. Below we briefly present these constructions.

First, we describe the VSS construction of Feldman [Fel87], which provides **Computational Secrecy**. At a high level, to prevent a malicious dealer from distributing false shares (distributing the shares that does not lie in a polynomial of degree t), he is forced to commit his polynomial coefficients a_i by broadcasting $C_i = g^{a_i} \forall i \in \{0, 1, \dots, t\}$ before distributing the shares. After all the commitments $\{C_i\}_{i=0}^t$ are distributed, then the dealer proceeds to distribute the shares as he did in Shamir secret sharing schme. Since the coefficients a_i have

been committed, a participant P_i who wishes to check whether his share s_i is equal to $f(i)$ can take a look at the following equality

$$g^{s_i} = g^{f(i)} = g^{\sum_{k=0}^t a_k i^k} = \prod_{k=0}^t (C_k)^{i^k}. \quad (2.1)$$

Because P_i can calculate $(C_k)^{i^k}$ by himself, hence he can use (2.1) successfully verify whether his share s_i is correct. In addition, when P_i reveals his share s_i , other participants can also check whether $s_i = f(i)$ the same way by using (2.1), indicating that the share revealed by P_i is indeed valid. Finally, if there are at least $t + 1$ valid shares revealed, the secret s can be reconstructed the same way as in Shamir secret sharing scheme. With this idea, the VSS scheme is described in **Protocol 5** below.

Protocol 5 Feldman's Verifiable Secret Sharing Scheme

Let \mathbb{G} be a cyclic group with order p and generator g . Feldman's VSS scheme is described as follows.

VSSShare $\langle D(s), \{P_i\}_{i=1}^n \rangle :$

The dealer randomly chooses a polynomial $f(x) = a_0 + a_1x + a_2x^2 + \dots + a_tx^t$ such that $a_0 = s$. Then he compute $s_i = f(i)$ The dealer then broadcasts

$$C_i = g^{a_i} \quad \forall i \in \{0, 1, \dots, t\}.$$

The dealer then gives the participant P_i the share s_i . The commitment C is $C = \{C_i\}_{i=0}^t$

VSSShareVerify $(s_i, C) :$

Participant P_i can verify whether s_i is his secret ($s_i = f(i)$) by checking

$$g^{s_i} \stackrel{?}{=} g^{f(i)} = g^{\sum_{k=0}^t a_k i^k} = \prod_{k=0}^t (C_k)^{i^k}. \quad (2.2)$$

If the check fails, complain against D . The dealer, who receives the revealed share s_i that satisfies (2.3). If the revealed shares do not satisfy the equation, then P_i outputs 0. Otherwise, P_i outputs 1.

Protocol 5 Feldman's Verifiable Secret Sharing Scheme (Continued)

VSSReconstruct($\mathcal{V}, \{s_i\}_{i \in \mathcal{V}}, C$) :

From a set $\mathcal{V} \in \{1, \dots, n\}$ with $|\mathcal{V}| \geq t + 1$ and secrets $\{s_i\}_{i \in \mathcal{V}}$ submitted by P_i for $i \in \mathcal{V}$, the algorithm proceeds as follows:

1. For each $i \in \mathcal{V}$ such that $\text{VSSShareVerify}(s_i, C) = 0$, set $\mathcal{V} := \mathcal{V} \setminus \{i\}$.
2. After step 1, if $|\mathcal{V}| \leq t$, then the algorithm outputs \perp .
3. Otherwise, the secret s can be computed via Lagrange interpolation as follows.

$$s = f(0) = \sum_{i \in \mathcal{V}} f(i) \cdot \lambda_{i, \mathcal{V}} = \sum_{i \in \mathcal{V}} s_i \cdot \lambda_{i, \mathcal{V}}.$$

Example 2.4.9. We give an example of the process of the VSS scheme of Feldman above.

Let \mathbb{G} be the group of the group of elliptic curve $y^2 = x^3 + 7$ over \mathbb{F}_{43} . The order p of \mathbb{G} is 31 and its generator is $g = (7, 7)$. We set $n = 4$ and $t = 2$.

VSSShare : This phase proceeds as follows.

1. Suppose a dealer has a secret $s = 1$ and wants to split s among participants P_1, P_2, P_3 and P_4 . He chooses $f(x) = x^2 + x + 1$.
2. The dealer then publishes $C_0 = (7, 7)$, $C_1 = (7, 7)$, $C_2 = (7, 7)$.
3. His shares are $s_1 = 3$, $s_2 = 7$, $s_3 = 13$, $s_4 = 21$.

VSSShareVerify : For simplicity, we only demo the checking process of P_1 and P_2 .

1. P_1 check if $(7, 7) \cdot 3 \stackrel{?}{=} (7, 7) \cdot 1 + (7, 7) \cdot 1^1 + (7, 7) \cdot 1^2$. Since they are both equal to $(29, 31)$, he received a valid share, hence P_1 outputs 1.
2. P_2 check if $(7, 7) \cdot 7 \stackrel{?}{=} (7, 7) \cdot 1 + (7, 7) \cdot 2^1 + (7, 7) \cdot 2^2$. Since they are both equal to $(29, 31)$, he received a valid share, hence P_2 outputs 1.
3. Similarly, P_3 and P_4 check their respective shares and confirm they received valid results, and output 1.

VSSReconstruct : Given $\mathcal{V} = \{1, 2, 3, 4\}$ and the shares $s_1 = 3$, $s_2 = 7$, $s_3 = 13$, $s_4 = 20$, this algorithm proceeds as follows:

1. For the share $s_4 = 20$. Everyone else run `VSSShareVerify` and see that $(7, 7) \cdot 20 = (20, 40)$, while $(7, 7) \cdot 1 + (7, 7) \cdot 4 + (7, 7) \cdot 16 = (40, 25)$. Thus s_4 is invalid and now $\mathcal{V} = \{1, 2, 3\}$.
2. For the share $s_1 = 3$, Everyone else run `VSSShareVerify` and check that $(7, 7) \cdot 3 \stackrel{?}{=} (7, 7) \cdot 1 + (7, 7) \cdot 1^1 + (7, 7) \cdot 1^2$. Since they are both equal to $(29, 31)$, s_1 is a valid share. Similarly, s_2 and s_3 submit a valid share and pass the verification step.
3. Since s_1, s_2 and s_3 are valid shares, we have $|\mathcal{V}| = 3 > t$ and thus the secret s reconstructed using Lagrange interpolation as follows.

$$s = 3(2 \cdot 3)(1 \cdot 2)^{-1} + 7(1 \cdot 3)((-1) \cdot 1)^{-1} + 13(1 \cdot 2)((-2)(-1))^{-1} \pmod{31} = 1.$$

It should be noted that Feldman's scheme is only secure against **computationally bounded** adversaries. This is because, in Feldman's construction, the adversary is given $C_0 = g^{s_0}$. Hence, an unbounded adversary can find s_0 by brute force given C_0 . In some cases, we would like to construct a VSS scheme that is **perfectly secure**, i.e., even a computationally unbounded adversary cannot learn the secret share s_0 . Fortunately, in [Ped91], the author proposed a VSS scheme that is secure against any adversaries, even an unbounded one. The idea of Pedersen's VSS scheme is to add a random polynomial $f'(x)$ as follows

$$f'(x) = b_0 + b_1x + \dots + b_tx^t.$$

With this polynomial, the commitment C_i of Pedersen's VSS is recalculated to be

$$C_i = g^{a_i} h^{b_i} \forall i \in \{0, 1, \dots, t\}.$$

In this way, due to the uniform distribution of $b_i \in \mathbb{Z}_p$, then for each i , C_i now looks uniformly distributed in \mathbb{G} from the view of any adversaries because it is "protected" by h^{b_i} . Hence the adversary cannot brute force s_0 given C_0 like he did in Feldman's VSS scheme. With the discussions above, we now give a detailed description of this construction in **Protocol 6**. The security proof of Pedersen's VSS scheme can be found in [Ped91], where the author formally proved that the scheme indeed provides perfect secrecy.

Protocol 6 Pedersen's Verifiable Secret Sharing Scheme

Let \mathbb{G} be a cyclic group with order p and generators g and h . Pedersen's VSS scheme is described as follows.

VSSShare $\langle D(s), \{P_i\}_{i=1}^n \rangle :$

The dealer randomly chooses two polynomials $f(x) = a_0 + a_1x + a_2x^2 + \dots + a_tx^t$ and $f'(x) = b_0 + b_1x + b_2x^2 + \dots + b_tx^t$ such that $a_0 = s$. Then he compute $s_i = f(i)$ The dealer then broadcasts

$$C_i = g^{a_i} h^{b_i} \forall i \in \{0, 1, \dots, t\}.$$

Finally, he gives the participant P_i the tuple (s_i, s'_i) . The commitment C is $C = \{C_i\}_{i=0}^t$.

VSSShareVerify $((s_i, s'_i), C) :$

Participant P_i can verify whether (s_i, s'_i) is his secret ($s_i = f(i)$ and $s'_i = f'(i)$) by checking

$$g^{s_i} h^{s'_i} \stackrel{?}{=} g^{f(i)} h^{f'(i)} = \prod_{j=0}^t (g^{a_j} h^{b_j})^{i^j} = \prod_{k=0}^t (C_k)^{i^k}. \quad (2.3)$$

If the check fails, complain against D . The dealer, who receives reveals the share s_i that satisfies (2.3). If the revealed shares do not satisfy the equation, then P_i outputs 0. Otherwise, P_i outputs 1.

VSSReconstruct $(\mathcal{V}, \{(s_i, s'_i)\}_{i \in \mathcal{V}}, C) :$

From a set $\mathcal{V} \in \{1, \dots, n\}$ with $|\mathcal{V}| \geq t + 1$ and shares $\{(s_i, s'_i)\}_{i \in \mathcal{V}}$ submitted by participants P_i for $i \in \mathcal{V}$, the algorithm proceeds as follows:

1. For each $i \in \mathcal{V}$ such that $\text{VSSShareVerify}((s_i, s'_i), C) = 0$, set $\mathcal{V} := \mathcal{V} \setminus \{i\}$.
2. After step 1, if $|\mathcal{V}| \leq t$, then the algorithm outputs \perp .
3. Otherwise, the secret s can be computed via Lagrange interpolation as follows.

$$s = f(0) = \sum_{i \in \mathcal{V}} f(i) \cdot \lambda_{i, \mathcal{V}} = \sum_{i \in \mathcal{V}} s_i \cdot \lambda_{i, \mathcal{V}}.$$

Example 2.4.10. We give an example of the process of the VSS scheme of Pedersen.

Let \mathbb{G} be the group of the group of elliptic curve $y^2 = x^3 + 7$ over \mathbb{F}_{43} . The order p of \mathbb{G} is 31 and its generators are $g = (7, 7)$ and $h = (42, 7)$. We set $n = 4$ and $t = 2$.

VSSShare : *This phase proceeds as follows.*

1. Suppose a dealer has a secret $s = 1$ and wants to split s among participants P_1, P_2, P_3 and P_4 . He chooses $f(x) = x^2 + x + 1$ and $f'(x) = 2x + 1$.
2. The dealer then publishes $C_0 = (2, 12)$, $C_1 = (37, 36)$, $C_2 = (20, 3)$.
3. His shares are as follows: $(s_1, s'_1) = (3, 6)$, $(s_2, s'_2) = (7, 13)$, $(s_3, s'_3) = (13, 24)$, and $(s_4, s'_4) = (21, 8)$.

VSSShareVerify : *For simplicity, we only demo the checking process of P_1 and P_2 .*

1. P_1 check if $(7, 7) \cdot 3 + (42, 7) \cdot 6 \stackrel{?}{=} (2, 12) \cdot 1 + (37, 36) \cdot 1 + (20, 3) \cdot 1$. Since they are both equal to $(21, 18)$, he received a valid share. Hence P_1 outputs 1.
2. P_2 check if $(7, 7) \cdot 7 + (42, 7) \cdot 13 \stackrel{?}{=} (2, 12) \cdot 1 + (37, 36) \cdot 2 + (20, 3) \cdot 2^2$. Since they are both equal to $(21, 18)$, he received a valid share. Hence P_2 outputs 1.
3. Similarly, P_3 and P_4 check their respective shares and confirm they received valid results and they all output 1.

VSSReconstruct : *Given the share tuples $(s_1, s'_1) = (3, 6)$, $(s_2, s'_2) = (7, 13)$, $(s_3, s'_3) = (13, 24)$, $(s_4, s'_4) = (20, 8)$, the algorithm proceeds as follows:*

1. For the share (s_4, s'_4) , everyone else run **VSSShareVerify** and see that $(7, 7) \cdot 20 + (42, 7) \cdot 8 = (21, 25)$, while $(2, 12) \cdot 1 + (37, 36) \cdot 4 + (20, 3) \cdot 16 = (7, 36)$. Hence (s_4, s'_4) is an invalid share tuple and now $\mathcal{V} = \{1, 2, 3\}$.
2. For the share (s_1, s'_1) , Everyone run **VSSShareVerify** to check if $(7, 7) \cdot 3 + (42, 7) \cdot 6 \stackrel{?}{=} (2, 12) \cdot 1 + (37, 36) \cdot 1 + (20, 3) \cdot 1$. Since they are both equal to $(21, 18)$, the tuple defines a valid share. Similarly, (s_2, s'_2) and (s_3, s'_3) are valid shares and pass the verification step.
3. Since s_1, s_2 and s_3 are valid shares, we have $|\mathcal{V}| = 3 > t$ and thus the secret s can be reconstructed using Lagrange interpolation as follows

$$s = 3(2 \cdot 3)(1 \cdot 2)^{-1} + 7(1 \cdot 3)((-1) \cdot 1)^{-1} + 13(1 \cdot 2)((-2)(-1))^{-1} \pmod{31} = 1.$$

2.4.7.3 Publicly Verifiable Secret Sharing

Recall that, in a verifiable secret sharing scheme, each participant is able to check the correctness of his shared value. However, he will not be able to check the correctness of other participants' shared values, making the scheme only **privately verifiable**. Hence, Stadler [Sta96] proposed a Publicly Verifiable Secret Sharing (PVSS) scheme that allows anyone (not just the shareholders) can **publicly verify** that the secret was shared properly and be assured that it is recoverable.

Now we will formally describe the syntax of a PVSS scheme, which is different from an ordinary VSS scheme to ensure that each share can be publicly verified. The security properties of a PVSS are identical to an ordinary VSS, hence we do not present them here.

Definition 2.4.7 (Publicly Verifiable Secret Sharing). A (t, n) -**publicly verifiable secret sharing scheme** between a dealer D and participants P_1, P_2, \dots, P_n consists of an interactive protocol PVSSShare and three algorithms PVSSShareVerify , PVSSShareReveal , PVSSReconstruct defined as follows.

1. $(C, \{(E_i, \pi_i)\}_{i=1}^n) \leftarrow \text{PVSSShare} \langle D(S), \{P_i(\text{pk}_i, \text{sk}_i)\}_{i=1}^n \rangle$: This is an interactive protocol between D , who holds S and P_1, \dots, P_n , where each P_i holds a public-secret key pair $(\text{pk}_i, \text{sk}_i)$. At the end of the protocol, a commitment C , a list of encrypted shares $\{E_i\}_{i=1}^n$ and their proof of correct encryption $\{\pi_i\}_{i=1}^n$ is publicly known to all participants.
2. $b \leftarrow \text{PVSSShareVerify}(E_i, \text{pk}_i, \pi_i, C)$: This is a deterministic algorithm run by participants to verify the correctness of E_i . On input an encrypted share E_i , a public key pk_i , a proof π_i and a commitment C , it outputs a bit $b \in \{0, 1\}$.
3. $(S_i, \pi'_i) \leftarrow \text{PVSSShareReveal}(E_i, \text{sk}_i, \text{pk}_i)$: This algorithm is run by each participant P_i to decrypt E_i . On input an encrypted share E_i , a secret key sk_i , and a public key pk_i , it outputs the share S_i and a proof π'_i of correct decryption.
4. $S / \perp \leftarrow \text{PVSSReconstruct}(\mathcal{V}, \{(S_i, \text{pk}_i, \pi'_i, E_i)\}_{i \in \mathcal{V}})$: This algorithm is run by participants. From a set \mathcal{V} , a set of tuples $(S_i, \text{pk}_i, \pi'_i, E_i)$ each consisting of a share S_i , its encrypted share E_i , a proof π'_i and a public key pk_i for all $i \in \mathcal{V}$, this algorithm returns a value S , which is the original secret of the dealer, or \perp .

The simplest and most commonly used PVSS construction is of Schoenmakers [Sch99]. Below we will formally describe his construction in **Protocol 7** and give some remarks.

Protocol 7 Schoenmakers 's Publicly Verifiable Secret Sharing Scheme

Let \mathbb{G} be a group with a prime order p and generators g and h . Schoenmakers 's PVSS scheme is described as follows.

PVSSShare $\langle D(S), \{P_i(\mathbf{pk}_i = h^{\mathbf{sk}_i}, \mathbf{sk}_i)\}_{i=1}^n \rangle :$

This is an interaction between D , who holds a seed $s \in \mathbb{Z}_p$ along the secret $S = h^s$, and participants P_i who hold a secret key \mathbf{sk}_i and public key $\mathbf{pk}_i = h^{\mathbf{sk}_i}$. The interaction proceeds as follows.

1. The dealer chooses two polynomials $f(x) = a_0 + a_1x + a_2x^2 + \dots + a_tx^t$ such that $a_0 = s$. Recall that the secret to be shared is $S = h^s$, not s . Then he computes $s_i = f(i) \forall i \in \{1, \dots, N\}$.
2. The dealer then computes the commitments and encrypted shares as follows.

$$\begin{cases} C_i = g^{a_i} \forall i \in \{0, 1, \dots, t\}, \\ E_i = \mathbf{pk}_i^{s_i} = h^{\mathbf{sk}_i \cdot s_i} \forall i \in \{1, \dots, n\}. \end{cases}$$

To prove the correctness of encryption, i.e, $E_i = \mathbf{pk}_i^{s_i}$, D then additionally compute the proof of correct encryption as follows.

$$\begin{cases} A_i = g^{s_i} = \prod_{j=0}^t g^{a_j j^t} = \prod_{j=0}^t C_j^{i^j}, \\ \pi_i \leftarrow \text{NIZKDLOGProve}(g, A_i, \mathbf{pk}_i, E_i, s_i), \end{cases}$$

where **NIZKDLOGProve** was described in **Algorithm 2**. Finally, he publicly broadcasts $C = \{C_i\}_{i=0}^t$ and $(E_i, \pi_i)_{i=1}^n$.

PVSSShareVerify $(E_i, \mathbf{pk}_i, \pi_i, C = \{C_i\}_{i=0}^t) :$

This algorithm is run by each participant. Any participant, on input a value E_i a proof π_i and a commitment C , can check the correctness of E_i by computing

$$A_i = g^{s_i} = \prod_{j=0}^t g^{a_j j^t} = \prod_{j=0}^t C_j^{i^j}.$$

Since $A_i = g^{s_i}$, given g and \mathbf{pk} , recall that P_i can simply check whether $E_i \stackrel{?}{=} \mathbf{pk}_i^{s_i}$ by computing

$$b = \text{NIZKDLOGVerify}(g, A_i, \mathbf{pk}_i, E_i, \pi_i)$$

where **NIZKDLOGVerify** was described in **Algorithm 3**. Finally, the value b is the output of the algorithm.

Protocol 7 Schoenmakers 's Publicly Verifiable Secret Sharing Scheme (Continued)

PVSSShareReveal($E_i, \text{sk}_i, \text{pk}_i$) :

This algorithm is run by each participant. Participant P_i , on input an encrypted share E_i , a secret key sk_i , proceeds as follows:

1. Compute the share S_i as follows.

$$S_i = E_i^{1/\text{sk}_i} = h^{s_i}.$$

2. Compute the proof of correct decryption π'_i . Note that, given E_i , h and $\text{pk}_i = h^{\text{sk}_i}$, then P_i can prove that $S_i^{\text{sk}_i} = E_i$ by running the algorithm

$$\pi'_i \leftarrow \text{NIZKDLOGProve}(h, \text{pk}_i, S_i, E_i, \text{sk}_i).$$

Finally, the algorithm outputs (S_i, π'_i) .

PVSSReconstruct($\mathcal{V}, \{(S_i, \text{pk}_i, \pi'_i, E_i)\}_{i \in \mathcal{V}}$) :

This algorithm is run by participants. From a set \mathcal{V} and decrypted shares $\{S_i\}_{i \in \mathcal{V}}$ and their proof $\{\pi'_i\}_{i \in \mathcal{V}}$ and encrypted shares $\{E_i\}_{i \in \mathcal{V}}$, this algorithm proceed as follows:

1. For each $i \in \mathcal{V}$, compute $b_i = \text{NIZKDLOGVerify}(h, \text{pk}_i, S_i, E_i, \pi'_i)$ to verify the correctness of the revealed share S_i . If $b_i = 0$ then set $\mathcal{V} := \mathcal{V} \setminus \{i\}$.
2. After step 1, if $|\mathcal{V}| \leq t$, then the algorithm outputs \perp .
3. Otherwise, the secret S is reconstructed via Lagrange interpolation as follows.

$$S = h^s = h^{\sum_{i \in \mathcal{V}} s_i \cdot \lambda_{i, \mathcal{V}}} = \prod_{i \in \mathcal{V}} h^{s_i \cdot \lambda_{i, \mathcal{V}}} = \prod_{i \in \mathcal{V}} S_i^{\lambda_{i, \mathcal{V}}}.$$

Let us analyze how Schoenmakers 's PVSS is publicly verifiable. Compared to previous VSS schemes, one can see that in Schoenmakers 's protocol (or in the general syntax of a PVSS scheme), each participant P_i additionally holds a public-secret key pair $(\text{pk}_i, \text{sk}_i)$, and the dealer D no longer sends S_i secretly to P_i , but instead sends the encrypted share E_i of S_i which is publicly known to everyone. With the public key pk_i , the dealer can compute the encrypted value E_i of the i -th share S_i and send the proof $\pi_i = \text{NIZKDLOGProve}(g, c_i, \text{pk}_i, E_i, s_i)$ of correct encryption. This allow anyone, not just P_i , to verify the correctness of S_i using E_i

and π_i and commitments $\{C_i\}_{i=0}^t$ by executing $\text{NIZKDLOGVerify}(g, c_i, \text{pk}_i, E_i, \pi_i)$. Hence, the correctness of the shares can be publicly verified by anyone. However, only P_i , who processes the secret key sk_i , can decrypt E_i to learn his share S_i , while other participants do not learn anything about S_i .

Example 2.4.11. We give an example of the process of the PVSS scheme of Schoenmakers .

Let \mathbb{G} be the group of the group of elliptic curve $y^2 = x^3 + 7$ over \mathbb{F}_{43} . The order p of \mathbb{G} is 31 and its generators are $g = (7, 7)$ and $h = (42, 7)$. We set $n = 4$ and $t = 2$.

PVSSShare : This phase proceeds as follows.

1. Initially, suppose the secret and public keys of the participants are as follows.

Participant	P_1	P_2	P_3	P_4
sk_i	$\text{sk}_1 = 1$	$\text{sk}_2 = 2$	$\text{sk}_3 = 3$	$\text{sk}_4 = 4$
$\text{pk}_i = h^{\text{sk}_i}$	$\text{pk}_1 = (42, 7)$	$\text{pk}_2 = (40, 18)$	$\text{pk}_3 = (2, 31)$	$\text{pk}_4 = (20, 40)$

2. Suppose a dealer choose a value $s = 1$ and wants to split s among participants P_1, P_2, P_3 and P_4 . His secret to be shared is $S = (42, 7) \cdot 1 = (42, 7)$. He chooses $f(x) = x^2 + x + 1$.

3. For each i process of creating the proof π_i of the correctness of E_i can be found in Example 2.4.6, hence we omit it here for simplicity. The values of C_i are as follows.

Index	0	1	2
a_i	$a_0 = 1$	$a_1 = 1$	$a_2 = 1$
$C_i = g^{a_i}$	$C_0 = (7, 7)$	$C_1 = (7, 7)$	$C_2 = (7, 7)$

4. In addition, the values E_i are as follows.

Participant	P_1	P_2	P_3	P_4
$s_i = f(i)$	$s_1 = 3$	$s_2 = 7$	$s_3 = 13$	$s_4 = 21$
$E_i = \text{pk}_i^{s_i}$	$\text{pk}_1 = (2, 31)$	$\text{pk}_2 = (38, 22)$	$\text{pk}_3 = (13, 22)$	$\text{pk}_4 = (35, 21)$

PVSSShareVerify : Participants proceed to verify the correctness of E_i and C_i . The process verification is just executing the NIZKDLOGVerify and its steps can be found in Example 2.4.6. Hence, we omit this step for simplicity.

PVSSShareReveal : In this step, participant P_i reveal his share S_i and a proof π'_i . Again, we omit the creation of π'_i . The shares S_i of P_i are as follows.

Participant	1	2	3	4
$S_i = E_i^{1/\text{sk}_i}$	$S_1 = (2, 31)$	$S_2 = (32, 40)$	$S_3 = (29, 31)$	$S_4 = (25, 25)$

PVSSReconstruct: *In this phase, suppose P_1, P_2, P_3, P_4 have broadcasted their shares as in the previous step. We see that all the shares are valid, hence the original secret S of the dealer can be reconstructed as*

$$\begin{aligned} S = & (2, 31) \cdot (2 \cdot 3)(1 \cdot 2)^{-1} + (32, 40) \cdot (1 \cdot 3)((-1) \cdot 1)^{-1} \\ & + (29, 31) \cdot (1 \cdot 2)((-2)(-1))^{-1} = (42, 7). \end{aligned}$$

Finally, we would like to note that PVSS is one of the cryptographic primitives used in constructing DRNGs. The inner structure Schoenmakers ’s construction above will be used for analyzing the construction of many DRNGs in Section 3.2.2.

2.4.8 Random Oracle Model

The **random oracle model** was introduced by Bellare and Rogaway [BR93]. The idea of a random oracle model is to provide all participants of a protocol with access to a public function H and then prove the protocol to be secure, assuming that H maps each input to a truly random output. Later, in practice, one set H to be a function derived in some way from a standard cryptographic hash function like SHA256, Keccak or others. It is clear though that any specific function will not be random because it is deterministic, i.e., it returns the same value when given the same input. With the discussions above, one can realize the random oracle model H as follows:

1. For an input X , if $H(X)$ is not known previously, one choose a random value Y and set $H(X) := Y$. This captures the condition that H each input to a truly random output.
2. If the evaluation of H at input X has been asked, then it remains the same value every time X is queried. This captures the condition that H is deterministic.

The random oracle model buys efficiency and, as Rogaway claims, security guarantees, which, although not at the same level as those provided by the standard “provable security approach,” are arguably superior to those provided by a totally ad hoc protocol design. One might be skeptical that a security proof in the oracle model does not have any meaning since the function H used in the final protocol is not random. However, in practice, attacks on schemes or protocol involving a SHA256 derived H will treat H as random. Hence, we can apply the random oracle methodology to show that such attacks will fail.

An important result of random oracle model was taken by Canetti *et al.* in [vTJ11]. They showed that there exist protocols that are secure in the random oracle model but are not secure under instantiations by a function ensemble.

In comparison with a totally ad hoc design, proof in the random oracle model has the benefit of judging the protocol under a strong, formal notion of security, even if this assumes some underlying primitive to be very strong. This is better than not formally modeling the security of the protocol at all. In our thesis, the random oracle will be used for proving the security proof of the DVRF protocol in Section 4.6.

2.4.9 Homomorphic Encryption

In this subsection, we give a brief overview to homomorphic encryption, which serves as one of the cryptographic primitives in constructing DRNGs in Subsection 3.2.5.

2.4.9.1 Encryption Scheme

Loosely speaking, encryption schemes are supposed to enable private exchange of information between participants who communicate over an insecure channel. The goal of an encryption scheme is to let a sender transfer information to the receiver over an insecure channel without letting adversaries figure out this information. To do so, the sender must transform the message into a corresponding ciphertext such that the receiver can retrieve the message from the ciphertext, but the adversary cannot do so. Clearly, to retrieve the message from the cipher text, the receiver must process something that adversaries do not have. This thing is called a **key**. Now we will how an encryption scheme works.

At a high level, let M be the message, an encryption scheme consists of an **encryption algorithm** that uses an encryption key ek to transform M into a ciphertext C and a **decryption algorithm** that use a decryption key dk to transform C back to M . There are two types of encryption schemes: **symmetric encryption scheme** where $ek = dk$ and **public key encryption scheme** where $ek \neq dk$. We are interested in the latter type. Below we give a formal definition of encryption schemes.

Definition 2.4.8 (Encryption Scheme). Let \mathcal{M} be the message space and \mathcal{C} be the ciphertext space. A **encryption scheme** consists of three algorithms (**Setup**, **Enc**, **Dec**) as follows:

1. $(\mathbf{ek}, \mathbf{dk}) \leftarrow \text{Setup}(1^\lambda)$: On input a security parameter λ , this algorithm returns an encryption key \mathbf{ek} and a decryption key \mathbf{dk} .
2. $C \leftarrow \text{Enc}(M, \mathbf{ek})$: On input a message $M \in \mathcal{M}$ and an encryption key \mathbf{ek} , this algorithm returns a ciphertext $C \in \mathcal{C}$.
3. $M \leftarrow \text{Dec}(C, \mathbf{dk})$: On input a ciphertext $C \in \mathcal{C}$ and a decryption key \mathbf{dk} , this algorithm outputs a message $M \in \mathcal{M}$.

We would like an encryption scheme to achieve two required properties:

- **Correctness.** For all $M \in \mathcal{M}$ and $(\mathbf{ek}, \mathbf{dk}) \leftarrow \text{Setup}(1^\lambda)$ and $C \leftarrow \text{Enc}(M, \mathbf{ek})$, it holds that $\text{Dec}(C, \mathbf{dk}) = M$.
- **IND-CPA Security.** For all adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ with polynomial running time in λ , it holds that the advantage

$$\text{ADV}_{\text{IND-CPA}}(\mathcal{A}) = \left| \Pr[\text{ExpINDCPA}_{\text{Enc}}^{\mathcal{A}}(1^\lambda) = 1] - \frac{1}{2} \right|$$

is negligible in λ , where the experiment $\text{ExpINDCPA}_{\text{Enc}}^{\mathcal{A}}(1^\lambda)$ is defined as follows.

$(\mathbf{ek}, \mathbf{dk}) \leftarrow \text{Setup}(1^\lambda).$
 $(M_0, M_1) \leftarrow \mathcal{A}_1(\mathbf{ek}).$
 $C_0 \leftarrow \text{Enc}(M_0, \mathbf{ek}).$
 $C_1 \leftarrow \text{Enc}(M_1, \mathbf{ek}).$
 $b \xleftarrow{\$} \{0, 1\}.$
 $b' \leftarrow \mathcal{A}_2(C_b).$
 Return $b \stackrel{?}{=} b'.$

Figure 2.3: *Experiment* $\text{ExpINDCPA}_{\text{Enc}}^{\mathcal{A}}(1^\lambda)$

For more information about encryption schemes, we refer the reader to [Gol01].

2.4.9.2 Homomorphic Encryption

The term **homomorphic encryption** describes a class of encryption algorithms that allows one to perform operations on encrypted data (such as additions or multiplications). The result of such a computation remains in encrypted form, and can at a later point be revealed

by the owner of the secret key.

The concept of homomorphic encryption was first introduced by Rivest et al. in 1978 right after the introduction of public key cryptography. Since then, many cryptosystems, such as ElGamal [Gam85] or Paillier [Pai99], have been developed and support either addition or multiplication of encrypted data. As we will see in Section 3.2.5, homomorphic encryption, specifically the ElGamal construction, which will be given in Example 2.4.12 will be one of the cryptographic primitives for constructing DRNGs. Now, we give the formal definition of a homomorphic encryption scheme.

Definition 2.4.9 (Homomorphic Encryption). Assume that the message space (\mathcal{M}, \cdot) forms a group and the set of ciphertext space $(\mathcal{C}, *)$ forms a group. A **homomorphic encryption** scheme over \mathcal{M} and \mathcal{C} is an encryption scheme with the following property: For all messages $M_1, M_2 \in \mathcal{M}$, it holds that

$$\text{Enc}(M_1, \text{ek}) * \text{Enc}(M_2, \text{ek}) = \text{Enc}(M_1 \cdot M_2, \text{ek}).$$

Example 2.4.12. We give an example of a homomorphic encryption scheme. Let us continue with the ElGamal encryption scheme described in Example 2.4.3, since we have already defined the scheme there. Let \mathbb{G} to be a cyclic group with order p and generator g . Recall that in the ElGamal cryptosystem, the setup and encryption algorithm **Setup** and **Enc** is follows:

Setup(1^λ) : This algorithm sample $\text{dk} \in \mathbb{Z}_p$ and let the decryption key to be dk . The public key is $\text{ek} = g^{\text{dk}}$.

Enc(M, ek) : On input a message M and public key ek , it sample $s \in \mathbb{Z}_p$ and return the ciphertext $C = (g^s, M \cdot \text{ek}^s)$.

Dec(C, dk) : On input a ciphertext $C = (C_1, C_2)$ and a decryption key dk , output $M = C_2(C_1^{\text{dk}})^{-1}$. We see that $C_2(C_1^{\text{dk}})^{-1} = M \cdot g^{\text{dk} \cdot s} (g^{s \cdot \text{dk}})^{-1} = M$.

For any messages M_1 and M_2 , it holds that

$$\begin{aligned} \text{Enc}(M_1, \text{ek}) \cdot \text{Enc}(M_2, \text{ek}) &= (g^{s_1}, M_1 \cdot \text{ek}^{s_1}) \cdot (g^{s_2}, M_2 \cdot \text{ek}^{s_2}) \\ &= (g^{s_1+s_2}, M_1 \cdot M_2 \cdot \text{ek}^{s_1+s_2}) \\ &= \text{Enc}(M_1 \cdot M_2, \text{ek}). \end{aligned}$$

Hence we see that the ElGamal cryptosystem above satisfies the required property of a homomorphic encryption scheme.

2.4.10 Threshold Signature

In this subsection, we give a brief overview to threshold signature scheme, which serves as one of the cryptographic primitives in constructing DRNGs in Subsection 3.2.3.

2.4.10.1 Signature Scheme

The need to discuss “digital signatures” has arisen with the introduction of computer communication to the business environment (in which participants need to commit themselves to proposals and/or declarations that they make). Discussions of “unforgeable signatures” also took place in previous centuries, but the objects of discussion were handwritten signatures (and not digital ones), and the discussion was not perceived as related to “cryptography.” This was until 1976 when Diffie and Hellman described the notion of a digital signature scheme [DH76]. Informally, a scheme for unforgeable signatures should satisfy the following:

1. Each user can efficiently produce his/her own signature on documents of his/her choice.
2. Every user can efficiently verify whether a given string is a signature of another (specific) user on a specific document. In addition, it must be infeasible for one user to produce signatures of other users on documents that they did not sign.

With the discussions above, the formal definition of digital signatures can be described below.

Definition 2.4.10 (Signature Scheme). Let \mathcal{M} be the message space. A **signature scheme** consists of three algorithms (**Setup**, **Sign**, **Verify**), working as follows:

1. $(\text{pk}, \text{sk}) \leftarrow \text{Setup}(1^\lambda)$: On input a security parameter 1^λ , this algorithm outputs a public key pk and a secret key sk .
2. $\sigma \leftarrow \text{Sign}(M, \text{sk})$: On input a message $M \in \mathcal{M}$ and a secret key sk , this algorithm outputs a signature σ .
3. $b \leftarrow \text{Verify}(M, \sigma, \text{pk})$: On input a message $M \in \mathcal{M}$, a signature σ and a public key pk , this algorithm outputs a bit b certifying the validity of σ .

Additionally, we need a signature scheme to satisfy the following properties:

- **Correctness.** For all $M \in \mathcal{M}$, if $(\text{pk}, \text{sk}) \leftarrow \text{Setup}(1^\lambda)$ and $\sigma = \text{Sign}(M, \text{sk})$, then it holds that $\text{Verify}(M, \sigma, \text{pk}) = 1$.
- **Unforgeability.** For all adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ with polynomial running time in λ , it holds that the advantage

$$\text{ADV}_{\text{Sign}}(\mathcal{A}) = \Pr[\text{ExpForge}_{\text{Sign}}^{\mathcal{A}}(1^\lambda) = 1]$$

is negligible in λ , where the experiment $\text{ExpForge}_{\text{Sign}}^{\mathcal{A}}(1^\lambda)$ is defined as follows.

```

(pk, sk) ← Setup(1λ).
(Mi, σi)i=1t ←  $\mathcal{A}_1^{\mathcal{O}_{\text{Sign}}(\cdot)}(\text{pk})$ .
(M*, σ*) ←  $\mathcal{A}_2(\text{pk}, (M_i, \sigma_i, \pi_i)_{i=1}^t)$ .
If M* = M'i for some i return 0.
If Verify(M*, σ*, pk) ≠ 1 return 0.
Else return 1.

```

Figure 2.4: Experiment $\text{ExpForge}_{\text{Sign}}^{\mathcal{A}}(1^\lambda)$:

In the experiment above, the oracle $\mathcal{O}_{\text{Sign}}(M)$ is queried by the adversary \mathcal{A} to get the signature σ given the message M . The oracle is formally defined as follows.

```

σ ← Sign(M, sk).
Return σ.

```

Figure 2.5: Oracle $\mathcal{O}_{\text{Sign}}(M)$

Finally, for more information about signature schemes, we refer the reader to [Gol01].

2.4.10.2 Threshold Signature Scheme

Threshold signature schemes can be understood as a distributed version of an ordinary signature scheme. Generally, a threshold signature scheme enables participants to share the power to issue digital signatures under a single public key. A threshold t is specified such that any subset of $t + 1$ participants can jointly sign, but any smaller subset cannot. Generally, the goal is to produce signatures that are compatible with an existing centralized signature scheme [GG19]. In a threshold scheme the key generation is replaced by a communication

protocol between the participants and the signing algorithm will be divided into two “smaller” algorithm: one algorithm for producing a “partial” signature and a combined algorithm that combine any $t + 1$ partial signatures into a unique valid signature. However, the verification algorithm remains identical to the verification of a signature issued by a centralized participant.

With the discussion above, we are now ready to formalize the definition of a threshold signature scheme. Note that we will focus on the non-interactive version.

Definition 2.4.11 (Non-interactive Threshold Signature Scheme). A **non-interactive threshold signature scheme** on a set of participants $\mathcal{P} = \{P_1, P_2, \dots, P_n\}$ consists of three algorithms TSSSign , TSSCombine , TSSVerify and a protocol TSSSetup as follows:

1. $((\text{pk}, \text{sk}), (\text{pk}_i, \text{sk}_i)_{i \in \text{QUAL}}, \text{QUAL}) \leftarrow \text{TSSSetup}(1^\lambda) \langle \{P\}_{P \in \mathcal{P}} \rangle$: This is an interactive protocol is run by all participants in \mathcal{P} to determine the member list QUAL for signing. At the end of the interaction, a public-secret key pair (pk, sk) is created, and each participant $P_i \in \text{QUAL}$ also obtains his public-secret key pair $(\text{pk}_i, \text{sk}_i)$ where the secret key sk_i is only known to him.
2. $\sigma_i \leftarrow \text{TSSSign}(M, \text{sk}_i)$: On input a message $M \in \mathcal{M}$ and a secret key sk_i , this algorithm outputs a partial signature σ_i .
3. $\sigma \leftarrow \text{TSSCombine}(\mathcal{V}, \{\sigma_i\}_{i \in \mathcal{V}})$: On input a set \mathcal{V} and a list signatures $\{\sigma_i\}_{i \in \mathcal{V}}$, this algorithm outputs a signature σ . If $\{\sigma_i\}_{i \in \mathcal{V}}$ are all valid signatures of M , then the TSSCombine algorithm must satisfy $\text{TSSCombine}(\mathcal{V}, \{\sigma_i\}_{i \in \mathcal{V}}) = \text{TSSSign}(M, \text{sk})$.
4. $b \leftarrow \text{TSSVerify}(M, \sigma, \text{pk})$: On input a message M , a signature σ , a public key pk , this algorithm outputs a bit b certifying the validity of σ .

The security properties of a (t, n) -non-interactive threshold signature scheme are identical to an ordinary signature scheme defined above, except that the adversary \mathcal{A} is allowed to corrupt t out of n participants. Finally, beyond the application of creating digital signatures, threshold signature is also one of the cryptographic primitives employed in some DRNG constructions for generating randomness, as we will see in Section 3.2.3.

2.4.11 Distributed Key Generation

Distributed key generation is a main component of threshold cryptosystems. It allows n participants to take part and generate a pair of public-secret keys according to the distribution

defined by the underlying cryptosystem without having to rely on a trusted participant (dealer). Each participant in addition receives a partial secret key and a partial public key. While the public key is output in the clear, the private key is maintained as a (virtual) (t, n) -secret shared via a secret sharing scheme, where each share is the partial secret key of a participant [GJKR99]. No adversary is able to learn anything about the secret key if it does not control the required amount of participants. This grand private key can be later used by a **threshold cryptosystem**, e.g., to compute signatures or decryptions, without ever being reconstructed in a single location. In our thesis, DKG is one of the main components of our specified DRNG protocol in Chapter 4.

Below we are going to describe the required security properties of a DKG.

Definition 2.4.12. Let $\text{DKG}(1^\lambda)(\{P_i\}_{i=1}^n)$ be a (t, n) -**distributed key generation** protocol, then $\text{DKG}(1^\lambda)(\{P_i\}_{i=1}^n)$ satisfies the following properties:

- **Correctness.** If $((\text{pk}, \text{sk}), (\text{pk}_i, \text{sk}_i)_{i \in \text{QUAL}}, \text{QUAL}) \leftarrow \text{DKG}(1^\lambda)(\{P_i\}_{i=1}^n)$, then the following conditions must hold:
 1. For any subsets $\mathcal{V} \subseteq \text{QUAL}$ such that $|\mathcal{V}| \geq t + 1$, then from the set of partial secret keys $\{\text{sk}_i\}_{i \in \mathcal{V}}$, it is possible to reconstruct the secret key sk .
 2. All honest participants agree the same value of the public key $\text{pk} = g^{\text{sk}}$.
 3. The secret key sk is uniformly distributed in \mathbb{Z}_p .
- **Computational Secrecy.** For any computational bounded adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ who corrupts up to t participants, it holds that the advantage

$$\text{ADV}_{\text{DKG}}(\mathcal{A}) = \Pr[\text{ExpSec}_{\text{DKG}}^{\mathcal{A}}(1^\lambda) = 1]$$

is negligible in λ , where the experiment $\text{ExpSec}_{\text{DKG}}^{\mathcal{A}}(1^\lambda)$ is defined as follows.

```
|  |
| --- |
| $\text{tr}_1 \leftarrow \mathcal{A}_1(1^\lambda, \mathcal{B})$ |
| $((\text{pk}, \text{sk}), (\text{pk}_i, \text{sk}_i)_{i \in \text{QUAL}}, \text{QUAL}) \leftarrow \text{DKG}(1^\lambda)(\{P_i\}_{i=1}^n)$ |
| $\text{tr}_2 \leftarrow \text{View}(\text{DKG}(1^\lambda)(\{P_i\}_{i=1}^n))$ |
| $\text{sk}^* \leftarrow \mathcal{A}_2(\text{tr}_1, \text{tr}_2, \{\text{sk}_i\}_{i \in \mathcal{B}}, \text{pk}, \{\text{pk}_i\}_{i \in \text{QUAL}})$ |
| Return  $\text{sk}^* \stackrel{?}{=} \text{sk}$ . |

```

Figure 2.6: Experiment $\text{ExpSec}_{\text{DKG}}^{\mathcal{A}}(1^\lambda)$:

where \mathcal{B} denotes the set of participants corrupted by \mathcal{A} and $\text{View}(\text{DKG}(1^\lambda)(\{P_i\}_{i=1}^n))$ denotes the public transcript made by participants in the execution of the DKG. Informally, the experiment above says that given the public keys pk , $\{\text{pk}_i\}_{i \in \text{QUAL}}$ and the public transcript tr_2 of participants in the DKG execution, then \mathcal{A} cannot learn any information about sk , except with negligible probability.

Note that, some DKG protocols, namely [GJKR99] can only achieve computational secrecy, assuming the discrete log problem is hard to solve. This is due to the fact that, given the public key $\text{pk} = g^{\text{sk}}$, a computationally unbounded adversary can find sk by brute force. In Section 4.3, we present an explicit construction of a DKG due to [GJKR99], which will be used in the DRNG protocol we are going to specify in Chapter 4.

2.4.12 Verifiable Random Function

Verifiable random functions were introduced by Micali, Rabin and Vadhan in 1999 [MRV99]. A verifiable random function is a public key version of a pseudo-random function $F_{\text{sk}}(X)$. It produces a pseudo-random output and a proof certifying that the output is computed correctly. A VRF includes a pair of public-secret key pair (pk, sk) . The secret key sk , along with the input is used by the holder to compute the value of a VRF and its proof, while the public key pk is used by anyone to verify the correctness of the computation.

The motivation of VRF is to make the output of pseudo-random functions become publicly verifiable. The issue with traditional pseudo-random functions is that their output cannot be verified without the knowledge of the secret key sk . Thus a malicious adversary, given an input X can choose an output $Y' \neq F_{\text{sk}}(X)$ that benefits him and claim that it is the output $F_{\text{sk}}(X)$ of the function without fear of being detected. VRF solves this by introducing a public key pk and forcing the owner to create a proof π certifying $Y = F_{\text{sk}}(X)$. The proof π together with the public key pk can be used to publicly verify the correctness of Y while these values reveal nothing about sk . On the other hand, the owner can keep the secret key sk to produce numbers indistinguishable from randomly chosen ones. In our thesis, VRF is one of the main cryptographic primitives for constructing DRNGs in Section 3.2.4 and it also one of the main components of our specified DRNG protocol in Chapter 4.

Below we are going to present the syntax and security properties of a VRF scheme.

Definition 2.4.13 (Verifiable Random Function). A **verifiable random function** consists

of three algorithms (VRFSetup, VRF Eval, VRF Verify) working as follows:

1. $(\text{pk}, \text{sk}) \leftarrow \text{VRFSetup}(1^\lambda)$: This algorithm takes as input as a security parameter λ and outputs a key pair (pk, sk) .
2. $(Y, \pi) \leftarrow \text{VRF Eval}(X, \text{sk})$: This algorithm takes as input a secret key sk and a value X and outputs a value $Y \in \{0, 1\}^\lambda$ and a proof π .
3. $b \leftarrow \text{VRF Verify}(\text{pk}, X, Y, \pi)$: This algorithm takes an input a public key pk , a value X , a value Y , a proof π and outputs a bit b that determines whether $Y = \text{VRF Eval}(X, \text{sk})$.

Additionally, we need a VRF to satisfy the following properties:

- **Correctness.** If $(\text{pk}, \text{sk}) \leftarrow \text{VRFSetup}(1^\lambda)$ and $(Y, \pi) = \text{VRF Eval}(\text{sk}, X)$ then it holds that $\text{VRF Verify}(\text{pk}, X, Y, \pi) = 1$.
- **Uniqueness.** There do not exist tuples (Y, π) and Y', π' with $Y \neq Y'$ and

$$\text{VRF Verify}(\text{pk}, X, Y, \pi) = \text{VRF Verify}(\text{pk}, X, Y', \pi') = 1.$$

- **Pseudo-randomness.** For any adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ it holds that the advantage

$$\text{ADV}_{\text{VRF}}(\mathcal{A}) = \left| \Pr[\text{ExpRand}_{\text{VRF}}^{\mathcal{A}}(1^\lambda, 0) = 1] - \Pr[\text{ExpRand}_{\text{VRF}}^{\mathcal{A}}(1^\lambda, 1) = 1] \right|$$

is negligible in λ , where the experiment $\text{ExpRand}_{\text{VRF}}^{\mathcal{A}}(1^\lambda, b)$ is defined as follows.

$(\text{pk}, \text{sk}) \leftarrow \text{VRFSetup}(1^\lambda).$
 $(X^*, \text{st}, (X_i, Y_i)_{i=1}^t) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\text{VRF}}(\cdot)}(\text{pk}).$
 If $X_i = X^*$ for some i return 0.
 $Y_0^* \leftarrow \text{VRF Eval}(X^*, \text{sk}).$
 $Y_1^* \xleftarrow{\$} \{0, 1\}^\lambda.$
 $b' \leftarrow \mathcal{A}_2(Y_b^*, \text{st}, (X_i, Y_i)_{i=1}^t).$
 Return b' .

Figure 2.7: Experiment $\text{ExpRand}_{\text{VRF}}^{\mathcal{A}}(1^\lambda, b)$:

In the experiment above, the oracle $\mathcal{O}_{\text{VRF}}(X)$ is queried by the adversary \mathcal{A} to get the output and proof (Y, π) of the VRF given input X . It is formally defined as follows.

$$(Y, \pi) \leftarrow \text{VRF}(X, \text{sk}).$$

$$\text{Return } (Y, \pi).$$

Figure 2.8: Oracle $\mathcal{O}_{\text{VRF}}(X)$

It is interesting to see that, VRF can be used for signing messages. However, several signing algorithms such as ECDSA cannot be used to build a VRF. For a given message and a secret key, there can be multiple valid signatures, thus an adversarial prover could produce different valid outputs from a given input, and choose the one that benefits him. This contradicts the uniqueness property of VRF.

Remark 2.4.14. The concept of providing proofs in VRF is actually very similar to NIZK. At a high level, we might view VRF as a combination of a PRF and a NIZK, where the output Y of the VRF is actually equal to a pseudo-random $F_{\text{sk}}(X)$ and the proof π certifying $Y = F_{\text{sk}}(X)$ is provided using a NIZKProve algorithm, where (X, Y, pk) is the instance and sk is the witness satisfying a certain relation \mathcal{R} . Finally, given the instance (X, Y, pk) and the proof π , one can just execute the corresponding NIZKVerify algorithm to check whether $Y = F_{\text{sk}}(X)$. Bitansky [Bit20] and Goyal *et al.* [GHKW17] showed that a **non-interactive witness indistinguishable proof** (NIWI), a weaker version of NIZK is actually sufficient to construct a VRF from the idea above, the exact detail of their constructions are more complicated through.

2.4.13 Verifiable Delay Function

Verifiable Delay Functions (VDF) were introduced by Boneh et al [BBBF18] in 2018. Informally, a VDF is a function that requires a specified number of steps to compute its output, even with a large number of parallel processors, but the correctness of the output can be quickly verified. From the time it was proposed, VDF has become an important cryptographic primitive in constructing many DRNGs to prevent withholding attacks of the last participants, as we will see in Section 3.2.6.

While there can be many functions that require a specified step to compute its output, not all of them are qualified to become VDFs. For example, let H be a cryptographic hash function and consider the function $f(X) = H^t(X) = \underbrace{H(H(\dots(H(X))\dots))}_{t \text{ times}}$. Calculating $f(X)$ requires t iterations, even on a parallel computer. However, the function does not satisfy the efficiently verifiable requirement of a VDF, since verification would also require t iterations,

while VDF verification time must be within $\mathcal{O}(\text{poly}(\log t))$.

Below we are going to present the syntax and security properties of a VDF scheme.

Definition 2.4.15 (Verifiable Delay Function). A **verifiable delay function** consists of three algorithms (VDFSetup, VDFEval, VDFVerify) where:

1. $\text{pp} \leftarrow \text{VDFSetup}(1^\lambda, t)$: This algorithm takes as input as a security parameter λ , and outputs public parameter pp .
2. $(Y, \pi) \leftarrow \text{VDFEval}(X, \text{pp}, t)$: This algorithm takes an input a public parameter pp , a value X a time parameter t and outputs a value $Y \in \{0, 1\}^\lambda$ and a proof π .
3. $b \leftarrow \text{VDFVerify}(\text{pp}, X, Y, \pi, t)$: This algorithm takes an input a public parameter pp , a value X , a value Y , a proof π , a time parameter t and outputs a bit b that determines whether $Y = \text{VDFEval}(X, \text{pp}, t)$.

We require a VDF to have the following security properties:

- **Correctness.** For all parameter X, t and $\text{pp} \leftarrow \text{VDFSetup}(1^\lambda, t)$, if $(Y, \pi) = \text{VDFEval}(\text{pp}, X, t)$ then it holds that $\text{VDFVerify}(\text{pp}, X, Y, \pi, t) = 1$.
- **Soundness.** A VDF is sound if every algorithms \mathcal{A} can solve the following problem with negligible probability in λ : Given pp , output X, Y, π such that $(Y, \pi) \neq \text{VDFEval}(\text{pp}, X, t)$ and $\text{VDFVerify}(\text{pp}, X, Y, \pi, t) = 1$.
- **Sequentiality.** A VDF is t -sequentiality if for all algorithms \mathcal{A} with at most $\mathcal{O}(\text{poly}(t))$ parallel processors and runs within time $\mathcal{O}(\text{poly}(t))$, it holds that the advantage

$$\text{ADV}_{\text{VDF}}(\mathcal{A}) = \Pr[\text{ExpSeq}_{\text{VDF}}^{\mathcal{A}}(1^\lambda) = 1]$$

is negligible in λ , where the experiment $\text{ExpSeq}_{\text{VDF}}^{\mathcal{A}}(1^\lambda)$ is defined as follows.

$$\begin{aligned} &\text{pp} \leftarrow \text{VDFSetup}(1^\lambda, t). \\ &X \xleftarrow{\$} \{0, 1\}^\lambda. \\ &(Y, \pi) \leftarrow \mathcal{A}(X, \text{pp}). \\ &\text{Return } (Y, \pi) \stackrel{?}{=} \text{VDFEval}(X, \text{pp}, t). \end{aligned}$$

Figure 2.9: Experiment $\text{ExpSeq}_{\text{VDF}}^{\mathcal{A}}(1^\lambda)$

- **Efficiently Verifiable:** For all X, Y, π, pp , the algorithm $\text{VDFVerify}(\text{pp}, X, Y, \pi, t)$ terminate after $\mathcal{O}(\text{poly}(\lambda, \log t))$ steps.

As discussed earlier, VDF is one of the cryptographic primitives for constructing DRNGs, for which we will delve into the details in Section 3.2.6. For more information about applications and candidate constructions of VDF, we refer the reader to [BBBF18].

2.5 Blockchain

2.5.1 Blockchain Definition

The concept of blockchain was first introduced by an anonymous individual under the alias Satoshi Nakamoto in 2008 [Nak]. According to Nakamoto, blockchain is a platform that enables data to be added to a shared system without relying on a trusted third participant and serves as a public ledger to store all transaction data on the Bitcoin cryptocurrency system.

According to Imran Bashir [Ba117], from a technical perspective, **blockchain** is a distributed, public, and decentralized ledger. The data stored in this ledger is securely protected using cryptographic techniques, making it difficult to alter or delete, and can only be added through consensus mechanisms among participants. This ledger publicly stores transactions that have been processed within the blockchain system. In another sense, blockchain can be seen as a decentralized database that applies various cryptographic techniques to its data. This database is shared with all **nodes** in the blockchain network and forms the **public ledger**. The decentralization of the database is demonstrated by the fact that all nodes in the system store an identical copy of this ledger and automatically download newly executed transactions. Additionally, this new data needs to be processed and subjected to various cryptographic techniques before being updated into the system.

Each node is connected to the network using a server to perform the task of synchronizing transactions with other nodes. Among the nodes, there will be a small group of nodes that carry out additional tasks such as authentication and transaction propagation, known as **miners**. The most recent transactions will be sequentially added to **blocks** by the nodes. To add a new block to the database, each miner has to solve a certain puzzle (also known as **mining**) and publish his result and block proposal. Other nodes verify the validity of the

solution. If the solution is valid, the block is added to the current database.

As discussed, a blockchain can be decomposed into blocks containing the information of transactions. Each block B is identified by its hash value, i.e. $B.h$. To link the blocks together to the chain, each block B' also consists of the hash value of the previous block of the chain, denoted by $B.h^{-1}$. A chain is valid if and only if all $B.h^{-1}$ is equal to the hash value of the previous block.

2.5.2 Blockchain Platforms

As mentioned in Subsection 2.5.1, the concept of Bitcoin was first introduced by Satoshi Nakamoto on October 31, 2008 [Nak] in a scientific paper sent to the general community of cryptography. In early 2009, Nakamoto implemented and released Bitcoin as open-source software. Until now, no one knows whether Nakamoto is an individual or an organization and the true identity of this person remains unknown.

Bitcoin was initially created with the purpose of addressing financial issues, specifically enabling direct payments between two individuals or organizations without the need for an intermediary financial institution. Additionally, the transfer of funds had to ensure resistance to “double spending,” a form of fraud where the same amount of money is spent in two different transactions without the assistance of a third participant or a centralized server. Bitcoin has become a pioneering project in the field of blockchain technology development, also known as Blockchain 1.0.

Recognizing the limitations of Bitcoin in serving only financial purposes and the significant potential of blockchain technology, in 2014, Vitalik Buterin proposed a new blockchain platform called **Ethereum**. This platform became the first to enable programming and execution of **smart contracts**.

2.5.3 Smart Contract

2.5.3.1 History of Smart Contracts

The principle of smart contracts was described by Nick Szabo [N9696] in 1996, long before the appearance of smart contracts. According to Szabo, a smart contract is a programmable transaction protocol designed to enforce the terms of a contract. The goal of a smart contract

is to satisfy the conditions specified in a traditional contract (such as payment terms, privileges, security provisions, regulations, etc.), minimize potential exceptions (including attacks and unexpected incidents), and reduce the need for trust in a third participant. However, at this time, some necessary conditions to operate a smart contract, particularly a decentralized ledger, have not yet materialized.

In 2014, six years after the launch of Bitcoin, the Ethereum blockchain platform emerged, enabling the practical use of smart contracts. As of the current moment, Ethereum holds the second position in total market capitalization and is among the leading blockchains that allow the development of smart contracts.

The development of smart contracts has facilitated the development of Decentralized Applications (Dapps). Dapps are fundamentally similar to traditional web applications. The main difference between them is that, instead of using APIs to access databases like traditional web applications, Dapps utilizes smart contracts to connect and interact with the blockchain.

2.5.3.2 How Smart Contract Works

A smart contract is used to put all the terms signed between the participants of a contract in real life into a transaction, to be placed on the blockchain. The obligations of the participating participants are represented in the form of “if-then” statements. For example, “if participant A transfers money to participant B, then participant A has the right to use the apartment”. The participants can be individuals or organizations, and there can be more than two participants involved in the contract. Whenever the conditions are met, the smart contract will independently execute the transactions and ensure that the original agreements are still adhered to.

A smart contract allows the conversion of various forms of currencies, commodities, real estate, and other types of assets. The machine code of the smart contract will be stored on the blockchain and replicated to all nodes in the network. As such, this machine code, along with its operational history, is recorded and inherits properties similar to a regular transaction (immutable and tamper-resistant). The challenge of digitizing real-world assets for processing within a smart contract is still being addressed.

2.5.3.3 Application of Smart Contracts

The capabilities of smart contracts open up opportunities for many decentralized applications to be realized. State of the DApps ¹ indicates that there are currently 2,113 decentralized applications being built on Ethereum, EOS, and POA; and the range of fields where these applications can be applied is extensive.

At present, the Finance, Exchange, Gaming, and Gambling sectors occupy a significant proportion of the total decentralized applications and the number of contracts being developed across the three platforms. Besides the aforementioned applications, the potential of decentralized applications can also address essential social issues, such as Social Networks, Real Estate, Identity, and more. Although other fields may not have many decentralized applications currently in development, they still showcase the diversity in applying smart contracts to these domains.

¹<https://www.stateofthedapps.com>

Chapter 3

A Comprehensive Survey of Existing DRNGs

Numerous attempts have been made to construct a DRNG. Due to the multitude of existing directions and techniques employed in constructing DRNGs, especially when many new constructions are proposed recently, a systematic literature review is essential to provide readers with a comprehensive overview of these constructions. However, to the best of our knowledge, only one paper [RG22] has so far attempted to give such a review. However, the details of the constructions described in the paper are still rather sketchy, especially in describing PVSS-based constructions. Therefore, in this chapter, we present a more detailed systematic literature review by describing each existing DRNG construction. This review serves as a tutorial, enabling a deeper comprehension of the current state of the problem. Finally, we provide a comparison table for the mentioned constructions. All the materials in this chapter will help us in specifying the most suitable DRNG for blockchain-based application in Chapter 4.

3.1 Decentralized Random Number Generator

Decentralized random number generator (DRNG) was originally introduced by Blum in [Blu83]. A DRNG provides a method for multiple participants to collaboratively agree on a pseudo-random value without the involvement of a central party. DRNGs offer wide-ranging applications in various contexts, including blockchain. For instance, consider a GameFi

developer who builds a blockchain-based poker game. The developer must assure users that the cards were shuffled randomly and are unknown to anyone. In such a game, using randomness produced in a distributed manner is desirable to increase fairness. Furthermore, the random generation process should be verifiable by everyone so that users can be assured that the shuffle process is performed correctly.

Below, we formally define the syntax that captures existing DRNGs and the required security properties of a DRNG protocol. The syntax and formalized security properties are our own formulation, aiming to capture the security properties of DRNGs in literature, which are only presented informally. We would like to note that [RG22, GLOW21] also attempted to propose a definition of DRNG; however, by dividing the generating process into a **tuple of algorithms**, their definition only captures non-interactive DRNGs, where each participant can generate their contribution independently without the need for interaction with others. Consequently, their definition fails to capture constructions that require multi-round interactions between participants, such as SPURT [DKIR22] or RandHerd [SJK⁺17]. Our definition aims to capture all DRNG constructions by using a single interactive protocol DRNGGen to capture the whole output generating process.

3.1.1 Formal Definition

Definition 3.1.1 (Decentralized Random Number Generator). A (t, n) -DRNG protocol on a set of participants $\mathcal{P} = \{P_1, P_2, \dots, P_n\}$ is an epoch-based protocol. Each epoch r consists of two interactive protocols DRNGSetup, DRNGGen, and an algorithm DRNGVerify and a global state \mathbf{st} , working as follows.

1. $(\mathbf{st} = \mathbf{st}_0, \mathbf{QUAL}, \mathbf{pp}, \{\mathbf{sk}_i\}_{i \in \mathbf{QUAL}}) \leftarrow \text{DRNGSetup}(1^\lambda) \langle \{P\}_{P \in \mathcal{P}} \rangle$: This is an interactive protocol run by all participants in \mathcal{P} to determine the member list of qualified committees. At the end of the interaction, a set \mathbf{QUAL} of qualified participants is determined, the \mathbf{st} is initialized to be \mathbf{st}_0 , and a list \mathbf{pp} of public information is known to all participants. Each P_i with $i \in \mathbf{QUAL}$ also obtains his secret key \mathbf{sk}_i that is only known to him.
2. $(\mathbf{st} := \mathbf{st}_{r+1}, \Omega_r, \pi_r) \leftarrow \text{DRNGGen}(\mathbf{st} = \mathbf{st}_r, \mathbf{pp}) \langle \{P_i(\mathbf{sk}_i)\}_{i \in \mathbf{QUAL}} \rangle$: This is an interactive protocol between participants in a set \mathbf{QUAL} each holding the secret key \mathbf{sk}_i and common inputs \mathbf{pp} and the current state \mathbf{st}_r . At the end of the interaction, all honest participants a value Ω_r , and a proof π_r certifying the correctness of Ω_r made by the interaction. In

addition, the state \mathbf{st} is updated into a new state \mathbf{st}_{r+1} for the $(r + 1)$ -th epoch, denoted by $\mathbf{st} := \mathbf{st}_{r+1}$.

3. $b \leftarrow \text{DRNGVerify}(\mathbf{st} := \mathbf{st}_r, \Omega_r, \pi_r, \mathbf{pp})$: This algorithm can be publicly run by anyone. On input a the current state \mathbf{st}_r , a value Ω_r , a proof π_r , a public parameter \mathbf{pp} , this algorithm output a bit $b \in \{0, 1\}$ indicating the validity of Ω_r .

In the definition, we separate two parameters \mathbf{pp} and \mathbf{st} . Even though they are public parameters, \mathbf{pp} is the public setup information of all participants (often consists of their public keys), and thus is fixed in all epochs. On the other hand, \mathbf{st} consists of the current epoch the public seed that is used to calculate the output for the current epoch, and hence, it is updated and may differ every epoch, we see that it can be realized by assigning $\mathbf{st} := \mathbf{st}_r$ in the r -th epoch.

For better visualization, Figure 3.1 describe the relation between the DRNGSetup, DRNGGen protocol and DRNGVerify algorithm used in our DRNG definition.

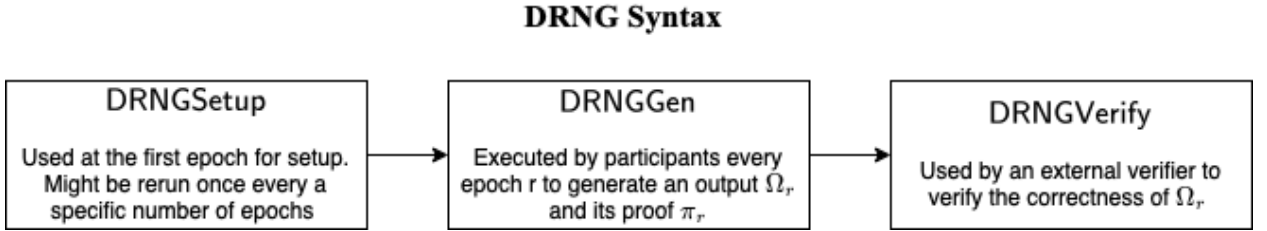


Figure 3.1: A visualization of a DRNG syntax

3.1.2 Security Properties

Before going to formally describe the security properties, we would like to make some notations: An adversary \mathcal{A} may corrupt participants in \mathcal{P} as the DRNG executes. Hence, when we write $(\mathbf{st}_{r+1}, \Omega_r, \pi_r) \leftarrow \text{DRNGGen}_{\mathcal{A}}(\mathbf{st}_r, \mathbf{pp}) \langle \{P_i(\mathbf{sk}_i)_{i \in \text{QUAL}}\} \rangle$ to denote that DRNGGen is executed when some participants are corrupted by \mathcal{A} . In this case the output $(\mathbf{st}_{r+1}, \Omega_r, \pi_r)$ will be affected by the actions of corrupted participants, for example, Ω_r can be equal to \perp if a dishonest participant causes the protocol to abort.

A secure DRNG protocol satisfies the following properties.

- **Pseudo-randomness.** We say that a (t, n) -DRNG satisfies Pseudo-randomness if for any epochs r any PPT adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ who corrupts up to t participants in \mathcal{P} ,

it holds that the advantage

$$\text{ADV}_{\text{rand-DRNG}}(\mathcal{A}) = \left| \Pr \left[\text{ExpRand}_{\text{DRNG}}^{\mathcal{A}}(1^\lambda, 0) = 1 \right] - \Pr \left[\text{ExpRand}_{\text{DRNG}}^{\mathcal{A}}(1^\lambda, 1) = 1 \right] \right|$$

is negligible in λ , where the experiment $\text{ExpRand}_{\text{DRNG}}^{\mathcal{A}}(1^\lambda, b)$ is defined as follows.

Setup:

$(\text{st} = \text{st}_0, \text{QUAL}, \text{pp}, \{\text{sk}_i\}_{i \in \text{QUAL}}) \leftarrow \text{DRNGSetup}(\lambda) \langle \{P\}_{P \in \mathcal{P}} \rangle$.

Queries:

$\text{tr} \leftarrow \mathcal{A}_1(\text{st}_r, \text{pp}, \mathcal{B}, \{\text{sk}_i\}_{i \in \mathcal{B}})^{\mathcal{O}_{\text{GetRandomness}}(\cdot)}$.

Challenge:

$(\text{st} := \text{st}_{r+1}, \Omega_r, \pi_r) \leftarrow \text{DRNGGen}_{\mathcal{A}}(\text{pp}, \text{st} := \text{st}_r) \langle \{P(\text{sk}_i)_{i \in \text{QUAL}}\} \rangle$.

If $b = 0$ then set $Y = \Omega_r$ else set $Y \xleftarrow{\$} \{0, 1\}^\lambda$.

Guess:

$b' \leftarrow \mathcal{A}_2(\text{tr}, \text{pp}, Y)$.

Return b' .

Figure 3.2: Experiment $\text{ExpRand}_{\text{DRNG}}^{\mathcal{A}}(1^\lambda, b)$.

The Oracle $\mathcal{O}_{\text{GetRandomness}}$ is described as follows.

Denote j to be the current epoch.

$(\text{st} := \text{st}_{j+1}, \Omega_j, \pi_j) \leftarrow \text{DRNGGen}_{\mathcal{A}}(\text{st} := \text{st}_j, \text{pp}) \langle \{P_i(\text{sk}_i)\}_{i \in \text{QUAL}} \rangle$.

Return $(\text{st}_j, \Omega_j, \pi_j)$.

Figure 3.3: Oracle $\mathcal{O}_{\text{GetRandomness}}(\cdot)$.

In the experiment, we let \mathcal{B} to be the set of participants corrupted by \mathcal{A} . Hence, the adversary can see all the secret keys of participants in \mathcal{B} . The experiment tells that \mathcal{A} and his corrupted “minions” are allowed to know the outputs $\Omega_1, \dots, \Omega_{r-1}$ of the first $r - 1$ epochs, but still cannot distinguish between the output of epoch r , which is Ω_r and a random value in $\{0, 1\}^\lambda$, which captures our informal definition.

A weaker version of **Pseudo-randomness** is **Unpredictability**. In this version, we only require the adversary to guess the output Ω_r with non-negligible probability, given $\Omega_1, \dots, \Omega_{r-1}$. In fact, many DRNG constructions are only proven to achieve Unpredictability instead of Pseudo-randomness due to employing cryptographic primitives like BLS signature or VDFs, which are not proven to produce pseudo-random outputs.

- **Unbiasability.** We say that the DRNG satisfies Unbaisability if any adversary \mathcal{A} who corrupts up to t participants in \mathcal{P} should not be able to affect future random beacon values for his own goal. More formally, for any epoch r and its current state \mathbf{st}_r , there exists a negligible function negl such that

$$\text{ADV}_{\text{bias-DRNG}}(\mathcal{A}) = \left| \Pr [\text{ExpBias}_{\text{DRNG}}^{\mathcal{A}}(1^\lambda)] - \frac{1}{2} \right| \leq \text{negl}(\lambda)$$

where the experiment $\text{ExpBias}_{\text{DRNG}}^{\mathcal{A}}(1^\lambda)$ is defined as follows:

```

(st := st0, QUAL, pp, {ski}i∈QUAL) ← DRNGSetup(1λ) ⟨{P}P∈P⟩
(st := st(r+1)0, Ωr0, πr0) ← DRNGGenℳ(st := str, pp) ⟨{Pi(ski)}i∈QUAL⟩ .
(st := st(r+1)1, Ωr1, πr1) ← DRNGGen(st := str, pp) ⟨{Pi(ski)}i∈QUAL⟩ .
b  $\xleftarrow{\$}$  {0, 1}.
b' ← ℳ(st, pp, Ωrb).
Return b  $\stackrel{?}{=}$  b'.

```

Figure 3.4: Experiment $\text{ExpBias}_{\text{DRNG}}^{\mathcal{A}}(1^\lambda)$

In the definition above, we let \mathcal{A} to corrupts t participant and affect the protocol to provide a possible output Ω_{r0} , while Ω_{r1} is a possible output when all participants are honest. However, from the view of \mathcal{A} , he cannot distinguish between Ω_{r0} and Ω_{r1} . Hence, from the view of \mathcal{A} , the affected output Ω_{r0} is not different than the real output Ω_{r1} , and thus, it does not give \mathcal{A} any advantages over Ω_{r1} .

- **Liveness (Availability).** We say that the DRNG satisfies the Liveness property if, for any epoch, and for any adversary \mathcal{A} who corrupts up to t participants in \mathcal{P} , the DRNG is guaranteed to produce an output. Formally, for any epoch r and its current state \mathbf{st}_r of the current epoch, we have:

$$\Pr \left[\Omega_r = \perp \mid \begin{array}{l} (\mathbf{st} := \mathbf{st}_0, \text{QUAL}, \text{pp}, \{\text{sk}_i\}_{i \in \text{QUAL}}) \leftarrow \text{DRNGSetup}(1^\lambda) \langle \{P\}_{P \in \mathcal{P}} \rangle \\ (\mathbf{st} := \mathbf{st}_{r+1}, \Omega_r, \pi_r) \leftarrow \text{DRNGGen}_{\mathcal{A}}(\mathbf{st} := \mathbf{st}_r, \text{pp}) \langle \{P_i(\text{sk}_i)\}_{i \in \text{QUAL}} \rangle \end{array} \right] = 0.$$

- **Public Verifiability.** We say that the DRNG satisfies Public Verifiability if for any epoch r , given the current global state \mathbf{st}_r , the public parameter pp and values $\Omega_r^*, \pi_r^* \in \{0, 1\}^*$, then an external verifier can run $\text{DRNGVerify}(\mathbf{st}_r, \text{pp}, \Omega_r^*, \pi_r^*)$ to determine if Ω_r^* is correctly

calculated from the DRNGGen protocol.

Remark 3.1.2. Our definition is the traditional way to define the syntax and security for a scheme or protocol. Another way to capture the execution and required security properties in a protocol is the **Universal Composable Security** framework of Ran Canetti [Can01]. This framework is better for capturing the execution of protocols in a realistic environment. However, the framework itself is rather complicated to understand and design, so we were not able to formulate DRNG execution and security using the framework. Hence we decided to define the syntax and security in the traditional way.

3.2 DRNG Classifications

The problem of decentralized random number generation began when Blum [Blu83] introduced a protocol that can generate random bits by a telephone line. However, the protocol does not provide Unbiasability. Cleve [Cle86] showed that for any epoch, there is an adversary that can affect the outputs of the honest participants. Since then, the problem of decentralized number generation has received attention from the community of cryptography.

In this section, we conduct a systematic literature review of existing DRNG protocols by sorting them through their cryptographic primitives, then we will analyze and compare their strengths and weaknesses based on several aspects in Sections 3.3 and 3.4. Based on the result of the comparison, we will select the most suitable DRNG construction for blockchain-based applications and specify the reason for our choice in Section 4.1. Because the review is going to be very long and involves a lot of DRNG constructions, we refer to Figure 3.5 to give an overview of our classifications of DRNG constructions and the lists of constructions in each category.

In addition, as we will present later in each category, many protocols use a specific primitive or idea to fix the mistake or optimize previous protocols. They may also follow the framework of the previous protocol. It will be hard to read all these connections in words. Hence we also give an overview of the connections between these protocols through Figure 3.6.

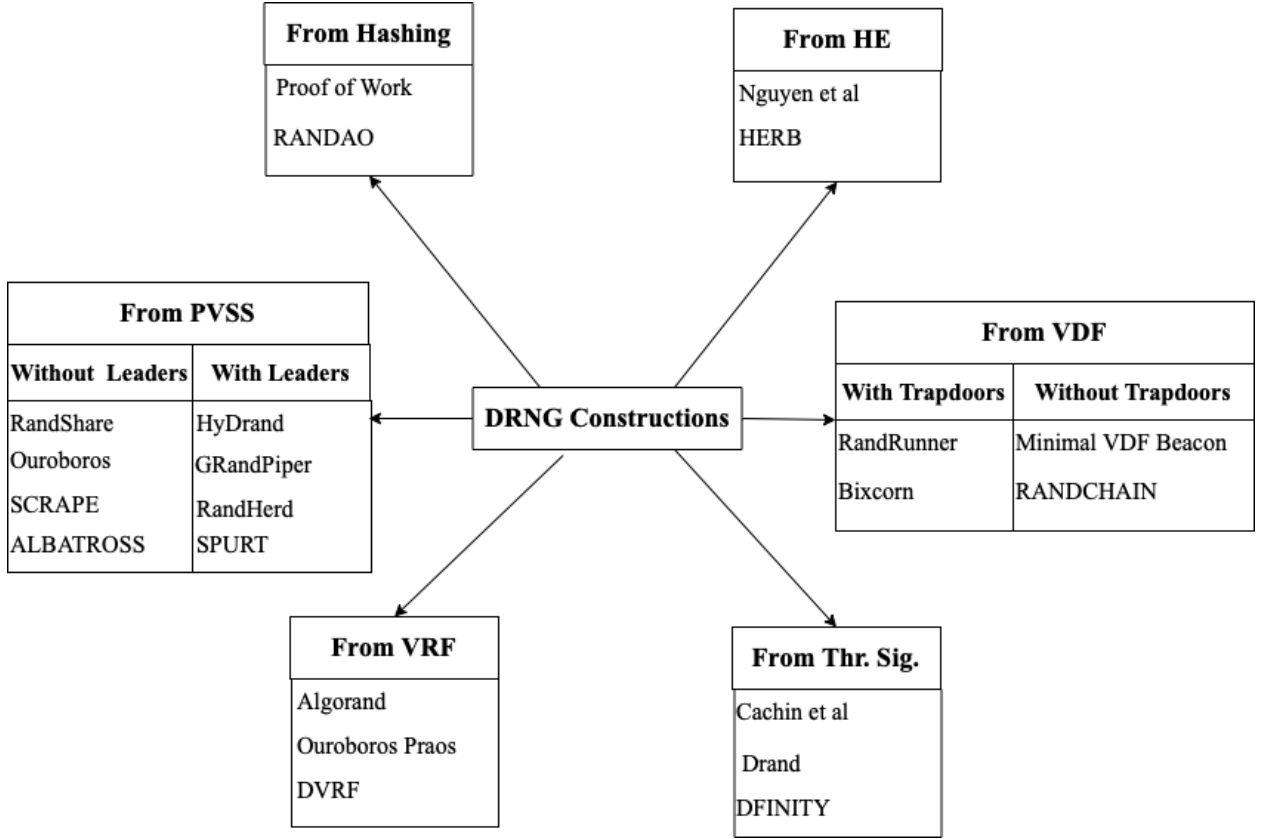


Figure 3.5: *An overview of existing DRNG constructions*

Before going to the details of each protocol, we make some important remarks used in the efficiency analysis in each protocol.

- For our analysis of communication complexity, we only consider the cases where all the DRNG constructions use a network where each pair of nodes is connected with a secure **point-to-point channel** and without any **broadcast channel**. In such a network, participants can reach consensus on common messages via some mechanism such as **Byzantine agreement protocols** [LSP82]. We believe that the assumption is realistic for practical situations: true broadcast is only achievable in practice using Byzantine agreement but secure channels can be achieved through a public key infrastructure.
- For computation and verification complexity analysis, some protocols employ Lagrange interpolation and require computing $\mathcal{O}(n)$ Lagrange coefficients. The **optimal cost** of computing these coefficients will be set to be $\mathcal{O}(n \log^2 n)$. The reason for this optimal cost will be specified in Subsection 4.5.3.

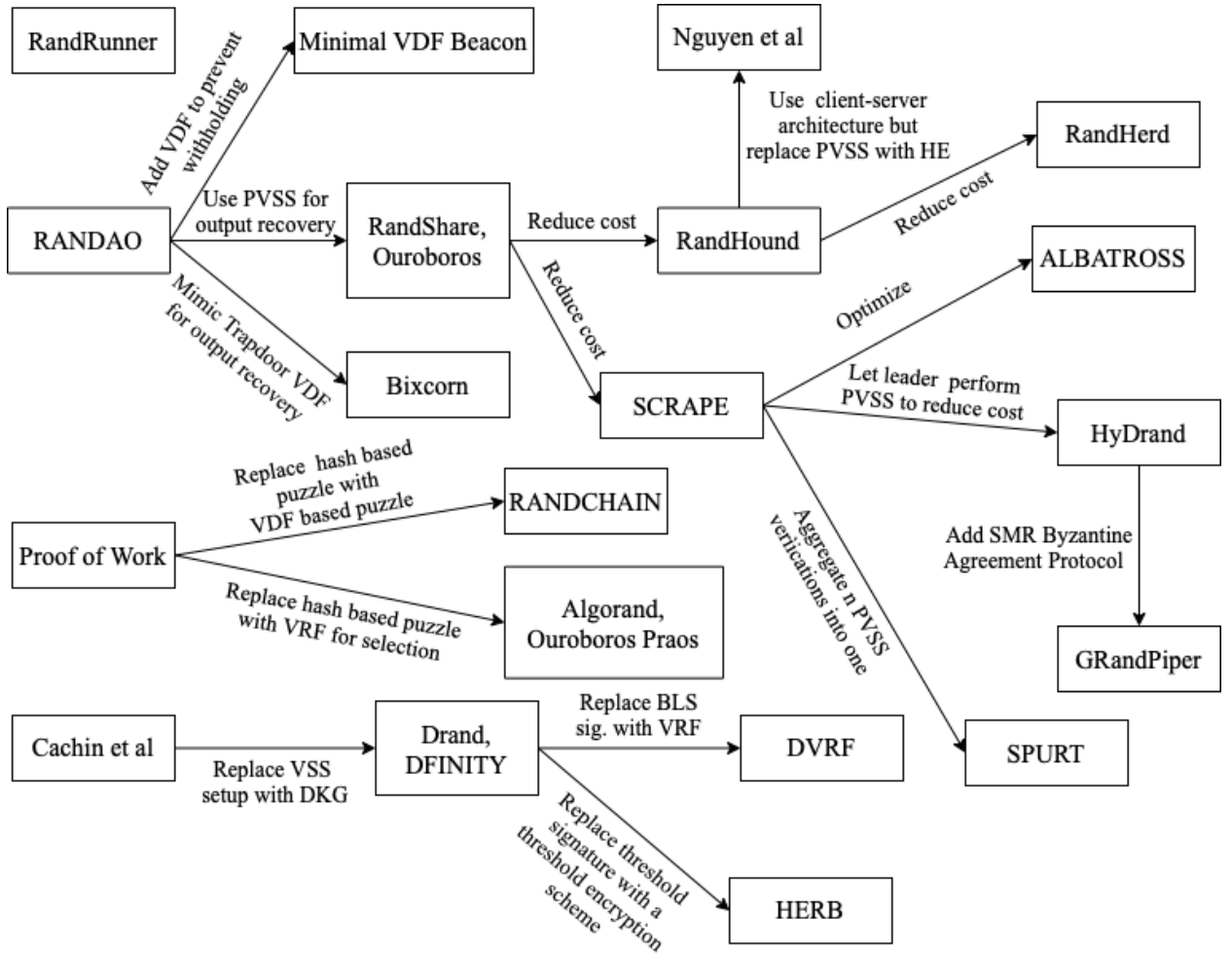


Figure 3.6: *The relation between DRNGs*

Finally, we would like to note that since we are conducting a systematic literature survey, we only describe the DRNG constructions informally to give a high-level overview of them. Hence the formal syntax in Section 3.1.1 will not be used to describe these constructions.

3.2.1 DRNG from Hashing

Some DRNGs, namely [Nak, Ran17] follow this direction and rely on a cryptographic hash function in Section 2.4.5 to produce unpredictable outputs. These protocols have a simple construction and enjoy low cost. However, the common drawback of these protocols is that they suffer from a withholding attack, where a participant may refuse to submit his partial output if the resulting protocol output does not benefit him.

1. Proof-of-Work

Proof-of-work [Nak] attempts to produce randomness for blockchain from a set of n participants. This protocol follows the “priority selection” style, i.e., from a set of n participants, determine the one with the highest priority to produce a Block B and add it to the current chain. Recall that in Subsection 2.5.1, this selection process is done by forcing the participants (or miners in the blockchain) to solve a certain puzzle. The first participant to successfully solve the puzzle earns the privilege to propose a Block B . The resulting randomness is computed from the hash of the Block, i.e. $B.h$, which is inherently unpredictable. With the discussions above, the concept of generating randomness using proof-of-work can be summarized as follows.

Overview of Proof-of-Work

1. Recall that each Block B is identified by its hash value $B.h$. Each participant competes to be the highest “priority” participant by finding a solution to a certain puzzle that involves a hash function. An example of a puzzle is as follows.

Given a value **target**, find a Block B such that $H(B.h) < \text{target}$.

The difficulty of the puzzle, i.e., the parameter **target** can be adjusted to ensure that at least one participant can find a solution within a certain time.

2. The first participant P_i to solve a puzzle is the one with the highest “priority” and gets the right to propose a Block B_i and adds it to the current chain. The output of the protocol Ω is calculated as the hash of the Block B_i , i.e.

$$\Omega = B_i.h.$$

Analysis. Proof-of-work exhibits $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$ communication complexity in the best and worst case and $\mathcal{O}(1)$ verification complexity only, this will be explained in Subsection 3.3.2. This establishes proof-of-work as highly cost-effective, given its minimal communication and verification overhead. The computation complexity of each participant is very high though, depending on the difficulty of the puzzle. Proof-of-work, although simple, has two drawbacks. First, it assumes that the probability of an adversary proposing the next Block **before** an

honest participant is less than $1/2$ to achieve Unpredictability. However, an adversary with enhanced computational capabilities may solve the puzzle and successfully mine a Block much earlier than honest participants. In this case, he has already known the output Ω while other participants are struggling to solve the puzzle. In addition, it is susceptible to withholding attacks. A participant, when proposing a valid Block, may discard the Block and continue mining another Block until he finds the one that gives the output Ω that benefits him.

2. RANDAO

RANDAO [Ran17] employed a Random number generator based on this approach on Ethereum blockchain. Unlike the “priority selection” style of proof-of-work, where each participant competes to be the one with the highest priority, the output of RANDAO is produced in a **collaborative** style as follows. Each participant contributes his local inputs and these inputs are aggregated into a single output. With this style, the randomness generation of RANDAO consists of the three main steps as follows.

Overview of RANDAO

1. Each participant P_i who wants to take part in the protocol chooses a secret value s_i and publishes his commitment $C_i = H(s_i)$.
2. When all parties committed their values $H(s_i)$, each participant P_i reveals his secret s_i . Other participants can verify the correctness of s_i by checking $C_i \stackrel{?}{=} H(s_i)$.
3. If all checks are correct, the final random number is calculated by computing the exclusive-or (XOR, \oplus) of the collected secret values as follows.

$$\Omega = \bigoplus_{i=1}^n s_i.$$

Analysis. In the construction above, one may ask why the protocol requires participants to publish their commitment. The reason is simple. Suppose a participant P_n wishes the output of the protocol to be s . If no commitments are required, he can simply wait until $n - 1$ other participants P_1, \dots, P_{n-1} submit their shares s_1, \dots, s_{n-1} and alter $s_n = s \oplus \bigoplus_{i=1}^{n-1} s_i$, thus the

last participant can completely control the output. When commitments are required, the value s_n are “fixed” before the reveal, since s_n must satisfy $H(s_n) = C_n$. Hence, P_n can no longer alter s_n to control the output.

Now that an adversary cannot control the output, he can only hope to learn the output before everything is revealed. However, after the first step, only the commitments are revealed, and due to the one-way property of H , the adversary cannot learn the secret s_1 of an honest participant P_1 . Hence, he cannot predict the output of s of the protocol unless s_1 is revealed.

Now, one may argue that the protocol is secure as long as at least one s_i is uniformly chosen. However, there is one issue with the approach. The final participant P_n , upon learning the values s_1, \dots, s_{n-1} revealed by previous participants, can still choose not to reveal his secret and force the protocol to abort. He can force the protocol to restart until he obtains an output of his interest. This makes RANDAO not achieve Unbiasability and Liveness.

Finally, we analyze the complexity of RANDAO. The communication complexity of RANDAO is $\mathcal{O}(n^2)$ since each participant has to send the commitment and secrets to $n - 1$ other participants and reach consensus via a Byzantine agreement protocol. The computation complexity is $\mathcal{O}(1)$ and the verification complexity of an external verifier is $\mathcal{O}(n)$, since an external verifier has to verify n secrets given their commitments.

3.2.2 DRNG from Publicly Verifiable Secret Sharing

An approach to construct a DRNG that provides unpredictable and unbiased output is to use a PVSS scheme $PVSS = (PVSSShare, PVSSShareVerify, PVSSShareReveal, PVSSReconstruct)$ described in Subsubsection 2.4.7.3. Many protocols, namely [KRDO17, CD17, SJK⁺17, SJSW20, DKIR22, CD20] construct a DRNG based on PVSS. PVSS-based protocols can be further partitioned into two types: **constructions with leader** [SJSW20, SJK⁺17, DKIR22] and **constructions without leader** [CD17, SJK⁺17, CD20, KRDO17].

The idea of constructing a PVSS-based scheme actually comes from the mistake of RANDAO. In RANDAO, participants have to commit their secret with a hash function and then reveal it. However, the last participant can refuse to reveal his secret. Non-leader PVSS-based protocol forces each participant P_i to execute the $PVSSShare$ protocol of PVSS scheme to commit his secret S_i by broadcasting the encrypted shares instead of using a hash function. The protocol $PVSSShare$ allow P_i to provide sufficient information that other participant can check the correctness of S_i via the $PVSSShareVerify$ algorithm, while any t

dishonest participants cannot learn S_i . In addition, after verifying the shares, if the participant does not reveal his secret, then honest participants can use `PVSSShareReveal` to reveal their share and use `PVSSReconstruct` to construct back S_i . Hence, the withholding attack of RANDAO made by the last participant does not work against these protocols because the secret S_i can always be reconstructed when there are $t + 1$ honest participants.

The main advantage of PVSS-based DRNG is that most of these protocols achieve full security properties as long as there are at most t dishonest participants. The problem with these protocols is that all non leader-based protocols suffer from $\mathcal{O}(n^3)$ communication complexity, the reason for this will be explained in Subsection 3.3.2. Leader-based protocols, on the other hand, enjoy $\mathcal{O}(n^2)$ communication complexity. However, leader-based protocols are vulnerable to adversaries, who can immediately corrupt the leader as soon as he is elected.

Note that in some protocols we will have to work with the inner structure of the PVSS again. Hence in these protocols, we consider \mathbb{G} to be the group of order p and denote h to be one of its generators, as we defined in the PVSS scheme in Subsubsection 2.4.7.3.

3.2.2.1 Constructions without Leaders

As stated earlier, these DRNG constructions require each participant P_i to generate his secret S_i to execute the PVSS scheme simultaneously to distribute S_i . The protocol that started this approach is RandShare, which achieves full security properties but incurs high complexity. Many future DRNGs based on PVSS are inspired by RandShare and attempt to optimize it to reduce the complexity, for example, SCRAPE, ALBATROSS, and many leader-based PVSS protocols. Below we briefly describe these protocols.

1. RandShare and Ouroboros

RandShare [SJK⁺17] and Ouroboros [KRDO17] are the simplest non leader-based PVSS protocols that serve as the motivation for many future PVSS-based protocols. While RandShare is an independent protocol dedicated to producing randomness, Ouroboros uses the process as a sub-protocol in its blockchain design and mainly focuses on the low-level blockchain protocol. In both protocols, each participant P_i chooses a secret S_i and performs the PVSS scheme in Subsubsection 2.4.7.3 to commit S_i to distribute the shares of S_i to other participants. After validating all shares, the reconstruction of S_i will be done by all other participants. Finally, the output Ω will be calculated by combining all the valid S_i s. More

specifically, the idea above can be described using a five-step process as described below.

Overview of RandShare and Ouroboros

1. Each participant P_i randomly chooses a secret $s_i \in \mathbb{Z}_p$, then set $S_i = h^{s_i}$, then distributes his secret S_i by executing

$$(C_i, \{(E_{ij}, \pi_{ij})\}_{j=1}^n) \leftarrow \text{PVSSShare} \langle P_i(S_i), \{P_j(\text{pk}_j, \text{sk}_j)\}_{j=1}^n \rangle$$

and then broadcasts the commitments, encrypted shares and proofs $(C_i, \{(E_{ij}, \pi_{ij})\}_{j=1}^n)$ to all other participants.

2. For each participant P_i , each participant P_j checks the correctness of E_{ij} by running the algorithm $\text{PVSSShareVerify}(E_{ij}, \text{pk}_j, \pi_{ij}, C_i)$. If verification fails P_j complains against P_i . Each P_i receives more than $t + 1$ complaints are marked invalid.

3. For each **valid** participant P_i , each participant P_j executes

$$(S_{ij}, \pi'_{ij}) \leftarrow \text{PVSSShareReveal}(E_{ij}, \text{sk}_j, \text{pk}_j)$$

to reveal his share S_{ij} from P_i and its proof of correct decryption π'_{ij} .

4. For each revealed share S_{ij} , participant verify the correctness of S_{ij} using π'_{ij} . For a fixed i , when at least $t + 1$ shares $\{S_{ij}\}_{j \in \mathcal{V}}$ are revealed for some $\mathcal{V} \subseteq \{1, \dots, n\}$, valid participants execute the algorithm

$$S_i \leftarrow \text{PVSSReconstruct}(\mathcal{V}, \{(S_{ij}, E_{ij}, \text{pk}_j, \pi'_{ij})\}_{j \in \mathcal{V}})$$

to reconstruct the secret S_i of P_i . Aborts if there are $t + 1$ secrets S_i equal to \perp .

5. If at least $n - t$ secrets $\{S_i\}_{i=1}^{n-t}$ of valid participants are successfully reconstructed, the final output Ω of the protocol is computed by multiplying all the S_i as follows.

$$\Omega = \prod_{i=1}^{n-t} h^{s_i} = \prod_{i=1}^{n-t} S_i.$$

Analysis. As we have discussed earlier, unlike RANDAO, after all participants have executed PVSSShare, then **at least** $n - t$ secrets S_i are distributed by honest participants, and these secrets can always be reconstructed by $t + 1$ honest participants using PVSSShareReveal and PVSSReconstruct. Hence, these protocols achieve Liveness and Unbiasability. The protocol also achieves Pseudo-randomness since the output Ω is uniformly distributed each epoch, assuming at least one honest participant P_i who generates S_i uniformly. However, they incur [SJK⁺17] communication complexity and computation complexity $\mathcal{O}(n^3)$ due to letting all participants use the PVSS construction of Schoenmakers [Sta96]. Let us analyze the reason: In Step 2 of RandShare, each participant has to check the correctness of the encrypted shares using PVSSShareVerify. Recall that in Schoenmakers' construction (see **Protocol 7**), to execute PVSSShareVerify, each participant requires $\mathcal{O}(n)$ computation to compute each commitment

$$A_{ij} = g^{s_{ij}} = \prod_{k=1}^t g^{a_{ij}j^k} = \prod_{k=1}^t C_{ij}^{j^k}.$$

Note that since there are n^2 such commitments c_{ij} , the total computation complexity is $\mathcal{O}(n^3)$ for a participant, and it is also the dominant cost in a PVSS of Schoenmaker. This makes RandShare and Ouroboros very inefficient when used for many participants. Nevertheless, many future PVSS-based constructions are based on them and employ other techniques to reduce communication and computation complexity.

2. SCRAPE

SCRAPE [CD17] is also a non-leader based PVSS scheme that follows exactly the five-step process of RandShare and Ouroboros. However, SCRAPE attempts to replace Schoenmakers' PVSS construction with a new PVSS scheme using coding theory [CD17], which aims to cut down the computation complexity of RandShare and Ouroboros from $\mathcal{O}(n^3)$ down to $\mathcal{O}(n^2 \log^2 n)$. This can be done by just modifying the PVSSShareVerify algorithm of the PVSS of Schoenmakers in Step 2 so that verifying all shares of a dealer can be done within $\mathcal{O}(n)$ computation, making the computation cost of PVSS is now dominated by computing Lagrange coefficients, which is $\mathcal{O}(n \log^2 n)$. For more details, the main difference between the PVSS construction of SCRAPE and RandShare can be seen below.

SCRAPE's Optimization from Randshare

1. In Step 1 of RandShare, at the beginning of the PVSSShare protocol, each participant P_i chooses $f_i(x) = a_{i0} + a_{i1}x + \dots + a_{it}x^t$ and commits the shares

$$C_i = \{C_{ij}\}_{i=1}^n = \{g^{s_{ij}}\}_{i=1}^n$$

instead of the coefficients $C_i = \{C_{ij}\}_{i=1}^t = \{g^{a_{ij}}\}_{i=1}^t$ as in RandShare.

2. In Step 2 of RandShare, for any fixed i the algorithm PVSSShareVerify now receives inputs $\{(E_{ij}, \pi_{ij})\}_{j=1}^n$, pk_j and C_i . Given the commitment $C_{ij} = g^{s_{ij}}$ of the share s_{ij} , it first check the correctness of the encryption $E_{ij} = \text{pk}_j^{s_{ij}}$ by running the algorithm

$$\text{NIZKDLGVerify}(g, C_{ij}, \text{pk}_j, E_{ij}, \pi_{ij})$$

in Subsubsection 2.4.6.2 for each j . Finally, it needs to check whether $\{s_{ij}\}_{j=1}^n$ are the shares are coming from a polynomial of degree t . This check can be done within $\mathcal{O}(n)$ computation steps as follows:

- (a) Sample an element $\mathbf{v} \in \mathcal{D}$, where \mathcal{D} is defined below.

$$\mathcal{D} = \left\{ \mathbf{v} \in \mathbb{F}^n \mid \sum_{i=1}^n \mathbf{v}_i f(i) = 0 \ \forall f \in \mathbb{F}_{\leq t}[X] \right\}.$$

In coding theory, it can be proved that \mathcal{D} can be efficiently sampled and if $f \notin \mathbb{F}_{\leq t}[X]$ then $\mathbf{v}_i f(i) \neq 0$ with overwhelming probability, where $\mathbb{F}_{\leq t}[X]$ denote the set of all polynomials of degree no more than t in a field \mathbb{F} .

- (b) With the vector \mathbf{v} above, check if the following equation holds:

$$\prod_{j=1}^n C_{ij}^{\mathbf{v}_j} \stackrel{?}{=} 1.$$

The equation is equivalent to $\sum_{j=1}^n \mathbf{v}_j s_{ij} = 0$, indicating that $s_{ij} = f_i(j) \ \forall j \in \{1, \dots, n\}$ for some $f \in \mathbb{F}_{\leq t}[X]$ with overwhelming probability.

Analysis. With this new PVSS scheme, it requires only $\mathcal{O}(n)$ verification complexity to verify all shares of a dealer, since a verifier only has to compute $\prod_{j=1}^n C_{ij}^{\mathbf{v}_j}$. Hence, the cost of this PVSS scheme is dominated by the cost of **optimally** (not naively) computing $\mathcal{O}(n)$ Lagrange coefficients in each epoch, which can be optimally reduced to $\mathcal{O}(n \log^2 n)$, this will be discussed in Subsection 4.5.3. This helps SCRAPE achieve $\mathcal{O}(n^2 \log^2 n)$ computation complexity, dominated by the cost of computing Lagrange coefficients. SCRAPE also achieve full security properties like RandShare and Ouroboros due to having the same structure, but having better computation complexity, as mentioned above. However, the computation complexity of SCRAPE is still quadratic in terms of participants and the communication complexity is still $\mathcal{O}(n^3)$ due to having the same number of messages as RandShare.

3. ALBATROSS

Extending SCRAPE, ALBATROSS [CD20] provides an improved amortized communication complexity of $\mathcal{O}(\ell)$ per beacon output by generating a batch of $\mathcal{O}(\ell^2)$ beacon outputs per epoch (as opposed to one), where $\ell = n - 2t$. To generate $\mathcal{O}(\ell^2)$ beacon outputs, ALBATROSS introduces a new PVSS scheme that employs **Packed Shamir secret sharing**. At a high level, Packed Shamir secret sharing generalizes a PVSS scheme by allowing the dealer D to share a secret \mathbf{s} which is a vector in \mathbb{G}^ℓ . In other words, the secret \mathbf{s} has the form

$$\mathbf{s} = (h^{s_0}, h^{s_1}, \dots, h^{s_{\ell-1}}),$$

where $s_0, s_1, \dots, s_{\ell-1} \xleftarrow{\$} \mathbb{Z}_p$. With this new PVSS scheme, in the first step participants just execute PVSSShare, PVSSShareVerify, and PVSSShareReveal to distribute and recover the secrets S_i as in RandShare or SCRAPE. Now, after $n - t$ valid shares \mathbf{s}_i are recovered, say $\mathbf{s}_1, \mathbf{s}_2, \dots, \mathbf{s}_{n-t}$, we denote the matrix $\mathbf{T} \in \mathbb{G}^{(n-t) \times \ell}$ to be the matrix satisfying $\mathbf{T}_{ij} = \mathbf{s}_{ji}$ for all $j \leq n - t$ and $i \leq \ell$, where \mathbf{s}_{ji} is the i -th component of \mathbf{s}_j . The matrix \mathbf{T} consists of $\ell \cdot (n - t)$ group elements, where each row is the recovered secret \mathbf{s}_i of P_i . Now, one might naturally choose all $\ell \cdot (n - t)$ elements of \mathbf{T} to be the output of the protocol. However, recall that t rows in \mathbf{T} are controlled by dishonest participants, hence the cells in \mathbf{T} are not uniformly distributed because of these rows. To produce ℓ^2 uniformly distributed output, the idea is to “multiply” \mathbf{T} by a matrix \mathbf{V} . In ALBATROSS, the authors consider a matrix $\mathbf{V} \in \mathbb{Z}_p^{\ell \times (n-t)}$ satisfying $\mathbf{V}_{ij} = \omega^{(i-1)(j-1)} \forall 1 \leq i, j \leq \ell$ for some $\omega \in \mathbb{Z}_p$. Next, the authors

consider the matrix $\mathbf{M} \in \mathbb{G}^{\ell \times \ell}$, which is formed by “multiplying” \mathbf{T} with \mathbf{V} as follows.

$$\mathbf{M}_{ij} = \prod_{k=1}^{n-t} \mathbf{T}_{kj}^{\mathbf{V}_{ik}} \quad \forall 1 \leq i, j \leq \ell.$$

In [CD20, Section 2], the authors proved that with the choice of \mathbf{V} above, each cell in \mathbf{M} are uniformly distributed, even when t rows in \mathbf{T} are controlled by dishonest adversaries (in fact, this is the reason the authors the parameter ℓ is set to be $n - 2t$). With this property, the output Ω will be the ℓ^2 elements in \mathbf{M} . Finally, if ω is chosen to be a n -th root of unity, then the matrix \mathbf{M} can be efficiently computed in $O(n^2 \log n)$ computation complexity. To see this, for each $1 \leq i \leq \ell$ it holds that the column $\mathbf{T}^{(i)}$ of the matrix \mathbf{T} has the form below.

$$\mathbf{T}^{(i)} = (h^{s_{1(i-1)}}, h^{s_{2(i-1)}}, \dots, h^{s_{(n-t)(i-1)}}). \quad (3.1)$$

Now, if we consider the polynomial

$$f_i(x) = s_{1(i-1)} + x \cdot s_{2(i-1)} + \dots + x^{n-t} \cdot s_{(n-t)(i-1)} \quad (3.2)$$

then we can see that, for each i the column $\mathbf{M}^{(i)}$ of \mathbf{M} has the form below

$$\mathbf{M}^{(i)} = (h^{f_i(1)}, h^{f_i(\omega^1)}, \dots, h^{f_i(\omega^\ell)}). \quad (3.3)$$

From (3.1), (3.2) and (3.3), one see that calculating $\mathbf{M}^{(i)}$ is just equivalent to performing Fast Fourier Transform with input $\mathbf{T}^{(i)}$. Recall that FFT has $\mathcal{O}(n \log n)$ computation complexity, hence we can generate $\mathcal{O}(\ell^2)$ randomness output with $\mathcal{O}(n^2 \log n)$ computation complexity in this way. With the discussions above, ALBATROSS can be summarized as follows.

Overview of ALBATROSS

1. Let ω be a n -th root of unity. Participants agree on the matrix $\mathbf{V} \in \mathbb{Z}_p^{\ell \times (n-t)}$ satisfying

$$\mathbf{V}_{ij} = \omega^{(i-1)(j-1)} \quad \forall 1 \leq i, j \leq \ell.$$

2. Each participant P_i chooses $s_{i0}, s_{i1}, \dots, s_{i(\ell-1)} \xleftarrow{\$} \mathbb{Z}_p$ and then sets his secret \mathbf{s}_i to be

$$\mathbf{s}_i := (h^{s_{i0}}, h^{s_{i1}}, \dots, h^{s_{i(\ell-1)}})$$

then execute Steps 1, 2, 3 of RandShare (or SCRAPE) above to commit, distribute and reconstruct the secret \mathbf{s}_i using PVSS for each $i \in \{1, \dots, n\}$.

3. After Step 3 of RandShare (or SCRAPE), at least $n - t$ valid secrets \mathbf{s}_i are reconstructed, say $\mathbf{s}_1, \dots, \mathbf{s}_{n-t}$. Let $\mathbf{T} \in \mathbb{G}^{(n-t) \times \ell}$ to be the matrix satisfying $\mathbf{T}_{ij} = \mathbf{s}_{ji}$ for all $j \leq n - t$ and $i \leq \ell$, where \mathbf{s}_{ji} is the i -th component of \mathbf{s}_j . As discussed above, it can be seen that for each $1 \leq i \leq \ell$, the i -th column $\mathbf{T}^{(i)}$ of \mathbf{T} can be calculated as

$$\mathbf{T}^{(i)} = (h^{s_{1(i-1)}}, h^{s_{2(i-1)}}, \dots, h^{s_{(n-t)(i-1)}}).$$

Finally, let $\mathbf{M} \in \mathbb{G}^{\ell \times \ell}$ to be a matrix formed by “multiplying” \mathbf{T} with \mathbf{V} as follows.

$$\mathbf{M}_{ij} = \prod_{k=1}^{n-t} \mathbf{T}_{kj}^{\mathbf{V}_{ik}} \quad \forall 1 \leq i, j \leq \ell.$$

As discussed above, for each i , the column $\mathbf{M}^{(i)}$ of \mathbf{M} is calculated via FFT as follows.

$$\mathbf{M}^{(i)} \leftarrow \text{FFT}(\mathbf{T}^{(i)}, \omega) \quad \forall i \in \{1, 2, \dots, \ell\}.$$

Once \mathbf{M} is calculated, the output Ω of ALBATROSS is set to be ℓ^2 elements of \mathbf{M} .

Analysis. As discussed above, all ℓ^2 elements produced by ALBATROSS are uniformly distributed as long as at most t participants are dishonest. ALBATROSS also achieves Unbiasability, Liveness and public verifiability like RandShare and SCRAPE since their structures are the same. However, the only drawback of ALBATROSS is that it requires

$\mathcal{O}(n^2 \log n)$ computation complexity to produce ℓ^2 random outputs. On the bright side, this means that we just have to run ALBATROSS once every ℓ^2 epoch. However, since ℓ^2 is not very big, this means we still have to execute ALBATROSS quite often, and each execution requires $\mathcal{O}(n^2 \log n)$ computation complexity, which is very inefficient. Finally, the communication complexity of ALBATROSS is $\mathcal{O}(n^3)$ like RandShare and SCRAPE.

3.2.2.2 Construction with Leaders

As indicated by the title, in every epoch, these constructions include a leader who executes the whole PVSS scheme, and participants act as the verifiers of the shares distributed as the dealer and reconstruct the shares. This type of DRNG aims to cut down the cost of non leader-based PVSS constructions, both in communication and computation complexity at least by a factor of n . However, as a security tradeoff, they are susceptible to adaptive adversaries, who can corrupt the leader after he is selected to learn or bias the output. Below we will describe these protocols.

1. HyDrand and GRandPiper

HyDrand [SJSW20] and GRandPiper [BSL⁺21] are leader-based protocols that improve upon the complexity of SCRAPE and RandShare’s PVSS protocol. In each epoch, a leader is selected deterministically from the set of candidates by using the output of the previous epoch. HyDrand and GRandPiper follow the very concept of a leader-based DRNG, where the leader L performs the PVSS scheme to share his secret S with other participants each epoch. The secret S will be reconstructed by participants in the next epoch L will be the leader again. It is assumed that each participant P is expected to become a leader once every n epochs. The design of these protocols is rather complicated, involving processes such as message authentication and the development of novel mechanisms for achieving consensus on messages. However, for simplicity, we will focus solely on the step that involves the computation of Ω . The random generation process in these protocols can be succinctly summarized in a three-step procedure as follows.

Overview of HyDrand and GRandPiper

1. Let r be the current epoch. The leader P_ℓ chooses a secret seed $s \in \mathbb{Z}_p$, with his secret to be shared is $S = h^s$. He then computes

$$(C, \{(E_i, \pi_i)\}_{i=1}^n) \leftarrow \text{PVSSShare}\langle P_\ell(S), \{P_i(\text{pk}_i, \text{sk}_i)\}_{i=1}^n \rangle$$

and broadcast the information $D = (C, S^*, \{(E_i, \pi_i)\}_{i=1}^n)$ to all participants, where $S^* = h^{s^*}$ is the previous secret of the last epoch where he was elected as the leader.

2. Participants after receiving D check the correctness of S^* using the previous public information D^* , which contains the commitment of S^* . He also checks the encrypted shares with SCRAPE's PVSS verify function $\text{PVSSShareVerify}(\text{pk}_i, C, \{(E_j, \pi_j)\}_{j=1}^n)$.
3. If case the leader P_ℓ does not reveal his last secret S^* in Step 1, or broadcasts the wrong secret, each participant P_i use the previous public information $D^* = (C^*, \{(E_i^*, \pi_i^*)\}_{i=1}^n)$ sent by P_ℓ in his last epoch to reveal his previous share S_i^* by computing

$$(S_i^*, \pi_i'^*) \leftarrow \text{PVSSShareReveal}(E_i^*, \text{sk}_i, \text{pk}_i).$$

The secret S^* is then reconstructed by other participants as follows.

$$S^* \leftarrow \text{PVSSReconstruct}(\mathcal{V}, \{(S_i^*, E_i^*, \text{pk}_i, \pi_i'^*)\}_{i \in \mathcal{V}}).$$

The output of the current epoch Ω_r is computed from S^* and the previous epoch. For example, HyDrand's output Ω_r is computed using a hash function H as follows.

$$\Omega_r = \text{H}(\Omega_{r-1} || S^*).$$

Finally, from the last t outputs $\Omega_r, \dots, \Omega_{r-t+1}$, the leader P_ℓ for the $(r+1)$ -th epoch will be deterministically determined. For simplicity, we view this process as a black box.

Analysis. HyDrand and GRandPiper both employ SCRAPE's PVSS to optimize computation and verification complexity of each participant down to $\mathcal{O}(n \log^2 n)$, dominated by the cost of optimally computing Lagrange coefficients. In addition, since only a single

PVSS scheme is performed by the leader, both HyDrand and GRandPiper incur $\mathcal{O}(n^2)$ communication complexity and $\mathcal{O}(n \log^2 n)$ computation and verification complexity. The primary difference between these protocols does not lie in the generation of Ω , but rather in the mechanisms employed to achieve consensus on messages, as GRandPiper proposes its own Byzantine agreement protocol with the aim of reducing communication complexity and achieving optimal resilience ($t = n/2$).

Now we describe the disadvantage of these protocols. Given their leader-based nature, both do not provide Unbiasability against an adaptive adversary. This adversary could corrupt up to t participants at the first t epochs during the protocol to learn or withhold the outputs. Hence, these protocols only provide security from the $(t + 1)$ -th epoch forward.

2. RandHound

RandHound [SJK⁺17] is a PVSS-based DRNGs that aims to reduce the communication complexity of RandShare by following a client-server architecture. At a high level, RandHound includes a requester to divide the set of participants into smaller groups $T_1, T_2, \dots, T_{n/c}$, each having size c , where each group executes the RandShare protocol separately and sends their result to the client, who verify and combines the result to get the final output by himself.

Given RandShare as a sub-protocol, the idea of RandHound can be presented in more detail as follows: Each group T_i executes steps 1 and 2 RandShare protocol to commit and distribute the secrets among the group and sends the public transcript tr_{1,T_i} to the client, which consists of the commitments $\{C_j\}_{j \in T_i}$ and encrypted shares and proofs $\{(E_{jk}, \pi_{jk})\}_{j,k \in T_i}$ generated by PVSSShare. Given tr_{1,T_i} , the client can verify the correctness of all shares in each group T_i by himself using PVSSShareVerify. Upon verification, the client requests each group to decrypt their shares. To do this, each group executes Step 3 of RandShare and sends the public transcript tr_{2,T_i} to the client, which consists of the decrypted shares and proofs $\text{tr}_{2,T_i} = ((S_{jk}, \pi'_{jk})_{j,k \in T_i})$. Given tr_{2,T_i} , the client can reconstruct Ω_{T_i} by executing steps 4 and 5 of RandShare by himself and combine them to get the final result Ω . With the discussion above, the idea of RandHound can be summarized below.

Overview of RandHound

1. Each group T_i executes Steps 1 and 2 RandShare protocol described in Subsubsection **3.2.2.1** and sends the public transcript $\text{tr}_{1,T_i} = (\{C_j\}_{j \in T_i}, \{(E_{jk} \cdot \pi_{jk})\}_{j,k \in T_i})$ to the client.
2. The client, upon receiving tr_{1,T_i} for each i , verifies the correctness of E_{jk} for each $j, k \in T_i$ by using $\text{PVSSShareVerify}(E_{jk}, \text{pk}_j, \pi_{jk})$. After that, the client then sends a decryption request to each group T_i .
3. Each group T_i executes Step 3 of RandShare protocol to decrypt the shares S_{jk} for each $j, k \in T_i$ along with its proof π'_{jk} . The group then sends the public transcript $\text{tr}_{2,T_i} = ((S_{jk}, \pi'_{jk})_{j,k \in T_i})$ to the client.
4. For each T_i , upon receiving tr_{2,T_i} , the client executes steps 4 and 5 of RandShare protocol on behalf of the participants to obtain the output Ω_{T_i} and abort if $\Omega_{T_i} = \perp$.
5. The final output Ω is computed by the client by aggregating all the Ω_{T_i} as follows.

$$\Omega = \prod_{i=1}^{n/c} \Omega_{T_i}.$$

Finally, an external verifier can use tr_{1,T_i} and tr_{2,T_i} to verify the correctness of Ω .

Analysis. RandHound partitions the n participants into groups, each containing c members. Since performing PVSS requires $\mathcal{O}(c^3)$ communication complexity for a group of size c , and there are n/c such groups, the communication complexity of RandHound is reduced to $\mathcal{O}(c^2n)$. The same reason applies to computation and verification complexity. Thus Randhound achieve $\mathcal{O}(c^2n)$ in all complexities.xx Now to the disadvantage of RandHound, the Liveness property of RandHound requires the client to be honest. It can be seen that the protocol aborts when one of the group T_i does not follow the protocol. A dishonest client can partition the participants so that he controls the group T_1 and learns Ω_{T_1} at the start by ordering T_1 to pre-execute Step 1 and 3 in secret and bias the output as follows: After other groups T_j finish executing Step 3, the client can execute step4 by himself to learn Ω_{T_j} for each $j \neq 1$ and multiplying them to learn Ω . If Ω somehow benefits him, he orders T_1 to perform Step 3 honestly. Otherwise, he orders T_1 to abort the protocol in Step 4 by not executing Step 3.

3. RandHerd

RandHerd [SJK⁺17] is a leader-based PVSS protocol that uses RandHound as a setup. It also divides the set of participants into smaller groups $T_1, T_2, \dots, T_{n/c}$ like RandHound. RandHerd attempts to remove the involvement of a client in RandHound, since a dishonest client can cause the protocol to abort. It instead consists of a leader L who aggregates the output of each group T_i , and each group T_i also consists of a group leader L_i who aggregates the output of each group member to produce one single group output. The main idea of RandHerd is to employ both VSS and PVSS to generate a Schnorr Signature [Sch91] on a message M , which serves as the random output of the protocol. At a high level, the Schnorr signature $\Omega = (c, v)$ for a message M has the following form

$$v = r - x \cdot c, \quad c = H(C||M), \quad C = g^r,$$

where $r \xleftarrow{\$} \mathbb{Z}_p$ and x is the secret key of the signature scheme. As we can see above, to generate Ω , it suffices to generate r, x in a distributed manner. The generation of x is complicated: It employs a one-time setup protocol that uses RandHound as a sub-protocol (this is where PVSS is used). At the end of the setup, all participants are divided into groups of size c , and the leader L_i of group T_i is also determined from the setup. In addition, the leader L_i receives a secret x_i , and the secret x is equal to $x = \sum_i x_i$. On the other hand, generating r is simple: After the setup, the leader L_i of each group T_i samples his own secret $r_i \xleftarrow{\$} \mathbb{Z}_p$ and executes the VSS of Feldman (see **Protocol 5**) to commit r_i , which can be reconstructed by participants. At this time, recall in **Protocol 5** that the value $C_i = g^{r_i}$ is included in the commitment of r_i in Feldman's VSS, hence all participants can calculate $C = g^r = \prod_i C_i$ and $c = H(C||M)$. When r_i is reconstructed, and the value r is equal to $r = \sum_i r_i$. Finally, given r_i and x_i , the leader L_i can calculate $v_i = r_i - c \cdot x_i$, and send v_i to the L , who calculates $v = \sum_i v_i$. Hence, the value v is actually equal to $r - c \cdot x$. With this, the main steps of RandHerd involve the generation Ω are as follows.

Overview of RandHerd

1. Each participant T_i executes a one-time setup phase using RandHound as a sub-protocol. In the end, the participant is divided into n/c groups T_i , each having size c , and each leader L_i holds a secret x_i and its commitment $P_i = g^{x_i}$ is known to all participants in T_i as well as the leader L . The leader L then aggregate

$$P = g^x = \prod_{i=1}^{n/c} P_i.$$

2. The leader L choose a message $M \xleftarrow{\$} \mathbb{Z}_p$ and broadcast M to all groups. The main idea of the protocol is to produce a valid Schnorr signature on M .
3. Each group leader L_i chooses a secret $r_i \xleftarrow{\$} \mathbb{Z}_p$ and perform the **VSSShare** protocol of Feldman on r_i (see **Protocol 5**) to create the shares of r_i and its commitment $C_i = g^{r_i}$. Participants execute **VSSShareVerify** to check the validity of their shares and send their check result to the leader L .
4. The leader L aborts if any group has an invalid check. Otherwise, L aggregates

$$C = g^r = \prod_{i=1}^{n/c} C_i.$$

He then computes the challenge $c = \text{H}(C||M)$ and sends c to all group leaders L_i .

5. The participants in each group T_i execute **VSSReconstruct** (see **Protocol 5**) to recover r_i . The leader L_i then computes $v_i = r_i - cx_i$ and sends the value v_i to the leader L .
6. The leader L aggregates $v = \sum_{i=1}^{n/c} v_i$. Finally, the output Ω is computed as

$$\Omega = (c, v).$$

One can check that Ω is a valid Schnorr signature of M , as discussed above.

7. An external verifier can simply check the correctness of Ω by computing $C^* = g^c P^v$ and check if $c \stackrel{?}{=} \text{H}(C^*||M)$ which is exactly the verification produce of Schnorr signature.

Analysis. Although the authors of RandHerd claim that RandHerd achieves only $\mathcal{O}(c^2 \log n)$ communication and computation complexity, we cannot see the reason for this claim. Since each group has to perform a PVSS scheme to create the secret r_i , at least $\Omega(c^3)$ messages are sent in each group. Hence the communication complexity of RandHerd is $\Omega(c^2 n)$. For computation complexity, the leader has to aggregate n/c group elements, hence the computation complexity cannot be less than $\Omega(n/c)$. For verification complexity, RandHerd achieve $\mathcal{O}(1)$ verification cost, since a verifier only has to verify a single Schnorr signature. For security analysis, RandHerd does not provide Unbiasability and Liveness. This is because RandHerd introduces a leader in each group, and a single dishonest leader who does not follow the protocol can cause RandHerd to abort. For example in Step 5, consider the leader L_1 and L that are both corrupted. Then L secretly learns v_1 of L_1 and compute v in Step 6 after all other groups L_i send v_i . If v does not benefit him, he can order L_1 to withhold v_1 . Hence, RandHerd requires additional mechanisms to provide Unbiasability and Liveness, which increases the communication complexity and may affect scalability. Nevertheless, even with additional mechanisms, the Unbiasability and Liveness properties can only be achieved against a static adversary. An adaptive adversary can outright cause the protocol to abort when he corrupts all participants in T_1 after the group is defined after the setup.

4. SPURT

SPURT [DKIR22] is a leader-based PVSS DRNGs that aims to reduce the cost of SCRAPE. The main idea of SPURT is to let the leader “aggregate” n PVSS checkings of SCRAPE into one single PVSS checking. To do so, let us take a look at the shares $S_{ij} = h^{s_{ij}}$, share commitments $C_{ij} = g^{s_{ij}}$ and encrypted shares $E_{ij} = \text{pk}_j^{s_i}$ of SCRAPE’s PVSS construction described in Subsubsection 2.4.7.3 and Subsubsection 3.2.2. Let us denote $f_i(x)$ to be the polynomial chosen by P_i , we consider the polynomial $f^*(x) = \sum_{i=1}^n f_i(x)$ and let $s_j^* = f^*(j) \forall j \in \{0, \dots, N\}$.

With this setting above, one can see that, if $S_{ij} = h^{f_i(j)} = h^{s_{ij}}$ is the j -th share of P_i , then $S_j^* = \prod_{i=1}^n S_{ij} \forall j \in \{1, \dots, n\}$ is actually equal to $h^{s_j^*}$, which is the j -th share of $h^{s_0^*}$. Similarly, the values $C_j^* = \prod_{i=1}^n C_{ij} = g^{s_j^*}$ and $E_j^* = \prod_{i=1}^n E_{ij} = \text{pk}_j^{s_j^*}$ are the commitment and encrypted share of the secret $S_j^* = h^{s_j^*}$. With this in mind, SPURT lets participant P_i execute PVSSShare protocol of the PVSS scheme to produce $C_i = \{C_{ij}\}_{j=1}^n, \{E_{ij}\}_{j=1}^n$, but send them to the leader instead of distributing them. The leader then compute $\{C_i^*\}_{i=1}^n, \{E_i^*\}_{i=1}^n$ as

above and acts as the PVSS dealer of $S_0^* = h^{s^*(0)} = h^{f^*(0)}$, which is the output of the protocol. However, there is one problem remains: Recall in **Protocol 7**, to act as the PVSS leader of S_0^* , the leader has to create a proof π_j^* certifying $(g, C_i^*, \mathbf{pk}_i, E_i^*, s_j^*) \in \mathcal{R}_{\text{DLOG}}$, where $\mathcal{R}_{\text{DLOG}}$ is defined in Subsubsection 2.4.6. In addition, he has to prove that the commitments $\{C_i^*\}_{i=1}^n$ are commitments that come from a polynomial of degree t and the values E_i^* and C_i^* are aggregated correctly. However, since the leader does not know the value s_j^* he cannot create such a proof π_j^* by himself. Fortunately, the leader can still prove all the above by using the proofs $\{\pi_{ij}\}_{1 \leq i, j \leq n}$. For each j , recall that the proof π_{ij} certifies the relation

$$(g, C_{ij}, \mathbf{pk}_j, E_{ij}, s_{ij}) \in \mathcal{R}_{\text{DLOG}}. \quad (3.4)$$

In addition, recall that the values C_j^*, E_j^* and s_j^* are calculated as follows

$$C_j^* = \prod_{i=1}^n C_{ij}, \quad E_j^* = \prod_{i=1}^n E_{ij}, \quad s_j^* = \sum_{i=1}^n s_{ij}. \quad (3.5)$$

From (3.4) and (3.5), it follows that $(g, C_j^*, \mathbf{pk}_j, E_j^*, s_j^*) \in \mathcal{R}_{\text{DLOG}}$. In addition, [DKIR22, Appendix C] also proves the check above also implies that the leader has aggregated the shares correctly. Hence, participants have finished checking $(g, C_i^*, \mathbf{pk}_i, E_i^*, s_j^*) \in \mathcal{R}_{\text{DLOG}}$ using the proof $\pi_j^* = \{(E_{ij}, C_{ij}, \pi_{ij})\}_{i=1}^n$. Now, to check that $\{C_i^*\}_{i=1}^n$ are commitments that come from a polynomial of degree t , then participants can execute Steps 2a and 2b of RandShare using $\{C_i^*\}_{i=1}^n$ as the input. Hence all checks are finished. However, note that using the checks above, the leader has to send $\{\pi_i^*\}_{i=1}^n$ to all participants, which incurs $\mathcal{O}(n^3)$ communication complexity. Hence the leader only **sends** π_j^* and $\{(E_i^*, C_i^*)\}_{i=1}^n$ **secretly to participant** P_j to perform the check for each j , and thus communication complexity is reduced to $\mathcal{O}(n^2)$. In addition, to let participants reach consensus on the common values $\{(E_i^*, C_i^*)\}_{i=1}^n$ with low communication complexity, the leader additionally computes **digest** = $\text{H}(\{(E_j^*, C_j^*)\}_{j=1}^n)$ and sends **digest** to all participants, who execute a State Machine Replication protocol to reach consensus on **digest**. Finally, given the agreed message **digest**, participants can check if the common value $\{(E_i^*, C_i^*)\}_{i=1}^n$ is sent consistently by the leader.

Now that all participants have finished checking the validity of $\{C_i^*, E_i^*\}_{i=1}^n$, the remaining is just to decrypt the shares S_i^* and reconstruct the original secret S_0^* using **PVSSShareReveal** and **PVSSReconstruct**. With this, the main steps of SPURT can be summarized as follows.

Overview of SPURT

1. Each participant P_i prepares his secret share S_i and compute

$$(C_i, \{(E_{ij}, \pi_{ij})\}_{j=1}^n) \leftarrow \text{PVSSShare}(S_i)$$

and sends all values $(C_i, \{(E_{ij}, \pi_{ij})\}_{j=1}^n)$ to the leader.

2. For each j , the leader computes the following:

$$\pi_j^* = \{(E_{ij}, C_{ij}, \pi_{ij})\}_{i=1}^n, \quad E_j^* = \text{pk}_j^{s_j^*} = \prod_{i=1}^n E_{ij}, \quad C_j^* = g^{s_j^*} = \prod_{i=1}^n C_{ij},$$

and sends π_j^* and $\{(E_i^*, C_i^*)_{i=1}^n\}$ to participant P_j .

3. For each i , participant P_i check the correctness of E_i^* and C_i^* as follows:

- (a) Check if $\text{H}(\{(E_i^*, C_i^*)\}_{i=1}^n) \stackrel{?}{=} \text{digest}$.
- (b) Check if $\text{NIZKDLOGVerify}(g, C_{ji}, \text{pk}_i, C_{ji}, \pi_{ji}) \stackrel{?}{=} 1 \forall 1 \leq j \leq n$.
- (c) Check if $C_i^* \stackrel{?}{=} \prod_{j=1}^n C_{ji}$ and $E_i^* \stackrel{?}{=} \prod_{j=1}^n E_{ji}$.
- (d) Execute Steps 2a and 2b of SCRAPE (see Subsubsection 3.2.2.1) to check that the values $\{C_i^*\}_{i=1}^n$ are commitments come from a polynomial of degree t .

If any checks fail, P_i broadcasts a complaint. If there are $t + 1$ complaints, then the leader is marked as invalid. Otherwise, participants proceed to Step 4.

4. Each participant P_i reveals his share S_i^* by executing

$$(S_i^*, \pi_i'^*) \leftarrow \text{PVSSShareReveal}(E_i^*, \text{sk}_i, \text{pk}_i).$$

When at least $t + 1$ valid shares $\{S_i^*\}_{i \in \mathcal{V}}$ are revealed for some $\mathcal{V} \subseteq \{1, \dots, n\}$, the final output Ω can be computed as follows.

$$\Omega = \text{PVSSReconstruct}(\mathcal{V}, \{(S_i^*, E_i^*, \text{pk}_i, \pi_i'^*)\}_{i \in \mathcal{V}}).$$

Analysis. For security, SPURT achieves full security properties against a static adversary and is not susceptible to adaptive adversaries, unlike most leader-based PVSS constructions. For efficiency, each participant has to send $\mathcal{O}(n)$ shares to the leader, and the leader also has to send $\mathcal{O}(n)$ group elements (the values $\{(E_i^*, C_i^*)\}_{i=1}^n$) to each participant, hence the total communication complexity is dominated by $\mathcal{O}(n^2)$. In addition, they employ State Machine Replication to lower the communication complexity compared to the broadcast channel used by other DRNGs. The usage of State Machine Replication also allows them to achieve Liveness under a partial synchronous network model. However, in each epoch, it suffers $\mathcal{O}(n^2)$ computation complexity, this is because the leader has to compute E_i^* from $\{E_{ij}\}_{j=1}^n$ for each i , this takes $\mathcal{O}(n)$ computation complexity. Since there are n values E_i^* , the computation complexity in each epoch is $\mathcal{O}(n^2)$.

3.2.3 DRNG from Threshold Signature

Recall that a (t, n) -threshold signature $\text{TSS} = (\text{TSSSetup}, \text{TSSSign}, \text{TSSCombine}, \text{TSSVerify})$ with n participants allows any set of $t + 1$ participants to sign a message, but any set of no more than t participants cannot forge a valid signature for a message. Due to the unpredictability of the signature, especially with the involvement of multiple participants in threshold signatures, some DRNG constructions, namely [Drave, CKS05, HMW18], employ threshold signatures described in Subsubsection 2.4.10.2 as the core ingredient to produce unpredictable random outputs. Although there exist various threshold signature schemes, most existing threshold signature-based DRNG rely on the BLS threshold signature [BLS04]. This is because BLS signatures are deterministic, hence the signature, or the protocol output cannot be biased (changed) by any adversaries. The framework of all threshold-based DRNG can be described as follows.

Overview of Threshold Signature-based DRNGs

1. Participants follow the setup protocol TSSSetup

$$((\mathbf{pk}, \mathbf{sk}), (\mathbf{pk}_i, \mathbf{sk}_i)_{i \in \text{QUAL}}, \text{QUAL}) \leftarrow \text{TSSSetup}(1^\lambda) \langle \{P\}_{P \in \mathcal{P}} \rangle$$

to select the list QUAL of qualified participants and generate a public-secret key pair $(\mathbf{pk}, \mathbf{sk})$ and partial public-secret key pairs $(\mathbf{pk}_i, \mathbf{sk}_i)$ for participant P_i . This setup protocol often incurs high complexity and is executed once every a fixed number of epochs.

2. Let r be the current epoch X_r be the current input. Each participant P_i in the set QUAL uses his secret key \mathbf{sk}_i to create a partial signature

$$\sigma_{ri} \leftarrow \text{TSSign}(X_r, \mathbf{sk}_i)$$

and publishes σ_{ri} . Other participants verify σ_{ri} by running $\text{TSSVerify}(X_r, \sigma_{ri}, \mathbf{pk}_i)$.

3. When $t + 1$ partial signatures $\{\sigma_{ri}\}_{i \in \mathcal{V}_r}$ are confirmed valid, participants execute

$$\Omega_r = \text{TSSCombine}(\mathcal{V}_r, \{\sigma_{ri}\}_{i \in \mathcal{V}}) = \text{TSSSign}(X_r, \mathbf{sk})$$

to produce a unique signature Ω_r which serve as the output of the protocol. The next input X_{r+1} will be determined from Ω_r . For example, if threshold BLS signature is used, then the TSSCombine algorithm returns the output $\Omega_r = \sum_{i \in \mathcal{V}} \sigma_{ri}^{\lambda_{i,\mathcal{V}}}$, where $\lambda_{i,\mathcal{V}} = \prod_{j \in \mathcal{V}, j \neq i} \frac{j}{j-i}$ is the Lagrange coefficient w.r.t \mathcal{V} .

4. An external verifier can validate the output Ω by executing $\text{TSSVerify}(X_r, \Omega_r, \mathbf{pk})$.

Although having the same framework, each DRNG has different instantiation in its protocol. Below we present a description of threshold signature-based DRNGs.

1. Cachin *et al.*

Cachin [CKS05] *et al.* presented a common coin protocol using a threshold signature scheme and follows the exact framework above to generate random outputs. For setup, a trusted dealer chooses a random secret key \mathbf{sk} and performs a VSS scheme of Feldman in Subsubsection

2.4.7.2 to distribute the secret key shares to all participants. Each participant P_i then uses his share \mathbf{sk}_i to partially produce the DRNG output, as described above. Cachin’s *et al.* construction enjoys $\mathcal{O}(n^2)$ communication complexity for setup due to letting a single trusted dealer distribute the secret key \mathbf{sk} . However, relying on a trusted dealer is a very strong assumption and this contradicts the concept of a decentralized protocol.

2. Drand and DFINITY

Drand [Drave] and DFINITY [HMW18] follow a similar idea, however, they replace a VSS scheme with a DKG of Gennaro *et al.* [GJKR99] to produce the public-secret key pair $(\mathbf{pk}, \mathbf{sk})$ and partial public-secret key pairs $(\mathbf{pk}_i, \mathbf{sk}_i)$ for participants. In addition, they instantiate the threshold signature with a threshold BLS signature [BLS04] to produce a deterministic signature. Although the DKG yields $\mathcal{O}(n^3)$ communication complexity, it requires no trusted setup, making them achieve better security. In addition, by using deterministic signature, the output of Drand and DFINITY is unpredictable yet deterministic, hence preventing an adversary from biasing the output for his goal. The difference between Drand and DFINITY is that DFINITY has additional network-related mechanisms to achieve security in a partially synchronous network, while Drand can only achieve security in a synchronous network.

Analysis. For efficiency, it can be noted that all threshold signature-based DRNGs require a third party, or a DKG to distribute the secret key and public key for qualified participants. Using DKG incurs $\mathcal{O}(n^3)$ communication complexity for setup in the first epoch, and creates a disadvantage for the need of replacing participants, which we will discuss later in the next section. From the second epoch forward, the communication complexity of all these protocols is $\mathcal{O}(n^2)$, since each participant P_i has to send his partial signature σ_i to $\mathcal{O}(n)$ other participants. If threshold BLS signature [BLS04] is used, the computation complexity for each participant is $\mathcal{O}(n \log^2 n)$ upon optimization, dominated by the cost of computing Lagrange coefficients as seen in the framework above, this will be explained later in Section 4.5.3.

For security, if unique signature (for example, BLS signature) is used, then from any set $t + 1$ valid partial signatures $\{\sigma_i\}_{i \in \mathcal{V}}$, the $\text{TSSCombine}(\mathcal{V}, \{\sigma_i\}_{i \in \mathcal{V}})$ always returns an **unique signature Ω , regardless of the set \mathcal{V}** . This prevents an adversary from biasing the output. In addition, since there are at least $t + 1$ honest participants who always submit valid signatures, the signature Ω is guaranteed to be produced correctly. Hence

threshold signature-based DRNGs achieve full security properties. The security against adaptive adversaries of threshold signature-based DRNGs depends on the instantiation of the DKG protocol, as the construction of Gennaro *et al.* [GJKR99] is only secure against static adversaries, while [CGJ⁺99] provides security against adaptive adversaries. However, this is a tradeoff between security and efficiency, since [CGJ⁺99] incurs more communication and computation complexity than [GJKR99].

3.2.4 DRNG from Verifiable Random Function

Recall that the goal of a VRF scheme is to produce pseudo-random outputs in a way such that these outputs can be publicly verified. Hence, it is natural to think of a decentralized protocol that employs a VRF scheme $\text{VRF} = (\text{VRFSetup}, \text{VRF Eval}, \text{VRF Verify})$ in Section 2.4.12 to produce publicly verifiable random outputs. Protocols that follow this approach are [GHM⁺17, DGKR18, GLOW21]. It's noteworthy that all VRF-based DRNGs exhibit low communication and computation complexity, making them suitable for scenarios involving a substantial number of participants. Based on protocol design, VRF-based DRNG can be further categorized into two types: Algorand and Ouroboros [GHM⁺17, DGKR18], which follow the “priority selection” approach like proof-of-work (PoW), while the other is the DVRF protocol of Galindo *et al.* [GLOW21], which is based on threshold signature. Below we describe these protocols.

1. Algorand and Ouroboros Praos

Algorand [GHM⁺17] and Ouroboros Praos [DGKR18] also adopt the “priority selection” approach, with the distinction that each participant employs a Verifiable Random Function (VRF) to produce an output instead of undertaking the task of solving a puzzle. The selection process in these protocols is based on the VRF output produced by participants. The advantage of Algorand and Ouroboros over PoW is that solving PoW puzzles requires very high computational cost and hardware power, hence a lot of energy will be wasted. By utilizing VRF, these protocols achieve minimal computation complexity, as VRF computations can be efficiently performed by every participant. Now, let us delve into the primary steps of these protocols to see how they work. Algorand and Ouroboros Praos can be briefly described using a three-step process below:

Overview of Algorand and Ouroboros Praos

1. Let r be the current epoch and Ω_{r-1} be the output of the previous epoch. Each participant P_i , with a given role $role_i$, computes

$$(Y_{ri}, \pi_{ri}) \leftarrow \text{VRFEval}(\Omega_{r-1} || role_i, \mathbf{sk}_i)$$

which produces an output Y_{ri} and its proof π_{ri} .

2. From the set of participants that have contributed their VRF outputs, a selection algorithm **Select** is used to determine a participant P_j with the highest priority. For example, Algorand's selection algorithm **Select** is set to be

$$\text{Select}(P_j) = Y_{rj} \stackrel{?}{<} \text{target}$$

where **target** is chosen so that with high probability, exactly one participant is selected.

3. Participants then execute $\text{VRFVerify}(\Omega_r || role_j, Y_{rj}, \pi_{rj}, \mathbf{pk}_j)$ to check the validity of Y_{rj} . If valid, the output Ω_r of the protocol is computed by P_j using his secret key \mathbf{sk}_j and the outputs of previous epochs. For example, Algorand's output Ω_r is set to be

$$\Omega_r = \text{VRFEval}(\Omega_{r-1} || r, \mathbf{sk}_j).$$

Analysis. Both protocols follow the same framework described above, with minor difference between the two in the computation of Ω_r and the selection algorithm **Select**. The communication complexity of both protocols is $\mathcal{O}(n^2)$ in the worst case because each participant has to send his VRF output to $n - 1$ other participants. The computation complexity of each protocol is only $\mathcal{O}(1)$, since only the computation of VRFEval , **Select**, Ω_r are done, and each require only $\mathcal{O}(1)$ computation steps by each participant. The verification complexity of an external verifier of these protocols is also $\mathcal{O}(1)$, since an external verifier only has to execute perform a single VRF verification on the output Ω_r , making them very cost efficient.

However, despite the efficiency, both protocols do not provide Unbaisability. We see that the output of the protocol is the VRF value of a selected participant through some selection

algorithm **Seclect**. Consequently, the output is influenced by the subset of participants contributing to the VRF result. Consider a malicious participant P_i , whose value Y_{ri} passes the selection algorithm. He then checks the output Ω_r is computed by his VRF in step3. If it does not favor him, he can choose to withhold his output Y_{ri} and rely on the remaining participants (including his colluding participants) to produce an output Ω_r that benefits him. Additionally, both protocols fall short of achieving Pseudo-randomness, despite their use of VRF. This is because in the event that a corrupted participant P_i is selected as a leader, an adversary can look at the output of P_i and distinguish it from random, and the probability that such a corrupted participant P_i is chosen is $1/n$, which is non-negligible. However, they still achieve Unpredictability, the weaker version of Pseudo-randomness.

2. DVRF

Inspired by threshold BLS signatures, in 2021, Galindo *et al.* [GLOW21] proposed a **distributed verifiable random function** (DVRF). Their idea is motivated by a (t, n) -BLS threshold signature. The concept was drawn from (t, n) -BLS threshold signatures, with the aim of developing a distributed version of a VRF that, given any $t + 1$ valid partial VRF values, allows the computation of a unique final VRF output. However, any subset of t partial VRF values would be insufficient to produce such an output. This mirrors the underlying principle of a threshold signature but with the signature scheme replaced by a VRF. The authors leveraged a specific VRF construction from [PWH⁺17], which closely resembled the structure of a BLS signature scheme, enabling them to achieve the desired (t, n) -threshold property. With this concept, the construction of DVRF is briefly described below.

Overview of DVRF

1. In the first epoch, all participants take part in a DKG protocol to select the list **QUAL** of qualified participants and create a public-secret key pair $(\mathbf{pk}, \mathbf{sk})$ and shared public-secret key pairs $(\mathbf{pk}_i, \mathbf{sk}_i)$ for each participant.
2. From the second epoch forward, let X_r be the input of the current epoch. Each participant in the set **QUAL** uses his secret key \mathbf{sk}_i to compute VRF partial output values

$$(Y_{ri}, \pi_{ri}) \leftarrow \text{VRF Eval}(X_r, \mathbf{sk}_i)$$

and broadcasts (Y_{ri}, π_{ri}) . Other participants verify Y_{ri} using $\text{VRF Verify}(X_r, Y_{ri}, \pi_{ri}, \mathbf{pk}_i)$

3. When $t + 1$ valid VRF outputs $\{Y_{ri}\}_{i \in \mathcal{V}}$ have been published, a combination procedure is used to produce a final output Ω via Lagrange interpolation as follows.

$$\Omega_r = \prod_{i \in \mathcal{V}} Y_{ri}^{\lambda_{i,\mathcal{V}}}$$

where $\lambda_{i,\mathcal{V}} = \prod_{j \in \mathcal{V}, j \neq i} \frac{j}{j-i}$ is the Lagrange coefficient w.r.t \mathcal{V} . Using the VRF construction of [PWH⁺17], it is guaranteed that the final output Ω_r will always be equal to $\text{VRF Eval}(X_r, \mathbf{sk})$. The next input X_{r+1} will be determined from Ω_r .

Analysis. It is evident that the output of DVRF is deterministically determined by the input, preventing any biasing attempts by adversaries. The communication complexity and computation complexity are both $\mathcal{O}(n^2)$ and $\mathcal{O}(n \log^2 n)$ respectively, which is a consequence of sharing the same structural characteristics as threshold signature-based DRNGs. DVRF also achieves full security properties and has the same drawback using DKG like threshold signature-based DRNGs. This makes DVRF one of the most balanced DRNG construction, since it achieves quasi-linear computation complexity, quadratic communication complexity and satisfies all security properties. However, DVRF is better than threshold signature-based DRNGs in the sense that it is formally proven achieves Pseudo-randomness, while threshold signature-based DRNGs is only proven to achieve the weaker notion Unpredictability.

3.2.5 DRNG from Homomorphic Encryption

These DRNGs employ a homomorphic encryption scheme $\text{HE} = (\text{HESetup}, \text{HEEnc}, \text{HEDec})$ as detailed in Subsection 2.4.9 to generate randomness. Protocols that follow this direction are [NNL⁺19, CSS19]. Notably, all of these protocols rely on the ElGamal encryption scheme (see Example 2.4.12) as the chosen homomorphic encryption scheme for their construction.

The motivation for using encryption schemes is also drawn from the approach of RANDAO. As a quick recap, encryption allows the use of a public key pk to transform a plaintext M into a ciphertext C in a way that, the owner of the secret key sk can decrypt C back to M , while those without access to sk gain no information about M . Hence, one may naturally construct a RANDAO style DRNG where M_i acts as the secret of the participant P_i and C_i acts as the “commitment” of M_i , where the secret key owner can efficiently verify the correctness of M_i using C_i . Assume that (\mathcal{M}, \cdot) forms a group, then the output M is the “sum” of all M_i , i.e.,

$$M = M_1 \cdot M_2 \cdot \dots \cdot M_n.$$

Similarly, assuming that $(\mathcal{C}, *)$ also forms a group, one can utilize a homomorphic encryption scheme (for example, ElGamal encryption scheme) in Subsection 2.4.9 to enable the values C_i to be collectively aggregated into a single value C , i.e.

$$C = C_1 * C_2 * \dots * C_n.$$

Due to the homomorphic property, it is evident that C is the encrypted ciphertext of M . Hence, instead of all n checks of M_i using C_i as in RANDAO, it suffices to aggregate all the C_i into single value C to check the correctness of M , which greatly reduces the cost of verification. The critical question that arises is: Who owns the secret key sk to perform the decryption of C back to M and verify the correctness of M using C ? Furthermore, given that RANDAO is susceptible to withholding attacks, a direct replication of its steps may not be suitable. Therefore, modifications are needed to mitigate this vulnerability. Nguyen et al. and HERB offer distinct solutions to these challenges, which we will explore in detail below.

Now, let us delve into the technical details of these DRNGs. Broadly speaking, the process of generating randomness of these DRNGs consists of two smaller phases: **Encryption** and **Decryption**. In the **Encryption** phase, both protocols follow the same framework, as

described below.

Encryption Phase of HE-based DRNGs

1. Each participant P_i , with the common public key pk known to all participants, uniformly sample a secret plaintext $M_i \in \mathcal{M}$, then encrypt M_i by computing

$$C_i = \text{HEEnc}(M_i, \text{pk}).$$

Participant P_i then publishes the ciphertext C_i to all other participants.

2. When all of the ciphertexts C_1, C_2, \dots, C_n are published by participants, they are homomorphically added to produce a unique ciphertext C as follows. $C = \prod_{i=1}^n C_i$.

Due to the homomorphic property of HE, the ciphertext C is the encrypted ciphertext of the plaintext $M = \prod_{i=1}^n M_i$ i.e, it holds that $C = \text{HEEnc}(M, \text{pk})$.

Now that the ciphertext C is published, the subsequent task is to decrypt C back to the plaintext M via the **Decryption** phase. In the **Decryption** phase, each construction employs a different method to decrypt the ciphertext C to the plaintext M . This plaintext M ultimately serves as the output of the protocol. Now that we have gotten the concept of HE-based DRNGs, below we shall describe the difference between the two protocols.

1. Nguyen *et al.*

Nguyen *et al.* 's DRNG [NNL⁺19] follows a client-server architecture, similar to RandHound, which also involves a requester. In Nguyen *et al.* 's construction, the requester will generate (pk, sk) and publish pk so that participants can perform the **Encryption** phase. After the **Encryption** phase is finished, the client uses his secret key sk to decrypt C back to M in the **Decryption** phase. In addition, the protocol does not allow all n participants to execute the protocol. Instead, it employs a sampling process using VRF to select a smaller set of qualified participants to participate in the random generation process in each epoch. The reason for this sampling process is to further improve the efficiency of the protocol since the communication and computation complexity depend on the number of participants. Some remarkable steps of the construction of Nguyen *et al.* can be seen below.

Overview of Nguyen *et al.*

1. The requester uses **HESetup** to produce a public-secret key pair \mathbf{pk}, \mathbf{sk} and generate a value T . The public key \mathbf{pk} acts as the common public key.
2. Participants join in a sampling process to select a number of t qualified members by using a VRF scheme $\text{VRF} = (\text{VRFSetup}, \text{VRFEval}, \text{VRFVerify})$. Each participant P_i use **VRFSetup** to create a public-secret key $(\mathbf{pk}_i, \mathbf{sk}_i)$, then compute

$$(Y_i, \pi_i) = \text{VRFEval}(T, \mathbf{sk}_i).$$

3. A threshold **target** is selected to determine the list of qualified members. A participant is qualified if and only if $Y_i < \text{target}$. This is similar to the selection process of Algorand.
4. Qualified participants execute the **Encryption** phase described above to obtain the joined ciphertext C and send C to the requester.
5. The requester executes the **Decryption** phase by using his secret key \mathbf{sk} to decrypt the joined ciphertext C by computing

$$M = \text{HEDec}(C, \mathbf{sk}).$$

Then the requester publishes M with the proof of correct decryption.

Analysis. In Nguyen *et al.* construction, once the ciphertext C is aggregated from C_i , the requester owning \mathbf{sk} can decrypt C back to M directly, hence participants are not required to reveal their partial secret M_i like RANDAO. This makes the construction safe against the withholding attack of RANDAO, because during the protocol execution, only the ciphertexts C_i s are published by the participants, which reveals no information of the plaintext M_i . In addition, the final plaintext M can only be revealed by the requester after he receives it from decrypting C , but at this time the participants have already finished their role.

The communication, computation and verification complexity of [NNL⁺19]’s construction is $\mathcal{O}(n^2)$, $\mathcal{O}(n)$ and $\mathcal{O}(n)$ respectively, rendering the protocol highly cost-effective. However, for security, the protocol crucially relies on the requester’s honesty. A trivial attack on this

scheme, when the participant is dishonest, can be described as follows: If the requester desires the output to be M^* and gives his secret key \mathbf{sk} to a colluding participant, say P_n , then P_n can decrypt the sum of the secret of other participants by calculating

$$M' = \text{HEDec} \left(\prod_{i=1}^{n-1} C_i, \mathbf{sk} \right).$$

Subsequently, P_n calculate $M_n = M^* \cdot (M')^{-1}$ and finally publishes the ciphertext $C_n = \text{HEEnc}(M_n, \mathbf{pk})$. In this case, the aggregated ciphertext C is calculated to be

$$C = \prod_{i=1}^n C_i = \text{HEEnc}(M' \cdot M_n, \mathbf{pk}) = \text{HEEnc}(M^*, \mathbf{pk}).$$

Hence, when the client decrypts C he will get the output M^* . This highlights that the protocol of Nguyen *et al.* does not provide Unbiasality when the client is dishonest.

Another problem of Nguyen *et al.*'s protocol is that it assumes that $t = n - 1$, i.e., only one participant is honest. However, the protocol includes a process of selecting a set of t qualified members. By treating VRF as a random function, we assume that each participant has an equal probability of being qualified. By some calculation, we see that the probability of selecting t dishonest participants is $\binom{n-1}{t} / \binom{n}{t} = (n-t)/n$. Hence, if we let $t = n/2$, then with probability $1/2$, the protocol consists of only dishonest participants. Therefore, it is not reasonable to make such an assumption.

2. HERB

HERB [CSS19] is also a HE-based DRNG. However, it is inspired by threshold BLS signatures and thus follows a different approach from Nguyen *et al.* Unlike Nguyen *et al.*, HERB eliminates the involvement of a client in the random generation process. Instead, HERB utilizes a (t, n) -**threshold ElGamal encryption scheme** together with a Distributed Key Generation (DKG) protocol for setup. Broadly speaking, a threshold ElGamal encryption scheme enables n participants to jointly decrypt a ciphertext C produced by the ElGamal encryption scheme (again, see Example 2.4.12) on a plaintext M in a way that: Any $t + 1$ participants can successfully decrypt the ciphertext C back to the plaintext M , while any t participants cannot. One can see that this idea mirrors the concept of a threshold signature.

Now, the two primary distinctions between HERB and Nguyen et al. are outlined as follows.

Difference between HERB and Nguyen *et al.*

1. For setup, participants execute the DKG protocol of Gennaro *et al.* [GJKR99] to select the list **QUAL** of qualified participants and create a public-secret key pair $(\mathbf{pk}, \mathbf{sk})$ and partial public-secret keys $(\mathbf{pk}_i, \mathbf{sk}_i)$ for participant P_i . This differs from the construction of Nguyen *et al.* in two ways:
 - (a) The public key \mathbf{pk} and secret key \mathbf{sk} is created by the DKG protocol instead of the requester. In addition, each participant P_i also receive a public-secret key $(\mathbf{pk}_i, \mathbf{sk}_i)$ to partially decrypt C .
 - (b) The list of qualified **QUAL** participants is determined by the DKG protocol instead of running a VRF scheme as in Nguyen's *et al.*'s protocol.
2. In the **Decryption** phase, the client is no longer needed. Instead, participants jointly decrypt a ciphertext of ElGamal cryptosystem in Example 2.4.12, which has the form

$$C = (C_1, C_2) \text{ where } C_1 = g^r \text{ and } C_2 = M \cdot g^{\mathbf{sk} \cdot r}.$$

To decrypt the ciphertext, each participant uses his secret key \mathbf{sk}_i to compute

$$(D_i, \pi_i) = (C_1^{\mathbf{sk}_i}, \text{NIZKDLOGProve}(g, \mathbf{pk}, C_1, D_i, \mathbf{sk}_i))$$

and publish D_i along with its proof π_i of correctness. From a set \mathcal{V} of size $t + 1$ and a list of valid partial decryptions $\{D_i\}_{i \in \mathcal{V}}$, a combine algorithm is executed to recover the plaintext M via Lagrange interpolation as follows.

$$D = \prod_{i \in \mathcal{V}} D_i^{\lambda_{i, \mathcal{V}}} = C_1^{\sum_{i \in \mathcal{V}} \mathbf{sk}_i \cdot \lambda_{i, \mathcal{V}}} = C_1^{\mathbf{sk}} \quad \text{and} \quad M = C_2 \cdot D^{-1}$$

where $\lambda_{i, \mathcal{V}} = \prod_{j \in \mathcal{V}, j \neq i} \frac{j}{j-i}$ is the Lagrange coefficient w.r.t \mathcal{V} . It can be proved that any $t + 1$ valid decryption determines the same unique plaintext $M = \text{HEDec}(C, \mathbf{sk})$.

Analysis. By splitting the decryption job among the participants, this prevents any adversary from affecting the output of the protocol. However, as a trade-off, the communication complexity of HERB for setup in the first epoch is $\mathcal{O}(n^3)$ because of the DKG protocol. From the second epoch forward, the complexities of HERB are the same as DVRF and threshold signature-based DRNGs due to having the same approach.

We would like to make a comparison between HERB and DVRF: It can be seen that the **Decryption** phase of HERB is almost identical to DVRF. In HERB, participants calculate a "partial" decryption value $D_i = \text{HEDec}(C, \text{sk}_i)$ whereas in DVRF, participants calculate a partial VRF value $Y_i = \text{VRFVal}(X, \text{sk}_i)$. The verification of D_i and Y_i also incur the same computation complexity and both are better than using bilinear maps in threshold BLS signatures-based DRNGs. Then, both protocols combine $t + 1$ valid partial decryption/VRF output via Lagrange interpolation to produce a single output. Hence, we can see that HERB is less efficient than DVRF, because HERB has to additionally execute the **Encryption** phase before the abovementioned **Decryption** phase, thus doubling the communication and computation complexity compared to DVRF.

Remark 3.2.1. Apart from all the advantages and disadvantages above, both protocols share an interesting property: It is possible to replace the existing Elgamal encryption scheme in both protocols with a fully homomorphic encryption scheme to achieve post-quantum security. In the case of HERB, the fully homomorphic encryption scheme [Gen09] supports both the distributed key generation and threshold encryption [CSS19]. Hence, HE-based DRNGs serve as a candidate to achieve post-quantum security in the future.

3.2.6 DRNG from Verifiable Delay Function

These DRNG constructions are based on a VDF described in Section 2.4.13. Recall that a VDF requires a specified number of steps to compute the output, yet the output can be verified efficiently. The fundamental concept behind employing VDF is to mitigate the risk of a withholding attack by the last participant. When the outputs of the random beacon are computed using VDFs, the adversary is unable to manipulate the output of the random beacon, as it cannot compute the output within the given time. However, the output can be quickly verified by an external verifier. Consequently, several protocols employ a VDF scheme $\text{VDF} = (\text{VDFSetup}, \text{VDFEval}, \text{VDFVerify})$ to construct DRNGs that solve the Unbiasability

problem. Protocols that follow this direction are [SJH⁺21, Dra66, WN21, CATB23]. VDF-based DRNGs can be further divided into two sub-approaches: **with trapdoors** [Dra66, WN21] and **without trapdoors** [SJH⁺21, CATB23]. VDF without trapdoors is just our ordinary VDF, while VDF with trapdoors includes additional trapdoor information that allows the trapdoor holder to compute the output quickly.

3.2.6.1 Constructions without Trapdoors

As discussed above, these constructions employ the ordinary VDF scheme VDF that strictly satisfies all the properties in Subsection 2.4.13. The security of these constructions is based on the t -**Sequential** property, i.e., it requires any participants to compute the VDFEval function within $\Omega(t)$ time to provide the output, **with no possible shortcut**. Hence, the constructions in this category incur very high computation complexity as t is large, and may not be suitable for all applications where random values need to be generated frequently. Below we describe the constructions in this category.

1. Minimal VDF Randomness Beacon

Justin Drake [Dra66] introduced a DRNG construction that combines RANDAO and VDF with the aim of mitigating the withholding attack made by the last participant in RANDAO. The protocol’s underlying concept is straightforward: Participants have a specific deadline to jointly execute RANDAO and finish their roles. The biasable seed s generated by RANDAO will serve as the input of the VDF, whose output cannot be predicted before the deadline. The final output will VDF result. With this abovementioned concept, the construction can be succinctly outlined as follows.

Overview of Minimal VDF Randomness Beacon

1. In each epoch, participants must execute the RANDAO protocol within a specific time t_0 , as described in Subsection 3.2.1, to generate a seed X that serves as the input for the VDF computation. If any participant P does not reveal his secret s_i within time t_0 , he will be discarded.
2. The seed X then used to compute the protocol output Ω via VDF as follows.

$$(\Omega, \pi) = \text{VDFEval}(X, \text{pp}, t).$$

where t is the time parameter. The value Ω can be efficiently verified by anyone using the verify algorithm $\text{VDFVerify}(X, Y, \pi, \text{pp}, t)$.

Analysis. Since the VDF computation time is very long for sufficiently large t , the adversary will not be able to know the value Ω of the VDF within time t_0 before he is discarded, thus he does not have the incentive to abort the protocol like RANDAO. The advantage of this protocol is that it inherits the same complexity as RANDAO (except the VDF computation complexity, which is very high) and can also withstand up to $n - 1$ dishonest participants like RANDAO. Nevertheless, the idea of the construction is completely intuition-based and there is no formal security proof given by the author for this protocol. This absence of formal proof makes it challenging to assess its security in comparison to other constructions.

2. RANDCHAIN

RANDCHAIN [HLY20] is a competitive DRNG where each participant competes to be the leader responsible for producing the protocol output. RANDCHAIN is a “priority selection” type DRNG and its workflow mimics proof-of-work, except that the puzzle of proof-of-work is replaced with a puzzle involving a VDF. RANDCHAIN employs the VDF construction of Sloth [LW15] to construct this new type of puzzle. At a high level, the puzzle is follow: Given values X and **target**, find a positive integer i such that

$$X_i < \text{target} \text{ and } (X_i, \pi_i) = \text{VDFEval}(X_{i-1}, \text{pp}, t) \forall i \geq 0$$

where $X_0 = X$. For ease of notation, we let

$$(X_i, \pi_i) \leftarrow \text{VDFEval}^i(X, \text{pp}, t)$$

to denote the above sequence of computation. The specific Sloth's VDF construction has a special property: If $(X_i, \pi_i) \leftarrow \text{VDFEval}^i(X, \text{pp}, t)$ then $\text{VDFVerify}(X, X_i, \pi_i, \text{pp}, t \cdot i) = 1 \ \forall i \in \mathbb{N}$, and the converse also holds, indicating that X_i is computed by applying VDF i times to X . Finally, when a participant who has solved the puzzle proposes a Block B , the output Ω is calculated using the VDF evaluation function, the output of which is determined by B . With the discussion above, RANDCHAIN's construction is described as follows.

Overview of RANDCHAIN

1. Each Block B in RANDCHAIN's blockchain has the following format:

$$B = (h^{-1}, h, t, S, \text{pk}, \pi)$$

where $B.h = \text{H}(B.\text{pk}||B.S)$ and $B.h^{-1}$ is the hash previous Block.

2. Let r be the current epoch the current Block be B_{r-1} . Each participant P_i finds a positive integer j such that

$$Y_{ij} < 2^n/t \text{ and } (Y_{ij}, \pi_{ij}) = \text{VDFEval}^j(B_{r-1}.h, \text{pp}_i, t).$$

Participant P_i then broadcasts (j, Y_{ij}, π_{ij}) confirming that he solved the puzzle.

3. Other participants can check the validity of the solution Y_{ij} broadcasted by P_i using $\text{VDFVerify}(B_{r-1}.h, Y_{ij}, \pi_{ij}, \text{pp}_i, j \cdot t)$ of Sloth and check if $Y_{ij} < 2^n/t$. The first participant to produce a valid solution to the puzzle gets the right to propose a Block B_r . Finally, the output of the protocol is computed as

$$\Omega = \text{VDFEval}(B_r.\text{pk}||B_r.S, \text{pp}, t).$$

Analysis. Since RANDCHAIN mimics the workflow of proof-of-work, it also inherits the same communication and computation complexity as proof-of-work. More specifically, it

also relies on the same assumption that the probability of an adversary solving the VDF puzzle before honest participants is less than $1/2$, which may not hold true if the adversary has significantly enhanced computational power. In addition, it also suffers from the secret withholding attack, where participants with significantly improved computational power may find multiple solutions to the VDF puzzles before honest participants and submit one of them, or not submit at all if it does not benefit them.

RANDCHAIN also claims that their random output each epoch is **uniformly distributed**. However, this claim is unlikely to be true, given that the output Ω_r is computed as $\Omega_r = \text{VDFEval}(B_r.X, B_r.pp, t)$ and the Block B' is computed from the hash value of the previous Block. None of these functions are even proven to achieve Pseudorandomness. Even if we view the hash function as a random oracle model, to make B look uniform, Ω_r is still not uniformly distributed, since VDF outputs are not proved to achieve Pseudo-randomness. However, RANDCHAIN claims that the uniform distribution of Ω is due to the result of computing these functions. Hence we conclude that the claim of uniformly distributed output in RANDCHAIN is not true.

3.2.6.2 Construction with Trapdoors

These constructions employ a **trapdoor verifiable delay function** (trapdoor VDF) was proposed by Wesolowski [Wes19]. Much like an ordinary VDF, trapdoor VDF allows anyone to compute the VDF given the input. However, it includes an additional algorithm TVDFTrapdoorEval that takes an extra input known as a trapdoor (or a shortcut) sk , enabling those in possession of sk to compute the output significantly faster.

We delve into the details of Wesolowski's construction to see how is it possible to construct such a trapdoor. At its core, Wesolowski's setup introduces a group \mathbb{G} with an unknown order with a generator g . One might ask if such a group \mathbb{G} exists, and the answer is affirmative. For two primes p, q let $N = p \cdot q$ and consider the group

$$\mathbb{Z}_N^* = \{x \in \{0, 1, \dots, N-1\} \mid \gcd(x, N) = 1\}$$

It is well known that the order of \mathbb{Z}_N^* , i.e., the number of its element is equal to $(p-1)(q-1)$. If the factorization of N is unknown, for example, when p and q are large and kept confidential, then the order of \mathbb{Z}_N^* is not known by anyone. Hence, the group \mathbb{Z}_N^* is actually a group of

unknown order when the factorization of $N = pq$ is unknown. Now that we have specified a group \mathbb{G} whose order is unknown, we proceed to explore how a trapdoor is established within Wesolowski’s construction. To highlight the “trapdoor” property in Wesolowski’s construction, we describe the difference between the trapdoor function `TVDFTrapdoorEval` and the standard evaluation function `TVDFEval` in Wesolowski’s construction as follows.

Overview of Wesolowski’s Trapdoor VDF

1. `TVDFTrapdoorEval`($X, \text{sk}, \text{pp}, t$): The public parameter pp is a group \mathbb{G} with unknown order. The secret key sk is the order of \mathbb{G} , i.e $\text{sk} = |\mathbb{G}|$. The value X is hashed to a generator g of \mathbb{G} , the owner then compute

$$s = 2^t \pmod{\text{sk}}$$

then output $Y = g^s$. Computing Y requires $\mathcal{O}(\log t)$ steps, dominated by computing s .

2. `TVDFEval`(X, pp, t): Without the knowledge of $|\mathbb{G}|$, the value $Y = g^{2^t}$ can be computed after t steps by initializing $Y = g$ and set $Y := Y^2$ for each step.

The common idea of DRNGs using trapdoor VDF is that: If all participants are honest, then the output of the protocol can be calculated via the `TVDFTrapdoorEval` function, otherwise, the output can still be reconstructed via the `TVDFEval` functions. To see the mechanism behind this idea, we will briefly describe the DRNG constructions in this category.

1. RandRunner

In RandRunner’s construction [SJH⁺21], each participant possess their own trapdoor VDF of Wesolowski described above and keeps their own trapdoor in secret. It follows the fundamental concept of the trapdoor VDF constructions mentioned earlier: In each epoch, a leader is selected and uses his own `TVDFTrapdoorEval` function to evaluate the output of the epoch. In the event that the leader chooses to withhold it, the remaining participants can still reconstruct the output using `TVDFEval`. Note that the reconstruction process is significantly more time-consuming due to not having the trapdoor. Similar to HyDrand, it is assumed that each participant P is expected to become a leader once every n epochs. With this concept, RandRunner can be concisely summarized as follows.

Overview of RandRunner

1. The leader P_ℓ , with his trapdoor \mathbf{sk}_ℓ then uses the input X_r of the r -th epoch and the trapdoor VDF TDVF to **quickly** compute the VDF value

$$(Y_{r\ell}, \pi_{r\ell}) = \text{TVDFTrapdoorEval}(X_r, \mathbf{sk}_\ell, \mathbf{pp}_\ell, t).$$

2. In the event that the leader P_ℓ does not submit the output, the remaining participants proceed to reconstruct the VDF value $Y_{r\ell}$ by computing

$$(Y_{r\ell}, \pi_{r\ell}) = \text{TVDFEval}(X_r, \mathbf{pp}_\ell, t).$$

Note that $Y_{r\ell}$ will be computed with a **slower** time due to not having the trapdoor.

3. After $Y_{r\ell}$ is published, the output Ω_r and the next input X_{r+1} is calculated as follows.

$$\Omega_r = \text{H}(Y_{r\ell}) \text{ and } X_{r+1} = \text{H}(\Omega_r).$$

Finally, given the output Ω_r of the current epoch, the leader P_ℓ for the $(r+1)$ -th epoch of RandRunner will be deterministically determined by the formula $\ell = \Omega_r \pmod{n} + 1$.

Analysis. Similar to DVRF and threshold BLS signature, RandRunner's output is deterministic, preventing an adversary from biasing the output. Liveness is guaranteed as the VDF output can be computed by the TVDFEval by a single honest participant. The verification complexity of RandRunner is only $\mathcal{O}(1)$ in the best case, since only the leader's VDF output is verified. In addition, the communication complexity of RandRunner is $\mathcal{O}(n)$ in the best case, where an honest leader broadcasts his VDF output to other participants. In the worst case, the communication complexity becomes $\mathcal{O}(n^2)$ since $n - 1$ remaining participants have to compute the VDF output locally and each has to broadcast the output to other participants, then use Byzantine agreement protocols to reach consensus on the output.

However, despite many advantages above, RandRunner does suffer from a significant drawback. In an epoch, if the leader P_ℓ quickly computes the VDF output but chooses not to submit it, an honest participant would be required to compute the output at a much slower

time while P_ℓ has known the output already. This implies that a dishonest leader could trivially learn the result of the DRNG significantly faster than other participants and may secretly reveal it outside to benefit himself before the output is publicly known.

2. Bicorn

Bicorn [CATB23] follows the three-step idea of RANDAO and additionally employs the VDF construction of Wesolowski [Wes19] to construct a DRNG. In Bicorn, participants also commit their secret, then reveal it and combine all the secrets to get the output. However, unlike RANDAO, Bicorn additionally has a reconstruction step that allows participants to reconstruct the output when participants do not reveal their secret. To see how Bicorn reconstructs the output using VDF, we describe the details of Bicorn as follows.

Overview of Bicorn

1. For setup, Bicorn initially chooses a group \mathbb{G} with unknown order, a time parameter t , a value X and compute

$$(g, h = g^{2^t}, \pi) = \text{TVDFTrapdoorEval}(X, \text{sk} = |\mathbb{G}|, \text{pp}, t).$$

2. In each epoch, each participant P_i chooses a random value s_i and publish the commitment C_i of s_i by quickly computing $C_i := g^{s_i}$.
3. When all C_i are broadcasted, each participant P_i then reveals s_i . Other participants can verify the correct of s_i by checking $g^{s_i} \stackrel{?}{=} C_i$.
4. If all checks are correct, the final output Ω can be **quickly** computed by anyone as

$$\Omega = \prod_{i=1}^n h^{s_i}.$$

5. If any participant withholds s_i , then h^{s_i} can be recovered by any honest participant at a much **slower** time by computing $h^{s_i} = C_i^{2^t} = g^{s_i 2^t}$, hence Ω can still be reconstructed.

Analysis. One can see that Steps 2, 3, and 4 in Bicorn closely resemble steps 1, 2, and 3 of RANDAO, as outlined in Section 3.2.1. However, in the final step, Bicorn offers the advantage of being able to reconstruct the output in Step 5 albeit with a slower time. This crucial feature means that Bicorn is not susceptible to withholding attacks, in contrast to RANDAO. Moreover, the complexities of Bicorn mirror those of RANDAO, with the exception of Step 5, owing to their shared structural framework.

Nonetheless, it is worth noting that Bicorn shares the same limitation as RandRunner. In any given epoch, the last participant P_n , after observing the revealed secrets of other participants, can **quickly** calculate the final output $\prod_{i=1}^n h^{s_i}$ and choose not to reveal their secret s_n . In this scenario, the honest participants have to reconstruct the final output by **slowly** calculating $h^{s_n} = C_n^{2^t}$, which requires t sequential steps. This implies that the adversary could acquire knowledge of the final output significantly faster than the honest participants and exploit it to their advantage before it is publicly disclosed.

3.3 Discussion

In this section, we discuss several aspects of existing DRNGs. These aspects provide deeper insights to enhance our comprehension of existing DRNG constructions. They also provide a metric to make a comparison between DRNG constructions in Section 3.4 and help us in specifying the suitable construction for blockchain-based applications in Chapter 4.

3.3.1 Advantages and Disadvantages of DRNG Approaches

Recall that in Section 3.2, we have presented an overview of existing DRNG constructions based on their cryptographic primitives. All of them have upsides and downsides and none of them completely dominates another. Some produce full security properties but suffer from high complexity and vice versa. To facilitate a more comprehensive understanding of these approaches, we create Table 3.1 to summarize the advantages and disadvantages of each cryptographic primitive approach.

Table 3.1: *A summary of existing DRNG approaches*

Primitives	Advantages	Disadvantages
Hash-based DRNGs	These DRNGs have simple construction and enjoy the lowest possible communication and verification complexity.	All Hash-based DRNGs do not provide Unbiasability. More specifically, they suffer from withholding attacks.
PVSS based DRNGs	<ol style="list-style-type: none"> 1. All PVSS-based DRNGs provide uniformly distributed randomness output in each epoch. 2. Non leader-based PVSS achieves full security properties. 	<ol style="list-style-type: none"> 1. Non leader-based PVSS suffer from high communication and computation complexity. 2. Leader-based PVSS are vulnerable to adaptive adversaries.
VRF-based DRNGs	<ol style="list-style-type: none"> 1. All of these DRNGs do not have any trusted setup and achieve Unpredictability. 2. These DRNGs incur less computation, communication and verification complexity. 	Some DRNGs, namely [GHM ⁺ 17, DGKR18] follows the priority selection style of PoW and do not provide Unbiasability.
HE-based DRNGs	<ol style="list-style-type: none"> 1. Partial homomorphic encryption schemes used in these DRNGss can be replaced by a lattice-based fully homomorphic scheme to ensure post-quantum security. 2. All HE-based DRNGs provide uniformly distributed randomness output in each epoch. 	HE-based DRNGs have trade-off between security and efficiency (the construction of [NNL ⁺ 19] provides low communication complexity, but does not provide Unbiasability, while the construction of [CSS19] provide full security properties but suffers from high communication complexity.)
VDF-based DRNGs	<ol style="list-style-type: none"> 1. VDF-based DRNGs incur low communication and verification complexity. 2. Most of these DRNGs provide Unbiasability as long as there is at least one honest participant. 	<ol style="list-style-type: none"> 1. VDF-based DRNGs require high computation complexity due to using VDFs. This makes them not suitable for certain applications where randomness needs to be provided quickly. 2. In trapdoor VDF-based DRNGs, the adversary can learn the randomness output of the current epoch significantly faster than the honest participants.
Thres. Sig. based DRNGs	These DRNGs achieve full security properties while enjoying low communication and computation complexity.	<ol style="list-style-type: none"> 1. These DRNGs require either a trusted setup or DKG, hence do not offer a reconfiguration-friendly setup. 2. Security of the DRNGs depends on the security assumptions of elliptic curve pairings due to the use of BLS-signature.

3.3.2 Complexity Analysis

The efficiency of a DRNG is determined by its communication, computation and verification complexity. DRNGs with different approaches exhibit different complexity. Constructing a DRNG protocol with a balanced communication, computation and verification complexity has always been a challenging task. Hence, an extensive amount of work has been devoted to reducing the complexity of these DRNG protocols. Here we present an overview of the three abovementioned complexities of existing DRNG protocols.

- **Communication Complexity.** It is defined as the total number of bits that all participants need to send in every epoch. So far, with the exception of [SJK⁺17], almost no DRNG have been able to achieve less than $\mathcal{O}(n^2)$ communication complexity. This is because in all DRNGs, each participant has to send a certain common message M to $n - 1$ other participants use mechanisms such as Byzantine agreement protocols so that they can reach consensus on M . Some protocols [CD17, CD20, SJK⁺17, NNL⁺19, GLOW21] instead assume a broadcast channel where a participant can simply broadcast M on such a channel. However, a broadcast channel, when actually implemented using a distributed protocol, has a communication lower bound of $\Omega(n^2)$ [DR85]. On the other hand, by using point-to-point channel and achieving consensus via Byzantine agreement protocols, note that each participant has to send $\mathcal{O}(n)$ messages, assuming each message consists of 1 bit. Since there are $\mathcal{O}(n)$ participants, a total of $\mathcal{O}(n^2)$ messages are sent. In addition, most Byzantine agreement protocols also require $\mathcal{O}(n^2)$ communication complexity [CL02], hence the optimal cost $\mathcal{O}(n^2)$ is possible to reach this way. Thus, we see that the communication complexity when executing most DRNG protocols is always at least $\Omega(n^2)$ regardless of broadcast channel or point-to-point channel.

Now we shall analyze the communication complexity of DRNG constructions, assuming that participants using a network with only secure point-to-point channel by default. Non leader-based DRNGs, specifically RandShare and SCRAPE, incur a communication complexity of $\mathcal{O}(n^3)$ because in the PVSS protocol outlined in [Sta96], each participant must 'broadcast' $\mathcal{O}(n)$ encrypted shares, resulting in $\mathcal{O}(n)$ messages sent to each of the $n - 1$ remaining participants. HyDrand aims to reduce the communication complexity to $\mathcal{O}(n^2)$ by adopting a leader-based approach where only the leader has to perform the PVSS share distribution. RandHound, RandHerd divides a smaller set of participants of size c ,

where each group works independently and only communicates to the requester/leader, and hence, achieves communication less than $\mathcal{O}(n^2)$. However, such a procedure can be immediately subject to attacks by an adaptive adversary who can corrupt the committee once it is determined. Most leader-based PVSS, VRF and VDF and threshold signature-based DRNGs achieve the optimal $\mathcal{O}(n^2)$ communication complexity.

We would like to note that the case of RANDCHAIN and proof-of-work actually have worst-case communication complexity equal to $\mathcal{O}(n^2)$. In these protocols, recall that only the leader/highest priority participant, say P has to broadcast his messages to $n - 1$ other participants, one would expect these protocols to have $\mathcal{O}(n)$ communication complexity. However, if P provides a false computation result, then we have to wait until the next participant broadcasts his messages. Hence, in the worst case, the total communication complexity is $\mathcal{O}(tn)$, since we have t dishonest participants. Because we assume $t = \Omega(n)$, we conclude that in the worst case, the communication complexity of these protocols is $\mathcal{O}(n^2)$.

- **Computation Complexity per Node.** It is defined as the number of operations needed to be performed by a participant during one epoch of the DRNG protocol. Non leader PVSS-based DRNGs, namely SCRAPE, RandShare and ALBATROSS, incur a high computation complexity, from $\mathcal{O}(n^3)$ or $\mathcal{O}(n^2 \log n)$. Leader-based PVSS protocols [SJSW20, SJK⁺17] manage to reduce the computation complexity down to $\mathcal{O}(n \log^2 n)$, dominated by the cost of **optimally** computing $\mathcal{O}(n)$ Lagrange coefficients in a single PVSS. VDF-based DRNGs also suffer from high computation complexity due to the **Sequential** property of a VDF. On the other hand, VRF, threshold signature and hash-based protocols enjoy minimum computation complexity, from $\mathcal{O}(n \log^2 n)$ [HMW18, Drave, GLOW21] to $\mathcal{O}(1)$ [GHM⁺17, DGKR18].
- **Verification Complexity per Verifier.** Verification complexity refers to the number of operations performed by an external participant to verify the output of a beacon protocol. Non leader-based PVSS protocols [CD17, SJK⁺17] suffer from quadratic verification complexity since a verifier has to verify the validity of n^2 share broadcasted by n participants. Although VDF-based DRNGs incur high computational complexity, they do provide efficient verification and hence incur much less verification complexity. The most efficient DRNGs with regard to verification complexity are mostly based on

VRF with “priority selection”, VDF, and Hashing, namely [GHM⁺17, DGKR18, Nak, SJH⁺21, WN21, CATB23] with only $\mathcal{O}(1)$ verification complexity.

3.3.3 Setup Assumptions

Many DRNG protocols [SJK⁺17, CKS05, CD17, GLOW21, CSS19, Drave, NNL⁺19] require an initial **trusted setup** assumption, wherein private keys for the participants and uniformly random public parameters are generated either by a trusted third party (dealer) or through a distributed key generation (DKG) protocol. The security of a DRNG with a trusted third party crucially depends on the actions and capabilities of this trusted party. This assumption is quite strong, as the entire security of the protocol hinges on this single trusted entity. Note that all DRNGs with client-server architecture fall into the trusted setup category since they require the client to generate the keys or dividing the client into groups. In contrast, DKG does not necessitate a trusted setup and thus provides enhanced security. However, the DKG protocol comes with a considerable setup cost, involving high communication complexity. Additionally, using DKG restricts the ability to replace participants. When a participant needs to be replaced, the costly DKG process must be repeated. Therefore, DKG-based DRNGs are favored when the participants are static. Consequently, some DRNGs [DKIR22, CD17, DGKR18, GHM⁺17] choose neither of the aforementioned approaches. Instead, they construct a protocol with a **transparent setup**, in which the public parameters are trapdoor-free. A protocol with a transparent setup does not rely on any trusted parties, while still allowing for participants to be replaced frequently. This makes such protocols suitable for scenarios where participants need to be changed frequently.

3.3.4 Adversarial and Network Model

As mentioned in Subsubsection 2.4.2.1, an adversary in a protocol can be classified into three categories: static, adaptive, or proactive. However, only static and adaptive adversaries have been considered in DRNG protocols so far. Most DRNGs assume a static adversary, while some [HLY20, DGKR18, SJH⁺21] assume an adaptive adversary.

An adversary can impact the security guarantees of DRNG systems in various ways, as described in Subsubsection 2.4.2.1. Different protocol approaches are vulnerable to distinct types of attacks, outlined as follows:

- With the exception of SPURT, all leader-based DRNGs [SJK⁺17, SJSW20, SJH⁺21, BSL⁺21] are insecure against adaptive adversaries because they can corrupt the leader as soon as they are selected. During the first t epochs, an adaptive adversary can corrupt up to t leaders. When the leader is corrupted, then Unbiasability and Liveness properties in these protocols are not guaranteed.
- Most leader-based, priority selection based DRNGs [Nak, Ran17, GHM⁺17, WN21, DGKR18] also suffer from withholding attacks. Hence the Unbiasability and Liveness properties in these protocols are also not guaranteed.
- DRNG using client-server architectures [NNL⁺19, SJK⁺17] are outright insecure against a dishonest client who easily deviates from the protocol.
- In all existing VDF-based DRNGs, the outputs of these DRNGs can be learned significantly faster by the adversary with improved computational power.

Most of the PVSS, threshold signature, and VRF and VDF-based DRNGs require all messages of honest participants to be delivered on time to successfully produce outputs and achieve perfect Liveness. As discussed in Subsubsection 2.4.2.2, this prerequisite can be fulfilled by assuming a synchronous network in these protocols. While the assumption of a synchronous network simplifies these protocols by allowing their Liveness property to be directly derived from it, such a network is very hard to achieve in real-world scenarios. Certain DRNGs, specifically [DKIR22, HMW18] employ additional mechanisms that allow them to achieve Liveness in a partial synchronous network. Finally, only RandShare has assumed an asynchronous network so far, however the Liveness property of RandShare will be lost in such a network.

3.4 Summary of DRNGs

We create Table 3.3 to summarize several aspects of the approaches mentioned above. Regarding Section 3.3, the table focuses on the following aspects:

1. *Pseudo-randomness*: The outputs in any epoch of the protocol are indistinguishable from random. This is the most essential property of a DRNG protocol. A DRNG must be designed so that one does not see any intelligible pattern in the outputs of the protocol.

Otherwise, he might be able to guess the future outputs of the protocol based on the outputs of the previous epochs.

2. *Unpredictability*: This is a weaker version of the Pseudorandomness property but still the second most important property for a DRNG. The output of the protocol cannot be guessed correctly by the adversary until it is revealed, but it may contain some bias. For example, the adversary may be able to guess the last bit of the protocol output but cannot guess exactly the output.
3. *Unbiasability*: This is the third most crucial property of a DRNG. It means that no adversary can affect the output of the protocol. If an adversary can affect the output of the protocol, he can force the output into values that benefit him.
4. *Liveless*: No adversary can force the protocol to delay or abort. A protocol needs to ensure that it can always produce outputs properly to serve its purpose for the system.
5. *Public Verifiability*: Everyone can verify the output of the protocol. A DRNG protocol needs to prove that it produces correct output to the public so that it can gain the trust of the public.
6. *Communication Complexity*: The cost of bitwise point-to-point communication between participants. If the communication complexity is low, the protocol can be used for more participants, hence increasing its fairness.
7. *Computation Complexity per Node*: The number of computation steps per node or participant. The lower the computation complexity, the more efficient the protocol provides, and thus the protocol can also be used for more participants.
8. *Verification Complexity per Verifier*: The number of verification steps per public verifier. Once an output of a DRNG is announced publicly, we would like it to be verified as quickly as possible.
9. *Honest Nodes*: The minimal number of honest nodes assumed in each protocol. Because there is no way to know withstand a participant will follow the protocol, we would like a DRNG protocol that can withstand a lot of dishonest participants but can still produce pseudo-random outputs.

10. *Adaptive Adversary*: Check whether the protocol is secure against an adaptive adversary or not. It is more desirable for a protocol to be secure against an adaptive adversary.
11. *Network Model*: The assumption of the network in the DRNG. It is preferable if a DRNG can handle a partial synchronous network or an asynchronous network.
12. *Setup Assumption*: The type of setup in a DRNG. Setup using DKG is more preferable than trusted setup, and trans. setup is the most preferred setup type in a DRNG, since it does not have any limitation.
13. *Primitive*: The cryptographic primitive used in the protocol. Every protocol uses a different cryptographic primitive. We mention the cryptographic primitive in each protocol to see the strengths and weaknesses of each cryptographic primitive in that protocol.

Based on Table 3.3, we have some following remarks:

1. Some DRNG constructions do not specify every aspect in the table. We label these unspecified aspects “unknown”.
2. RandShare assumes a asynchronous model. However, in this model, the Liveness property of RandShare is lost.
3. Even though Nguyen *et al.* ’s protocol does not provide security guarantee against $n - 1$ dishonest participants, as discussed in Subsection 3.2.5, we still follow their honest minority assumption in the table.
4. The computation complexity of a VDF is very high in VDF-based DRNGs. Thus we cannot compare the computation complexity of this approach to others. The same reason also applies to proof-of-work.
5. RANDCHAIN and proof-of-work assume that malicious participants can solve the puzzle before honest participants with probability less than 0.5. This is a very strong assumption, hence we denote $\checkmark^?$ to raise our concern whether these properties can be accepted using the assumption.
6. In HyDrand and other leader-based DRNGs, the **unpredictability** property is only achieved from epoch $t + 1$ forward, where t is the maximal number of dishonest participants.

7. The constant c in RandHound and RandHerd is actually depend on n . Larger c make the protocols incur high cost, while smaller c make the protocols more likely to be attacked by adaptive adversaries
8. In (t, n) -threshold-based protocols, such as HERB, DVRF, threshold signature-based DRNGs, and also in some leader-based PVSS protocols like HyDrand, the computation complexity per node is dominated by $\mathcal{O}(n \log^2 n)$, the cost of optimally computing Lagrange coefficients in each epoch in the worst case. The reason for this cost will be explained in Subsection 4.5.3. The same also applies to SCRAPE, where its computation cost is dominated by $\mathcal{O}(n^2 \log^2 n)$ due to computing Lagrange coefficients for recovering $\mathcal{O}(n)$ secrets, each require $\mathcal{O}(n \log^2 n)$ computation cost.

Table 3.3: *A comparison of existing DRNGs.*

	Pseudorandomness	Unpredictability	Unbiasability	Liveness	Public Verifiability	Comm. Complexity	Comp. Complexity / Node	Verf. Complexity / Verf.	Honest Nodes	Adaptive Adversary	Network Model	Setup Assumption	Primitives
RANDAO	✓	✓	✗	✗	✓	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	1	✗	sync.	trans.	Hash
RandShare	✓	✓	✓	✓	✓	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	$2n/3$	✗	async.	trans.	PVSS
SCRAPE	✓	✓	✓	✓	✓	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2 \log^2 n)$	$\mathcal{O}(n^2 \log^2 n)$	$2n/3$	✗	sync.	trans.	PVSS
RandHound	✓	✓	✗	✓	✓	$\mathcal{O}(c^2 n)$	$\mathcal{O}(c^2 n)$	$\mathcal{O}(c^2 n)$	$2n/3$	✗	sync.	trusted	PVSS
RandHerd	✓	✓	✓	✓	✓	$\mathcal{O}(c^2 n)$	$\mathcal{O}(n/c)$	$\mathcal{O}(1)$	$2n/3$	✗	sync	trans.	PVSS + Schnorr Sig
ALBATROSS	✓	✓	✓	✓	✓	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(1)$	$n/2$	✗	sync.	trans.	PVSS
SPURT	✓	✓	✓	✓	✓	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$2n/3$	✗	partial sync.	trans.	PVSS+pairing
HyDrand	✗	✓	✓	✓	✓	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(n \log^2 n)$	$2n/3$	✗	sync.	trans.	PVSS
Algorand	✗	✓	✗	✓	✓	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$2n/3$	✗	sync.	trans.	VRF
Ouroboros Praos	✗	✓	✗	✓	✓	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$n/2$	✓	sync.	trans.	VRF
Nguyen <i>et al.</i>	✗	✓	✗	✓	✓	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	1	✗	unknown	trusted	VRF + Hom Enc
HERB	✓	✓	✓	✓	✓	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(n \log^2 n)$	$2n/3$	✓	sync.	DKG	Hom Enc
RandRunner	✗	✓	✓	✓	✓	$\mathcal{O}(n^2)$	very high	$\mathcal{O}(1)$	$n/2$	✗	sync	trans.	VDF
Minimal VDF	✗	✓	✗	✓	✓	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$n/2$	✗	unknown	trans.	VDF+Hash
RANDCHAIN	✗	✓?	✗	✗	✓	$\mathcal{O}(n^2)$	very high	$\mathcal{O}(1)$	$n/2$	✓?	sync	trans.	VDF
Bicorn	✗	✓	✓	✓	✓	$\mathcal{O}(n^2)$	very high	$\mathcal{O}(n)$	$n/2$	✗	sync	trans.	VDF
DVRF	✓	✓	✓	✓	✓	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(n \log^2 n)$	$n/2$	✗	sync.	DKG	VRF
Cachin <i>et al.</i>	✗	✓	✓	✓	✓	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(n \log^2 n)$	$n/2$	✗	sync.	trusted	BLS Signature
Drand	✗	✓	✓	✓	✓	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(1)$	$n/2$	✗	sync.	DKG	BLS Signature
DFINITY	✗	✓	✓	✓	✓	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(1)$	$2n/3$	✗	partial sync.	DKG	BLS Signature

Chapter 4

Distributed Verifiable Random Function Protocol

In this chapter, we will specify our choice of DRNG construction suitable for blockchain-based application and formally describe the protocol. The chapter has three main parts. First, we present our research and the decision to use the DVRF protocol based on the results of our systematic literature review in Chapter 3. Then, we formally describe the DVRF protocol and analyze its security and efficiency. Finally, we propose a method to reduce the computation complexity of the DVRF protocol.

4.1 Why DVRF Protocol

As specified in Chapter 1, we are making a survey of existing DRNG constructions to find a suitable DRNG for blockchain-based applications. Recall that in Section 3.2, we have made a systematic literature review of existing DRNG constructions and then discussed and compared them Sections 3.3 and 3.4, which now helps us in specifying the most suitable DRNG choice for blockchain-based applications. Based on the discussions in Section 3.3 and the comparison result in Table 3.3, we made the following decision:

- Non-leader based PVSS based DRNGs [SJK⁺17, CD17, CD20] suffer from high communication and communication complexity, and are thus too costly for blockchain-based applications. On the other hand, leader-based PVSS [DKIR22, SJK⁺17,

[SJSW20](#)] are vulnerable to adaptive adversaries, hence we do not follow this approach.

- We immediately exclude the approach of hash-based DRNGs [[Nak, Ran17](#)] since they do not provide bias resistance, i.e, they suffer from withholding attack. A client will not place their trust in an application that does not provide sufficient security.
- Some VRF-based DRNGs, namely Algorand [[GHM⁺17](#)] and Ouroboros Praos [[DGKR18](#)] and the HE-based construction of Nguyen *et al.* [[NNL⁺19](#)], although enjoy low cost, do not provide bias resistance. Hence, we exclude these protocols.
- VDF-based DRNGs [[WN21](#), [CATB23](#), [SJH⁺21](#)] require a specific time to produce the outputs, hence they have very high computation cost. This makes these protocols not suitable for applications where randomness needs to be produced quickly and frequently, say blockchain-based games, since the prizes and results come frequently for players.
- Threshold signature-based DRNGs [[CKS05](#), [Drave](#), [HMW18](#)], HERB [[CSS19](#)] and DVRF [[GLOW21](#)] are more balanced. They achieve full security properties and provide less communication and computation complexity, making them seem to be the most suitable choice. However, among them, signature-based DRNGs require more complicated techniques like bilinear pairing, and their verification algorithm is also 2 to 4 times slower than the DVRF protocol [[GLOW21](#)]. In addition, we have already compared the performance between HERB and DRVF in Subsection [3.2.5](#), and apparently HERB is worse than DVRF in performance.

This leaves the opinion of the DVRF protocol of Galindo *et al.* [[GLOW21](#)]. The protocol achieves full security properties while enjoying low communication and computation complexity. As we have analyzed, the protocol is more efficient than HERB and threshold-based DRNGs, with a similar approach and design. The DVRF protocol requires a DKG for setup, making it inefficient when the participants need to be replaced frequently. However, to our knowledge, blockchain-based applications do not require replacing participants frequently. As we will analyze later, the DKG protocol has to be re-run once every r_{rerun} rounds in theory for a parameter r_{rerun} . However, the parameter r_{rerun} is large, and hence, we do not need to rerun it very often. In addition, while the DVRF construction uses the DKG construction of [[GJKR99](#)], which is only proven to be secure against static adversaries, it is possible to replace the DKG of Gennaro *et al.* with the one in [[CGJ⁺99](#)] to achieve security against adaptive adversaries.

However, since no attacks have by adaptive adversaries been made against the DKG protocol of Gennaro *et al.* both in theory and practical, we could still use it to achieve lower cost.

Hence, we decide to use the DVRF protocol of Galindo *et al.* above and deploy it for blockchain-based applications.

4.2 Protocol Overview

Now we are ready to present the high level of the DVRF protocol of [GLOW21]. The protocol consists of two main components: **distributed key generation** (DKG) and **verifiable random function** (VRF). The explicit construction of DKG and VRF will be described in Sections 4.3 and 4.4, respectively. Now, the protocol is described informally below:

- In the DRNGSetup protocol, all participants execute the DKG protocol of Gennaro *et al.* in Section 4.3 to select a list of qualified participants and generate a pair of public-secret key $(\mathbf{pk}, \mathbf{sk})$. In addition, each qualified participant P_i receives a pair of partial secret key \mathbf{sk}_i and public key \mathbf{pk}_i . The partial secret key \mathbf{sk}_i of participant P_i will be used to compute his output Y_i using VRF Eval, while the partial public key \mathbf{pk}_i is used to verify the output of P_i using VRF Verify. The DRNGSetup protocol will be rerun once every $2^{r_{\text{rerun}}}$ epochs for a parameter r_{rerun} depending on λ .
- In each epoch r , to jointly produce a pseudo-random output, each qualified participant executes the DRNGGen protocol, which consists of the following steps:
 1. Each participant P_i use VRF Eval in Section 4.4 to calculate an output Y_{ri} and its proof π_{ri} . Then he broadcasts (Y_{ri}, π_{ri}) to other participants.
 2. When P_i broadcasted his output Y_{ri} , other participants use the public key \mathbf{pk}_i of P_i and run the VRF Verify algorithm in Section 4.4 to verify the correctness of Y_{ri} .
 3. When there are at least $t+1$ valid outputs from participants, a combination algorithm is used to combine these outputs to produce an output Ω_r and its proof π_r . The combination algorithm initially incur $\mathcal{O}(n^2)$ computation complexity, however the complexity can be further reduced to $\mathcal{O}(n \log^2 n)$ in Subsubsection 4.5.3.
 4. Finally, the output Ω_r will be used to compute the input X_{r+1} for the next epoch.

- An external verifier, who wishes to verify the correctness of (Ω_r, π_r) , can use the **DRNGVerify** algorithm using the proof π_r and the public keys \mathbf{pk}_i of the participants.

A more formal and explicit description of the DVRF protocol will be described in Section 4.5. In Section 4.6, we will formally prove that the DVRF protocol achieves Pseudo-randomness, Liveness, Unbiasability, and Public Verifiability. Finally, the protocol enjoys quasi-linear computation and verification complexity, for which we will analyze in Section 4.7.

4.3 Distributed Key Generation of Gennaro

The DKG protocol of Gennaro *et al.* was described in their paper in 1999 [GJKR99], which is based on the DKG protocol of Pedersen. To see the intuition behind the DKG construction of Gennaro *et al.*, we first briefly describe the DKG construction of Pedersen. At a high level, the main steps of Pedersen’s DKG are as follows:

1. Each participant P_i with secret $s_i \in \mathbb{Z}_p$ executes the **VSSShare** protocol of Feldman VSS (see **Protocol 5**) to distribute s_i to other participants.
2. Each participant P_j uses **VSSShareVerify** to verify the correctness the share of s_{ij} he received from P_i . After this, a set **QUAL** of qualified participants is determined.
3. For each participant P_j , the secret key \mathbf{sk}_j is the sum of all the shares he received, i.e., $\mathbf{sk}_j = \sum_{i \in \mathbf{QUAL}} s_{ij}$, where s_{ij} is the j -th share of P_i . The secret key \mathbf{sk} is computed via Lagrange interpolation from the secret keys \mathbf{sk}_j . The i -th public key \mathbf{pk}_i and grand public key \mathbf{pk} is equal to $g^{\mathbf{sk}_i}$ and $g^{\mathbf{sk}}$ respectively.

However, in [GJKR99, Section 3], Gennaro *et al.* showed that the secret key \mathbf{sk} of Pedersen’s DKG can be biased by constructing a strategy for two corrupted participants P_1 and P_2 who collaborate to make the final bit of the public key \mathbf{pk} equal to 0 with high probability before the set **QUAL** is selected. Fortunately, Gennaro *et al.*’s DKG also addresses this issue by replacing Feldman’s VSS in Step 1 with Pedersen’s VSS (see **Protocol 6**), which effectively conceals the public key \mathbf{pk} before the set **QUAL** is selected. After the set **QUAL** is selected, it can be proven that the public key \mathbf{pk} and partial public keys \mathbf{pk}_i become fixed and cannot be biased anymore. In addition, the secret key \mathbf{sk} and each partial secret key \mathbf{sk}_i of honest participants are uniformly distributed from the view of any adversaries.

We are now ready to describe the DKG protocol in the paper of Gennaro *et al.* The DKG protocol consists of two phases, namely, **Generating** and **Extracting**, both of which are detailed in **Protocol 8** below.

Protocol 8 Distributed Key Generation

Generating sk

1. Each participant P_i chooses two random polynomials $f_i(x) = a_{i0} + a_{i1}x + \dots + a_{it}x^t$ and $f'_i(x) = b_{i0} + b_{i1}x + \dots + b_{it}x^t$ and broadcasts the commitments

$$C_{ik} = g^{a_{ik}} h^{b_{ik}} \quad \forall k \in \{0, 1, \dots, t\}.$$

2. Each participant P_i computes $s_{ij} = f_i(j)$ and $s'_{ij} = f'_i(j)$ and securely sends (s_{ij}, s'_{ij}) to P_j for all $j \neq i$.
3. Each participant P_j , upon receiving (s_{ij}, s'_{ij}) verifies the shares he received from each P_i by checking whether

$$g^{s_{ij}} h^{s'_{ij}} \stackrel{?}{=} g^{f_i(j)} h^{f'_i(j)} = \prod_{k=0}^t g^{a_{ik}j^k} h^{b_{ik}j^k} = \prod_{k=0}^t C_{ik}^{j^k}. \quad (4.1)$$

If the check fails for some index i , P_j broadcasts a complaint against P_i .

4. Each P_i who receives a complaint from P_j broadcasts (s_{ij}, s'_{ij}) that satisfy (4.1).
5. A participant P_i is disqualified if he receives $t + 1$ complaints or answers with value that does not satisfy (4.1). A set **QUAL** of qualified participants is then determined.
6. For each i , the secret key sk_i of P_i is equal to

$$\text{sk}_i = \sum_{j \in \text{QUAL}} f_j(i) = \sum_{j \in \text{QUAL}} s_{ji}.$$

For any set \mathcal{V} of at least $t + 1$ participants, the secret key sk can be calculated via Lagrange interpolation as follows.

$$\text{sk} = \sum_{i \in \mathcal{V}} \text{sk}_i \cdot \lambda_{i,\mathcal{V}} = \sum_{i \in \mathcal{V}} \left(\sum_{j \in \text{QUAL}} f_j(i) \right) \cdot \lambda_{i,\mathcal{V}} = \sum_{j \in \text{QUAL}} f_j(0).$$

Protocol 8 Distributed Key Generation (continued)

Extracting $\text{pk} = g^{\text{sk}}$

1. Each participant P_i in the set **QUAL** publishes $A_{ik} = g^{a_{ik}} \forall k \in \{0, 1, 2, \dots, t\}$.
2. For each i , each participant P_j verifies the correctness of $A_{ik} \forall k \in \{0, 1, \dots, t\}$ by checking whether

$$g^{s_{ij}} = g^{f_i(j)} = g^{\sum_{k=0}^t a_{ik} j^k} \stackrel{?}{=} \prod_{k=0}^t A_{ik}^{j^k}. \quad (4.2)$$

If verification fails at i , then P_j complains against P_i by broadcasting values (s_{ij}, s'_{ij}) that satisfies (4.1) but not (4.2).

3. For each i that P_i receives at least one valid complaint, all other participants execute the **VSSReconstruct** algorithm of Pedersen VSS (see **Protocol 6**) to reconstruct $f_i(x)$, and restore s_{i0} and A_{ij} for $j \in \{0, 1, \dots, t\}$. The public key pk is equal to

$$\text{pk} = g^{\text{sk}} = g^{\sum_{i \in \text{QUAL}} f_i(0)} = \prod_{i \in \text{QUAL}} A_{i0}.$$

Finally, the public key pk_i of P_i is calculated as

$$\text{pk}_i = g^{\text{sk}_i} = \prod_{j \in \text{QUAL}} g^{s_{ji}} = \prod_{j \in \text{QUAL}} \prod_{k=0}^t A_{jk}^{j^k}.$$

Example 4.3.1. We give a quick demonstration of the DKG protocol in action below. For simplicity, we assume all participants follow the protocol honestly and omit the verification steps in both phases.

\mathbb{G} : the group of elliptic curve $y^2 = x^3 + 7$ over \mathbb{F}_{43} . The order p of \mathbb{G} is 31.

Generators: $g = (7, 7)$, $h = (42, 7)$.

The values n and t are set to be $n = 4$, $t = 2$.

Generating phase: This phase proceed as follows:

1. Each participant P_i choose two polynomials $f_i(x) = a_{i0} + a_{i1}x + a_{i2}x^2$ and $f'_i(x) = b_{i0} + b_{i1}x + b_{i2}x^2$. Suppose the polynomials $f_i(x)$ and $f'_i(x)$ of P_i is listed as below.

Participant	P_1	P_2	P_3	P_4
$f_i(x)$	$x^2 + x + 1$	$x^2 + 2x + 3$	$x^2 + 3x + 5$	$x^2 + 4x + 7$
$f'_i(x)$	$2x^2 + x + 3$	$2x^2 + 3x + 6$	$2x^2 + 5x + 9$	$2x^2 + 7x + 12$

2. Each participant P_i publishes C_{ij} and (s_{ij}, s'_{ij}) . For example P_1 publishes $C_{10} = (2, 12)$, $C_{11} = (37, 36)$, $C_{12} = (20, 3)$. His shares are $(s_{11}, s'_{11}) = (3, 6)$, $(s_{12}, s'_{12}) = (7, 13)$, $(s_{13}, s'_{13}) = (13, 24)$, $(s_{14}, s'_{14}) = (21, 8)$. This is exactly the numerical example given in Example 2.4.10.

3. Similarly, we get the list of (C_{ij}) and (s_{ij}, s'_{ij}) for each P_i as follows.

Participant	P_1	P_2	P_3	P_4
C_{i0}	$C_{10} = (2, 12)$	$C_{20} = (21, 18)$	$C_{30} = (25, 18)$	$C_{40} = (42, 7)$
C_{i1}	$C_{11} = (37, 36)$	$C_{21} = (35, 21)$	$C_{31} = (29, 12)$	$C_{41} = (38, 22)$
C_{i2}	$C_{12} = (20, 3)$	$C_{22} = (20, 3)$	$C_{32} = (20, 3)$	$C_{42} = (20, 3)$

Participant	P_1	P_2	P_3	P_4
(s_{i0}, s_{i1})	$(s_{11}, s'_{11}) = (3, 6)$	$(s_{21}, s'_{21}) = (6, 11)$	$(s_{31}, s'_{31}) = (9, 16)$	$(s_{41}, s'_{41}) = (12, 21)$
(s_{i2}, s_{i2})	$(s_{12}, s'_{12}) = (7, 13)$	$(s_{22}, s'_{22}) = (11, 20)$	$(s_{32}, s'_{32}) = (15, 27)$	$(s_{42}, s'_{42}) = (19, 3)$
(s_{i3}, s_{i3})	$(s_{13}, s'_{13}) = (13, 24)$	$(s_{23}, s'_{23}) = (18, 2)$	$(s_{33}, s'_{33}) = (23, 11)$	$(s_{43}, s'_{43}) = (28, 20)$
(s_{i4}, s_{i4})	$(s_{14}, s'_{14}) = (21, 8)$	$(s_{24}, s'_{24}) = (27, 19)$	$(s_{34}, s'_{34}) = (2, 30)$	$(s_{44}, s'_{44}) = (8, 10)$

4. Each participant now verifies the shares of other participants. An example of share verification when P_1 is the dealer can be found in Example 2.4.10, the numerical examples are identical there for P_1 . We omit this step here for simplicity.

5. Because each participant followed the protocol honestly, they pass the verification step. Hence, the set of qualified participants is $\text{QUAL} = \{1, 2, 3, 4\}$

6. The secret key sk_1 of P_1 is calculated as $\text{sk}_1 = 3 + 6 + 9 + 12 \pmod{37} = 30$. Similarly, the secret key of each participant is listed below.

Participant	P_1	P_2	P_3	P_4
Secret key sk_i	$\text{sk}_1 = 30$	$\text{sk}_2 = 21$	$\text{sk}_3 = 20$	$\text{sk}_4 = 27$

7. Finally, the secret key \mathbf{sk} is set to be \mathbf{sk} is 16

Extracting phase: This phase proceed as follows:

1. Each participant P_i now publish his commitment $A_{ij} = g^{a_{ij}}$ for $j = 0, 1, \dots, t$. Suppose that each participant P_i has broadcasted the values A_{ij} below.

Participant	P_1	P_2	P_3	P_4
A_{i0}	$A_{10} = (7, 7)$	$A_{20} = (29, 31)$	$A_{30} = (42, 7)$	$A_{40} = (34, 40)$
A_{i1}	$A_{11} = (7, 7)$	$A_{21} = (21, 18)$	$A_{31} = (29, 32)$	$A_{41} = (32, 40)$
A_{i2}	$A_{21} = (7, 7)$	$A_{22} = (7, 7)$	$A_{32} = (7, 7)$	$A_{42} = (7, 7)$

2. P_j now verifies A_{ij} for each j . This is similar to the share verification step in Example 2.4.9 when P_1 is the dealer, with the same numerical example there. Once again, we omit this step for simplicity.
3. The public key \mathbf{pk}_1 is calculated as $\mathbf{pk}_1 = (7, 7) \cdot 30 = (7, 36)$. Similarly, we get the public keys of each participant as follows.

Participant	P_1	P_2	P_3	P_4
Public key $\mathbf{pk}_i = g^{\mathbf{sk}_i}$	$\mathbf{pk}_1 = (7, 36)$	$\mathbf{pk}_2 = (40, 25)$	$\mathbf{pk}_3 = (20, 40)$	$\mathbf{pk}_4 = (32, 3)$

4. Finally, the public key \mathbf{pk} is set to be $\mathbf{pk} = (2, 12)$

The DKG construction of Gennaro *et al.* above achieves Correctness and Secrecy, the two required properties of a DKG mentioned in Section 2.4.11. Its security proof can be found in Chapter 4 of [GJKR99].

4.4 Verifiable Random Function Based on Elliptic Curves

In this section, we describe the **VRF based on Elliptic Curve** (EVCRF) in the paper of [PWH⁺17], which will be used in the DVRF protocol. While there exist various constructions of VRF, namely [DY05, HW10, BMR10, Jag15], our focus is on a VRF construction that can be made distributed in a (t, n) threshold manner. Among the existing VRF constructions in

the literature, only the construction of [DY05] and [PWH⁺17] can be made distributed in a (t, n) threshold manner via Lagrange interpolation. Comparatively, the construction of [DY05] requires bilinear pairing and needs 1000 bit inputs to achieve 128 bit security (this is due to the exponential security loss in their security proof). In contrast, the ECVRF construction only needs 256 bit inputs to achieve the same security and does not rely on bilinear pairings. Consequently, the ECVRF emerges as the most suitable choice for the construction of the DVRF protocol.

We now delve into the details of the ECVRF construction based on elliptic curves in [PWH⁺17]. The name ECVRF simply comes from the fact that the output can be computed using only group operations and hashing, and the group \mathbb{G} in the VRF construction is instantiated with an elliptic curve described in Section 2.3. At a high level, let H_1 be a hash function mapping an input X to a group element in \mathbb{G} , then the output Y of the VRF is straightforwardly computed as $Y = H_1(X)^{sk}$. Now given the public key $pk = g^{sk}$, the output Y and the value $h = H_1(X)$, the prover has to create a proof π to convince the verifier that $Y = h^{sk}$. Fortunately, recall that proving $Y = h^{sk}$ is exactly equivalent to proving that

$$((g, h, pk, Y), sk) \in \mathcal{R}_{\text{DLOG}},$$

where the relation $\mathcal{R}_{\text{DLOG}}$ was defined in Subsection 2.4.6.2. As a result, we can employ the algorithms `NIZKDLOGProve` and `NIZKDLOGVerify` to create the proof and validate whether $Y = h^{sk}$ respectively. With the discussions above, we are now ready to define the algorithms of ECVRF. The algorithms of ECVRF can be described as follows.

Public parameters. Let p be a prime number and \mathbb{G} be a cyclic group of order p and generator g . Let H_1 be a hash function that maps a bit string to an element in \mathbb{G} . Let H_3 be a hash function that maps an arbitrary number of elements in \mathbb{G} into an integer. Recall the H_3 is the hash function H used in the `NIZKDLOGProve` and `NIZKDLOGVerify` algorithm in Subsection 2.4.6.2. The hash functions H_1, H_3 are modeled as random oracle models. The public parameters are $p, \mathbb{G}, g, H_1, H_3$.

Algorithm 9 VRFSetup(1^λ)

Input: 1^λ

Output: (pk, sk)

Sample $sk \xleftarrow{\$} \mathbb{Z}_p$ and set the secret key to be sk . The public key is $pk = g^{sk}$.

Given the secret key \mathbf{sk} and an input X , the output value Y and its proof π is computed in **Algorithm 10** below.

Algorithm 10 VRF Eval(\mathbf{sk}, X)

Input: \mathbf{sk}, X

Output: Y, π

1. Compute $h = H_1(X)$ and $\gamma = h^{\mathbf{sk}}$.
 2. Compute $\pi' \leftarrow \text{NIZKDLOGProve}(g, \mathbf{pk}, h, \gamma, \mathbf{sk})$ (See **Algorithm 2**).
 3. Output $(Y, \pi) = (\gamma, (\gamma, \pi'))$.
-

Given the public key \mathbf{pk} , the VRF input X , the VRF output Y and its proof $\pi = (\gamma, \pi')$, the verification step proceeds as in **Algorithm 11** below.

Algorithm 11 VRF Verify(\mathbf{pk}, X, Y, π)

Input: \mathbf{pk}, X, Y, π

Output: $b \in \{0, 1\}$

1. Parse $\pi = (\gamma, \pi')$.
 2. If $Y \neq \gamma$ return 0.
 3. Compute $h = H_1(X)$.
 4. Compute $b = \text{NIZKDLOGVerify}(g, \mathbf{pk}, h, \gamma, \pi')$ (See **Algorithm 3**).
 5. Output b .
-

Example 4.4.1. We give a quick demonstration of the ECVRF.

\mathbb{G} : the group of elliptic curve $y^2 = x^3 + 7$ over \mathbb{F}_{43} . The order p of \mathbb{G} is 31.

Generator: $g = (7, 7)$.

Input: $X = 8$.

VRFSetup: It chooses a random value in \mathbb{Z}_{31} , say 5, and returns the secret key $\mathbf{sk} = 5$ and public key $\mathbf{pk} = (7, 7) \cdot 5 = (42, 7)$.

VRFEval: From the input $X = 8$ and $sk = 5$, it does the following:

1. Compute $h = H_1(8)$. Because H_1 acts as a black box, we can just choose a point, say $(38, 22)$ that this function always returns $h = (38, 22)$ for input 8 for simplicity.
2. Compute $\gamma = (28, 22) \cdot 5 = (13, 22)$.
3. Choose a random k in \mathbb{Z}_{31} , say $k = 11$.
4. Compute $(7, 7) \cdot 11 = (20, 3)$ and $(38, 22) \cdot 11 = (42, 36)$. Then, compute $c = H_3((7, 7), (38, 22), (42, 7), (13, 22), (20, 3), (42, 36))$. Since H_3 acts as a black box, we can choose $c = 9$ for simplicity, and H_3 always return 9 for inputs $(7, 7)$, $(38, 22)$, $(42, 7)$, $(13, 22)$, $(20, 3)$ and $(42, 36)$.
5. Compute $s = 11 - 9 \cdot 5 \pmod{31} = 28$.
6. The output of the VRF is $Y = \gamma = (13, 22)$.
7. The proof of the VRF is $\pi = (\gamma, c, s) = ((13, 22), 9, 28)$.

VRFVerify: From the input $X = 8$, $Y = (13, 22)$, $\pi = ((13, 22), 9, 28)$ and $pk = (42, 7)$, it does the following:

1. Compute $u = (42, 7) \cdot 9 + (7, 7) \cdot 28 = (20, 3)$.
2. Compute $h = H_1(8) = (38, 22)$.
3. Compute $v = (13, 22) \cdot 9 + (38, 22) \cdot 28 = (42, 36)$.
4. Finally, we see that $H_3((7, 7), (38, 22), (42, 7), (13, 22), (20, 3), (42, 36)) = 9$ and $Y = (13, 22) = \gamma$. Since both conditions are satisfied, the function returns 1.

The ECVRF construction above achieves Correctness, Uniqueness and Pseudo-randomness, the three required properties of a VRF mentioned in Section 2.4.12. Its security proof can be found in Appendix B of [PWH⁺17].

4.5 Construction

Now that we have described the two main components, in this section we formally describe the DVRF protocol of Galindo *et al.* in [GLOW21] using the syntax in Subsection 3.1.1.

4.5.1 Intitution

The DVRF protocol is divided into epochs. Initially, all participants execute the **DRNGSetup** protocol by following the DKG protocol of Gennaro *et al.* Starting from the second epoch, the value Ω_{r+1} of the next epoch is deterministically obtained from the output Ω_r and the state **st** of the current epoch via the **DRNGGen** protocol. Finally, an external verifier can then validate Ω_r using the **DRNGVerify** algorithm.

The idea behind this construction is that after participants execute the DKG protocol of Gennaro *et al.*, a public-secret key pair $(\mathbf{pk}, \mathbf{sk})$ is created. In addition, each participant additionally receives a public-secret key pair $(\mathbf{pk}_i, \mathbf{sk}_i)$. As in Subsection 4.3, it can be shown that, for any set $\mathcal{V} \subseteq \{1, \dots, n\}$, with $|\mathcal{V}| = t + 1$, by Lagrange interpolation it holds that

$$\begin{cases} \mathbf{sk} = \sum_{i \in \mathcal{V}} \mathbf{sk}_i \cdot \lambda_{i, \mathcal{V}}, \\ \mathbf{pk} = g^{\mathbf{sk}} = g^{\sum_{i \in \mathcal{V}} \mathbf{sk}_i \cdot \lambda_{i, \mathcal{V}}} = \prod_{i \in \mathcal{V}} \mathbf{pk}_i^{\lambda_{i, \mathcal{V}}}. \end{cases} \quad (4.3)$$

Now, let us move to the random generation process. By employing the ECVRF construction (as detailed in Section 4.4), participant P_i generates his output and its accompanying proof (Y_{ri}, π_{ri}) , then broadcast them to all remaining participants. Recall that in the construction of ECVRF, for each i it holds that the output Y_{ri} has the form $Y_{ri} = \mathbf{H}_1(X)^{\mathbf{sk}_i}$.

Now suppose there is a set \mathcal{V} of participants who submitted valid ECVRF outputs and proofs $(Y_{ri}, \pi_{ri})_{i \in \mathcal{V}}$. If $|\mathcal{V}| \geq t + 1$ then we can use Lagrange interpolation to produce the output Ω_r and its proof π_r by computing

$$\begin{cases} \gamma_r = \prod_{i \in \mathcal{V}} Y_{ri}^{\lambda_{i, \mathcal{V}}} \text{ and } \Omega_r = \mathbf{H}_2(\gamma_r), \\ \pi_r = \{(i, Y_{ri}, \pi_{ri}) \mid i \in \mathcal{V}\}. \end{cases} \quad (4.4)$$

We can see that, due to (4.3) and (4.4), the output Ω_r is actually deterministic regardless of the set \mathcal{V} and can be calculated as follows.

$$\Omega_r = \mathbf{H}_2(\gamma_r) = \mathbf{H}_2\left(\prod_{i \in \mathcal{V}} Y_{ri}^{\lambda_{i, \mathcal{V}}}\right) = \mathbf{H}_2\left(\prod_{i \in \mathcal{V}} \mathbf{H}_1(X)^{\mathbf{sk}_i \cdot \lambda_{i, \mathcal{V}}}\right) = \mathbf{H}_2(\mathbf{H}_1(X)^{\mathbf{sk}}).$$

Here \mathbf{H}_2 is used to ensure that the output Ω_r is a bitstring since γ_r is a group element. Hence, the output Ω_r can always be constructed from the valid outputs of $t + 1$ participants. If we

assume the number of dishonest participants to be at most t and $t \leq n/2$, then the DVRF can always produce the correct output in each epoch.

Finally, an external verifier, given the output Ω_r and the proof π_r above, can verify the correctness of the protocol by simply checking

$$\begin{cases} \text{VRFVerify}(X, Y_{ri}, \pi_{ri}, \text{pk}_i) \stackrel{?}{=} 1 \ \forall \ i \in \mathcal{V}_r, \\ \gamma_r = \prod_{i \in \mathcal{V}} Y_{ri}^{\lambda_{i,\nu}} \quad \text{and} \quad \Omega_r \stackrel{?}{=} \text{H}_2(\gamma_r). \end{cases}$$

Later, we will prove that an external verifier can really check the correctness of the protocol by doing these checks in Subsection 4.6.4.

4.5.2 Actual Construction

In this subsection, we present the actual construction of the DVRF protocol. The protocol assumes a static adversary and at any time, the number of corrupted participants t is always less than $n/2$. In addition, the protocol also assumes a synchronous network, as described in Section 2.4.2.2.

The formal description of the DVRF protocol can be seen at **Protocol 12** below. For better visualization, Figure 4.1 presents the flow of the DRNGSetup protocol, Figure 4.2 describes the flow of the DRNGGen protocol, and Figure 4.3 describe the flow of the DRNGVerify algorithm.

Protocol 12 Distributed VRF

In the protocol, the global state \mathbf{st} has the form $\mathbf{st} = (\mathbf{st}_r.X, \mathbf{st}_r.r)$. In addition, we introduce a parameter r_{rerun} indicating that the DRNGSetup has to be replaced once every $2^{r_{\text{rerun}}}$ epoch.

$\text{DRNGSetup}(1^\lambda) \langle \{P\}_{P \in \mathcal{P}} \rangle :$

All participants $P \in \mathcal{P}$ follow the DKG protocol of Gennaro *et al.* (see Subsection 4.3) to select the list QUAL of qualified participants and obtain their secret and public keys. Denote \mathbf{sk}_i as the i -th secret key, and let \mathbf{sk} be the grand secret key. Similarly for \mathbf{pk}_i and \mathbf{pk} . The public parameter \mathbf{pp} is set to $\mathbf{pp} := (\{\mathbf{pk}_i\}_{i \in \text{QUAL}})$. The global state \mathbf{st} is initialized to be $\mathbf{st} = (\text{H}_3(\{\mathbf{pk}_i\}_{i \in \text{QUAL}}), 0)$.

$\text{DRNGGen}(\mathbf{st} := \mathbf{st}_r, \mathbf{pp}) \langle \{P_i(\mathbf{sk}_i)\}_{i \in \text{QUAL}} \rangle :$

Let r be the current epoch and \mathbf{st}_r be the state of the r -th epoch. The interactive protocol is executed by participants in QUAL as follows:

1. The i -th participant in the set QUAL execute the algorithm

$$(Y_{ri} = \text{H}_1(\mathbf{st}_r.X)^{\mathbf{sk}_i}, \pi_{ri}) \leftarrow \text{VRF Eval}(\mathbf{st}_r.X, \mathbf{sk}_i) \text{ (see Algorithm 10)}$$

of the ECVRF to compute the value $Y_{ri} = \text{H}_1(\mathbf{st}_r.X)^{\mathbf{sk}_i}$ and broadcasts his value Y_{ri} and his proof π_{ri} to other participants.

2. When a participant publishes his partial ECVRF output, other participants execute

$$b_i \leftarrow \text{VRF Verify}(\mathbf{st}_r.X, Y_{ri}, \pi_{ri}, \mathbf{pk}_i) \text{ (see Algorithm 11)}$$

to check the correctness of Y_{ri} using π_{ri} and \mathbf{pk}_i .

3. From a set \mathcal{V}_r of participants with size $\geq t + 1$ who broadcasted correct ECVRF outputs, participants calculate the output Ω_r and its proof π_r as follows:

- (a) Suppose $\{Y_{ri}\}_{i \in \mathcal{V}_r}$ are the valid outputs of participants in \mathcal{V}_r , calculate γ_r via Lagrange interpolation as follows.

$$\gamma_r = \text{H}_1(\mathbf{st}_r.X)^{\mathbf{sk}} = \prod_{i \in \mathcal{V}_r} \text{H}_1(\mathbf{st}_r.X)^{\lambda_{i, \mathcal{V}_r}} = \prod_{i \in \mathcal{V}_r} Y_{ri}^{\lambda_{i, \mathcal{V}_r}}. \quad (4.5)$$

The output Ω_r of epoch r is calculated to be $\Omega_r = \text{H}_2(\gamma_r)$, which is a bitstring.

- (b) The proof π_r of Ω_r is calculated as follows.

$$\pi_r = \{(i, Y_{ri}, \pi_{ri}) \mid i \in \mathcal{V}_r\}.$$

Protocol 12 Distributed VRF (Continued)

4. Set $\text{st} := \text{st}_{r+1}$ $\text{st}_{r+1}.r = \text{st}_r.r + 1$ and $\text{st}_{r+1}.X$ is formed by concatenating the first $\lambda - r_{\text{rerun}}$ bits of Ω_r and the last r_{rerun} bits of r .

DRNGVerify($\text{st}_r, \Omega_r, \pi_r, \text{pp}$) :

Let r be the current epoch. An external verifier, on input the current state $\text{st}_r = (\text{st}_r.X, \text{st}_r.r)$ value Ω_r , a proof π_r and a public parameter pp can verify the correctness of Ω_r as follows:

1. Parse $\pi_r = \{(i, Y_{ri}, \pi_{ri}) \mid i \in \mathcal{V}_r\}$ and $\text{pk} = \{\text{pk}_i\}_{i \in \text{QUAL}}$.
2. Check whether $\text{VRFVerify}(\text{st}_r.X, Y_{ri}, \pi_{ri}, \text{pk}_i) \stackrel{?}{=} 1$ for all $i \in \mathcal{V}_r$ (see **Algorithm 11**).
3. Check whether the equalities below hold:

$$\begin{cases} \gamma_r \stackrel{?}{=} \prod_{i \in \mathcal{V}_r} Y_{ri}^{\lambda_i, \nu_r}, \\ \Omega_r \stackrel{?}{=} \text{H}_2(\gamma_r). \end{cases}$$

4. If all checks in Step 2 and Step 3 pass, output 1, otherwise output 0.
-

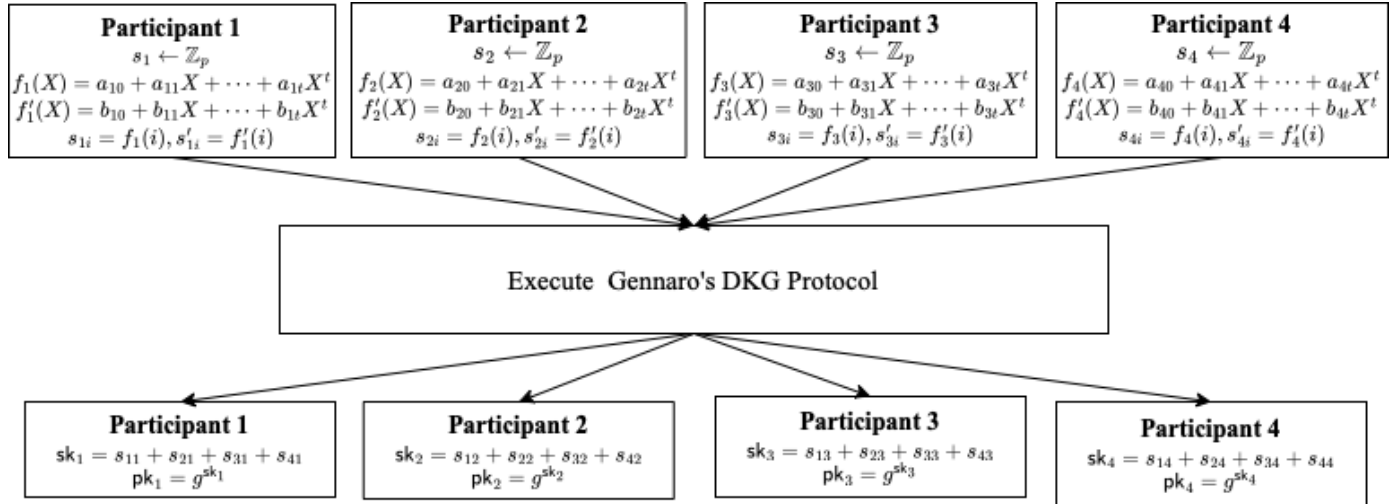


Figure 4.1: *The workflow of DRNGSetup*

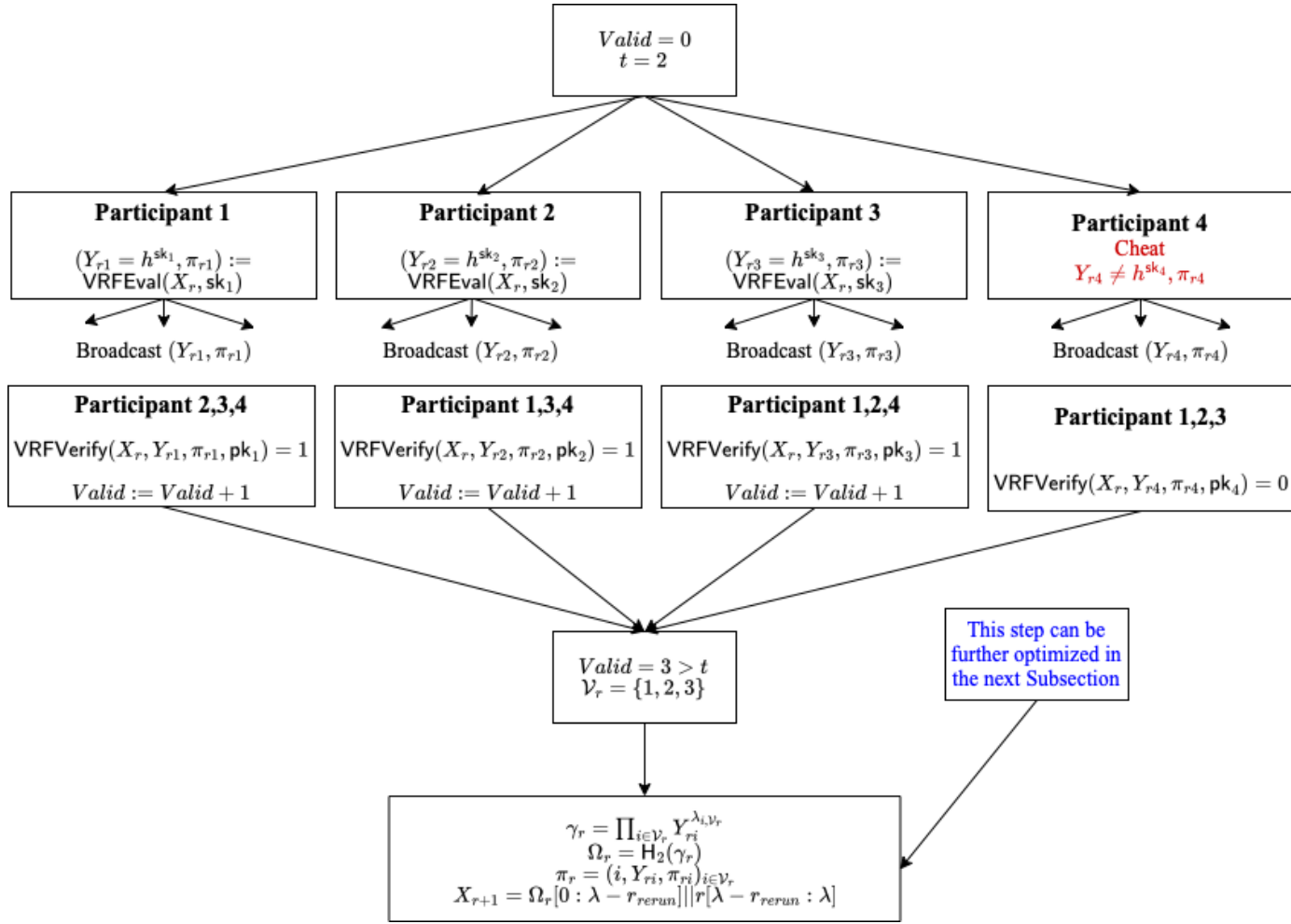


Figure 4.2: The workflow of DRNGGen

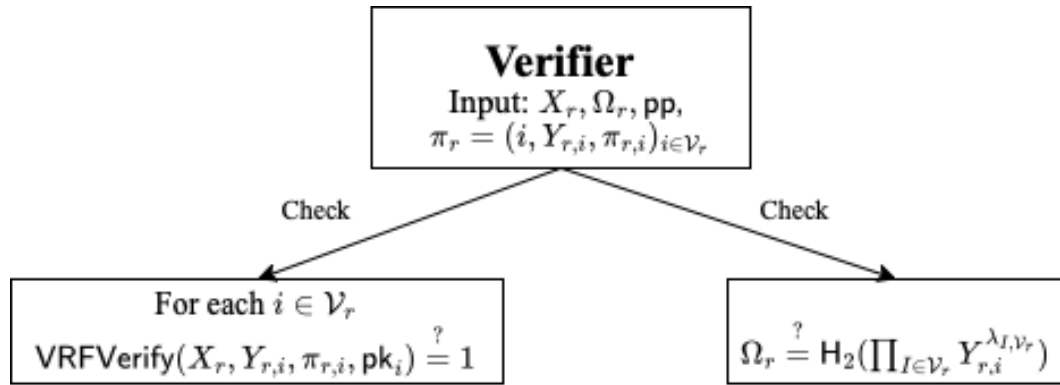


Figure 4.3: The workflow of DRNGVerify

Example 4.5.1. We give a quick demonstration of the DVRF protocol. We continue with Example 4.3.1 to use the same public-secret key pairs $\{(\mathbf{pk}_i, \mathbf{sk}_i)\}_{i=1}^4$ generated by participants. We also continue with Example 4.4.1 to use the same input $\mathbf{st}_5.X$ and the black box assignment of H_1 . The parameters are also the same as follows.

\mathbb{G} : the group of elliptic curve $y^2 = x^3 + 7$ over \mathbb{F}_{43} . The order p of \mathbb{G} is 31.

Generators: $g = (7, 7)$.

The values n and t are set to be $n = 4$, $t = 2$.

DRNGSetup : All participants execute the DKG protocol of Gennaro et al. to generate public-secret key pairs $\{(\mathbf{pk}_i, \mathbf{sk}_i)\}_{i=1}^4$. The process of running the DKG protocol can be found in Example 4.3.1, and hence we omit it here for simplicity. Here we use the same public-secret key pairs in Example 4.3.1.

DRNGGen : Assume the current epoch r is $r = 5$ and the input is $\mathbf{st}_5.X = 8$. Let $h = H_1(8)$, since H_1 acts as a black box, we can assume $h = (42, 7)$ for simplicity. These are the same numerical example found in Example 4.4.1. Now, the phase proceeds as follows.

1. Participant 1, with his secret key $\mathbf{sk}_1 = 30$ runs $\text{VRFEval}(8, 30)$ to compute $Y_{51} = h_1^{\mathbf{sk}} = (42, 7) \cdot 30 = (42, 24)$ and outputs his proof π_{51} of Y_{51} . The process running of VRFEval can be found in Example 4.4.1, hence we omit it here.
2. Similarly, with their corresponding secret key \mathbf{sk}_i , the VRF output Y_{5i} of each participant is listed below.

Participant	P_1	P_2	P_3	P_4
\mathbf{sk}_i	30	21	20	27
$Y_{5i} = h^{\mathbf{sk}_i}$	(42, 24)	(25, 25)	(34, 40)	(20, 3)

3. For each i , each participant proceed to verify the correctness of Y_{5i} by running $\text{VRFVerify}(8, Y_{5i}, \pi_{5i}, \mathbf{pk}_i)$. For example, to verify Y_{51} , other participants execute $\text{VRFVerify}(8, (42, 24), \pi_{51}, (7, 36))$. An example of verification can be found in Example 4.4.1. As stated earlier, for simplicity, we omit this step here.

4. Since we have $3 \geq t + 1$ valid outputs from P_1, P_2, P_3 , we have $\mathcal{V}_5 = \{1, 2, 3\}$. Hence can calculate γ_5 via Lagrange interpolation as follows.

$$\begin{aligned} \gamma_5 = & (42, 24) \cdot ((2 \cdot 3)(1 \cdot 2)^{-1} \pmod{31}) + (25, 25) \cdot ((1 \cdot 3)((-1) \cdot 1)^{-1} \pmod{31}) \\ & + (34, 40) \cdot ((1 \cdot 2)((-2)(-1))^{-1} \pmod{31}) = (12, 12). \end{aligned}$$

Note that the Lagrange coefficients are computed in modulo 31, the order of \mathbb{G} .

5. The output Ω_5 is $\Omega_5 = H_2((12, 12))$. Since H_2 acts as a black box, we can assume that H_2 always returns 21 for the input $(12, 12)$. Thus $\Omega_5 = 21$.

6. The proof π_5 of Ω_5 is set to be

$$\pi_5 = ((1, (42, 24), \pi_{51}), (2, (25, 25), \pi_{52}), (3, (34, 40), \pi_{53}))$$

7. The input for the sixth epoch is computed as $\text{st}_6.X := 216 \pmod{31} = 30$.

DRNGVerify : To verify the correctness of the protocol, a verifier given public keys $\text{pk}_1, \text{pk}_2, \text{pk}_3$ and proof $\pi = \{(I, Y_{5i}, \pi_{5i})\}_{i=1}^3$ as above process as follows:

1. Run $\text{Verify}(X, Y_{5i}, \pi_{5i}, \text{pk}_i) \forall i \in \{1, 2, 3\}$. Again, the verification process can be found in Example 4.4.1, hence we omit it here.
2. Check if $(12, 12) = \gamma_5 = Y_1 * ((2 \cdot 3)(1 \cdot 2)^{-1} \pmod{31}) + Y_2 * ((1 \cdot 3)((-1) \cdot 1)^{-1} \pmod{31}) + Y_3 * ((1 \cdot 2)((-2)(-1))^{-1} \pmod{31}) = (12, 12)$.

Since all checks passed, he accepts the output.

4.5.3 Optimization

In the DVRF protocol above, for each epoch r , we aggregate the outputs from a set \mathcal{V}_r of participants who have produced valid ECVRF outputs to obtain the final result. However, the authors neglected a crucial aspect: The set \mathcal{V}_r can be different for each r . For example, some participants, say P_n contributed outputs in the $(r - 1)$ -th epoch but suddenly became offline in the r -th epoch, and vice versa. Consequently, P_n belongs to the set \mathcal{V}_{r-1} but not \mathcal{V}_r . Recall that, to calculate γ_r , first, we need to calculate all Lagrange coefficients $\lambda_{i, \mathcal{V}_r}$ for all $i \in \mathcal{V}_r$. Since the set \mathcal{V}_r changes with each epoch r , these Lagrange coefficients must also

be recalculated for each epoch r . By doing naive Lagrange interpolation this results in a computational complexity of $\mathcal{O}(t^2)$, hence the protocol becomes inefficient when the number of parties is large, say, more than 1000. Fortunately, in [TCZ⁺20], the authors showed the feasibility of computing all Lagrange coefficients within $\mathcal{O}(n \log^2 n)$ steps. Hence, we can apply their technique in the DVRF protocol to reduce its computation complexity.

Recall that, given a set \mathcal{V} participants of size t , we need to compute all Lagrange coefficients $\lambda_{i,\mathcal{V}}$ for all $i \in \mathcal{V}$ in $\mathcal{O}(n \log^2 n)$ steps. We assume the number of participants and the threshold t are both powers of two and $t < n$. We will replace the participants' ID from $\{1, 2, \dots, n\}$ to $\{1, \omega_n, \omega_n^2, \dots, \omega_n^{n-1}\}$, where ω_n is a n -th root of unity in \mathbb{Z}_p . For a set \mathcal{V} with $|\mathcal{V}| = t + 1$, let

$$L_i^{\mathcal{V}}(x) = \prod_{j \in \mathcal{V}, j \neq i} (x - \omega_n^j) / (\omega_n^i - \omega_n^j).$$

With this new ID, the value γ_r in (4.7) is recalculated to be

$$\gamma_r = \prod_{i \in \mathcal{V}} H_1(X)^{L_i^{\mathcal{V}}(0)} = \prod_{i \in \mathcal{V}} Y_{ri}^{L_i^{\mathcal{V}}(0)}.$$

Our primary objective is to calculate $L_i^{\mathcal{V}}(0)$ for all $i \in \mathcal{V}$. To do this, let

$$f(x) = \prod_{i \in \mathcal{V}} (x - \omega_n^i). \tag{4.6}$$

We additionally introduce $f_i(x)$ and D_i as follows

$$\begin{cases} f_i(x) = f(x) / (x - \omega_n^i), \\ D_i = \prod_{j \in \mathcal{V}, j \neq i} (\omega_n^i - \omega_n^j). \end{cases}$$

We see that $L_i^{\mathcal{V}}(x) = f_i(x) / D_i$. Given these settings, we now provide intuition to calculate $L_i^{\mathcal{V}}(0)$ for all $i \in \mathcal{V}$ within $\mathcal{O}(t \log^2 t)$ steps. First, notice that $D_i = f_i(\omega_n^i) \forall i \in \mathcal{V}$ and $f_i(0) = f(0) / (-\omega_n^i) \forall i \in \mathcal{V}$. Hence it suffices to calculate $f(0)$ and $f_i(\omega_n^i)$ for all $i \in \mathcal{V}$.

On the other hand, given that $f(x)$ has the form of (4.6) the derivative $f'(x)$ of $f(x)$ is calculated as follows

$$f'(x) = \sum_{i \in \mathcal{V}} \prod_{j \in \mathcal{V}, j \neq i} (x - \omega_n^j) = \sum_{i \in \mathcal{V}} f_i(x).$$

It can be easily shown that $f_j(\omega_n^i) = 0 \forall j \neq i$, hence we have $f'(\omega_n^i) = f_i(\omega_n^i) \forall i \in \mathcal{V}$. Given the coefficients of $f'(x)$, then we can utilize FFT algorithm described in Subsection 2.2.3 to get $f'(\omega_n^i) \forall i \in \{0, 1, \dots, n\}$. In addition, the coefficients of $f'(x)$ can be derived from the coefficients of $f(x)$, hence, the problem is reduced to computing all the coefficients of $f(x)$. Finally, we can compute the coefficients of $f(x)$ within $\mathcal{O}(n \log^2 n)$ steps, as we will describe in **Algorithm 14** below.

With the outlined approach above, we sum up the process to calculate $L_i^{\mathcal{V}}(0)$ for all $i \in \mathcal{V}$ within $\mathcal{O}(n \log^2 n)$ steps using the procedure **LagrangeCoefficients** as follows.

Procedure 13 LagrangeCoefficients(\mathcal{V})

Input: A set $\mathcal{V} \subseteq \{0, 1, \dots, n\}$ and $|\mathcal{V}| = t + 1$ and $t < n$.

Output: A list of Lagrange coefficients $L_i^{\mathcal{V}}(0)$ for all $i \in \mathcal{V}$.

1. Get all coefficients a_0, a_1, \dots, a_{n-1} of $f(x)$ recursively. This can be done within $\mathcal{O}(n \log^2 n)$ steps using the algorithm **GetCoefficients**($f(x)$) (see **Algorithm 14**).
2. Calculate $f_i(0) \forall i \in \mathcal{V}$ as follows.

$$f_i(0) = f(0)/(-\omega_n^i) \forall i \in \mathcal{V}.$$

3. Calculate the derivative $f'(x)$ of $f(x)$. Remember that we calculated all the coefficients of $f(x)$ previously in Step 1. Hence if the coefficients of $f(x)$ are a_0, a_1, \dots, a_{n-1} then the coefficients of $f'(x)$ are $a_1, 2 \cdot a_2, \dots, (n-1) \cdot a_{n-1}$.
4. Given the coefficients of $f'(x)$, calculate $f'(\omega_n^i)$ for all $i \in \mathcal{V}$ through a single Fast Fourier Transform (see **Algorithm 1**). Finally, we see that

$$L_i^{\mathcal{V}}(0) = f_i(0)/f'(\omega_n^i) \forall i \in \mathcal{V}.$$

In the procedure **LagrangeCoefficients** above, we can easily check that steps 2 and 3 require $\mathcal{O}(t)$ computations and Step 4 require $\mathcal{O}(n \log n)$ computations due to FFT. Hence, to prove that the algorithm **LagrangeCoefficients** requires $\mathcal{O}(n \log^2 n)$ computations, it suffices to prove that the algorithm **GetCoefficients** in Step 1 has the same computation complexity.

Below we describe the algorithm **GetCoefficients**, which allows us to obtain all the coefficients of $f(x)$ and prove that it terminates within $\mathcal{O}(n \log^2 n)$ steps, thereby concluding our subsection. At a high level, the algorithm also employs a divide and conquer method

by factoring $f(x)$ into $f_0(x)$ and $f_1(x)$ as follows

$$f_0(x) = \prod_{i \in \mathcal{V}[1:t/2]} (x - \omega_n^i) \quad \text{and} \quad f_1(x) = \prod_{i \in \mathcal{V}[t/2+1:t]} (x - \omega_n^i).$$

With $f_0(x)$ and $f_1(x)$ defined above, the idea is to recursively call the algorithm using $f_0(x)$ and $f_1(x)$ as input to get the vectors \mathbf{f}^0 and \mathbf{f}^1 which contains the coefficients of $f_0(x)$ and $f_1(x)$ respectively. Hence it suffices to obtain \mathbf{f} from \mathbf{f}^0 and \mathbf{f}^1 , where \mathbf{f} is the vector containing the coefficients of $f(x)$. The rest is simple: From \mathbf{f}^0 we can obtain $(f_0(\omega_n^i))_{i=0}^{n-1}$ via FFT, and we can do the same for \mathbf{f}^1 . Finally, given $(f_0(\omega_n^i))_{i=0}^{n-1}$ and $(f_1(\omega_n^i))_{i=0}^{n-1}$, we can calculate $f(\omega_n^i) = f_0(\omega_n^i)f_1(\omega_n^i)$ and obtain \mathbf{f} via inverse FFT. With this, we can summarize the algorithm **GetCoefficients** in **Algorithm 14** below.

Algorithm 14 **GetCoefficients**($f(x)$)

Input: $f(x) = \prod_{i \in \mathcal{V}} (x - \omega_n^i)$

Output: A vector of coefficients $\mathbf{f} = (a_0, a_1, \dots, a_{n-1})$ satisfying $f(x) = \sum_{i=0}^{n-1} a_i \cdot x^i$.

1. If $|\mathcal{V}| = 1$ and $\mathcal{V} = \{\omega_n^a\}$ for some a then output $\mathbf{f} = (-\omega_n^a, 1)$. Otherwise, execute steps 2, 3, 4, 5, 6, 7 below.
2. Factor $f(x) = f_0(x)f_1(x)$ where

$$\begin{cases} f_0(x) = \prod_{i \in \mathcal{V}[1:t/2]} (x - \omega_n^i), \\ f_1(x) = \prod_{i \in \mathcal{V}[t/2+1:t]} (x - \omega_n^i). \end{cases}$$

3. Recursively execute $\mathbf{f}^0 = \text{GetCoefficient}(f_0(x))$ and $\mathbf{f}^1 = \text{GetCoefficient}(f_1(x))$ get the vectors \mathbf{f}^0 and \mathbf{f}^1 , which contain the coefficients of $f_0(x)$ and $f_1(x)$ respectively.
4. Compute $(f_0(\omega_n^i))_{i=0}^{n-1}$ and $(f_1(\omega_n^i))_{i=0}^{n-1}$ as follows.

$$\begin{cases} (f_0(\omega_n^i))_{i=0}^{n-1} = \text{FFT}(\mathbf{f}^0, \omega_n), \\ (f_1(\omega_n^i))_{i=0}^{n-1} = \text{FFT}(\mathbf{f}^1, \omega_n) \quad (\text{see Algorithm 1}). \end{cases}$$

5. Compute $f(\omega_n^i) = f_0(\omega_n^i)f_1(\omega_n^i) \forall i \in \{0, 1, \dots, n-1\}$.
 6. Compute $\mathbf{f} = \text{FFT}((f(\omega_n^i)/n)_{i=0}^{n-1}, \omega_n^{-1})$ (see **Algorithm 1**).
 7. Finally, output \mathbf{f} .
-

Finally, we show that the algorithm `GetCoefficient` above terminates within $\mathcal{O}(n \log^2 n)$ steps via Theorem 4.5.1 below.

Theorem 4.5.1. *For all polynomials $f(x) = \prod_{i \in \mathcal{V}} (x - \omega_n^i)$ the algorithm `GetCoefficient`($f(x)$) above terminates within $\mathcal{O}(n \log^2 n)$ steps.*

Proof. Let $T(n)$ consider the upper bound of the number of steps when executing the algorithm `GetCoefficient`($f(x)$) for any polynomial $f(x)$ of degree at most n . We see that, for when factoring $f(x) = f_0(x)f_1(x)$, we recursively call `GetCoefficient`($f_0(x)$) and `GetCoefficient`($f_1(x)$). Since both $f_0(x)$ and $f_1(x)$ has degree at most $n/2$, each call require $T(n/2)$ computation steps, thus Step 2 of the algorithm require $2T(n/2)$ computation steps. In addition, computing $f_0(\omega_n^i)$ and $f_1(\omega_n^i)$ in Step 4 require $\mathcal{O}(n \log n)$ steps using FFT. Finally, given $(f(\omega_n^i))_{i=0}^{n-1}$, calculating the coefficients of f require $\mathcal{O}(n \log n)$ Step 6 using inverse FFT. Thus we have that $T(n) = 2T(n/2) + \mathcal{O}(n \log n)$. Hence $T(n) = \mathcal{O}(n \log^2 n)$, as desired. \square

4.6 Security Proofs

In this section, we present the security proof of the DVRF protocol. The security proof requires that the DDH problem in Section 2.4.3 is hard to solve. We assume a **static** adversary and a **synchronous** network. Note that, even though we do not state it explicitly, there are several variables, such as γ, \mathcal{V} that differ in each epoch r . Hence we automatically use γ_r and \mathcal{V}_r in the proof to denote such variables.

4.6.1 Pseudo-randomness

We prove the Pseudo-randomness property of the DVRF protocol in the following sense: If `DRNGSetup` is executed at epoch 0, then DVRF satisfies the Pseudo-randomness property at Subsection 3.1.2 for all epochs $0, 1, \dots, 2^{r_{\text{rerun}}} - 1$. Then in the $2^{r_{\text{rerun}}}$ -th epoch, we rerun the `DRNGSetup` protocol, and the Pseudo-randomness property is still preserved.

We apply the security reduction method described in Subsection 2.4.4 to prove both Pseudo-randomness. Our main theorem is Theorem 4.6.2, which directly proves Pseudo-randomness. At a high level, assuming an adversary \mathcal{A} that breaks the Pseudo-randomness experiment with a non-negligible probability, we construct a reduction

adversary \mathcal{S} who breaks the DDH assumption in Subsection 2.4.3 with non-negligible probability.

Before going to the main theorem, we present a lemma, which formally proves that the output of the protocol is deterministic, for which we only informally described its intuition in Subsection 4.5.1. The lemma will be used in the theorem.

Lemma 4.6.1. *For all epoch r , given $\{\text{sk}_i\}_{i \in \text{QUAL}}$, the output Ω_r of the DRNGGen protocol is deterministic-ally computed from the input $\text{st}_r.X$, even in the presence of t dishonest participants. More specifically, it holds that $\Omega_r = \text{H}_2(\gamma_r)$ where $\gamma_r = \text{H}_1(\text{st}_r.X)^{\text{sk}}$.*

Proof. Consider a set $\mathcal{V}_r \subseteq \text{QUAL}$ of $t + 1$ participants such that $\text{VRFVerify}(\text{st}_r.X, Y_{ri}, \pi_{ri}, \text{pk}_i) = 1$ for all $i \in \mathcal{V}_r$, then γ_r is calculated as follows

$$\gamma_r = \prod_{i \in \mathcal{V}_r} Y_{ri}^{\lambda_{i, \mathcal{V}_r}}. \quad (4.7)$$

This is described in the protocol. Due to the **Uniqueness** property of ECVRF, it holds that

$$Y_{ri} = \text{H}_1(\text{st}_r.X)^{\text{sk}_i} \quad \forall i \in \mathcal{V}_r. \quad (4.8)$$

Finally, due to the correctness property of DKG, we see that for any set \mathcal{V}_r of $t + 1$ participants, it holds that

$$\text{sk} = \sum_{i \in \mathcal{V}} \text{sk}_i \cdot \lambda_{i, \mathcal{V}_r}. \quad (4.9)$$

By combining (4.7), (4.8) and (4.9) we have the following equality:

$$\gamma_r = \prod_{i \in \mathcal{V}_r} Y_{ri}^{\lambda_{i, \mathcal{V}_r}} = \prod_{i \in \mathcal{V}_r} \text{H}_1(\text{st}_r.X)^{\text{sk}_i \lambda_{i, \mathcal{V}_r}} = \text{H}_1(\text{st}_r.X)^{\sum_{i \in \mathcal{V}_r} \text{sk}_i \lambda_{i, \mathcal{V}_r}} = \text{H}_1(\text{st}_r.X)^{\text{sk}}.$$

Hence, the output Ω_r is computed as $\Omega_r = \text{H}_2(\gamma_r)$, where $\gamma_r = \text{H}_1(\text{st}_r.X)^{\text{sk}}$, as desired. \square

Now, we state the main theorem for proving Pseudo-randomness and give a new proof for it. Our proof is more simple than the one used in [GLOW21].

Theorem 4.6.2. *[Main Theorem] Let \mathcal{A} be an adversary such that $\text{ADV}_{\text{rand-DRNG}}(\mathcal{A}) \geq \epsilon$, Then there exists an adversary \mathcal{S} such that $\text{ADV}_{\text{DDH}}(\mathcal{S}) \geq \frac{\epsilon}{4Q_P} \left(1 - \frac{Q_P \cdot (Q_H + (n-t)Q_P)}{p}\right)$ where $Q_P \leq 2^{r_{\text{rerun}}}$ denote the number of times that \mathcal{A}*

is allowed to access to $\mathcal{O}_{\text{GetRandomness}}$ and Q_H denote the number of times that \mathcal{A} is allowed to query H_1, H_2 and H_3 .

Proof. For every adversary \mathcal{A} , we shall construct an adversary \mathcal{S} that uses \mathcal{A} to break the DDH assumption. Instead of following the rather complicated proof of [GLOW21], we combine the security proof from [GJKR99] and [PWH⁺17] to prove the Pseudo-randomness property.

We denote by \mathcal{B} the set of participants corrupted by \mathcal{A} and \mathcal{G} the set of honest participants run by \mathcal{S} . Without loss of generality, $\mathcal{B} = \{1, 2, \dots, t\}$ and $\mathcal{G} = \{t+1, t+2, \dots, n\}$. Because we are assuming a **static** adversary, then \mathcal{B} is fixed during the experiment.

Note that since \mathcal{S} uses \mathcal{A} , it knows all internal states of \mathcal{A} . In addition, there is a reason we let $Q_P \leq 2^{r_{\text{rerun}}}$, that is to make the public input $\text{st}_r.X$ differs in each epoch r , because the last r_{rerun} bits of $\text{st}_r.X$ is equal to the last r_{rerun} bits of the epoch r (recall that $\text{st}_{r+1}.X$ is formed by concatenating the first $\lambda - r_{\text{rerun}}$ bits of Ω_r and the last r_{rerun} bits of r).

The input of \mathcal{S} is a tuple $(g, g^{\text{sk}}, h, \gamma')$ and it has to decide whether $\gamma' = h^{\text{sk}}$ or γ' is a random element in \mathbb{G} . \mathcal{S} proceeds step by step to create a simulated view of \mathcal{A} that matches each phase in experiment $\text{ExpRand}_{\text{DRNG}}^A(1^\lambda, b)$ as follows:

- **Setup**

1. Execute the **Generating** phase of the **DRNGSetup** protocol described in Section 4.3 honestly on the behalf of honest participants. After this phase, the following holds:
 - The set **QUAL** is well defined. Note that $\mathcal{G} \in \text{QUAL}$.
 - The view of \mathcal{A} consists of the polynomials $f_i(x)$, $f'_i(x)$ for $i \in \mathcal{B}$, the shares (s_{ij}, s'_{ij}) for $i \in \text{QUAL}$, $j \in \mathcal{B}$ and the public commitments C_{ik} for all $i \in \text{QUAL}$, $k \in \{0, 1, 2, \dots, t\}$.
2. Broadcast the values A_{ik} on the behalf of $i \in \mathcal{G} \setminus \{n\}$. It is trivial that the values A_{ik} of each P_i for $i \in \mathcal{G} \setminus \{n\}$ pass the verification step in the **Extraction** phase.
3. For participant P_n , let $f_f^*(x) = a_{n0}^* + a_{n1}^*x + \dots + a_{nt}^*x^t$ be a polynomial such that

$$\begin{cases} f_n^*(j) = s_{nj} \ \forall j \in \{1, 2, \dots, t\}, \\ f_f^*(0) = \text{sk} - \sum_{i \in \text{QUAL} \setminus \{n\}} f_i(0). \end{cases}$$

Let $s_{nj}^* = f_n^*(j)$. Note that \mathcal{S} does not know the polynomial $f_f^*(x)$. The idea is that, instead of broadcasting A_{nk} , \mathcal{S} broadcasts $A_{nk}^* = g^{a_{nk}^*}$ on the behalf of P_n . In this way,

from the view of \mathcal{A} , if the values A_{nk}^* pass the verification step **for each participant controlled by \mathcal{A}** , this means P_n will be convinced to be honest from the view of \mathcal{A} and thus the public key will be set to be $\mathbf{pk} = \left(\prod_{i \in \text{QUAL} \setminus \{n\}} A_{i0} \right) A_{n0}^* = g^{\mathbf{sk}}$ according to Step 3 of the **Extraction** phase.

4. Now \mathcal{S} proceed to compute broadcast A_{nk}^* by the following steps:

- Set $A_{n0}^* = g^{\mathbf{sk}} \cdot \prod_{i \in \text{QUAL} \setminus \{n\}} (A_{i0})^{-1}$. It can be seen that $A_{n0}^* = g^{f_f^*(0)}$.
- From Lagrange Interpolation's theorem in Subsection 2.4.7, we know that

$$f_f^*(x) = \sum_{i=0}^t f_n^*(i) \prod_{j \leq t, j \neq i} \frac{x-j}{i-j}. \quad (4.10)$$

In the right hand side, note that for each $i \in \{1, \dots, t\}$, we have $f_n^*(i) = s_{ni}$, where the values s_{ni} are known by \mathcal{S} . Thus, by letting

$$C_f(x) = \prod_{j < t, j \neq i} \frac{x-j}{i-j} \text{ and } D_f(x) = \sum_{i=1}^t s_{ni} \prod_{j \leq t, j \neq i} \frac{x-j}{i-j}.$$

then $C_f(x)$ and $D_f(x)$ are known by \mathcal{S} given $\{s_{ni}\}_{i=1}^t$. By (4.10), it holds that $f_f^*(x) = f_f^*(0) \cdot C_f(x) + D_f(x)$. Hence, by computing the coefficients of $C_f(x)$ and $D_f(x)$, \mathcal{S} can effectively find constants $\{c_k\}_{k=0}^t$ and $\{d_k\}_{k=0}^t$ such that:

$$f_f^*(x) = \sum_{k=0}^t (c_k + d_k f_f^*(0)) x^k.$$

- With the values $\{c_k\}_{k=0}^t$ and $\{d_k\}_{k=0}^t$ found above, \mathcal{S} now computes

$$A_{nk}^* = g^{c_k} A_{n0}^{*d_k} \quad \forall k = \{1, 2, \dots, t\}.$$

It can be seen that the value A_{nk}^* is equivalent to $g^{a_{nk}^*}$ for $k \in \{0, 1, 2, \dots, t\}$. Thus, they are valid commitments of P_n from the view of each P_i for $i \in \mathcal{B}$ in Step 2 of the **Extraction** phase described in Section 4.3.

- Broadcast A_{nk}^* on the behalf of P_n for $k \in \{0, 1, 2, \dots, t\}$.

5. Honestly follow Step 2 and 3 of the **Extracting** phase described in Section 4.3.

From the view of \mathcal{A} , the public key is now set to be:

$$\text{pk} = A_{n0}^* \prod_{i \in \text{QUAL} \setminus \{n\}} A_{i0} = g^{\text{sk}}.$$

The public keys pk_i of P_i can also be computed similarly according to Step 3 of the **Extracting** phase.

6. Set the public parameter pp to be $\text{pp} := \{\text{pk}_i\}_{i \in \text{QUAL}}$ and the global state st is set to be $\text{st} = (\text{H}_3(\{\text{pk}_i\}_{i \in \text{QUAL}}), 0)$.

• Queries to Random Oracle Models

When \mathcal{A} make a query X to H_1 . Since H_1 is viewed as a random oracle model, normally \mathcal{S} can follow the realization in Subsection 2.4.8 to return $\text{H}_1(X)$ to \mathcal{A} . However, in our proof, \mathcal{S} additionally marks the type of each query he is about to return. This is a well-known technique that originated from [Cor00], which is used to prove the security of cryptographic schemes that involve random oracle models. Back to our proof, \mathcal{S} answers to each query of \mathcal{A} as follows:

1. Check if X has been queried previously. If yes, return the same answer in the previous query.
2. Otherwise, toss a biased coin to determine if X belongs to **type-eval** (with probability $Q_P/(Q_P + 1)$) or **type-challenge** (with probability $1/(Q_P + 1)$). If X belongs to type-eval, \mathcal{S} chooses a random $\rho \in \mathbb{Z}_p$ and returns g^ρ to \mathcal{A} , else \mathcal{S} returns h^ρ to \mathcal{A} .

When \mathcal{A} make a query X to H_2 , or H_3 , \mathcal{S} answer to the queries of \mathcal{A} as follows:

1. Check if the input has been queried previously. If yes, return the same answer in the previous query.
2. Otherwise, return a random value in \mathbb{Z}_p to \mathcal{A} .

• Queries

We adopt the techniques from [PWH⁺17]. For each epoch $r \in \{1, 2, \dots, Q_P\}$, \mathcal{A} queries $\mathcal{O}_{\text{GetRandomness}}$ to get Ω_r and \mathcal{S} has to answer to \mathcal{A} a value Ω_r that **matches the view** of \mathcal{A} when the protocol is executed by all participants (even participants in $\text{QUAL} \cap \mathcal{B}$). First, if the public input $\text{st}_r.X$ of the r -th epoch has not been queried to H_1 then \mathcal{S} perform

the query to get ρ_r and the type of the query. If $\mathbf{st}_r.X = \alpha_i$ for α_i of type-challenge, then \mathcal{S} **aborts** and goes to the **Guess** phase. Otherwise, it does the following:

1. For each $i \in \mathcal{G}$, \mathcal{S} proceeds as follows:

- Compute $\gamma_{ri} = \mathbf{pk}_i^{\rho_r}$.
- Uniformly samples $s_{ri}, c_{ri} \in \mathbb{Z}_p$ and compute $u_{ri} = g^{s_{ri}}(\mathbf{pk}_i)^{c_{ri}\rho_r}$ and $v_{ri} = (g^{\rho_r})^{s_{ri}}(\mathbf{pk}_i)^{c_{ri}\rho_r}$. If $(g, g^{\rho_r}, \mathbf{pk}_i, \gamma_{ri}, u_{ri}, v_{ri})$ has been queried by \mathcal{A} for \mathbf{H}_3 , then \mathcal{S} **aborts** and goes to the **Guess** phase. Otherwise, set $c_{ri} = \mathbf{H}_3(g, g^{\rho_r}, \mathbf{pk}_i, \gamma_{ri}, u_{ri}, v_{ri})$. Finally, set the values Y_{ri} and π_{ri} as follows.

$$\begin{cases} Y_{ri} = \gamma_r, \\ \pi_{ri} = (\gamma_{ri}, c_{ri}, s_{ri}). \end{cases}$$

If \mathcal{S} does not abort, then one can see that the output Y_{ri} and the proof π_{ri} matches the definition of the output and proof of the ECVRF described in Section 4.4 and passes the verification algorithm.

2. For each $i \in \mathbf{QUAL} \cap \mathcal{B}$, wait for each participants P_i to submit his output (Y_{ri}, π_{ri}) . If P_i does not submit his output within a time limit, set $(Y_{ri}, \pi_{ri}) = (\perp, \perp)$.
3. Execute Step 2 of the DRNGGen protocol honestly. In this step, \mathcal{S} just verifies the ECVRF output of each participant in \mathcal{B} and \mathcal{G} .
4. If \mathcal{S} does not choose to **abort**, then \mathcal{S} compute Ω_r and π_r as follows.

$$\begin{cases} \Omega_r = \mathbf{H}_2(\prod_{i \in \mathcal{G}} Y_{ri}^{\lambda_{i,\mathcal{G}}}), \\ \pi_r = \{(i, Y_{ri}, \pi_{ri}) \mid i \in \mathcal{G}\}. \end{cases}$$

and update $\mathbf{st} := \mathbf{st}_{r+1}$ where $\mathbf{st}_{r+1}.r = r + 1$ and $\mathbf{st}_{r+1}.X$ is formed by concatenating the first $\lambda - r_{\text{rerun}}$ bits of Ω_r and the last r_{rerun} bits of r . Finally, \mathcal{S} returns Ω_r and π_r to \mathcal{A} .

• Challenge

Suppose \mathcal{A} has queried the oracle $\mathcal{O}_{\text{GetRandomness}}$ Q_P times, and has to guess the output at epoch $\mathbf{st}.r = Q_P$. Consider the public input $\mathbf{st}_{Q_P}.X$ at this epoch, if $\mathbf{st}_{Q_P}.X$ has not been queried to \mathbf{H}_1 then \mathcal{S} perform the query to get ρ_{Q_P} . If $\mathbf{st}_{Q_P}.X = \alpha_i$ for some α_i of

type-eval, then \mathcal{S} **aborts** and goes to the **Guess** phase. Otherwise \mathcal{S} returns the value $H_2(\gamma'^{\rho_{Q_P}})$ to \mathcal{A} .

- **Guess**

If \mathcal{S} aborts in any previous phases above, then it chooses a random bit $b \in \{0, 1\}$ and returns b . Otherwise, then after returning $H_2(\gamma'^{\rho_{Q_P}})$ to \mathcal{A} , \mathcal{S} receives the guess bit b from \mathcal{A} and outputs b to the DDH challenger.

From the view of \mathcal{A} , during the **Setup** step, the public key is set to $\mathbf{pk} = g^{\mathbf{sk}}$ where \mathbf{sk} is unknown. Note that \mathbf{sk} is uniformly distributed in \mathbb{Z}_p from the view of \mathcal{A} , since it is uniformly chosen by the challenger. For each **query**, if \mathcal{S} does not abort, then each participant $P_i \in \mathcal{G}$ have sent the correct output $(Y_{ri} = H_1(\mathbf{st}_r.X)^{\mathbf{sk}_i}, \pi_{ri})$ of the ECVRF, and thus by Lemma 4.6.1, in Step 4 of each query \mathcal{S} has returned the correct output of the r -th epoch that matches the view \mathcal{A} , even when participants in \mathcal{B} behaves arbitrary.

During the **Challenge** phase, if $\gamma' = h^{\mathbf{sk}}$, by Lemma 4.6.1 \mathcal{S} has returned the **correct** DRNG output of the $(Q_P + 1)$ -th epoch to \mathcal{A} . Otherwise, γ' is **uniformly distributed** in \mathbb{G} , and the output \mathcal{A} received has the form of $H_2(U_G)$. Thus, if \mathcal{A} can distinguish between $H_2(h^{\mathbf{sk}})$ and $H_2(U_G)$, then \mathcal{S} can distinguish between $h^{\mathbf{sk}}$ and a random element in \mathbb{G} if and only \mathcal{S} does not choose to abort. Note that \mathcal{S} does not abort if (a): there is no collision in the queries of \mathcal{H}_3 by \mathcal{A} and (b): the queries of \mathcal{H}_1 (type-eval and type-challenge) does not lead to an abort. We see the following:

1. For each epoch $r = 1, 2, \dots, Q_P$, \mathcal{S} aborts if any query to \mathcal{H}_3 has been defined previously. We see that at any time, the number of inputs that have been defined in \mathcal{H}_3 does not exceed $(n - t)Q_P + Q_H$, since \mathcal{A} can query to \mathcal{H}_3 at most Q_H times, and for each epoch r , \mathcal{S} queries to \mathcal{H}_3 at most $n - t$ times. Because u is randomly chosen, the probability that a tuple $(g, g^\rho, \mathbf{pk}_i, \gamma, u, v)$ belongs to the defined inputs is at most $(Q_H + (n - t)Q_P)/p$. Since we have Q_P epochs, the probability that (a) happens is $1 - Q_P \cdot (Q_H + (n - t)Q_P)/p$.
2. For each epoch $r = 1, 2, \dots, Q_P$, we shall analyze the probability that \mathcal{S} **does not abort** due to type-eval and type-challenge queries. Note that, for each epoch r , if $\mathbf{st}_r.X$ has been queried in previous epoch i , then the probability of $\mathbf{st}_r.X$ being eval-type of challenge-type depends on the previous input $\mathbf{st}_i.X$. Otherwise, this probability is always equal to $Q_P/(Q_P + 1)$ for eval-type and $1/(Q_P + 1)$ for challenge-type, and is independent of anything because its value has not been initialized yet. However, since

we have $Q_P \leq 2^{r_{\text{rerun}}}$, it holds that the last r_{rerun} bits $\text{st}_r.X$ differs each round and thus has $\text{st}_r.X$ not been queried, hence the latter must hold.

Hence we conclude the probability of non-abortion is equal to $Q_P/(Q_P+1)$ (when $\text{st}_r.X$ is type-eval), and for the guessing epoch Q_P+1 is $1/(Q_P+1)$ (when $\text{st}_r.X$ is type-challenge). Hence, the probability that (b) happens is equal to $(Q_P/(Q_P+1))^{Q_P}(1/Q_P) \geq 1/(4Q_P)$.

Thus we see that $\text{ADV}_{\text{DDH}}(\mathcal{S}) \geq \frac{\epsilon}{4Q_P} \left(1 - \frac{Q_P \cdot (Q_H + (n-t)Q_P)}{p}\right)$, as desired. \square

Remark 4.6.3. In our proof, the **DRNGSetup** protocol has to be rerun once every $2^{r_{\text{rerun}}}$ epoch. The same also applies to all constructions where the **DRNGGen** protocol is **deterministic**, such as RandRunner and threshold BLS signature-based DRNGs. One can see that, given the same set of secret keys $\{\text{sk}_i\}_{i \in \text{QUAL}}$ and public parameter pp , then upon executing sufficient enough epochs, there are two epochs $r < r'$ having the same state $\text{st}_r.X$, hence we can immediately guess the output of epoch r' , which is the same as the output of epoch r . This is also a tradeoff between security and efficiency for these protocols. In theory, $r_{\text{rerun}} \leq \lambda$ for DVRF, hence we may only rerun **DRNGSetup** once every 2^λ epochs. However, for implementation, given the unpredictable nature of hash functions, such a collision is very unlikely to happen, thus we may set r_{rerun} to be even higher.

4.6.2 Unbiasability

Proving Unbiasability is surprisingly simple since using Lemma 4.6.1 is sufficient.

Theorem 4.6.4. *DVRF satisfies Unbaisability.*

Proof. After executing the **DRNGSetup** protocol, the set **QUAL** the public parameter pp and the secret key sk is fixed, and the input $\text{st}_r.X$ is always deterministic from the epoch Ω_{r-1} . From Lemma 4.6.1, the for any epoch r with it global state $\text{st} := \text{st}_r$, given the input $\text{st}_r.X$, DRNG output Ω_r is always computed as

$$\Omega_r = \text{H}_2(\text{H}_1(\text{st}_r.X)^{\text{sk}})$$

even in the presence of t dishonest participants. In other words, for any adversaries \mathcal{A} that corrupt t participants, the protocol $\text{DRNGGen}_{\mathcal{A}}(\text{st}_r, \text{pp}) \langle \{P_i(\text{sk}_i)\}_{i \in \text{QUAL}} \rangle$ always returns $\Omega_{r0} = \text{H}_2(\text{H}_1(\text{st}_r.X)^{\text{sk}})$. When all participants honestly follow the protocol, the protocol

$\text{DRNGGen}(\text{st}_r, \text{pp}) \langle \{P_i(\text{sk}_i)\}_{i \in \text{QUAL}} \rangle$ also returns $\Omega_{r1} = \text{H}_2(\text{H}_1(\text{st}_r.X)^{\text{sk}})$ due to Lemma 4.6.1 again. Hence we conclude that $\Omega_{r0} = \Omega_{r1}$ and thus

$$\text{ADV}_{\text{bias-DRNG}}(\mathcal{A}) = \left| \Pr \left[\text{ExpBias}_{\text{DRNG}}^{\mathcal{A}}(1^\lambda) = 1 \right] \right| = \frac{1}{2}.$$

□

4.6.3 Liveness

Theorem 4.6.5. *DVRF satisfies Liveness.*

Proof. Assuming the network model is synchronous, we prove that the protocol can always produce an output, even in the presence of t dishonest participants.

All honest participants will be able to compute and successfully broadcast their ECVRF output. For any epoch r , let \mathcal{V}_r denote the set of participants that have broadcasted the ECVRF values. Verifying the outputs of participants $\in \mathcal{V}_r$ can be successfully done by all honest participants. The 2-nd step of the DRNGGen protocol, as described in Section 4.5, includes verifying all the validity of (Y_{ri}, π_{ri}) for all $i \in \mathcal{V}_r$, this can be done by all honest participants. We see that there is always a set $\mathcal{V}'_r \subset \mathcal{V}_r$ of size $t + 1$ that submitted valid partial outputs since the number of honest parties is at least $n - t \geq t + 1$. From this set \mathcal{V}'_r , Ω_r is successfully computed as $\text{H}_2(\gamma_r)$ where $\gamma_r = \prod_{i \in \mathcal{V}'_r} Y_{ri}^{\lambda_i, \mathcal{V}'_r}$, according to Lemma 4.6.1. Hence the DRNGGen protocol always returns Ω_r and its proof $\pi = (i, Y_{ri}, \pi_{ri})_{i \in \mathcal{V}'_r}$, and thus the output of the DRNG is guaranteed to be delivered, i.e. $\Omega_r \neq \perp$. □

4.6.4 Public Verifiability

Theorem 4.6.6. *DVRF satisfies Public Verifiability.*

Proof. After executing the DRNGSetup protocol, the set QUAL is known, and all the values $A_{ij} = g^{a_{ij}}$ is successfully broadcasted by all qualified participants. Everyone can verify the public key pk_i of participant P_i by checking

$$\text{pk}_i \stackrel{?}{=} \prod_{j \in \text{QUAL}} \prod_{k=0}^t A_{jk}^{i^k}.$$

According to Lemma 4.6.1, the output Ω_r is valid if and only if it is equal to $H_2(H_1(\mathbf{st}_r.X)^{\mathbf{sk}})$. Given the current global state $\mathbf{st}_r = (\mathbf{st}_r.X, \mathbf{st}_r.r)$, and $\mathbf{pp} = \{\mathbf{pk}_i\}_{i \in \text{QUAL}}$, the output Ω_r^* , the proof $\pi^* = \{(i, Y_{ri}, \pi_{ri})\}_{i \in \mathcal{V}_r}$ an external verifier can check whether $\Omega_r^* \stackrel{?}{=} H_2(H_1(\mathbf{st}_r.X)^{\mathbf{sk}})$ by executing $\text{DRNGVerify}(\mathbf{st}_r, \Omega_r^*, \pi_r, \mathbf{pp})$ as follows:

1. To verify the correctness of each Y_{ri} , an external verifier runs $\text{VRFVerify}(\mathbf{st}_r.X, Y_{ri}, \pi_{ri}, \mathbf{pk}_i)$ for all $i \in \mathcal{V}_r$. Due to the Uniqueness property of ECVRF, this successfully convinces the verifier whether the Y_{ri} was computed correctly.
2. If all the Y_{ri} are correct, the outputs Ω_r of the protocol deterministically computed as

$$\Omega_r = H_2 \left(\prod_{i \in \mathcal{V}_r} Y_{ri}^{\lambda_{i, \mathcal{V}_r}} \right).$$

Hence in the second step, using the Y_{ri} for $i \in \mathcal{V}_r$, an external verifier can simply check:

$$\begin{cases} \gamma_r \stackrel{?}{=} \prod_{i \in \mathcal{V}_r} Y_{ri}^{\lambda_{i, \mathcal{V}_r}}, \\ \Omega_r \stackrel{?}{=} H_2(\gamma_r). \end{cases}$$

Thus an external verifier can successfully determine whether Ω_r is correctly computed by $\text{DRNGGen}(\{\mathbf{st}_r, \mathbf{pp}\}) \langle \{P_i(\mathbf{sk}_i)\}_{i \in \text{QUAL}} \rangle$, as desired.

□

4.7 Complexity Analysis

In this section, we analyze the efficiency of the DVRF protocol. Recall that in Section 3.3.2, the efficiency of a DRNG protocol is determined by three factors: The communication complexity of each epoch, the computation complexity of each participant required to generate an output, and the verification complexity of an external verifier. We shall analyze the three factors in the setup protocol DRNGSetup and the output generation protocol DRNGGen separately. We will report the communication complexity in the number of bits that a participant needs to send in every epoch. We assume that each message is $\mathcal{O}(\lambda)$ bits long. We also assume that it takes $\mathcal{O}(1)$ steps to add two points and multiply a point in an elliptic curve.

- **Communication Complexity.** In the DRNGSetup protocol, each participant P_i is required to send t group elements $\{C_{ij}\}_{j=1}^t$ to $n - 1$ remaining participants, followed by securely sending n shares $\{s_{ij}\}_{j=1}^n$ to the other participants. Given that there are n participants, and each participant P_i has to send $\mathcal{O}(t)$ group elements to each other participant, thus a total of $\mathcal{O}(\lambda n^2 t)$ bits are sent.

During the DRNGGen protocol, each qualified participant P_i are required compute $(Y_{ri}, \pi_{ri}) = \text{VRFEval}(\text{st}_r.X, \text{sk}_i)$ and send (Y_{ri}, π_{ri}) to $n - 1$ other participants. Subsequently, participants employ a Byzantine agreement protocol to achieve consensus on common messages. Given that both Y_{ri} and π_{ri} comprise $\mathcal{O}(\lambda)$ group elements, and considering the presence of $\mathcal{O}(n)$ participants, a total of $\mathcal{O}(\lambda n^2)$ bits are sent.

- **Computation Complexity per Node.** In the DRNGSetup protocol, each participant P_i is required to compute $(s_{ij}, s'_{ij}) = f_i(j)$. Using fast Fourier transform, this takes $\mathcal{O}(n \log n)$ field operations. Following this, he has to check the validity of the shares he received from other participants. Since checking each share requires $\mathcal{O}(n)$ multiplications of group elements, checking all shares requires $\mathcal{O}(n^2)$ group element multiplications. Thus we conclude that the computation complexity of each participant is $\mathcal{O}(n^2)$ in this setup phase.

In the DRNGGen protocol, each participant P_i must compute their VRF output $(Y_{ri}, \pi_{ri}) = \text{VRFEval}(\text{st}_r.X, \text{sk}_i)$. This operation has a computational cost of $\mathcal{O}(1)$. Finally, when $t + 1$ valid outputs have been published, one of the nodes must compute γ using **Procedure 13**, which results in a computation complexity of $\mathcal{O}(n \log^2 n)$. Consequently, the total computation complexity per node is always bounded within $\mathcal{O}(n \log^2 n)$.

- **Verification Complexity per Verifier.** To validate the DRNG output Ω_r , an external verifier must execute the DRNGVerify algorithm. The first step of the algorithm involves confirming the validity of (Y_{ri}, π_{ri}) for each participant P_i that has broadcasted their outputs using $\text{VRFVerify}(\text{st}_r.X, Y_{ri}, \pi_{ri}, \text{pk}_i)$. Given that each verification operation incurs a computational cost of $\mathcal{O}(1)$, validating all partial outputs requires $\mathcal{O}(n)$ computations. Subsequently, the verifier needs to calculate γ using **Procedure 13** and verify if $\Omega_r = \text{H}_2(\gamma)$. This computation entails a cost of $\mathcal{O}(n \log^2 n)$ operations. Consequently, the total verification complexity per verifier does not surpass $\mathcal{O}(n \log^2 n)$.

Chapter 5

Our Proposed DRNG System for Blockchain-based Applications

As highlighted in Chapter 1, blockchains lack the capability to generate secure random numbers autonomously. Nevertheless, numerous decentralized applications developed on blockchain platforms rely on random numbers for essential processes like lotteries, gambling, and gaming. However, existing solutions to this challenge often suffer from security vulnerabilities, posing risks to the fairness of these applications. To address this, a decentralized approach to generating randomness is favored for blockchain-based applications, as it enhances fairness and transparency. Moreover, it is crucial for the random generation process to be verifiable by all participants to ensure user trust and confidence in the accuracy of the shuffle process.

The DVRF protocol presented in Chapter 4 is a secure DRNG, making it an ideal candidate for generating randomness in blockchain-based applications, as we have specified our reason in Section 4.1. Therefore, we propose implementing the DVRF protocol as the foundation of a DRNG tailored for blockchain systems, focusing on Ethereum, the blockchain platform boasting the highest **total value locked**. By incorporating the DVRF protocol into Ethereum, we can offer a reliable and secure source of random numbers to enhance the credibility and fairness of Ethereum-based applications. This strategic integration has the potential to significantly impact the blockchain landscape, empowering Ethereum's decentralized ecosystem with a robust and verifiable random number generation mechanism, ultimately benefiting a wide range of applications and use cases within the platform.

5.1 Proposed System Architecture

Our proposal presents two main components: a random system for generating verifiable and secure random numbers and a smart contract system acting as an intermediary for integrating these random numbers into various processes. We emphasize the significant benefits that External Decentralized Applications can gain by utilizing our systems. Their interactions are described in Figure 5.1.

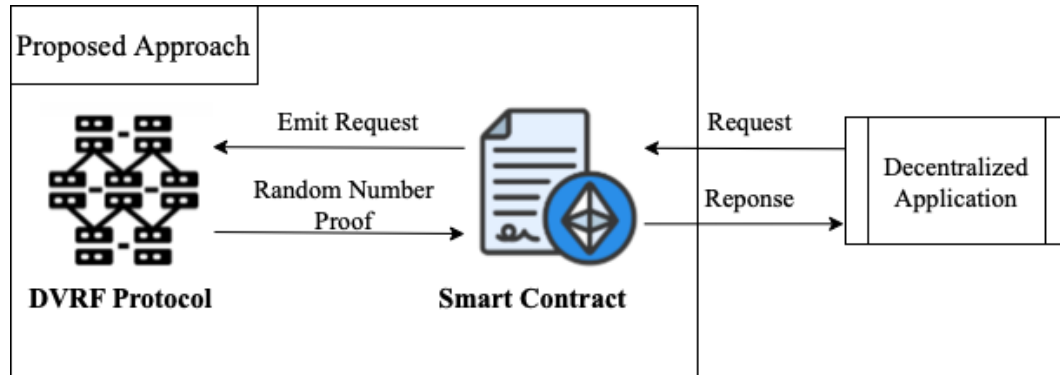


Figure 5.1: *System Architecture.*

- **Decentralized Application.** An external component that seamlessly integrates the random numbers generated by our system into its operations. By utilizing the reliable and verifiable random numbers from our decentralized random number generator, the Dapp can enhance various functionalities and use cases. These applications can leverage random numbers for gaming mechanics to create unpredictable outcomes, ensure fair gameplay, and engage users in dynamic experiences. In lottery drawings, the Dapp can employ unbiased random numbers to select winners transparently, instilling participant trust. Moreover, for secure key generation, the Dapp can rely on our system to produce high-quality random numbers, enhancing the cryptographic security of sensitive information. Additionally, any other use case requiring randomnesses, such as simulations, randomized algorithms, or data sampling, can be enriched by integrating our reliable random number generation. This integration empowers Dapps to build trust, transparency, and efficiency within their decentralized ecosystems, enabling various innovative applications.
- **Smart Contract.** In our system, the smart contract serves as a vital intermediary,

receiving random number requests from other contracts and forwarding them to the DVRF protocol for generation. To uphold transparency and trust, we have implemented a robust verification mechanism. After the DVRF protocol generates the random numbers, the smart contract makes them available for independent verification by the requesting contracts. By examining the cryptographic proofs provided by the DVRF protocol, these contracts can ensure the authenticity and validity of the generated random numbers. This verification process instills confidence in the integrity of our random number generation system, fostering a trustful environment for all participants in our decentralized application ecosystem. It ensures that the random numbers used in various applications are genuinely random and free from manipulation, contributing to our decentralized platform's overall reliability and fairness.

- **DVRF Protocol.** At the heart of our system lies the Decentralized Verifiable Random Function (DVRF) protocol, which empowers participants to generate a pseudo-random output for their applications and proof of correctness. The DVRF protocol ensures that the generated random numbers are unpredictable and verifiable, providing a robust foundation for various use cases within our decentralized ecosystem.

5.2 Proposed System Workflow

When an arbitrary decentralized application (Dapp) needs to generate a random number from our system, it follows the process in Figure 5.2. The detail is followed.

1. In the initial stage, all participants execute the distributed key generation (DKG) protocol to create a pair of cryptographic keys collectively, denoted as (pk, sk) . Additionally, each individual participant, denoted as P_i , receives a partial public-secret key pair, represented as (pk_i, sk_i) .
2. When a Dapp requires a random number, it initiates the process by sending a request to a smart contract. Acting as an intermediary, the smart contract then forwards the request to the DVRF protocol by emitting events, which commences the random number generation process.

3. Upon receiving the event notification from the smart contract, all participants actively execute the DVRF protocol to generate the required random numbers, along with their corresponding proof of correctness. Through this collaborative effort, the decentralized system ensures the generation of unbiased and unpredictable random values. Once the random number and its proof have been generated, they are returned as a response to the smart contract. This allows the smart contract to access and utilize the verified random number for further processing.
4. Once the verification process is completed, the output is seamlessly transmitted to the Dapp, allowing it to proceed with its intended operations. By receiving the verified output, the Dapp can confidently continue its processes, ensuring the reliability and integrity of the information used within the decentralized ecosystem. This streamlined flow of information enhances the overall efficiency and trustworthiness of Dapp’s functionalities, enabling seamless and secure interactions for its users.

Our system is built on the foundation of the secure DVRF protocol, inheriting all its robust security properties for trustworthy random number generation. With careful optimization, our protocol achieves an impressive $\mathcal{O}(n^2)$ communication complexity and $\mathcal{O}(n \log^2 n)$ computation complexity while maintaining full security. Furthermore, it can seamlessly integrate with the widely-used **secp256k1** curve, which is employed by Ethereum and supported by numerous libraries, making our solution highly applicable within the blockchain context. This compatibility ensures efficient and secure random number generation for various decentralized applications, bolstering fairness, transparency, and user trust. As a result, our system presents a valuable contribution to the blockchain ecosystem, driving innovation and adoption in this ever-evolving landscape.

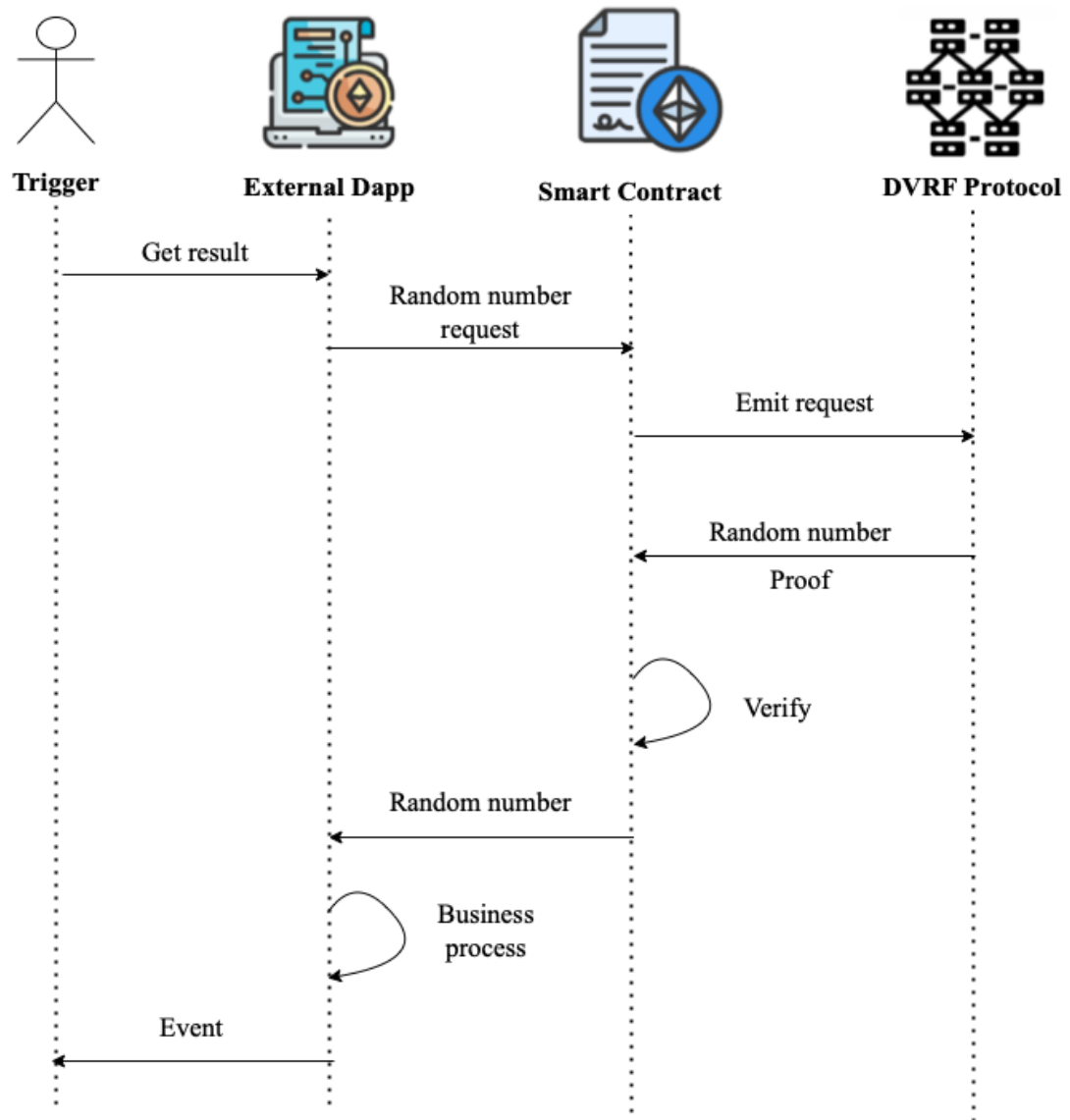


Figure 5.2: *Random number request flow for blockchain-based applications*

Chapter 6

Experiments

In this chapter, we conduct experiments and provide the benchmark results of the protocol. The GitHub repository that contains all the contents of the implementation in this chapter can be found at <https://github.com/phamhatminh1292001/thesis>.

6.1 Programming Language

There are several programming languages for cryptography. Some of the famous languages for cryptography include Python, Rust, and Go. Even though Python provides simple implementation, the main drawback of Python is that it has a very poor performance, and thus, it is rarely used in cryptography for industrial purposes. On the other hand, Rust provides a much better performance; this is useful when we need to generate random numbers quickly for applications, such as games. Another important aspect is that Rust has more secure memory management than Go and Python, making errors such as buffer overflows and use-after-free less likely². This makes Rust a more secure choice among programming languages for cryptography. Moreover, Rust has numerous libraries that support the famous curve secp256k1, such as `curv` and `libsecp256k1`. Finally, Rust implements well-known cryptosystems for blockchain-based applications, such as Halo2, STARK, and threshold ECDSA. Hence, with the abovementioned advantages, we choose **Rust** as our programming language.

²<https://cussy.io/en/blog/rust-for-cryptography>

6.2 Elliptic Curve Dependencies

6.2.1 Library Choice

Various elliptic curves can be used for implementation, such as secp256k1 and ed25519, where the DDH problem is assumed to be hard on the group of these curves. However, we choose the curve **secp256k1** for our implementation since this is the curve used by Bitcoin and Ethereum, the two most well-known blockchains today. We use the library **libsecp256k1**¹ to support the primary operations of the curve secp256k1 in Rust. The library uses the struct **Affine** to represent a point on a curve and the struct **Scalar** to represent a 256 bit integer.

6.2.2 Scalar and Point Structure

Scalar Structure. In libsecp256k1, a 256-bit scalar value, denoted as n , is represented as an array $[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$ where $n = \sum_{i=0}^7 a_i \cdot 2^{32i}$. Such an array structure representing 256 bit integers is defined as follows.

```
1 pub struct Scalar(pub [u32; 8]);
```

However it would be inconvenient when outputs are printed in the format above. Fortunately, it is possible to convert the **Scalar** struct above into a hex string by the following syntax

```
1 let scalar=Scalar([0,0,0,0,0,0,0,0]);
2 let hexstr=hex::encode(scalar.b32());
```

Hence, with the syntax above, all scalar outputs will be formatted as hexstrings.

Point Structure. The **Affine** struct is used to represent a point on the curve secp256k1. The structure is defined as follows.

```
1 pub struct Affine {
2     pub x: Field,
3     pub y: Field,
```

¹<https://github.com/paritytech/libsecp256k1>


```

4     pub infinity: bool,
5 }

```

The public key of the participants will be represented using the **Affine** structure.

6.3 Verifiable Random Function Based on Elliptic Curves

This section provides our implementation of the ECVRF construction described in Section 4.4. Besides the three main functions of a VRF, we additionally implement auxiliary functions that support the computations of the main functions. Section 6.3.2 describes our implementation of the VRF main functions.

6.3.1 Structure

In this section, we describe the structure of an ECVRF in our implementation. The structure of ECVRF contains the following components:

- **secret_key**: The secret key of the VRF.
- **public_key**: The public key of the VRF.
- **ctx_gen**: The generator of the group \mathbb{G} .

An example of an ECVRF structure is as follows.

```

1 pub struct ECVRF<'a> {
2     secret_key: SecretKey,
3     public_key: PublicKey,
4     ctx_mul: &'a ECMultContext,
5     ctx_gen: &'a ECMultGenContext,
6 }

```

6.3.2 Main Functions

There are three main functions of the ECVRF we described in Section 4.4. The `Setup` function returns public and secret keys. In the implementation, it acts as the constructor of the ECVRF class.

```
1 pub fn new(secret_key: SecretKey) -> Self {
2     ECVRF {
3         secret_key: secret_key,
4         public_key: PublicKey::from_secret_key(&secret_key),
5         ctx_gen: &ECMULT_GEN_CONTEXT,
6         ctx_mul: &ECMULT_CONTEXT,
7     }
8 }
```

The `Eval` function computes the ECVRF output and provides proof that its output is computed correctly.

```
1 pub fn prove_contract(self, alpha: &Scalar) -> ECVRFContractProof {
2     let mut pub_affine: Affine = self.public_key.into();
3     let mut secret_key: Scalar = self.secret_key.into();
4     pub_affine.x.normalize();
5     pub_affine.y.normalize();
6     // On-chain compatible HASH_TO_CURVE_PREFIX
7     let h = self.hash_to_curve_prefix(alpha, &pub_affine);
8     // gamma = H * sk
9     let gamma = ecmult(self.ctx_mul, &h, &secret_key);
10    // k = random()
11    // We need to make sure that k < GROUP_ORDER
12    let mut k = randomize();
13    while scalar_is_gte(&k, &GROUP_ORDER) || k.is_zero() {
14        k = randomize();
15    }
16    // Calculate k * G = u
17    let kg = ecmult_gen(self.ctx_gen, &k);
18    // U = c * pk + s * G
19    // u_witness = ecrecover(c * pk + s * G)
20    // this value equal to address(keccak256(U))
21    // It's a gas optimization for EVM
22    let u_witness = calculate_witness_address(&kg);
```

```

23      // Calculate  $k * H = v$ 
24      let kh = ecmult(self.ctx_mul, &h, &k);
25      //  $c = \text{ECVRF\_hash\_points\_prefix}(H, pk, \text{gamma}, u\_witness, k * H)$ 
26      let c = self.hash_points_prefix(&h, &pub_affine, &gamma, &u_witness, &kh);
27      //  $s = (k - c * sk)$ 
28      // Based on Schnorr signature
29      let mut neg_c = c.clone();
30      neg_c.cond_neg_assign(1.into());
31      let s = k + neg_c * secret_key;
32      secret_key.clear();
33      // Gamma witness
34      //  $\text{witness\_gamma} = \text{gamma} * c$ 
35      let witness_gamma = ecmult(self.ctx_mul, &gamma, &c);
36      // Hash witness
37      //  $\text{witness\_hash} = h * s$ 
38      let witness_hash = ecmult(self.ctx_mul, &h, &s);
39      //  $V = \text{witness\_gamma} + \text{witness\_hash}$ 
40      //  $= c * \text{gamma} + s * H$ 
41      //  $= c * (sk * H) + (k - c * sk) * H$ 
42      //  $= k * H$ 
43      let v = projective_ec_add(&witness_gamma, &witness_hash);
44      // Inverse does not guarantee that  $z$  is normalized
45      // We need to normalize it after we have done the inverse
46      let mut inverse_z = v.z.inv();
47      inverse_z.normalize();
48      ECVRFContractProof {
49          pk: self.public_key,
50          gamma,
51          c,
52          s,
53          y: keccak256_affine_scalar(&gamma),
54          alpha: *alpha,
55          witness_address: address_to_scalar(&u_witness),
56          witness_gamma,
57          witness_hash,
58          inverse_z,
59      }
60  }

```

The Verify function checks whether the output of the ECVRF is computed correctly.

```

1 pub fn verify(self, alpha: &Scalar, vrf_proof: &ECVRFProof) -> bool {
2     let mut pub_affine: Affine = self.public_key.into();
3     pub_affine.x.normalize();
4     pub_affine.y.normalize();
5     assert!(pub_affine.is_valid_var());
6     assert!(vrf_proof.gamma.is_valid_var());
7     // H = ECVRF_hash_to_curve(alpha, pk)
8     let h = self.hash_to_curve(alpha, Some(&pub_affine));
9     let mut jh = Jacobian::default();
10    jh.set_ge(&h);
11    // U = c * pk + s * G
12    //     = c * sk * G + (k - c * sk) * G
13    //     = k * G
14    let mut u = Jacobian::default();
15    let pub_jacobian = Jacobian::from_ge(&pub_affine);
16    self.ctx_mul
17        .ecmult(&mut u, &pub_jacobian, &vrf_proof.c, &vrf_proof.s);
18    // Gamma witness
19    let witness_gamma = ecmult(self.ctx_mul, &vrf_proof.gamma, &vrf_proof.c);
20    // Hash witness
21    let witness_hash = ecmult(self.ctx_mul, &h, &vrf_proof.s);
22    // V = c * gamma + s * H = witness_gamma + witness_hash
23    //     = c * sk * H + (k - c * sk) * H
24    //     = k * H
25    let v = Jacobian::from_ge(&witness_gamma).add_ge(&witness_hash);
26    // c_prime = ECVRF_hash_points(G, H, pk, gamma, U, V)
27    let computed_c = self.hash_points(
28        &AFFINE_G,
29        &h,
30        &pub_affine,
31        &vrf_proof.gamma,
32        &jacobian_to_affine(&u),
33        &jacobian_to_affine(&v),
34    );
35    // y = keccak256(gama.encode())
36    let computed_y = keccak256_affine_scalar(&vrf_proof.gamma);
37    // computed values should equal to the real one
38    computed_c.eq(&vrf_proof.c) && computed_y.eq(&vrf_proof.y)
39 }

```

An example of an ECVRF result is given in Figure 6.1.

```

Input:
"fffffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141"
Secret key:
"5deffe7c9a4d696166d795ccee710a27c1360e59572c8044a0e755b4be5399"
Public key:
("311b9e8390f26a8ba97d25b694861a72a001dca0df873d85c1bb8f58d53025ad", "807f17da0470d6a89d23e830b539f022d9cd967039a72013f3feb6ac6281c0b8")
Output:
("ebcd3aed9b2d42224d3bb0b8ac317f6b738b121000085b041f197725ffa9dee4", "3b513a7ab3470dfad2acfe306240ced67de0e5e8d0259028b429b085b37326c")
Proof:
Proof { gamma: ("ebcd3aed9b2d42224d3bb0b8ac317f6b738b121000085b041f197725ffa9dee4", "3b513a7ab3470dfad2acfe306240ced67de0e5e8d0259028b429b085b37326c"), c: "ca1b24fa811a1a3a5a314b188a546feb56049dce3545c0126a5ea2b98a6718ac", s: "1be8b6a4c1fc7dbb390dbdd13c036bd0673aa60ff5fdb773e957b1b477ef47f7" }
Verifying...
Result: true

```

Figure 6.1: *ECVRF Example*

6.4 Implementation Flow

We give a quick demo of the workflow of our implementation. We set $n = 4$ and $t = 3$. Recall that our implementation is supported by the library `libsecp256k1`. Our implementation consists of three following stages:

1. All participants run a DKG protocol to generate a public-secret key pair $(\mathbf{pk}, \mathbf{sk})$. In addition, each participant P_i get a partial public-secret key $(\mathbf{pk}_i, \mathbf{sk}_i)$. After finishing the DKG protocol, the public keys \mathbf{pk}_i are displayed publicly, as shown in Figure 6.2.
2. Each qualified participant P_i uses ECVRF to generate an output Y_i and its proof π_i . When a participant publishes their ECVRF output, other participants use the public key \mathbf{pk}_i and the `VRFVerify` function to verify the correctness of Y_i , as shown in Figure 6.3.
3. Once $t + 1$ partial ECVRF values are confirmed as valid, they are aggregated to generate a single random output, as shown in Figure 6.4. The output format will be in the form of a hex string. Finally, the result will be verified on-chain by a smart contract.

```

Public key of participant 1:
("584f3cbb052a102a1eb1e53dce2e9c0023ff4516b92f4fc01ff5786283d75ed9", "77d2fa6b37760b0c39891c9467ab09ea8efa9ab70d8fe80f3f5963008492c52e")

Public key of participant 2:
("a81638e4cfa176a9c74e983ebe813940aef0021e7ec3b21e932cfda8f56c8236", "1de93832d89a60553cc7001b7cdc553fcd468ab23036e6745ef69d97b784c95")

Public key of participant 3:
("328d61f97479021ab61ede694a1e00f4027908a62b5c8328cd8189c8daebaf8a", "489e0cd38a90ed42d07939ac7df1356876ffe2a002a5d55f5a89ddb97c4cbf27")

```

Figure 6.2: *The public key of participants*

```

Output of participant 1:
("f4e8d9cfe01099c669650d75608c878ec67cd64c67d0507a77ca92910683d792", "22f8ff947bd2934b28a7267d871d05b940e7708a3b7064ba7753a9ddb07141a0")
Proof of participant 1:
Proof { gamma: ("f4e8d9cfe01099c669650d75608c878ec67cd64c67d0507a77ca92910683d792", "22f8ff947bd2934b28a7267d871d05b940e7708a3b7064ba7753a9ddb07141a0"), c: "034a151ff29d3f1456be6137305768ff37d495c0bc8a24a54f4f5292be291c76", s: "41802973b62618c40e7c515447e457d348a152bf9a461cd27ff0e2b0bd6f6963" }
Verifying the output of participant 1:
Result: true

Output of participant 2:
("f4e8d9cfe01099c669650d75608c878ec67cd64c67d0507a77ca92910683d792", "22f8ff947bd2934b28a7267d871d05b940e7708a3b7064ba7753a9ddb07141a0")
Proof of participant 2:
Proof { gamma: ("f4e8d9cfe01099c669650d75608c878ec67cd64c67d0507a77ca92910683d792", "22f8ff947bd2934b28a7267d871d05b940e7708a3b7064ba7753a9ddb07141a0"), c: "dd78400467bba0d3b795b48186a815c2e462507e79020dafcbad476ad93ea6f7", s: "f5d3419e53c01c7c517a80fd820bd030462d0ace6c596681b53c646ad6005fb8" }
Verifying the output of participant 2:
Result: true

Output of participant 3:
("f4e8d9cfe01099c669650d75608c878ec67cd64c67d0507a77ca92910683d792", "22f8ff947bd2934b28a7267d871d05b940e7708a3b7064ba7753a9ddb07141a0")
Proof of participant 3:
Proof { gamma: ("f4e8d9cfe01099c669650d75608c878ec67cd64c67d0507a77ca92910683d792", "22f8ff947bd2934b28a7267d871d05b940e7708a3b7064ba7753a9ddb07141a0"), c: "2cf6441c1bd98a45affff0b629425de7e8a824723fbd9aae4d8bd2d12d4b02bdd", s: "5af13864ccdbf395b9772817a7374a1e8b0e39b497b564dacb44072080dbd561" }

```

Figure 6.3: *Partial ECVRF Output*

```

There are 3 participants that have produced valid outputs: 1,2,3
Combining final output...
Final output: 883010e9779657db869b5b9cf7baed6472cbfdc062b8b93d71b43b0691c0334b

```

Figure 6.4: *Final DRNG Output*

6.5 Evaluation and Result

It is essential to test whether the outputs of the RNG satisfy the required statistical properties of a truly random output bit. There are two ways to conduct such a test—**theoretical** or **statistical**. Theoretical tests focus on the inner structure of the RNG. This type is considered to be the most powerful; if an RNG passes a theoretical test, it will likely pass all other statistical tests. Each test of this kind is only applied to some specific RNGs and needs great effort to construct. It requires a good knowledge of the RNG, how the RNG works, and the properties of each operation in the RNG structure. On the other hand, statistical tests focus on one specific mathematical feature of the number sequence in order to answer whether it follows a **reference distribution**. They require no knowledge of how the RNG works; therefore, they are widely applied to all RNGs. Hence, statistical tests are widely used to evaluate the randomness of the numbers produced by a generator in practice.

It is important to note that a single randomness test is insufficient to determine whether a sequence is random since the sequence can be non-random in various ways. Various algorithms have been developed, each checking whether the sequence has a specified pattern. If any of these patterns are detected, then the sequence is highly non-random. Hence, the more tests passed, the more likely the sequence is random. Randomness is a probabilistic property, which means that how random a sequence is can be characterized in terms of probability. Therefore, sometimes an RNG can produce bit strings that do not seem random, even a TRNG, and thus, some tests will fail. However, as long as the number of failed tests is small enough (i.e. within statistical limits), this does not raise any doubt about the RNG's security.

6.5.0.1 The NIST Test Suite

Among existing statistical tests, we propose to use the NIST SP 800-22 Test Suite, since it is widely used in the industry standard. The U.S. government institution NIST developed and maintained the NIST Statistical Test Suite. It is one of the most well-known used test suites to assess the outputs of random number generators.

The test suite consists of different tests to test the randomness of binary sequences produced by hardware or software-based cryptographic random or pseudo-random number generators following different statistical tests. The tests are described in Table 6.1.

These tests are formulated to test a specific null hypothesis (H_0), saying that the input sequence is random. For each test, the p -value is the strength of the evidence against (H_0). If the p -value is determined to be 1, then the input sequence is perfectly random, while a zero p -value indicates a completely non-random sequence. The significance level (denoted by α) is the probability that a random sequence will be detected non-random by the test; the common value of α in cryptography is 0.01. If $p\text{-value} < \alpha$, (H_0) is rejected; otherwise, we accept (H_0).

For example, an α of 0.01 means that we expect 1 out of 100 sequences to be rejected. If $p\text{-value} \geq \alpha$, the input sequence is considered to be random with a confidence of 99%. If $p\text{-value} \leq \alpha$, the input sequence is considered to be non-random with a confidence of 99%.

Table 6.1: *NIST Test Suite.*

Test name	Description
Frequency	Check the number of zeros and ones.
Frequency in a Block	Check the number of zeros and ones in a M -bit block.
Run	Check if the oscillation between zeros and ones is too fast or too slow
Longest Run of Ones in a Block	Check if the oscillation between zeros and ones in a M -bit block is too fast or too slow.
Binary Matrix Rank	Check for linear dependence among fixed length substrings of the bit sequence.
Discrete Fourier Transform	Test the peak heights in the Discrete Fourier Transform of the bit sequence.
Non-Overlapping Template Matching	Detect template matching in a non-overlapping manner..
Overlapping Template Matching	Detect template matching in an overlapping manner.
Universal Statistical	Detect whether or not the sequence can be significantly compressed without loss of information.
Linear Complexity	Check if the numbers in the sequence are linearly dependent.
Serial	Check the frequency of all possible m -bit patterns in the bit sequence.
Approximate Entropy	Check the frequency of all possible m -bit patterns in the bit sequence.
Cumulative Sums	Check whether the sum of the partial sequences occurring in the tested sequence is too large or too small.

6.5.0.2 The Interpretation of Empirical Results

We have tested our implementation with 20 numbers; each number is 100000 bits long. Table 6.2 shows the result of the test.

Table 6.2: *NIST Test Result.*

Test name	p-value	Pass rate
Frequency	0.122325	20/20
Frequency in a Block	0.437274	20/20
Run	0.035174	20/20
Longest Run of Ones in a Block	0.275709	19/20
Binary Matrix Rank	0.637119	20/20
Discrete Fourier Transform	0.350485	20/20
Non-Overlapping Template Matching	0.964295	20/20
Overlapping Template Matching	0.213309	20/20
Universal Statistical	0.967352	20/20
Linear Complexity	0.004301	19/20
Serial	0.739918	20/20
Approximate Entropy	0.048716	19/20
Cumulative Sums	0.275709	20/20

As in [NIS01], for $m = 20$, the acceptable range is equal to $0.99 \pm 3\sqrt{\frac{0.99(1 - 0.99)}{20}} = [0.923, 1]$. As we can see, all the pass rates of the tests exceed 19/20. Thus, we conclude that the DRNG protocol passes the NIST Test Suite.

Chapter 7

Conclusion

7.1 Conclusion

A brief version of our work in this thesis has been accepted for publication [NCA23] at The 10th International Symposium on Integrated Uncertainty in Knowledge Modelling and Decision Making (IUKM).² In this thesis, we have achieved the following:

1. We have understood the importance of randomness in real life, the challenges associated with generating it, and why we should generate randomness in a decentralized manner.
2. In Subsection 3.1.1, we aimed to provide a comprehensive definition of a DRNG, encompassing both interactive and non-interactive variants. This extends beyond previous work [GLOW21], which primarily focused on non-interactive DRNGs.
3. In Subsection 3.2, we conducted a systematic literature review of various DRNG construction approaches, delving into their details. These constructions are primarily based on hash functions, PVSS, VRF, VDF, threshold signatures, and homomorphic encryption. Each of these approaches has its own advantages and disadvantages in terms of security and efficiency. We analyzed them and made a comparison of these DRNGs in Section 3.4.
4. In Subsection 4.1, based on our systematic literature review, we proposed choosing the DVRF protocol for generating randomness for blockchain-based applications. In addition,

²The conference will take place from 02 - 04 November 2023, in Kanazawa, Japan,
URL: <https://www.jaist.ac.jp/IUKM/IUKM2023>

we presented a new proof for the Pseudo-randomness property of the DVRF protocol in Subsection 4.6. This proof is simpler compared to the original proof of [GLOW21].

5. In Subsection 4.5.3, we applied the optimization method from [TCZ⁺20] to enhance the performance of the DVRF protocol by reducing the time randomness generation process from $\mathcal{O}(n^2)$ down to $\mathcal{O}(n \log^2 n)$.
6. In Chapter 5, we proposed a DRNG system for blockchain-based system using the DVRF protocol. The system has $\mathcal{O}(n^2)$ communication complexity and $\mathcal{O}(n \log^2 n)$ computation complexity after optimization while achieving full security properties. In addition, the system can be implemented with the curve secp256k1, the curve used by Bitcoin, Ethereum and supported by many existing libraries. Thus our system offers great application in the context of blockchain.
7. We conducted experiments to assess the security of our system, using the Rust programming language in Chapter 6.

7.2 Future Work

There are various future directions that can be taken for this work as follows:

1. **Reduce the Proof Size for secp256k1 Implementation.** The disadvantage of the DVRF protocol is that the proof size of the protocol is $\mathcal{O}(t)$ since it consists of all $(t + 1)$ ECVRF proofs of partial outputs. We can use bilinear maps to aggregate the proofs to reduce their size. However, bilinear maps can only be used on certain curves, for instance, BN254. Most libraries in Rust only support the curve secp256k1, and this curve does not allow us to use bilinear maps. We hope to reduce the proof size of the DVRF protocol in the future while still using the curve secp256k1 for implementation.
2. **Making the Protocol Post-quantum Secure.** This is an interesting direction, especially given the ongoing development of theoretical post-quantum cryptography. Considering that randomness is fundamental to most cryptographic protocols, ensuring the security of the randomness-generating process against threats like quantum computers is imperative. In [EKS⁺21], the authors constructed a VRF from the lattice, which is based on lattice assumptions that provide security against quantum computers. Currently,

the newest lattice-based VRF construction is in [ESLR22]. Hence, it is desirable to use the framework of the DVRF protocol to construct a post-quantum secure DRNG using these VRFs above.

3. **Universal Composable Security.** As described in Section 3.1.2, it is possible to use the Universal Composable Security framework (UC) of Ran Canneti to capture the execution and security properties of DRNGs. UC framework realizes the execution of protocols better. However, the framework is also surprisingly hard to design and explain carefully. As a result, we were unable to model and establish the security of the DVRF protocol within the UC framework. Notably, ALBATROSS [CD20] endeavored to define the ideal functionality $\mathcal{F}_{\text{CT}}^{k,\mathcal{D}}$ for coin tossing protocols within a single epoch, where the outputs of honest participants are uniformly distributed in \mathcal{D} . Nevertheless, it was tailored primarily for PVSS-based DRNGs, relying on participant computations via coin tossing, independent of prior epochs. This framework didn't encompass the broader framework of executing the DVRF protocol, characterized by deterministic computations influenced by previous epochs. Consequently, we hope to provide the ideal functionality $\mathcal{F}_{\text{DRNG}}$ for all DRNG executions and prove that the DVRF protocol is secure using the UC framework in the future.

Bibliography

- [AAEHR22] Sherif H AbdElHaleem, Salwa K Abd-El-Hafiz, and Ahmed G Radwan. A generalized framework for elliptic curves based prng and its utilization in image encryption. *Scientific reports*, 12(1):1–16, 2022.
- [Ba117] *Mastering Blockchain*. Packt Publishing, 2017.
- [BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *CRYPTO 2018*, volume 10991 of *LNCS*, pages 757–788. Springer, 2018.
- [Bit20] Nir Bitansky. Verifiable random functions from non-interactive witness-indistinguishable proofs. *J. Cryptol.*, 33(2):459–493, 2020.
- [BKP11] Michael Backes, Aniket Kate, and Arpita Patra. Computational verifiable secret sharing revisited. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *LNCS*, pages 590–609. Springer, 2011.
- [BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptol.*, 17(4):297–319, 2004.
- [Blu83] Manuel Blum. Coin flipping by telephone a protocol for solving impossible problems. *SIGACT News*, 15(1):23–27, 1983.
- [BMR10] Dan Boneh, Hart William Montgomery, and Ananth Raghunathan. Algebraic pseudorandom functions with improved efficiency from the augmented cascade. In *ACM CCS 2010*, pages 131–140. ACM, 2010.

- [BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993*, pages 62–73. ACM, 1993.
- [BSL⁺21] Adithya Bhat, Nibesh Shrestha, Zhongtang Luo, Aniket Kate, and Kartik Nayak. Randpiper - reconfiguration-friendly random beacons with quadratic communication. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 3502–3524. ACM, 2021.
- [Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001.
- [CATB23] Kevin Choi, Arasu Arun, Nirvan Tyagi, and Joseph Bonneau. Bicorn: An optimistically efficient distributed randomness beacon. *IACR Cryptol. ePrint Arch.*, page 221, 2023.
- [CD17] Ignacio Cascudo and Bernardo David. SCRAPE: scalable randomness attested by public entities. In *ACNS 2017*, volume 10355 of *LNCS*, pages 537–556. Springer, 2017.
- [CD20] Ignacio Cascudo and Bernardo David. ALBATROSS: publicly attestable batched randomness based on secret sharing. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part III*, volume 12493 of *Lecture Notes in Computer Science*, pages 311–341. Springer, 2020.
- [CGJ⁺99] Ran Canetti, Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Adaptive security for threshold cryptosystems. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology*

Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings, volume 1666 of *Lecture Notes in Computer Science*, pages 98–115. Springer, 1999.

- [CGMA85] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults (extended abstract). In *FOCS 1985*, pages 383–395. IEEE Computer Society, 1985.
- [CKS05] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *J. Cryptol.*, 18(3):219–246, 2005.
- [CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.
- [Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 364–369. ACM, 1986.
- [CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.
- [CMB23] Kevin Choi, Aathira Manoj, and Joseph Bonneau. Sok: Distributed randomness beacons. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 75–92. IEEE, 2023.
- [Cor00] Jean-Sébastien Coron. On the exact security of full domain hash. In Mihir Bellare, editor, *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, volume 1880 of *Lecture Notes in Computer Science*, pages 229–235. Springer, 2000.
- [CP92] David Chaum and Torben P. Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August*

- 16-20, 1992, *Proceedings*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 1992.
- [CSS19] Alisa Cherniaeva, Ilia Shirobokov, and Omer Shlomovits. Homomorphic encryption random beacon. *IACR Cryptol. ePrint Arch.*, page 1320, 2019.
- [DGKR18] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *EUROCRYPT 2018*, volume 10821 of *LNCS*, pages 66–98. Springer, 2018.
- [DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.
- [DKIR22] Sourav Das, Vinith Krishnan, Irene Miriam Isaac, and Ling Ren. Spurt: Scalable distributed randomness beacon with transparent setup. In *IEEE S&P 2022*, pages 2502–2517. IEEE, 2022.
- [DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.
- [DM17] Dilip Kumar Yadav Dindayal Mahto. Rsa and ecc: A comparative analysis. *International Journal of Applied Engineering Research*, 12(19):9053–9061, 2017.
- [DR85] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, 1985.
- [Dra66] Justin Drake. Minimal vdf randomness beacon, 2018, URL: <https://ethresear.ch/t/minimal-vdf-randomness-beacon/3566>.
- [Drave] Drand.love. Drand - distributed randomness beacon, 2020, URL: <https://drand.love>.
- [DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *PKC 2005*, volume 3386 of *LNCS*, pages 416–431. Springer, 2005.
- [EKS⁺21] Muhammed F. Esgin, Veronika Kuchta, Amin Sakzad, Ron Steinfeld, Zhenfei Zhang, Shifeng Sun, and Shumo Chu. Practical post-quantum few-time verifiable

- random function with applications to algorand. In Nikita Borisov and Claudia Díaz, editors, *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part II*, volume 12675 of *Lecture Notes in Computer Science*, pages 560–578. Springer, 2021.
- [ESLR22] Muhammed F. Esgin, Ron Steinfeld, Dongxi Liu, and Sushmita Ruj. Efficient hybrid exact/relaxed lattice proofs and applications to rounding and vrfs. *IACR Cryptol. ePrint Arch.*, page 141, 2022.
- [Fel87] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 427–437. IEEE Computer Society, 1987.
- [FR97] Benjamin Fine and Gerhard Rosenberger. *The fundamental theorem of algebra*. Springer Science & Business Media, 1997.
- [FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.
- [Gam85] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory*, 31(4):469–472, 1985.
- [Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, USA, 2009.
- [GG19] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. *IACR Cryptol. ePrint Arch.*, page 114, 2019.
- [GHKW17] Rishab Goyal, Susan Hohenberger, Venkata Koppula, and Brent Waters. A generic approach to constructing and proving verifiable random functions. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15,*

2017, *Proceedings, Part II*, volume 10678 of *Lecture Notes in Computer Science*, pages 537–566. Springer, 2017.

- [GHM⁺17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68. ACM, 2017.
- [GJKR99] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *EUROCRYPT 1999*, volume 1592 of *LNCS*, pages 295–310. Springer, 1999.
- [GLOW21] David Galindo, Jia Liu, Mihai Ordean, and Jin-Mann Wong. Fully distributed verifiable random functions and their application to decentralised random beacons. In *IEEE EuroS&P 2021*, pages 88–102. IEEE, 2021.
- [GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In Robert Sedgewick, editor, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 291–304. ACM, 1985.
- [Gol01] Oded Goldreich. *The Foundations of Cryptography - Volume 1: Basic Techniques*. Cambridge University Press, 2001.
- [GSM18] Fuchun Guo, Willy Susilo, and Yi Mu. *Introduction to Security Reduction*. Springer, 2018.
- [HLY20] Runchao Han, Haoyu Lin, and Jiangshan Yu. Randchain: A scalable and fair decentralised randomness beacon. Cryptology ePrint Archive, Paper 2020/1033, 2020. <https://eprint.iacr.org/2020/1033>.
- [HMW18] Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018.
- [HW10] Susan Hohenberger and Brent Waters. Constructing verifiable random functions with large input spaces. In *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 656–672. Springer, 2010.

- [Jag15] Tibor Jager. Verifiable random functions from weaker assumptions. In *TCC2015*, volume 9015 of *LNCS*, pages 121–143. Springer, 2015.
- [KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.
- [KRDO17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 357–388. Springer, 2017.
- [Lin20] Yehuda Lindell. Secure multiparty computation (MPC). *IACR Cryptol. ePrint Arch.*, page 300, 2020.
- [LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
- [LW15] Arjen K. Lenstra and Benjamin Wesolowski. A random zoo: sloth, unicorn, and trx. *IACR Cryptol. ePrint Arch.*, page 366, 2015.
- [MRV99] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *FOCS 1999*, pages 120–130. IEEE Computer Society, 1999.
- [N9696] *Smart Contracts*. Unpublished manuscript, 1996.
- [Nak] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system.
- [NCA23] Minh Pham Nhat, Hiro Chiro, , and Khuong Nguyen An. Orand - a fast, publicly verifiable, scalable decentralized random number generator based on distributed verifiable random functions. In *Proceedings of The 10th International Symposium on Integrated Uncertainty in Knowledge Modelling and Decision Making, November 2-4, 2023 (Springer Lecture Notes in Artificial Intelligence series, Volume xxx), forthcoming, 14 page*. Springer, 2023.
- [NIS01] A statistical test suite for random and pseudorandom number generators for cryptographic applications, 2001.

- [NNL⁺19] Thanh Nguyen-Van, Tuan Nguyen-Anh, Tien-Dat Le, Minh-Phuoc Nguyen-Ho, Tuong Nguyen-Van, Nhat-Quang Le, and Khuong Nguyen-An. Scalable distributed random number generation based on homomorphic encryption. In *IEEE Blockchain 2019*, pages 572–579. IEEE, 2019.
- [Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.
- [Ped91] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO 1991*, volume 576 of *LNCS*, pages 129–140. Springer, 1991.
- [PWH⁺17] Dimitrios Papadopoulos, Duane Wessels, Shumon Huque, Moni Naor, Jan Včelák, Leonid Reyzin, and Sharon Goldberg. Making nsec5 practical for dnssec, 2017.
- [Ran17] Randao.org. Randao: Verifiable random number generation. *White Paper*. URL: https://www.randao.org/whitepaper/Randao_v0.85_en.pdf, 2017.
- [RG22] Mayank Raikwar and Danilo Gligoroski. Sok: Decentralized randomness beacon protocols. In Khoa Nguyen, Guomin Yang, Fuchun Guo, and Willy Susilo, editors, *Information Security and Privacy - 27th Australasian Conference, ACISP 2022, Wollongong, NSW, Australia, November 28-30, 2022, Proceedings*, volume 13494 of *Lecture Notes in Computer Science*, pages 420–446. Springer, 2022.
- [Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *J. Cryptol.*, 4(3):161–174, 1991.
- [Sch99] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 148–164, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
- [Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

- [SJH⁺21] Philipp Schindler, Aljosha Judmayer, Markus Hittmeir, Nicholas Stifter, and Edgar R. Weippl. Randrunner: Distributed randomness from trapdoor vdfs with strong uniqueness. In *NDSS 2020*. The Internet Society, 2021.
- [SJK⁺17] 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 *IEEE S&P 2017*, pages 444–460. IEEE Computer Society, 2017.
- [SJSW20] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar R. Weippl. Hydrand: Efficient continuous distributed randomness. In *IEEE S&P 2020*, pages 73–89. IEEE, 2020.
- [Sta96] Markus Stadler. Publicly verifiable secret sharing. In *EUROCRYPT 1996*, volume 1070 of *LNCS*, pages 190–199. Springer, 1996.
- [TCZ⁺20] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan-Gueta, and Srinivas Devadas. Towards scalable threshold cryptosystems. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 877–893. IEEE, 2020.
- [vTJ11] Henk C. A. van Tilborg and Sushil Jajodia, editors. *Encyclopedia of Cryptography and Security, 2nd Ed.* Springer, 2011.
- [Was03] Lawrence C. Washington. Elliptic curves: Number theory and cryptography. 2003.
- [Wes19] Benjamin Wesolowski. Efficient verifiable delay functions. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III*, volume 11478 of *Lecture Notes in Computer Science*, pages 379–407. Springer, 2019.
- [WN21] Gang Wang and Mark Nixon. Randchain: Practical scalable decentralized randomness attested by blockchain. *IACR Cryptol. ePrint Arch.*, page 450, 2021.