Practical Distributed Key Generation and Signatures

by

Chelsea Komlo

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2026

© Chelsea Komlo 2026

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Stanislaw Jarecki
Professor, School of Information and Computer Sciences,
University of California Irvine

Co-Supervisor(s): Ian Goldberg
Professor, School of Computer Science,
University of Waterloo

Co-Supervisor(s): Douglas Stebila
Associate Professor, Dept. of Combinatorics and Optimization,
University of Waterloo

Internal Member: Mohammad Hajiabadi
Assistant Professor, School of Computer Science,
University of Waterloo

Internal Member: Florian Kerschbaum
Professor, School of Computer Science,
University of Waterloo

Internal-External Member: Sam Jaques
Assistant Professor, Dept. of Combinatorics and Optimization,
University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

The contributions of Chapter 3 were previously published in collaboration with Ian Goldberg [KG20], with the exception of the direct proof for FROST to ℓ-AOMDL. The contributions of Chapter 4 were previously published in collaboration with Ian Goldberg and Douglas Stebila [KGS23]. The contributions of Chapter 5 were previously published in collaboration with Ian Goldberg [KG25]. The references were edited using Claude Code. All of the work appearing herein was conducted under the supervision of Ian Goldberg and Douglas Stebila.

Abstract

Threshold schemes are a critical cryptographic primitive that allows a set of n total parties and a threshold of at least t parties to collaborate to jointly perform some function, such as generating key material or issuing a digital signature. Threshold schemes allow for improved robustness in the case of failure, and distribute trust among many parties. In particular, the security of the scheme assumes $t - 1$ players are corrupted, and so can deviate arbitrarily from the protocol. The security of the scheme ensures that in spite of a subset of corrupted players, the scheme can provide important properties such as robustness, unforgeability, or indistiguishability from some target distribution.

In this work, we examine the special cases of *distributed key generation* and *threshold signing*. In particular, we present constructions that optimize for considerations that are important to implementations in practice. Such considerations include simplicity, network round efficiency, computational and bandwidth efficiency, and low use of broadcast channels.

Firstly, we present FROST, a Flexible Round-Optimized Schnorr Threshold signature scheme. FROST improves upon prior threshold Schnorr signature schemes in that signatures can be generated with only two network rounds among participants, while remaining secure against concurrent adversaries. We show that FROST is secure under the Algebraic One-More Discrete Logarithm (ℓ-AOMDL) assumption in the Random Oracle Model (ROM).

Secondly, we present Storm, a simplified three-round distributed key generation protocol (DKG). Storm presents a simplified alternative to prior DKGs with a similar security model, assuming the Discrete Logarithm Problem (DLP) is hard, and provides a generic construction that may be applicable beyond discrete-logarithm assumptions.

Finally, we present Arctic, a two-round deterministic threshold Schnorr signature scheme. Arctic allows signers to remain stateless, with the exception of persisting state of their long-lived signing keys. Arctic requires a slightly weaker trust model in that it assumes the majority of signers are honest, but shows improved efficiency over alternative deterministic threshold Schnorr signature schemes for small signing coalitions (fewer than 25 signers). We show that Arctic is secure assuming DLP in the ROM.

Acknowledgements

I would like to first thank my co-advisors Ian Goldberg and Douglas Stebila, for their incredible support and kindness. I feel incredibly lucky to have worked with advisors that go over and beyond for their students, and hope to emulate their leadership and contributions to the field in my own career in the future.

I also want to extend a thank you to Dan Boneh, whom I was lucky enough to collaborate with over the years. Dan is a true role model, his enthusiasm and kindness set an example for leaderhsip within our field.

Finally, I would like to thank my family and partner, for always believing in me.

Table of Contents

Examining Committee Membership ii

Author's Declaration iii

Statement of Contributions iv

Abstract v

Acknowledgements vi

List of Figures xi

List of Tables xiii

1 Introduction 1

1.1 Overview of Results . . . 3

1.1.1 Chapter 3: An Efficient Randomized Threshold Schnorr Signature Scheme . . . 3

1.1.2 Chapter 4: A Generic Distributed Key Generation (DKG) Construction . . . 5

1.1.3 Chapter 5: An Efficient Deterministic Threshold Schnorr Signature Scheme . . . 5

1.2 Impact in Practice . . . 6

2 Background **8**

2.1 General Background 8

2.1.1 Notation 8

2.1.2 Assumptions 9

2.2 General Forking Lemma 13

2.2.1 A Modified Forking Lemma 14

2.3 Digital Signatures and Schnorr Signatures 15

2.3.1 Schnorr Signatures 16

2.4 Non-Interactive Key Exchange (NIKE) 17

2.4.1 A Concrete NIKE 19

2.5 Threshold Secret Sharing Schemes 20

2.5.1 Shamir Secret Sharing 21

2.5.2 Verifiable Secret Sharing 22

2.6 Threshold Signatures 26

2.6.1 Static Security 28

2.6.2 Adaptive Security 31

2.7 ROS Attacks 32

2.8 Network Communication Channels 32

2.9 Additional Roles and Adversaries 33

3 FROST: A Flexible, Round-Optimized, Schnorr Threshold Signature Scheme **35**

3.1 Related Work 37

3.1.1 Prior Related Work 37

3.1.2 Subsequent Work 39

3.2 FROST Signatures 40

3.2.1 Key Generation 40

3.2.2 The FROST Signature Scheme 42

3.2.3 Correctness of FROST . . . 44
3.3 Security of FROST . . . 45
3.4 Implementation and Operational Considerations . . . 54

4 A Formal Treatment of Distributed Key Generation, and New Constructions 55
4.1 Our Contributions . . . 56
4.2 Related Work . . . 60
4.3 Aggregatable Verifiable Secret Sharing (AgVSS) . . . 62
4.3.1 A Discrete-Log Based AgVSS . . . 64
4.4 Distributed Key Generation (DKG) . . . 72
4.4.1 Robustness . . . 74
4.4.2 Zero Knowledge . . . 79
4.4.3 Indistinguishability . . . 80
4.5 A Generic DKG Construction . . . 82
4.5.1 Key Generation Protocol . . . 85
4.5.2 Recovery Algorithm . . . 87
4.5.3 Security of the Generic Protocol . . . 87
4.6 Storm, a Discrete-Logarithm Based DKG . . . 97
4.7 Future Work and Composability . . . 98

5 Arctic: Lightweight and Stateless Threshold Schnorr Signatures 99
5.1 Our Results . . . 101
5.2 Observations of Honest Majority Assumptions in Practice . . . 105
5.3 Related Work . . . 105
5.4 Deterministic Threshold Signature Schemes . . . 107
5.4.1 Unforgeability . . . 110
5.5 Verifiable Pseudorandom Secret Sharing . . . 111

5.5.1 Motivation . . . 111
5.5.2 VPSS Definition and Notions of Security . . . 112
5.5.3 Shine, A Concrete Verifiable Pseudorandom Secret Sharing Scheme 117
5.6 Arctic, A Deterministic and Stateless Two-Round Threshold Schnorr Signature Scheme . . . 125
5.6.1 The Construction . . . 127
5.6.2 Possible Extensions . . . 128
5.6.3 Correctness and Security . . . 129
5.7 Extending Shine and Arctic to be Robust . . . 143
5.7.1 Communication Model for Robust Arctic . . . 145
5.8 Performance Analysis of Arctic . . . 145
5.8.1 Comparison to Related Work . . . 147

6 Conclusion **149**
6.1 Future Work . . . 150
6.2 Reflections . . . 150

References **152**

List of Figures

2.1 The Discrete Logarithm (DL), One-More Discrete Logarithm (ℓ-OMDL), and Algebraic One-More Discrete Logarithm (ℓ-AOMDL) games. 10

2.2 The general forking algorithm $\mathsf{Fork}^{\mathcal{C}}(X)$ and the modified general forking algorithm $\mathsf{Fork}^{\mathcal{C}}_{m}(X)$. 13

2.3 The EUF-CMA security game for a digital signature scheme $\mathcal{DS}$. 15

2.4 Unrecoverability experiment for NIKE 18

2.5 Binding experiment defining the advantage of an adversary $\mathcal{A}$ against a non-interactive key exchange (NIKE) NK. 19

2.6 Secrecy experiment defining the advantage of an adversary $\mathcal{A}$ against a Verifiable Secret Sharing scheme (VSS) S that is zero-indexed. 24

2.7 Uniqueness experiment defining the advantage of an adversary $\mathcal{A}$ against a Verifiable Secret Sharing scheme (VSS) S. 25

2.8 Static unforgeability game for a threshold signature scheme. 29

3.1 The FROST threshold signature scheme 41

4.1 Notion of correctness for an AgVSS. 63

4.2 Aggregated secrecy experiment for a zero-indexed Aggregatable Verifiable Secret Sharing scheme (AgVSS). 65

4.3 Algorithms to simulate secret sharing and tweak generation in an AgVSS to an adversary. 68

4.4 Robustness experiment for a DKG. 76

4.5 Common oracles for DKG security experiemnts. 77

4.6 Zero-knowledge experiment for a DKG. 80
4.7 Indistinguishability experiment for a DKG. 83
4.8 A generic DKG protocol. See corresponding text for full description. 84
4.9 Game 2 for the Zero Knowledge Proof for Generic Construction. 94

5.1 Unforgeability experiment for a deterministic threshold signature scheme with a κ-round signing protocol. 109
5.2 Uniqueness game for a VPSS. 114
5.3 Verifiability game for a VPSS. Note the environment enforces the authenticity of participant contributions. 115
5.4 Pseudorandomness game for a VPSS. 116
5.5 Simulating algorithm SimGen for Shine. 123
5.6 Arctic, a deterministic threshold Schnorr signature scheme. 126
5.7 Single-core wall clock time for for Arctic. 146
5.8 Scaling experiment. 147

List of Tables

4.1 Comparison of DKG Constructions 57

5.1 Comparison of multi-party deterministic Schnorr signature schemes in the ROM that support general choices of n. 103

Chapter 1

Introduction

While the goal of cryptography can be described in many ways, one common motivation is often *authorization* within a digital environment; i.e., the assurance that only a designated party is allowed to perform some action. The goal of authorization is generally accomplished by granting the authorized party *knowledge* of a secret, and then later challenging the party to demonstrate knowledge of this secret value. This intuition describes the strategy underlying interactive identification protocols and digital signature schemes–common cryptographic building blocks in secure protocols.

A challenge emerges when considering the possibility that this authorized party may later become compromised by an adversarial party, or might simply fail. For example, a database holding a secret value might become corrupted. One real-world example is that of certificate authorities, which have been long-documented to perform untrusted actions [SSL25] In this setting, even if the designated party correctly demonstrates knowledge of a secret key, the party may still violate trust.

As cryptographers, the question then emerges as whether new cryptographic primitives can be designed to better reflect the trust required in a real-world setting. For this particular example, instead of distributing one secret value to authorize a designated party, one might ask whether is it possible to instead designate multiple parties that must cooperate to perform a trusted action.

In this dissertation, we address this question by investigating the real-world utility of *threshold* schemes. Threshold schemes are a critical cryptographic primitive that requires at least a threshold t parties out of a set of n total parties to collaborate to perform some function. This function depends on the use case. For example, the parties might perform threshold signing [KG20, BCK+22, Lin24], outputting a joint signature at the end of the

protocol. Alternatively, participants might participate to jointly generate cryptographic key material [Ped91, GJKR07, KG20, Gro21, GJM+21, BS20, NSR16, Lin24, DYX+22, BL22, AJM+22]. This key may then be used within the context of other threshold schemes, such as evaluating a pseudorandom function (PRF) in a threshold manner [JKR18].

Threshold schemes are an appealing primitive due to their ability to ensure robustness in the case of failure, and to distribute trust among many parties. In particular, the security of the scheme assumes $t-1$ players are corrupted, and so can deviate arbitrarily from the protocol. The security of the scheme ensures that in spite of a subset of corrupted players, the scheme remains secure. As such, threshold schemes offer practical security benefits, allowing for dynamic key management and defense in depth against potential adversarial corruptions. While some threshold schemes can be instantiated by each party simply performing a single-party protocol in parallel, a practical goal for many threshold schemes is to preserve compatibility with existing single-party schemes, to minimize implementation complexity and to ensure efficiency.

Motivation. Interest in threshold schemes is growing in practice. For example, as of early 2025, NIST is expected to issue a call for threshold schemes ranging from distributed key generation to threshold zero-knowledge proofs [BD22, BP23]. The NIST call specifically requests threshold variants of single-party signature schemes, including Schnorr signatures [Sch91], which is a core focus of this thesis.

However, significant challenges arise when using threshold schemes in practice, such as gracefully handling failure cases, performing multi-party operations at scale, and operating in a setting where signers are not guaranteed to always be online. Furthermore, multi-party, multi-round schemes such as threshold signatures introduce significant complexity and performance costs over single-party signature schemes. For this reason, schemes that are simple to implement, provide defense-in-depth measures, and incur low round complexity are attractive in practice. However, such schemes are difficult to prove secure using standard security assumptions. For example, for discrete logarithm-based threshold signatures, the most efficient two-round schemes require the one-more discrete logarithm assumption to demonstrate unforgeability [KG20, BCK+22], whereas less-efficient three-round schemes can be proven secure using the discrete logarithm assumption directly [CKM23, BCK+22, Lin24].

Problem to Overcome. While threshold schemes offer benefits such as redundancy and increasing the cost to a potential adversary, the downside to threshold schemes in practice is their complexity and performance overhead. Multi-round, multi-entity protocols

incur additional performance and complexity challenges beyond the single-party setting. These challenges are compounded when considering real-world failures, such as network failures and timeouts. As such, even though efficient threshold schemes might possibly incur stronger security assumptions, these tradeoffs are often practically worthwhile when considering improvements in reduced complexity and increased performance.

With the goals of efficiency, simplicity, and security in mind, in this work, we investigate how to design efficient threshold schemes, and propose a range of constructions that provide improved efficiency over prior works in the literature. In particular, our thesis statement is as follows:

> *It is possible to construct efficient threshold key generation and signature schemes that are practically useful, with reasonable security assumptions. By practical utility, we refer to the properties that implementations often require, such as the simplicity of the scheme, low number of communication rounds among parties, and computational and bandwidth efficiency.*

1.1 Overview of Results

This dissertation presents new techniques to simplify and improve the practicality of threshold Schnorr signatures and discrete-logarithm based distributed key generation. Chapter 3 gives an efficient randomized threshold Schnorr signature scheme, Chapter 4 gives a generic construction for a distributed key generation (DKG) scheme. Chapter 5 gives an efficient deterministic threshold Schnorr signature scheme.

1.1.1 Chapter 3: An Efficient Randomized Threshold Schnorr Signature Scheme

The first contribution of this dissertation is to define FROST, a Flexible Round-Optimized Schnorr Threshold Signature scheme. FROST is a two-round randomized threshold Schnorr signature scheme, which can optionally be used in the preprocessing setting. FROST defines a multi-party signing protocol that outputs a signature that is verifiable under the single-party Schnorr signature verification algorithm [Sch91].

By saying that FROST can be used in the preprocessing setting, we mean that FROST can be used in the setting where the first round is performed in a batched manner. In

other words, performing the threshold signing protocol may require only a single online round of interaction among all parties, if the first round is performed in a batched manner by each signing party.

While threshold Schnorr signature schemes certainly existed in the literature prior to FROST, all prior schemes fall into one of two categories. Prior schemes either require more than two rounds of interaction among all parties [SS01], or are insecure in a concurrent setting [GJKR03] due to a polynomial-time forgery attack [DEF+19, BLL+21].

One might ask why limiting the number of communication rounds when performing signing is in fact desirable. What we have learned from practitioners is that round efficiency is highly desirable for implementations in practice, due to the complexities that arise when coordinating communication rounds among multiple parties. In the setting where parties could fail to reply or respond in a delayed manner, managing timeouts and session state across concurrent signing sessions becomes highly complex when signing is performed at scale.

We note that FROST achieves improved round efficiency in the optimistic case that no participant fails to respond or misbehaves. However, in the case where a participant contributes malformed values during the protocol or simply fails to respond, honest parties can identify and exclude the misbehaving participant, and then re-run the protocol. In other words, FROST does allow for some form of identifiable abort, but is not robust, in that any deviation from the protocol requires aborting and re-running.

We prove the security of FROST with respect to the Algebraic One-More Discrete Logarithm (ℓ-AOMDL) assumption, formalized by Nick et al. [NRS21]. which is a weaker variant of the One-More Discrete Logarithm (ℓ-OMDL) assumption [BNPS03, BFP21]. While FROST was proven secure with respect to ℓ-OMDL [BCK+22], we additionally show how this proof extends to ℓ-AOMDL. ℓ-OMDL defines an interactive discrete-logarithm assumption, in which the adversary is given $\ell + 1$ discrete logarithm challenges, and is required to output $\ell + 1$ solutions, while being given ℓ queries to a discrete-logarithm solution oracle. ℓ-OMDL is a stronger security assumption, as the ℓ-OMDL oracle must output solutions to arbitrary group elements, and so the oracle is not guaranteed to run in polynomial time. ℓ-AOMDL additionally requires the adversary to give as input the representation of the input group element with respect to all challenge group elements. As such, the ℓ-AOMDL solution oracle is guaranteed to run in polynomial time, therefore making ℓ-AOMDL a weaker assumption than ℓ-OMDL.

1.1.2 Chapter 4: A Generic Distributed Key Generation (DKG) Construction

The second contribution of this dissertation is to define a simple generic distributed key generation (DKG) construction. While key generation for threshold signature schemes can be performed in a centralized manner using a trusted dealer, often distribution of trust is likewise required during the generation of key material. As such, DKG schemes allow participants to collaborate to generate asymmetric key material in such a way that the secret key is unknown and secret shared among all participants.

Our generic DKG construction relies on three cryptographic building blocks: a secure hash function, an *Aggregatable Verifiable Secret Sharing* (AgVSS) scheme, and a *Non-Interactive Key Exchange* (NIKE) scheme. We show how Feldman's VSS [Fel87] can be augmented to be used as an AgVSS.

The goal for presenting a generic DKG is to enable flexibility for implementations and standardization efforts. Future implementations may wish to opt for different building blocks, and so a generic construction allows for different choices while maintaining formal guarantees of security. Further, this flexibility may be worthwhile to ongoing NIST standardization efforts for threshold schemes [BP23]. The utility of a generic DKG is that any instantiation of the DKG will remain secure, so long as each underlying building block is secure.

We give a concrete construction of our generic DKG which we call Storm, which builds upon Feldman's VSS [Fel87] and a straightforward discrete-logarithm NIKE, which we define in Section 2.4.1.

1.1.3 Chapter 5: An Efficient Deterministic Threshold Schnorr Signature Scheme

The third contribution of this dissertation is to define a *deterministic* threshold Schnorr signature scheme. While FROST is desirable in practice, one downside is that participants cannot re-use secret state across signing sessions, as doing so will result in complete exposure of their secret signing key. In the real-world setting where machines can go offline and reboot without warning, such a strong requirement can prove challenging in practice. The goal of a deterministic threshold signature scheme is to protect against such cases, as participants are no longer required to maintain state when performing signing.

Towards this goal, we present Arctic, a deterministic and stateless threshold Schnorr signature scheme. The primary goal of Arctic is simplicity. While prior deterministic

threshold Schnorr signature schemes have been defined [NRSW20, GKMN21, KOR23] in the literature, these prior schemes rely on either generic multi-party computation techniques, or generic non-interactive zero-knowledge proof systems, and therefore incur complexity and performance overhead. The observation underlying Arctic is that in settings where the expected number of honest parties outnumbers the number of corrupt parties, we can define a simpler and more efficient deterministic threshold Schnorr signature scheme.

In particular, for moderately sized groups (i.e., when $n \leq 20$), Arctic is more than an order of magnitude more efficient than existing deterministic schemes in the literature [NRSW20, GKMN21, KOR23] Arctic is three orders of magnitude more efficient when $n \leq 10$. For larger groups, Arctic scales linearly relative to the number of processing cores. Importantly, signers in Arctic are required to generate and maintain secret keys, but otherwise do not need to manage secret state.

We prove the security of Arctic by showing a reduction to the Discrete-Logarithm (DL) problem in the Random Oracle Model (ROM). Showing a reduction to standard security assumptions in the ROM is highly desirable, as these assumptions are the same as is required for the security of the single-party Schnorr signature scheme [Sch91].

1.2 Impact in Practice

The main impact this thesis has had in practice is the implementation and standardization of FROST, which we now describe.

The Zcash Foundation has implemented FROST and incorporated FROST into the core Zcash protocol [Zca25, CG22]. FROST is used to partition a spending key for a Zcash wallet into multiple shares. As such, FROST allows a Zcash user to issue a transaction only after a threshold number of signing shares are used to produce a signature. FROST improves over prior schemes such as Bitcoin's multisig design [Bit24], in that a FROST signature does not leak the number of signers and hence avoids fingerprinting attacks on-chain. Bitcoin's multisig design, on the other hand, simply concatenates t signatures into one joint signature, where the public key is the set of n public keys.

FROST has been defined as an informational RFC for the Crypto Forum Research Group (CFRG) within the IRTF [CKGW24]. FROST is the first threshold signature scheme to be defined within an IRTF RFC. The scheme is defined to be agnostic to any particular choice of parameters, and supports a range of parameter choices. Furthermore, the RFC does not define how key generation is performed, and simply assumes that participants are initialized with secret signing shares that are Shamir secret shares of a signing

key. The RFC therefore is compatible with any key generation RFC that might be defined in the future which outputs keys of this form.

Other blockchains beyond Zcash have investigated the use of FROST to perform multi-party signing, most notably, Bitcoin. There is existing effort to define FROST as a Bitcoin Improvement Proposal (BIP) [Pos25] and build additional infrastructure with FROST in mind, including distributed key generation [RN25]. FROST is also used within the context of bridging mechanisms, allowing funds to flow from Bitcoin to other blockchain networks [Arc25]. The security of FROST has also been discussed within the Ethereum community [Kom22].

There is additionally a team working to define a FROST submission for the NIST threshold signature scheme call for proposals [BD22, BP23]. Portions of this dissertation will be employed towards this effort; namely, the proof for ℓ-AOMDL security of FROST, which is a weaker assumption than ℓ-OMDL, which prior proofs of security rely upon [BCK+22]. We hope that the submission of FROST to the NIST call for threshold schemes allows for more widespread adoption of threshold signatures beyond the cryptocurrency space, into domains such as banking and digital certificate management.

Chapter 2

Background

In this chapter, we review the relevant background necessary for the remainder of this thesis.

2.1 General Background

In this section, we review general background, including notation, cryptographic building blocks, notions of security, and security assumptions.

2.1.1 Notation

We represent a security parameter as $\lambda \in \mathbb{Z}$. We denote the assignment of an element y to the value x as $y \leftarrow x$, and sampling an element from some set S uniformly at random as $x \xleftarrow{\$} S$. For a randomized algorithm A, we write $x \xleftarrow{\$} A()$ to indicate the random variable x that is output from the execution of A. We say that f is a negligible function of λ if it holds that $\lim_{\lambda \to \infty} f(\lambda)\lambda^c = 0$, for all $c > 0$ [BS20].

Let $\mathbb{G}$ be a cyclic group of prime order q, and $\mathbb{Z}_q$ be the field of integers modulo q. Let g be a generator of $\mathbb{G}$.

We use $[n]$ to represent the set $\{1, \ldots, n\}$, and represent integer vectors in bold-face font. For a set S, we denote $\binom{S}{t}$ to mean the set that consists of all size-t subsets of S.

Polynomial Interpolation. A polynomial $f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{t-1} x^{t-1}$ of degree $t-1$ over a field $\mathbb{F}$ can be interpolated by t points. Let η be a set of t distinct integers corresponding to the x-coordinates $x_i \in \mathbb{F}, i \in \eta$, of these points. Then the Lagrange polynomial $L_i(x)$ has the form:

$$L_i(x) = \prod_{j \in \eta; j \neq i} \frac{x - x_j}{x_i - x_j} \tag{2.1}$$

Given a set of t points $(x_i, f(x_i))_{i \in [\eta]}$, any point $f(x_\ell)$ on the polynomial f can be determined by Lagrange interpolation as follows:

$$f(x_\ell) = \sum_{k \in \eta} f(x_k) \cdot L_k(x_\ell)$$

Game-Based Security. An experiment (or game) $\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{C},\mathcal{A}}$ is played with respect to a security notion sec, an adversary $\mathcal{A}$ modeled by a probabilistic (randomized) algorithm, and a cryptographic primitive $\mathcal{C}$. The experiment is initialized within a main procedure, whose output is the output of the game. The goal of $\mathcal{A}$ is to win the game, the conditions under which is defined within the experiment. The probability of $\mathcal{A}$'s success thus defines how secure $\mathcal{C}$ is with respect to the security notion sec.

PPT Adversaries. We say that an adversary $\mathcal{A}$ is probabilistic polynomial-time (PPT) if it is run on random coins and its execution is performed in time that is polynomial relative to the size of its input. In other words, there exists a polynomial f, such that for every input $x \in \{0,1\}^n$, $\mathcal{A}(x)$ terminates within at most $f(n)$ steps.

Group Generation. Let $\mathsf{GroupGen}$ be a probabilistic polynomial-time algorithm that takes as input a security parameter 1^λ and outputs a group description $(\mathbb{G}, q, g)$ consisting of a group $\mathbb{G}$ of order q, where q is a 2λ-bit prime, and a generator g of $\mathbb{G}$.

2.1.2 Assumptions

We next review the various cryptographic assumptions that we will need in the remainder of this thesis.

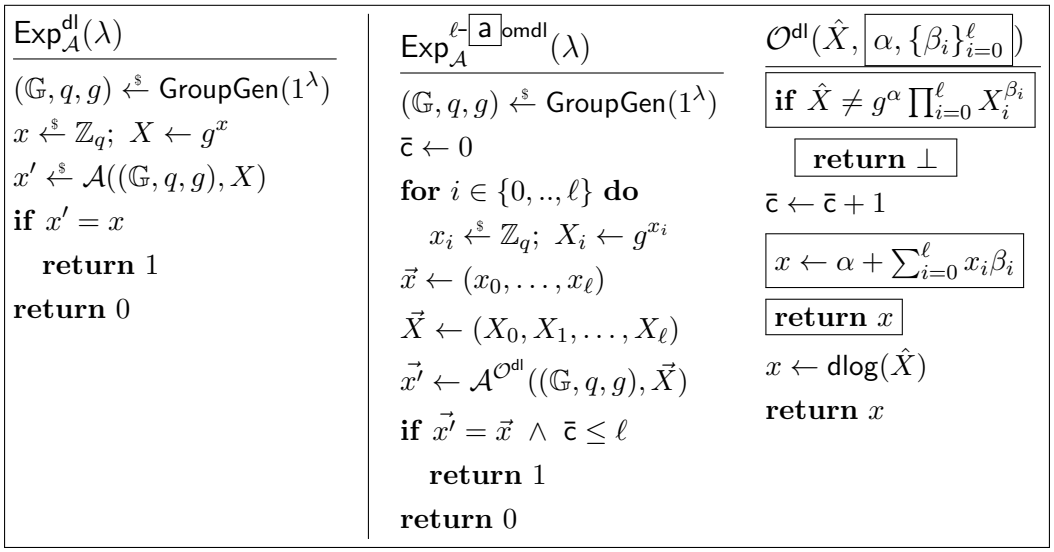


Figure 2.1: The Discrete Logarithm (DL), One-More Discrete Logarithm (ℓ-OMDL), and Algebraic One-More Discrete Logarithm (ℓ-AOMDL) games. The boxes indicate the differences in the ℓ-AOMDL game to the ℓ-OMDL game. ℓ is the bound on the number of times the adversary $\mathcal{A}$ is allowed to query the DL oracle.

Discrete Logarithm (DL) Assumption. The DL assumption is shown in Figure 2.1. $\mathbb{G}$ is a cyclic group with prime order q and generator $g \in \mathbb{G}$; these parameters are generated by the algorithm $(\mathbb{G}, q, g) \xleftarrow{\$} \mathsf{GroupGen}(1^\lambda)$, which accepts as input the security parameter λ.

To begin, a PPT adversary $\mathcal{A}$ is given a value $X \in \mathbb{G}$ such that $X = g^x$ without knowing the value of x. The challenge to the adversary is to output x.

We say that the DL assumption holds if the advantage to a PPT adversary in successfully producing such an x is a negligible function of λ.

Computational Diffie-Hellman Assumption (CDH). The Computational Diffie-Hellman Assumption (CDH) can be described as follows. To begin, public parameters are generated by the algorithm $(\mathbb{G}, q, g) \xleftarrow{\$} \mathsf{GroupGen}(1^\lambda)$, where $\mathbb{G}$ is a cyclic group with prime order q and generator $g \in \mathbb{G}$.

A PPT adversary $\mathcal{A}$ is then given the following values:

$$(g, g^a, g^b)$$

The challenge to the adversary is then to compute g^{ab}. The CDH assumption holds if the advantage to any PPT adversary in successfully doing so is a negligible function of λ.

One-More Discrete Logarithm Assumption (ℓ-OMDL). The One-More Discrete Logarithm (ℓ-OMDL) assumption [BNPS03, BFP21] extends the DL assumption, which is non-interactive, into an interactive assumption. We show the ℓ-OMDL assumption along with the DL assumption in Figure 2.1. We denote the bound on the number of times the adversary $\mathcal{A}$ is allowed to query its DL oracle by ℓ.

The differences between the ℓ-OMDL assumption and the Algebraic One-More Discrete Logarithm (ℓ-AOMDL) assumption experiments are highlighted in a box; the key distinction is that in the ℓ-AOMDL game, the adversary can only query for linear combinations of its challenge group elements; in the ℓ-OMDL game, the environment must return the discrete logarithm of an arbitrary group element. It does so by the algorithm dlog, which is an algorithm that finds the discrete logarithm relation of the input $X \in \mathbb{G}$ and g.

To begin in the ℓ-OMDL experiment, public parameters are generated by the algorithm $(\mathbb{G}, q, g) \xleftarrow{\$} \mathsf{GroupGen}(1^\lambda)$, where $\mathbb{G}$ is a cyclic group with prime order q and generator $g \in \mathbb{G}$. Then, a PPT adversary $\mathcal{A}$ is given $\ell+1$ discrete logarithm challenges $(X_0, \ldots, X_\ell) \in \mathbb{G}^{\ell+1}$ with respect to g. It is also given access to a discrete logarithm oracle, that when queried on an arbitrary group element $\hat{X} \in \mathbb{G}$, returns the discrete logarithm of that value. The adversary is said to win the ℓ-OMDL game if it makes ℓ or fewer queries to the discrete logarithm oracle, and produces correct solutions $(x_0, \ldots, x_\ell)$ to its $\ell + 1$ challenges such that $X_i = g^{x_i}$, $i \in \{0, \ldots, \ell\}$.

The ℓ-OMDL assumption holds if the advantage to any PPT adversary in winning the ℓ-OMDL game is a negligible function of λ.

Algebraic One-More Discrete Logarithm Assumption (ℓ-AOMDL). The algebraic one-more discrete logarithm assumption (ℓ-AOMDL) was formalized by Jonas, Ruffing, and Seurin [NRS21]. We highlight differences with the ℓ-OMDL assumption in Figure 2.1. The ℓ-AOMDL assumption holds if the advantage to any PPT adversary in winning the ℓ-AOMDL game is a negligible function of λ.

Note that while the ℓ-OMDL assumption is not falsifiable (because the adversary is allowed to query for the discrete logarithm of any arbitrary group element), the ℓ-AOMDL assumption is falsifiable. This is because the input to the discrete logarithm oracle is a linear combination of all challenges issued to the adversary, so it is guaranteed that

the environment runs in polynomial time with respect to a PPT adversary. Hence, the ℓ-AOMDL assumption is a strictly weaker assumption than standard ℓ-OMDL.

An adversary in the ℓ-AOMDL game is initialized in exactly the same manner as an ℓ-OMDL adversary, with the same winning condition. The only distinction between the games is the inputs to the discrete logarithm oracle. In the ℓ-AOMDL setting, the adversary provides a vector $(\alpha, \beta_0 \ldots, \beta_\ell)$ of coefficients to the discrete logarithm oracle. The oracle then responds with the integer linear combination of these coefficients with respect to the discrete logarithm of the set of challenges $(X_0, \ldots, X_\ell)$. Informally, an ℓ-AOMDL adversary can be used to win the ℓ-OMDL game, therefore implying that if ℓ-OMDL is hard, then ℓ-AOMDL is hard.

Algebraic Group Model (AGM). We next introduce the Algebraic Group Model (AGM). The AGM is relevant when discussing related work and prior proofs of security for FROST, as presented in Chapter 3.

Introduced by Fuchsbauer, Kiltz, and Loss [FKL18], the Algebraic Group Model (AGM) gives an idealized representation of a group, by requiring the adversary to act algebraically. In other words, when an adversary outputs a group element $X \in \mathbb{G}$ to the environment, it additionally is required to output its representation with respect to all other group elements the adversary has seen thus far. In other words, the adversary must output a coefficient vector $\mathbf{v} = (\alpha, b_1, b_2, \ldots)$, such that $X = g^\alpha A_1^{b_1} A_2^{b_2} \ldots$.

Knowledge of Exponent Assumption (KoE). We next introduce the Knowledge of Exponent (KoE) assumption. KoE is employed in related work to Storm, which we discuss in Chapter 4.

The KoE assumption says that for an adversary that is given input $(g, h = g^s, q)$, if the adversary outputs a value $(C, Y) \in \mathbb{G}^2$ such that $C^s = Y$, then there exists an extractor algorithm that given $(g, h = g^s, q)$, the extractor can output x such that $C = g^x$ (and, consequently $Y = C^x$). Intuitively, KoE assumes that in order to output such a (C, Y), the adversary must know x.

The Knowledge of Exponent (KoE) assumption [Dam91, BP04] is an unfalsifiable assumption, as it is impossible to show that such an extractor must exist. However, this assumption is useful to prove security for both SNARKs [Gro10] and threshold schemes [BCK+22].

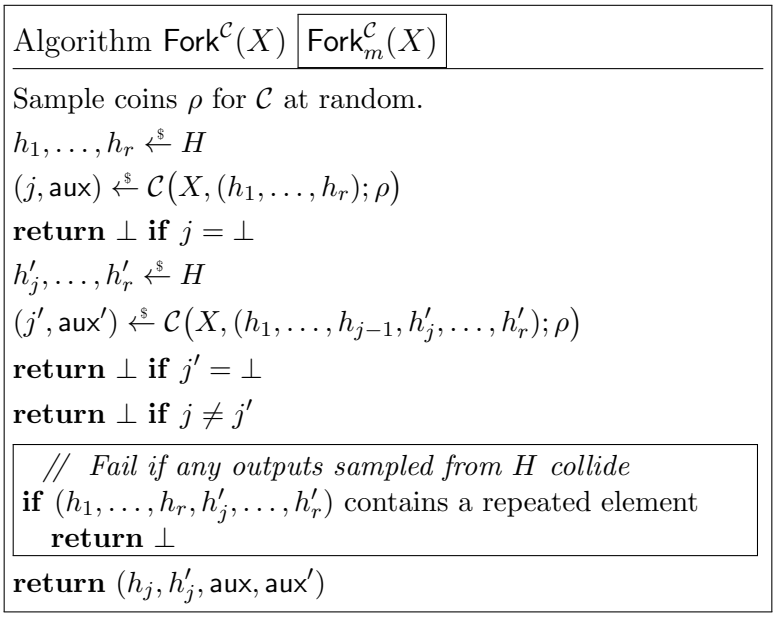


Figure 2.2: The general forking algorithm $\mathsf{Fork}^{\mathcal{C}}(X)$ and the modified general forking algorithm $\mathsf{Fork}^{\mathcal{C}}_m(X)$.

2.2 General Forking Lemma

We review the general forking lemma by Bellare and Neven [BN06], which itself is a formalization of the forking lemma introduced by Pointcheval and Stern [PS96]. The forking lemma is a useful tool to proving the security of signature schemes in the random oracle model (ROM). We use the forking lemma to prove the security of FROST in the ROM in Chapter 3, and Storm in Chapter 4.

We show the general forking algorithm in Figure 2.2, as well as a slight modification, which we describe in Section 2.2.1.

The algorithms are defined with respect to an algorithm $\mathcal{C}$ and instance X. The difference between the general forking algorithm and modified variant is shown in a box, for emphasis. In summary, the modified general forking algorithm aborts if *any* of the $(h_1, \ldots, h_r, h'_j, \ldots, h'_r)$ collide. The modified variant allows us to account for hash collisions in a generalized manner, as the forking algorithm aborts on any hash collision.

Lemma 1 (General Forking Lemma [BN06])**.** *Let H be a finite set and $r \geq 1$ be an integer. Let IG be a randomized instance generator and let $X \xleftarrow{\$} \mathsf{IG}$ be an instance. Let $\mathcal{C}$*

be a randomized algorithm that takes as input X, quantities $h_1, \ldots, h_r \in H$, and a random tape ρ. Let $\mathsf{accept}(\mathcal{C})$ *be the probability that* $\mathcal{C}$ *outputs an accepting answer, namely*

$$\mathsf{accept}(\mathcal{C}) := \Pr\left[j \neq \perp \ : \ \begin{array}{c} X \xleftarrow{\$} \mathsf{IG}, \quad h_1, \ldots, h_r \xleftarrow{\$} H \\ (j, \mathsf{aux}) \xleftarrow{\$} \mathcal{C}\big(X, (h_1, \ldots, h_r); \rho\big) \end{array} \right].$$

Let $\mathsf{Fork}^{\mathcal{C}}(X)$ *be the general forking algorithm defined in Figure 2.2 and let*

$$\mathsf{accept}(\mathsf{Fork}^{\mathcal{C}}) := \Pr\left[\alpha \neq \perp \ : \ X \xleftarrow{\$} \mathsf{IG}, \quad \alpha \xleftarrow{\$} \mathsf{Fork}^{\mathcal{C}}(X)\right].$$

Then, $\mathsf{accept}(\mathsf{Fork}^{\mathcal{C}})$ *is bounded by*

$$\mathsf{accept}(\mathsf{Fork}^{\mathcal{C}}) \geq \mathsf{accept}(\mathcal{C}) \cdot \left(\frac{\mathsf{accept}(\mathcal{C})}{r} - \frac{1}{|H|} \right).$$

2.2.1 A Modified Forking Lemma

We will employ a slight modification of the forking experiment, and give a corollary on how it impacts the accepting probability of its output.

The modification to the forking experiment is natural; intuitively, we add an additional abort condition if any of the values $h_1, \ldots, h_r, h'_j, \ldots, h'_r$ collide. Because there are at most $2r$ values, and they are all sampled uniformly at random from the set H, the probability that any of them collide is at most $2r^2/|H|$. By considering this case here, we can be sure that such collisions are considered in our proof of security for Arctic.

Corollary 1. *Let* $\mathsf{Fork}^{\mathcal{C}}_m$ *be the forking experiment in Figure 2.2. Then using the notation of Lemma 1 we have*

$$\begin{aligned} \mathsf{accept}(\mathsf{Fork}^{\mathcal{C}}_m) &\geq \mathsf{accept}(\mathsf{Fork}^{\mathcal{C}}) - \Pr[\mathsf{BadHashEvent}] \\ &\geq \mathsf{accept}(\mathsf{Fork}^{\mathcal{C}}) - 2r^2/|H| \end{aligned} \tag{2.2}$$

Where BadHashEvent *denotes the event that* $\mathsf{Fork}^{\mathcal{C}}_m$ *returns* $\perp$ *due to the boxed lines in Figure 2.2.*

Proof Intuition for Corollary 1. The intuition for Corollary 1 is straightforward, as it simply bounds the probability that two values sampled uniformly at random from a set (with replacement) might collide. As such, it bounds the probability that the same value might be sampled twice, when $2r$ total sampling events are performed.

$\mathsf{Exp}^{\mathsf{uf}}_{\mathcal{DS},\mathcal{A}}(\lambda)$	$\mathcal{O}^{\mathsf{Sign}}(m)$
$\mathsf{par} \leftarrow \mathsf{Setup}(1^\lambda)$	$Q_m \leftarrow Q_m \cup \{m\}$
$Q_m \leftarrow \emptyset$	$\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$
$(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}()$	**return** σ
$(m^*, \sigma^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{Sign}}}(\mathsf{pk})$	
return 0 **if** $m^* \in Q_m$	
$\quad \vee\, \mathsf{Verify}(\mathsf{pk}, m^*, \sigma^*) \neq 1$	
return 1	

Figure 2.3: The EUF-CMA security game for a digital signature scheme $\mathcal{DS}$.

2.3 Digital Signatures and Schnorr Signatures

Definition 2.3.1 (Digitial Signatures)**.** *A digital signature scheme consists of polynomial-time algorithms* $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$*, defined as follows:*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{par}$*: On input a security parameter* 1^λ*, this deterministic algorithm outputs public parameters* par *(which are given implicitly as input to all other algorithms).*
- $\mathsf{KeyGen}() \to (\mathsf{pk}, \mathsf{sk})$*: This probabilistic algorithm outputs a public key* pk *and secret key* sk*.*
- $\mathsf{Sign}(\mathsf{sk}, m) \to \sigma$*: On input a secret key* sk *and a message* m*, this (possibly probabilistic) algorithm outputs a signature* σ*.*
- $\mathsf{Verify}(\mathsf{pk}, m, \sigma) \to 0/1$*: On input a public key* pk*, a message* m*, and a purported signature* σ*, this deterministic algorithm outputs* 1 *(accept) if* σ *is a valid signature on* m *under* pk*; else, it outputs* 0 *(reject).*

A digital signature scheme must be *correct*. It is said to be *secure* if it is existentially unforgeable under chosen-message attacks (EUF-CMA).

Definition 2.3.2 (Correctness)**.** *A digital signature scheme* $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *is* correct *if for all security parameters* $\lambda \in \mathbb{N}$*, all public parameters* $\mathsf{par} \leftarrow \mathsf{Setup}(1^\lambda)$*, all key pairs* $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}()$*, and all messages* m*, we have:*

$$\Pr[\mathsf{Verify}(\mathsf{pk}, m, \mathsf{Sign}(\mathsf{sk}, m)) = 1] = 1$$

Definition 2.3.3 (EUF-CMA)**.** *Let the advantage of an adversary $\mathcal{A}$ playing the Existential Unforgeability under Chosen-Message Attack (EUF-CMA) game, $\mathsf{Exp}^{\mathsf{uf}}$, as defined in Figure 2.3, be as follows:*

$$\mathsf{Adv}^{\mathsf{uf}}_{\mathcal{DS},\mathcal{A}}(\lambda) = \left|\Pr[\mathsf{Exp}^{\mathsf{uf}}_{\mathcal{DS},\mathcal{A}}(\lambda) = 1]\right|$$

A digital signature scheme $\mathcal{DS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ is existentially unforgeable under chosen-message attacks *if for all PPT adversaries $\mathcal{A}$, $\mathsf{Adv}^{\mathsf{uf}}_{\mathcal{DS},\mathcal{A}}(\lambda)$ is negligible.*

2.3.1 Schnorr Signatures

A Schnorr signature [Sch91] is a Sigma protocol zero-knowledge proof of knowledge of the discrete logarithm of the public key pk. A Sigma protocol is characterized by the prover first outputting a commitment, the verifier then sending a random challenge to the prover, and then the prover outputting a response. The Schnorr signature scheme is made non-interactive and bound to the message m by the Fiat–Shamir transform [FS86]. Schnorr signatures are secure under the discrete logarithm assumption in the random oracle model [PS00].

Definition 2.3.4 (Schnorr Signatures [Sch91])**.** *The Schnorr signature scheme consists of polynomial-time algorithms* $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$*, defined as follows:*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{par}$*: On input a security parameter 1^λ, run $(\mathbb{G}, q, g) \leftarrow \mathsf{GroupGen}(1^\lambda)$ and select a hash function $\mathsf{H} : \mathbb{G} \times \mathbb{G} \times \{0,1\}^* \to \mathbb{Z}_q$. Output public parameters $\mathsf{par} \leftarrow ((\mathbb{G}, q, g), \mathsf{H})$ (which are given implicitly as input to all other algorithms).*
- $\mathsf{KeyGen}() \to (\mathsf{pk}, \mathsf{sk})$*: Sample a secret key $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$ and compute the public key as $\mathsf{pk} \leftarrow g^x$. Output key pair $(\mathsf{pk}, \mathsf{sk})$.*
- $\mathsf{Sign}(\mathsf{sk}, m) \to \sigma$*: On input a secret key $\mathsf{sk} = x$ and a message $m \in \{0,1\}^*$, sample a nonce $r \xleftarrow{\$} \mathbb{Z}_q$. Then, compute a nonce commitment $R \leftarrow g^r$, the challenge $c \leftarrow \mathsf{H}(R, \mathsf{pk}, m)$, and the response $z \leftarrow r + cx$. Output a signature $\sigma \leftarrow (R, z)$.*
- $\mathsf{Verify}(\mathsf{pk}, m, \sigma) \to 0/1$*: On input a public key pk, a message $m \in \{0,1\}^*$, and a purported signature $\sigma = (R, z)$, compute $c \leftarrow \mathsf{H}(R, \mathsf{pk}, m)$. Output 1 (accept) if $R \cdot \mathsf{pk}^c = g^z$; else, output 0 (reject).*

2.4 Non-Interactive Key Exchange (NIKE)

In Chapter 4, we give a generic Distributed Key Generation (DKG) scheme that builds upon a Non-Interactive Key Exchange (NIKE) [Ber06, CKS08, FHKP13] scheme. We additionally define a verification algorithm that checks whether a tuple is in fact a valid keypair.

Definition 2.4.1. *A* **Non-Interactive Key Exchange (NIKE)** NK *with verification is the tuple* $\mathsf{NK} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Verify}, \mathsf{SharedKey})$*, where*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{par}$*: On input a security parameter* 1^λ*, this deterministic algorithm outputs public parameters* par *(which are given implicitly as input to all other algorithms).*
- $\mathsf{KeyGen}() \to (\mathsf{sk}, \mathsf{pk})$*: This probabilistic algorithm generates a secret key* sk *and corresponding public key* pk*.*
- $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}) \to \{0, 1\}$*: On input the secret key* sk *and public key* pk*, this deterministic algorithm verifies that* $(\mathsf{sk}, \mathsf{pk})$ *is a valid output from* $\mathsf{NK.KeyGen}$*.*
- $\mathsf{SharedKey}(\mathsf{sk}_1, \mathsf{pk}_2) \to \psi$*: On input secret key* sk_1 *and public key* pk_2*, this deterministic algorithm outputs the shared key* ψ *generated by the combination of one party's secret key* sk_1 *and the other party's public key* pk_2*.*

For correctness, we require that for all $(\mathsf{sk}_1, \mathsf{pk}_1) \xleftarrow{\$} \mathsf{NK.KeyGen}()$ and $(\mathsf{sk}_2, \mathsf{pk}_2) \xleftarrow{\$} \mathsf{NK.KeyGen}()$, the following conditions hold:

$$\mathsf{NK.Verify}(\mathsf{sk}_i, \mathsf{pk}_i) = 1, \text{ for } i \in \{1, 2\} \text{ and}$$
$$\mathsf{NK.SharedKey}(\mathsf{sk}_1, \mathsf{pk}_2) = \mathsf{NK.SharedKey}(\mathsf{sk}_2, \mathsf{pk}_1)$$

In addition to the requirement that a NIKE be *session-key unrecoverable*, we additionally require that a NIKE be *binding*. We discuss both properties next.

Session-Key Recovery. A NIKE that is *session-key unrecoverable* ensures that given the public keys of two parties $\mathsf{pk}_1, \mathsf{pk}_2$, it should be hard for an adversary to obtain the shared secret. While prior security notions for non-interactive key exchange exist in the literature [CKS08, FHKP13], these notions allow an adversary to act as an active participant. Our constructions require only the more restricted model where the adversary learns

$\mathsf{Exp}^{\mathrm{rec}}_{\mathsf{NK},\mathcal{A}}(1^\lambda)$

$\mathsf{par} \leftarrow \mathsf{NK.Setup}(1^\lambda)$
$(\mathsf{sk}_1, \mathsf{pk}_1) \xleftarrow{\$} \mathsf{NK.KeyGen}()$
$(\mathsf{sk}_2, \mathsf{pk}_2) \xleftarrow{\$} \mathsf{NK.KeyGen}()$
$\psi \leftarrow \mathsf{NK.SharedKey}(\mathsf{sk}_1, \mathsf{pk}_2)$
$\psi' \xleftarrow{\$} \mathcal{A}(\mathsf{pk}_1, \mathsf{pk}_2)$
return 1 **if** $\psi' = \psi$
return 0

Figure 2.4: Unrecoverability experiment defining the advantage of an adversary $\mathcal{A}$ against a non-interactive key exchange (NIKE) NK.

key material but does not contribute itself, and must compute the shared secret directly. We present this restricted notion in Figure 2.4.

The advantage of an adversary $\mathcal{A}$ against NK in the unrecoverability experiment as defined in Figure 2.4 is

$$\mathsf{Adv}^{\mathrm{rec}}_{\mathsf{NK},\mathcal{A}}(\lambda) = \Pr[\mathsf{Exp}^{\mathrm{rec}}_{\mathsf{NK},\mathcal{A}}(\lambda) = 1]$$

Definition 2.4.2. *A NIKE is* **session-key unrecoverable** *if for all probabilistic polynomial time adversaries $\mathcal{A}$, the function $\mathsf{Adv}^{\mathrm{rec}}_{\mathsf{NK},\mathcal{A}}(\lambda)$ is negligible.*

Binding. We introduce an additional security property that must hold when the adversary is allowed to generate both sides of the key exchange. Informally, a NIKE is *binding* when $\mathsf{NK.SharedKey}(\mathsf{sk}_1, \mathsf{pk}_2)$ must equal $\mathsf{NK.SharedKey}(\mathsf{sk}_2, \mathsf{pk}_1)$, even when the adversary is allowed to generate $(\mathsf{sk}_1, \mathsf{pk}_1), (\mathsf{sk}_2, \mathsf{pk}_2)$ itself. The only restriction to the adversary is that both keypairs must be valid, and the adversary can perform all NK operations directly. We formalize this requirement in Figure 2.5.

The advantage of an adversary $\mathcal{A}$ against NK in the binding experiment as defined in Figure 2.5 is

$$\mathsf{Adv}^{\mathrm{bind}}_{\mathsf{NK},\mathcal{A}}(\lambda) = \Pr[\mathsf{Exp}^{\mathrm{bind}}_{\mathsf{NK},\mathcal{A}}(\lambda) = 1]$$

Definition 2.4.3. *A NIKE is* **binding** *if for all probabilistic polynomial time adversaries $\mathcal{A}$, the function $\mathsf{Adv}^{\mathrm{bind}}_{\mathsf{NK},\mathcal{A}}(\lambda)$ is negligible.*

$\mathsf{Exp}^{\mathrm{bind}}_{\mathsf{NK},\mathcal{A}}(\lambda)$

$\mathsf{par} \leftarrow \mathsf{NK.Setup}(1^\lambda)$

$(\mathsf{sk}_1^*, \mathsf{pk}_1^*, \mathsf{sk}_2^*, \mathsf{pk}_2^*) \xleftarrow{\$} \mathcal{A}(\mathsf{par})$

return 0 **if** $\mathsf{NK.Verify}(\mathsf{sk}_1^*, \mathsf{pk}_1^*) \neq 1$

return 0 **if** $\mathsf{NK.Verify}(\mathsf{sk}_2^*, \mathsf{pk}_2^*) \neq 1$

return 1 **if** $\mathsf{NK.SharedKey}(\mathsf{sk}_1^*, \mathsf{pk}_2^*) \neq \mathsf{NK.SharedKey}(\mathsf{sk}_2^*, \mathsf{pk}_1^*)$

return 0

Figure 2.5: Binding experiment defining the advantage of an adversary $\mathcal{A}$ against a non-interactive key exchange (NIKE) NK.

2.4.1 A Concrete NIKE

We now describe a concrete NIKE which is a non-interactive Diffie–Hellman key exchange, with additionally a verification check to ensure that $\mathsf{pk} = g^{\mathsf{sk}}$.

- $\mathsf{Setup}(1^\lambda) \to \mathsf{par}$: On input a security parameter 1^λ, run $(\mathbb{G}, q, g) \leftarrow \mathsf{GroupGen}(1^\lambda)$. Output public parameters $\mathsf{par} \leftarrow (\mathbb{G}, q, g)$ (which are given implicitly as input to all other algorithms).
- $\mathsf{NK.KeyGen}() \to (\mathsf{sk}, \mathsf{pk})$: Sample secret key $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$; derive public key $\mathsf{pk} \in \mathbb{G}$ as $\mathsf{pk} = g^{\mathsf{sk}}$. Output $(\mathsf{sk}, \mathsf{pk})$.
- $\mathsf{NK.Verify}(\mathsf{sk}, \mathsf{pk}) \to \{0, 1\}$: Output 1 if $\mathsf{pk} = g^{\mathsf{sk}}$; otherwise, output 0.
- $\mathsf{NK.SharedKey}(\mathsf{sk}_1, \mathsf{pk}_2) \to \psi$: Derive the shared key $\psi \in \mathbb{G}$ as $\psi \leftarrow \mathsf{pk}_2^{\mathsf{sk}_1}$. Output ψ.

We next demonstrate that the concrete scheme is unrecoverable assuming the Computational Diffie–Hellman (CDH) problem is hard, and is unconditionally binding.

Theorem 1. *The concrete NIKE of Section 2.4.1 is unrecoverable against a probabilistic polynomial-time adversary $\mathcal{A}$, assuming the Computational Diffie–Hellman (CDH) assumption holds in $\mathbb{G}$. More specifically, for any PPT adversary $\mathcal{A}$ that attacks NK, there exists an adversary $\mathcal{B}$ with approximately the same running time as $\mathcal{A}$ such that*

$$\mathsf{Adv}^{\mathrm{rec}}_{\mathsf{NK},\mathcal{A}}(\lambda) \leq \mathsf{Adv}^{\mathrm{cdh}}_{\mathcal{B}}(\lambda)$$

Because breaking session key recovery in our concrete scheme simply requires an adversary to guess a valid Diffie–Hellman shared secret knowing only two public keys, we omit the proof for Theorem 1 as it is a direct reduction to CDH.

Theorem 2. *The concrete NIKE of Section 2.4.1 is binding against a computationally unbounded adversary $\mathcal{A}$.*

Proof. We now show that any adversary $\mathcal{A}$ cannot win the binding experiment $\mathsf{Exp}^{\text{bind}}$ in Figure 2.5 for the concrete NIKE of Section 2.4.1. $\mathcal{A}$ wins when it can produce some $(\mathsf{sk}_1^*, \mathsf{pk}_1^*, \mathsf{sk}_2^*, \mathsf{pk}_2^*)$ such that:

1. $\mathsf{NK.Verify}(\mathsf{sk}_1^*, \mathsf{pk}_1^*)$ outputs 1, and
2. $\mathsf{NK.Verify}(\mathsf{sk}_2^*, \mathsf{pk}_2^*)$ outputs 1, but
3. $\mathsf{NK.SharedKey}(\mathsf{sk}_1^*, \mathsf{pk}_2^*) \neq \mathsf{NK.SharedKey}(\mathsf{sk}_2^*, \mathsf{pk}_1^*)$

However, in the setting of the concrete scheme, this means that $\mathsf{pk}_1^* = g^{\mathsf{sk}_1^*}$ and $\mathsf{pk}_2^* = g^{\mathsf{sk}_2^*}$, yet $(\mathsf{pk}_2^*)^{\mathsf{sk}_1^*} \neq (\mathsf{pk}_1^*)^{\mathsf{sk}_2^*}$, which is impossible. □

2.5 Threshold Secret Sharing Schemes

Definition 2.5.1. *A (n, t)-**threshold secret sharing scheme** is a tuple of algorithms $\mathsf{TS} = (\mathsf{Setup}, \mathsf{Share}, \mathsf{Recover})$, such that:*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{par}$*: On input the security parameter, this deterministic algorithm outputs public parameters* par *which are given implicitly as input to all other algorithms.*
- $\mathsf{Share}(s, n, t) \to (\{(1, \phi_1), \ldots, (n, \phi_n)\})$*: A probabilistic algorithm that accepts as input the security parameter λ, the secret s, the number of participants n, and the threshold t, where n and t are positive integers, and $n \geq t$. Outputs the list of shares $\{(1, \phi_1), \ldots, (n, \phi_n)\}$.*
- $\mathsf{Recover}(t, \{(i, \phi_i)\}_{i \in C}) \to s/\perp$*: A deterministic algorithm that accepts a threshold t and a set of shares $\{(i, \phi_i)\}_{i \in C}$. Output $\perp$ if $C \not\subseteq [n]$ or if $|C| < t$. Otherwise, recover s using the set of shares.*

For correctness, for all security parameters λ and n, t, C, such that $n \geq t$, $t \geq 1$, $C \subseteq [n]$, and $|C| \geq t$, the following must hold:

$$\mathsf{TS.Recover}(t, \{(i, \phi_i)\}_{i \in C}) = s, \text{ where } \mathsf{TS.Share}(s, n, t) \to (\{(i, \phi_i)\}_{i \in [n]})$$

Remark 1 (Required number of shares for recovery)**.** Because Recover requires a set of size at least t, the definition implicitly ensures that the algorithm receives a sufficient number of shares for recovery.

2.5.1 Shamir Secret Sharing

We now define Shamir secret sharing as one concrete instantiation of a threshold secret sharing scheme.

Definition 2.5.2 (Shamir secret sharing [Sha79])**.** *The Shamir secret sharing scheme is a concrete instantiation of a threshold secret sharing scheme that consists of the following algorithms:*

- $\mathsf{Shamir.Setup}(1^\lambda) \to \mathsf{par}$*:*
  On input the security parameter, run $(\mathbb{G}, q, g) \leftarrow \mathsf{GroupGen}(1^\lambda)$*. Output public parameters* $\mathsf{par} = (\mathbb{G}, q, g)$*, given implicitly as input to all other algorithms.*

- $\mathsf{Shamir.Share}(s, n, t) \to \{(1, \phi_1), \ldots, (n, \phi_n)\}$*: On input a secret* s*, the number of participants* n*, and a threshold* t*, perform the following. First, define a polynomial* $f(Z) = s + a_1 + a_2 Z^2 + \cdots + a_{t-1} Z^{t-1}$ *by sampling* $t - 1$ *coefficients at random* $(a_1, \ldots, a_{t-1}) \xleftarrow{\$} \mathbb{Z}_q$*. Then, set each participant's share* $\phi_i, i \in [n]$*, to be the evaluation of* $f(i)$*:*

  $$\phi_i \leftarrow s + \sum_{j \in [t-1]} a_j i^j$$

  Output $\{(i, \phi_i)\}_{i \in [n]}$*.*

- $\mathsf{Shamir.Recover}(t, \{(i, \phi_i)\}_{i \in C}) \to s/\perp$*: On input a threshold* t *and a set of shares* $\{(i, \phi_i)\}_{i \in C}$*, output* $\perp$ *if* $C \not\subseteq [n]$ *or if* $|C| < t$*. Otherwise, recover* s *as follows:*

  $$s \leftarrow \sum_{i \in C} \lambda_i \phi_i$$

  where the Lagrange coefficient for participant i *in the set* C *is defined by*

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

2.5.2 Verifiable Secret Sharing

Verifiable Secret Sharing (VSS) allows a dealer to share a secret s in such a way that participants can ensure that combining their shares allows for recovery of a secret that corresponds to a public commitment to s, without learning s in the process. Participants verify that their shares can correctly recover s using this commitment. While prior definitions of VSS schemes [KL14] define algorithms to issue and verify shares, and recover the secret, we additionally define an algorithm to derive shares in a public domain. This notion has been employed implicitly in prior DKGs [KG20], and we formalize this construct here.

Definition 2.5.3. *A* **Verifiable Secret Sharing (VSS) scheme** S *is the tuple* $(\mathsf{Setup}, \mathsf{Share}, \mathsf{Recover}, \mathsf{Verify}, \mathsf{GetPub})$ *and parameterized by a one-way map* $\mathsf{SecretToPublic} : \hat{S} \to \hat{P}$ *that maps elements in the secret domain* $\hat{S}$ *to the public domain* $\hat{P}$*, where*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{par}$*: On input a security parameter* 1^λ*, this algorithm outputs public parameters* par *(which are given implicitly as input to all other algorithms).*
- $\mathsf{Share}(s, n, t) \to (\{(1, \phi_1), \ldots, (n, \phi_n)\}, D)$*: A probabilistic algorithm performed by a dealer that accepts as input the secret* $s \in \hat{S}$ *from the secret domain* $\hat{S}$*, the number of participants* n*, and the threshold* t*, where* n *and* t *are positive integers, and* $n \geq t$*. Outputs the list of shares* $\{(1, \phi_1), \ldots, (n, \phi_n)\}$ *and commitment* D*.*
- $\mathsf{Verify}(i, \phi_i, D) \to \{0, 1\}$*: A deterministic algorithm performed by a recipient of a share that accepts an identifier* i*, a share* ϕ_i*, and a commitment* D*. Outputs* 1 *if the share is valid with respect to* D*, otherwise outputs* 0*.*
- $\mathsf{Recover}(t, \mathsf{M}) \to s/\perp$*: A deterministic algorithm that accepts a recovery set* $\mathsf{M} = \{(j, \phi_j)\}_{j \in C}$*, where* C *is a set of participant identifiers such that* $|C| \geq t$*. Employ* M *to obtain* s*, or output* $\perp$ *in the case of failure.*
- $\mathsf{GetPub}(i, D) \to W_i$*: A deterministic algorithm that accepts an identifier* i *and a commitment* D*. Outputs a share commitment* W_i *in the public domain associated with identifier* i*.*

For correctness, for all security parameters $\mathsf{par} \leftarrow \mathsf{Setup}(1^\lambda)$, for all $s \in \hat{S}$, and n, t, C, such that $n \geq t$, $t \geq 1$, $C \subseteq [n]$, and $|C| \geq t$, the following must hold:

$$
\begin{aligned}
&\mathsf{S.Verify}(i, \phi_i, D) = 1 \text{ for all } i \in [n], \text{ and} \\
&\mathsf{S.GetPub}(i, D) = \mathsf{S.SecretToPublic}(\phi_i) \text{ for all } i \in [n], \text{ and} \\
&\mathsf{S.Recover}(t, \mathsf{M}) = s, \text{ where} \\
&\mathsf{S.Share}(s, n, t) \to (\{(i, \phi_i)\}_{i \in [n]}, D) \text{ and } \mathsf{M} = \{(i, \phi_i)\}_{i \in C}
\end{aligned}
$$

Zero-Indexing. For the purposes of our definitions, we additionally require that a VSS be *zero-indexed*, which we define next.

Definition 2.5.4. *A* **zero-indexed VSS** *is one which satisfies the correctness condition augmented with the following line:*

$$\mathsf{S.Verify}(0, s, D) = 1 \text{ and } \mathsf{S.GetPub}(0, D) = \mathsf{S.SecretToPublic}(s) \ . \tag{2.3}$$

A prototypical example of a zero-indexed VSS is Feldman's secret sharing [Fel87], which itself builds on Shamir secret sharing [Sha79]. Zero-indexing is important later in our generic construction given in Chapter 4, to demonstrate a relation between the VSS and a NIKE, where the secret that is input to the VSS is the secret key output from the NIKE key generation algorithm.

We now employ the notion of a zero-indexed VSS to define the security notions of secrecy and uniqueness for VSS schemes, building upon prior definitions in the literature [KL14, BS20], but formalized in game-based notation.

Secrecy. A VSS scheme is *secret* if an adversary that is allowed to corrupt up to $t-1$ players has a negligible probability of learning the secret s. This notion can be also expressed in terms of *simulatability*; i.e., an environment that can simulate secret sharing for an unknown challenge ensures that the adversary learns no additional information than is known by the environment. We formalize this notion in Figure 2.6, with respect to an algorithm $\mathsf{SimShare}$ that is defined in the context of the concrete construction. The experiment takes as input any secret s from the secret space $\hat{S}$. Importantly, we do not restrict secrets to be sampled uniformly at random; even degenerate cases should hold. In other words, even "seemingly bad" cases are allowed, such as the identity element.

The advantage of an adversary $\mathcal{A}$ against a VSS scheme S in the secrecy experiment as defined in Figure 2.6 is

$$\mathsf{Adv}^{\mathrm{sec}}_{\mathsf{S},\mathcal{A},\mathsf{SimShare}}(\lambda) = \max_{s \in \hat{S}}(\left|\Pr[\mathsf{Exp}^{\mathrm{sec}}_{\mathsf{S},\mathcal{A},\mathsf{SimShare}}(\lambda, s) = 1] - 1/2\right|)$$

Definition 2.5.5. *A VSS scheme* S *is* **secret** *if there exists an algorithm* $\mathsf{SimShare}$ *such that for all probabilistic polynomial time adversaries* $\mathcal{A}$, $\mathsf{Adv}^{\mathrm{sec}}_{\mathsf{S},\mathcal{A},\mathsf{SimShare}}(\lambda)$ *is a negligible function of* λ.

$\mathsf{Exp}^{\mathrm{sec}}_{\mathsf{S},\mathcal{A},\mathsf{SimShare}}(\lambda, s)$

$b \xleftarrow{\$} \{0,1\}$
$\mathsf{par} \leftarrow \mathsf{Setup}(1^\lambda)$
$(n, t, \mathsf{corrupt}, \mathsf{st}_A) \xleftarrow{\$} \mathcal{A}(\mathsf{par})$
return 0 **if** $t > n$ **or** $\mathsf{corrupt} \not\subseteq [n]$ **or** $|\mathsf{corrupt}| \geq t$
if $b = 0$
  $(\{(j, \phi_j)\}_{j\in[n]}, D) \leftarrow \mathsf{S.Share}(s, n, t)$
else // $b = 1$ *case*
  $\Delta \leftarrow \mathsf{S.SecretToPublic}(s)$
  $(\{(j, \phi_j)\}_{j\in\mathsf{corrupt}}, D) \xleftarrow{\$} \mathsf{SimShare}(\Delta, n, t, \mathsf{corrupt})$
  for $j \in \mathsf{corrupt}$ **do** // *The simulation must produce* $|\mathsf{corrupt}|$ *valid shares*
    return 1 **if** $\mathsf{S.Verify}(j, \phi_j, D) \neq 1$
  return 1 **if** $\mathsf{S.GetPub}(0, D) \neq \Delta$ // *The commitment must be valid with respect to* Δ
$b' \xleftarrow{\$} \mathcal{A}(\mathsf{st}_A, \{(j, \phi_j)\}_{j\in\mathsf{corrupt}}, D)$
return 1 **if** $b' = b$
return 0

Figure 2.6: Secrecy experiment defining the advantage of an adversary $\mathcal{A}$ against a Verifiable Secret Sharing scheme (VSS) S that is zero-indexed.

Uniqueness. Informally, a VSS scheme S that is *unique* is one where the public commitment D uniquely determines the output from $\mathsf{S.Recover}$. We formalize this notion in Figure 2.7.

The advantage of an adversary $\mathcal{A}$ against S in the uniqueness experiment as defined in Figure 2.7 is

$$\mathsf{Adv}^{\mathrm{unq}}_{\mathsf{S},\mathcal{A}}(\lambda) = \Pr[\mathsf{Exp}^{\mathrm{unq}}_{\mathsf{S},\mathcal{A}}(\lambda) = 1]$$

Definition 2.5.6. *A VSS scheme is* **unique** *if for all probabilistic polynomial time adversaries* $\mathcal{A}$, $\mathsf{Adv}^{\mathrm{unq}}_{\mathsf{S},\mathcal{A}}(\lambda)$ *is a negligible function of* λ.

2.5.2.1 Feldman's VSS

As a concrete VSS in the discrete logarithm setting, Feldman's Verifiable Secret Sharing [Fel87] is an important building block. Feldman's VSS itself is an augmentation to

$\mathsf{Exp}^{\mathrm{unq}}_{\mathsf{S},\mathcal{A}}(\lambda)$

$\mathsf{par} \leftarrow \mathsf{Setup}(1^\lambda)$

// $\mathcal{A}$ chooses two recovery sets and a commitment with respect to the same threshold

$(t^*, \mathsf{M}_1^*, \mathsf{M}_2^*, D^*) \xleftarrow{\$} \mathcal{A}(\mathsf{par})$

for $(i, \phi_i) \in \mathsf{M}_1^*$ **do**

  return 0 **if** $\mathsf{S.Verify}(i, \phi_i, D^*) \neq 1$

for $(i', \phi_i') \in \mathsf{M}_2^*$ **do**

  return 0 **if** $\mathsf{S.Verify}(i', \phi_i', D^*) \neq 1$

return 0 **if** $\mathsf{S.Recover}(t^*, \mathsf{M}_1^*) = \perp \vee \mathsf{S.Recover}(t^*, \mathsf{M}_2^*) = \perp$

// $\mathcal{A}$ wins if it can generate two different recovery sets for the same

// commitment, but which recover distinct secrets

return 1 **if** $\mathsf{S.Recover}(t^*, \mathsf{M}_1^*) \neq \mathsf{S.Recover}(t^*, \mathsf{M}_2^*)$

return 0

Figure 2.7: Uniqueness experiment defining the advantage of an adversary $\mathcal{A}$ against a Verifiable Secret Sharing scheme (VSS) S.

Shamir's secret sharing [Sha79], allowing participants to verify the correctness of their in zero knowledge, by committing to the polynomial "in the exponent." Here, the map $\mathsf{S.SecretToPublic}$ is simply $a \mapsto g^a$, where g is the generator of the group $\mathbb{G}$, and $a \in \mathbb{Z}_q$.

We now define the scheme more precisely.

- $\mathsf{Setup}(1^\lambda) \to \mathsf{par}$: On input a security parameter 1^λ, run $(\mathbb{G}, q, g) \leftarrow \mathsf{GroupGen}(1^\lambda)$ and select a hash function $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_q$. Output public parameters $\mathsf{par} \leftarrow ((\mathbb{G}, q, g), \mathsf{H})$ (which are given implicitly as input to all other algorithms).
- $\mathsf{Share}(s, n, t) \to (\{(1, \phi_1), \ldots, (n, \phi_n)\}, D)$. Accepts as input the security parameter λ, a secret $s \in \mathbb{Z}_q$, the number of participants $n \in \mathbb{N}$, and the threshold $t \in \mathbb{N}$ such that $1 \leq t \leq n$. Perform the following steps:
  1. Sample $t-1$ coefficients at random: $(a_1, \ldots, a_{t-1}) \xleftarrow{\$} \mathbb{Z}_q^{t-1}$.
  2. Using s as the constant term and $a_1, \ldots, a_{t-1}$ as the remaining coefficients, define the $(t-1)$-degree polynomial $f(x) = s + \sum_{i=1}^{t-1} a_i x^i$.
  3. Generate n shares $\phi_j \in \mathbb{Z}_q$ by deriving $\phi_j \leftarrow f(j),\ j \in [n]$.

4. Define $A_0 = g^s$ and $A_i = g^{a_i}$, for $i \in [t-1]$.
5. Define the VSS commitment D as the vector of commitments to the coefficients of f: $D \leftarrow (A_0, \ldots, A_{t-1})$.
6. Output $(\{(i, \phi_i)\}_{i \in [n]}, D)$, where each i represents the identifier of the recipient of ϕ_i.

- $\mathsf{Verify}(i, \phi_i, D) \rightarrow \{0, 1\}$: Accepts as input a participant identifier $i \in [n]$, the share ϕ_i that belongs to participant i, and a VSS commitment D that is a tuple of group elements, such that $|D| = t$. Verifies that (i, ϕ_i) is a valid point on f as follows:

  1. Parse $(A_0, \ldots, A_{t-1}) \leftarrow D$.
  2. Output 1 if Equation 2.4 holds; otherwise, output 0.

$$g^{\phi_i} \stackrel{?}{=} \prod_{k=0}^{t-1} A_k^{i^k} \tag{2.4}$$

- $\mathsf{Recover}(t, \mathsf{M}) \rightarrow s/\perp$: Accepts as input the threshold t, a recovery set $\mathsf{M} = \{(j, \phi_j)\}_{j \in C}$ consisting of secret shares for a coalition $C \subseteq [n]$ where $|C| \geq t$. If $|C| < t$, output $\perp$. Using the inputs, perform the following steps:

  1. Derive Lagrange coefficients for each identifier $\{L_j(0)\}_{j \in C}$.
  2. Derive $s \leftarrow \sum_{j \in C} \phi_j \cdot L_j(0)$.
  3. Output s.

- $\mathsf{GetPub}(i, D) \rightarrow W_i$: Accepts as input a participant identifier i and a VSS commitment. Parse $(A_0, \ldots, A_{t-1}) \leftarrow D$. Outputs $W_i = g^{f(i)}$ by computing $W_i \leftarrow \prod_{k=0}^{|D|-1} A_k^{i^k}$.

2.6 Threshold Signatures

This section defines threshold signature schemes. Portions of this section are adapted from previous work by myself and co-authors [CKM23].

A threshold signature scheme allows a set of signers—t at minimum, out of n possible signers—to jointly produce a signature on a message such that the signature verifies under a public key that represents the set of all n signers. A threshold signature scheme should remain unforgeable even if t_c signers are corrupted, where $t_c \leq t - 1$.

Note that our definition of threshold signatures simply assumes that participants exchange protocol messages $\rho_{j,i}$, where j represents the j^{th} signing round, and i represents participant i. We model threshold signatures assuming a coordinator role, and so do not distinguish between peer-to-peer and broadcast messages. Our notion of unforgeability assumes that the adversary can play the role of the coordinator, and so can selectively choose which messages each participant receives in each round of signing.

To simplify notation, the below definition assumes a centralized key generation algorithm to generate the joint public key representing all n signers and the shares of the corresponding secret key. However, a distributed key generation (DKG) protocol may replace this trusted dealer. We discuss DKGs further in Chapter 4.

Definition 2.6.1. *A* **threshold signature scheme** TS *whose signing protocol consists of κ rounds is a tuple of the following algorithms* $\mathsf{TS} = (\mathsf{Setup}, \mathsf{KeyGen}, (\mathsf{Sign}_1, \ldots, \mathsf{Sign}_\kappa), \mathsf{Combine}, \mathsf{Verify})$*, defined as follows:*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{par}$*: A deterministic algorithm that accepts as input a security parameter λ, and outputs public parameters* par *(which are given implicitly as input to all other algorithms).*
- $\mathsf{KeyGen}(n, t) \to (\mathsf{pk}, \{\mathsf{pk}_i\}_{i\in[n]}, \{\mathsf{sk}_i\}_{i\in[n]})$*: A probabilistic algorithm that takes as input the number of signers n and the threshold t and outputs the public key* pk *representing the set of n signers and the set $\{\mathsf{pk}_i\}_{i\in[n]}$ of public keys for each signer, such that $\mathsf{pk}_i \in \hat{P}$ and $\mathsf{pk} \in \hat{P}$, where $\hat{P}$ is the domain of public keys for* TS*. The algorithm also outputs set $\{\mathsf{sk}_i\}_{i\in[n]}$ of secret shares corresponding to a secret key* sk*, where $\mathsf{sk}_i \in \hat{S}$ and $\mathsf{sk} \in \hat{S}$, such that $\hat{S}$ is the domain of secret keys for* TS*. Each sk_i is given to participant i, for $i \in [n]$. It is assumed that participant i is sent its respective share sk_i privately.*
- $\mathsf{Sign}_1 \to \rho_{1,i}, \ldots, \mathsf{Sign}_\kappa \to \rho_{\kappa,i}$*: A set of probabilistic algorithms where each subsequent algorithm represents a single stage from rounds $1, \ldots, \kappa$ in an interactive signing protocol, performed by each signing party i in a signing set $C \subseteq [n], |C| \geq t$, with respect to a message m, defined as follows:*

$$\begin{aligned}(\rho_{1,i}, \mathsf{st}_{1,i}) &\leftarrow \mathsf{Sign}_1(i, \mathsf{sk}_i, C, m)\\ (\rho_{j,i}, \mathsf{st}_{j,i}) &\leftarrow \mathsf{Sign}_2(\mathsf{st}_{j-1,i}, \mathcal{PM}_{j-1}) \text{for } 2 \leq j \leq \kappa - 1\\ \rho_{\kappa,i} &\leftarrow \mathsf{Sign}_\kappa(\mathsf{st}_{\kappa-1,i}, \mathcal{PM}_{\kappa-1})\end{aligned} \tag{2.5}$$

where $\rho_{1,i}, \ldots, \rho_{\kappa,i}$ are sets of protocol messages for each signing round, $\mathcal{PM}_j = \{\rho_{j,i}\}_{i\in C}$ is a set of protocol messages, and $\mathsf{st}_{j,i}$ is the state of signing party i in C

in round j. The signing set C and message m are shown here as inputs to the first round, but they may be deferred to subsequent rounds, depending on the scheme.

- $\mathsf{Combine}(C, m, \mathcal{PM}_1, \mathcal{PM}_2, \ldots, \mathcal{PM}_\kappa) \to \sigma$*:*
  A deterministic algorithm that takes as input the signing set C, the message m, and a set of protocol messages $\mathcal{PM}_1, \mathcal{PM}_2, \ldots, \mathcal{PM}_\kappa$ from all κ signing rounds, and outputs a signature σ. (Note that this algorithm is separate from the signing rounds and may be performed by one of the signers or an external party.)

- $\mathsf{Verify}(\mathsf{pk}, m, \sigma) \to 0/1$*: A deterministic algorithm that takes as input the public key pk, a message m, and a purported signature σ, and outputs 1 (accept) if the signature verifies; else, 0 (reject).*

Remark 2. The syntax assumes that the signing set C, message m, and secret key share sk_i are given as input in the first round of signing. However, constructions may defer these values to later rounds of signing, if the proof of security so allows.

Remark 3. The key generation algorithm assumes that the public key shares and public keys are from the same domain $\hat{P}$, and likewise the secret key shares and secret keys from the domain $\hat{S}$. Note however that some constructions may have different domains for key shares and the joint keys.

Remark 4 (On Protocol Message Types)**.** Our definition of threshold signatures does not distinguish between broadcast messages or peer-to-peer messages, and simply defines protocol messages ρ. The concrete construction of the threshold signature scheme will provide the necessary structures to realize any necessary assumptions for protocol message delivery.

Correctness. A threshold signature scheme is *correct* if for all security parameters $\lambda \in \mathbb{N}$, public parameters $\mathsf{par} \leftarrow \mathsf{Setup}(1^\lambda)$, all $1 \leq t \leq n$, all $C \subseteq [n]$ such that $|C| \geq t$, all messages m and for $(\mathsf{pk}, \{\mathsf{pk}_i\}_{i\in[n]}, \{\mathsf{sk}_i\}_{i\in[n]}) \xleftarrow{\$} \mathsf{KeyGen}(n, t)$, the signing algorithms as defined in Equation 2.5 return signature shares that, when combined with all other shares, produce a signature σ such that $\mathsf{Verify}(\mathsf{pk}, m, \sigma) = 1$.

2.6.1 Static Security

Most threshold signature schemes in the literature are proven in the *static* security model, which captures a limited adversary that can only corrupt parties at the onset of the protocol. A stronger adversarial model is one that considers an *adaptive* adversary, which we

$\mathsf{Exp}^{\mathsf{tuf}}_{\mathsf{TS},\mathcal{A}}(\lambda, n, t, t_c)$

$\mathcal{ID} \leftarrow \emptyset;\ Q_m \leftarrow \emptyset$
$\mathsf{rnd} \leftarrow \emptyset$
$\mathsf{par} \leftarrow \mathsf{Setup}(1^\lambda)$
$(\mathsf{corrupt}, \mathsf{st}_\mathcal{A}) \xleftarrow{\$} \mathcal{A}(\mathsf{par}, n, t, t_c)$
return $\bot$ **if** $\mathsf{corrupt} \not\subset [n] \vee |\mathsf{corrupt}| > t_c$
$\mathsf{honest} \leftarrow [n] \setminus \mathsf{corrupt}$
$(\mathsf{pk}, \{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]}) \xleftarrow{\$} \mathsf{KeyGen}(n, t)$
$\mathsf{input} \leftarrow (\mathsf{pk}, \{\mathsf{pk}_i\}_{i\in[n]}, \{\mathsf{sk}_j\}_{j\in\mathsf{corrupt}}, \mathsf{st}_\mathcal{A})$
$(m^*, \sigma^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{Sign}_1}, \mathcal{O}^{\mathsf{Sign}_2}, \ldots, \mathcal{O}^{\mathsf{Sign}_\kappa}}(\mathsf{input})$
return 1 **if** $m^* \notin Q_m$
$\quad \wedge\, \mathsf{Verify}(\mathsf{pk}, m^*, \sigma^*) = 1$
return 0

$\mathsf{Init}(\mathsf{eid}, k, C, m)$

if $k \notin \mathsf{honest} \cap C \vee$
$\quad C \not\subseteq [n] \vee |C| < t \vee$
$\quad\ \mathsf{rnd}[\mathsf{eid}, k] \neq \bot$
$\quad\quad$ **return** $\bot$
if $\mathsf{eid} \notin \mathcal{ID}$
$\quad \mathcal{ID} \leftarrow \mathcal{ID} \cup \{\mathsf{eid}\}$
$Q_m \leftarrow Q_m \cup \{m\}$

$\mathcal{O}^{\mathsf{Sign}_1}(\mathsf{eid}, k, C, m)$

return $\bot$ **if** $\bot \leftarrow \mathsf{Init}(\mathsf{eid}, k, C, m)$
$(\mathcal{PM}_1[\mathsf{eid}, k], \mathsf{st}_1[\mathsf{eid}, k])$
$\quad\quad \leftarrow \mathsf{Sign}_1(k, \mathsf{sk}_k, C, m)$
$\mathcal{PM}_{1,k} \leftarrow \mathcal{PM}_1[\mathsf{eid}, k];\ \mathsf{rnd}[\mathsf{eid}, k] \leftarrow 1$
return $\mathcal{PM}_{1,k}$

$\mathcal{O}^{\mathsf{Sign}_j}(\mathsf{eid}, k, \mathcal{PM}_{j-1})$ for $2 \leq j \leq \kappa - 1$

return $\bot$ **if** $\mathsf{eid} \notin \mathcal{ID}$
$\quad \vee\, k \notin \mathsf{honest} \vee \mathsf{rnd}[\mathsf{eid}, k] \neq j - 1$
parse $\{\mathcal{PM}_{j-1,i}\}_{i\in C} \leftarrow \mathcal{PM}_{j-1}$
return $\bot$ **if** $\mathcal{PM}_{j-1,k} \neq \mathcal{PM}_{j-1}[\mathsf{eid}, k]$
$(\mathcal{PM}_j[\mathsf{eid}, k], \mathsf{st}_j[\mathsf{eid}, k])$
$\quad\quad \leftarrow \mathsf{Sign}_j(\mathsf{st}_{j-1}[\mathsf{eid}, k], \mathcal{PM}_{j-1})$
$\mathcal{PM}_{j,k} \leftarrow \mathcal{PM}_j[\mathsf{eid}, k];\ \mathsf{rnd}[\mathsf{eid}, k] \leftarrow j$
return $\mathcal{PM}_{j,k}$

$\mathcal{O}^{\mathsf{Sign}_\kappa}(\mathsf{eid}, k, \mathcal{PM}_{\kappa-1})$

return $\bot$ **if** $\mathsf{eid} \notin \mathcal{ID}$
$\quad \vee\, k \notin \mathsf{honest} \vee \mathsf{rnd}[\mathsf{eid}, k] \neq \kappa - 1$
parse $\{\mathcal{PM}_{\kappa-1,i}\}_{i\in C} \leftarrow \mathcal{PM}_{\kappa-1}$
return $\bot$ **if** $\mathcal{PM}_{\kappa-1,k} \neq \mathcal{PM}_{\kappa-1}[\mathsf{eid}, k]$
$\mathcal{PM}_\kappa[\mathsf{eid}, k] \leftarrow \mathsf{Sign}_\kappa(\mathsf{st}_{\kappa-1}[\mathsf{eid}, k], \mathcal{PM}_{\kappa-1})$
$\mathcal{PM}_{\kappa,k} \leftarrow \mathcal{PM}_\kappa[\mathsf{eid}, k];\ \mathsf{rnd}[\mathsf{eid}, k] \leftarrow \kappa$
return $\mathcal{PM}_{\kappa,k}$

Figure 2.8: Static unforgeability game for a threshold signature scheme with κ signing rounds. Note this definition captures "weak" unforgeabiilty, as compared to static security for strong unforgeability of threshold signatures [BCK+22].

describe in Section 2.6.2. However, proving security assuming an adaptive adversary is more challenging, and often requires tradeoffs in the performance overhead of the scheme, or stronger security models [CKM23, BL22]. For this reason, we focus on proving the

security of FROST in Chapter 3 and Arctic in Chapter 5 assuming a static adversary.

We provide a game-based definition for unforgeability of threshold signatures in Figure 2.8, inspired by the definitions as given by Crites et al. [CKM23]. and Libert et al. [LJY16]. The static security game is analogous to existential unforgeability under chosen message attack (EUF-CMA) [GMR88] for standard, single-party signature schemes, as defined in Figure 2.3. The public parameters par are implicitly given as input to all algorithms, and $\mathcal{PM}_1, \mathcal{PM}_2, \ldots, \mathcal{PM}_\kappa$ represent sets of protocol messages defined within the construction. eid denotes an execution identifier, which is *not* assumed to be consistent among participants.

In the static (weak) unforgeability game, the environment accepts as input a security parameter λ, total number of players n, a reconstruction threshold t, and a corruption threshold t_c that the adversary may corrupt, such that $t_c < t \leq n$.

The environment generates public parameters par and initializes the adversary with $(\mathsf{par}, n, t, t_c)$. The adversary chooses a set of corrupted parties $\mathsf{corrupt}$, and the challenger sets the honest parties $\mathsf{honest} \leftarrow [n] \setminus \mathsf{corrupt}$. The challenger then runs KeyGen to derive the joint public key pk representing all n signers, the individual public key shares $\{\mathsf{pk}_i\}_{i\in[n]}$, and the secret key shares $\{\mathsf{sk}_i\}_{i\in[n]}$. It returns pk, $\{\mathsf{pk}_i\}_{i\in[n]}$, and the set of corrupt signing shares $\{\mathsf{sk}_j\}_{j\in\mathsf{corrupt}}$ to the adversary.

After key generation has concluded, the adversary can then query signing oracles $\mathcal{O}^{\mathsf{Sign}_1}, \ldots, \mathcal{O}^{\mathsf{Sign}_\kappa}$ for honest signers to engage in subsequent signing rounds, across different sessions. The adversary can participate as a signer in the signing protocol, on behalf of $t-1$ number of corrupted parties of its choosing. The adversary controls all protocol messages given to honest parties throughout the signing protocol. The environment performs various checks during the signing rounds to ensure each participant being called is in fact uncorrupted, is in the correct signing round, and that their protocol message in each round is faithfully carried to the next. Each of these checks is carried out with respect to the execution identifier, denoted by eid. The adversary wins if it can produce a valid forgery $\sigma^* = (R^*, z^*)$ with respect to the joint public key pk on a message m^* that has not been previously queried.

Note that notions of strong static unforgeability also exist for threshold signatures [BCK+22], and requires that the tuple (m^*, σ^*) was not output from a valid run of the protocol (involving honest parties), a stronger condition than weak unforgeability, which requires only that m^* was not queried to honest signers. We do not consider this notion of security in this thesis, but FROST has been proven to achieve strong static unforgeability in external work [BCK+22]. We model the adversary as *rushing*, as it may wait to produce its outputs after having seen honest protocol messages in each round. The adversary is

allowed to be *concurrent*, as it may open simultaneous signing sessions at once or choose not to complete a signing session. We model concurrency explicitly in our definition to ensure that our notion of security protects against practical attacks which can occur under a concurrent adversary [BLL+21].

Definition 2.6.2 (Static Security)**.** *Let the advantage of an adversary $\mathcal{A}$ playing the static security game against a threshold signature scheme $\mathsf{Exp}^{\mathsf{tuf}}_{\mathsf{TS},\mathcal{A}}(\lambda, n, t, t_c)$, as defined in Figure 2.8, be as follows:*

$$\mathsf{Adv}^{\mathsf{tuf}}_{\mathsf{TS},\mathcal{A}}(\lambda, n, t, t_c) = \left|\Pr[\mathsf{Exp}^{\mathsf{tuf}}_{\mathsf{TS},\mathcal{A}}(\lambda, n, t, t_c) = 1]\right|$$

A threshold signature scheme TS *is* statically secure *if for all PPT adversaries $\mathcal{A}$, $\mathsf{Adv}^{\mathsf{tuf}}_{\mathsf{TS},\mathcal{A}}(\lambda, n, t, t_c)$ is negligible, for $(n, t, t_c) \in \mathbb{N}, t \leq n,\ t_c < t$.*

2.6.2 Adaptive Security

In this work, we consider only static security for our threshold Schnorr signature constructions. While adaptive adversaries are certainly closer to a real-world setting, the complexity of modeling adversarial corruptions in a proof of security introduces challenging complexities. We leave proving the adaptive security of FROST in Chapter 3, Storm in Chapter 4, and Arctic in Chapter 5 to future work. We however next provide a high-level discussion of adaptive security here, for comparison.

In the adaptive setting, the adversary is given the added capability to corrupt parties *at any point during the protocol*, based on all protocol messages it has seen. This is in contrast to the static setting, where the adversary must declare the set of corrupt parties at the onset of the protocol, before any messages have been sent. Adaptive corruption means the adversary receives not only the corrupted party's secret key, but also its *state across all signing sessions.*

At any time throughout the experiment, the adversary may query an additional oracle $\mathcal{O}^{\mathsf{Corrupt}}$ to corrupt up to t_c honest signers, obtaining *both* their secret key and as well as state across all signing sessions. As such, in the adaptive setting, the adversary is not restricted to choosing its set of corrupt participants $\mathsf{corrupt}$ only at the beginning of the game, and learns additional information throughout the game (e.g., the signing state for each corrupted participant).

2.7 ROS Attacks

Unlike prior multi-party Schnorr signature schemes in the literature [DEF+19, BLL+21], both FROST in Chapter 3 and Arctic in Chapter 5 are secure against concurrent adversaries which may attempt ROS-style attacks. The ROS (Random Inhomogeneities in a Overdetermined Solvable System of Linear Equations) problem [Sch06, BLL+21, Sch01, FPS20] over $\mathbb{Z}_q$ in dimension ℓ is defined as follows. At a high level, the goal is to find a solution to $\ell + 1$ linear equations modulo q, where some elements in the equations are output from a random oracle. In other words, given a random oracle $\mathsf{H}_{\mathsf{ros}}$ and a set of arbitrary functions $f_j, j \in \{1, \ldots, \ell + 1\}$, where $f_j(\mathbb{Z}_q^\ell) \to \mathbb{Z}_q$, the goal is to find coefficients $a_i \in \mathbb{Z}_q, i \in [\ell]$ such that

$$a_1 c_1 + a_2 c_2 + \ldots + a_\ell c_\ell = \mathsf{H}_{\mathsf{ros}}(f_{\ell+1}((a_i)_{i \in [\ell]}, \mathsf{aux}))$$

where $c_j = \mathsf{H}_{\mathsf{ros}}(f_j((a_i)_{i \in [\ell]}, \mathsf{aux}))$. It is always possible to find the first $\ell - 1$ values simply by setting $a_i = 1, i \in [\ell - 1]$; the challenge is to find the ℓ^{th} solution.

Wagner [Wag02] and Benhamouda et al. [BLL+21] showed how to solve the ROS problem in subexponential time, and demonstrated its applicability to the forgeability in a range of blind and multi-party signature schemes—including blind and threshold Schnorr signatures—assuming a concurrent adversary, which may open an arbitrary number of signing sessions simultaneously. The proofs for prior multi-party Schnorr signature schemes shown to be insecure by Drijvers et al. [DEF+19] and Benhamouda et al. [BLL+21] either considered only a sequential adversary which opened only one signing session at a time, or overlooked subtle errors in the proof.

2.8 Network Communication Channels

Underlying any multi-party scheme is the requirement that participants send and receive messages over a network channel. We now describe two different types of channels that we employ later for defining and constructing a distributed key generation protocol, and additional roles and assumptions later used in our threshold signature schemes.

Synchronicity. In a synchronous model of communication, we assume all honest parties wait to proceed to a subsequent round until they have received the expected inputs from all other participants from a prior round. Additionally, we assume that the network is reliable, and is guaranteed to deliver all messages to the intended recipients within a fixed amount of time known to all parties.

Broadcast channels. A broadcast channel sends one message to all other players, Messages sent and received over a broadcast channel are denoted as inb_i, which is the set of broadcast messages that is received as input for the i^{th} protocol round. The vector inb_i consists of broadcast messages $\mathsf{bmsg}_1, \ldots, \mathsf{bmsg}_n$, sent by participants $1, \ldots, n$. Participant k is the sender of a broadcast message bmsg_k, and all other participants are the receivers of this message.

Broadcast channels have the following properties:

1. *Consistent.* Each participant has the same view of the message sent over the channel.
2. *Authenticated.* Players know that the message was in fact sent by the claimed sender. In practice, this requirement is often fulfilled by a PKI.
3. *Reliable Delivery.* Player i knows that the message it sent was in fact received by the intended participants.
4. *Unordered.* The channel does *not* guarantee ordering of messages.

Peer-to-peer channels. Messages sent and received over this channel are denoted as inp_i and outp_i, where inp_i is the vector of peer-to-peer messages that is received as input for protocol round i, and outp_i is the vector of peer-to-peer messages that is output at the end of network round i. The channels inp_i and outp_i consist of peer-to-peer messages which we denote as $\mathsf{pmsg}_{i,j}$, where here, participant i is the sender and participant j is the receiver.

Peer-to-peer channels are authenticated, reliable, and unordered. Additionally, peer-to-peer channels are *confidential*; i.e., only participants i and j are allowed to know the contents of $\mathsf{pmsg}_{i,j}$.

2.9 Additional Roles and Adversaries

In this section, we discuss additional considerations for adversarial power in the multi-party setting. We also discuss an additional role for multi-party protocols beyond a participant.

Rushing Adversary. A **rushing adversary** is one that "speaks last"; i.e., it waits to receive inputs from honest participants before publishing its own. Such an adversary models the capabilities of a real-world adversary, as threshold schemes assume any participant can be malicious, and honest participants may not be able to distinguish corrupt from honest parties. Importantly, a rushing adversary may be able to choose its input *adaptively* after observing inputs from all other participants.

Security definitions consider such an adversary, by allowing the adversary to query for honest participants' contributions in a round before publishing its own. We consider rushing adversaries in the proofs of security given in Chapters 3, 4, and 5.

Coordinator Role. In lieu of a broadcast channel, an entity that plays the role of a coordinator may instead be used. In this setting, the coordinator would be trusted to reliably route messages. However, even if the coordinator failed to do so, for the schemes presented in this thesis, the outcome would only result in a denial of service (i.e., the coordinator cannot impact the unforgeability of the scheme). This fact remains true even if the coordinator were to drop or re-order messages, or send different messages to different participants.

We consider the coordinator role in Chapters 3 and 5 as a mechanism to reliably route messages among signing participants.

Chapter 3

FROST: A Flexible, Round-Optimized, Schnorr Threshold Signature Scheme

Threshold signature schemes are a cryptographic primitive to facilitate joint ownership over a private key by a set of participants, such that a threshold number of participants must cooperate to issue a signature that can be verified by a single public key. Threshold signatures are useful across a range of settings that require distribution of trust among a set of equally trusted parties.

Similarly to signing operations in a single-party setting, some implementations of threshold signature schemes require performing signing operations at scale and under heavy load. For example, threshold signatures can be used by a set of signers to authenticate financial transactions in cryptocurrencies [GGK+15], or to sign a network consensus produced by a set of trusted authorities [MOT+11]. In both of these examples, as the number of signing parties or signing operations increases, the number of communication rounds between participants required to produce the joint signature becomes a performance bottleneck, in addition to the increased load experienced by each signer. This problem is further exacerbated when signers utilize network-limited devices or unreliable networks for transmission, or protocols that wish to allow signers to participate in signing operations asynchronously. As such, optimizing the network overhead of signing operations is highly beneficial to real-world applications of threshold signatures.

Today in the literature, the best threshold signature schemes are those that rely on pairing-based cryptography [BLS04, BDN18], and can perform signing operations in a

single round among participants. However, relying on pairing-based signature schemes is undesirable for some implementations in practice, such as those that do not wish to introduce a new cryptographic assumption, or that wish to maintain backwards compatibility with an existing signature scheme such as Schnorr signatures. Surprisingly, today's best non-pairing-based threshold signature constructions that produce Schnorr signatures with unlimited concurrency [SS01] require at least three rounds of communication during signing operations, whereas constructions with fewer network rounds [GJKR03] must limit signing concurrency to protect against a forgery attack by Drijvers et al. [DEF+19]. In particular, these schemes are insecure against ROS (Random inhomogeneities in a Overdetermined Solvable system of linear equation) attacks [DEF+19, BLL+21]. We discuss the ROS problem more in Section 2.7.

In this chapter, we present FROST, a Flexible Round-Optimized Schnorr Threshold signature scheme that addresses the need for efficient threshold signing operations while improving upon prior constructions to ensure strong security properties *without* limiting the parallelism of signing operations. FROST can be used as either a two-round protocol where signers send and receive two messages in total, or optimized to a (non-broadcast) single-round signing protocol with a pre-processing stage. FROST achieves improved efficiency in the optimistic case that no participant misbehaves. However, in the case where a misbehaving participant contributes malformed values during the protocol, honest parties can identify and exclude the misbehaving participant, and re-run the protocol.

The flexible design of FROST lends itself to supporting a number of practical use cases for threshold signing. Because the preprocessing round can be performed separately from the signing round, signing operations can be performed *asynchronously*; once the preprocessing round is complete, signers only need to receive and eventually reply with a single message to create a signature. Further, while some threshold schemes in the literature require all participants to be active during signing operations [GJKR03, DJN+22], and refer to the threshold property of the protocol as merely a security property, FROST allows any threshold number of participants to produce valid signatures. Consequently, FROST can support use cases where a subset of participants (or participating devices) can remain offline, a property that is often desirable for security in practice.

Contributions. In this chapter, we present the following contributions.

- We review related threshold signature schemes and present a detailed analysis of their performance and designs.
- We present FROST, a Flexible Round-Optimized Schnorr Threshold signature scheme. FROST improves upon prior Schnorr threshold signatures by defining a

signing protocol that can be optimized to a (non-broadcast) single-round operation with a preprocessing stage. Unlike many prior Schnorr threshold schemes, FROST remains secure against known forgery attacks without limiting the concurrency of signing operations.

- We then present a proof of security and correctness for FROST. Note that subsequent works have also given a direct proof for FROST [BCK+22] to the One-More Discrete Logarithm (ℓ-OMDL) problem; we present a slightly modified variant which shows a reduction to the Algebraic One-More Discrete Logarithm (ℓ-AOMDL) problem.

3.1 Related Work

In this section, we present related work to FROST. We consider work published prior to FROST, as well as subsequent work. Importantly, all prior related Schnorr threshold signatures are either vulnerable to a forgery attack against a concurrent adversary [DEF+19], or do not allow for single-round online signing.

3.1.1 Prior Related Work

We now review prior threshold schemes with a focus on Schnorr-based designs, and split our review into robust and non-robust schemes. Robust schemes ensure that so long as t participants correctly follow the protocol, the protocol is guaranteed to complete successfully, even if a subset of participants (at most $n - t$) contribute malformed shares. Conversely, designs that are not robust simply abort after detecting any participant misbehaviour.

Robust Threshold Schemes. Stinson and Strobl [SS01] present a threshold signature scheme producing Schnorr signatures, using the modification of Pedersen's DKG presented by Gennaro et al. [GJKR07] to generate both the secret key s during key generation as well as the random nonce k for each signing operation. This construction requires at minimum four rounds for each signing operation (assuming no participant misbehaves): three rounds to perform the DKG to obtain k, and one round to distribute signature shares and compute the group signature. Each round requires participants to send values to every other participant.

Gennaro et al. [GJKR03] present a threshold Schnorr signature protocol that uses Pedersen's DKG [Ped91] to generate both s during key generation and the random nonce k for signing operations. However, this scheme is insecure against ROS attacks [DEF+19].

Pedersen's DKG requires two rounds among participants, and so this construction requires an additional round for signing. Each round requires that all participants send values to all other participants. The authors also discuss an optimization that leverages a *signature aggregator* role, an entity trusted to gather signatures from each participant, perform validation, and publish the resulting signature, a role we also adopt in our work. In their optimized variant, participants can perform Pedersen's DKG to generate multiple k values in a pre-processing stage independently of performing signing operations. In this variant, to compute ℓ signatures, signers first perform two rounds of ℓ parallel executions of Pedersen's DKG, thereby generating ℓ random nonces. The signers can then store these pre-processed values to later perform ℓ single-round signing operations.

Importantly, the signing operation defined by FROST *does not* perform a DKG during signing operations as is done in both of the above schemes [SS01, GJKR03], but instead make use of additive secret sharing and share conversion. Consequently, FROST trades off robustness for more efficient signing operations, such that a misbehaving participant can cause the signing operation to abort. However, such a tradeoff is practical to many real-world settings.

Non-Robust Threshold Schemes. FROST is not unique in trading off favouring increased network efficiency over robustness. Gennaro and Goldfeder [GG18] present a threshold ECDSA scheme that similarly requires aborting the protocol in the case of participant misbehaviour. Their signing construction uses a two-round DKG to generate the nonce required for the ECDSA signature, leveraging additive-to-multiplicative share conversion. This DKG has been also applied in a Schnorr threshold scheme context to generate the random nonce for more efficient distributed key generation operations [KZe19] in combination with threshold Schnorr signing operations [SS01]. In later work [GG20], Gennaro and Goldfeder define an optimization to a single-round ECDSA signing operation with a preprocessing stage, which assumes the protocol will abort in the case of failure or participant misbehaviour. Their end-to-end protocol with identifiable aborts has eight network rounds, six of which require broadcasting to all other signing participants, and two of which require performing pairwise multiplicative-to-additive share conversion protocols. Further, while the protocol can be optimized into a preprocessing phase, the choice of the signing coalition must be determined at the time of preprocessing. FROST defines a more efficient preprocessing phase as secret nonces can be generated in a distributed manner in the preprocessing phase entirely non-interactively. Further, participants can "mix" preprocessed values across different signing coalitions, as FROST requires that the choice for the signing coalition be made only during the second and final round of signing.

Recent work by Damgård et al. [DJN+22] defines an efficient threshold ECDSA construction that similarly to FROST requires aborting in the case of misbehaviour. How-

ever, the security of their scheme requires the majority of participants to be honest, and $n \geq 2t+1$. Further, their scheme requires all n participants take part in signing operations, where the threshold t is simply a security parameter.

Similarly to FROST, Abidin, Aly, and Mustafa [AAM20] present a design for authentication between devices, and use additive secret sharing to generate the nonce for Schnorr signatures in a threshold setting, a technique also used by FROST. However, the authors do not consider the Drijvers attack [DEF+19] and consequently their design is similarly limited to restricted levels of parallelism; i.e., it is not secure against a concurrent adversary. Further, their design does not include validity checks for responses submitted by participants when generating signatures and consequently does not detect nor identify misbehaving participants.

FROST improves upon prior work in Schnorr threshold schemes by providing a single-round signing variant with a preprocessing stage that is agnostic to the choice of the signing coalition. Further, the number of signing participants in FROST is required to be simply some $t \leq n$, while remaining secure against the Drijvers attack [DEF+19] and other ROS [Sch06] attacks.

3.1.2 Subsequent Work

Since the publication of FROST, the insecurity of the ROS problem has been underscored [BLL+21], and additional concurrently secure threshold Schnorr signature schemes have been proposed. We now review subsequent works.

Efficient Algorithms for Solving ROS Problems. Whereas prior works [DEF+19, Wag02] established the existence of a sub-exponential time algorithm for solving the ROS (Random inhomogeneities in a Overdetermined Solvable system of linear equation) problem, (first introduced by Schnorr [Sch06]), Benhamouda et al. [BLL+21] give a polynomial-time algorithm. We discuss the ROS problem more in Section 2.7. Benhamouda et al. then show how the ROS problem can be applied to blind and threshold Schnorr signatures, to produce a forgery in polynomial time.

Concurrently Secure Threshold Schnorr Signatures. FROST2 [BCK+22] is an optimized version of FROST that reduces the number of exponentiations required for signing from t to one. A further optimization of FROST2, called FROST3, is introduced by Ruffing et al. [RRJ+22], and proven secure with a DKG by Chu et al. [CGRS23]. All

three of FROST, FROST2, FROST3 are proven statically secure under the algebraic one-more discrete logarithm assumption (ℓ-AOMDL) [BCK+22].

Lindell presents a three-round threshold Schnorr signature [Lin24] with the goal of defining a scheme that is both secure against ROS attacks and secure in the random oracle model under the discrete logarithm assumption only. Static security is modeled in the universal composability framework (UC) [Can01] and therefore captures a concurrent adversary.

Sparkle [CKM23] is a three-round threshold Schnorr signature scheme that is proven secure under $t-1$ adaptive corruptions. In order to achieve full adaptive security, the authors rely on the algebraic one-more discrete logarithm assumption (ℓ-AOMDL) in the AGM and ROM. When the adversary is limited to making $t-1/2$ corruptions, however, the authors prove the adaptive security of Sparkle under only ℓ-AOMDL in the ROM, by a reduction that rewinds.

Makriyannis [Mak22] defines a commit-reveal threshold Schnorr signature scheme similar to Sparkle, called Classic S., and proves security with respect to an idealized notion of threshold signatures by Canetti et al. [CGG+20]. They consider adaptive security but employ the guessing argument that incurs exponential tightness loss relative to the number of parties.

3.2 FROST Signatures

We next present the FROST signature scheme, which is a threshold Schnorr signature, outputting a signature that is valid under the Schnorr signature verification algorithm [Sch91]; see Section 2.3.1.

3.2.1 Key Generation

The FROST threshold signature scheme assumes that all parties are given Shamir secret shares of a signing key $\mathsf{sk} \in \mathbb{Z}_q$, such that the public key is $\mathsf{pk} = g^{\mathsf{sk}}$. In other words, we assume each participant $i \in [n]$ receives the secret signing key share sk_i, such that $\{(i, \mathsf{sk}_i)\}_{i=1}^{n} \xleftarrow{\$} \mathsf{Shamir.Share}(\mathsf{sk}, n, t)$. Key generation can be performed via a trusted dealer, or in a distributed manner via a distributed key generation (DKG) protocol.

$\mathsf{Setup}(1^\lambda)$

$(\mathbb{G}, q, g) \xleftarrow{\$} \mathsf{GroupGen}(1^\lambda)$

// *Select two hash functions* $\mathsf{H}_{\mathsf{non}}, \mathsf{H}_{\mathsf{sig}}$

$\mathsf{par} \leftarrow ((\mathbb{G}, q, g), \mathsf{H}_{\mathsf{non}}, \mathsf{H}_{\mathsf{sig}})$

return par

// par *is given implicitly*

// *as input to all other algorithms*

$\mathsf{KeyGen}(n, t)$

// *Shown in the centralized setting,*

// *but a DKG can likewise be used.*

$\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q; \mathsf{pk} \leftarrow g^{\mathsf{sk}}$

$\{(i, \mathsf{sk}_i)\}_{i\in[n]} \xleftarrow{\$} \mathsf{Shamir.Share}(\mathsf{sk}, n, t)$

// *Shamir secret sharing of* sk

for $i \in [n]$ **do**

  $\mathsf{pk}_i \leftarrow g^{\mathsf{sk}_i}$

return $(\mathsf{pk}, \{\mathsf{pk}_i\}_{i\in[n]}, \{\mathsf{sk}_i\}_{i\in[n]})$

$\mathsf{Sign}_1()$ // *For party* k

$d_k, e_k \xleftarrow{\$} \mathbb{Z}_q^2$

$D_k \leftarrow g^{d_k};\ E_k \leftarrow g^{e_k};$

$\mathsf{st}_k \leftarrow (D_k, d_k, E_k, e_k)$

$\rho_{1,k} \leftarrow (D_k, E_k)$

return $(\rho_{1,k}, \mathsf{st}_k)$

$\mathsf{Sign}_2(\mathsf{st}_k, \mathsf{sk}_k, m, C, \mathcal{PM}_1)$ // *For party* k

// Sign_2 must be called at most once per st_k

parse $(D'_k, d_k, E'_k, e_k) \leftarrow \mathsf{st}_k$

parse $\{D_i, E_i\}_{i\in C} \leftarrow \mathcal{PM}_1$

return $\perp$ **if** $(D'_k, E'_k) \notin \{(D_i, E_i)\}_{i\in C}$

for $i \in C$ **do**

  $a_i \leftarrow \mathsf{H}_{\mathsf{non}}(i, \mathsf{pk}, m, \{(i, D_i, E_i)\}_{i\in C})$

$R \leftarrow \prod_{i\in C} D_i \cdot {E_i}^{a_i}$

$c \leftarrow \mathsf{H}_{\mathsf{sig}}(R, \mathsf{pk}, m)$

$z_k \leftarrow d_k + e_k a_k + c\lambda_k \mathsf{sk}_k$

// λ_k is the k^{th} Lagrange coefficient for C

$\rho_{2,k} \leftarrow z_k$

return $\rho_{2,k}$

$\mathsf{Combine}(C, m, \mathcal{PM}_1, \mathcal{PM}_2)$

parse $\{(D_i, E_i)\}_{i\in C} \leftarrow \mathcal{PM}_1, \{z_i\}_{i\in C} \leftarrow \mathcal{PM}_2$

for $i \in C$ **do**

  $a_i \leftarrow \mathsf{H}_{\mathsf{non}}(i, \mathsf{pk}, m, \{(i, D_i, E_i)\}_{i\in C})$

$R \leftarrow \prod_{i\in C} D_i \cdot {E_i}^{a_i};\ z \leftarrow \sum_{i\in C} z_i;\ \sigma \leftarrow (R, z)$

return σ

$\mathsf{Verify}(\mathsf{pk}, m, \sigma)$

parse $(R, z) \leftarrow \sigma$

$c \leftarrow \mathsf{H}_{\mathsf{sig}}(R, \mathsf{pk}, m)$

if $R \cdot \mathsf{pk}^c = g^z$

  return 1

else return 0

Figure 3.1: The FROST threshold signature scheme

3.2.2 The FROST Signature Scheme

We now introduce the signing protocol for FROST. This operation builds upon known techniques in the literature [AAM20, SU17] by employing additive secret sharing and share conversion to non-interactively generate the nonce value for each signature. However, signing operations in FROST additionally leverage a novel binding technique to avoid known forgery attacks without limiting concurrency. Verification of signatures is identical to (single-party) Schnorr signatures. As discussed in Remark 2 in Section 2.6, giving the coalition C and message m can be safely deferred to Round 2.

We present FROST signing in two parts: a pre-processing phase and a single-round signing phase. However, these stages can be combined into a two-round protocol if desired.

At a high level, the ROS-style attack of Drijvers et al. [DEF+19] requires the adversary to either see the victim's commitment values in the first round before selecting their own commitment, or to adaptively choose the message to be signed, so that the adversary can manipulate the resulting challenge for the set of participants performing a group signing operation. To prevent this attack without limiting concurrency, FROST "binds" each participant's response to a specific message as well as the set of participants and their commitments used for that particular signing operation. In doing so, combining responses over different messages or participant/commitment pairs results in an invalid signature, thwarting ROS-style attacks such as those of Drijvers et al.

Coordinator Role. Our presentation of FROST assumes a coordinator role to choose the set $C \subseteq \{1, \ldots, n\}$ of signers, where $t \leq |C| \leq n$. Note that in Figure 3.1 and in this description, we assume $|C| = t$, for how each λ_i is defined, per Section 2.1.1. However, the protocol can be defined for the more general case of $t \leq |C|$, such as by changing how Lagrange coefficients are defined, or requiring the coordinator to pick a subset of t partial signatures. The coordinator is trusted to not perform denial of service attacks, but otherwise is untrusted. Even if the coordinator were fully malicious, FROST remains unforgeable. The coordinator can be a stand-alone entity, or may also be a signer as well.

First Signing Round Sign_1. As a reminder, the public parameters par are implicitly given as input to all algorithms and protocols. The coalition C is required to be ordered to ensure consistency.

In this setting, ℓ determines the number of nonces that are generated and their corresponding commitments that are published in a single preprocessing step. Implementations that do not wish to cache commitments can instead use a two-round signing protocol, where

participants publish a single commitment in the first round. Note that preprocessing for FROST is *not* reusable, meaning that each commitment generated in the preprocessing stage must be used at most once during an online signing stage.

Each participant $i \in C$ begins by generating a list of *single-use* private nonce pairs and corresponding public commitment shares $((d_{ij}, D_{ij} = g^{d_{ij}}), (e_{ij}, E_{ij} = g^{e_{ij}}))_{j=1}^{\ell}$, where j is a counter that identifies the next nonce/commitment share pair available to use for signing. Each participant $i \in C$ then publishes its set of commitments $(D_{ij}, E_{ij})_{j=1}^{\ell}$. These sets are then stored by any entity that might perform the coordinator role during signing.

This first signing round can be performed in a batched manner, as an optional preprocessing stage. The first signing round can be batched and performed in an offline manner, as the message to be signed is required as input only in the second (online) signing round.

Second Signing Round Sign_2. At the beginning of the signing protocol in Figure 3.1, the coordinator selects a coalition of parties $C \subseteq [n], |C| \geq t$ to perform signing. The coordinator then selects the next available commitment $(D_i, E_i) : i \in C$, which are later used to generate a secret share to a random commitment R for the signing group.[1] The coordinator then creates the set B, where B is the ordered list of tuples $\langle (D_i, E_i) \rangle_{i \in C}$. The coordinator then sends (m, B) to every participant $i \in C$.

After receiving (m, B) from the coordinator to initialize a signing operation, each participant checks that m is a message they are willing to sign. Then, using m and B, all participants derive the "binding values" $a_i, i \in C$ such that $a_i = \mathsf{H}_{\mathsf{non}}(i, \mathsf{pk}, m, B)$, where $\mathsf{H}_{\mathsf{non}}$ is a hash function whose outputs are in $\mathbb{Z}_q^*$.

Each participant then computes the commitment R_i for each participant in C by deriving $R_i = D_i \cdot (E_i)^{a_i}$. Doing so binds the message, the set of signing participants, and each participant's commitment to each signature share. This binding technique thwarts the attack of Drijvers et al.[DEF+19], as attackers cannot combine signature shares across disjoint signing operations or permute the set of signers or published commitments for each signer.

The commitment for the set of signers is then simply $R = \prod_{i \in C} R_i$. As in single-party Schnorr signatures, each participant computes the challenge $c = \mathsf{H}_{\mathsf{sig}}(R, \mathsf{pk}, m)$.

The resulting secret nonce is $k = \sum_{i \in C} k_i$, where each $k_i = d_i + e_i \cdot a_i$, and (d_i, e_i) correspond to the $(D_i = g^{d_i}, E_i = g^{e_i})$ values published during the preprocessing stage.

[1] Each participant contributes to the group commitment R, which corresponds to the commitment g^k to the nonce k in step 1 of the single-party Schnorr signature scheme in Section 2.3.1.

Each participant's response z_i to the challenge can be computed using the single-use nonces (d_i, e_i) and the long-term secret shares sk_i, converted to additive form:

$$z_i = d_i + (e_i \cdot a_i) + \lambda_i \cdot \mathsf{sk}_i \cdot c$$

Combine. The coordinator (optionally) checks the consistency of each participant's reported z_i with their commitment share (D_i, E_i) and their public key share pk_i. If every participant issued a correct z_i, the group's response is $z = \sum_{i \in C} z_i$, and the group signature on m is $\sigma = (R, z)$. This signature is verifiable to anyone performing a standard Schnorr verification operation with pk as the public key (Section 2.3.1).

Handling Ephemeral Outstanding Shares. Because each nonce and commitment share generated during the first round of signing described in Figure 3.1 must be used *at most once*, participants should delete these values after using them in a signing operation,. An accidentally reused (d_{ij}, e_{ij}) can lead to exposure of the participant's long-term secret sk_i.

However, if the coordinator chooses to re-use a commitment set (D_i, E_i) during the signing protocol, doing so simply results in the participant P_i aborting the protocol, and consequently does not increase the power of the coordinator.

3.2.3 Correctness of FROST

Signatures in FROST are constructed from two polynomials; the first polynomial $f_1(x)$ defines the secret sharing of the private signing key sk (such that $\mathsf{pk} = g^{\mathsf{sk}}$) and the second polynomial $f_2(x)$ defines the secret sharing of the nonce k such that $k = \sum_{i \in C} d_i + e_i \cdot a_i$ using the associated public data (m, B) to determine a_i. During the key generation phase, the first polynomial $f_1(x)$ is generated by the dealer picking its coefficients at random, such that the secret key shares are $\mathsf{sk}_i = f_1(i)$ and the secret key is $\mathsf{sk} = f_1(0)$.

During the signature phase, each of the participants in the coalition C selected for signing use a pair of nonces (d_i, e_i) to define a degree $|C|-1$ polynomial $f_2(x)$, interpolating the values $(i, \frac{d_i + e_i \cdot \mathsf{H}_{\mathsf{non}}(i, \mathsf{pk}, m, B)}{\lambda_i})$, such that $f_2(0) = \sum_{i \in C} d_i + e_i \cdot a_i$.

Then let $f_3(x) = f_2(x) + c \cdot f_1(x)$, where $c = \mathsf{H}_{\mathsf{sig}}(R, \mathsf{pk}, m)$. Now z_i equals $d_i + (e_i \cdot a_i) + \lambda_i \cdot \mathsf{sk}_i \cdot c = \lambda_i(f_2(i) + c \cdot f_1(i)) = \lambda_i f_3(i)$, so $z = \sum_{i \in C} z_i$ is simply the Lagrange interpolation of $f_3(0) = (\sum_{i \in C} d_i + e_{ij} \cdot a_i) + c \cdot \mathsf{sk}$. Because $R = g^{\sum_{i \in C} d_i + e_i \cdot a_i}$ and $c = \mathsf{H}_{\mathsf{sig}}(R, \mathsf{pk}, m)$, (R, z) is a correct Schnorr signature on m.

3.3 Security of FROST

We now present a direct proof of static (weak) unforgeability for the FROST signature construction given in Figure 3.1. This proof builds on prior proofs of security for FROST [BCK+22], but gives a reduction to ℓ-AOMDL, as opposed to ℓ-OMDL. As discussed in Section 2.1.2, ℓ-AOMDL is a falsifiable and hence weaker assumption than ℓ-OMDL. Note that prior work also established strong unforgeability of FROST [BCK+22], but with respect to ℓ-OMDL.

Our proof assumes a centralized key generation mechanism.

Theorem 3. *FROST is statically unforgeable against an adversary $\mathcal{A}$ that is allowed to corrupt up to $t-1$ participants, assuming the ℓ-AOMDL assumption holds for GroupGen, in the ROM. In other words, for every static unforgeability adversary $\mathcal{A}$ that wins the (static) unforgeability game as given in Definition 2.6.2, there exists a PPT adversary $\mathcal{D}$ that wins the ℓ-AOMDL game.*

The advantage of $\mathcal{A}$ is bounded by

$$\mathsf{Adv}^{\mathsf{uf}}_{\mathsf{FROST},\mathcal{A}}(\lambda, n, t, t-1) \leq \sqrt{q_r \cdot \mathsf{Adv}^{\mathrm{aomdl}}_{\mathcal{D}}(\lambda)} + \mathsf{negl}(\lambda) \tag{3.1}$$

where $\ell = 2q_s$, such that $q_r = q_n + q_h + q_s + 1$ is the number of times $\mathcal{A}$ is allowed to query the random oracle, where q_n is the number of times $\mathcal{A}$ is allowed to query $\mathsf{H}_{\mathsf{non}}$, q_h is the number of times $\mathcal{A}$ is allowed to query $\mathsf{H}_{\mathsf{sig}}$, and q_s is the number of times $\mathcal{A}$ is allowed to query its signing oracle.

To prove Theorem 3, we rely on two lemmas.

Lemma 2. *There exists an algorithm SIM which can simulate FROST with respect to an ℓ-AOMDL instance to an adversary $\mathcal{A}$ playing against the unforgeability game as in Figure 2.8. In other words, $\mathcal{A}$ has negligible additional advantage in winning the unforgeability game when interacting with SIM that it does when playing against FROST directly.*

Concretely, in the setting where $\mathcal{A}$ is allowed up to $t-1$ corruptions, then

$$\mathsf{Adv}^{\mathsf{uf}}_{\mathsf{FROST},\mathcal{A}}(\lambda, n, t, t-1) \leq \Pr[\ \mathsf{SIM}^{\mathcal{A}}(\mathsf{par}, n, t, t-1, \hat{X}) \neq \perp\ ] + \frac{q_r}{q} + \frac{q_r^2}{2q} \tag{3.2}$$

Lemma 3. *Given any PPT adversary $\mathcal{A}$ that is allowed to corrupt $t-1$ participants and wins the unforgeability game against FROST, and a simulating algorithm SIM which can*

simulate FROST, then there exists a PPT ℓ-AOMDL *adversary* $\mathcal{D}$ *that can use* $\mathcal{A}$ *as a subroutine to solve for its* ℓ-AOMDL *challenge.*

Concretely,

$$\mathsf{Adv}^{\mathrm{aomdl}}_{\mathcal{D}^{\mathsf{SIM}}}(\mathsf{par}, n, t, t-1, \hat{X}) \geq \frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathsf{FROST},\mathcal{A}}(\lambda, n, t, t-1)^2}{q_r} - \frac{2q_r}{q} - \frac{3q_r^2}{q} - \frac{2q_s}{q} \tag{3.3}$$

Proof of Lemma 2. We give the proof by one game hop and then describing the simulator SIM.

Game 0. This is the unforgeability game as defined in Definition 2.6.2, instantiated with FROST.

Let $\mathcal{A}$ be an adversary playing the static unforgeability game. $\mathcal{A}$ is allowed to query $\mathsf{H}_{\mathsf{non}}$ q_n number of times, $\mathsf{H}_{\mathsf{sig}}$ q_h number of times, and is allowed to make q_s signing queries. Without loss of generality, we assume $\mathcal{A}$ queries $\mathsf{H}_{\mathsf{sig}}$ on its forgery (R^*, pk, m^*). Finally, let $q_r = q_n + q_h + q_s + 1$.

Let W_0 be the event that $\mathcal{A}$ wins Game 0. Then, the advantage of $\mathcal{A}$ is simply

$$\mathsf{Adv}^{\mathrm{uf}}_{\mathsf{FROST},\mathcal{A}}(\lambda) = \Pr[W_0] \tag{3.4}$$

Game 1. The only difference between Game 0 and Game 1 is as follows. If $\mathcal{A}$ queries $\mathsf{H}_{\mathsf{sig}}$ or $\mathsf{H}_{\mathsf{non}}$ on any input that produces a colliding output with another query, then the environment aborts.

Difference between Game 1 and Game 0. Let W_1 be the event that $\mathcal{A}$ wins Game 1. Because we model $\mathsf{H}_{\mathsf{sig}}$ and $\mathsf{H}_{\mathsf{non}}$ as random oracles, then

$$|\Pr[W_1] - \Pr[W_0]| \leq \frac{q_r^2}{2q} \tag{3.5}$$

Game 2. We now describe a simulating algorithm SIM that simulates FROST to $\mathcal{A}$.

Setup. SIM is initialized with public parameters $\mathsf{par} = (\mathbb{G}, q, g)$, threshold parameters the total number of parties n, threshold t, and corruption threshold $t-1$. Additionally, SIM is

initialized with the instance $\hat{X}$, which is comprised of a set of $2q_s + 1$ ℓ-AOMDL challenges $\{X_0, \ldots, X_{2q_s}\}$, and a set of values $\{h_1, \ldots, h_{q_r}\}$ to program its random oracles. SIM can query the ℓ-AOMDL solution oracle $\mathcal{O}^{dl}$ up to q_s times.

SIM then initializes the following values:

1. tables $Q_{\mathsf{non}} \leftarrow \emptyset$, $Q_{\mathsf{sig}} \leftarrow \emptyset$ to simulate random oracles $\mathsf{H}_{\mathsf{non}}, \mathsf{H}_{\mathsf{sig}}$;
2. table $Q_1 \leftarrow \emptyset$ to maintain state during its simulation;
3. tables W, W' to maintain open and closed sessions,
4. the table $Q_m \leftarrow \emptyset$ to maintain the set of messages queried by $\mathcal{A}$ to honest signers,
5. the counter $\bar{\mathsf{c}} \leftarrow 1$ to keep track of which ℓ-AOMDL challenges have yet been used.

SIM runs $\mathcal{A}(\mathsf{par}, n, t-1, t)$ on the public parameters par. $\mathcal{A}$ chooses the initial set of corrupted parties $\mathsf{corrupt} \leftarrow [n], |\mathsf{corrupt}| \leq t-1$. SIM sets $\mathsf{honest} \leftarrow [n] \setminus \mathsf{corrupt}$ and must reveal the secret keys of the corrupted parties to $\mathcal{A}$, which it does in the next step.

Simulating KeyGen. To simulate key generation where SIM is the trusted dealer, SIM does the following:

1. It sets $\mathsf{pk} = X_0$.
2. To simulate key generation for the signing keys, SIM first picks $t-1$ secret key shares for the corrupt parties $(\mathsf{sk}_i)_{i \in \mathsf{corrupt}} \xleftarrow{\$} \mathbb{Z}_q$, and defines their corresponding public keys $\mathsf{pk}_i \leftarrow g^{\mathsf{sk}_i}$ for all $i \in \mathsf{corrupt}$.
3. SIM then derives the public key shares $\mathsf{pk}_j, j \in \mathsf{honest}$ for the remaining honest parties in the exponent, as follows:

$$\mathsf{pk}_j = (\mathsf{pk})^{L_0(j)} \cdot \prod_{i \in \mathsf{corrupt}} g^{\mathsf{sk}_i L_i(j)} \tag{3.6}$$

   where $L_i(x)$ is the i^{th} Lagrange polynomial defined by $\mathsf{corrupt} \cup \{0\}$.

Then, SIM runs $\mathcal{A}^{\mathcal{O}^{\mathsf{Sign}_1}, \mathcal{O}^{\mathsf{Sign}_2}}(\mathsf{pk}, \{\mathsf{pk}_j\}_{j \in [n]}, \{\mathsf{sk}_i\}_{i \in \mathsf{corrupt}}, \mathsf{st}_A)$, and responds to its oracle queries, as we describe next.

Simulating Random Oracles. To simulate $\mathcal{A}$'s random oracle queries, SIM programs $\mathsf{H}_{\mathsf{non}}, \mathsf{H}_{\mathsf{sig}}$ with values from its input $\{h_1, \ldots, h_{q_r}\}$, as follows.

- $\underline{\mathsf{H}_{\mathsf{non}}}$: When $\mathcal{A}$ queries $\mathsf{H}_{\mathsf{non}}$ on $(k, \mathsf{pk}, m, \{(i, D_i, E_i)\}_{i \in C})$, To respond, SIM does the following:

  1. First, SIM checks if $Q_{\mathsf{non}}[(k, \mathsf{pk}, m, \{(i, D_i, E_i)\}_{i \in C})] \neq \perp$. If so, SIM simply returns $Q_{\mathsf{non}}[(k, \mathsf{pk}, m, \{(i, D_i, E_i)\}_{i \in C})]$,
  2. Otherwise, for all $j \in C$, SIM does the following:
     (a) SIM sets $a_j \leftarrow h_\ell$ and $Q_{\mathsf{non}}[(j, \mathsf{pk}, m, \{(i, D_i, E_i)\}_{i \in C})] = a_j$ where $h_\ell, \ell \in [q_r]$ is the next-unused hash value provided to SIM on initialization.
  3. Then, SIM derives $R \leftarrow \prod_{i \in C} D_i \cdot E_i^{a_i}$.
  4. SIM checks if $Q_{\mathsf{sig}}[(R, \mathsf{pk}, m)] = \perp$; if the check does not hold, SIM aborts with $\mathsf{BadEvent}_1$.
  5. Otherwise, SIM sets $c \leftarrow h_{\ell+1}$, where h_ℓ is the next-unused random oracle output, and stores $Q_{\mathsf{sig}}[(R, \mathsf{pk}, m)] = c$.
  6. Finally, SIM returns a_k.

- $\underline{\mathsf{H}_{\mathsf{sig}}}$: When $\mathcal{A}$ queries $\mathsf{H}_{\mathsf{sig}}$ on (R, pk, m), SIM checks whether $Q_{\mathsf{sig}}[(R, \mathsf{pk}, m)] \neq \perp$; if the check does not hold, then SIM returns $Q_{\mathsf{sig}}[(R, \mathsf{pk}, m)]$. Else, SIM sets $c \leftarrow h_\ell$, where h_ℓ is the next-unused random oracle output, stores $Q_{\mathsf{sig}}[(R, \mathsf{pk}, m)] = c$ and returns c.

Simulating Signing Oracles. To simulate signing oracles, SIM does as follows:

- Round 1 ($\mathcal{O}^{\mathsf{Sign}_1}(k, \mathsf{eid})$): When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_1}$ on honest participant identifier $k \in \mathsf{honest}$ and session identifier eid, SIM does the following:

  1. Checks if this signing session has already started by checking if $(k, \mathsf{eid}) \in W$, returning $\perp$ to $\mathcal{A}$ if the check holds.
  2. Otherwise, it opens a new signing session for participant k by storing $W \leftarrow W \cup \{(k, \mathsf{eid})\}$.
  3. Next, SIM sets $D_k \leftarrow X_{\bar{\mathsf{c}}}$ and $E_k \leftarrow X_{\bar{\mathsf{c}}+1}$, where $\bar{\mathsf{c}}$ represents the next-unused value from the set of challenges $(X_1, \ldots, X_{2q_S})$. It then increments $\bar{\mathsf{c}} \leftarrow \bar{\mathsf{c}} + 2$.

4. SIM then sets $\mathsf{st}_{k,\mathsf{eid}} \leftarrow (k, \mathsf{eid}, D_k, E_k)$, and stores $Q_1[(k, \mathsf{eid})] = (D_k, E_k)$,
5. Finally, SIM returns (D_k, E_k) to $\mathcal{A}$.

- Round 2 ($\mathcal{O}^{\mathsf{Sign}_2}(k, \mathsf{eid}, m, C, \{(D_i, E_i)\}_{i \in C})$): When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_2}$ for honest participant identifier $k \in \mathsf{honest}$, session identifier eid, message m, signing set C, and set of nonce commitments $\{(D_i, E_i)\}_{i \in C}$, SIM does as follows:
  1. Checks if this signing session has already closed by checking if $(k, \mathsf{eid}) \in W'$. If the check holds, return $\perp$ to $\mathcal{A}$.
  2. Checks if this signing session has not yet been opened, by if $(k, \mathsf{eid}) \notin W$, returning $\perp$ if the check holds.
  3. Checks that $(k, \mathsf{eid}, D_k, E_k) \in Q_1$ and $Q_2[(D_k, E_k)] = \perp$ to ensure that they were indeed issued in Round 1 and have not yet been used, returning $\perp$ if the check does not hold.
  4. Otherwise, SIM follows the protocol honestly to derive each $a_i, i \in C$, and then the group commitment $R \leftarrow \prod_{i \in C} D_i \cdot {E_i}^{a_i}$ and the challenge $c \leftarrow \mathsf{H}_{\mathsf{sig}}(R, \mathsf{pk}, m)$.
  5. SIM then sets $Z_k \leftarrow D_k \cdot E_k^{a_k} \cdot {\mathsf{pk}_k}^{c\lambda_k}$, and queries the ℓ-AOMDL oracle $\mathcal{O}^{dl}$ on Z_k and coefficients $(\alpha, (\beta_i)_{i \in [2q_s+1]})$, where each coefficient is zero, with these exceptions; $\alpha = c\lambda_k \cdot \sum_{i \in \mathsf{corrupt}} \mathsf{sk}_i L'_i(k)$, and $(\beta_0 = c\lambda_k \cdot L'_0(k), \beta_\gamma = 1, \beta_{\gamma+1} = a_i)$. Here, $(\gamma, \gamma+1)$ corresponds to the ℓ-AOMDL challenges $(D_k = X_\gamma, E_k = X_{\gamma+1})$ for this signing query, where L'_i is the Lagrange polynomial for $i \in \mathsf{corrupt}$ associated with the set $\{0\} \cup \mathsf{corrupt}$, such that $\mathsf{pk}_k^c = \mathsf{pk}^{L'_0(k)c} \cdot g^{c\alpha}$.
  6. SIM then receives z_k in return from its ℓ-AOMDL oracle, which is the discrete logarithm of Z_k.
- SIM then stores $Q_2[(D_k, E_k)] = (z_k, c\lambda_k)$.
- SIM then returns z_k to $\mathcal{A}$.

Analysis of Simulation. SIM's simulation of key generation and signing is perfect. For key generation, $\mathcal{A}$ receives a valid secret key that is the point i on the polynomial f, where $\mathsf{pk}_i = g^{f(i)}$. For signing, simulation is perfect because SIM receives z_k from its ℓ-AOMDL oracle, which is the discrete logarithm of Z_k, with respect to D_k, E_k, pk_k.

SIM's simulation of the random oracles is perfect, with exception of when it aborts due to a bad event. SIM responds to random oracle queries by lazy sampling; SIM aborts due to $\mathsf{BadEvent}_1$ with probability $1/q$ for each query to $\mathsf{H}_{\mathsf{non}}$, for a total of q_r possible queries, hence; it aborts with total probability q_r/q each time SIM is exeucted.

Output. At the end of the game, $\mathcal{A}$ outputs a forgery $(m^*, \sigma^* = (R^*, z^*))$. If $g^{z^*} \neq R^* \cdot (\mathsf{pk})^{c^*}$ (meaning that the forgery is invalid), SIM outputs $\perp$. Without loss of generality, we assume that $\mathcal{A}$ queried $\mathsf{H}_{\mathsf{sig}}$ on (R^*, pk, m^*).

Otherwise, if $\mathcal{A}$'s forgery is valid, SIM outputs the tuple (ℓ, aux), where ℓ is the index corresponding to $c^* = h_\ell$, and $\mathsf{aux} = (\{\mathsf{sk}_i\}_{i \in \mathsf{corrupt}}, Q_1, (m^*, \sigma^*))$ is the tuple including the corrupted parties' secret key shares, the set of signature shares in Q_1 output by SIM on behalf of honest parties during its simulation, and the adversary's forgery with respect to $\mathsf{pk} = X_0$.

Difference between Game 1 and Game 2. The only additional advantage introduced in Game 2 is that SIM aborts on $\mathsf{BadEvent}_1$ with probability q_r/q total probability. Let W_2 be the event that $\mathcal{A}$ wins in Game 2. Then

$$|\Pr[W_2] - \Pr[W_1]| \leq \frac{q_r}{q} \tag{3.7}$$

Finishing the Proof. The advantage of $\mathcal{A}$ against SIM is given by Game 2. Combining Equations 3.4, 3.5, and 3.7 gives Equation 3.1. This completes the proof. □

Proof of Lemma 3. We now show that if there exists a PPT adversary $\mathcal{A}$ against the unforgeability of FROST and an algorithm SIM as described in Lemma 2 that simulates FROST to $\mathcal{A}$, then we can define a PPT adversary $\mathcal{D}$ that can efficiently solve for an ℓ-AOMDL challenge instance.

$\mathcal{D}$ begins by accepting parameters $\mathsf{par} = (\mathbb{G}, q, g)$ and a set of $2q_s + 1$ ℓ-AOMDL challenges $\{X_0, \ldots, X_{2q_s}\}$. It then chooses a (t, n) such that $n \geq t$.

$\mathcal{D}$ then executes the modified forking algorithm $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ as described in Section 2.2 and shown in Figure 2.2, providing as input the instance $X = (\mathsf{par}, \{X_0, \ldots, X_{2q_s}\}, n, t)$. Furthermore, $\mathcal{D}$ manages SIM's queries to the ℓ-AOMDL oracle. Each instance of SIM will "use" up to q_s queries to $\mathcal{D}$'s ℓ-AOMDL oracle, and so $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ will "use" $2q_s$ ℓ-AOMDL oracle queries in total.

$\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ then either outputs $\perp$ when it fails or the value (j, aux) when it accepts. Corollary 1 upper-bounds the probability that $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ will output $\perp$ and Lemma 2 upper-bounds the additional advantage of $\mathcal{A}$ against SIM.

We next show that when $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ outputs an accepting output (i.e., it does not output $\perp$), then $\mathcal{D}$ can solve for its ℓ-AOMDL challenge with perfect success.

Extracting the Discrete Logarithm of X_0. In the case that $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ outputs an accepting output, it outputs the tuple $(h_j, h'_j, \mathsf{aux}, \mathsf{aux}')$. From these values, $\mathcal{D}$ then parses the following:

$$\begin{aligned} (m^*, \sigma^*) &\leftarrow \mathsf{aux}, \quad (m^{**}, \sigma^{**}) \leftarrow \mathsf{aux}', \\ (R^*, z^*) &\leftarrow \sigma^*, \quad (R^{**}, z^{**}) \leftarrow \sigma^{**}, \\ h_j &\leftarrow c^*, \quad h'_j \leftarrow c^{**} \end{aligned}$$

Because h_j, h'_j are with respect to the same index j corresponding to $\mathcal{A}$'s random query for c^*, c^{**} in its first and second execution, then we know that $R^* = R^{**}$ and $m^* = m^{**}$. $\mathcal{D}$ can then solve for the discrete logarithm y to its challenge $X_0 = g^y$, as follows:

$$x_0 = \mathsf{sk} = \frac{z^* - z^{**}}{c^* - c^{**}} \tag{3.8}$$

Because Fork_m outputs $\perp$ in the case when any random oracle outputs collide (i.e., in the case when any $h_i = h'_i$), then we know that this equation is solveable.

Solving for Honest Parties' Secret Keys $\mathsf{sk}_k, k \in \mathsf{honest}$. In order to perform the next step of solving for the remaining ℓ-AOMDL challenges $X_1, \ldots, X_{2q_s}$. $\mathcal{D}$ needs to be able to derive the secret keys of each honest party. However, it can do so by using $\{\mathsf{sk}_i\}_{i \in \mathsf{corrupt}}$ and x_0, simply by deriving $\mathsf{sk}_k \leftarrow x_0 \lambda'_0 + \sum_{i \in \mathsf{corrupt}} \mathsf{sk}_i \lambda'_i$, for each $k \in \mathsf{honest}$, where λ'_i is the Lagrange coefficient associated with the set $\{0\} \cup \mathsf{corrupt}$.

Extracting the Discrete Logarithms of $X_1, \ldots, X_{2q_s}$. Finally, $\mathcal{D}$ must solve for all remaining $(D_i = X_i, E_i = X_{i+1})$ ℓ-AOMDL challenges. $\mathcal{D}$ can do so in the same manner as prior reductions to ℓ-OMDL [BCK+22], since $\mathcal{D}$ has already made its ℓ-AOMDL queries and obtained valid solutions. We give these techniques below, for completeness.

How $\mathcal{D}$ solves for the discrete logarithm of each challenge pair depends on several factors. We describe each case. First, recall after the two (successful) executions of $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$, $\mathcal{D}$ will obtain $(\mathsf{aux}, \mathsf{aux}')$, the signature pairs generated by SIM during its simulation of signing, as shown in Equation 3.9.

$$\begin{aligned} &(\mathsf{eid}_i, a_{i0}, c_{i0}, D_{i0}, E_{i0}, z_{i0}), i \in [q_s], \text{and} \\ &(\mathsf{eid}_i, a_{i1}, c_{i1}, D_{i1}, E_{i1}, z_{i1}), i \in [q_s] \end{aligned} \tag{3.9}$$

Because (1) SIM is deterministic (as SIM is run on the same random tape for each execution, apart from the values $\{h_1, \ldots, h_{q_r}\}$ to program its random oracles) and (2) each commitment pair is associated with exactly one session identifier eid_i, the commitment pairs (D_{ij}, E_{ij}) will be issued with the same eid_i each time SIM is executed. Hence, $D_{i0} = D_{i1}$ and $E_{i0} = E_{i1}$.

Consequently, the following cases depend only on the relation between (a_{i0}, a_{i1}) and (c_{i0}, c_{i1}), which in turn determine (z_{i0}, z_{i1}).

Case 1 $(a_{i0} \neq a_{i1}, c_{i0} \neq c_{i1})$**.** Because $a_{i0} \neq a_{i1}$, and $c_{i0} \neq c_{i1}$, then $z_{i0} \neq z_{i1}$ with overwhelming probability (note that if $a_{i0} \neq a_{i1}$ and $c_{i0} \neq c_{i1}$ but $z_{i0} = z_{i1}$, $\mathcal{O}^{dl}$ will have been queried only once and so $\mathcal{D}$ can query the oracle directly to solve, similarly to what is described in the case below).

Hence, in this case, $\mathcal{O}^{dl}$ will have been queried twice during the two executions of SIM to obtain z_{i0} and z_{i1} However, $\mathcal{D}$ can solve for both $x_i = d_i$ and $x_{i+1} = e_i$ without making any additional $\mathcal{O}^{dl}$ queries, by performing the following steps:

- **Step One.** Derive $z_{i0} - z_{i1}$ (where unknown terms are underlined), resulting in Equation 3.10, after cancelling out the d_i term:

$$z_{i0} - z_{i1} = \underline{e_i} \cdot a_{i0} + \mathsf{sk}_i\lambda_i \cdot c_{i0} - \underline{e_i} \cdot a_{i1} - \mathsf{sk}_i\lambda_i \cdot c_{i1} \tag{3.10}$$

- **Step Two.** $\mathcal{D}$ can then solve efficiently for e_i, by Equation 3.11

$$\underline{e_i} = \frac{z_{i0} - z_{i1} + \mathsf{sk}_i\lambda_i(c_{i1} - c_{i0})}{a_{i0} - a_{i1}} \tag{3.11}$$

  Because $a_{i0} \neq a_{i1}$, Equation 3.11 is solveable.

- **Step Three.** $\mathcal{D}$ can then solve efficiently for d_i, as follows:

$$\underline{d_i} = z_{i0} - e_i \cdot a_{i0} - c_{i0} \cdot \mathsf{sk}_k$$

So, in this case, exactly two $\mathcal{O}^{dl}$ queries were made for the two challenges $(X_i = D_i, X_{i+1} = E_i)$, and $\mathcal{D}$ can efficiently solve for $y_i = d_i$ and $y_{i+1} = e_i$.

Case 2 ($a_{i0} = a_{i1}, c_{i0} = c_{i1}$)**.** In this case, because $a_{i0} = a_{i1}$ and $c_{i0} = c_{i1}$, then $z_{i0} = z_{i1}$. Consequently, SIM will have made only one unique $\mathcal{O}^{dl}$ to obtain $z_{i0} = z_{i1}$, leaving one additional query available in the allowed "budget."

$\mathcal{D}$ can then query $\mathcal{O}^{dl}$ one additional time on D_i to obtain $d_i = x_i$. $\mathcal{D}$ can then solve for $x_{i+1} = e_i$ directly, by solving the linear relation in Equation 3.12 where the single unknown term e_i is underlined for emphasis:

$$\underline{e_i} = \frac{z_{i0} - d_i - \mathsf{sk}_i \lambda_i \cdot c_{i0}}{a_{i0}} \tag{3.12}$$

So, again, in this case, exactly two $\mathcal{O}^{dl}$ queries were made for the two challenges ($X_i = D_i, X_{i+1} = E_i$), and $\mathcal{D}$ has solved for $y_i = d_i$ and $y_{i+1} = e_i$.

Case 3 ($a_{i0} \neq a_{i1}, c_{i0} = c_{i1}$)**.** Because a_i is (indirectly, via R) an input to the random oracle when generating c_i, then this event occurs only when there is a collision in c_i. If this event occurs, then $\mathcal{D}$ must abort. We refer to this bad event as $\mathsf{BadEvent}_2$. This case occurs with negligible probability $1/q$ for each signing query that $\mathcal{A}$ makes, with total probability q_s/q.

Case 4 ($a_{i0} = a_{i1}, c_{i0} \neq c_{i1}$)**.** Again, for similar reasons as in the case above, when $\mathsf{H}_{\mathsf{sig}}$ forks, then subsequent queries to $\mathsf{H}_{\mathsf{non}}$ will also fork, because SIM always programs its random oracle with the next-available value h_i. We refer to this bad event as $\mathsf{BadEvent}_3$. This case occurs with negligible probability $1/q$ for each signing query that $\mathcal{A}$ makes, with total probability q_s/q.

Finishing the Proof. $\mathcal{D}$ will hence succeed with advantage

$$\begin{aligned}
\mathsf{Adv}^{\mathrm{aomdl}}_{\mathcal{D}^{\mathsf{SIM}}}(\lambda) &\geq \Pr[\mathsf{Fork}^{\mathsf{SIM}}_m(\mathsf{par}, \{X_0, \ldots, X_{2q_s}\}, n, t) \neq \perp] - \Pr[\mathsf{BadEvent}_2] - \Pr[\mathsf{BadEvent}_3] \\
&\geq \frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathsf{FROST},\mathcal{A}}(\lambda, n, t, t-1)^2}{q_r} - \frac{2q_r}{q} - \frac{3q_r^2}{q} - \Pr[\mathsf{BadEvent}_2] - \Pr[\mathsf{BadEvent}_3] \\
&\qquad\qquad \textit{// Combining Corollary 1 and Lemma 2} \\
&\geq \frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathsf{FROST},\mathcal{A}}(\lambda, n, t, t-1)^2}{q_r} - \frac{2q_r}{q} - \frac{3q_r^2}{q} - \frac{2q_s}{q}
\end{aligned}$$

First, Lemma 2 gives the relation $\Pr[\mathsf{SIM}(\mathsf{par}, n, t, t-1, \hat{X}) \neq \perp] \geq \mathsf{Adv}^{\mathrm{uf}}_{\mathsf{FROST},\mathcal{A}}(\lambda, n, t, t-1) - \frac{q_r}{q} - \frac{q_r^2}{2q}$. Then, because $\mathcal{D}$ runs the forking algorithm Fork_m on SIM,

and Fork_m runs SIM twice, then by Corollary 1, $\Pr[\mathsf{Fork}_m^{\mathsf{SIM}}(\mathsf{par}, n, t, t-1, \hat{X}) \neq \perp]$ is greater than or equal to $\frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathsf{FROST},\mathcal{A}}(\lambda,n,t,t-1)^2}{q_r} - \frac{2q_r}{q} - \frac{2q_r^2}{q} - \frac{2q_r^2}{2q}$, or, simplified, $\frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathsf{FROST},\mathcal{A}}(\lambda,n,t,t-1)^2}{q_r} - \frac{2q_r}{q} - \frac{3q_r^2}{q}$.

This bound quantifies the probability that:

1. SIM will not abort across its two executions,
2. SIM outputs the same index across its two executions, and
3. none of the random oracle inputs collide.

Then, we have to account for the probability that $\mathsf{BadEvent}_2$ and $\mathsf{BadEvent}_3$ occur, which causes the reduction $\mathcal{D}$ to abort. Recall that $\Pr[\mathsf{BadEvent}_2] = \Pr[\mathsf{BadEvent}_2] = q_s/q$, and each of these events occurs when $\mathcal{D}$ attempts to extract its ℓ-AOMDL solution (i.e., these events occur after $\mathsf{Fork}_m^{\mathsf{SIM}}$ has finished its execution). Hence, the probability of these bad events adds an additional $2q_s/q$ term to yield the bound.

This completes the proof. □

3.4 Implementation and Operational Considerations

We have implemented FROST in Rust, using Ristretto over curve25519 [LdV20] for the group operations. Our source code can be found at https://crysp.uwaterloo.ca/software/frost.

We now discuss two topics that may be of interest to implementors.

Publishing commitments. The preprocessing step for FROST requires some agreed-upon location for participants to publish their commitments to, such as a commitment server, which is trusted to provide the correct (i.e., valid and unused) commitment shares upon request. If malicious, it could perform a denial-of-service attack, or it could provide stale or malformed commitment values on behalf of honest participants. However, simply having access to the set of a participant's *public* published commitments does not grant any additional powers.

Performing two-round FROST without central roles. While the round complexity of FROST can be optimized using central roles such as the signature aggregator, some implementations may wish to remain completely decentralized. In this setting, participants can simply broadcast the outputs of Round One and Round Two to each other.

Chapter 4

A Formal Treatment of Distributed Key Generation, and New Constructions

Threshold signature schemes such as FROST generally require a distributed protocol to generate key material among all signing participants in an untrusted manner; such protocols are referred to as *Distributed Key Generation* (DKG) schemes. DKG schemes underlie many multi-party cryptographic primitives in use today in addition to threshold signature schemes [KG20, BCK+22, Lin24], such as distributed randomness beacons [dra25]. At a high level, a DKG allows a set of n participants to cooperate to generate a keypair, such that the public key represents the entire set of participants, and each participant holds a share of the corresponding secret key. DKGs ensure that the public key is agreed upon by all (honest) parties at the end of the protocol, but no single party knows the corresponding secret key. However, this secret key can be recovered given a threshold t number of cooperating parties, where $t \leq n$.

Despite widespread interest in DKGs, existing constructions fall short in their complexity or security guarantees. One example is Pedersen's DKG [Ped91] and variants of Pedersen's DKG [KG20, BCK+22], discussed in Chapter 3. While Pedersen's DKG is appealing because of its simplicity and efficiency, its security to date has only been proven in the context of specific applications [GJKR03, GJM+21, BCK+22]. More specifically, it is possible that the use of Pedersen's DKG may be secure in some settings, but not in others. It has been speculated that Pedersen's DKG can be proven secure under the UC-definition of idealized discrete-logarithm key generation with additive bias, where the bias is chosen by the adversary [Kat24]. However, no such proof yet exists. While many alternative

DKG constructions have been presented in the literature [GJKR07, Gro21, GJM+21, BS20, NSR16, Lin24, DYX+22, BL22, AJM+22], these constructions incur undesirable protocol complexity or require pairings.

Conversely, growing interest in threshold schemes points towards the need for alternative discrete-logarithm DKGs that are friendly to standardization efforts [BD22, BP23]. For example, the FROST Schnorr threshold signature scheme [KG20, BCK+22], introduced in Chapter 3, for which an IETF specification exists [CKGW24], is written in a way to be agnostic to any particular key generation scheme. Ideally, a candidate DKG to be standardized should be applicable not only to a specific threshold signature scheme, but also to other applications that require secret key material to be distributed among a set of trusted parties [DSQ+22].

Proving the Security of DKGs. However, challenges arise when proving the security of a DKG in a standalone manner; i.e., that the DKG is secure in *any* setting in place of its target (single-party) key generation scheme. Although existing simulatability-based notions for DKGs exist [GJKR07, GJM+21, Lin24, BL22, BS20, BKP13], these notions prove to be either incomplete or insufficient. For example, many simulatability-based notions assume important details such as the consistency of honest participants' state or that adversarial players do not cause the protocol to abort. However, such assumptions are *not* guaranteed in a distributed, multi-party setting. Notably, participants that force the protocol to abort during the last round can be employed as a vector for a key-bias attack [GJKR07]. Under the hood, many definitions assume the protocol to be robust; i.e., that the protocol will always complete successfully in spite of misbehaving parties. However, doing so increases the complexity for the protocol by requiring additional complaint and voting rounds, which are difficult to implement correctly in practice.

4.1 Our Contributions

A Generic DKG. In this chapter, we present a simple generic DKG construction that is a good candidate for ongoing threshold standardisation efforts. Our construction relies on three simple cryptographic building blocks: a secure hash function, an *Aggregatable Verifiable Secret Sharing* (AgVSS) scheme, and a *Non-Interactive Key Exchange* (NIKE) scheme. The ability to aggregate Feldman's VSS [Fel87] has been extensively employed in prior DKG constructions [Ped91, GJKR07, KG20, GJM+21, Lin24]; we simply formalize the notion of a VSS that is aggregatable, and present new security notions to reflect this

	GJKR [GJKR07]	Storm	PedPop [KG20, BCK+22]
Optimistic Rounds	3	3	2
Total Rounds	6	4	2
Robustness	Strong	Weak	No
Bandwidth	$2nt + 3n$	$nt + 5n$	$nt + 3n$
Computation	$2nt + 2n + t$	$nt + 2n$	$nt + 2n$
Assumption	DLP	CDH	KoE/AGM
Threshold	$n \geq 2t - 1$	$n \geq 2t - 1$	$n \geq t$

Table 4.1: Comparison between Storm and related, non-publicly-verifiable, synchronous constructions.

property. We additionally provide formal definitions and notions of security for a NIKE, as well as concrete AgVSS and NIKE constructions.

Why a Generic Construction? The goal for presenting a generic DKG as opposed to a monolithic construction is to enable flexibility for implementations and future standardization efforts. For example, while we define concrete AgVSS and NIKE constructions, future implementations may instead wish to employ a standardized NIKE [BBLW22] instead. Further, as NIST is currently in the process of standardizing threshold primitives [BP23], it may be worthwhile to leverage future VSS standards, which may possibly diverge from the concrete VSS defined in this work. Or, implementations may wish to define a construction that employs publicly verifiable randomness, a polynomial commitment scheme that is more efficient, or derives secret values deterministically, such as from a seed phrase. The utility of a generic DKG is that so long as the chosen building blocks are secure, any instantiation of our generic DKG using these building blocks will remain secure.

Concrete Instantiation: Storm. We then describe a concrete instantiation of our generic construction that we call Storm (Synchronous, disTributed, Optimized geneRation of key Material).

The security of Storm reduces to the Computational Diffie-Hellman (CDH) problem in the random oracle model. Storm is compatible with existing threshold cryptosystems such as threshold Schnorr signatures [KG20, CKGW24] and similar discrete-log based systems. Storm is simpler and therefore (slightly) more efficient than related synchronous DKGs in the discrete-logarithm setting with similar security guarantees [GJKR03]. Its efficiency improvement is due to the use of a NIKE, which allows the parties to derive a "tweaked" key at the end of the protocol, preventing adaptive attacks.

Comparing Storm to Related Schemes. We compare in Table 4.1 the practicality and security of Storm to related constructions in the literature that require only standard (i.e., non-pairing based) groups that are secure in the random oracle model, and have formal proofs of security.

Optimistic rounds considers only when no participant cheats; total rounds includes additional rounds required to recover from players that misbehave during the protocol. Robustness indicates whether the protocol always succeeds assuming some number of honest players (strong) or allows non-fatal aborts (weak). Bandwidth efficiency denotes the number of group and field elements sent and received by each participant throughout the protocol. Computational efficiency denotes the number of exponentiations required (i.e., scalar multiplications in an elliptic curve group). The value n is the total number of participants in the protocol, t is the threshold number required to recover the secret key, and the number of corrupted parties is assumed to be *at most* $t-1$. DLP denotes the Discrete Logarithm Problem, KoE the Knowledge of Exponent model; AGM is the Algebraic Group Model, and CDH the Computational Diffie-Hellman Problem.

Foramlized Notions of Security for DKGs. Our goal is to prove the security of our generic construction in such a way that (i) demonstrates the DKG can be securely used in any setting where its target key generation scheme is used, and (ii) captures implicit notions such as handling adversarial aborts and honest participant consistency. Towards this end, we introduce formal game-based security notions for DKGs that generalize and extend prior notions in the literature. Our games make explicit important details such as communication patterns between honest and adversarial players, where prior notions have omitted some details. For example, our games explicitly model the adversary as *rushing*, in that it can always query for honest player's contributions before producing its own. We then prove the security of our generic DKG with respect to these notions.

For a DKG to be secure, we require that the DKG be strongly or weakly robust, zero-knowledge, and indistinguishable. Strong versus weak robustness captures the distinction between a protocol that always succeeds assuming some threshold of honest participants (strong robustness), and one which allows for non-fatal aborts that do not impact the security of the scheme (weak robustness). Similar to prior simulation-based notions, zero-knowledge requires that the DKG is simulatable with respect to a public key. Indistinguishability requires that the output of a DKG be indistinguishable from its target (single-party) key generation scheme. By fulfilling the notion of indistinguishability, the DKG can be securely employed in any setting where the target (single-party) key generation scheme could be used. However, for cryptosystems that ensure secret and public keys

have a unique correspondence (such as discrete-log based schemes), zero-knowledge implies indistinguishability.

We instead opt for a game-based approach, so we can explicitly and independently define each notion.

Future Work: Proving Equivalence and Composability. We chose to employ a game-based approach to defining the security of DKGs, with the goal of clearly separating notions and defining how the adversary interacts with honest participants throughout the protocol execution. However, our game-based notions do not demonstrate that a DKG is secure even when composed with other protocols, such as threshold signatures. One approach to demonstrating composability is using the Universal Composability (UC) framework [Can01]. However, another approach would be to prove equivalence between our game-based notions and a simulation-based notion of security for a DKG. We discuss this second approach further in Section 4.7.

What We Do Not Do. Our definitions and constructions assume a strictly synchronous setting; i.e., that every participant in the protocol terminates a round before beginning the next. While asynchronous DKGs exist [NSR16, DYX+22, AJM+21, KMS20, KHG12, YLF+22], these constructions require the tradeoff of increased complexity. For similar reasons, we require the existence of a secure broadcast channel.

Additionally, while some prior DKGs build upon a publicly verifiable secret sharing scheme (PVSS) [Gro21, BS20], we explicitly do not, due to the tradeoff in increased implementation complexity. Furthermore, we do not consider scalability for large numbers of parties [CS04, ZXH+22, YLF+22], although such optimizations could be employed to define a more efficient instantiation of our generic construction.

We leave proofs of adaptive security for our generic construction for future work. Techniques to achieve tight proofs of adaptive security in stronger security models (such as the AGM) have recently been demonstrated in concurrent work [BL22, AJM+22]; we expect these techniques to similarly apply to our constructions.

Summary of Our Contributions. In this chapter, we present the following contributions.

- We introduce the notion of an aggregated Verifiable Secret Sharing (AgVSS) scheme, building upon its implicit use in prior literature. We then give concrete AgVSS and Non-Interactive Key Exchange (NIKE) constructions.

- We present formal game-based definitions for the security of a DKG, that makes explicit details such as communication patterns between honest and corrupted players, expectations of participant consistency and requirements of key validity. Further, our separation of the notion of robustness into a strong and weak variant allows for a simpler and more efficient DKG construction.
- We give a generic DKG construction, and prove its security using our new notions. Our generic construction requires a secure hash function, an AgVSS, and a NIKE.
- We introduce Storm, a concrete instantiation of our generic construction. The security of Storm reduces to the Computational Diffie-Hellman (CDH) problem in the random oracle model.

4.2 Related Work

Synchronous VSS-Based Discrete-Logarithm Constructions. Pedersen's DKG [Ped91] is a two-round protocol among n parties, such that each party plays the role of the dealer for Feldman Verifiable Secret Sharing (VSS) [Fel87]. Gennaro et al. [GJKR07] describe an attack against Pedersen's DKG, which we refer to as the Key-Influence attack, that considers a *rushing adversary*, who is allowed to speak last. In this attack, an adversary can influence the distribution of key material by adaptively choosing their contribution. While the authors later demonstrate unforgeability of a Schnorr threshold signature scheme with Pedersen's DKG employed for key generation [GJKR03], this proof of security does *not* extend to other applications. Gurkan et al. [GJM+21] show that Pedersen's DKG can be used securely in any setting where the cryptographic scheme is "rekeyable". In other words, if the scheme produces some cryptographic output o_1 (such as a ciphertext or signature) that is valid with respect to a secret key sk_1, it is possible to obtain a second value $o_2 \leftarrow f_1(o_1, \mathsf{sk}', \alpha)$, such that o_2 is valid with respect to a related secret key $\mathsf{sk}_2 \leftarrow f_2(\mathsf{sk}_1, \mathsf{sk}', \alpha)$, where α is a side input and f_1, f_2 are efficient update functions. While some constructions such as BLS signatures [BLS04] do support rekeyability, others such as Schnorr signatures do not.

Gennaro et al. [GJKR07] present a construction secure against the Key-Influence attack that is (similar to our generic construction) three rounds in the optimistic setting, but it has higher round complexity when parties misbehave. While their description of their construction implies that a round to broadcast failure messages is optional, their construction is in fact insecure without this round, as a malicious party may force a subset of parties to fail, and so selectively choose which participant contributions are used towards

the creation of the output public key and secret key shares (thereby biasing the output key material). While requiring higher round complexity, their construction offers strong robustness as players negotiate in the second and third round which qualified set of players to continue the protocol with. Our generic construction simply aborts the protocol, requiring the implementation to investigate the set of cheating players outside the protocol, a reasonable assumption in practice.

Lindell [Lin24] introduced a DKG that builds upon Schnorr proofs of knowledge and online-extractable zero-knowledge proofs of knowledge. However, the protocol incurs computational costs due to the requirement of online extractors [Fis05]. Furthermore, the security of the scheme is only generally discussed without reference to a concrete notion of security.

Many efficient pairing-based DKGs and secret-sharing primitives are described in the literature [GJM+21, Gro21, AJM+22, ZXH+22]. Further, concrete DKGs have been presented in the ECDSA and RSA settings [LN18, DM10, FMY98, CGG+20]. This work presents a generic construction that can be instantiated without pairings.

Synchronous PVSS-Based Discrete-Logarithm Constructions. Our generic constructions assume VSS as a building block, and consequently incur an extra round of communication. It is possible to instead use a publicly verifiable secret sharing scheme (PVSS), as is demonstrated by alternative DKGs in the literature. However, PVSS schemes require that secret key material be encrypted to receivers' public keys, thereby requiring a complicated decryption step. For this reason, we define our construction to explicitly incur an additional network round. However, our construction could instead be adapted to use a PVSS and thereby use only two network rounds.

Definitions of DKG Security. Gurkan et al. [GJM+21] present a definition of DKG security, in which the DKG is secure if its use within some larger protocol does not weaken the security of the overall scheme from the centralized (single-party) setting. We build upon this notion by defining the security of a DKG with respect to a target key generation scheme. Bacho and Loss [BL22] introduce the security notions of consistency and oracle-aided algebraic simulatability for a DKG. Their notion of consistency is encompassed by our notions of strong and weak robustness. Because their notion of oracle-aided simulatability assumes a discrete logarithm oracle, it is not applicable to schemes without such algebraic structure. Boneh and Shoup [BS20] present a generic definition for the security of a distributed key generation protocol using a simulation-based approach, with respect to a target trusted secret sharing scheme. However, their definition is given in a UC-based approach, whereas we present security experiments using a game-based approach to explicitly define subtle

details such as participation communication patterns and aborts. However, our notions do not guarantee security under composition with other schemes.

4.3 Aggregatable Verifiable Secret Sharing (AgVSS)

We now augment the notion of a verifiable secret sharing scheme as presented in Definition 2.5.3 to define an *aggregatable* verifiable secret sharing scheme (AgVSS) A. The ability to aggregate Feldman VSS has been demonstrated before [GJM+21], and is employed implicitly in nearly every DKG that has been defined in the literature [Ped91, GJKR03, Gro21, Lin24] but we formalize the notion of an AgVSS more generally.

An AgVSS builds upon a base VSS scheme, and is defined with respect to the same map $\mathsf{SecretToPublic} : \hat{S} \to \hat{P}$ as the base VSS that maps elements in the secret space $\hat{S}$ to the public space $\hat{P}$, and an auxiliary distribution AX that has guessing entropy linear in the security parameter λ.

Definition 4.3.1. *An* **Aggregatable Verifiable Secret Sharing (AgVSS) scheme** A *is the tuple* $(\mathsf{Setup}, \mathsf{Share}, \mathsf{Recover}, \mathsf{Verify}, \mathsf{GetPub}, \mathsf{AggPriv}, \mathsf{AggPub}, \mathsf{Tweak})$*, where* $\mathsf{Share}, \mathsf{Recover}, \mathsf{Verify}, \mathsf{GetPub}$ *are identical to the base VSS scheme as in Definition 2.5.3, and where*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{par}$*: A deterministic algorithm that accepts as input a security parameter* λ*, and outputs public parameters* par *(which are given implicitly as input to all other algorithms).*
- $\mathsf{Tweak}(O, \mathsf{aux}) \to v$*: A deterministic algorithm that accepts as input a public aggregation set* O *consisting of* ℓ *VSS commitments* $\{D_1, \ldots, D_\ell\}$*, and an auxiliary string* aux *drawn from* AX*. Output a tweak* $v \in \hat{S}$*.*
- $\mathsf{AggPriv}(P, v) \to \hat{w}_i$*: A deterministic algorithm that accepts as input a private aggregation set* P *consisting of* ℓ *shares* $\{\phi_j\}_{j \in [\ell]}$*, and a tweak* v*. Output the aggregated share* $\hat{w}_i$ *for party* i*.*
- $\mathsf{AggPub}(O, v) \to C$*: A deterministic algorithm that accepts as input a public aggregation set* O *consisting of* ℓ *VSS commitments* $\{D_1, \ldots, D_\ell\}$*, and a tweak* v*. Output the aggregated commitment* C*.*

Similarly to the VSS setting, we require that the AgVSS be zero-indexed.

$\mathsf{Exp}^{\mathrm{corr}}_{\mathsf{A}}(\lambda, n, t, C, \ell)$

return 0 **if** $t \geq n \vee C \not\subseteq [n] \vee |C| < t$
$\mathsf{par} \leftarrow \mathsf{Setup}(1^{\lambda})$
$\mathsf{aux} \xleftarrow{\$} \mathsf{A.AX}$
for $j \in [\ell]$ **do**
  $s_j \xleftarrow{\$} \hat{S}$; $(\{(i, \phi_{ji})\}_{i\in[n]}, D_j) \leftarrow \mathsf{A.Share}(s_j, n, t)$
$v \leftarrow \mathsf{A.Tweak}(\{D_j\}_{j\in[\ell]}, \mathsf{aux})$
for $i \in [n]$ **do**
  $\hat{w}_i \leftarrow \mathsf{A.AggPriv}(\{\phi_{ji}\}_{j\in[\ell]}, \{D_j\}_{j\in[\ell]}, v)$
$C \leftarrow \mathsf{A.AggPub}(\{D_j\}_{j\in[\ell]}, v)$
for $i \in [n]$ **do**
  return 0 **if** $\mathsf{A.Verify}(i, \hat{w}_i, C) \neq 1$
$\hat{s} \leftarrow \mathsf{A.Recover}(t, \{(k, \hat{w}_k)\}_{k\in C})$
return 0 **if** $\mathsf{A.Verify}(0, \hat{s}, C) \neq 1$
// *This check assumes the AgVSS is zero-indexed*
return 1

Figure 4.1: Notion of correctness for an AgVSS.

Aggregation Correctness. An AgVSS A must fulfill the notion of correctness for a VSS scheme. In addition, an AgVSS must fulfill the notion of correctness when the scheme is aggregatable; i.e., that for all n, t, ℓ, C, where $n, t, \ell \in \mathbb{N}$, $C \subseteq [n]$, and $|C| \geq t$,

$$\Pr\left[\mathsf{Exp}^{\mathrm{corr}}_{\mathsf{A}}(\lambda, n, t, C, \ell) = 1\right] = 1$$

where $\mathsf{Exp}^{\mathrm{corr}}_{\mathsf{A}}(\lambda, n, t, C, \ell)$ is presented in Figure 4.1.

Intuitively, the experiment in Figure 4.1 ensures that an AgVSS is correct if the corresponding secret output from $\mathsf{A.Recover}$ is valid, meaning that $\mathsf{A.Verify}$ outputs 1 when checked with respect to the AgVSS commitment C.

An AgVSS must satisfy aggregated secrecy and uniqueness, as discussed next.

Aggregated Secrecy. Unlike in the plain VSS setting, an AgVSS allows for aggregating shares and commitments that could both be honestly as well as maliciously generated.

Hence, we require that the AgVSS fulfills the notion of *aggregated secrecy*, which we formalize in Figure 4.2.

Remark 5 (Aggregated secrecy for an AgVSS is a generalization of VSS secrecy)**.** The secrecy game for a VSS scheme as shown in Figure 2.6 only allows the adversary to receive a single set of shares and their corresponding commitment from the environment. Hence, the aggregated secrecy game in Figure 4.2 is a strict generalization of the VSS secrecy game, as the adversary in the secure aggregation game can not only query for many shares and commitments, but submit its own. As a consequence, any AgVSS that fulfills aggregated secrecy is also secret.

When executing the experiment shown in Figure 4.2, the environment simulates one secret sharing operation with respect to a challenge Δ when the random bit $b = 1$. The environment then allows the adversary to query for corrupted parties' shares by $\mathcal{O}^{\mathsf{Share}}$, and submit its own shares for honest parties by $\mathcal{O}^{\mathsf{RecvShare}}$. The adversary is then provided with the auxiliary string aux. The adversary automatically wins if the aggregated commitment C does not fulfill the check $\mathsf{A.GetPub}(0, C) \neq \Delta$. Otherwise, $\mathcal{A}$ outputs a guess b' to guess if it is in the real or simulated environment.

The advantage of an adversary $\mathcal{A}$ against A in the aggregated secrecy experiment as defined in Figure 4.2 with respect to algorithms $\mathsf{SimShare}, \mathsf{SimTweak}$ is

$$\mathsf{Adv}^{\mathrm{asec}}_{\mathsf{A},\mathcal{A},\mathsf{SimShare},\mathsf{SimTweak}}(\lambda) = \max_{s \in \hat{S}} \left| \Pr[\mathsf{Exp}^{\mathrm{asec}}_{\mathsf{A},\mathcal{A},\mathsf{SimShare},\mathsf{SimTweak}}(\lambda, s) = 1] - 1/2 \right|$$

Definition 4.3.2. *An AgVSS scheme* A *achieves* **aggregated secrecy** *if there exist efficient algorithms* $\mathsf{SimShare}, \mathsf{SimTweak}$ *such that for all probabilistic polynomial-time adversaries* $\mathcal{A}$*, the function* $\mathsf{Adv}^{\mathrm{asec}}_{\mathsf{A},\mathcal{A},\mathsf{SimShare},\mathsf{SimTweak}}(\lambda)$ *is a negligible function of* λ*.*

Uniqueness. The uniqueness property for an AgVSS is the same as that of the base VSS, as presented in Figure 2.7. Notably, the adversary in this experiment can perform aggregation directly, and so the experiment does not change.

4.3.1 A Discrete-Log Based AgVSS

We now present a concrete AgVSS scheme that is secure in the discrete logarithm setting, builds upon Feldman's VSS, and assumes an honest majority of participants. The algorithms $\mathsf{Share}, \mathsf{Recover}, \mathsf{Verify}, \mathsf{GetPub}$ are identical to Feldman's VSS, as described in Section 2.5.2.1.

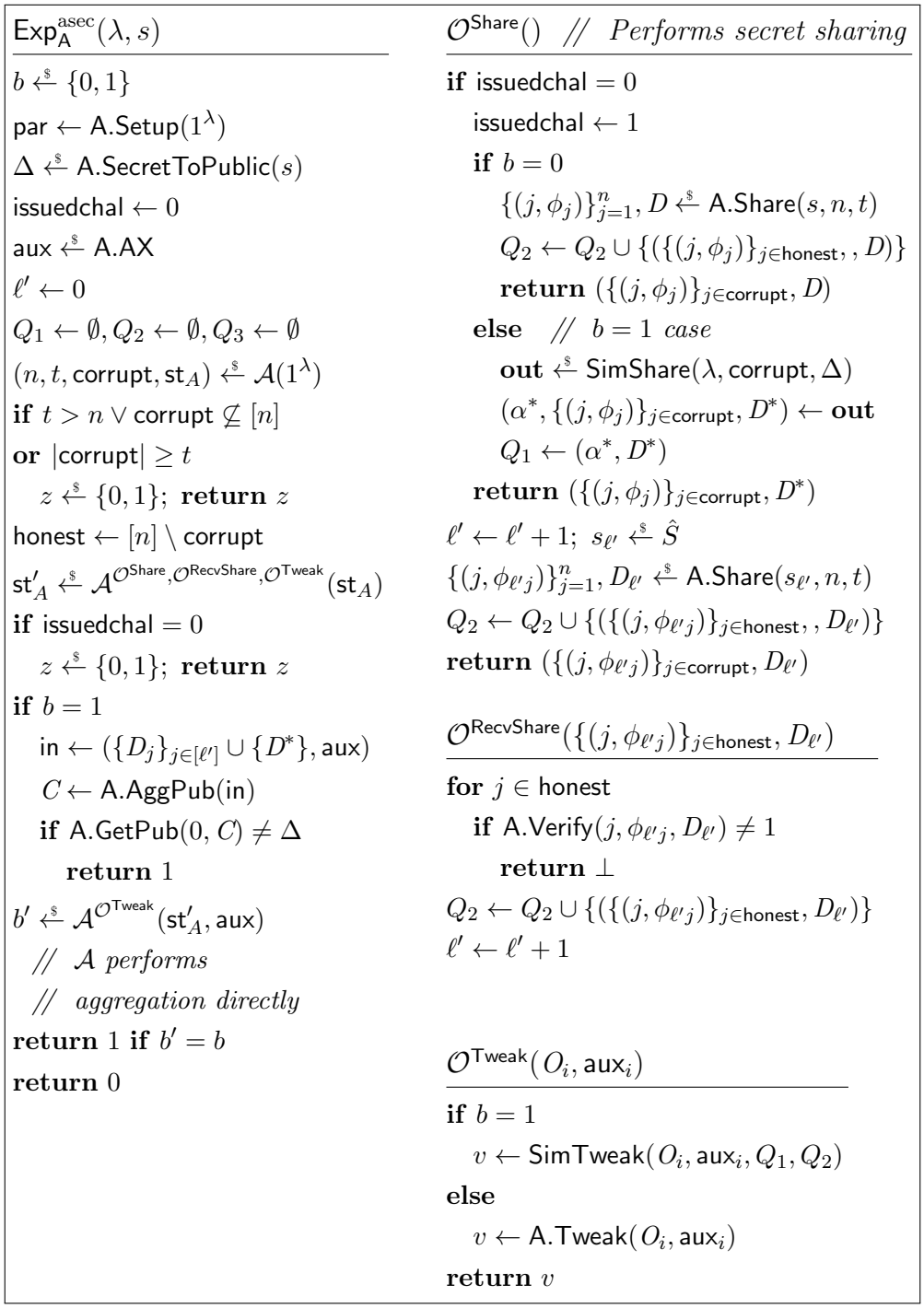


Figure 4.2: Aggregated secrecy experiment for a zero-indexed Aggregatable Verifiable Secret Sharing scheme (AgVSS).

Let the distribution AX be the uniform distribution on $\mathbb{Z}_q$. We now define the additional algorithms for this concrete AgVSS.

- $\mathsf{A.Setup}(1^\lambda) \to \mathsf{par}$: On input a security parameter 1^λ, run $(\mathbb{G}, q, g) \leftarrow \mathsf{GroupGen}(1^\lambda)$. Let $\mathsf{H} : \mathbb{G}^* \times \mathbb{Z}_q \to \mathbb{Z}_q$ be a cryptographically secure hash function, whose inputs are a tuple consisting of a vector of group elements in $\mathbb{G}$, and an element in $\mathbb{Z}_q$. Output public parameters $\mathsf{par} \leftarrow ((\mathbb{G}, q, g), \mathsf{H})$ (which are given implicitly as input to all other algorithms).

- $\mathsf{A.Tweak}(O, \mathsf{aux})$: Let $O = \{D_j\}_{j\in[\ell]}$ be the public aggregation set of the corresponding ℓ Feldman VSS commitments to P, such that $|O| = t$, and O is sorted in a canonical order. Further, let aux be an element in $\mathbb{Z}_q$. Generate $v \leftarrow \mathsf{H}(O, \mathsf{aux})$. Output v.

- $\mathsf{A.AggPriv}(P, v)$: Let $P = \{\phi_{ji}\}_{j\in[\ell]}$ be a private aggregation set of ℓ Shamir secret shares, and $v \in \mathbb{Z}_q$ be an auxiliary string.

  1. Derive the aggregate share via Equation 4.1.

$$\hat{w}_i \leftarrow v + \sum_{j\in[\ell]} \phi_{ji} \tag{4.1}$$

  2. Output $\hat{w}_i$

- $\mathsf{A.AggPub}(O, v)$: The input set O is as defined above, and v is the output of $\mathsf{A.Tweak}$. Perform the following:

  1. Derive the commitment to the aggregated constant coefficient by Equation 4.2:

$$\hat{A}_0 = g^v \cdot \prod_{j\in[\ell]} D_j[0] \tag{4.2}$$

  2. Derive the commitment to the $k \in \{1, \ldots, t-1\}$ aggregated coefficients by Equation 4.3:

$$\hat{A}_k = \prod_{j\in[\ell]} D_j[k] \tag{4.3}$$

     Let $C = \langle \hat{A}_0, \ldots, \hat{A}_{t-1} \rangle$

  3. Output C

We demonstrate the correctness and security properties of the concrete scheme next.

4.3.1.1 Proofs for Discrete-Logarithm Based AgVSS

In Section 4.3.1, we define a concrete AgVSS which builds upon Feldman's VSS and is secure in the honest majority setting. We demonstrate the correctness, aggregated secrecy, and uniqueness of that scheme now.

Correctness. The honest majority AgVSS scheme of Section 4.3.1 fulfills the notion of correctness for the base VSS scheme, as the algorithms $\mathsf{Share}, \mathsf{Recover}, \mathsf{Verify}, \mathsf{GetPub}$ are identical to Feldman's VSS, as is the map $\mathsf{SecretToPublic}$. We now demonstrate that the concrete AgVSS scheme fulfills the additional notion of correctness for an AgVSS scheme.

Theorem 4. *The honest majority AgVSS scheme of Section 4.3.1 is correct in the sense of Figure 4.1.*

Proof. We analyze each case where a return value of 0 could occur in Figure 4.1, and demonstrate that that event will not occur.

Case 1. We first show that for each $i \in [n]$, $\mathsf{A.Verify}(i, \hat{w}_i, C) = 1$. Recall that $\hat{w}_i$ is simply

$$\hat{w}_i = v + \sum_{j \in [\ell]} \phi_{ji}$$

by Equation 4.1. Because each ϕ_{ji} was generated by the $\mathsf{A.Share}$ algorithm, then $\phi_{ji} = f_j(i)$ for some polynomials $f_1, \ldots, f_\ell$ each of degree $t-1$. Let $f' = v + \sum_{i \in [\ell]} f_i$. Then $(i, \hat{w}_i)$ is a valid point on f', because

$$\hat{w}_i = v + \sum_{j \in [\ell]} \phi_{ji} = v + \sum_{j \in [\ell]} f_j(i) = f'(i)$$

Because C is a commitment to the same f' as shown in Equations 4.2 and 4.3, then $\mathsf{A.Verify}(i, \hat{w}_i, C)$ will output 1. Note also that $\mathsf{A.Recover}$ will output $\hat{s} = f'(0) = v + \sum_{j \in [\ell]} s_j$.

Case 2. We next prove that $\mathsf{A.Verify}(0, \hat{s}, C) = 1$. As above, after performing $\mathsf{A.AggPub}$, the resulting aggregated commitment C can be represented as a commitment to the polynomial f'. $\mathsf{A.Verify}(0, \hat{s}, C)$ checks that $g^{\hat{s}} = \hat{A}_0$ (from Eq. 2.4 with $i = 0$). From Eq. 4.2, $\hat{A}_0 = g^v \cdot \prod_{j \in [\ell]} D_j[0] = g^v \cdot \prod_{j \in [\ell]} g^{f_j(0)} = g^v \cdot \prod_{j \in [\ell]} g^{s_j}$. Since $\hat{s} = v + \sum_{j \in [\ell]} s_j$ as above, $\mathsf{A.Verify}(0, \hat{s}, C)$ will output 1.

This concludes the proof. □

Aggregated Secrecy. We now show that the concrete scheme of Section 4.3.1 fulfills the notion of aggregated secrecy from Figure 4.2. To do so, we first show how the environment simulates SimShare and H. Without loss of generality, we assume $|\mathsf{corrupt}| = t - 1$.

$\mathsf{SimShare}(\lambda, \mathsf{corrupt}, \Delta)$

$\alpha \xleftarrow{\$} \mathbb{Z}_q;\ A_0 = \Delta \cdot g^{\alpha}$
for $j \in \mathsf{corrupt}$ **do**
  $\phi_j \xleftarrow{\$} \mathbb{Z}_q$
for $k \in \{1, \ldots, t-1\}$ **do**
  $A_k = A_0^{L'_{0,k}} \cdot \prod_{j \in \mathsf{corrupt}} g^{L'_{j,k} \cdot \phi_j}$
$D = \langle A_0, \ldots, A_{(t-1)} \rangle$
return $(\alpha, \{(j, \phi_j)\}_{j \in \mathsf{corrupt}}, D)$

$\mathsf{H}(O_i, \mathsf{aux}_i)$

// *Q_1, Q_2, Q_3 are defined*
// *by the experiment*
if $Q_3[O_i, \mathsf{aux}_i] \neq \perp$
  return $Q_3[O_i, \mathsf{aux}_i]$
$D^* \leftarrow Q_1$
$\{D_j\}_{j \in [\ell']} \leftarrow Q_2$
$O = \{D_j\}_{j \in [\ell']} \cup D^*$
if $\mathsf{aux}_i \neq \mathsf{aux}$ **or** $O_i \neq O$
  // *Respond honestly if this is*
  // *not the challenge query*
  $Q_3[O_i, \mathsf{aux}_i] \xleftarrow{\$} \mathbb{Z}_q$
  return $Q_3[O_i, \mathsf{aux}_i]$
$\alpha \leftarrow Q_1$
for $\{(j, \phi_{kj})\}_{j \in \mathsf{honest}, k \in [\ell']} \in Q_2$
  $\mathsf{in} \leftarrow (t, \{(j, \phi_{kj})\}_{j \in \mathsf{honest}})$
  $s_k \leftarrow \mathsf{A}[\mathsf{H}].\mathsf{Recover}(\mathsf{in})$
$v \leftarrow (-1) \cdot (\alpha + \sum_{i \in [n]} s_i)$
$Q_3[O_i, \mathsf{aux}_i] \leftarrow v$
return v

Figure 4.3: Algorithms to simulate secret sharing and tweak generation in an AgVSS to an adversary.

Lemma 4. *When playing the secure aggregation game as in Figure 4.2 against the concrete scheme, the output of* $\mathcal{O}^{\mathsf{Share}}$ *when* $b = 0$ *(i.e., when the environment honestly performs* A.Share*) is indistinguishable to an adversary from the output when* $b = 1$ *(i.e., when the environment performs* SimShare *as shown in Figure 4.3), assuming the adversary controls no more than* $|\mathsf{corrupt}| = t - 1$ *participants.*

Proof. $\mathsf{SimShare}$ accepts a challenge $\Delta \in \mathbb{G}$, and outputs a randomly sampled secret α, $|\mathsf{corrupt}|$ shares, and an AgVSS commitment. To begin, it picks $\alpha \in \mathbb{Z}_q$ at random, and then generates a blinded commitment A_0 to the (unknown) discrete logarithm of Δ using α as the blinding factor, by Equation 4.4.

$$A_0 = \Delta \cdot g^{\alpha} \tag{4.4}$$

$\mathsf{SimShare}$ then picks $t-1$ shares for the corrupted parties at random, resulting in the list of shares $\langle \phi_j \rangle_{j \in \mathsf{corrupt}} \xleftarrow{\$} \mathbb{Z}_q^{(t-1)}$ and then generates commitments to coefficients $A_1, \ldots, A_{(t-1)}$ with respect to these shares and A_0, by Equation 4.5.

$$A_k = A_0^{L'_{0,k}} \cdot \prod_{j \in \mathsf{corrupt}} g^{L'_{j,k} \cdot \phi_j} \tag{4.5}$$

Here, $L'_{i,k}$ denotes the k^{th} coefficient of the Lagrange polynomial $L'_i(x)$, which is computed with respect to the set $\{0\} \cup \mathsf{corrupt}$. The commitment D to the $|\mathsf{corrupt}|$ shares is then $D = \langle A_0, \ldots, A_{(t-1)} \rangle$. The simulation is perfect because D simply is committing to a random polynomial f "in the exponent", whose constant term is the discrete logarithm of A_0, (where A_0 is a blinded commitment to the discrete logarithm of Δ) and with $t-1$ random points that are the shares distributed to the adversary. Each additional coefficient $a_k, k \in [t-1]$ of f is committed to "in the exponent" via A_k, which can be determined via Equation 4.5. Hence, so long as the adversary controls fewer than $t-1$ players, $\mathsf{A.Verify}(i, \phi_i, D)$ for all $i \in \mathsf{corrupt}$ will output 1. □

Lemma 5. *The adversary has negligible advantage in distinguishing the environment's simulation of* Tweak *in the concrete scheme of Section 4.3.1 in the (programmable) random oracle model, assuming an honest majority.*

Proof. Recall that Tweak is instantiated by H in the concrete scheme. For any query (O', aux') where $\mathsf{aux}_i \neq \mathsf{aux}$, the environment simply programs H to return a random value. However, when it receives a query for (O, aux), it programs H to return v as in Equation 4.6. Recall that aux is provided to the adversary only after it has finished querying $\mathcal{O}^{\mathsf{Share}}$ and $\mathcal{O}^{\mathsf{RecvShare}}$, and so similarly O is fixed at the time that the adversary can query with aux as input (i.e., the environment will never have to guess which query to program).

$$v \leftarrow -\alpha - \sum_{j \in [\ell']} s_j \tag{4.6}$$

There are three requirements that must be satisfied for the environment to program H correctly.

1. The programmed output v must be indistinguishable from a random value.
2. The environment must be able to derive $\{s_j\}_{j\in[\ell']}$ without the involvement or detection of $\mathcal{A}$.
3. The environment can correctly program H strictly before $\mathcal{A}$ can query on inputs (O, aux).

We first demonstrate that the output v from Equation 4.6 is indistinguishable from any other output from H. Recall that α is chosen at random when $\mathsf{SimShare}$ is performed, and published only within the commitment A_0 as shown in Equation 4.4, serving as a blinding factor for Δ. Because Δ is chosen at uniformly at random from the public domain, and because α is chosen at random, the v value in Equation 4.6 is indistinguishable from a random element to $\mathcal{A}$. Moreover, even if the adversary submitted inputs to $\mathcal{O}^{\mathsf{RecvShare}}$ in the attempt to cancel out the outputs from $\mathcal{O}^{\mathsf{Share}}$, the adversary would be unable to do so, because the adversary sees only the blinded commitment A_0 as shown in Equation 4.4, and because Δ is chosen uniformly at random. Therefore, programming H at exactly one point with v is indistinguishable to any other point that is honestly programmed with a random value chosen from $\mathbb{Z}_q$.

Second, the environment can derive $\{s_j\}_{j\in[\ell']}$ without the involvement or detection of $\mathcal{A}$, because we operate in the honest majority setting, and assume that the environment simulates at least t honest players. The environment can therefore reconstruct the adversary's s_j values (from Q_2), and it knows its own (from both Q_1 and Q_2).

Finally, the adversary does not learn aux until *after* it has completed its queries to $\mathcal{O}^{\mathsf{Share}}$ and $\mathcal{O}^{\mathsf{RecvShare}}$. Before returning aux to $\mathcal{A}$, the environment programs H with v on inputs (O, aux). Because $\mathsf{AX} = \mathbb{Z}_q$ and aux is some uniform element from AX, then $\mathcal{A}$ had negligible probability in learning the output H with input aux until the environment has programmed it. The environment is able to correctly program on these inputs using v because it is guaranteed that $\mathcal{A}$ cannot further influence O. In other words, there is no danger that the environment must make a decision on which input to program using v.

This completes the proof. □

Theorem 5. *The concrete scheme of Section 4.3.1 fulfills aggregated secrecy against an adversary $\mathcal{A}$ as in Figure 4.2, assuming an honest majority.*

Proof. Recall that that in the aggregated secrecy game, $\mathcal{A}$ has three opportunities to distinguish the $b = 0$ game from the $b = 1$ game:

1. by how the environment responds to $\mathcal{O}^{\mathsf{Share}}$,
2. by how the environment responds to $\mathcal{O}^{\mathsf{Tweak}}$, and
3. whether or not $\mathsf{A.GetPub}(0, C)$ evaluates to the challenge Δ when $b = 1$.

For (1), by Lemma 4, $\mathcal{O}^{\mathsf{Share}}$ is indistinguishable to the adversary when $b = 0$ (i.e., shares are honestly generated) and when $b = 1$ (i.e., $\mathsf{SimShare}$ is performed), For (2), by Lemma 5, the environment can simulate H in a way that is indistinguishable to the adversary in the $b = 0$ case (the output is chosen at random) and the $b = 1$ case (the output is programmed). We complete the proof by showing that the adversary has zero advantage for (3).

Recall that the concrete AgVSS is zero-indexed, per Definition 2.5.4. By Lemma 4, the environment can embed Δ in the first query of the adversary to $\mathcal{O}^{\mathsf{Share}}$ in a way that is indistinguishable to $\mathcal{A}$. $\mathcal{A}$ must query $\mathcal{O}^{\mathsf{Share}}$ at least once (i.e., if $\mathsf{issuedchal} = 0$ at the end of the experiment, then the experiment returns a random bit and $\mathcal{A}$ has no advantage). Hence, it is guaranteed that the public recovery set $\{D_j\}_{j\in[\ell]}$ that is input to $\mathsf{A.AggPub}$ will include a contribution with respect to the discrete logarithm of Δ.

Equation 4.7 shows that the commitment to the constant term of C (where $C = \{\hat{A}_0, \ldots, \hat{A}_{t-1}\}$) evaluates to Δ. Because the concrete AgVSS is zero-indexed, $\mathsf{A.GetPub}(0, C)$ also evaluates to Δ.

$$\begin{aligned}\hat{A}_0 &= g^v \cdot \prod_{j\in[\ell]} D_j[0] \\ &= g^v \cdot \Delta \cdot g^\alpha \cdot g^{\sum_{i\in[\ell']} s_i} \\ &= g^{-\alpha-\sum_{i\in[\ell']} s_i} \cdot \Delta \cdot g^\alpha \cdot g^{\sum_{i\in[\ell']} s_i} \\ &= \Delta\end{aligned} \tag{4.7}$$

Hence, the adversary gains no advantage in its guess of b', regardless of whether it is operating in the real ($b = 0$) or simulated ($b = 1$) setting. This concludes the proof. □

Uniqueness. The uniqueness of the concrete AgVSS also follows from that of Feldman's VSS, as we now show. As an observation, this proof might apply to AgVSS schemes in general, where the scheme builds on a secure VSS, and where the aggregation algorithm accepts only "public" values.

Theorem 6. *The concrete AgVSS scheme is unique.*

Proof. We now show that the additional aggregation algorithm in our AgVSS construction gives the adversary no additional power in the uniqueness game than in the plain Feldman's VSS setting.

Recall the VSS uniqueness experiment in Figure 2.7. The adversary is required to output a valid tuple $(\mathsf{M}_1^*, \mathsf{M}_2^*, D^*)$, for which $\mathsf{S.Recover}(\mathsf{M}_1^*) \neq \mathsf{S.Recover}(\mathsf{M}_2^*)$. As above, we know that the adversary cannot win the uniqueness game in the Feldman's VSS setting, no matter what strategy it employs, including using the AgVSS aggregation algorithm, since the aggregation algorithm uses no information hidden from the adversary. Therefore the adversary cannot win the uniqueness game in the AgVSS setting either. □

4.4 Distributed Key Generation (DKG)

Informally, a *Distributed Key Generation* (DKG) scheme allows a set of n players to jointly generate a public key pk and a secret key sk, such that all parties learn the public key pk but no single party learns sk. Instead, each party i has a share sk_i of the secret key, such that a threshold t number of shares are required to recover sk, where $t \leq n$. A DKG generates secret and public keys that are compatible with some *target* key generation scheme which operates in the centralized (single-party) setting.

We now formalize the definition of a DKG. We begin by defining a target single-party key generation scheme TK that the DKG implements in a distributed manner.

Definition 4.4.1. *A* **target key generation scheme** TK *is the tuple* $\mathsf{TK} = (\mathsf{KeyGen}, \mathsf{Verify})$*, where:*

- $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$*: A probabilistic algorithm that accepts as input a security parameter* λ *and outputs a secret and public keypair* $(\mathsf{sk}, \mathsf{pk})$*, where* $\mathsf{sk} \in \hat{S}$ *and* $\mathsf{pk} \in \hat{P}$*.*
- $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}) \to \{0, 1\}$*: A deterministic algorithm that accepts as input a secret and public keypair* $(\mathsf{sk}, \mathsf{pk})$*, and outputs* 0 *if the keypair is invalid; otherwise, outputs* 1*.*

- *For correctness, we require that for all* $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{TK.KeyGen}(\lambda)$*, we have*

$$\mathsf{TK.Verify}(\mathsf{sk}, \mathsf{pk}) = 1$$

Definition 4.4.2. *A* **distributed key generation scheme***, or* **DKG***, is the tuple of the protocols* $\mathsf{D} = (\mathsf{KeyGen}, \mathsf{Recover})$ *parameterized by the number of rounds* κ*. A DKG is defined with respect to a target key generation scheme* TK*, where:*

- $\mathsf{KeyGen}(1^\lambda, n, t) \to \langle(\mathsf{pk}, \mathsf{qual}, \mathsf{aux}), (\mathsf{status}_i, \mathsf{sk}_i)_{i\in[n]}\rangle$*: An interactive protocol between* n *participants. Accepts as input the security parameter* λ*, and positive integers* $n \geq t \geq 1$*, where* n *is the number of participants and* t *is the threshold.*

  The protocol outputs a public value $(\mathsf{pk}, \mathsf{qual}, \mathsf{aux})$ *and private values* $(\mathsf{status}_i, \mathsf{sk}_i), i \in [n]$ *where:*

  - pk *is the public key representing the group; it will be* $\perp$ *if the protocol terminated unsuccessfully.*
  - $\mathsf{qual} \subseteq [n]$ *is the set of qualified players remaining at the end of the protocol; it will be* $\perp$ *if the protocol terminated unsuccessfully.*
  - aux *is any auxiliary information that may be required for a particular application, such as participant public keys,*
  - status_i *indicates if participant* i *completed the protocol successfully, and can be either* abort*,* $\perp$*, or* accept*. Here,* abort *indicates the protocol terminated early,* $\perp$ *indicates a critical fail state has been reached, and* accept *indicates that the protocol completed successfully.*
  - sk_i *is participant* i*'s secret key share; it will be* $\perp$ *if the protocol terminated unsuccessfully.*

  Internally to the KeyGen *protocol, each participant performs the following:*

  - $\mathsf{PerformRound}_0(1^\lambda, n, t, \mathsf{id}_i) \to (\mathsf{st}_i, \mathsf{outp}, \mathsf{bmsg}_i)$*: The algorithm run by the participant to initiate the protocol. Accepts the security parameter* λ*, the number of participants* n*, the threshold* t*, and the participant identifier* id_i*. Outputs the state* st_i *for that participant, the vector* outp *of any outgoing peer-to-peer messages to other parties, and a broadcast message* bmsg_i *to all other parties. See Section 2.8 for definitions of these network channels.*

- $\mathsf{PerformRound}_k(\mathsf{st}_i, \mathsf{inp}, \mathsf{inb}) \to (\mathsf{st}_i, \mathsf{outp}, \mathsf{bmsg}_i)$*: The algorithm run for each intermediate round* $k \in \{1, \ldots, \kappa - 1\}$ *in the protocol. Accepts as input the participant's internal state* st_i*, the vector* inp *of any peer-to-peer messages sent by other participants in the previous round, and the set* inb *of any broadcast messages sent by other participants in the previous round. Outputs the tuple* $(\mathsf{st}_i, \mathsf{outp}, \mathsf{bmsg}_i)$*.*
- $\mathsf{Finalize}(\mathsf{st}_i, \mathsf{inp}, \mathsf{inb}) \to \langle(\mathsf{pk}, \mathsf{qual}, \mathsf{aux}), (\mathsf{status}_i, \mathsf{sk}_i)_{i\in[n]}\rangle$*: Accepts participant state* st_i*, peer-to-peer messages* outp*, and broadcast messages* inb*. Outputs a public output* $(\mathsf{pk}, \mathsf{qual}, \mathsf{aux})$ *and a private output* $(\mathsf{status}_i, \mathsf{sk}_i)$*.*

- $\mathsf{Recover}(\mathsf{pk}, t, \{(i, \mathsf{sk}_i)\}_{i\in C}) \to \mathsf{sk}/\perp$*: Accepts as input the public key* pk*, the threshold* t*, and a set of secret key shares* $\{(i, \mathsf{sk}_i)\}_{i\in C}$*, where* $|C| \geq t$*. Derive* sk *from* $\{\mathsf{sk}_i\}_{i\in C}$*. If* $\mathsf{TK.Verify}(\mathsf{sk}, \mathsf{pk}) \neq 1$*, output* $\perp$*. Otherwise, output* sk*.*

- *For correctness, we require that, when all participants follow the protocol:*

$$\begin{aligned} &\textit{if } \mathsf{D.KeyGen}(1^\lambda, n, t) \to \big((\mathsf{pk}, \mathsf{qual}), \langle(\mathsf{status}_i, \mathsf{sk}_i)\rangle_{i\in[n]}\big) \\ &\textit{then } \mathsf{D.Recover}(\mathsf{pk}, t, \{(i, \mathsf{sk}_i)\}_{i\in C}) \to \mathsf{sk} \textit{ for all } C \subseteq [n] \cap \mathsf{qual}, |C| \geq t, \\ &\textit{and } \mathsf{TK.Verify}(\mathsf{sk}, \mathsf{pk}) \to 1 \end{aligned}$$

A DKG must be correct, and fulfill the security notions of zero-knowledge and indistinguishability. Additionally, a DKG may fulfill the notion of either strong or weak robustness. We explain these concepts next.

4.4.1 Robustness

We now review the notion of robustness, building upon the formalism presented by Gurkan et al. [GJM+21]. In the literature, robustness today refers to the property that so long as the number of corrupted and honest participants meet some pre-defined threshold, then the protocol will complete successfully. The definition by Gurkan et al. deems a DKG to be robust if honest parties' shares combine to output a secret key that is valid with respect to a public key. However, the notion by Gurkan et al. does not consider the consistency of players' ending states.

We present several clarifications to the notion of robustness using a game-based definition in Figure 4.4. First, our definition is generalizable beyond discrete-logarithm based constructions. Second, our definition encodes two different forms of robustness. *Strong*

robustness defines the existing notion of robustness in the literature. We additionally introduce the notion of *weak robustness*, which allows the protocol to non-fatally abort, so long as all honest participants maintain the same status. In particular, participants can exit either with an abort status or a $\perp$ status, where $\perp$ indicates a critical failure. A critical failure depends on the context of the construction; for the generic construction we present in Section 4.5, a critical failure occurs when the adversary can cause the protocol to abort in the final round of the protocol, which in turn has consequences for the proof of security.

Prior notions in the literature considered only strong robustness; however, introducing a weaker variant allows us to define a more efficient construction that fulfills our other notions of secrecy and indistinguishability.

Remark 6 (Number of Honest Players)**.** The robustness experiment in Figure 4.4 assumes an honest majority of players. If an adversary were allowed to control the majority of participants, it could split the view of honest participants, unless some additional structure is assumed, such as a public bulletin board.

Common Oracles. We define common oracles that are used within the respective security experiments for a DKG in in Figure 4.5. Boxed algorithms are used by the robustness and indistinguishability games; dashed boxed algorithms are used by the zero-knowledge game.

4.4.1.1 Strong Robustness

We show the strong robustness attack experiment in Figure 4.4, where its differences with weak robustness are outlined in a box.

The advantage of an adversary $\mathcal{A}$ against D in the strong robustness experiment is

$$\mathsf{Adv}^{\text{str-rbst}}_{\mathsf{D},\mathcal{A}}(\lambda) = \Pr[\mathsf{Exp}^{\text{str-rbst}}_{\mathsf{D},\mathcal{A}}(\lambda) = 1]$$

Definition 4.4.3. *A DKG* D *is* **strongly robust** *if for all probabilistic polynomial-time adversaries* $\mathcal{A}$*, the function* $\mathsf{Adv}^{\text{str-rbst}}_{\mathsf{D},\mathcal{A}}(\lambda)$ *is negligible.*

The strong robustness attack game allows $\mathcal{A}$ to participate in key generation and control up to $t-1$ players, the set of which we denote by $\mathsf{corrupt}$. We denote the set of honest players as honest. The robustness attack game has oracles $\mathcal{O}^{\mathsf{PerformRound}_\ell}, \ell \in \{0, \ldots, \kappa - 1\}$ and $\mathcal{O}^{\mathsf{Finalize}}$ in common with other games, so we show these oracles in Figure 4.5. The adversary

$\boxed{\mathsf{Exp}^{\text{str-rbst}}_{\mathsf{D},\mathcal{A}}(\lambda)}$ $\boxed{\mathsf{Exp}^{\text{wk-rbst}}_{\mathsf{D},\mathcal{A}}(\lambda)}$

$\mathsf{rc} \leftarrow 0$ // *Ensure rounds are queried in order*

$\mathsf{inp}_{11} \leftarrow \emptyset, \ldots, \mathsf{inp}_{1n} \leftarrow \emptyset$ // *Simulate peer-to-peer channels*

$(n, t, \mathsf{corrupt}, \mathsf{st}_A) \leftarrow \mathcal{A}(1^\lambda)$; **return** 0 **if** $n < t \vee \mathsf{corrupt} \not\subset [n] \vee |\mathsf{corrupt}| \geq t$

$\mathsf{honest} \leftarrow [n] \setminus \mathsf{corrupt}$

return 0 **if** $|\mathsf{honest}| < t$

for $i \in \mathsf{honest}$ **do**

$(\mathsf{st}_i, \mathsf{outp}_{0i}, \mathsf{bmsg}_i) \leftarrow \mathsf{PerformRound}_0(\lambda, n, t, i)$

$\mathsf{inp}_{1k}[i] \leftarrow \mathsf{outp}_{0i}[k], \ \forall\ k \in [n]$

// *Set party i's peer-to-peer messages for all other parties*

$\mathcal{A}^{\mathcal{O}^{\mathsf{PerformRound}_\ell}, \mathcal{O}^{\mathsf{Finalize}}}(\mathsf{st}_A, \{\mathsf{inp}_{1i}\}_{i \in \mathsf{corrupt}}, \{\mathsf{bmsg}_i\}_{i \in \mathsf{honest}})$

return 0 **if** $\mathsf{rc} \neq \kappa$ // *Prevents trivial win by forcing coordination*

$\boxed{\textbf{if } \exists i \in \mathsf{honest} \ : \ \mathsf{status}_i \neq \mathsf{accept}}$

$\boxed{\textbf{return } 1}$

$\boxed{\textbf{if } \exists i, j \in \mathsf{honest} \ : \ \mathsf{status}_i \neq \mathsf{status}_j}$

$\boxed{\textbf{return } 1}$

$\boxed{\text{// } \mathcal{A} \textit{ wins if any honest party ends in a disjoint state}}$

$\boxed{\textbf{if } \forall i \in \mathsf{honest} \ : \ \mathsf{status}_i = \mathsf{abort}}$

$\boxed{\textbf{return } 0}$

$\boxed{\text{// } \mathcal{A} \textit{ loses if all honest parties end in a consistent aborted state}}$

for $i, j \in \mathsf{honest}$ **do**

return 1 **if** $\mathsf{pk}_i \neq \mathsf{pk}_j$

return 1 **if** $\mathsf{qual}_i \neq \mathsf{qual}_j$

return 1 **if** $\mathsf{aux}_i \neq \mathsf{aux}_j$

$\mathsf{pk} = \mathsf{pk}_i, i \in \mathsf{honest}$ // *All honest parties at this point have the same view of* pk

$\mathsf{sk} \leftarrow \mathsf{D.Recover}(\mathsf{pk}, \{(i, \mathsf{sk}_i)\}_{i \in \mathsf{honest}})$

return 1 **if** $\mathsf{sk} = \perp$

return 0

Figure 4.4: Robustness experiment for a DKG. Weak robustness differences are highlighted in a dashed box; strong robustness differences are highlighted in a solid box.

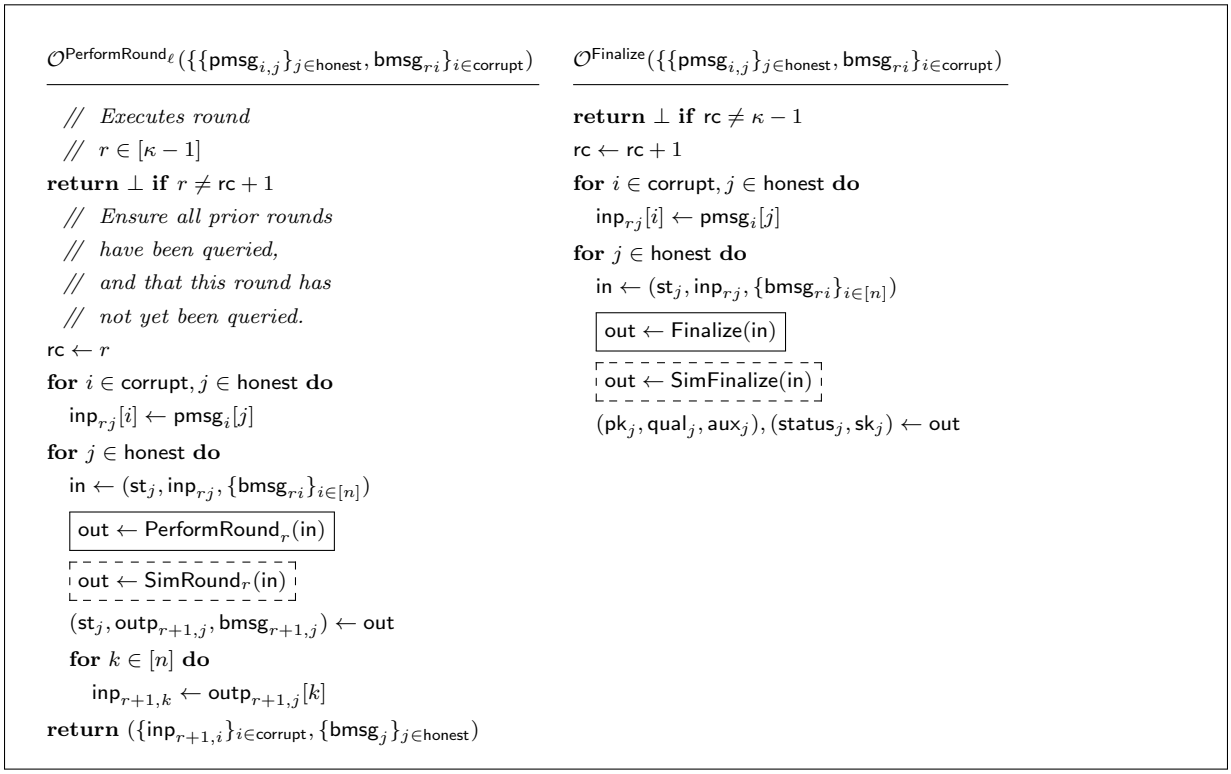


Figure 4.5: Common oracles used within security experiments for a distributed key generation scheme. Boxed algorithms are used by the robustness and indistinguishability games; dashed boxed algorithms are used by the zero-knowledge game.

participates in the protocol by exchanging peer-to-peer and broadcast messages with honest players in each round ℓ via the oracle $\mathcal{O}^{\mathsf{PerformRound}_\ell}$. The experiment is responsible for exchanging messages between parties and enforcing assumptions of the underlying network channels. Additionally, the experiment ensures that rounds must be queried in order, and that no round is repeated.

$\mathcal{A}$ wins the strong robustness game under three conditions. First, if it causes any honest player to end in a non-accepting state. Second, if it causes any honest player's view of $\mathsf{pk}, \mathsf{qual}$, or aux to be different from any other honest player's view. Finally, $\mathcal{A}$ wins if it can cause the resulting keypair to be invalid with respect to the target key generation algorithm (recall that $\mathsf{D.Recover}$ tests for validity of the keypair, and outputs $\perp$ if sk is invalid).

4.4.1.2 Weak Robustness

We similarly present the weak robustness attack experiment in Figure 4.4, where its differences with strong robustness are outlined in a dashed box. Intuitively, weak robustness differs from strong robustness in that it allows honest parties to end in an aborted state abort, so long as *all* honest parties end in the same state. Otherwise, if all honest parties end in an accepting state, it is identical to the notion of strong robustness.

When considering simulation-based definitions, the definition either considers a scheme that is strongly robust, or where the adversary can abort the protocol at any point during the protocol execution, even after seeing the output key material. However, weak robustness allows the adversary to abort the protocol execution, but only before some critical state; i.e., before the adversary has learned any information about the output key. As such, it is a more fine-grained notion of tolerated aborts than a simulation-based notion of abort, as weak robustness considers where in the protocol execution the adversary is allowed to abort.

The advantage of an adversary $\mathcal{A}$ against the DKG D in the weak robustness experiment is

$$\mathsf{Adv}^{\text{wk-rbst}}_{\mathsf{D},\mathcal{A}}(\lambda) = \Pr[\mathsf{Exp}^{\text{wk-rbst}}_{\mathsf{D},\mathcal{A}}(\lambda) = 1]$$

Definition 4.4.4. *A DKG* D *is* **weakly robust** *if for all probabilistic polynomial-time adversaries* $\mathcal{A}$*, the function* $\mathsf{Adv}^{\text{wk-rbst}}_{\mathsf{D},\mathcal{A}}(\lambda)$ *is negligible.*

$\mathcal{A}$ wins the weak robustness game under three conditions. First, $\mathcal{A}$ wins if honest players do not all finish with the same status. Otherwise, if all honest players have status accept, then $\mathcal{A}$ wins if the honest players' view of pk or qual are not all the same or if the resulting keypair is invalid with respect to the target key generation algorithm.

4.4.2 Zero Knowledge

We next turn to defining zero-knowledge for a DKG. Zero knowledge requires that a probabilistic polynomial-time adversary $\mathcal{A}$ allowed to control up to $t-1$ participants does not gain any additional advantage in its effort to learn sk than it would against the target key generation algorithm. This notion is formalized by the requirement of *simulatability*: that the DKG can be simulated to an adversary with respect to some challenge $\hat{\Delta} \in \hat{P}$, where $\hat{\Delta}$ is a valid public key for the target key generation algorithm. We formalize this notion in Figure 4.6. The oracle $\mathcal{O}^{\mathsf{PerformRound}_\ell}$ is defined in Figure 4.5.

Note that we implicitly assume weak robustness, as the adversary cannot win if it causes all participants to non-fatally abort.

Our definition of zero-knowledge is distinct from the simulation-based notion of secrecy introduced by Gennaro et al. [GJKR07] in that we formally define the notion with respect to a target key generation algorithm. whereas the definition by Gennaro et al. assumes random group elements as the public key to simulate. In some settings (where public and secret keys have a unique correspondence), then these definitions are the same. If not, then such simulation-based notions fail to consider indistinguishability, as discussed below.

We now define this notion of zero-knowledge more formally. The advantage of an adversary $\mathcal{A}$ against a DKG D with respect to the simulation algorithms $(\mathsf{SimRound}_i, i \in \{0, \ldots, \kappa-1\}, \mathsf{SimFinalize})$ in the zero-knowledge experiment as defined in Figure 4.6 is

$$\mathsf{Adv}^{\mathrm{zk}}_{\mathsf{D},\mathcal{A},\mathsf{SimRound}_i,\mathsf{SimFinalize}}(\lambda) = \left|\Pr[\mathsf{Exp}^{\mathrm{zk}}_{\mathsf{D},\mathcal{A},\mathsf{SimRound}_i,\mathsf{SimFinalize}}(\lambda) = 1] - 1/2\right|$$

Definition 4.4.5. *A DKG* D *is* **zero-knowledge** *if there exist efficient algorithms* $\mathsf{SimRound}_i$ *(for each* $i < \kappa$*) and* $\mathsf{SimFinalize}$ *such that for all probabilistic polynomial-time adversaries* $\mathcal{A}$*:*

$$\mathsf{Adv}^{\mathrm{zk}}_{\mathsf{D},\mathcal{A},\mathsf{SimRound}_i,\mathsf{SimFinalize}}(\lambda) \text{ is negligible.}$$

The zero-knowledge attack game in Figure 4.6 requires $\mathcal{A}$ to successfully guess if the environment that it is in is real or simulated with respect to a challenge $\hat{\Delta}$, where $\hat{\Delta}$ is a public key generated by $\mathsf{TK.KeyGen}$. The game accepts as input the security parameter λ, and internally to the experiment, samples a bit $b \in \{0,1\}$ at random. In the simulated setting (when $b = 1$), the environment simulates the honest participants with respect to $\hat{\Delta}$ by employing the simulation algorithms $(\mathsf{SimRound}_i, i \in \{0, \ldots, \kappa-1\}, \mathsf{SimFinalize})$. The adversary wins by default if the environment cannot simulate the protocol; i.e, the resulting group public key pk does not equal $\hat{\Delta}$. Otherwise, the adversary wins if it successfully guesses whether it is playing against the real or simulated protocol.

$\mathsf{Exp}^{\mathrm{zk}}_{\mathsf{D},\mathcal{A},\mathsf{SimRound}_i,\mathsf{SimFinalize}}(\lambda)$

$b \xleftarrow{\$} \{0,1\}$
$z \xleftarrow{\$} \{0,1\}$ // *Random coin for losing conditions*
$\mathsf{rc} \leftarrow 0$ // *Ensure rounds are queried in order*
$\mathsf{inp}_{11} \leftarrow \emptyset, \ldots, \mathsf{inp}_{1n} \leftarrow \emptyset$ // *Simulate peer-to-peer channels*
$(n, t, \mathsf{corrupt}, \mathsf{st}_A) \leftarrow \mathcal{A}(1^\lambda)$; $\mathsf{honest} \leftarrow [n] \setminus \mathsf{corrupt}$
if $n < t \vee \mathsf{corrupt} \not\subset [n] \vee |\mathsf{corrupt}| \geq t$
  return z
$(\cdot, \hat{\Delta}) \xleftarrow{\$} \mathsf{TK.KeyGen}(1^\lambda)$
for $i \in \mathsf{honest}$ **do**
  if $b = 0$ **then** $(\mathsf{st}_i, \mathsf{outp}_i, \mathsf{bmsg}_i) \leftarrow \mathsf{PerformRound}_0(\lambda, n, t, i)$
  else $(\mathsf{st}_i, \mathsf{outp}_i, \mathsf{bmsg}_i) \leftarrow \mathsf{SimRound}_0[\mathsf{H}](\lambda, n, t, i, \hat{\Delta})$
  $\mathsf{inp}_{1k}[i] \leftarrow \mathsf{outp}_{0i}[k],\ \forall\ k \in [n]$
  // *Set party i's peer-to-peer messages for all other parties*
$b' \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{PerformRound}_\ell}, \mathcal{O}^{\mathsf{Finalize}}}(\mathsf{st}_A, \{\mathsf{inp}_{1j}\}_{j \in \mathsf{corrupt}}, \{\mathsf{bmsg}_i\}_{i \in \mathsf{honest}})$
if $\mathsf{rc} \neq \kappa$ // *Prevents trivial win by forcing coordination*
  return z
if $\mathsf{status}_j = \mathsf{abort}$ for all $j \in \mathsf{honest}$
  return z
  // *$\mathcal{A}$ cannot win if all honest parties non-fatally abort*
if $b = 1$
  return 1 **if** $\exists\ i \in \mathsf{honest} : \mathsf{pk}_i \neq \hat{\Delta}$
  // *$\mathcal{A}$ wins when $b = 1$ if the game is not simulated with respect to $\hat{\Delta}$*
return 1 **if** $b \stackrel{?}{=} b'$
return 0

Figure 4.6: Zero-knowledge experiment for a DKG.

4.4.3 Indistinguishability

The security of any key generation protocol requires that the secret key be hard to guess or learn from the public transcript. In the single-party setting, the adversary does not

have any influence over the sampling of key material, and so guaranteeing this property is straightforward. However, because the adversary is an active participant in a distributed key generation protocol, we must ensure that a subset of at most $t-1$ colluding players cannot bias key material.

To establish this property, we require that a DKG must generate keys that are *indistinguishable* from keys output by the target key generation algorithm. This property is implied by the notion of secrecy if the underlying construction assumes a one-to-one correspondence between secret and public keys. For example, the Gennaro et al. notion of simulatability assumes that indistinguishability of the resulting keypair is an implicit consequence of the ability to simulate with respect to a public key. However, simulatability does not necessarily imply indistinguishability, because the zero-knowledge game *only* guarantees the indistinguisahbility of public key material between the real and simulated environments. If there is a one-to-one relation between public and secret keys, proving zero-knowledge implies indistinguishability. For cryptosystems where a one-to-many relation can exist between public and private keys, such as in some lattice-based cryptosystems, proving that a DKG is zero-knowledge may not necessarily imply indistinguishability from its target key generation scheme.

Because our generic construction similarly assumes a one-to-one correspondence of secret and public keys, we do not make use of this notion of indistinguishability in the proof of security for our generic construction, as the proof of zero-knowledge implies indistinguishability. For future reference, we give the notion of indistinguishability more formally next.

Note that we implicitly assume weak robustness, as the adversary cannot win if it causes all participants to non-fatally abort.

The advantage of a distinguishing adversary $\mathcal{E}$ against a DKG D in the indistinguishability experiment in Figure 4.7 with respect to the target key generation algorithm TK and participating adversary $\mathcal{A}$ is

$$\mathsf{Adv}^{\mathrm{ind}}_{\mathsf{D},\mathsf{TK},\mathcal{A},\mathcal{E}}(\lambda) = \left|\Pr[\mathsf{Exp}^{\mathrm{ind}}_{\mathsf{D},\mathsf{TK},\mathcal{A},\mathcal{E}}(\lambda) = 1] - 1/2\right|$$

Definition 4.4.6. *A DKG* D *is* **indistinguishable** *from its target key generation algorithm if for all probabilistic polynomial-time adversaries* $\mathcal{A}$ *and computationally unbounded distinguishers* $\mathcal{E}$, $\mathsf{Adv}^{\mathrm{ind}}_{\mathsf{D},\mathsf{TK},\mathcal{A},\mathcal{E}}(\lambda)$ *is negligible.*

The attack game in Figure 4.7 allows for two distinct adversaries, $\mathcal{A}$ and $\mathcal{E}$. The oracles $\mathcal{O}^{\mathsf{PerformRound}_\ell}$ and RO are defined in Figure 4.5. $\mathcal{A}$ participates in the protocol and controls up to $t-1$ players. $\mathcal{E}$ is simply given two sets of keypairs after the protocol completes,

and is required to guess which keypair was generated by the distributed key generation protocol and which by the target key generation protocol. Note that $\mathcal{A}$ and $\mathcal{E}$ are assumed to be black box and act arbitrarily, but are not allowed to share state. Otherwise, the game could trivially be won simply by $\mathcal{A}$ informing $\mathcal{E}$ which public key was output from the distributed key generation protocol. To similarly prevent this trivial win condition, the keypair output from the distributed key generation execution must be valid, otherwise $\mathcal{A}$ could force the protocol to output an invalid keypair, which $\mathcal{E}$ could trivially distinguish from the output of $\mathsf{TK.KeyGen}$.

4.5 A Generic DKG Construction

We now introduce a weakly robust generic DKG construction that assumes an honest majority, shown in Figure 4.8. The DKG is initialized with a NIKE NK as defined in Section 2.4, an AgVSS A as defined in Section 4.3, and a hash function H. We present the concrete construction in Section 4.6. Recall that setup functions $\mathsf{par} \leftarrow \mathsf{NK.Setup}(1^\lambda)$; $\mathsf{par}' \leftarrow \mathsf{A.Setup}(1^\lambda)$ are performed at setup, not at the time of key generation.

Our generic construction employs the following building blocks:

- A secure hash function H, where H_1 and H_2 are domain-separated instances of H, such that:
  - $\mathsf{H}_1 : (\{D_j\}_{j\in[n]}, \mathsf{aux}) \mapsto v$: Accepts as input a tuple, where the first element in the tuple is a set of n AgVSS commitments ordered canonically, and the second element is an auxiliary element aux as output by H_2 below. The output is the tweak employed by the AgVSS aggregation algorithm.
  - $\mathsf{H}_2 : \{\psi_j\}_{j\in[n]} \mapsto \mathsf{aux}$: Accepts as input a set of n elements, and outputs an auxiliary element aux.
- An aggregatable verifiable secret sharing scheme $\mathsf{A}[\mathsf{H}_1] = (\mathsf{Setup}, \mathsf{Share}, \mathsf{Verify}, \mathsf{Recover}, \mathsf{GetPub}, \mathsf{AggPriv}, \mathsf{AggPub}, \mathsf{Tweak})$ that is zero-indexed.
- A non-interactive key exchange (NIKE) $\mathsf{NK} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Verify}, \mathsf{SharedKey})$.

The secret and public domains of the AgVSS and NIKE must be the same as that of the target key generation scheme, and the distribution over the secret domain for the NIKE must be the same as that of the target key generation scheme. Further, we assume that the NIKE outputs secret and public keys with strictly a one-to-one relation.

$\mathsf{Exp}^{\mathrm{ind}}_{\mathsf{D},\mathsf{TK},\mathcal{A},\mathcal{E}}(\lambda)$

$b \xleftarrow{\$} \{0,1\}$
$z \xleftarrow{\$} \{0,1\}$ // *Random coin for losing conditions*
$\mathsf{rc} \leftarrow 0$ // *Ensure rounds are queried in order*
$\mathsf{inp}_{11} \leftarrow \emptyset, \ldots, \mathsf{inp}_{1n} \leftarrow \emptyset$ // *Simulate peer-to-peer channels*
$(n, t, \mathsf{corrupt}, \mathsf{st}_A) \leftarrow \mathcal{A}(\lambda)$
if $n < t \vee \mathsf{corrupt} \not\subset [n] \vee |\mathsf{corrupt}| \geq t$ **then return** z
$\mathsf{honest} \leftarrow [n] \setminus \mathsf{corrupt}$
for $i \in \mathsf{honest}$ **do**
  $(\mathsf{st}_i, \mathsf{outp}_{0i}, \mathsf{bmsg}_i) \leftarrow \mathsf{PerformRound}_0[\mathsf{H}](\lambda, n, t, i)$
  $\mathsf{inp}_{1k}[i] \leftarrow \mathsf{outp}_{0i}[k],\ \forall\ k \in [n]$
$\mathcal{A}^{\mathcal{O}^{\mathsf{PerformRound}_\ell}, \mathcal{O}^{\mathsf{Finalize}}, \mathsf{RO}}(\{\mathsf{inp}_{1i}\}_{i \in \mathsf{corrupt}}, \{\mathsf{bmsg}_i\}_{i \in \mathsf{honest}}, \mathsf{st}_A)$
if $\mathsf{rc} \neq \kappa$ // *Prevents trivial win by forcing coordination*
  return z
if $\{j : \mathsf{status}_j = \mathsf{abort}\}_{j \in \mathsf{honest}} = \mathsf{honest}$
  return z
if $\exists i \in \mathsf{honest} : \mathsf{status}_i \neq \mathsf{accept}$
  return z

for $i, j \in \mathsf{honest}$ **do**
  return 1 **if** $\mathsf{pk}_i \neq \mathsf{pk}_j$
  return 1 **if** $\mathsf{qual}_i \neq \mathsf{qual}_j$
$\mathsf{pk}_0 \leftarrow \mathsf{pk}$
$\mathsf{sk}_0 \leftarrow \mathsf{D}[\mathsf{H}].\mathsf{Recover}(\mathsf{pk}_0, t, \{(i, \mathsf{sk}_i)\}_{i \in \mathsf{honest}})$
// *All parties at this point have the same view of* pk *and* qual
return 1 **if** $\mathsf{sk}_0 = \perp$
$(\mathsf{sk}_1, \mathsf{pk}_1) \xleftarrow{\$} \mathsf{TK}.\mathsf{KeyGen}(\lambda)$
$b' \leftarrow \mathcal{E}((\mathsf{pk}_b, \mathsf{sk}_b), (\mathsf{pk}_{1-b}, \mathsf{sk}_{1-b}))$
return 1 **if** $b' = b$
return 0

Figure 4.7: Indistinguishability experiment for a DKG.

$\mathsf{PerformRound}_0[\mathsf{H}](\lambda, n, t, i)$

$(\alpha_i, A_i) \xleftarrow{\$} \mathsf{NK.KeyGen}()$
$\mathsf{in} \leftarrow (\alpha_i, n, t)$
$\{(j, \phi_{ij})\}_{j=1}^n, D_i \xleftarrow{\$} \mathsf{A}[\mathsf{H}_1].\mathsf{Share}(\mathsf{in})$
$(\beta_i, \mathsf{E}_i) \xleftarrow{\$} \mathsf{NK.KeyGen}()$
$\mathsf{bmsg}_{0,i} \leftarrow (A_i, \mathsf{E}_i, D_i)$
for $j \in [n], j \neq i$ **do**
  $\mathsf{pmsg}_{0,ij} \leftarrow (j, \phi_{ij})$
$\mathsf{st}_i \leftarrow (\beta_i, \phi_{ii}, D_i, \mathsf{E}_i)$
$\mathsf{outp} \leftarrow \{\mathsf{pmsg}_{0,ij}\}_{j \in [n], j \neq i}$
return $(\mathsf{st}_i, \mathsf{out}_p, \mathsf{bmsg}_{0,i})$

$\mathsf{PerformRound}_1[\mathsf{H}](\mathsf{st}_i, \mathsf{inp}_1, \mathsf{inb}_1)$

$\mathsf{parse}\ \{(i, \phi_{ji})\}_{j=1}^n \leftarrow \mathsf{inp}_1$
$\mathsf{parse}\ \{(A_j, \mathsf{E}_j, D_j)\}_{j=1}^n \leftarrow \mathsf{inb}_1$
for $j \in [n], j \neq i$ **do**
  if $\mathsf{A}[\mathsf{H}_1].\mathsf{Verify}(i, \phi_{ji}, D_j) \neq 1$
  or $\mathsf{A}[\mathsf{H}_1].\mathsf{GetPub}(0, D_j) \neq A_j$
    $\mathsf{st}_i.\mathsf{status} = \mathsf{abort};\ \mathsf{bmsg}_{1,i} \leftarrow \perp$
    return $(\mathsf{st}_i, \perp, \mathsf{bmsg}_{1,i})$
$\mathsf{st}_i \leftarrow \mathsf{st}_i \cup \{(j, \phi_{ji})\}_{j \in [n]}$
$\mathsf{bmsg}_{1,i} \leftarrow \mathsf{accept}$
return $(\mathsf{st}_i, \perp, \mathsf{bmsg}_{1,i})$

$\mathsf{PerformRound}_2[\mathsf{H}](\mathsf{st}_i, \emptyset, \mathsf{inb}_2)$

if $\perp \in \mathsf{inb}_2$
  $\mathsf{st}_i.\mathsf{status} = \mathsf{abort}$
  return $(\mathsf{st}_i,\ \perp,\ \perp)$
$\mathsf{bmsg}_{2,i} \leftarrow \beta_i$
return $(\mathsf{st}_i,\ \perp,\ \mathsf{bmsg}_{2,i})$

$\mathsf{Finalize}[\mathsf{H}](\mathsf{st}_i, \emptyset, \mathsf{inb}_3)$

if $\mathsf{st}_i.\mathsf{status} = \mathsf{abort}$
  return $(\perp, \perp, \perp), (\mathsf{abort}, \perp)$
$\mathsf{parse}\ \{\beta_j\}_{j \in [n], j \neq i} \leftarrow \mathsf{inb}_3$
$\mathsf{qual} \leftarrow \emptyset;\ \mathsf{corrupt} \leftarrow \emptyset$
for $j \in [n], j \neq i$ **do**
  if $\mathsf{NK.Verify}(\beta_j, \mathsf{E}_j) = 1$
    $\psi_j \leftarrow \mathsf{NK.SharedKey}(\beta_j, A_j)$
    $\mathsf{qual} \leftarrow \mathsf{qual} \cup \{j\}$
  else $\mathsf{corrupt} \leftarrow \mathsf{corrupt} \cup \{j\}$
if $\mathsf{corrupt} \neq \emptyset$
  Broadcast $\{\phi_{ji}\}_{j \in \mathsf{corrupt}}$
  Receive $\{\phi_{jk}\}_{j \in \mathsf{corrupt}, k \in \mathsf{qual}}$
  for $k \in \mathsf{qual}$ **do**
    if $\exists\ j\ :\ \mathsf{A}[\mathsf{H}_1].\mathsf{Verify}(i, \phi_{jk}, D_j) \neq 1$
      $\mathsf{qual} \leftarrow \mathsf{qual} \setminus \{k\}$
for $j \in \mathsf{corrupt}$ **do**
  $\alpha_j \leftarrow \mathsf{A}[\mathsf{H}_1].\mathsf{Recover}(t, \{(k, \phi_{jk})\}_{k \in \mathsf{qual}})$
  $\psi_j \leftarrow \mathsf{NK.SharedKey}(\alpha_j, \mathsf{E}_j)$
$v \leftarrow \mathsf{A}[\mathsf{H}_1].\mathsf{Tweak}(\{D_j\}_{j=1}^n, \{\psi_j\}_{j=1}^n)$
$\mathsf{sk}_i \leftarrow \mathsf{A}[\mathsf{H}_1].\mathsf{AggPriv}(i, \{\phi_{ji}\}_{j \in [n]}, v)$
$C \leftarrow \mathsf{A}[\mathsf{H}_1].\mathsf{AggPub}(\{D_j\}_{j \in [n]}, v)$
$\mathsf{pk} \leftarrow \mathsf{A}[\mathsf{H}_1].\mathsf{GetPub}(0, C)$
  // *Derive public keys*
for $i \in \mathsf{qual}$ **do**
  $\mathsf{pk}_i = \mathsf{A}[\mathsf{H}_1].\mathsf{GetPub}(i, C)$
$\mathsf{aux} \leftarrow \{\mathsf{pk}_i\}_{i \in \mathsf{qual}}$
return $(\mathsf{pk}, \mathsf{qual}, \mathsf{aux}), (\mathsf{accept}, \mathsf{sk}_i)$

Figure 4.8: A generic DKG protocol. See corresponding text for full description.

In addition to these building blocks, our construction requires that these primitives are algebraically composable. Intuitively, the secret key output from $\mathsf{NK.KeyGen}$ should be a valid secret that can be shared via $\mathsf{A.Share}$, and the AgVSS commitment should be verifiable with respect to the public NIKE key. Towards this latter point, we require Assumption 1 to hold.

Assumption 1. *For* $\mathsf{par} \leftarrow \mathsf{NK.Setup}(1^\lambda)$ *and* $\mathsf{par}' \leftarrow \mathsf{A.Setup}(1^\lambda)$ *any* $(\mathsf{sk}, \mathsf{pk}) \xleftarrow{\$} \mathsf{NK.KeyGen}()$*, and any* $(n, t) \in \mathbb{N}$ *such that* $n \geq t$*; and any* $C \subseteq [n], |C| \geq t$*, Equation 4.8 holds:*

$$\begin{aligned} \mathsf{NK.Verify}(\mathsf{sk}, \mathsf{pk}) = 1, \ \textit{where} \\ (\{(i, \phi_i)\}_{i\in[n]}), D) \xleftarrow{\$} \mathsf{A}[\mathsf{H}_1].\mathsf{Share}(\mathsf{sk}, n, t) \ \textit{and} \\ \mathsf{pk} \leftarrow \mathsf{A}[\mathsf{H}_1].\mathsf{GetPub}(0, D) \ \textit{and} \\ \mathsf{sk} \leftarrow \mathsf{A}[\mathsf{H}_1].\mathsf{Recover}(t, \{(i, \phi_i)\}_{i\in C}) \end{aligned} \tag{4.8}$$

4.5.1 Key Generation Protocol

As shown in Figure 4.8, the key generation protocol is completed in three network rounds in the optimistic setting and four network rounds in the worst-case setting.

Each $\mathsf{PerformRound}_r$ operation is defined with respect to participant i. The peer-to-peer channel inputs for round r are inp_r and outputs are outp_r, composed of the peer-to-peer messages pmsg_j. The broadcast channel inputs are inb_r composed of the broadcast messages bmsg_j, sent by participant $j, j \in [n]$.

First, in $\mathsf{PerformRound}_0$, each participant i performs $\mathsf{NK.KeyGen}$ twice; first to generate a secret key α_i and its commitment A_i, and second to generate a random value β_i and its commitment E_i. Then, each participant i performs an AgVSS secret sharing of α_i, outputting n secret shares $\{(j, \phi_{ij})\}_{j\in[n]}$, and an AgVSS commitment D_i. Finally, each participant i broadcasts its NIKE and AgVSS commitments to all other players via a broadcast message $\mathsf{bmsg}_{0,i}$. Additionally, each participant i sends one secret share to every other player $j \neq i$ via a peer-to-peer message $\mathsf{pmsg}_{0,ij}$, keeping one share for itself.

In $\mathsf{PerformRound}_1$, each participant i receives shares $\{(i, \phi_{ji})\}_{j\in[n],j\neq i}$ from all other players via the peer-to-peer channel input inp_1 and commitments $\{(A_j, \mathsf{E}_j, D_j)\}_{j\in[n],j\neq i}$ from all other players via the broadcast channel input inb_1. Each participant then verifies the correctness of their received share with respect to its corresponding commitment. Additionally, each participant i verifies that the secret committed to in D_i is the same secret committed to in A_i, by checking that $\mathsf{A.GetPub}(0, D_i) = A_i$. If any check fails, the participant will broadcast a $\perp$ message, and set their status to abort. Any participant that

sets their status to abort results in that participant terminating (immediately exiting) the protocol. Else, it will broadcast an accept message.

In $\mathsf{PerformRound}_2$, participants receive as input status messages from all other participants via the broadcast channel input inb_2. If any participant received a $\perp$ message, that participant immediately exits $\mathsf{PerformRound}_2$ after setting their status to abort.[1] Otherwise, each participant i broadcasts β_i to all other participants (which is the opening to the commitment E_i sent in $\mathsf{PerformRound}_0$).

Finally, in Finalize, each participant i receives as input the openings $\{\beta_j\}_{j\in[n],j\neq i}$ from all other participants via the broadcast channel inb_3. For the values they received from every other player $j \neq i$, they first verify the correctness of the opening β_j with respect to E_j. Depending on the output of this check, each participant i does the following:

- If the opening β_j is valid, participant i then derives the blinding factor ψ_j by performing $\psi_j \leftarrow \mathsf{NK.SharedKey}(\beta_j, A_j)$.
- If the opening β_j is invalid, the set of participants qual that followed the protocol honestly cooperate to derive the correct ψ_j for the cheating participants corrupt. They do so by broadcasting their respective shares from the misbehaving parties, and then use the set of received shares to recover α_j. Note that even if an adversary at this stage were to broadcast incorrect shares, because we assume an honest majority of players, then all honest participants will receive a sufficient number of honest shares to recover successfully, and can verify the correctness of the recovered secret against the commitment sent in earlier rounds. They can do so by checking if $\mathsf{A}[\mathsf{H}_1].\mathsf{Verify}(i, \phi_{jk}, D_j) = 1$ for each identifier $j \in \mathsf{corrupt}, k \in \mathsf{qual}$, updating $\mathsf{qual} \leftarrow \mathsf{qual} \setminus \{k\}$ if the check does not hold.

  Each remaining participant then performs $\mathsf{NK.SharedKey}(\alpha_j, \mathsf{E}_j)$, obtaining ψ_j as the result. Due to the assumption that at least t honest participants exist in qual, this step will always complete successfully.

Next, each participant derives $v \leftarrow \mathsf{A}[\mathsf{H}_1].\mathsf{Tweak}(O, \{\psi_1, \ldots, \psi_n\})$, where $O = \{D_j\}_{j\in[n]}$ is the set of all AgVSS commitments for each participant. Each participant i then performs A.AggPriv and A.AggPub, using as input the set of all their received shares $P = \{\phi_{ji}\}_{j\in[n]}$, the set of AgVSS commitments O, and v. Each participant i employs the output of A.AggPriv and A.AggPub to determine their aggregated secret key share sk_i and commitment C. The group public key pk is then derived using C. Similarly, the set of each

[1] Because the fail message $\perp$ is sent on a broadcast channel, if any participant receives a fail message, then all participants receive that message

participant's individual public key pk_i is derived using C, and then output as the auxiliary data.

The tweak ensures that the resulting public key is unknown to an adversary at the time when it picks its contributions to the protocol. As such, at the beginning of the DKG, the adversary cannot adaptively pick its contributions so that the resulting public key can be influenced predictably. In other words, the adversary cannot force the resulting public key to "hit" some desired outcome, such as being even or odd, for example.

Participants complete $\mathsf{Finalize}$ after setting their status to accept and updating their internal state. The public output at the end of $\mathsf{Finalize}$ is $(\mathsf{pk}, \mathsf{qual}, \mathsf{aux})$. The private output to each participant i is $(\mathsf{status}, \mathsf{sk}_i)$.

4.5.2 Recovery Algorithm

While most practical DKG applications do not require the participants to actually come together to recover the secret key, it is possible that some applications may wish to do so. We define the recovery algorithm now.

- $\mathsf{Recover}(\mathsf{pk}, t, \{(i, \mathsf{sk}_i)\}_{i \in C}) \to \mathsf{sk}/\perp$: Accepts the public key $\mathsf{pk} \in \mathbb{G}$ and $|C| \geq t$ tuples $\{(i, \mathsf{sk}_i)\}_{i \in C}$ consisting of participant i's identifier and their secret key. Using these inputs, perform the following steps:

  1. Derive $\mathsf{sk} = \mathsf{A.Recover}(t, \{(i, \mathsf{sk}_i)\}_{i \in C})$.
  2. If $\mathsf{TK.Verify}(\mathsf{sk}, \mathsf{pk}) \neq 1$, output $\perp$.
  3. Output sk.

4.5.3 Security of the Generic Protocol

We next demonstrate that the generic protocol in Figure 4.8 is weakly robust and zero-knowledge.

4.5.3.1 Robustness

Theorem 7. *The generic construction is weakly robust in the honest majority setting ($t - 1 < n/2$) against an adversary $\mathcal{A}$ controlling up to $t - 1$ participants in the random*

oracle model, given a zero-indexed and secure AgVSS A *and a binding NIKE* NK*, such that Assumption 1 holds.*

Concretely, for every adversary $\mathcal{A}$ *that attacks the scheme, there exist adversaries* $(\mathcal{B}_1, \mathcal{B}_2)$ *that run in about the same time as* $\mathcal{A}$ *such that*

$$\mathsf{Adv}^{\text{wk-rbst}}_{\mathsf{D},\mathcal{A}}(\lambda) \leq \mathsf{Adv}^{\text{bind}}_{\mathsf{NK},\mathcal{B}_1}(\lambda) + \mathsf{Adv}^{\text{unq}}_{\mathsf{A}[\mathsf{H}_1],\mathcal{B}_2}(\lambda). \tag{4.9}$$

Proof. We prove Theorem 7 by a sequence of games.

Game 0. This is the DKG weak robustness game as in Figure 4.4, applied to the construction in Figure 4.8.

Let W_0 be the event that $\mathcal{A}$ wins in Game 0. Then,

$$\mathsf{Adv}^{\text{wk-rbst}}_{\mathsf{D},\mathcal{A}}(\lambda) = \Pr[W_0]$$

Game 1. The only difference in Game 1 is if $\mathcal{A}$ outputs a tuple $(\mathsf{E}_j, \mathsf{pk}_j, D_j, \beta_j), j \in \mathsf{corrupt}$ where $(\mathsf{E}_j, \mathsf{pk}_j, D_j)$ are output in $\mathsf{PerformRound}_0$ and β_j is output in $\mathsf{PerformRound}_2$ such that Equation 4.10 holds, then the experiment aborts.

$$\begin{aligned} &\mathsf{A}[\mathsf{H}_1].\mathsf{GetPub}(0, D_j) = \mathsf{pk}, \text{ and} \\ &\mathsf{NK.Verify}(\beta_j, \mathsf{E}_j) \to 1, \text{ but} \\ &\mathsf{NK.SharedKey}(\beta_j, \mathsf{pk}_j) \neq \mathsf{NK.SharedKey}(\alpha_j, \mathsf{E}_j) \end{aligned} \tag{4.10}$$

However, if the NIKE is binding and Assumption 1 holds, then Game 1 and Game 0 are indistinguishable to $\mathcal{A}$.

Reduction to NIKE Binding. Let $\mathcal{B}_1$ be an adversary playing against the NIKE binding game, as shown in Figure 2.5. Recall that in the binding game, the adversary accepts as input only the public security parameters par, and is required to output two pairs $(\mathsf{sk}_1^*, \mathsf{pk}_1^*, \mathsf{sk}_2^*, \mathsf{pk}_2^*)$ such that Verify outputs 1 for both pairs, but $\mathsf{NK.SharedKey}(\mathsf{sk}_1^*, \mathsf{pk}_2^*) \neq \mathsf{NK.SharedKey}(\mathsf{sk}_2^*, \mathsf{pk}_1^*)$.

As such, $\mathcal{B}_1$ simulates Game 1 to $\mathcal{A}$ in exactly the same way as Game 0. However, in $\mathsf{Finalize}$, if $\mathcal{A}$ outputs some β_j with respect to the commitments $(\mathsf{pk}_j, \mathsf{E}_j, D_j)$ output in $\mathsf{PerformRound}_0$ such that Equation 4.10 holds, then $\mathcal{B}_1$ recovers α_j, and outputs

$(\alpha_j, \mathsf{pk}_j, \beta_j, \mathsf{E}_j)$ as its output to the NIKE binding game. Recall that the only inputs to $\mathcal{B}_1$ from the NIKE binding game are the NIKE public parameters.

$\mathcal{B}_1$ recovers α_j by performing $\alpha_j \leftarrow \mathsf{A}[\mathsf{H}_1].\mathsf{Recover}(t, \mathsf{M}_j)$, where $\mathsf{M}_j = \{(k, \phi_{jk})\}_{k \in \mathsf{honest}}$. $\mathcal{B}_1$ can recover s_j because the checks in $\mathsf{PerformRound}_1$ completed successfully (else the honest participants would have aborted the protocol in $\mathsf{PerformRound}_1$ and $\mathsf{PerformRound}_2$), then each share in M_j is a valid share of the secret α_j committed to by D_j and pk_j. Further, because $\mathcal{B}_1$ simulates at least t honest players, it has a sufficient number of shares to perform this recovery step.

Hence, when $\mathcal{A}$ distinguishes Game 1 from Game 0 (i.e., by causing Game 1 to abort), then $\mathcal{B}_1$ wins.

Difference between Game 0 and Game 1. Let W_1 be the event that $\mathcal{A}$ wins in Game 1. Then,

$$\left|\Pr[W_1] - \Pr[W_0]\right| \leq \mathsf{Adv}^{\mathrm{bind}}_{\mathsf{NK},\mathcal{B}_1}(\lambda) \tag{4.11}$$

Game 2. The only difference in Game 2 is that in $\mathsf{Finalize}$, if Equation 4.12 holds, then the experiment aborts.

$$\begin{aligned} &\mathsf{sk} \leftarrow \mathsf{A}.\mathsf{Recover}(t, \{(i, \mathsf{sk}_i)\}_{i \in \mathsf{honest}}), \text{ but} \\ &\mathsf{NK}.\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}) \to 0 \end{aligned} \tag{4.12}$$

However, if the AgVSS is unique, then Game 1 and Game 2 are indistinguishable.

Reduction to AgVSS Uniqueness. Let $\mathcal{B}_2$ be an adversary playing against the AgVSS uniqueness game.

$\mathcal{B}_2$ follows the protocol as defined in Figure 4.8 honestly for all parties $i \in \mathsf{honest}$. However, in $\mathsf{Finalize}$, $\mathcal{B}_2$ recovers sk and tests to see if Equation 4.12 holds. $\mathcal{B}_2$ can recover sk without the involvement of $\mathcal{A}$ because it simulates at least t honest parties.

Recall that

$$\mathsf{sk}_i \leftarrow \mathsf{A}[\mathsf{H}_1].\mathsf{AggPriv}(i, \{\phi_{i,j}\}_{j \in [n]}, \{D_j\}_{j \in [n]}, \mathsf{aux})$$

and

$$C \leftarrow \mathsf{A}[\mathsf{H}_1].\mathsf{AggPub}(\{D_j\}_{j \in [n]}, \mathsf{aux})$$

The only values that differ between each honest party when performing this aggregation step are the sets $\{\phi_{i,j}\}_{j\in[n]}$.

If Equation 4.12 holds, then this means that at least one honest party $k \in \mathsf{honest}$ received a share $\phi_{j,k}$ from a corrupted party $j \in \mathsf{corrupt}$ such that $\mathsf{A}[\mathsf{H}_1].\mathsf{Verify}(k, \phi_{j,k}, D_j) = 1$ but where Equation 4.13 holds.

$$\begin{aligned}&\mathsf{A}[\mathsf{H}_1].\mathsf{Recover}(t, \{(\ell, \phi_{j,\ell})\}_{\ell\in\mathsf{honest}}) \neq \\ &\mathsf{A}[\mathsf{H}_1].\mathsf{Recover}(t, \{(\ell, \phi'_{j,\ell})\}_{\ell\in\mathsf{honest}})\end{aligned} \tag{4.13}$$

where $\phi_{j,\ell} = \phi'_{j,\ell}$, except for $\ell = k$. Let $\mathsf{M}_1 = \{(\ell, \phi_{j,\ell})\}_{\ell\in\mathsf{honest}}$ and $\mathsf{M}_2 = \{(\ell, \phi'_{j,\ell})\}_{\ell\in\mathsf{honest}}$. When this event has occurred, $\mathcal{B}_2$ submits $(t, \mathsf{M}_1, \mathsf{M}_2, D_j)$ as its output to the AgVSS uniqueness game. When $\mathcal{A}$ can distinguish Game 2 from Game 1, then $\mathcal{B}_2$ wins the AgVSS uniqueness game.

Difference between Game 1 and Game 2. Let W_2 be the event that $\mathcal{A}$ wins in Game 2. Then,

$$\big|\Pr[W_2] - \Pr[W_1]\big| \leq \mathsf{Adv}^{\mathrm{unq}}_{\mathsf{A}[\mathsf{H}_1],\mathcal{B}_2}(\lambda) \tag{4.14}$$

Finishing the Proof. Recall that in the weak robustness game, the adversary has four opportunities to win, assuming all honest parties have not aborted the protocol. First, if any honest party has a disjoint view of pk, second, if any honest party has a disjoint view of qual, and third, if any honest party has a disjoint view of aux. Finally, the adversary wins if the sk that is recovered from the honest parties' shares is invalid.

Because we assume a broadcast channel, if any honest party aborts in $\mathsf{PerformRound}_1$ or $\mathsf{PerformRound}_2$, then all honest parties will likewise abort. And so if any honest party proceeds to $\mathsf{Finalize}$, then all honest parties will proceed to $\mathsf{Finalize}$, and all honest parties will set their status to accept. While honest parties might abort (meaning strong robustness cannot be met), honest parties will consistently either end in an aborted or accepting status, meaning the weak robustness game can be fulfilled. The only remaining opportunity for $\mathcal{A}$ to win the weak robustness game is to disrupt consistency of pk, qual, aux, or to make $\mathsf{D}[\mathsf{H}].\mathsf{Recover}$ output $\perp$.

Again, because we assume a broadcast channel, all parties will maintain the same view of $\{(\beta_j, \mathsf{E}_j, \mathsf{pk}_j, D_j)\}$. And so all parties will determine the set qual in $\mathsf{Finalize}$ using the same information. Hence, the adversary has no advantage in forcing a disjoint qual.

From Game 1, we know that all honest parties derive the same view of $\{\psi_j\}$ such that $\mathsf{NK}.\mathsf{SharedKey}(\beta_j, \mathsf{pk}_j) \to \psi_j$. Because the tweak value v is derived from H_2 using the values

$\{\psi_j\}_{j\in[n]}$, then all parties will obtain the same v. Because pk is derived using v and the set $\{D_j\}_{j\in[n]}$, then the adversary has no advantage in forcing a disjoint pk. Similarly, the adversary has no advantage in forcing a disjoint $\mathsf{aux} = \{\mathsf{pk}_i\}_{i\in\mathsf{qual}}$, which are also derived from v and the set $\{D_j\}_{j\in[n]}$.

Finally, from Game 2, we know that $\mathsf{D}[\mathsf{H}].\mathsf{Recover}$ will output a sk that is valid with respect to pk.

The adversary in Game 2 then has no advantage in winning the DKG weak robustness game. Concretely, combining the advantages in Equations 4.11 and 4.14 gives Equation 4.9. This completes the proof. □

4.5.3.2 Zero-Knowledge

Theorem 8. *The generic construction is zero-knowledge in the honest majority setting ($t-1 < n/2$) in the random oracle model, against a PPT adversary $\mathcal{A}$ controlling up to $t-1$ participants, assuming the generic construction itself is weakly robust, the AgVSS A achieves aggregated secrecy, and the NIKE NK is unrecoverable.*

Concretely, for every adversary $\mathcal{A}$ that attacks the construction, there exist adversaries $(\mathcal{B}_1, \mathcal{B}_3, \mathcal{B}_4)$ that run in about the same time as $\mathcal{A}$, and algorithms $\mathsf{SimRound}_i, i \in \{0,\ldots,2\}, \mathsf{SimFinalize}$, such that

$$\begin{aligned}\mathsf{Adv}^{\mathrm{zk}}_{\mathsf{D},\mathcal{A},\mathsf{SimRound}_i,\mathsf{SimFinalize}}(\lambda) \leq\ & \mathsf{Adv}^{\mathrm{sec}}_{\mathsf{A}[\mathsf{H}_1],\mathcal{B}_1}(\lambda) \ + \mathsf{Adv}^{\mathrm{rec}}_{\mathsf{NK},\mathcal{B}_3}(\lambda)\cdot q_h \\ & + \mathsf{Adv}^{\mathrm{asec}}_{\mathsf{A}[\mathsf{H}_1],\mathcal{B}_4,\mathsf{SimShare},\mathsf{SimTweak}}(\lambda)\end{aligned} \tag{4.15}$$

where $\mathsf{SimShare}, \mathsf{SimTweak}$ are as defined for aggregated secrecy of the AgVSS, as in Definition 4.3.2, and q_h denotes the number of queries $\mathcal{A}$ is allowed to make to H_1.

Proof. We now complete the proof of Theorem 8 via a sequence of games.

Game 0. This is the DKG zero-knowledge game when $b = 0$ as in Figure 4.6, applied to the construction in Figure 4.8.

Let W_0 be the event that $\mathcal{A}$ outputs 1 in Game 0. Then,

$$\mathsf{Adv}^{\mathrm{zk}}_{\mathsf{D},\mathcal{A},\mathsf{SimRound}_i,\mathsf{SimFinalize}}(\lambda) = \Pr[W_0]$$

Let $\mathcal{C}$ be the experiment that implements the DKG zero-knowledge experiment to $\mathcal{A}$. We now describe in further detail how $\mathcal{C}$ simulates the game.

Setup. To begin, $\mathcal{C}$ initializes $Q_1 \leftarrow \emptyset$ to simulate H_1 and $Q_2 \leftarrow \emptyset$ to simulate H_2, in addition to initializing each honest party as shown in Figure 4.6. $\mathcal{C}$ handles $\mathcal{A}$'s random oracle queries by lazy sampling, as follows:

$\underline{\mathsf{H}_1}$: When the adversary queries H_1 on inputs (O_i, aux_i), $\mathcal{C}$ checks if it is in Q_1, and if so, returns the corresponding v_i. Otherwise, $\mathcal{C}$ randomly samples v_i from its respective domain, sets $Q_1[(O_i, \mathsf{aux}_i)] = v_i$, and then returns v_i.

$\underline{\mathsf{H}_2}$: When the adversary queries H_2 on inputs $\{\psi_1, \ldots, \psi_n\}$, $\mathcal{C}$ checks if it is in Q_2, and if so, returns the corresponding aux_i. Otherwise, $\mathcal{C}$ randomly samples aux_i from its respective domain, sets $Q_2[\{\psi_1, \ldots, \psi_n\}] = \mathsf{aux}_i$, and then returns aux_i.

$\mathcal{C}$ performs all signing oracle queries honestly.

Game 1. The only difference between Game 0 and Game 1 is that in Game 1, $\mathcal{C}$ performs $\mathsf{SimRound}_0, \ldots, \mathsf{SimRound}_{\kappa-1}, \mathsf{SimFinalize}$ for participant $\tau \xleftarrow{\$} \mathsf{honest}$ with respect to the challenge $\hat{\Delta}$. More specifically, when performing $\mathsf{SimRound}_0$, $\mathcal{C}$ employs $(\{\phi^*_{\tau,j}\}_{j \in \mathsf{corrupt}}, D^*_\tau) \leftarrow \mathsf{A}[\mathsf{H}_1].\mathsf{SimShare}(\lambda, n, t, \tau, \hat{\Delta})$ for an honest participant τ sampled from the set of honest participants at random, such that Equation 4.16 holds:

$$\mathsf{S.GetPub}(0, D^*_\tau) = \hat{\Delta} \tag{4.16}$$

$\mathcal{C}$ performs $\mathsf{SimRound}_1, \mathsf{SimRound}_2, \mathsf{SimFinalize}$ as in the real protocol. However, one consequence of $\mathcal{C}$ simulating secret sharing with respect to $\hat{\Delta}$ is that $\mathcal{C}$ cannot correctly derive each honest players' sk_i, for all $i \in \mathsf{honest}$ when performing $\mathsf{SimFinalize}$. This however does not impact the ability of $\mathcal{C}$ to simulate all honest players, in part because $\mathcal{C}$ never reveals $\{\mathsf{sk}_i\}_{i \in \mathsf{honest}}$ directly.

If the AgVSS is secret, then Game 0 and Game 1 are indistinguishable to $\mathcal{A}$.

Reduction to AgVSS (Non-Aggregated) Secrecy. We will construct a reduction $\mathcal{B}_1$ which is an adversary against the AgVSS (non-aggregated) secrecy game as defined in Definition 2.5.5.

$\mathcal{B}_1$ is constructed similar to Game 0, with the following exception. $\mathcal{B}_1$ simply performs $\mathsf{PerformRound}_1$ for each honest non-simulated party $i \in \mathsf{honest}, i \neq \tau$. However, for the simulated honest party τ, instead of following the protocol honestly, $\mathcal{B}_1$ instead employs the challenge received as input $(\{(j, \phi_j)\}_{j \in \mathsf{corrupt}}, D)$ from the AgVSS secrecy experiment. In particular, it sets $D_\tau \leftarrow D$, and $(\{(j, \phi_{\tau,j})\}_{j \in \mathsf{corrupt}} \leftarrow (\{(j, \phi_j)\}_{j \in \mathsf{corrupt}}$. It then follows

the remainder of the protocol honestly. While it will not be able to derive the honest player's secret keys in $\mathsf{PerformRound}_3$, this will not have an impact on its ability to correctly simulate, as corrupt parties will receive valid shares (and so can derive their secret keys correctly), and honest parties' shares (and secret keys) are never revealed.

The simulation of Game 0 by $\mathcal{B}_1$ is identical to when $b = 0$ in the AgVSS secrecy game. Hence, the only distinction between Game 0 and Game 1 is when $b = 1$ in the AgVSS secrecy game.

When $\mathcal{A}$ outputs b' as its guess, then $\mathcal{B}_1$ outputs b'. Hence, when $\mathcal{A}$ distinguishes Game 0 from Game 1, $\mathcal{B}_1$ wins the AgVSS secrecy game.

Difference between Game 0 and Game 1. If the AgVSS is secret, then the difference between Game 0 and Game 1 is negligible. Let W_1 be the event that $\mathcal{A}$ wins in Game 1. Then

$$\left|\Pr[W_1] - \Pr[W_0]\right| \leq \mathsf{Adv}^{\mathrm{sec}}_{\mathsf{A},\mathcal{B}_1}(\lambda) \tag{4.17}$$

Game 2. The difference between Game 1 and Game 2 is code movement from $\mathsf{Finalize}$ to $\mathsf{PerformRound}_1$, as shown in Figure 4.9, which shows code movement. In particular, the code that is moved is highlighted, moved from $\mathsf{Finalize}$ to $\mathsf{PerformRound}_1$.

Rather than learning corrupted players' blinding factors as input to $\mathsf{Finalize}$, $\mathcal{C}$ instead derives blinding factors for all corrupted parties in $\mathsf{PerformRound}_1$, (importantly, before it reveals the honest players' blinding factors to $\mathcal{A}$) using the shares from all honest players. If $\mathcal{C}$ is unable to recover these blinding factors, then it returns $\perp$.

However, because $\mathcal{C}$ simulates at minimum t honest players (honest majority model), it can perform $\mathsf{A.Recover}$ without the involvement of corrupted parties.

Difference between Game 1 and Game 2. Let W_2 be the event that $\mathcal{A}$ wins in Game 2. Then,

$$\Pr[W_2] = \Pr[W_1] \tag{4.18}$$

Game 3. The only difference between Game 3 and Game 2 is that if the adversary queries the random oracle H_1 with respect to any of the honest parties' blinding values $\{\psi_i\}_{i\in\mathsf{honest}}$ before being provided with the witness values $\{\beta_i\}_{i\in\mathsf{honest}}$ as input to $\mathsf{PerformRound}_3$, then the game aborts.

If the NIKE is session-key unrecoverable, the additional advantage to $\mathcal{A}$ is negligible.

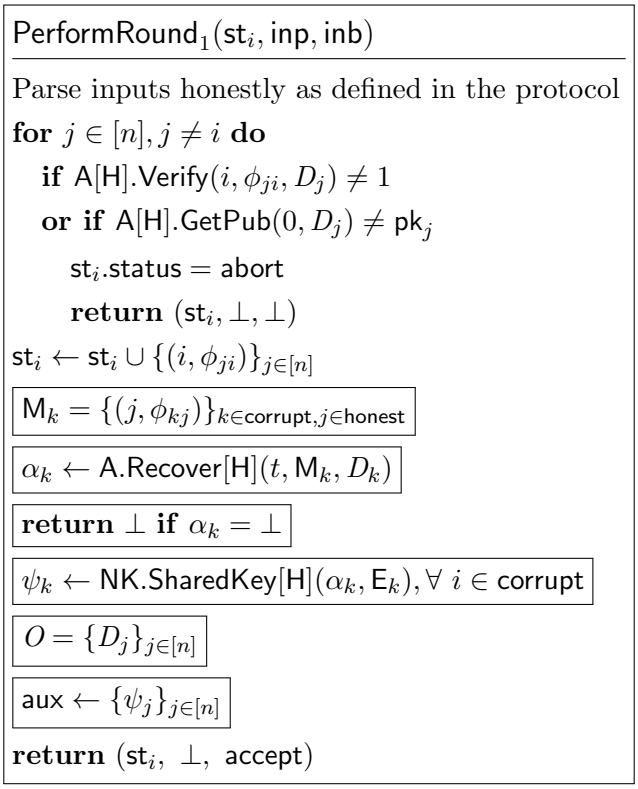


Figure 4.9: Game 2 for the Zero Knowledge Proof for Generic Construction. The boxes indicate steps that are performed in addition to Game 1.

Reduction to NIKE Unrecoverability. We will construct a reduction $\mathcal{B}_3$ which is an adversary against the NIKE unrecoverability game, such that $\mathcal{B}_3$ simulates Game 3 to $\mathcal{A}$.

$\mathcal{B}_3$ begins by receiving $(\mathsf{pk}_1, \mathsf{pk}_2)$ in the NIKE unrecoverability game. It then simulates Game 3 to $\mathcal{A}$ in exactly the same manner as in Game 2, with the following exception. $\mathcal{B}_3$ simply performs $\mathsf{PerformRound}_0$ honestly for each honest party $i \in \mathsf{honest}, i \neq \tau$. However, for honest party τ, $\mathcal{B}_3$ instead simulates the protocol via $\mathsf{SimRound}$; it sets $\mathsf{pk}_\tau \leftarrow \mathsf{pk}_1$ and $\mathsf{E}_\tau \leftarrow \mathsf{pk}_2$ $\mathcal{B}_3$ then simulates secret sharing via $\mathsf{SimShare}$ with respect to pk_τ. From Game 1, we know that doing so is indistinguishable to $\mathcal{A}$. $\mathcal{B}_3$ then follows the remainder of $\mathsf{PerformRound}_1$ as for all the other honest parties.

When simulating $\mathcal{O}^{\mathsf{PerformRound}_3}$, $\mathcal{B}_3$ acts honestly for all honest parties $i \in \mathsf{honest}, i \neq \tau$. However, for participant τ, $\mathcal{B}_4$ does the following. Because it does not know the corresponding β_c to E_c, it instead must guess. It looks at all of the queries made to H_1 that $\mathcal{A}$ has made up to that point, and randomly selects one of the inputs. Let this randomly selected input be $(\{\psi_j\}^*_{j\in[n]}, \mathsf{aux}^*)$.

$\mathcal{B}_3$ then picks the simulated party's element $\psi_\tau \leftarrow \{\psi_j\}^*_{j\in[n]}$, and outputs $\psi' = \psi_\tau$ as its guess for the NIKE unrecoverability game. If $\mathcal{A}$ queries H_1 with ψ_τ before $\mathcal{B}_3$ has completed $\mathcal{O}^{\mathsf{PerformRound}_2}$, then $\mathcal{B}_3$ wins the NIKE game with probability $\frac{1}{q_h}$.

Difference between Game 2 and Game 3. Let W_3 be the event that $\mathcal{A}$ wins in Game 3. The additional ability for $\mathcal{A}$ to win Game 3 is then bounded by its ability to win the NIKE unrecoverability game. Hence,

$$\left|\Pr[W_3] - \Pr[W_2]\right| \leq \mathsf{Adv}^{\mathrm{rec}}_{\mathsf{NK},\mathcal{B}_3}(\lambda) \cdot q_h \tag{4.19}$$

Game 4. The only difference between Game 3 and Game 4 is that in Game 4, $\mathcal{C}$ simulates secret sharing for participant $\tau \xleftarrow{\$} \mathsf{honest}$ with respect to the challenge $\hat{\Delta}$. From Game 1, we know that $\mathcal{C}$ can do so in such a way that is indistinguishable to $\mathcal{A}$. However, in Game 4, it additionally does so in such a way that Equation 4.20 holds after performing $\mathsf{Finalize}$.

$$C[\mathsf{H}_1].\mathsf{GetPub}(0, C) = \hat{\Delta} \tag{4.20}$$

More specifically, in $\mathcal{O}^{\mathsf{PerformRound}_0}$, $\mathcal{C}$ performs

$$(\{(j, \phi_{\tau j})\}_{j\in\mathsf{corrupt}}, D_\tau) \xleftarrow{\$} \mathsf{A}[\mathsf{H}_1].\mathsf{SimShare}(\lambda, \mathsf{corrupt}, \hat{\Delta})\,.$$

It then follows the remainder of $\mathsf{PerformRound}_0$ as in Game 3.

Then, in $\mathcal{O}^{\mathsf{PerformRound}_3}$, it derives v' as $v' \xleftarrow{\$} \mathsf{A}[\mathsf{H}_1].\mathsf{SimTweak}(O, \mathsf{st}_\tau)$. It then programs $\mathsf{H}_2(O, \mathsf{aux})$ with v'. From Game 3, we know that $\mathcal{C}$ can derive aux in $\mathcal{O}^{\mathsf{PerformRound}_3}$, but $\mathcal{A}$ has only negligible probability of doing so.

We will see that, if the AgVSS is securely aggregatable, then the additional advantage to the adversary in Game 4 is negligible.

Reduction to AgVSS Aggregated Secrecy. We will construct a reduction $\mathcal{B}_4$ which is an adversary against the AgVSS aggregated secrecy game, such that $\mathcal{B}_4$ simulates either Game 3 or Game 4 to $\mathcal{A}$. When the hidden bit b in the AgVSS aggregated secrecy game is $b = 0$, $\mathcal{B}_4$ ends up simulating Game 3; when $b = 1$, $\mathcal{B}_4$ ends up simulating Game 4.

$\mathcal{B}_4$ is constructed similar to Game 3, except as follows. When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{PerformRound}_1}$, for each honest player $k \in \mathsf{honest}$: $\mathcal{B}_4$ queries the AgVSS oracle $\mathcal{O}^{\mathsf{Share}}()$, receiving $(\{\phi_j\}_{j\in\mathsf{honest}}, D^*)$. $\mathcal{B}_4$ then uses these outputs to simulate participant τ for $\mathcal{O}^{\mathsf{PerformRound}_1}$.

$\mathcal{B}_4$ follows the protocol honestly for all other players, and submits all $(\{\phi_{kj}\}_{j\in\mathsf{honest}}, D_k)$ for $k \in [n] \setminus \tau$ to the AgVSS oracle $\mathcal{O}^{\mathsf{RecvShare}}$.

Then, $\mathcal{B}_4$ receives aux', and then queries the AgVSS aggregated secrecy oracle $\mathcal{O}^{\mathsf{Tweak}}$ on inputs (O, aux'), receiving v' in return. $\mathcal{B}_4$ then programs its own $\mathsf{H}_1(O, \mathsf{aux}) = v'$. From Game 3, $\mathcal{A}$ can query for its output with negligible probability before $\mathcal{B}_4$ programs it. $\mathcal{B}_4$ then follows the remainder of Game 3 as before.

When $\mathcal{A}$ outputs b', then $\mathcal{B}_4$ outputs b'. Hence, when $\mathcal{A}$ distinguishes Game 3 from Game 4, $\mathcal{B}_4$ wins the AgVSS aggregated secrecy game.

Difference between Game 3 and Game 4. Let W_4 be the event that $\mathcal{A}$ wins in Game 4. The difference between W_3 and W_4 is bounded by the advantage of $\mathcal{A}$ in the AgVSS aggregated security game.

$$\left|\Pr[W_4] - \Pr[W_3]\right| \leq \mathsf{Adv}^{\mathrm{asec}}_{\mathsf{A}[\mathsf{H}_1],\mathcal{B}_4,\mathsf{SimShare},\mathsf{SimTweak}}(\lambda) \tag{4.21}$$

Finishing the Proof. The resulting public key in Game 4 is identical to the challenge $\hat{\Delta}$, and hence is identical to the DKG zero-knowledge game when $b = 1$. The adversary then has at most negligible advantage in winning the DKG zero-knowledge game. Concretely, combining the advantage of the adversary in (4.17)-(4.21) proves Equation 4.15. This concludes the proof. □

4.6 Storm, a Discrete-Logarithm Based DKG

Finally, we instantiate our generic DKG presented in Section 4.5 with a concrete construction, Storm. Due to the proof of security for the generic construction, Storm is secure under the Computational Diffie-Hellman (CDH) problem in the random oracle model, in the honest majority setting.

Target Key Generation The target key generation scheme TK for Storm is defined as follows. The secret domain $\hat{S}$ is the field $\mathbb{Z}_q$ and the public domain $\hat{P}$ is the group $\mathbb{G}$ generated by g.

- $\mathsf{KeyGen}(\lambda) \to (\mathsf{sk}, \mathsf{pk})$: Sample $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$. Generate $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$. Output $(\mathsf{sk}, \mathsf{pk})$.
- $\mathsf{Verify}(\mathsf{sk}, \mathsf{pk}) \to \{0, 1\}$: Output 1 if $g^{\mathsf{sk}} = \mathsf{pk}$. Otherwise, output 0.

Building Blocks. We next describe the concrete building blocks required for Storm:

1. the NIKE scheme presented in Section 2.4.1,
2. the DL-based AgVSS scheme presented in Section 4.3.1, and
3. a secure hash function H, where H_1, and H_2 are domain-separated instances of H. These functions are defined as follows:
   - $\mathsf{H}_1 : (\mathbb{G}^t)^n \times \mathbb{Z}_q \to \mathbb{Z}_q$: Accepts as input a vector, where the first element is a tuple of n AgVSS commitments $D_j \in \mathbb{G}^t, j \in [n]$ which are each a tuple of group elements of size t, and the second element is an auxiliary element $\mathsf{aux} \in \mathbb{Z}_q$.
   - $\mathsf{H}_2 : (\mathbb{G})^n \to \mathbb{Z}_q$: Accepts as input a vector of n blinding factors $\psi_j \in \mathbb{G}, j \in [n]$, and outputs an auxiliary element $\mathsf{aux} \in \mathbb{Z}_q$.

Correctness and Security. Storm is an instantiation of the generic construction presented in Section 4.5, such that Assumption 1 holds for the DL-based AgVSS scheme presented in Section 2.4.1 and the DL-based AgVSS scheme presented in Section 4.3.1. For this reason, the correctness and security of Storm follows from the proofs for the generic scheme.

For security, by Theorem 7 Storm is weakly robust, assuming the AgVSS is zero-indexed and secure, and the NIKE is binding. We establish the security of the zero-indexed DL-based AgVSS scheme in Section 4.3.1.1, and Theorem 2 establishes that the concrete NIKE of Section 2.4.1 is binding.

By Theorem 8, Storm is zero-knowledge, assuming the AgVSS achieves aggregated secrecy and the NIKE is unrecoverable. We establish that the zero-indexed DL-based AgVSS scheme indeed achieves aggregated secrecy in Section 4.3.1.1. The unrecoverability of the concrete NIKE of Section 2.4.1 is a direct reduction to CDH.

4.7 Future Work and Composability

Because we prove the generic construction secure with respect to our game-based notions, we do not inherently prove that the generic construction is secure with respect to any other protocol; i.e., we do not prove security under composability. One approach to demonstrating composability is using the Universal Composability (UC) framework [Can01]. However, a second approach would be to prove that a DKG that is secure under these game-based notions is likewise secure under a simulation-based notion. Such an approach was taken by Boneh et al. [BHLS25] to show equivalence between their game-based notions of security for an Exponent Verifiable Random Function (eVRF), and a simulation-based notion of security.

To do so, we first must determine which notion of simulation-based security our game-based notions capture. Katz [Kat24] gives several notions of simulation-based security for DKGs, defining full security as a DKG whose output keys are indistinguishable from keys from the target single-party key generation algorithm. He defines security with abort as a DKG that achieves full security with the exception of an adversary that can force honest participants to abort (and hence introduce bias).

We postulate that it is possible to prove a DKG that is weakly robust, zero-knowledge, and indistinguishable under the simulation-based notion of security with aborts. Note that if the weakly robust DKG additionally implemented identifiable aborts and re-ran the protocol, it may be possible for the DKG to achieve full security, as the adversary would be able to terminate the protocol early but not learn any information about the public key before doing so. Furthermore, we postulate that it is possible to prove a DKG that is strongly robust, zero-knowledge, and indistinguishable under the simulation-based notion of full security.

Chapter 5

Arctic: Lightweight and Stateless Threshold Schnorr Signatures

In this chapter, we focus our attention on deterministic threshold Schnorr signatures; i.e., a deterministic threshold signature scheme that is compatible with the (single-party) Schnorr signature verification algorithm [Sch91]. In this chapter, we move to the setting where the minimum number of signers μ is generalized to some $t \leq \mu \leq n$, while still preserving security when up to $t-1$ signers may be corrupted.

As we saw in Chapter 3, efficient (two-round) threshold Schnorr signatures exist in the randomized setting [KG20, BCK+22, RRJ+22], where each party is assumed to securely maintain state between signing rounds and have access to good sources of randomness. However, efficient and *deterministic* threshold Schnorr signatures has thus far remained an elusive goal.

Why Deterministic Threshold Signatures? Producing threshold signatures in a deterministic manner is useful for two reasons. First, it is useful as a general defense-in-depth measure, to protect against the event of temporarily losing access to good sources of randomness [RY10], such as if a machine randomly rebooted.

Second, threshold Schnorr signature schemes generally require participants to perform multiple rounds of communication before a joint signature can be issued. As such, participants must keep state between each round, and carefully delete state once a signing protocol is completed. In the setting where the signer must produce signatures at increasing scale and in a concurrent setting, managing state can become a significant performance bottleneck. Further, secure management of secret state can be considered a security risk.

In particular, for threshold Schnorr signatures, participants must generate secret nonces for each signing session. If a participant's state is mismanaged in such a way that it is used even twice, a fatal key-recovery attack is possible. However, in the setting where machines can go offline at any time, and signing is done at scale and concurrently, such careful management of secret state becomes even more challenging.

The Challenge of Efficient Deterministic Multi-Party Schnorr Signatures. While the goal of efficient and deterministic multi-party Schnorr signature schemes is desirable, producing such schemes has proven difficult. The challenge can be summarized as follows. While honest participants can certainly pick their nonces deterministically (say, by hashing their secret signing share and the message), a malicious party might deviate from the protocol and pick its nonce at random. Recall that for Schnorr signatures, the output signature $\sigma = (R, z)$ is a commitment R and response z, satisfying the relation $g^z = R \cdot \mathsf{pk}^c$ for a challenge $c \leftarrow H(R, \mathsf{pk}, m)$. In the setting for threshold Schnorr signatures, where (R, z) are contributed to by all signing parties, one party deviating from the protocol results in the challenge c likewise changing.

If honest parties cannot verify that other parties followed the protocol, such a deviation would allow an adversary to perform a key recovery attack with as little as two signing queries. A key-recovery attack is possible is because the adversary can force honest parties to re-use their nonces, while corrupted parties introduce fresh randomness by picking non-deterministic nonces. Doing so would lead to the group commitment R changing, which in turn leads to the challenge c changing. An honest participant's secret key share could then be obtained simply by solving the linear equation $\mathsf{sk}_i = \frac{z_1 - z_2}{c_1 - c_2}$, where z_1, z_2 are signature shares from two different signing sessions, and c_1, c_2 are the corresponding challenges.

Prior deterministic multi-party Schnorr signature schemes in the literature [NRSW20, GKMN21, KOR23] can be broken down into two general approaches. We give an overview of the approaches that support the general threshold setting with general (t, μ, n) in Table 5.1, with more context now.

The first approach requires that signers output zero-knowledge proofs that a Pseudorandom Function (PRF) was evaluated correctly [NRSW20, GKMN21]. However, this approach adds undesirable performance and complexity overhead. For example, while MuSig-DN [NRSW20] requires only two network rounds, proof generation requires approximately 1 second (regardless of the number of signers) for 256-bit security, and verification requires at least $10n$ ms. Similarly, the scheme by Garillot et al. [GKMN21] requires three rounds of communication between signers and high complexity and bandwidth overhead.

The second approach by Kondi et al. [KOR23] is to use as a black box a Pseudorandom Correlation Generator (PCG) [BCG+20] to generate nonces in a verifiable manner [KOR23]. However, while the PCG used in their construction can support 2-of-2 signing, [OSY21, KOR23], it remains an open problem as to how to extend such a PCG to the general threshold setting for any t, μ, or n [Kon24]. As such, the scheme by Kondi et al. [KOR23] supports only the $t = \mu = n = 2$ setting.

Therefore in this chapter, we address the following question:

> *Can we design a simple, deterministic, and stateless two-round threshold Schnorr signature scheme, with similar efficiency to existing randomized schemes and acceptable security assumptions in practice?*

5.1 Our Results

We answer the above question in the affirmative. In particular, we present Arctic, a deterministic and stateless threshold Schnorr signature scheme. For moderately sized groups (i.e., when $n \leq 20$), Arctic is more than an order of magnitude more efficient than existing deterministic schemes in the literature; it is three orders of magnitude more efficient when $n \leq 10$. For even larger groups, Arctic scales linearly relative to the number of processing cores available to most machines. Signers in Arctic are required to generate and maintain secret keys, but otherwise do not need to manage secret state.

To achieve efficient determinism, Arctic requires three tradeoffs. The first tradeoff is in the number of required signers; Arctic requires $\mu \geq 2t - 1$ parties to participate in signing. This requirement is because Arctic assumes the honest majority model. We discuss in Section 5.2 why the assumption of an honest majority is acceptable for some real-world applications.

The second tradeoff is in the required overhead as the signing set grows large. Under the hood, Arctic employs a replicated secret sharing scheme [ISN89], and so requires participants to store a secret key of size $\binom{n-1}{t-1}$. As such, Arctic outperforms other schemes in the literature for moderately sized groups, but incurs a crossover point as the signing set grows large.

The final tradeoff is that Arctic as defined does not support identifiable abort. While the protocol allows honest participants to identify that misbehavior has occurred, the protocol does not support identifying which participant caused the abort (which is possible for MuSig-DN [NRSW20], for example). However, we give a discussion on how Arctic can be

extended to support identifiable abort, and indeed robustness, in the *honest supermajority* setting (i.e., when $\mu \geq 3t - 2$).

Verifiable Pseudorandom Secret Sharing (VPSS). As a building block, we first formalize a cryptographic primitive that we call a *Verifiable Pseudorandom Secret Sharing (VPSS)* scheme. While pseudorandom secret sharing is a standard notion in the literature, and verifiability of such schemes had been employed implicitly in maliciously secure MPC-based schemes [BBG+21], we present a formal definition for a VPSS and give game-based security notions [BR06], building on existing notions of verifiable random functions in the literature [MRV99, NPR99, Dod03, GLOW21].

Intuitively, a pseudorandom secret sharing scheme allows a coalition of players holding pre-established secret shares of a random secret to generate shares of additional pseudorandom secrets. A *verifiable pseudorandom secret sharing scheme* is simply an extension to allow for public verifiability by collectively verifying the outputs for the set of participants. As such, misbehavior of players in a VPSS can be detected assuming some threshold of honest participants. As we will see in the case of Arctic, a VPSS allows for more efficiently verifying the correctness of the output of a pseudorandom function that is distributed among a set of players, without requiring each player individually to produce a zero-knowledge proof that it followed the protocol.

We then define a concrete VPSS construction that we call Shine that builds upon the pseudorandom secret sharing scheme by Cramer et al. [CDI05]. We augment the scheme by Cramer et al. to additionally define a verification algorithm that is publicly verifiable even when players only publish commitments to their shares; i.e., it preserves secrecy of players' shares while allowing players to verify that all other players followed the protocol honestly. Shine is efficient for moderately sized groups; concretely, evaluation requires $\binom{n-1}{t-1}$ sub-microsecond hash and field operations, and verification requires $2t^2 - t$ group exponentiations, where t is the corruption threshold. We prove that Shine is secure assuming the existence of a secure hash function in the honest majority model, i,e., when $\mu \geq 2t - 1$, where μ is the minimum number of participants required to participate in the protocol. Additionally, we require that as n grows, t remains $\mathcal{O}(1)$ so that $\binom{n-1}{t-1} = \text{poly}(n)$.

A New Two-Round, Deterministic, and Stateless Threshold Schnorr Signature. Next, we introduce Arctic, a new two-round, deterministic, and stateless threshold Schnorr signature. Arctic is an order of magnitude more efficient than related schemes in the literature for moderately sized groups ($n \leq 20$), due to its use of Shine as a building block.

	Computation	*Bandwidth*	*Num. Rounds*	*Assumptions*	*Min. Signers*
MuSig-DN [NRSW20]	$14210 + 24(n-1)$ group	1189 bytes	2	DDH	n
GKMN21 [GKMN21]	$132,000(t-1)$ AES $256(t-1)$ field	$1.01(t-1)$ MB	3	PRF	t
Arctic (this work)	$2t^2 - t + 2$ group $2\binom{n-1}{t-1}$ field, $2\binom{n-1}{t-1}$ hash	97 bytes	2	DL	$2t-1$

Table 5.1: Comparison of multi-party deterministic Schnorr signature schemes in the ROM that support general choices of n.

Furthermore, Arctic supports the general threshold setting for any t, μ and n, so long as $\mu \geq 2t - 1$ and $\mu \leq n$.

At a high level, Arctic uses Shine in the first round of signing to generate participant nonces and commitments; participants publish their commitments without having to maintain state of their (secret) nonce. Then, in the second round, participants use Shine to re-derive their nonce and commitment, and verify all other participants' commitments, ensuring that other participants followed the protocol correctly. If the verification check holds, participants derive the group commitment as the aggregation of all participants' individual commitments, and generate a signature share as a combination of their nonce, challenge, and secret signing share. The output signature is the aggregated group commitment and an aggregation of all participants' signature shares, and can be verified using the single-party Schnorr verification algorithm.,

We prove the unforgeability of Arctic assuming the hardness of the discrete logarithm problem in the random oracle model, assuming $\mu \geq 2t - 1$, the adversary corrupts at most $t-1$ players, and $\binom{n-1}{t-1} = \text{poly}(n)$. We describe in Section 5.2 why requiring honest majority assumptions can be practical for some real-world deployments of threshold signatures, at the benefit of improved simplicity and performance.

Comparison with Related Schemes. We give an overview of Arctic in Table 5.1 in comparison to related schemes in the literature. In the table, we review only schemes that support a general number n of parties (i.e., we exclude 2-of-2 schemes). Computation is given with respect to the operations that dominate; estimates for MuSig-DN are given with

respect to a 256-bit elliptic curve. We give further information is given in Section 5.8.1. Bandwidth similarly is given with respect to a 256-bit elliptic curve [NRSW20, GKMN21].

Performance Analysis of Arctic. To estimate the practicality of Arctic for various choices of n, t, and μ, we implement the scheme and provide concrete benchmarks.

Arctic is highly efficient for moderately sized groups, and so is a good choice for such settings. As we show in greater detail in Section 5.8, for a group where $2t-1 \leq \mu \leq n \leq 20$, Arctic signing operations require less than 100 milliseconds of computation for each signer, and for $n \leq 10$, less than 1 millisecond. Signing however increases in cost relative to the corruption threshold; for signing sets of size $n = 25, t = 11$, the computational overhead for each signer on a single core requires one second. However, Arctic is highly parallelizable, showing almost perfect linear speedups for up to 32 cores for that size of signing set.

For comparison, MuSig-DN requires about 1 second in computational overhead per signer [NRSW20], regardless of the size of the signing set. As such, Arctic is a practical choice for moderately sized groups of $n \leq 25$, whereas MuSig-DN or GKMN21 [GKMN21] may be good choices as the set of signers grows large, depending on the parallelization of each scheme.

Our Contributions. In summary, the contributions in this chapter are as follows:

- We formalize the definition of a Verifiable Pseudorandom Secret Sharing (VPSS) scheme, and give game-based security notions.
- We define a concrete VPSS scheme that we call Shine, which extends the pseudorandom scheme by Cramer et al. [CDI05] to allow for public verifiability, assuming an honest majority of participants.
- We then present Arctic, a deterministic and stateless two-round threshold Schnorr signature scheme. Arctic uses Shine as a building block to generate participant nonces and commitments, and verify that participants performed this action correctly.
- We prove that Arctic is secure under the discrete logarithm problem in the random oracle model, assuming Shine is a secure VPSS, the adversary corrupts fewer than t parties, at least $\mu \geq 2t - 1$ parties participate in the signing protocol, and $\binom{n-1}{t-1} = \text{poly}(n)$.
- We provide performance benchmarks for Arctic for different sizes of signing sets, and show that parallelization enables a linear speedup in signer computation.

5.2 Observations of Honest Majority Assumptions in Practice

While honest minority assumptions may be desirable for some real-world applications, we observe that honest majority assumptions may be an acceptable tradeoff for other applications in exchange for statelessness, improved performance, and protocol simplicity.

For applications that can easily support additional signers, moving from an honest minority setting to honest majority can be relatively straightforward. For example, an application that currently uses a $(t = 2, \mu = 2, n = 3)$ configuration can instead move to a $(t = 2, \mu = 3, n = 4)$ configuration.

In settings that require liveness, honest majority requirements are already assumed, to ensure usability of the shared secret key even when corrupt players refuse to participate. For example, applications such as cryptocurrency wallets often implicitly require honest majority assumptions, to ensure that corrupt players cannot prevent use of funds by simply refusing to participate in signing operations.

Finally, some applications may see the requirement for additional signers as an acceptable tradeoff to mitigate the security risk of protocol complexity, as protocol complexity increases the risk of exploitable implementation errors [MYG24].

5.3 Related Work

In this section, we review prior deterministic threshold signature schemes in the literature. We review randomized threshold Schnorr signature schemes in Section 3.1.

The BLS signature scheme [BLS04] is deterministic, because the signature is of the form $z \leftarrow H(m)^{\mathsf{sk}}$. Likewise, threshold BLS signatures are similarly deterministic [Bol03]. However, such schemes are often not viable in a practical setting that requires backwards compatibility with Schnorr signature verification, or due to the requirement of pairings.

Deterministic threshold Schnorr signatures have been described in the literature, but rely on heavyweight zero-knowledge proofs to demonstrate that each participant followed the protocol correctly. The challenge of deterministic threshold Schnorr signatures is that of *verifiability*, because if even one participant deviates from the protocol and chooses its nonce randomly, a complete key-recovery attack is possible.

Nick et al. [NRSW20] define a threshold Schnorr n-of-n multisignature that uses SNARKs to demonstrate in zero-knowledge that a participant generated its nonce cor-

rectly using a PRF with respect to a pre-established keypair. However, the authors report the computational overhead of at least 943 ms for a single execution, independent of the number of signers, due to the overhead of proving the PRF was evaluated correctly in zero-knowledge. In particular, to instantiate the PRF, a non-algebraic cryptographic hash function $H : \{0,1\} \to \mathbb{G}$ must be used, along with a regular function $f : \mathbb{G} \to \mathbb{Z}_q$. The complete PRF F_{sk} is defined by $F_{\mathsf{sk}}(x) = f(\mathsf{sk} \cdot H(x))$, where sk is a PRF key. Then, signers generate a Bulletproof [BBB+18] to prove in zero-knowledge that the nonce was derived with respect to F, sk, and some input $H(x)$.

Bronte et al. [BST21] define an MPC protocol to partition the functionality of EdDSA, which itself is deterministic. However, their MPC protocol is randomized and requires multiple rounds of interaction, and therefore is stateful.

Garillot et al. [GKMN21] similarly rely on parties sampling and committing to a PRF key at the time of key generation, but instead make use of the Zero-Knowledge from Garbled Circuits (ZKGC) paradigm [JKO13] to demonstrate that the PRF was derived correctly. The specific function that is garbled is $C(x) = \phi(F(x))$, where F is a boolean circuit such as AES, and ϕ is standard exponentiation (or curve multiplication). The authors give efficiency estimates for a 256-bit curve and using SHA-512 as the PRF F; the performance overhead is dominated by performing 132,000 AES invocations and 256 additions in $\mathbb{Z}_q$ per proof generated and verified. For a signing invocation involving t parties, each party must then perform $256t$ field operations and $132{,}000t$ AES invocations, accounting for each proof a signer generates and the $t - 1$ proof verifications for all other signers.

Kondi, Orlandi, and Roy [KOR23] take a different approach, and define a two-round stateless and deterministic two-party Schnorr signature scheme using pseudorandom correlation functions (PCFs) [BCG+20] as a building block. In particular, their scheme employs a Paillier-based PRF [OSY21], but additionally define a verification mechanism to ensure that parties honestly followed the protocol. However, their scheme is restricted to the two-party setting, and as written, cannot be extended to the general (t, n) threshold setting. In particular, the PCF used as a building block by their scheme assumes only two parties. Extending their scheme to any (t, n) setting requires designing a new n-party PCF [Kon24].

Honest Majority Threshold Signatures. Honest majority threshold schemes have been proposed in the literature as a means to achieve properties that are either impossible, or require higher performance overhead, in the honest minority setting. Notably, honest majority schemes have been demonstrated to achieve robustness or to circumvent requiring the use of heavyweight zero-knowledge proofs [CGG+20].

Gennaro et al. [GJKR96] use error correcting codes in the honest majority setting

to achieve a robust threshold DSS scheme. Similarly, Ruffing et al. [RRJ+22] present a wrapper to the FROST threshold signature scheme to achieve robustness, in the honest majority setting.

In the randomized threshold ECDSA setting, Damgård et al. [DJN+22], Doerner et al. [DKLS24], and Delskov [DOK+20] show how the pseudorandom secret sharing scheme by Cramer et al. [CDI05] can be employed as a sub-protocol for improved efficiency. However, our work targets the setting of deterministic threshold Schnorr signatures.

Distributed VRFs. Galindo et al. [GLOW21] give a formalization of distributed VRFs and their security notions. While our notions for a Verifiable Pseudorandom Secret Sharing (VPSS) scheme are similar, our definition for a VPSS does not require that players' outputs are accompanied by a zero-knowledge proof that the protocol was performed correctly. Instead, the VPSS verification function *collectively* verifies all parties' outputs.

Pseudorandom Secret Sharing. Our VPSS construction Shine builds on the pseudorandom secret sharing scheme by Cramer et al. [CDI05]. Note that while Cramer et al. additionally define a Non-Interactive Verifiable Secret-Sharing (NIVSS) variant, their construction assumes that players output shares in the clear, and requires an honest supermajority so that the secret can be recovered. In our case, players verify the correctness of their shares with respect to other players' public commitments, and requires only honest majority (as players simply need to determine if any other party deviated from the protocol).

While we are the first to do so in a threshold Schnorr signature setting, pseudorandom secret sharing has been used as a building block in other other threshold settings. For example, Jarecki, Krawczyk, and Resch [JKR18] define a threshold Oblivious PRF which likewise builds upon pseudorandom secret sharing.

5.4 Deterministic Threshold Signature Schemes

We begin with the definition of a deterministic threshold signature scheme, and then define the notion of unforgeability. We build upon standard definitions and notions of unforgeability for threshold signatures in the literature [CKM23, GJKR01, GRJK07, GG18].

Intuitively, our definition for deterministic threshold signature schemes is identical to that of randomized threshold signature schemes, with the exception that in the deterministic setting, the signing algorithms are deterministic and the signer does not maintain state

between rounds. Furthermore, each signing round in the deterministic setting is given as input the message m, coalition of signers C, and secret signing key share sk_i.

Definition 5.4.1 (Deterministic Threshold Signatures)**.** *A* **deterministic threshold signature scheme** DT *with an interactive signing protocol consisting of κ rounds is a tuple of PPT algorithms $\mathsf{TS} = (\mathsf{Setup}, \mathsf{KeyGen}, (\mathsf{Sign}_1, \ldots, \mathsf{Sign}_\kappa), \mathsf{Combine}, \mathsf{Verify})$, defined as follows:*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{par}$*: A probabilistic algorithm that accepts as input a security parameter λ and outputs public parameters par, which are then implicitly given as input to all other algorithms.*
- $\mathsf{KeyGen}(n, t, \mu) \to (\mathsf{pk}, \{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]})$*: A probabilistic algorithm that takes as input the total number of possible signers n, the corruption threshold t, and the minimum number of participating signers μ. Outputs the public key pk representing the set of n signers, the set $\{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]}$ of public and secret key shares for each signer.*
- $(\mathsf{Sign}_1, \ldots, \mathsf{Sign}_\kappa) \to \{\rho_1^{(k)}, \ldots \rho_\kappa^{(k)}\}_{k\in C}$*: A set of deterministic algorithms where each algorithm represents a single stage in an interactive signing protocol performed by signing party $k \in [n]$ in a signing set $C \subseteq [n], |C| \geq \mu$ with respect to a message m, defined as follows:*
  $\rho_1^{(k)} \leftarrow \mathsf{Sign}_1(k, \mathsf{sk}_k, m, C), \quad \ldots, \quad \rho_\kappa^{(k)} \leftarrow \mathsf{Sign}_\kappa(k, \mathsf{sk}_k, m, C, \{\rho_{\kappa-1}^{(i)}\}_{i\in C})$
  where $\rho_1^{(k)}, \ldots, \rho_\kappa^{(k)}$ are protocol messages produced by party $k \in C$.
- $\mathsf{Combine}(\mathsf{pk}, m, C, \{(\rho_1^{(i)}, \ldots, \rho_\kappa^{(i)})\}_{i\in C}) \to \sigma$*: A deterministic algorithm that takes as input the public key pk, the message m, the set of signers C, and the set of protocol messages sent by each party during the $\mathsf{Sign}_1, \ldots, \mathsf{Sign}_\kappa$ signing stages, and outputs a joint signature σ.*
- $\mathsf{Verify}(\mathsf{pk}, m, \sigma) \to 0/1$*: A deterministic algorithm that takes as input the public key pk, a message m, and signature σ and outputs 1 to indicate accept if the signature verifies; otherwise, it outputs 0 to indicate reject.*

Remark 7 (Distributed key generation)**.** Our definition assumes a centralized key generation algorithm KeyGen to generate the public key pk and public key shares $\{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]}$. However, our scheme and proofs can be adapted to use a fully decentralized distributed key generation protocol (DKG).

Remark 8 (Deferring the Choice of Coalition to Later Rounds)**.** Our definition assumes that the coalition of signers C is provided in the first round of signing Sign_1. However, some constructions (as is the case with Arctic) may defer the choice of C to later rounds.

$\mathsf{Exp}^{\mathsf{uf}}_{\mathsf{DT},\mathcal{A}}(\lambda, n, t, \mu)$

$\mathsf{par} \xleftarrow{\$} \mathsf{DT.Setup}(1^\lambda)$
$Q \leftarrow \emptyset$ // set of queried messages
$(\mathsf{corrupt}, \mathsf{st}_A) \xleftarrow{\$} \mathcal{A}(\mathsf{par}, n, t, \mu)$
if $|\mathsf{corrupt}| \geq t$
  return $\bot$
$\mathsf{honest} \leftarrow [n] \setminus \mathsf{corrupt}$
$(\mathsf{pk}, \{\mathsf{pk}_i, \mathsf{sk}_i\}_{i=1}^n) \xleftarrow{\$} \mathsf{DT.KeyGen}(n, t, \mu)$
$\mathsf{in} \leftarrow (\mathsf{pk}, \{\mathsf{pk}_i\}_{i=1}^n, \{\mathsf{sk}_j\}_{j \in \mathsf{corrupt}}, \mathsf{st}_A)$
$(m^*, \sigma^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{Sign}_i, i \in [\kappa]}}(\mathsf{in})$
if $m^* \notin Q \wedge \mathsf{DT.Verify}(\mathsf{pk}, m^*, \sigma^*) = 1$
  return 1
return 0

$\mathcal{O}^{\mathsf{Sign}_1}(k, m, C)$

// k denotes the participant identifier
$\rho_1^{(k)} \leftarrow \mathsf{DT.Sign}_1(k, \mathsf{sk}_k, m, C)$
return $\rho_1^{(k)}$

$\vdots$

$\mathcal{O}^{\mathsf{Sign}_j}(k, m, C, \{\rho_{j-1}^{(i)}\}_{i \in C})$

// *for* $j \in \{2, \ldots, \kappa - 1\}$
$\rho_j^{(k)} \leftarrow \mathsf{DT.Sign}_i(k, \mathsf{sk}_k, m, C, \{\rho_{j-1}^{(i)}\}_{i \in C})$
return $\rho_j^{(k)}$

$\vdots$

$\mathcal{O}^{\mathsf{Sign}_\kappa}(k, m, C, \{\rho_{\kappa-1}^{(i)}\}_{i \in C})$

$Q \leftarrow Q \cup \{m\}$
$\rho_\kappa^{(k)} \leftarrow \mathsf{DT.Sign}_\kappa(k, \mathsf{sk}_k, m, C, \{\rho_{\kappa-1}^{(i)}\}_{i \in C})$
return $\rho_\kappa^{(k)}$

Figure 5.1: Unforgeability experiment for a deterministic threshold signature scheme with a κ-round signing protocol.

Correctness. A deterministic threshold signature scheme DT is *correct* if for all security parameters λ, all $1 \leq t \leq \mu \leq n$, all participant identifiers $k \in [n]$, all $C \subseteq [n]$ such that $\mu \leq |C| \leq n$, and for all messages $m \in \{0,1\}^*$, the following relation holds:

$$\begin{aligned} \mathsf{DT.Verify}(\mathsf{pk}, m, \sigma) = 1, \text{ for} \\ (\mathsf{pk}, \{\mathsf{pk}_i, \mathsf{sk}_i\}_{i \in [n]}) \xleftarrow{\$} \mathsf{DT.KeyGen}(n, t, \mu), \text{ where} \\ \rho_1^{(k)} \leftarrow \mathsf{DT.Sign}_1(k, \mathsf{sk}_k, m, C), \ldots, \rho_\kappa^{(k)} \leftarrow \mathsf{DT.Sign}_\kappa(k, \mathsf{sk}_k, m, C, \{\rho_{\kappa-1}^{(i)}\}_{i \in C}), \text{ and} \\ \sigma \leftarrow \mathsf{DT.Combine}(\mathsf{pk}, m, C, \{\rho_1^{(i)}, \ldots, \rho_\kappa^{(i)}\}_{i \in C}) \end{aligned}$$

5.4.1 Unforgeability

We present a game-based definition of unforgeability for a deterministic threshold signature scheme in Figure 5.1. This notion of unforgeability is analogous to the standard notion of chosen message attack (EUF-CMA) for standard signature schemes [GMR88]. We present the adversary as a static adversary, which is allowed to corrupt at most $t-1$ signers.

The game assumes a static adversary that picks the players it corrupts at the beginning of the game. The public parameters par are implicitly given as input to all algorithms, and $\rho_1^{(k)}, \ldots, \rho_\kappa^{(k)}$ represent protocol messages sent by participant k throughout the interactive signing protocol. The static unforgeability game for a deterministic threshold signature scheme is nearly identical to that of a randomized scheme, with the exception that the environment does not maintain state between invocations of signing oracles by the adversary, and the adversary provides as input the message m and coalition of signers C (of its choosing) for each call to a signing oracle.

In the static unforgeability game for a deterministic threshold signature scheme, the environment allows the adversary to sample the corrupt parties $\mathsf{corrupt} \subset [n]$. If the set of corrupt parties is smaller than the corruption threshold t, it derives the set of honest parties $\mathsf{honest} \subseteq [n]$ as the remaining parties in $[n]$. The environment then runs KeyGen to derive the joint public key pk representing the set of n signers, the set of public key shares $\{\mathsf{pk}_i\}_{i\in[n]}$, and the secret key shares $\{\mathsf{sk}_i\}_{i\in[n]}$. The adversary is then run on input $n, t, \mu, \mathsf{par}, \mathsf{pk}, \{\mathsf{pk}_i\}_{i\in[n]}$, and the corrupt signing key shares $\{\mathsf{sk}_j\}_{j\in\mathsf{corrupt}}$. The adversary can then query any honest signers $k \in \mathsf{honest}$ of its choosing at each step in the signing protocol, and has full power over choosing the set of signers C and the message m. Additionally, the adversary may query the signing round oracles in arbitrary order. However, unlike in the randomized setting as shown in Figure 2.8, the environment for a deterministic threshold signature scheme does not maintain any session identifiers, or state about ongoing signing sessions. The adversary wins if it can produce a valid forgery σ^* with respect to the joint public key pk on a message m^* that has not been queried to $\mathcal{O}^{\mathsf{Sign}_\kappa}$ (i.e., in the final round of signing). Importantly, this definition allows the adversary to be *rushing*, meaning it can wait to produce its outputs after having seen the honest outputs first. The adversary may also be *concurrent*, meaning it can open simultaneous signing sessions at once, or choose not to complete a signing session.

Definition 5.4.2 (Unforgeability)**.** *Let the advantage of a static adversary $\mathcal{A}$ playing the unforgeability game against a deterministic threshold signature scheme DT as defined in Figure 5.1 be as follows:*

$$\mathsf{Adv}^{\mathsf{uf}}_{\mathsf{DT},\mathcal{A}}(\lambda, n, t, \mu) = \Pr[\mathsf{Exp}^{\mathsf{uf}}_{\mathsf{DT},\mathcal{A}}(\lambda, n, t, \mu) = 1]$$

A deterministic threshold signature scheme DT *is* unforgeable *if for all PPT adversaries* $\mathcal{A}$, $\mathsf{Adv}^{\mathsf{uf}}_{\mathsf{DT},\mathcal{A}}(\lambda, n, t, \mu)$ *is negligible in* λ, *for all* $n, t, \mu \in \mathbb{N}$, *where* $t \leq \mu \leq n$.

5.5 Verifiable Pseudorandom Secret Sharing

We now introduce an extension to pseudorandom secret sharing that we call a *Verifiable Pseodurandom Secret Sharing* (VPSS) scheme. We begin by motivating the need for a VPSS, and then formally define VPSS and its security properties. Finally, in Section 5.5.3, we give a concrete VPSS construction, Shine, that we later use as a building block for Arctic.

5.5.1 Motivation

A verifiable random function (VRF) [MRV99] is a keyed PRF, such that the PRF can be evaluated using only knowledge of a secret key, but is publicly verifiable given a public key. In particular, in addition to outputting the evaluation of the VRF, it also outputs a zero-knowledge proof that the VRF was evaluated correctly. A *distributed* VRF [NPR99, Dod03, GLOW21] allows the evaluation algorithm to be partitioned among a set of participants, all of whom are equally trusted.

However, the use case of generating deterministic nonces for threshold Schnorr signatures presents a slightly different challenge. Instead of directly verifying that the nonce r_i was correctly generated, we instead need to verify correctness in zero-knowledge, with respect to a commitment $R_i = g^{r_i}$. One approach in the literature is to employ non-algebraic pseudorandom functions to generate the nonce, and then prove in zero-knowledge the correctness of the corresponding commitment [NRSW20, GKMN21]. Unfortunately, generating such zero-knowledge proofs requires higher computational and complexity overhead.

We instead take a different approach towards verifying that a distributed pseudorandom function was correctly performed by a set of parties. Instead of each participant outputting a zero-knowledge proof that they individually performed the action correctly, we observe that the correctness of the evaluation can instead be *collectively* publicly verified, assuming some threshold of honest participants. Such an observation leads naturally to employing a pseudorandom secret sharing scheme [CDI05], which is categorized by all parties holding a set of secret key shares, and individually and non-interactively being able to generate secret shares of additional pseudorandom values. We show that pseudorandom secret sharing schemes have a natural extension to the publicly verifiability setting. Finally, we

give a concrete and efficient VPSS scheme, where all participants can publish commitments to their generated shares, and perform polynomial interpolation over elements in the public domain to ensure correctness.

We next build upon these observations by formalizing the notion of a verifiable pseudorandom secret sharing scheme.

5.5.2 VPSS Definition and Notions of Security

We now present an extension on pseudorandom secret sharing that we call a *Verifiable Pseudorandom Secret Sharing* (VPSS) scheme. In particular, a VPSS defines an additional verify algorithm to ensure that the pseudorandom function was performed correctly by collectively verifying outputs by all participants.

Definition 5.5.1. *A* **Verifiable Pseudorandom Secret Sharing** *scheme is the tuple of algorithms* $\mathsf{VP} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Gen}, \mathsf{Verify}, \mathsf{Agg}, \mathsf{Recover})$*, such that:*

- $\mathsf{Setup}(1^\lambda)$*: A probabilistic algorithm that accepts as input the security parameter* λ*, and outputs public parameters* par*, where* par *is given as implicit input to all other algorithms.*
- $\mathsf{KeyGen}(n, t, \mu) \to \perp/(\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$*: A probabilistic algorithm that accepts as input the total number of participants* n*, a corruption threshold* t*, and the minimum number of parties* μ *required to participate in* Gen*. On failure, output* $\perp$*, otherwise, output* n *secret keys, one for each of the* n *participants.*
- $\mathsf{Gen}(k, \mathsf{sk}_k, w) \to (d_k, D_k)$*: A deterministic algorithm that accepts as input a participant identifier* k*, the secret key for that participant* sk_k*, and some input* w*. Generates the pseudorandom secret share* $d_k \in \mathcal{P}$ *from some domain* $\mathcal{P}$*, using* sk_k *as the random seed and* w *as the corresponding input. Then, generates its public commitment* $D_k \in \mathcal{O}$ *to* d_k *from some domain* $\mathcal{O}$*. Outputs* (d_k, D_k)*.*
- $\mathsf{Verify}(t, \mu, C, \{D_j\}_{j \in C}) \to \{0, 1\}$*: A deterministic algorithm that accepts as input the corruption threshold* t*, the minimum number of participants* μ*, a coalition of participants* C *such that* $|C| \geq \mu$*, and a set of commitments to pseudorandom secret shares* $(D_j)_{j \in C}$ *for that coalition. Outputs* 1 *to indicate that the secret sharing was correctly performed, otherwise, output* 0*.*

- $\mathsf{Agg}(t, \mu, C, \{D_j\}_{j \in C}) \to D$: *A deterministic algorithm that accepts as input the corruption threshold t, the minimum number of participants μ, a coalition of participants C such that $|C| \geq \mu$, and the set of commitments to pseudorandom secret shares. Outputs the commitment to the aggregated pseudorandom secret D.*

- $\mathsf{Recover}(t, \mu, C, \{d_j\}_{j \in C}) \to \perp/(d, D)$: *A deterministic algorithm that accepts a corruption threshold t, the minimum number of participants μ, a coalition C with $|C| \geq \mu$, and a set of shares $\{d_j\}_{j \in C}$. It outputs $\perp$ if $|C| < \mu$ or if the shares are inconsistent. Otherwise, it recovers d using the set of shares, derives the corresponding commitment D, and outputs (d, D).*

Correctness. A VPSS is **correct** if for all λ, possible inputs w and choices of $t, \mu, n \in \mathbb{N}$ where $t \leq \mu \leq n$, when $\mathsf{par} \xleftarrow{\$} \mathsf{VP.Setup}(1^\lambda)$ and $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n) \xleftarrow{\$} \mathsf{VP.KeyGen}(n, t, \mu)$, there exists $d \in \mathcal{P}, D \in \mathcal{O}$ such that Equation 5.1 holds:

$$\begin{aligned} &\text{for all } i \in [n] \text{ and for all } C \subseteq [n], |C| \geq \mu, \text{ when} \\ &\qquad \mathsf{VP.Gen}(i, \mathsf{sk}_i, w) \to (d_i, D_i), \text{ then} \\ &\qquad \mathsf{VP.Verify}(t, \mu, C\{D_j\}_{j \in C}) = 1, \text{ and} \\ &\qquad \mathsf{VP.Recover}(t, \mu, C, (d_j)_{j \in C}) \to (d, D), \text{ and} \\ &\qquad \mathsf{VP.Agg}(t, \mu, C, (D_j)_{j \in C}) \to D \end{aligned} \tag{5.1}$$

5.5.2.1 Security

Similarly to verifiable random functions [MRV99, NPR99, Dod03, GLOW21], we say that a VPSS is *secure* if it is unique, verifiable, and pseudorandom.

Uniqueness. Intuitively, a VPSS is unique if it produces exactly one commitment to a (pseudorandom) value for each input w, regardless of the choice of coalition. More specifically, picking different coalitions should always result in the same commitment D, when the input w remains the same. We show the VPSS uniqueness experiment in Figure 5.2.

In the uniqueness experiment, the adversary is allowed to query honest participants for shares and commitments on inputs of its choosing. The adversary then outputs the evaluation input w, two coalitions C, C', and two sets of commitments $(D_i)_{i \in C \cap \mathsf{corrupt}}$, $(D'_i)_{i \in C' \cap \mathsf{corrupt}}$.

$\mathsf{Exp}^{\mathrm{uq}}_{\mathsf{VP},\mathcal{A}}(\lambda, n, t, \mu)$

$\mathsf{par} \xleftarrow{\$} \mathsf{VP.Setup}(1^\lambda)$
$(\mathsf{corrupt}, \mathsf{st}_A) \xleftarrow{\$} \mathcal{A}(\mathsf{par}, n, t, \mu)$
return $\perp$ **if** $|\mathsf{corrupt}| \geq t$
$\mathsf{honest} \leftarrow [n] \setminus \mathsf{corrupt}$
$(\mathsf{sk}_1, \ldots, \mathsf{sk}_n) \xleftarrow{\$} \mathsf{VP.KeyGen}(n, t, \mu)$
$(w, \mathsf{out}) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{Gen}}}((\mathsf{sk}_j)_{j \in \mathsf{corrupt}}, \mathsf{st}_A)$
$(C, (D_i)_{i \in C \cap \mathsf{corrupt}}), (C', (D'_i)_{i \in C' \cap \mathsf{corrupt}}) \leftarrow \mathsf{out}$
$(d_j, D_j) \leftarrow \mathsf{VP.Gen}(j, \mathsf{sk}_j, w), j \in \mathsf{honest}$
$S_1 \leftarrow (D_i)_{i \in C \cap \mathsf{corrupt}} \cup (D_i)_{i \in C \cap \mathsf{honest}}$
$S_2 \leftarrow (D'_i)_{i \in C' \cap \mathsf{corrupt}} \cup (D_i)_{i \in C' \cap \mathsf{honest}}$
return 0 **if** $\mathsf{VP.Verify}(t, \mu, C, S_1) \neq 1$
return 0 **if** $\mathsf{VP.Verify}(t, \mu, C', S_2) \neq 1$
$D \leftarrow \mathsf{VP.Agg}(t, \mu, C, S_1)$
$D' \leftarrow \mathsf{VP.Agg}(t, \mu, C', S_2)$
if $D \neq D'$
  return 1
return 0

$\mathcal{O}^{\mathsf{Gen}}(k, w_i)$

// k denotes the participant identifier
$(d_k, D_k) \leftarrow \mathsf{VP.Gen}(k, \mathsf{sk}_k, w_i)$
return (d_k, D_k)

Figure 5.2: Uniqueness game for a VPSS.

The environment then derives the honest players' shares and commitments on w, producing $(d_j, D_j)_{j \in \mathsf{honest}}$. After deriving the sets $S_1 \leftarrow (D_i)_{i \in C \cap \mathsf{corrupt}} \cup (D_i)_{i \in C \cap \mathsf{honest}}$ and $S_2 \leftarrow (D'_i)_{i \in C' \cap \mathsf{corrupt}} \cup (D_i)_{i \in C' \cap \mathsf{honest}}$, the adversary loses if the verification algorithm outputs 0 on either set. If both sets verify, the adversary wins if the corresponding commitments for each set are not equal.

We define uniqueness for a VPSS more formally in Definition 5.5.2.

Definition 5.5.2. *Let the advantage of an adversary $\mathcal{A}$ playing the uniqueness game as defined in Figure 5.2 be as follows:*

$$\mathsf{Adv}^{\mathrm{uq}}_{\mathsf{VP},\mathcal{A}}(\lambda, n, t, \mu) = \Pr[\mathsf{Exp}^{\mathrm{uq}}_{\mathsf{VP},\mathcal{A}}(\lambda, n, t, \mu) = 1]$$

A VPSS VP *is* **unique** *if for all PPT adversaries $\mathcal{A}$, $\mathsf{Adv}^{\mathrm{unq}}_{\mathsf{VP},\mathcal{A}}$ is a negligible function of λ, for $n, t, \mu \in \mathbb{N}$, $t \leq \mu \leq n$.*

$\mathsf{Exp}^{\mathrm{verf}}_{\mathsf{VP},\mathcal{A}}(\lambda, n, t, \mu)$

$\mathsf{par} \xleftarrow{\$} \mathsf{VP.Setup}(1^\lambda)$
$(\mathsf{corrupt}, \mathsf{st}_A) \xleftarrow{\$} \mathcal{A}(\mathsf{par}, n, t, \mu)$
return $\perp$ **if** $|\mathsf{corrupt}| \geq t$
$\mathsf{honest} \leftarrow [n] \setminus \mathsf{corrupt}$
$(\mathsf{sk}_1, \ldots, \mathsf{sk}_n) \xleftarrow{\$} \mathsf{VP.KeyGen}(n, t, \mu)$
$(w, C, (D_j)_{j \in C \cap \mathsf{corrupt}}) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{Gen}}}((\mathsf{sk}_j)_{j \in \mathsf{corrupt}}, \mathsf{st}_A)$
for $j \in C$ **do**
  $(d'_j, D'_j) \leftarrow \mathsf{VP.Gen}(j, \mathsf{sk}_j, w)$
if $(D_j)_{j \in C \cap \mathsf{corrupt}} = (D'_j)_{j \in C \cap \mathsf{corrupt}}$
  return 0
  // *$\mathcal{A}$ must deviate from the protocol to win*
$S \leftarrow (D_k)_{k \in C \cap \mathsf{corrupt}} \cup (D'_j)_{j \in C \cap \mathsf{honest}}$
if $\mathsf{VP.Verify}(t, \mu, C, S) = 1$
  return 1
return 0

$\mathcal{O}^{\mathsf{Gen}}(k, w_i)$

// k denotes the participant id
$(d_k, D_k) \leftarrow \mathsf{VP.Gen}(k, \mathsf{sk}_k, w_i)$
return (d_k, D_k)

Figure 5.3: Verifiability game for a VPSS. Note the environment enforces the authenticity of participant contributions.

Verifiability. Intuitively, a VPSS is verifiable if given a set of commitments to shares from a coalition of participants, the verify algorithm will detect if some subset of players deviated from the protocol. We show the VPSS verifiability experiment in Figure 5.3.

In the experiment, the adversary is allowed to query honest participants for shares and commitments on inputs of its choosing. The adversary then outputs a coalition C, a VPSS input w, and a set of commitments $(D_i)_{i \in C \cap \mathsf{corrupt}}$. The environment then follows the protocol, deriving both the corrupt and honest players' commitments on w, and producing $(D'_j)_{j \in C}$. The adversary loses if its output is identical to the honestly derived commitments for the corrupted players. Then, the environment checks if the set of commitments $(D_i)_{i \in C \cap \mathsf{corrupt}} \cup (D'_j)_{j \in C \cap \mathsf{honest}}$ are valid with respect to C and w. If so, the adversary wins, otherwise, it loses.

We define verifiability for a VPSS more formally in Definition 5.5.3.

Definition 5.5.3. *Let the advantage of an adversary $\mathcal{A}$ playing the verifiability game as*

$\mathsf{Exp}^{\mathrm{psdr}}_{\mathsf{VP},\mathcal{A}}(\lambda, n, t, \mu)$

$b \xleftarrow{\$} \{0,1\}$
$z \xleftarrow{\$} \{0,1\}$
$Q \leftarrow \emptyset$
$\mathsf{par} \xleftarrow{\$} \mathsf{VP.Setup}(1^\lambda)$
$(\mathsf{corrupt}, \mathsf{st}_A) \xleftarrow{\$} \mathcal{A}(\mathsf{par}, n, t, \mu)$
return $\perp$ **if** $|\mathsf{corrupt}| \geq t$
$\mathsf{honest} \leftarrow [n] \setminus \mathsf{corrupt}$
$(\mathsf{sk}_1, \ldots, \mathsf{sk}_n) \xleftarrow{\$} \mathsf{VP.KeyGen}(n, t, \mu)$
$\mathsf{out} \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{Gen}}}((\mathsf{sk}_j)_{j \in \mathsf{corrupt}}, \mathsf{st}_A)$
$(b', w^*, C^*, \{D_j\}_{j \in \mathsf{corrupt}}) \leftarrow \mathsf{out}$
if $b = 1$
  return z **if** $Q[w^*] = \perp$
  $(D, \{D_i\}_{i \in \mathsf{honest}}) \leftarrow Q[w^*]$
  if $\mathsf{VP.Verify}(t, \mu, C^*, \{D_k\}_{k \in C^*}) \neq 1$
    return z
  if $\mathsf{VP.Agg}(t, \mu, C^*, \{D_k\}_{k \in C^*}) \neq D$
    return 1
return 1 **if** $b \stackrel{?}{=} b'$
return 0

$\mathcal{O}^{\mathsf{Gen}}(k, w)$

// k denotes the participant identifier
return $\perp$ **if** $k \notin \mathsf{honest}$
if $b = 0$
  $(d_k, D_k) \leftarrow \mathsf{VP.Gen}(k, \mathsf{sk}_k, w)$
  return (d_k, D_k)
// $b = 1$ *case*
if $Q[w] \neq \perp$
  $(D, \{(d_i, D_i)\}_{i \in \mathsf{honest}}) \leftarrow Q[w]$
else
  $d' \xleftarrow{\$} \mathcal{P}$
  $\mathsf{in} \leftarrow (w, d', \mathsf{corrupt}, n, (\mathsf{sk}_j)_{j \in \mathsf{corrupt}})$
  $(D, \{(d_i, D_i)\}_{i \in \mathsf{honest}}) \leftarrow \mathsf{SimGen}(\mathsf{in})$
  $Q[w] = (D, \{(d_i, D_i)\}_{i \in \mathsf{honest}})$
return (d_k, D_k)

Figure 5.4: Pseudorandomness game for a VPSS.

defined in Figure 5.3 be as follows:

$$\mathsf{Adv}^{\mathrm{verf}}_{\mathsf{VP},\mathcal{A}}(\lambda, n, t, \mu) = \Pr[\mathsf{Exp}^{\mathrm{verf}}_{\mathsf{VP},\mathcal{A}}(\lambda, n, t, \mu) = 1]$$

A VPSS VP *is* **verifiable** *if for all PPT adversaries* $\mathcal{A}$, $\mathsf{Adv}^{\mathrm{verf}}_{\mathsf{VP},\mathcal{A}}$ *is a negligible function of* λ, *for* $n, t, \mu \in \mathbb{N}$, $t \leq \mu \leq n$.

Pseudorandom. Intuitively, a VPSS is pseudorandom if the adversary has negligible advantage distinguishing between a real VPSS output from one that is randomly sampled. We show the VPSS pseudorandomness experiment in Figure 5.4.

In the pseudorandomness experiment, the environment begins by picking a random bit $b \xleftarrow{\$} \{0,1\}$. It then performs key generation, and initializes the adversary with the public parameters and the corrupted parties' secret key shares.

The adversary is allowed to query $\mathcal{O}^{\mathsf{Gen}}$ on honest participants for shares and commitments on inputs of the adversary's choosing. If $b = 0$, the environment responds by following the VPSS protocol. If $b = 1$, the environment first checks to see if it has responded to the query before, replying with the response if so. Otherwise, it uses a simulating algorithm SimGen to simulate generating honest players' shares and commitments. The details of SimGen depend on the specifics of the VPSS construction.

The adversary outputs $(b', w^*, C^*, \{D_j\}_{j \in \mathsf{corrupt}})$, where b' is the adversary's guess for the value of b, and $(w^*, C^*, \{D_j\}_{j \in \mathsf{corrupt}})$ corresponds to one of the sessions the adversary initiated with $\mathcal{O}^{\mathsf{Gen}}$.

If $b = 1$, the environment checks if $(w^*, C^*, \{D_j\}_{j \in \mathsf{corrupt}})$ corresponds to an existing session and is consistent with the honest parties' commitments, outputting a random coin if it does not. Otherwise, the environment next checks that the aggregated commitment $\mathsf{VP.Agg}(t, \mu, C^*, \{D_k\}_{k \in C^*})$ is equal to the simulated commitment D for that session, if the check does not hold, the environment outputs 1.

Otherwise, the environment checks if the adversary's guess b' is equal to b; outputting 1 if the check holds, otherwise, outputting 0.

We define pseudorandomness for a VPSS more formally in Definition 5.5.4.

Definition 5.5.4. *Let the advantage of an adversary $\mathcal{A}$ playing the pseudorandomness game as defined in Figure 5.4 be as follows:*

$$\mathsf{Adv}^{\mathrm{psdr}}_{\mathsf{VP},\mathcal{A}}(\lambda, n, t, \mu) = \left|\Pr[\mathsf{Exp}^{\mathrm{psdr}}_{\mathsf{VP},\mathcal{A}}(\lambda, n, t, \mu) = 1] - 1/2\right|$$

A VPSS VP is **pseudorandom** *if for all PPT adversaries $\mathcal{A}$, $\mathsf{Adv}^{\mathrm{psdr}}_{\mathsf{VP},\mathcal{A}}$ is a negligible function of λ, for $n, t, \mu \in \mathbb{N}$, $t \leq \mu \leq n$.*

5.5.3 Shine, A Concrete Verifiable Pseudorandom Secret Sharing Scheme

We now define a concrete VPSS that we call Shine, that builds upon the pseudorandom secret sharing scheme as defined by Cramer, Damgård, and Ishai [CDI05]. However, Shine

additionally defines a verify algorithm that is publicly verifiable (assuming the authenticity of the inputs), as well as an algorithm to combine participant commitments.

As a reminder, the pseudorandom secret sharing scheme by Cramer, Damgård, and Ishai [CDI05] itself builds on replicated secret sharing [ISN89]. However, Cramer et al. define a mechanism to non-interactively convert shares of a replicated secret sharing scheme to shares of a Shamir-secret-shared value. They then show how this mechanism can be used as a distributed pseudorandom function; i.e., given a replicated secret sharing of a random value, participants can generate Shamir secret shares of an unbounded number of pseudorandom values, without interaction.

We extend this pseudorandom secret sharing scheme by Cramer et al. and additionally define a public verifiability mechanism. More specifically, we define a Verify function that uses the set of participants' commitments to ensure all participants performed the evaluation step in a consistent manner. Such consistency checks have been used in prior literature [BBG+21], but our check is performed over commitments to secret shares, as opposed to verifying the shares directly. We prove that Shine is secure assuming a cryptographically secure hash function, when at minimum $2t-1$ parties participate in evaluation and up to $t-1$ participants are corrupted (honest majority).

We now define Shine in more detail. We show KeyGen for Shine as a centralized operation, but this operation can easily be decentralized. Shine is additionally defined with respect to a hash function H, where $\mathsf{H} : \mathbb{Z}_q \times \{0,1\}^* \to \mathbb{Z}_q$ is a cryptographically secure hash function.

- $\mathsf{Shine.Setup}(1^\lambda)$: Accepts as input the security parameter λ, outputs public parameters $\mathsf{par} = (\mathbb{G}, q, g)$, where $(\mathbb{G}, q, g)$ is generated by $\mathsf{GroupGen}(1^\lambda)$. par is given as an implicit input to all other algorithms.

- $\mathsf{Shine[H].KeyGen}(n, t, \mu) \to \perp/(\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$: On input the total number of participants n, the corruption threshold t, and minimum number of participants μ, the dealer performs replicated secret sharing of a random secret $\mathsf{sk} \in \mathbb{Z}_q$ [CDI05], following the following steps:

  1. First, the dealer checks if $\mu \geq 2t-1$; if the check fails, output $\perp$.
  2. Let A be the set such that $A = \binom{[n]}{t-1}$ (notation as defined in Section 2.1.1), and let γ be the size of A; i.e, $\gamma = |A| = \binom{n}{t-1}$.
  3. Generate γ secret shares $\{\phi_i\}_{i\in[\gamma]}$, by sampling $\phi_i \xleftarrow{\$} \mathbb{Z}_q, i \in [\gamma]$. Implicitly, the secret sk is such that $\mathsf{sk} = \sum_{i\in\gamma} \phi_i$.

4. Initialize empty sets $\mathsf{sk}_1 = \emptyset, \ldots, \mathsf{sk}_n = \emptyset$.
5. For each set $a_i \in A, i \in [\gamma]$ and each participant identifier $j \in [n] \setminus a_i$, append $\mathsf{sk}_j \leftarrow \mathsf{sk}_j \cup \{(a_i, \phi_i)\}$.
6. Output $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$. Each sk_j is a set of size $\delta = \binom{n-1}{t-1}$, consisting of the tuples (a_i, ϕ_i). Each a_i is itself a set, such that $a_i \subset [n]$ and $|a_i| = t - 1$, where if $a_i \in \mathsf{sk}_j$, then $j \notin a_i$.

- $\mathsf{Shine}[\mathsf{H}].\mathsf{Gen}(k, \mathsf{sk}_k, w) \to (d_k, D_k)$: On input participant identifier $k \in [n]$, secret key share sk_k, and input $w \in \{0,1\}^*$, perform the following steps:

  1. Parse $\{(a_i, \phi_i)\}_{i \in [\delta]} \leftarrow \mathsf{sk}_k$.
  2. For each set a_i, let $L'_{a_i}(x)$ be the degree $t-1$ polynomial defined by the set a_i, as given in Equation 5.2:

$$L'_{a_i}(x) = \prod_{j \in a_i} \frac{j - x}{j} \tag{5.2}$$

     Note that $L'_{a_i}(j) = 0$ for all $j \in a_i$ and $L'_{a_i}(0) = 1$.
  3. Obtain the share via Equation 5.3:

$$d_k \leftarrow \sum_{i \in [\delta]} \mathsf{H}(\phi_i, w) \cdot L'_{a_i}(k) \tag{5.3}$$

     Note that the $L'_{a_i}(k)$ values can be precomputed by participant k, as they are independent of w, so this step requires δ evaluations of H and δ multiplications and additions in $\mathbb{Z}_q$.
  4. Derive the commitment to d_k as $D_k \leftarrow g^{d_k}$.
  5. Output (d_k, D_k).

- $\mathsf{Shine}[\mathsf{H}].\mathsf{Verify}(t, \mu, C, \{D_j\}_{j \in C}) \to \{0,1\}$: Perform the following steps:

  1. If $|C| \not\geq \mu$, return 0.
  2. Otherwise, define $B = (B_0, B_1, \ldots, B_{|C|-1})$ to be the tuple of $|C|$ commitments to *coefficients* of a polynomial f defined "in the exponent," where each $D_i, i \in C$ is a commitment to a *point* on the same polynomial f. Each B_i can be derived via Equation 5.4:

$$B_i = \prod_{j \in C} D_j^{L_{j,i}} \tag{5.4}$$

where $L_{j,i}$ is the coefficient of the i^{th} term x^i of the j^{th} Lagrange polynomial $L_j(x)$ for the coalition C, as described in Section 2.1.1.

3. Let $\ell = |C| - t$. The verifier now checks that all participants followed the protocol honestly, by checking in the exponent that their shares lie on a polynomial of degree $t-1$. In particular, the verifier ensures that B is a commitment to a polynomial of degree at most $t-1$,[1] by checking the last ℓ entries in B are equal to the identity of $\mathbb{G}$:[2]

$$B_{t-1+i} = I_{\mathbb{G}}, \forall\ i \in [\ell] \tag{5.5}$$

4. If the check in Equation 5.5 holds, output 1, otherwise, output 0.

- $\mathsf{Shine[H].Agg}(t, \mu, C, \{D_j\}_{j \in C}) \rightarrow D$: Accepts as input the corruption threshold t, minimum participants μ, the coalition C, and set of (verified) commitments $\{D_j\}_{j \in C}$). The commitments can be combined as in Equation 5.6, where the coefficients λ_j are determined by the coalition C.

$$D \leftarrow \prod_{j \in C} D_j^{\lambda_j} = B_0 \tag{5.6}$$

  Output D.

- $\mathsf{Shine[H].Recover}(t, \mu, C,\ \{d_j\}_{j \in C}) \rightarrow \perp/(d, D)$: Receives as input the corruption threshold t, the minimum number of participants μ, the coalition C, and the set of pseudorandom secret shares $\{d_j\}_{j \in C}$. Performs the following steps:

  1. If $|C| < \mu$, or $C \not\subset [n]$, output $\perp$.
  2. For each $d_j, j \in C$, derive $D_j \leftarrow g^{D_j}$.
  3. If $\mathsf{Shine.Verify}(t, \mu, C, \{D_j\}_{j \in C}) \neq 1$, then output $\perp$.
  4. Otherwise, the shares can be combined as in Equation 5.7, where the λ_j are determined by the coalition C.

$$d \leftarrow \sum_{j \in C} d_j \cdot \lambda_j \tag{5.7}$$

  5. Derive the commitment to d as $D \leftarrow g^d$.
  6. Output (d, D).

[1] When at least t parties follow the protocol honestly, their shares will completely define this polynomial.
[2] As an optimization, note that $B_1, \ldots, B_{t-1}$ need not be computed.

Correctness. By correctness of pseudorandom secret sharing as defined by Cramer et al. [CDI05], when key generation is honestly performed, the Gen algorithm produces secret shares $d_i = f(i)$ that are points on a degree $t-1$ Shamir secret sharing polynomial $f(x) = \sum_{i \in [\gamma]} \mathsf{H}(\phi_i, w) L'_{a_i}(x)$. The combined commitment $d = \sum_{i \in C} d_i \lambda_i$ for $C \subseteq [n], |C| \geq 2t-1$ is a commitment to a pseudorandom secret value $d = f(0) = \sum_{i \in [\gamma]} \mathsf{H}(\phi_i, w)$. Agg and Recover simply perform polynomial interpolation of these values with respect to the set of participants, and so are likewise correct.

Verify is correct because honest parties output commitments $D_i = g^{f(i)}$ for the degree $t-1$ polynomial f defined above. Therefore, if all parties are honest, the coefficients of x^t, $\ldots$, $x^{|C|-1}$ of $f(x)$ will be 0, so the commitments B_t, $\ldots$, $B_{|C|-1}$ to those coefficients will be $I_{\mathbb{G}}$ (the identity element of $\mathbb{G}$), and so Verify will output 1.

5.5.3.1 Security

Shine is verifiable, unique, and pseudorandom. We give the corresponding proofs next.

Theorem 9. *Shine is perfectly verifiable, assuming the adversary controls at most $t-1$ participants, at least t participants are honest, and participants exchange messages over an authenticated channel.*

Proof. Recall that in the verifiability game, the adversary is required to provide an input w, a coalition $C \subseteq [n]$, and a set $(D_i)_{i \in C \cap \mathsf{corrupt}}$, such that $(D_i)_{i \in C \cap \mathsf{corrupt}}$ does not equal the set $(D'_i)_{i \in C \cap \mathsf{corrupt}}$, where $(d'_i, D'_i) \leftarrow \mathsf{Shine.Gen}(i, \mathsf{sk}_i, w)$ is obtained honestly. The environment derives the values $(d_j, D_j) \leftarrow \mathsf{Shine.Gen}(j, \mathsf{sk}_j, w), j \in C \cap \mathsf{honest}$ for honest participants, reflecting the requirement that participants' messages are received over authenticated channels. The adversary wins when $\mathsf{Shine.Verify}(t, \mu, C, S)$ outputs 1, where $S = (D_j)_{j \in C}$.

For $\mathsf{Shine.Verify}(t, \mu, C, S)$ to output 1, then Equation 5.5 must hold. But for Equation 5.5 to hold, then $(D_j)_{j \in C}$ must define a degree $t-1$ polynomial in the exponent. However, this polynomial is completely defined by the set of honest participant commitments $(D_j)_{j \in C \cap \mathsf{honest}}$, because we assume $|\mathsf{honest}| \geq t$, and a degree $t-1$ polynomial is uniquely defined by t points.

Hence, when corrupted participants do not honestly follow the protocol, then the resulting polynomial committed to in S will be of higher degree than $t-1$. As such, Shine is perfectly verifiable, as the advantage to the adversary to win the game is zero. □

Theorem 10. *Shine is perfectly unique, assuming the adversary controls at most $t-1$ participants, at least t participants are honest, and participants exchange messages over an authenticated channel.*

Proof. In the uniqueness game given in Figure 5.2, the adversary is required to provide an input w and the tuples $(C, (D_i)_{i \in C \cap \mathsf{corrupt}}), (C', (D'_i)_{i \in C' \cap \mathsf{corrupt}})$. The environment then derives honest parties' contributions $(D_j) \leftarrow \mathsf{Shine.Gen}(j, \mathsf{sk}_j, w)$, again reflecting the authenticated channel, and the environment derives the commitment sets $S_1 = (D_j)_{j \in C \cap \mathsf{honest}} \cup (D_i)_{i \in C \cap \mathsf{corrupt}}$ and $S_2 = (D_j)_{j \in C' \cap \mathsf{honest}} \cup (D'_i)_{i \in C' \cap \mathsf{corrupt}}$. The adversary wins if both sets are valid for their respective coalitions, and yet the output of $\mathsf{Shine.Agg}$ outputs different combined commitments.

However, for $\mathsf{Shine.Verify}(C, S_1)$ and $\mathsf{Shine.Verify}(C', S_2)$ to both output 1, then Equation 5.5 must hold. But for Equation 5.5 to hold, then S_1 and S_2 must each define a degree $t-1$ polynomial. Further, because we assume at least t honest participants, then S_1 and S_2 must agree on at minimum t points.

As such, when both $\mathsf{Shine.Verify}(C, S_1)$ and $\mathsf{Shine.Verify}(C, S_2)$ both output 1, then the sets S_1 and S_2 must define the same degree $t-1$ polynomial. Because $\mathsf{Shine.Agg}$ simply performs polynomial interpolation in the exponent to output the commitment to the constant term of the polynomial, then information-theoretically, it must be the case that $\mathsf{Shine.Agg}(S_1) = \mathsf{Shine.Agg}(S_2)$. Therefore, uniqueness likewise holds perfectly for Shine, as the advantage to the adversary to win the game is zero. □

Theorem 11. *Shine is pseudorandom in the Random Oracle Model, assuming the adversary controls at most $t-1$ participants, at least t participants are honest, Shine is verifiable, and where $(n, t) \in \mathbb{N}$ are such that $\binom{n-1}{t-1} = poly(n)$. Concretely,*

$$\mathsf{Adv}^{\mathrm{psdr}}_{\mathsf{Shine},\mathcal{A}}(\lambda, n, t, \mu) \leq q_h/q \tag{5.8}$$

where q_h is the number of times $\mathcal{A}$ is allowed to query H.

Proof. We prove Theorem 11 by a sequence of games.

Game 0. This is the peudorandomness game as defined in Figure 5.4 instantiated with Shine, when $b = 0$. Let W_0 be the event that $\mathcal{A}$ outputs $b' = 1$ in Game 0.

$\mathsf{SimGen}(w, d, \mathsf{corrupt}, n, \{\mathsf{sk}_j\}_{j\in\mathsf{corrupt}})$

$D \leftarrow g^d$

$(d_j, D_j) \leftarrow \mathsf{Shine.Gen}(j, \mathsf{sk}_j, w)$, for $j \in \mathsf{corrupt}$

// *This step is possible because Shine is verifiable,*

// *and so it is guaranteed that the correct* $(d_j, D_j), j \in \mathsf{corrupt}$ *are derived*

Define $f(x) = d \cdot L_0(x) + \sum_{j\in\mathsf{corrupt}} d_j \cdot L_j(x)$

// $L_i(x)$ *is the Lagrange polynomial defined by* $\mathsf{corrupt} \cup \{0\}$

$\mathsf{honest} = [n] \setminus \mathsf{corrupt}$

for $i \in \mathsf{honest}$ **do**

  Derive $d_i \leftarrow f(i)$

  $D_i \leftarrow g^{d_i}$

return $(D, \{(d_i, D_i)\}_{i\in\mathsf{honest}})$

Figure 5.5: Simulating algorithm SimGen for Shine.

Game 1. Let $\mathsf{sk}_{\mathsf{corrupt}} = \cup_{i\in\mathsf{corrupt}}\mathsf{sk}_i$, and let $\mathsf{sk}_{\mathsf{honest}} = \cup_{j\in\mathsf{honest}}\mathsf{sk}_j$. Then $\mathsf{sk}_{\mathsf{honest}} \setminus \mathsf{sk}_{\mathsf{corrupt}}$ is necessarily non-empty, so let $\hat{\phi}$ be such that $(\cdot, \hat{\phi}) \in \mathsf{sk}_{\mathsf{honest}} \setminus \mathsf{sk}_{\mathsf{corrupt}}$.

Note that this $\hat{\phi}$ will be random in $\mathbb{Z}_q$ and unknown to $\mathcal{A}$. Then the only difference between Game 0 and Game 1 is that if $\mathcal{A}$ queries H on any input $(\hat{\phi}, \cdot)$, the environment aborts.

Difference between Game 1 and Game 0. When $\mathcal{A}$ is allowed to make up to q_h queries to H, then the probability the environment aborts is at most q_h/q. Let W_1 be the event that $\mathcal{A}$ outputs $b' = 1$ in Game 1. Then

$$|\Pr[W_1] - \Pr[W_0]| \leq \frac{q_h}{q} \tag{5.9}$$

Game 2. This is the pseudorandomness game as in Figure 5.4, when $b = 1$, and where SimGen is as shown in Figure 5.5, along with the abort condition of Game 1.

On start, the environment initializes a table $Q \leftarrow \emptyset$. Then, when $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Gen}}$ on input (k, w), instead of performing $\mathsf{Shine.Gen}$ honestly, the environment first checks if $Q[w] \neq \perp$; if the check holds, it parses $(D, \{(d_i, D_i)\}_{i\in\mathsf{honest}}) \leftarrow Q[w]$ and returns (d_k, D_k).

Otherwise, the environment samples $d \xleftarrow{\$} \mathcal{P}$, and then obtains $(D, \{(d_i, D_i)\}_{i \in \mathsf{honest}}) \leftarrow \mathsf{SimGen}(w, d, \mathsf{corrupt}, n, \{\mathsf{sk}_j\}_{j \in \mathsf{corrupt}})$, where SimGen is defined as in Figure 5.5. Note that SimGen requires that Shine to be verifiable, and hence that $|\mathsf{honest}| \geq t$. The environment updates $Q[w] = (D, \{(d_i, D_i)\}_{i \in \mathsf{honest}})$, and then returns (d_k, D_k).

Analysis of SimGen Because Game 1 aborts on the condition that the adversary guesses the honest participants' secret key, in Game 2, the adversary must win with knowledge of only $t-1$ secret keys. However, because we assume that the number of honest parties is at least t, then the honest parties will completely determine the combined d.

Furthermore, because Shine is verifiable, then SimGen can derive the corrupt parties' contributions $(d_j, D_j), j \in \mathsf{corrupt}$ before it determines honest parties' contributions. As such, corrupt parties cannot adaptively choose their contributions after the fact to influence the resulting $D \leftarrow \mathsf{Shine.Combine}(t, \mu, \{D_i\}_{i \in C^*})$. Hence, the simulation by SimGen is perfect.

Difference between Game 2 and Game 1. In Game 1, $\mathcal{O}^{\mathsf{Gen}}(k, C, w)$ returns $(f(k), g^{f(k)})$ where $f(x) = \sum_{i \in [\delta]} \mathsf{H}(\phi_i, w) \cdot L'_{a_i}(x)$, as defined in Equation 5.3. Since $\mathsf{H}(\hat{\phi}, w)$ is unknown to $\mathcal{A}$, $f(x)$ is a random degree-$(t-1)$ polynomial that passes through $(j, y_j)_{j \in \mathsf{corrupt}}$.

In Game 2, $\mathcal{O}^{\mathsf{Gen}}(k, C, w)$ returns $(f(k), g^{f(k)})$, where $f(x) = d \cdot L_0(x) + \sum_{j \in \mathsf{corrupt}} d_j \cdot L_j(x)$. Since d is random, this is also a random degree-$(t-1)$ polynomial that passes through $(j, y_j)_{j \in \mathsf{corrupt}}$.

There is no difference between the distributions of these outputs. Let W_2 be the event that $\mathcal{A}$ outputs $b' = 1$ in Game 2. Then

$$|\Pr[W_2] - \Pr[W_1]| = 0 \tag{5.10}$$

Game 3. Game 3 is identical to Figure 5.4 instantiated with Shine, when $b = 1$.

Difference between Game 3 and Game 2. The only difference between Game 3 and Game 2 is that Game 2 has the abort condition, but Game 3 does not. Let W_3 be the event that $\mathcal{A}$ outputs $b' = 1$ in Game 3. Then as above,

$$|\Pr[W_3] - \Pr[W_2]| \leq \frac{q_h}{q} \tag{5.11}$$

Finishing the Proof. Combining Equations 5.9–5.11 yields

$$\begin{aligned}
\mathsf{Adv}^{\mathrm{psdr}}_{\mathsf{Shine},\mathcal{A}}(\lambda, n, t, \mu) &= \left|\Pr[\mathsf{Exp}^{\mathrm{psdr}}_{\mathsf{Shine},\mathcal{A}}(\lambda, n, t, \mu) = 1] - 1/2\right| \\
&= \left|\Pr[b' = 0 \wedge b = 0] + \Pr[b' = 1 \wedge b = 1] - 1/2\right| \\
&= \left|\frac{1}{2}(1 - \Pr[W_0]) + \frac{1}{2}\Pr[W_3] - \frac{1}{2}\right| \\
&= \left|\frac{1}{2}(\Pr[W_3] - \Pr[W_0])\right| \leq \frac{q_h}{q},
\end{aligned}$$

which gives Equation 5.8. This completes the proof. □

5.6 Arctic, A Deterministic and Stateless Two-Round Threshold Schnorr Signature Scheme

We now introduce Arctic, an efficient, two-round, deterministic threshold Schnorr signature scheme for moderately sized groups of participants that does not require participants to keep state between rounds of the signing protocol. As a building block, Arctic uses Shine to generate nonces deterministically, and to verify that all other participants followed the protocol honestly. Arctic is secure assuming fewer than t participants are corrupted, and at least μ participated in the signing protocol, where $\mu \leq n$ but $\mu \geq 2t - 1$.

We further require that messages exchanged between participants are sent over an authenticated channel. Arctic builds upon the verifiable pseudorandom secret sharing scheme Shine defined in Section 5.5.3, as well as Shamir's secret sharing. Verification of signatures is identical to the single-party Schnorr verification algorithm.

Remark 9 (Distributed Key Generation)**.** The Arctic construction given in Figure 5.6 assumes a centralized key generation procedure; however, using a distributed key generation (DKG) scheme is equally possible. We discuss one possible DKG in Section 5.6.2.

Remark 10 (Requirement of Authenticated Channels)**.** We require that the messages exchanged between participants in the Arctic construction shown in Figure 5.6 be sent over authenticated channels; i.e., messages must be authenticated and verifiable as having come from their purported senders. Otherwise, an adversary can simply pick contributions that are consistent with a single honest party's R_i, which would result in a valid input for

$\mathsf{Setup}(1^\lambda)$

$(\mathbb{G}, q, g) \xleftarrow{\$} \mathsf{GroupGen}(1^\lambda)$
$\mathsf{par} \leftarrow ((\mathbb{G}, q, g), \mathsf{H}_1, \mathsf{H}_2, \mathsf{H}_3)$
return par
// par *is given implicitly*
// *to all other algorithms*

$\mathsf{KeyGen}(n, t, \mu)$

// *Performed by a trusted*
// *dealer, or DKG*
if $\mu < 2t-1$ or $\mu > n$
  return $\perp$
  // *Require* $\geq 2t-1$ *signers*
$\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$; $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$
$\{(i, \mathsf{sk}_i^{(2)})\}_{i=1}^n \xleftarrow{\$} \mathsf{Shamir.Share}(\mathsf{sk}, n, t)$
$(\mathsf{sk}_i^{(1)})_{i=1}^n \xleftarrow{\$} \mathsf{Shine}[\mathsf{H}_1].\mathsf{KeyGen}(n, t, \mu)$
for $i \in \{1, \ldots, n\}$ **do**
  $\mathsf{pk}_i^{(2)} \leftarrow g^{\mathsf{sk}_i^{(2)}}$
  $\mathsf{sk}_i \leftarrow (\mathsf{sk}_i^{(1)}, \mathsf{sk}_i^{(2)}, \mathsf{pk})$
  $\mathsf{pk}_i \leftarrow (\mathsf{pk}_i^{(2)})$
return $(\mathsf{pk}, \{(\mathsf{pk}_i, \mathsf{sk}_i)\}_{i\in[n]})$

$\mathsf{Sign}_1(k, \mathsf{sk}_k, m)$

$(\mathsf{sk}_k^{(1)}, \mathsf{sk}_k^{(2)}, \mathsf{pk}) \leftarrow \mathsf{sk}_k$
$y_k \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$
$(r_k, R_k) \leftarrow \mathsf{Shine}[\mathsf{H}_1].\mathsf{Gen}(k, \mathsf{sk}_k^{(1)}, y_k)$
return (y_k, R_k)

$\mathsf{Sign}_2(k, \mathsf{sk}_k, m, C, \{(y_j, R_j)\}_{j\in C})$

return $\perp$ **if** $|C| < 2t-1$
$(\mathsf{sk}_k^{(1)}, \mathsf{sk}_k^{(2)}, \mathsf{pk}) \leftarrow \mathsf{sk}_k$
$y' \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$
for $j \in C$ **do**
  if $y_j \neq y'$
    return $\perp$
(r_k, R'_k)
    $\leftarrow \mathsf{Shine}[\mathsf{H}_1].\mathsf{Gen}(k, \mathsf{sk}_k^{(1)}, y')$
  // *Re-derive state from* Sign_1
if $k \notin C$ or $R'_k \neq R_k$
  return $\perp$
$\mathsf{input} \leftarrow (t, \mu, C, \{R_j\}_{j\in C})$
if $\mathsf{Shine}[\mathsf{H}_1].\mathsf{Verify}(\mathsf{input}) \neq 1$
  return $\perp$
$R \leftarrow \mathsf{Shine}[\mathsf{H}_1].\mathsf{Agg}(\mathsf{input})$
$c \leftarrow \mathsf{H}_3(R, \mathsf{pk}, m)$
$z_k \leftarrow r_k + c \cdot \mathsf{sk}_k^{(2)}$
return z_k

$\mathsf{Combine}(\mathsf{pk}, m, C, \{(y_j, R_j), z_j\}_{j\in C}))$

$\mathsf{input} \leftarrow (t, \mu, C, \{R_j\}_{j\in C})$
$R \leftarrow \mathsf{Shine}[\mathsf{H}_1].\mathsf{Agg}(\mathsf{input})$
$c \leftarrow \mathsf{H}_3(R, \mathsf{pk}, m)$
// λ_j *are Lagrange*
// *coefficients for* C
$z \leftarrow \sum_{j\in C} z_j \lambda_j$
if $g^z \neq R \cdot \mathsf{pk}^c$
  return $\perp$
return $\sigma = (R, z)$

Figure 5.6: Arctic, a deterministic threshold Schnorr signature scheme.

$\mathsf{Shine.Verify}$. Note that we do *not* assume the authenticated channel maintains any session identifiers or state about messages; Arctic remains secure even if the adversary were to replay old authenticated messages. However, if a participant receives an unauthenticated message, we require that the participant aborts.

5.6.1 The Construction

We now give more detail for each stage in Arctic; see Figure 5.6 for a high-level overview.

Key Generation. All participants with identifiers $i \in [n]$ begin by receiving a secret signing share $\mathsf{sk}_i^{(2)}$ and a public signing share $\mathsf{pk}_i^{(2)} = g^{\mathsf{sk}_i^{(2)}}$. In Figure 5.6, we show key generation as a centralized procedure, but a DKG can likewise be used. Each $\mathsf{sk}_i^{(2)}$ is a t-of-n Shamir secret sharing of the group's joint secret key sk; participants use these keys for signing messages. Participants also receive the public signing keys $\{\mathsf{pk}_i^{(2)}\}_{i\in[n]}$ for all other participants.

Each participant receives a secret Shine key $\mathsf{sk}_i^{(1)}$ generated by performing $\mathsf{Shine.KeyGen}$. Participants use these keys to generate nonces and commitments for each signing session. Each participant's public key share pk_i is just one public key, where $\mathsf{pk}_i = \mathsf{pk}_i^{(2)}$. Each participant's secret key share is the tuple $\mathsf{sk}_i = (\mathsf{sk}_i^{(1)}, \mathsf{sk}_i^{(2)}, \mathsf{pk})$.

Coordinator Role. Arctic assumes an external mechanism to choose the set $C \subseteq \{1, \ldots, n\}$ of signers, such that $\mu \leq |C| \leq n$. The coordinator may perform denial of service attacks, but otherwise cannot impact the security of Arctic. The coordinator can be a stand-alone entity, or may also be a signer as well.

Signing. In the first round of signing, each participant k receive as input a message m. First, each party derives $y_k \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$. To generate their nonce r_k and commitment R_k, each participant performs $(r_k, R_k) \xleftarrow{\$} \mathsf{Shine.Gen}(k, \mathsf{sk}_k^{(1)}, y_k)$. Each participant then outputs (y_k, R_k); they do not need to keep any state.

In the second round of signing, all participants again receive as input a message m, as well as a set C representing the indices of at least μ signers, where $C \subseteq [n], |C| \geq \mu$. Additionally, participants receive the list of tuples $\{(y_j, R_j)\}_{j\in C}$. First, each party re-derives $y' \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$. Then, each party checks the consistency of all other parties' views of m by checking that for each $i \in C$, $y_i = y'$. Because we require that each protocol message

is authenticated by its respective party, then this check guarantees that an adversarial player cannot split the view of honest players by sending different messages or choices of coalitions. If any check fails, the party aborts.

Otherwise, if all consistency checks succeed, each participant re-derives their nonce and commitment using Shine, again performing $(r_k, R'_k) \xleftarrow{\$} \mathsf{Shine.Gen}(k, \mathsf{sk}_k^{(1)}, y')$. Then, the participant checks its identifier is in the coalition C and that $R'_k = R_k$, aborting if either check does not hold. Then, each participant verifies that all other participants followed the protocol to derive their commitment, by checking $\mathsf{Shine.Verify}(t, \mu, C, (R_j)_{j \in C})$. If the check fails, they abort the protocol.

Finally, if all of the above checks pass, each participant $k \in C$ will then derive the group commitment $R \leftarrow \mathsf{Shine.Agg}(t, \mu, C, \{R_j\}_{j \in C})$, and the challenge $c \leftarrow \mathsf{H}_3(R, \mathsf{pk}, m)$. Finally, each participant derives their signature share $z_k \leftarrow r_k + c \cdot \mathsf{sk}_k^{(2)}$. Each participant outputs z_k as its output for Sign_2.

Combination and Verification. To perform the $\mathsf{Combine}$ algorithm, the group commitment R is first derived using values output by participants from Sign_1. Then, the response z is derived by finding $z \leftarrow \sum_{j \in C} z_j \lambda_j$, where the λ_j are the Lagrange coefficients for the set C. The output from $\mathsf{Combine}$ is the Schnorr signature $\sigma = (R, z)$, which can be verified using the single-party Schnorr verification algorithm given in Section 2.3.1.

5.6.2 Possible Extensions

We now discuss two possible extensions to Arctic.

Robustness. Arctic as currently defined is not robust; if any party submits invalid commitments, then the output from $\mathsf{Shine.Agg}$ cannot be used. However, it is possible to extend Shine to be robust, therefore also ensuring that Arctic can likewise be extended. To do so in a secure manner, the robust extension would require additional players, along with a protocol to come to consensus about which players misbehaved.

In particular, by requiring that the minimum number of participants μ be of size at least $\mu \geq 3t - 2$, Shine and Arctic can be securely used in a robust manner. The requirement that $\mu \geq 3t - 2$ is referred to as the *honest supermajority* setting. We give further details on how robustness can be achieved in Section 5.7. In this setting, $\mathsf{Shine.Verify}$ can both detect any inconsistencies as well as identify the misbehaving players. This property could

likewise allow for extending Arctic to support robustness, under the same assumption that $\mu \geq 3t - 2$.

Distributed Key Generation. To define the signing keys $(\mathsf{pk}^{(2)}, (\mathsf{sk}_i^{(2)})_{i=1}^n)$, any DKG that Shamir secret shares a discrete logarithm secret key could be employed, such as the three-round DKG by Gennaro et al.[GJKR07].

As one possibility to perform $\mathsf{Shine}[\mathsf{H}_1].\mathsf{KeyGen}$ in a distributed manner, each subset of $n-t+1$ parties could assign one representative to generate a secret PRF key uniformly at random for that subset. The representative would then share this PRF key with all parties in their subset. To ensure security against fully malicious adversaries, parties within each subset could perform a subsequent broadcast round to compare their received inputs, to ensure consistency.

5.6.3 Correctness and Security

We next demonstrate the correctness and security of Arctic.

5.6.3.1 Correctness

In the first round of signing, participants will output nonces and commitments $(r_i, R_i) \leftarrow \mathsf{Shine.Gen}(i, \mathsf{sk}_i^{(1)}, y_i)$, for $i \in C$, where $y_i \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$, and $R_i = g^{r_i}$.

In the second round of signing, each participant receives the set of tuples $\{(y_j, R_j)\}_{j \in C}$. After deriving $y' \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$, then the check that $y_j = y'$ for each $j \in C$ will succeed.

Because Shine is correct, then $\mathsf{Shine.Verify}(t, \mu, C, \{R_j\}_{j \in C})$ will output 1 and the same group commitment $R = \prod_{j \in C} R_j^{\lambda_j}$ will be computed regardless of the choice of C. Because $\mathsf{Shine.Gen}$ is deterministic, then after deriving $y \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$, $\mathsf{Shine.Gen}(k, \mathsf{sk}_k^{(1)}, y)$ will output the same (r_k, R_k) as derived in the first round of signing.

After deriving (r_k, R_k), all signers in a coalition C output valid signature shares z_k with respect to R and challenge $c = \mathsf{H}_3(R, \mathsf{pk}, m)$, where $z_k = r_k + c \cdot \mathsf{sk}_k^{(2)}$. The aggregated signature is then $\sigma = (R, z)$, where $z = \sum_{j \in C} z_j \cdot \lambda_j$. Because $\mathsf{sk} = \sum_{j \in C} \mathsf{sk}_j^{(2)} \lambda_j$, $\mathsf{pk} = g^{\mathsf{sk}} = g^{\sum_{j \in C} \mathsf{sk}_j^{(2)} \lambda_j}$, and $R = \prod_{j \in C} R_j^{\lambda_j} = g^{\sum_{j \in C} r_j \cdot \lambda_j}$, we have that $g^z = R \cdot \mathsf{pk}^c$, as required.

5.6.3.2 Unforgeability

We next demonstrate the static (weak) unforgeability of Arctic, via Theorem 12. We believe it is possible to additionally prove that Arctic is strongly unforgeable, but leave this topic for future work.

Theorem 12. *Arctic is unforgeable in the ROM against a PPT adversary $\mathcal{A}$ playing the static unforgeability game as shown in Figure 5.1 against Arctic, assuming $\mathcal{A}$ can make up to $t-1$ corruptions, the number of honest parties is at least t, the discrete logarithm assumption, participants exchange messages over an authenticated channel, Shine is a secure VPSS, and where $(n,t) \in \mathbb{N}$ are such that $\binom{n-1}{t-1} = poly(n)$.*

Concretely, let $\mathsf{Adv}^{\mathrm{dl}}_{\mathcal{D}}(\lambda)$ be the advantage of an adversary $\mathcal{D}$ against the discrete logarithm assumption. The advantage of $\mathcal{A}$ is bounded by

$$\mathsf{Adv}^{\mathrm{uf}}_{\mathsf{Arctic},\mathcal{A}}(\lambda,n,t,\mu) \leq \sqrt{q_r \mathsf{Adv}^{\mathrm{dl}}_{\mathcal{D}}(\lambda) + \frac{2(q_1+q_3)}{q} + \frac{3q_r^2}{q}}$$

where $\mu \geq 2t-1$ and $n \geq \mu$, and where $q_r = q_2+q_3+2q_s+1$, such that q_s is the number of times $\mathcal{A}$ is allowed to query the signing oracles, q_1 is the number of times $\mathcal{A}$ is allowed to query H_1, q_2 is the number of times that $\mathcal{A}$ is allowed to query H_2, and q_3 is the number of times $\mathcal{A}$ is allowed to query H_3.

To prove Theorem 12, we rely on two lemmas.

Lemma 6. *There exists an algorithm SIM which can simulate Arctic with respect to a discrete logarithm challenge to an adversary $\mathcal{A}$ playing against the unforgeability game as in Figure 5.1. In other words, $\mathcal{A}$ has negligible additional advantage in winning the unforgeability game when interacting with SIM than it does against Arctic directly.*

Concretely, in the honest majority setting with authenticated channels, then

$$\mathsf{Adv}^{\mathrm{uf}}_{\mathsf{Arctic},\mathcal{A}}(\lambda,n,t,\mu) \leq \mathsf{Adv}^{\mathrm{uf}}_{\mathsf{SIM},\mathcal{A}}(\lambda,n,t,\mu) + \frac{(q_1+q_3)}{q} + \frac{q_r^2}{2q} \tag{5.12}$$

where $\mu \geq 2t-1$ and $n \geq \mu$, and where (n,t) are such that $\binom{n-1}{t-1} = poly(n)$.

Lemma 7. *Given any PPT adversary $\mathcal{A}$ that wins the unforgeability game against Arctic and a simulating algorithm SIM which can simulate Arctic, then there exists a PPT DL adversary $\mathcal{D}$ that can use $\mathcal{A}$ as a subroutine to solve for its DL challenge.*

Concretely, in the honest majority setting with authenticated channels,

$$\mathsf{Adv}^{\mathrm{dl}}_{\mathcal{D}}(\lambda) \geq \frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathsf{Arctic},\mathcal{A}}(\lambda, n, t, \mu)^2}{q_r} - \frac{2(q_1 + q_3)}{q} - \frac{3q_r^2}{q} \tag{5.13}$$

where $\mu \geq 2t - 1$ *and* $n \geq \mu$*, and* $q_r = q_2 + q_3 + 2q_s + 1$*, and where* $(n, t) \in \mathbb{N}$ *are such that* $\binom{n-1}{t-1} = poly(n)$*.*

Proof Intuition. We give the proof in two parts. First, Lemma 6 shows that a simulating PPT algorithm SIM exists which can simulate Arctic to an unforgeability PPT adversary $\mathcal{A}$ with respect to a public key pk for which SIM does not know the corresponding secret key. Recall that we give the unforgeability game in Figure 5.1. We show that the advantage of $\mathcal{A}$ with respect to SIM is negligibly different than its advantage against Arctic.

We prove Lemma 6 using a series of game hops, showing that because Shine is verifiable, unique, and pseudorandom, then SIM can simulate a signing session for a message m chosen by $\mathcal{A}$ with respect to pk. in a way that is indistinguishable to $\mathcal{A}$ from a real execution of the signing protocol. SIM does this by sampling (z, c) uniformly at random in $\mathcal{O}^{\mathsf{Sign}_1}$, deriving R as $R = g^z \cdot \mathsf{pk}^{-c}$, and then programming $\mathsf{H}_3(R, \mathsf{pk}, m)$ on c. Then, to simulate each honest party's commitment $R_i, i \in \mathsf{honest}$, SIM uses its knowledge of corrupt parties' Shine keys $\mathsf{sk}_j^{(1)}, j \in \mathsf{corrupt}$ to derive corrupt parties' nonces and commitments, by performing $(r_j, R_j) \leftarrow \mathsf{Shine.Gen}(j, \mathsf{sk}_j^{(1)}, y)$. SIM can then correctly derive each honest party's commitment $R_k, k \in \mathsf{honest}$ by performing polynomial interpolation in the exponent, with respect to R and the set $(R_j)_{j \in \mathsf{corrupt}}$.

Because Shine is verifiable, if $\mathcal{A}$ submits commitments $(R'_j)_{j \in C \cap \mathsf{corrupt}} \neq (R_j)_{j \in C \cap \mathsf{corrupt}}$ to $\mathcal{O}^{\mathsf{Sign}_2}$, SIM will output $\perp$ with the same probability as in the real scheme. Because Shine is unique, then R is fixed with respect to $y \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$, regardless of $\mathcal{A}$'s choice of coalition in the second round. Finally, because Shine is pseudorandom, then unless $\mathcal{A}$ can guess $\hat{\phi}$, where $\hat{\phi}$ is an honest party's portion of the replicated secret defined in $\mathsf{Shine.KeyGen}$, then SIM can simulate each honest party's $R_i, i \in \mathsf{honest}$ with negligible advantage to $\mathcal{A}$. We show that in order for $\mathcal{A}$ to break the pseudorandomness of Shine, it must guess a preimage of H_1, which it can do with negligible advantage less than q_1/q, where q_1 is the number of queries $\mathcal{A}$ is allowed to make to H_1.

If $\mathcal{A}$ wins the unforgeability experiment against Arctic, $\mathcal{A}$ will output a forgery $\sigma^* = (R^*, z^*)$ corresponding to a message m^* that is valid under a challenge public key pk. Lemma 7 shows that given a successful PPT adversary $\mathcal{A}$, a PPT reduction $\mathcal{D}$ exists against the Discrete Logarithm assumption. In particular, we show that $\mathcal{D}$ can use $\mathcal{A}$

and SIM to solve for its discrete logarithm challenge. $\mathcal{D}$ accepts a group element $Y \in \mathbb{G}$ from its discrete logarithm challenger, and its goal is to output y such that $Y = g^y$. It uses the modified forking algorithm $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ shown in Figure 2.2 on the instance $X = (\mathsf{par}, Y, n, t, \mu)$, where n, t, μ are chosen by $\mathcal{D}$. Fork_m runs SIM twice, the first time on X and the set $\{h_1, \ldots, h_{q_r}\}$, which is the set of random oracle outputs that SIM will use to program its random oracle. On the second execution of SIM, SIM is again given X and the set $\{h_1, \ldots, h_{j-1}, h'_j, \ldots, h'_{q_r}\}$, where h_j corresponds to the challenge $c^* = H(R^*, \mathsf{pk}, m^*)$ for the forgery σ^* output by $\mathcal{A}$ in its first execution. Corollary 1 lower-bounds the probability that $\mathcal{A}$ will output a second forgery $\sigma^{**} = (R^*, z^{**})$ on the same random oracle query $c^{**} = H(R^*, \mathsf{pk}, m^*)$ as in its first iteration, where the programmed outputs $c^* = h_j, c^{**} = h'_j$ are such that $c^* \neq c^{**}$. $\mathcal{D}$ can use the two forgeries output by $\mathcal{A}$ in its first and second execution to then solve for the discrete logarithm solution $y = \frac{z^* - z^{**}}{c^* - c^{**}}$.

Proof of Lemma 6. We prove Lemma 6 by a sequence of games.

Game 0. This is the unforgeability game as defined in Figure 5.1, instantiated with Arctic. Let $\mathcal{A}$ be an adversary playing against the unforgeability game.

Let W_0 be the event that $\mathcal{A}$ wins Game 0. Then, the advantage of $\mathcal{A}$ is simply

$$\mathsf{Adv}^{\mathrm{uf}}_{\mathsf{Arctic},\mathcal{A}}(\lambda) = \Pr[W_0] \tag{5.14}$$

Game 1. The only difference between Game 0 and Game 1 is as follows. If $\mathcal{A}$ queries H_2 or H_3 on any input that produces a colliding output with another query, then the environment aborts.

Difference between Game 1 and Game 0. Let W_1 be the event that $\mathcal{A}$ wins Game 1. Because we model H_2 and H_3 as random oracles, then

$$|\Pr[W_1] - \Pr[W_0]| \leq \frac{q_r^2}{2q} \tag{5.15}$$

Game 2. The only difference between Game 2 and Game 1 is as follows. If $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_2}$ on an input (k, m, C, S_1) where $S_1 = (y_i, R_i)_{i \in C}$, such that the following relation holds:

$$\begin{aligned} y \leftarrow \mathsf{H}_2(\mathsf{pk}, m);\ y = y_i, i \in C,\ \text{and} \\ |C \cap \mathsf{honest}| \geq t, \\ \mathsf{Shine.Gen}(i, \mathsf{sk}_i^{(1)}, y) = (r'_i, R'_i) \quad \forall i \in C \text{ and} \\ \mathsf{Shine.Verify}(t, \mu, C, (R_j)_{j \in C}) = 1,\ \text{but} \\ S_1 \neq (y_i, R'_i)_{i \in C} \end{aligned} \tag{5.16}$$

then the environment aborts.

Reduction to Verifiability of Shine. We now define a reduction $\mathcal{B}_2$ that uses $\mathcal{A}$ as a subroutine when it plays against the verifiability game of Shine.

To begin, $\mathcal{B}_2$ receives as input from the VPSS verifiability game as shown in Figure 5.3 the set of Shine secret keys $(\mathsf{sk}_i^{(1)})_{i \in \mathsf{corrupt}}$. It honestly follows the protocol to generate signing key material, and then sets $\mathsf{sk}_i \leftarrow (\mathsf{sk}_i^{(1)}, \mathsf{sk}_i^{(2)}, \mathsf{pk})$ for all $i \in \mathsf{corrupt}$. It then outputs to the adversary $\mathcal{A}$ the key material $(\mathsf{pk}, (\mathsf{pk}_j)_{j \in [n]}, (\mathsf{sk}_i)_{i \in \mathsf{corrupt}})$.

To simulate signing, when $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_1}$ on input (k, m), $\mathcal{B}_2$ derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$ honestly. $\mathcal{B}_2$ then queries its own VPSS verifiability oracle $\mathcal{O}^{\mathsf{Gen}}$ on (k, y) receiving (d_k, D_k) as the output. It sets $R_k = D_k$, and outputs (y, R_k).

When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_2}$ on input $(k, m, C, (y_j, R'_j)_{j \in C})$, $\mathcal{B}_2$ first performs $\mathsf{Shine.Verify}$ on the inputs, outputting $\perp$ if the input is invalid. Otherwise, $\mathcal{B}_2$ then derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$ honestly.

$\mathcal{B}_2$ then derives each corrupted party's nonce and commitment honestly, by performing $(r'_i, R'_i) \leftarrow \mathsf{Shine.Gen}(i, \mathsf{sk}_i^{(1)}, y)$, for $i \in \mathsf{corrupt} \cap C$. If any $(R_i \neq R'_i)$, for $i \in \mathsf{corrupt} \cap C$, then $\mathcal{B}_2$ terminates the execution of $\mathcal{A}$. $\mathcal{B}_2$ then submits $(y, C, (R_i)_{i \in \mathsf{corrupt} \cap C})$ as its own output to the VPSS verifiability experiment.

Otherwise, $\mathcal{B}_2$ simply simulates by querying $\mathcal{O}^{\mathsf{Gen}}$ for each honest party $\mathsf{honest} \cap C$ on input y, to re-derive (r_j, R_j) for $j \in \mathsf{honest} \cap C$. $\mathcal{B}_2$ then follows the remainder of the protocol honestly, outputting its signature share z_k.

The simulation by $\mathcal{B}_2$ is perfect because the simulation by the VPSS oracle $\mathcal{O}^{\mathsf{Gen}}$ is perfect. Further, the early termination of $\mathcal{A}$ has no consequence on the indistinguishability of the simulation.

Difference between Game 2 and Game 1. The additional advantage to $\mathcal{A}$ in Game 2 is upper-bounded by the advantage that $\mathcal{B}_2$ wins its VPSS verifiability game against Shine. Let W_2 be the event that $\mathcal{A}$ wins in Game 2, and let $\mathsf{Adv}^{\text{verf}}_{\mathsf{Shine},\mathcal{B}_2}$ be the advantage that $\mathcal{B}_2$ has in the verifiability experiment against Shine. Because Shine is perfectly verifiable in the honest majority setting, and Arctic likewise assumes an honest majority then

$$|\Pr[W_2] - \Pr[W_1]| \leq \mathsf{Adv}^{\text{verf}}_{\mathsf{Shine},\mathcal{B}_2} = 0 \tag{5.17}$$

Game 3. The only difference between Game 2 and Game 3 is as follows. If $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_2}$ on the inputs (m, C, S) and (m, C', S') such that $S = (y, R_i)_{i \in C}$ and $S' = (y, R'_i)_{i \in C'}$ and $(C, S) \neq (C', S')$, but the following relation holds:

$$\begin{aligned} y &\leftarrow \mathsf{H}_2(\mathsf{pk}, m) \\ (r_j, R_j) &\leftarrow \mathsf{Shine.Gen}(j, \mathsf{sk}_j^{(1)}, y), j \in \mathsf{honest} \cap (C \cup C') \\ |C \cap \mathsf{honest}| &\geq t, \quad |C' \cap \mathsf{honest}| \geq t \\ \mathsf{Shine.Verify}(t, \mu, C, (R_i)_{i \in C}) &= 1, \mathsf{Shine.Verify}(t, \mu, C', (R'_i)_{i \in C}) = 1, \text{ but} \\ \mathsf{Shine.Agg}(t, \mu, C, (R_i)_{i \in C}) &\neq \mathsf{Shine.Agg}(t, \mu, C', (R'_i)_{i \in C'}) \end{aligned} \tag{5.18}$$

then the environment aborts.

Reduction to Uniqueness of Shine. We now define a reduction $\mathcal{B}_3$ that uses $\mathcal{A}$ as a subroutine when it plays against the uniqueness game of Shine.

To begin, $\mathcal{B}_3$ receives as input from the VPSS uniqueness experiment the set of Shine secret keys $(\mathsf{sk}_i^{(1)})_{i \in \mathsf{corrupt}}$. $\mathcal{B}_3$ then generates a table Q to maintain queries by $\mathcal{A}$ to $\mathcal{O}^{\mathsf{Sign}_2}$. It honestly follows the protocol to generate signing key material, and then sets $\mathsf{sk}_i \leftarrow (\mathsf{sk}_i^{(1)}, \mathsf{sk}_i^{(2)}, \mathsf{pk})$ for all $i \in \mathsf{corrupt}$. It then outputs to the adversary $\mathcal{A}$ the key material $(\mathsf{pk}, (\mathsf{pk}_j)_{j \in [n]}, (\mathsf{sk}_i)_{i \in \mathsf{corrupt}})$.

To simulate signing, when $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_1}$ on input (k, m), $\mathcal{B}_3$ first derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$ honestly. It then queries its own VPSS uniqueness oracle $\mathcal{O}^{\mathsf{Gen}}$ on the input (k, y), receiving (d_k, D_k) as the output. It sets $R_k = D_k$, and then outputs (y, R_k).

When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_2}$ on input $(k, m, C, (y_j, R_j)_{j \in C})$, $\mathcal{B}_3$ again derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$ honestly. After checking that $y = y_i$ for each i and that $k \in C$, it then updates Q to be

$Q \leftarrow Q \cup \{(m, C, y, \{R_j\}_{j \in C})\}$. $\mathcal{B}_3$ then checks Q to see if any two entries $(m, C, y, \{R_j\}_{j \in C})$, $(m, C', y, \{R'_j\}_{j \in C'})$ exist such that Equation 5.18 holds.

If the check succeeds, then $\mathcal{B}_3$ terminates the execution of $\mathcal{A}$. $\mathcal{B}_3$ then outputs $(y, (C, (R_j)_{j \in C \cap \mathsf{corrupt}}), (C', (R'_j)_{j \in C' \cap \mathsf{corrupt}}))$ as its own output to the VPSS uniqueness game.

If the check fails, then $\mathcal{B}_3$ again queries its own VPSS uniqueness oracle $\mathcal{O}^{\mathsf{Gen}}$ on the input (k, y), receiving (d_k, D_k) as the output. $\mathcal{B}_3$ then sets $r_k = d_k$ and $R_k = D_k$. It follows the rest of the protocol honestly.

The simulation by $\mathcal{B}_3$ is perfect because the simulation by the VPSS oracle $\mathcal{O}^{\mathsf{Gen}}$ is perfect. Further, the early termination of $\mathcal{A}$ has no consequence on the indistinguishability of the simulation.

Difference between Game 3 and Game 2. The additional advantage to $\mathcal{A}$ in Game 2 is upper-bounded by the advantage that $\mathcal{B}_3$ wins its VPSS uniqueness game against Shine. Let W_3 be the event that $\mathcal{A}$ wins in Game 3, and let $\mathsf{Adv}^{\mathrm{unq}}_{\mathcal{B}_3, \mathsf{Shine}}$ be the advantage that $\mathcal{B}_3$ has in the verifiability experiment against Shine. Because Shine is perfectly unique in the honest majority setting with authenticated channels, then

$$|\Pr[W_3] - \Pr[W_2]| \leq \mathsf{Adv}^{\mathrm{unq}}_{\mathcal{B}_3, \mathsf{Shine}} = 0 \tag{5.19}$$

Game 4. The only differences between Game 4 and Game 3 are as follows. On initialization, the environment initializes a table $Q \leftarrow \emptyset$.

Let $\mathsf{sk}^{(1)}_{\mathsf{corrupt}} = \cup_{i \in \mathsf{corrupt}} \mathsf{sk}^{(1)}_i$, and let $\mathsf{sk}^{(1)}_{\mathsf{honest}} = \cup_{j \in \mathsf{honest}} \mathsf{sk}^{(1)}_j$. Let $\hat{\phi}$ be such that $(\cdot, \hat{\phi}) \in \mathsf{sk}^{(1)}_{\mathsf{honest}} \setminus \mathsf{sk}^{(1)}_{\mathsf{corrupt}}$. If $\mathcal{A}$ queries H_1 on $\hat{\phi}$, then the environment aborts.

Otherwise, when $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_1}$ on party $k \in \mathsf{honest}$ and message m, instead of producing each party's nonce using Shine, the environment first derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$ honestly. It then checks if $Q[y] \neq \perp$; if so, it parses $(r_i, R_i)_{i \in \mathsf{honest}} \leftarrow Q[y]$, and returns (y, R_k). Otherwise, it does the following:

1. Sample $r \xleftarrow{\$} \mathbb{Z}_q$.
2. Use the algorithm SimGen as defined by Shine in Figure 5.5 to generate $(R, \{(r_i, R_i)\}_{i \in \mathsf{honest}}) \leftarrow \mathsf{SimGen}(y, r, \mathsf{corrupt}, n, \{\mathsf{sk}^{(1)}_i\}_{i \in \mathsf{corrupt}})$.
3. Set $Q[y] = (r_i, R_i)_{i \in \mathsf{honest}}$.

4. Return (y, R_k).

When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_2}$ on party $k \in \mathsf{honest}$, message m, coalition C, and set of commitments $(R_i)_{i \in C}$, the environment first derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$ honestly. It then checks if $Q[y] = \perp$; if so, it returns $\perp$. Otherwise, it parses $(r_i, R_i)_{i \in \mathsf{honest}} \leftarrow Q[y]$. It follows the protocol honestly to derive z_k, and outputs z_k.

Reduction to Pseudorandomness of Shine. We now define a reduction $\mathcal{B}_4$ that uses $\mathcal{A}$ as a subroutine when it plays against the pseudorandomness game of Shine. We assume without loss of generality that $|\mathsf{corrupt}| = t - 1$.

On initialization, $\mathcal{B}_4$ receives as input from the VPSS pseudorandomness experiment the set of Shine secret keys $(\mathsf{sk}_i^{(1)})_{i \in \mathsf{corrupt}}$. It honestly follows the protocol to generate signing key material, and then sets $\mathsf{sk}_i \leftarrow (\mathsf{sk}_i^{(1)}, \mathsf{sk}_i^{(2)}, \mathsf{pk})$ for all $i \in \mathsf{corrupt}$. It then outputs to the adversary $\mathcal{A}$ the key material $(\mathsf{pk}, (\mathsf{pk}_j)_{j \in [n]}, (\mathsf{sk}_i)_{i \in \mathsf{corrupt}})$.

When $\mathcal{A}$ queries H_1 on inputs (ϕ_i, y), $\mathcal{B}_4$ does the following:

1. Sets k be the identifier of an honest party. Note that all of $\mathsf{sk}_k^{(1)}$ is known to $\mathcal{B}_4$ except the entry $(a_i, \hat{\phi})$ where $a_i = \mathsf{corrupt}$.
2. Queries $\mathcal{O}^{\mathsf{Gen}}$ on input (k, y), receiving in return (d_k, D_k).
3. Checks if the following relation holds:

$$\mathsf{H}_1(\phi_i, y) \stackrel{?}{=} d_k - \frac{\sum_{(a_\ell, \phi_\ell) \in \mathsf{sk}_k^{(1)}, \ell \neq i} \mathsf{H}_1(\phi_\ell, y) \cdot L'_{a_\ell}(k)}{L'_{a_i}(k)} \tag{5.20}$$

   where $L'_{a_\ell}(x)$ and $L'_{a_i}(x)$ are defined in Equation 5.2.

4. If the relation holds, then $\mathcal{B}_4$ outputs 0.

Otherwise, when $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_1}$ on input (k, m), $\mathcal{B}_4$ first derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$ honestly. It then queries its VPSS pseudorandomness oracle $\mathcal{O}^{\mathsf{Gen}}$ on input (k, y), receiving in return (d_k, D_k). It sets $r_k = d_k$ and $R_k = D_k$. $\mathcal{B}_4$ then outputs (y, R_k).

When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_2}$ on input $(k, m, C, \{(y_i, R_i)\}_{i \in C})$, $\mathcal{B}_4$ does the following:

1. Checks if $|C \cap \mathsf{corrupt}| < t$, if the check does not hold, $\mathcal{B}_4$ aborts.

2. Otherwise, $\mathcal{B}_4$ derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$ honestly.

3. It then checks that $y = y_i$ for all $i \in C$, and that $k \in C$. If any check fails, $\mathcal{B}_4$ outputs $\perp$.

4. It then queries its VPSS pseudorandomness oracle $\mathcal{O}^{\mathsf{Gen}}$ again on input (k, y), re-obtaining (d_k, D_k).

5. It then follows the remainder of the protocol honestly, deriving $z_k = r_k + c \cdot \mathsf{sk}_k^{(2)}$.

6. $\mathcal{B}_4$ then outputs to $\mathcal{A}$ the signature share z_k for party k as its output.

Finally, when $\mathcal{A}$ terminates and outputs its forgery (R^*, z^*), $\mathcal{B}_4$ outputs 1, along with (R^*, z^*) and the index associated with c^*.

Difference between Game 4 and Game 3. Because we assume $|\mathsf{corrupt}| < t$, $\mathcal{B}_4$ will abort with zero probability. The additional advantage to $\mathcal{A}$ in Game 4 is thus upper-bounded by the advantage that $\mathcal{B}_4$ wins its VPSS pseudorandomness game against Shine. In particular, when the pseudorandomness game bit $b = 0$, then $\mathcal{B}_4$'s simulation is identical to that of Game 3. and if $b = 1$, then it is identical to Game 4.

Let W_4 be the event that $\mathcal{A}$ wins in Game 4, and let $\mathsf{Adv}^{\mathrm{psdr}}_{\mathcal{B}_4,\mathsf{Shine}}$ be the advantage that $\mathcal{B}_4$ wins the pseudorandomness experiment against Shine. Then

$$|\Pr[W_4] - \Pr[W_3]| \leq \mathsf{Adv}^{\mathrm{psdr}}_{\mathcal{B}_4,\mathsf{Shine}} \leq \frac{q_1}{q} \tag{5.21}$$

where q_1 is the number of times $\mathcal{A}$ is allowed to query H_1.

Game 5. We now describe a simulating algorithm SIM that simulates Arctic to $\mathcal{A}$, with respect to a challenge group element $Y \in \mathbb{G}$ for which SIM does not know the corresponding discrete logarithm.

Setup. SIM accepts as input an instance X, which is a tuple consisting of public parameters $(\mathbb{G}, q, p)$, a discrete logarithm (DL) challenge $Y \in \mathbb{G}$, and the total number of parties n, the corruption threshold t, and the minimum number of signing parties μ. In addition, SIM accepts as input a set of $q_r = q_2 + q_3 + 2q_s + 1$ hash values $\{h_1, \ldots, h_{q_r}\}$.

Next, SIM initializes the following values:

- Initializes the table $\mathsf{Q_q} \leftarrow \emptyset$ to maintain signatures it issued for honest participants during its simulation, and the table $\mathsf{Q_m} \leftarrow \emptyset$ to maintain the set of messages queried by $\mathcal{A}$ to $\mathcal{O}^{\mathsf{Sign}_2}$.
- Initializes the tables $\mathsf{Q_1}, \mathsf{Q_2}, \mathsf{Q_3} \leftarrow \emptyset^3$ to manage random oracle queries and responses.
- Initializes the table $\mathsf{Q_y} \leftarrow \emptyset$ to manage state between signing queries.
- Initializes the counter $\mathsf{k_r} \leftarrow 1$ to track issued random oracle outputs.

SIM then runs $\mathcal{A}(\mathsf{par}, n, t, \mu)$. $\mathcal{A}$ outputs the set of corrupted parties $\mathsf{corrupt}$ such that $|\mathsf{corrupt}| \leq t-1$, as well as its internal state st_A. SIM sets $\mathsf{honest} \leftarrow [n] \setminus \mathsf{corrupt}$ and must reveal the secret keys of the corrupted parties to $\mathcal{A}$, which it does in the next step.

Simulating KeyGen. To simulate key generation where SIM is the trusted dealer, SIM does the following:

1. It sets $\mathsf{pk} = Y$.
2. To simulate key generation for the signing keys, SIM first picks $t-1$ secret key shares for the corrupt parties $(\mathsf{sk}_i^{(2)})_{i \in \mathsf{corrupt}} \xleftarrow{\$} \mathbb{Z}_q$, and defines their corresponding public keys $\mathsf{pk}_i^{(2)} \leftarrow g^{\mathsf{sk}_i^{(2)}}$ for all $i \in \mathsf{corrupt}$.
3. SIM then derives the public key shares $\mathsf{pk}_j^{(2)}, j \in \mathsf{honest}$ for the remaining honest parties in the exponent, as follows:

$$\mathsf{pk}_j^{(2)} = (\mathsf{pk}^{(2)})^{L_0(j)} \cdot \prod_{i \in \mathsf{corrupt}} g^{\mathsf{sk}_i^{(2)} L_i(j)} \tag{5.22}$$

   where $L_i(x)$ is the i^{th} Lagrange polynomial defined by $\mathsf{corrupt} \cup \{0\}$.
4. To simulate key generation for the Shine keys, SIM simply performs key generation honestly, deriving $(\mathsf{sk}_1^{(1)}, \ldots, \mathsf{sk}_n^{(1)}) \xleftarrow{\$} \mathsf{Shine.KeyGen}(n, t)$.
5. SIM then sets $\mathsf{sk}_j = (\mathsf{sk}_j^{(1)}, \mathsf{sk}_j^{(2)}, \mathsf{pk})$, for $j \in \mathsf{corrupt}$, and $\mathsf{pk}_i = (\mathsf{pk}_i^{(2)})$, for $i \in [n]$.

Then, SIM runs $\mathcal{A}^{\mathcal{O}^{\mathsf{Sign}_1, \mathsf{Sign}_2}}(\mathsf{st}_A, \mathsf{pk}, \{\mathsf{pk}_j\}_{j \in [n]}, \{\mathsf{sk}_i\}_{i \in \mathsf{corrupt}})$, and responds to its oracle queries, as we describe next.

Simulating Random Oracles. To simulate $\mathcal{A}$'s random oracle queries, SIM simply performs lazy sampling. It programs H_1 with values sampled from its own random tape; however, it programs $\mathsf{H}_2, \mathsf{H}_3$ with values from its input $\{h_1, \ldots, h_{q_r}\}$, as follows.

- $\underline{\mathsf{H}_1:}$ When $\mathcal{A}$ queries H_1 on inputs (ϕ_i, y), SIM does the following:
  1. Checks if $\phi_i = \hat{\phi}$, where $\hat{\phi}$ is such that $(\cdot, \hat{\phi}) \in \mathsf{sk}^{(1)}_{\mathsf{honest}} \setminus \mathsf{sk}^{(1)}_{\mathsf{corrupt}}$, and where $\mathsf{sk}^{(1)}_{\mathsf{corrupt}} = \cup_{i \in \mathsf{corrupt}} \mathsf{sk}^{(1)}_i$, and $\mathsf{sk}^{(1)}_{\mathsf{honest}} = \cup_{j \in \mathsf{honest}} \mathsf{sk}^{(1)}_j$.
  2. If this check holds, SIM aborts. We refer to this bad event as $\mathsf{BadEvent}_1$.
  3. Otherwise, SIM checks to see if $\mathsf{Q}_1[(\phi_i, y)] \neq \perp$, if the check holds, it returns $\mathsf{Q}_1[(\phi_i, y)]$,
  4. Otherwise, it samples $\gamma \xleftarrow{\$} \mathbb{Z}_q$ (using its own random tape).
  5. It then sets $\mathsf{Q}_1[(\phi_i, y)] = \gamma$.
  6. Finally, SIM returns γ.
- $\underline{\mathsf{H}_2:}$ When $\mathcal{A}$ queries H_2 on inputs (pk, m), SIM first checks to see if $\mathsf{Q}_2[(\mathsf{pk}, m)] \neq \perp$, If so, it returns $\mathsf{Q}_2[(\mathsf{pk}, m)]$. Otherwise, it sets $y \leftarrow h_{\mathsf{k_r}}$, and increments $\mathsf{k_r} \leftarrow \mathsf{k_r} + 1$. It then sets $\mathsf{Q}_2[(\mathsf{pk}, m)] = y$. Finally, SIM returns y.
- $\underline{\mathsf{H}_3:}$ When $\mathcal{A}$ queries H_3 on inputs (R, pk, m), SIM checks to see if $\mathsf{Q}_3[(R, \mathsf{pk}, m)] \neq \perp$. If so, it returns $\mathsf{Q}_3[(R, \mathsf{pk}, m)]$. Otherwise, it sets $c \leftarrow h_{\mathsf{k_r}}$, and increments $\mathsf{k_r} \leftarrow \mathsf{k_r} + 1$. it then sets $\mathsf{Q}_3[(R, \mathsf{pk}, m)] = c$. Finally, SIM returns c.

Simulating Signing Oracles. To simulate signing oracles, SIM does as follows:

- $\underline{\mathsf{Sign}_1(k, m)}$: When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_1}$ on participant identifier k and message m, SIM does the following:
  1. If $k \notin \mathsf{honest}$, output $\perp$.
  2. Derives $y \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$ honestly.
  3. SIM then checks to see if $\mathsf{Q_y}[y] \neq \perp$. If so,
     (a) SIM parses $(m', z, c, R, (R_j, z_j)_{j \in [n]}) \leftarrow \mathsf{Q_y}[y]$.
     (b) SIM then returns (y, R_k).

4. However, if this is the first time that $\mathcal{A}$ has queried $\mathcal{O}^{\mathsf{Sign}_1}$ on m, SIM then samples $z \xleftarrow{\$} \mathbb{Z}_q$, and sets $c \leftarrow h_{\mathsf{k_r}}$. It then increments the counter $\mathsf{k_r} \leftarrow \mathsf{k_r} + 1$.
5. SIM then derives $R \leftarrow g^z \cdot \mathsf{pk}^{-c}$.
6. Then, using its knowledge of $(\mathsf{sk}_i^{(1)}, \mathsf{sk}_i^{(2)}), i \in \mathsf{corrupt}$, SIM deterministically derives each $(r_i, R_i, z_i), i \in \mathsf{corrupt}$, by first deriving

$$(r_i, R_i) \leftarrow \mathsf{Shine.Gen}(i, \mathsf{sk}_i^{(1)}, y)$$

and then deriving

$$z_i \leftarrow r_i + c \cdot \mathsf{sk}_i^{(2)}$$

7. SIM then derives each honest party's commitment $R_j, j \in \mathsf{honest}$ with respect to the joint commitment R defined in Step 5 and each corrupted party's $R_i, i \in \mathsf{corrupt}$, via Equation 5.23:

$$R_j \leftarrow R^{L_0(j)} \cdot \prod_{i \in \mathsf{corrupt}} R_i^{L_i(j)} \tag{5.23}$$

where $L_i(x)$ is the i^{th} Lagrange polynomial defined by $\mathsf{corrupt} \cup \{0\}$.
8. SIM then derives each honest party's response $z_j, j \in \mathsf{honest}$ via Equation 5.24:

$$z_j \leftarrow z \cdot L_0(j) + \sum_{i \in \mathsf{corrupt}} z_i \cdot L_i(j) \tag{5.24}$$

9. SIM sets $\mathsf{Q_y}[y] = (m, z, c, R, (R_j, z_j)_{j \in [n]})$.
10. SIM then checks if $\mathsf{Q_3}[(R, \mathsf{pk}, m)] = \perp$; if the check does not hold, then a bad event has occurred. We refer to this bad event as $\mathsf{BadEvent}_2$. In this case, SIM aborts. Otherwise, SIM programs $\mathsf{Q_3}[(R, \mathsf{pk}, m)] = c$.
11. SIM then outputs (y, R_k).

- $\underline{\mathsf{Sign}_2(k, m, C, (y_j, R_j)_{j \in C})}$: When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sign}_2}$ on honest participant identifier k, message, coalition, and tuples $\{(y_j, R_j)\}_{j \in C}$, SIM does the following:
  1. Honestly follows the protocol to ensure that $|C| \geq 2t - 1$, and returns $\perp$ if the check does not hold.
  2. Checks to ensure that $k \in \mathsf{honest}$ and $k \in C$, if not, SIM outputs $\perp$.

3. Derives $y' \leftarrow \mathsf{H}_2(\mathsf{pk}, m)$ honestly.
4. If $y_i \neq y'$ for any $i \in C$, then SIM outputs $\perp$.
5. SIM verifies the query is valid, by checking that $\mathsf{Shine.Verify}(t, \mu, C, (R_j)_{j \in C}) = 1$. If the check does not hold, SIM outputs $\perp$.
6. SIM then checks if $\mathsf{Q_y}[y'] \neq \perp$. If the check does not hold, then a bad event has occurred; we refer to this bad event as $\mathsf{BadEvent}_3$. In this case, SIM aborts.
7. Otherwise, SIM parses the entry $(m, z', c', R', (R'_j, z'_j)_{j \in [n]}) \leftarrow \mathsf{Q_y}[y']$.
8. If $(R_j)_{j \in C} \neq (R'_j)_{j \in C}$, SIM aborts. We refer to this event as $\mathsf{BadEvent}_4$.
9. Otherwise, SIM derives $R \leftarrow \prod_{j \in C} R_j^{\lambda_j}$, where λ_j is defined by C. If $R \neq R'$, SIM aborts. We refer to this bad event as $\mathsf{BadEvent}_5$.
10. Otherwise, SIM returns z'_k.

Analysis of SIM's Simulation. The simulation of H_1, H_2, and H_3 are perfect because SIM simply simulates by lazy sampling.

SIM's simulation of key generation for signing keys with respect to pk for which it does not know the corresponding sk is perfect, because $\mathcal{A}$ is allowed to corrupt up to $t-1$ participants. As such, SIM chooses $t-1$ random secret keys $\mathsf{sk}_i^{(2)}, i \in \mathsf{corrupt}$, and then simulates signing for the (unknown) honest signing keys $\mathsf{sk}_j^{(2)}, j \in \mathsf{honest}$. SIM follows the protocol honestly when it derives Shine keys.

SIM's simulation of signing is indistinguishable from honestly following the protocol when SIM does not abort, because the output signature z'_k from $\mathcal{O}^{\mathsf{Sign}_2}$ is valid with respect to the relation $g^{z'_k} = R_k \cdot (\mathsf{pk}_k^{(2)})^c$, where R_k is output from $\mathcal{O}^{\mathsf{Sign}_1}$. Further, by Equations 5.23 and 5.24, all honest parties' commitments $(R_i)_{i \in \mathsf{honest}}$ and responses $(z_i)_{i \in \mathsf{honest}}$ are consistent with corrupted parties' commitments and responses for any signing session.

SIM aborts with negligible probability more than Game 4 because:

1. When SIM aborts due $\mathsf{BadEvent}_1$ and $|\mathsf{honest}| \geq t$, then Game 4 also aborts.
2. The probability that $\mathsf{BadEvent}_2$ occurs is q_3/q because it can only occur if $\mathcal{A}$ queries $\mathsf{H}_3(R, \mathsf{pk}, m)$ for the correct R for the given m as computed in step 5 of SIM's simulation of Sign_1.
3. SIM aborts due to $\mathsf{BadEvent}_3$ with zero probability, because we assume the use of authenticated channels.

4. If SIM aborts due to $\mathsf{BadEvent}_4$, then $|C \cap \mathsf{honest}| \geq t$, and so Game 2 also aborts.

5. If SIM aborts due to $\mathsf{BadEvent}_5$, then $|C \cap \mathsf{honest}| \geq t$, then both Game 3 and Game 4 would also abort.

Output. At the end of the game, $\mathcal{A}$ outputs a forgery $(m^*, \sigma^* = (R^*, z^*))$. If $g^{z^*} \neq R^* \cdot (\mathsf{pk})^{c^*}$ (meaning that the forgery is invalid), SIM outputs $\perp$. Without loss of generality, we assume that $\mathcal{A}$ queried H_3 on (R^*, pk, m^*).

Otherwise, if $\mathcal{A}$'s forgery is valid, SIM outputs the tuple (j, aux), where j is the index corresponding to $c^* = h_j$, and $\mathsf{aux} = (m^*, \sigma^*)$ is the adversary's forgery with respect to $\mathsf{pk} = Y$.

Difference between Game 5 and Game 4. The only additional advantage introduced in Game 5 is that SIM aborts on $\mathsf{BadEvent}_2$ with probability q_3/q if $\mathcal{A}$ queries H_3 before it is programmed. Let W_5 be the event that $\mathcal{A}$ wins in Game 4. Then

$$|\Pr[W_5] - \Pr[W_4]| \leq \frac{q_3}{q} \tag{5.25}$$

Finishing the Proof. The advantage of $\mathcal{A}$ against SIM is given by Game 5. Combining Equations 5.14, 5.15, 5.17, 5.19, 5.21, and 5.25 gives Equation 5.12. This completes the proof. □

Proof of Lemma 7. We now show that if there exists a PPT adversary $\mathcal{A}$ against the unforgeability of Arctic and an algorithm SIM as described in Lemma 6 that simulates Arctic to $\mathcal{A}$, then we can define a PPT adversary $\mathcal{D}$ that can efficiently solve for discrete logarithm challenges.

$\mathcal{D}$ begins by accepting a discrete logarithm challenge $Y \in \mathbb{G}$ and public parameters par. It then chooses a (t, μ, n) such that $n \geq \mu \geq 2t - 1$, and $\binom{n-1}{t-1} = \text{poly}(n)$.

$\mathcal{D}$ then executes the modified forking algorithm $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ as described in Section 2.2 and shown in Figure 2.2, providing as input the instance $X = (\mathsf{par}, Y, n, t, \mu)$.

$\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ then either outputs $\perp$ when it fails or the value (j, aux) when it accepts. Corollary 1 upper-bounds the probability that $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ will output $\perp$ and Lemma 6

upper-bounds the additional advantage of $\mathcal{A}$ against SIM. Putting these two together gives Equation 5.26.[3]

$$\mathsf{accept}(\mathsf{Fork}_m^{\mathsf{SIM}}) \geq \frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathsf{Arctic},\mathcal{A}}(\lambda, n, t, \mu)^2}{q_r} - \frac{2(q_1 + q_3)}{q} - \frac{3q_r^2}{q} \tag{5.26}$$

We now show that when $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ outputs an accepting output (i.e., it does not output $\perp$), then $\mathcal{D}$ can solve for its discrete logarithm challenge Y with perfect success.

In the case that $\mathsf{Fork}_m^{\mathsf{SIM}}(X)$ outputs an accepting output, it outputs the tuple $(h_j, h'_j, \mathsf{aux}, \mathsf{aux}')$. From these values, $\mathcal{D}$ then parses the following:

$$\begin{aligned} (m^*, \sigma^*) \leftarrow \mathsf{aux}, \quad (m^{**}, \sigma^{**}) \leftarrow \mathsf{aux}', \\ (R^*, z^*) \leftarrow \sigma^*, \quad (R^{**}, z^{**}) \leftarrow \sigma^{**}, \\ h_j = c^*, \quad h'_j = c^{**} \end{aligned}$$

Because h_j, h'_j are with respect to the same index j corresponding to $\mathcal{A}$'s random query for c^*, c^{**} in its first and second execution, then we know that $R^* = R^{**}$ and $m^* = m^{**}$. $\mathcal{D}$ can then solve for the discrete logarithm y to its challenge $Y = g^y$ by Equation 5.27:

$$y = \mathsf{sk} = \frac{z^* - z^{**}}{c^* - c^{**}} \tag{5.27}$$

Because Fork_m outputs $\perp$ in the case when any random oracle outputs collide (i.e., in the case when any $h_i = h'_i$), then we know that Equation 5.27 is solveable. This completes the proof. □

5.7 Extending Shine and Arctic to be Robust

As presented, Shine is not a robust algorithm: if corrupt players output incorrect values from Shine.Gen, then Shine.Verify will compute that the $|C| \geq 2t - 1$ commitments do not all lie on the same degree-$(t-1)$ polynomial in the exponent, and output 0. However, Shine.Verify will not necessarily be able to identify *which* party or parties misbehaved.

[3] Because SIM is run twice by Fork_m, the number of bad events that can occur is doubled.

This is because it is possible for the $t-1$ corrupt parties to output commitments that are wrong, but consistent with up to $t-1$ of the honest players' commitments, making it look like the remaining honest player was the outlier. Therefore, as we see in Figure 5.6, Arctic will have to just abort in the case that Shine.Verify outputs 0.

However, we observe that in the *honest supermajority* case, where $|C| \geq 3t-2$, and so there are at most $t-1$ corrupt parties and at least $2t-1$ honest parties, Shine.Verify can be modified to be robust: if any corrupt parties output incorrect values from Shine.Gen, Shine.Verify will be able to notice the inconsistency *and identify the misbehaving parties*. Those parties can then be kicked out of C, and Arctic will be able to continue and produce its correct deterministic signature. The misbehaving parties may also be able to be kicked out of the system entirely, or have other external sanctions applied. The same observation was made by Cramer et al. [CDI05] for their NIVSS, but in their setting, the *shares* are themselves published, whereas in Shine, only *commitments* are available to Shine.Verify.

The reason this works is based on a simple fact about polynomials, as used in error-correcting codes: there can only be at most one polynomial of degree $t-1$ that passes through at least $2t-1$ of a given set of $3t-2$ points. The reason is that if f_1 and f_2 each passed through a (possibly different) set of at least $2t-1$ of the $3t-2$ points, then those two sets must intersect in at least t points (by simple counting). Those t points uniquely define a polynomial of degree $t-1$, and so $f_1 = f_2$. In the honest supermajority setting, the at least $2t-1$ honest parties will give consistent outputs from Shine.Gen, and so there will be exactly one (not at most one) polynomial of degree $t-1$ that passes through at least $2t-1$ of the exponents of the $3t-2$ commitments.

The typical way to *find* this unique polynomial is with the Berlekamp-Welch algorithm, but Shine.Verify only has access to commitments to the purported polynomial evaluations, and not the evaluations themselves. However, we can take advantage of the fact that we are already assuming that $\binom{n-1}{t-1}$ is not too large, and take a more brute force approach. The modification to Shine.Verify is then as follows: if player k running Shine.Verify would output 0 in Step 4 of Shine.Verify, instead iterate through all $t-1$-sized subsets S of $C \setminus \{k\}$, and reconstruct (in the exponent) the polynomial that passes through player k's own point, and the points of S. If that polynomial fails to pass through at most $t-1$ of the points, those points are the misbehaving parties. Exclude them from C, and output 1.

There are $\binom{|C|-1}{t-1}$ possible subsets S, and at least $\binom{2t-1}{t-1}$ of those subsets contain only honest parties. If the iteration is done in a pseudorandom order (for example, keyed by d_k), then the expected number of iterations before the misbehaving parties are identified is at most $\binom{|C|-1}{t-1} / \binom{2t-1}{t-1}$, which is significantly smaller than $\binom{n-1}{t-1}$, which we already assumed is not too large.

Extending Shine to be robust will make round 1 of Arctic robust, so that the correct R can be computed even if some or all of the malicious parties submit incorrect R_j values. To make round 2 of Arctic also robust, we must also guard against an adversary submitting a correct R_j in round 1, but an incorrect signature share z_j in round 2.

In the event of the failure of the test $g^z \neq R \cdot \mathsf{pk}^c$ in Combine, one can identify the parties that submitted incorrect z_j values by checking $g^{z_i} \stackrel{?}{=} R_i \cdot (\mathsf{pk}_i)^c$. Once the parties submitting incorrect signature shares are eliminated, the remaining correct shares (of which there will be at least t) can be combined into the final signature $\sigma = (R, z)$.

5.7.1 Communication Model for Robust Arctic

For non-robust Arctic, we discuss in Remark 10 how our adversarial model requires authenticated channels, but does not assume replay protection. However, additionally in the robust setting, parties will require a communication channel that ensures protection against replayed messages, to ensure that an adversary cannot replay old messages from honest participants, resulting in a potential abort. Furthermore, to achieve robustness, the communication model likewise requires liveness; the adversary cannot prevent honest parties from exchanging messages.

5.8 Performance Analysis of Arctic

In this section, we analyze the performance of Arctic. In terms of the number of rounds of communication, Arctic matches the state of the art, with two, and it sends significantly less bandwidth per signature, at 65 bytes per participant. Therefore, we focus on the computational complexity.

There are two sources of potentially expensive computation: the two calls to Shine.Gen (one in each of Sign_1 and Sign_2), and the call to Shine.Verify in Sign_2. Which one dominates depends on the parameters n, t, and the number of participants in the signing protocol. Recall that C is a set representing the identifiers of participants in a particular signing session. Although the minimum number of participants required for signing is $|C| \geq \mu \geq 2t-1$, we assume $|C| = n$ for this analysis, to give an upper performance bound. As such, performance will be even better when $\mu \leq |C| < n$.

Shine.Gen primarily performs $\delta = \binom{n-1}{t-1}$ hash computations, field multiplications, and field additions, as seen in Equation 5.3 (recalling that the $L'_{a_i}(k)$ values can be precomputed). Shine.Verify computes $\ell = |C| - t$, $|C|$-way multiexponentiations. When t is small,

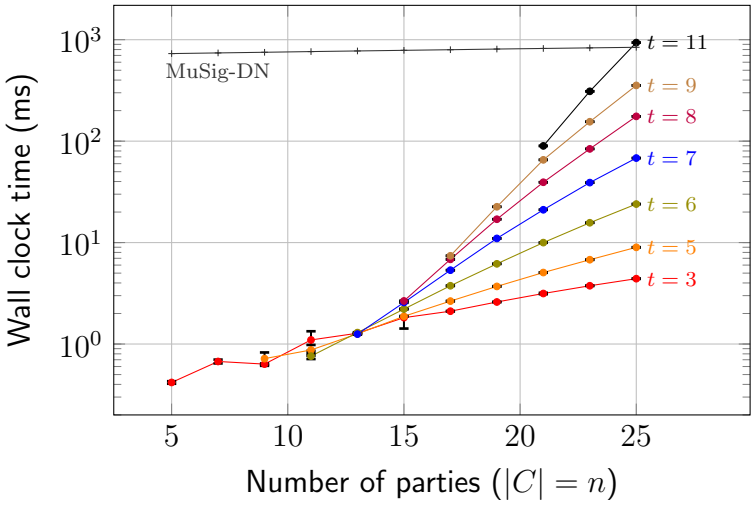


Figure 5.7: Single-core wall clock time for for Arctic.

we expect the Shine.Verify cost to dominate, and for larger t, the Shine.Gen cost should dominate.

To concretely evaluate the performance of Arctic, we implemented it in Rust.[4] We ran our implementation over all allowable combinations of parameters $t \geq 2$, $2t - 1 \leq |C| \leq n \leq 25$, using a 4-core 3.7 GHz Intel E-2374G CPU. We measured the computation time for each of Sign_1, Sign_2, and Combine, averaged over 10 signatures, for each parameter combination. In Figure 5.7, we show the (single-core) total runtime of Sign_1, Sign_2, and Combine for various combinations of parameters. We concretely measure Shine.Gen to take around 0.24δ microseconds for each of its two invocations, and Shine.Verify to take around $7\ell|C|$ microseconds. For $t \leq 4$, the latter dominates, for $t = 5$, they are roughly comparable, and for $t \geq 6$, the former quickly dominates.

For comparison, we also show the computational time for MuSig-DN, but *only* the zero-knowledge proof and verification components of their algorithm. We ran their code [Nic20] on our same machine to obtain these figures. We can see that for $n \leq 20$, Arctic is more than an order of magnitude faster than MuSig-DN, and for $n \leq 10$, it is three orders of magnitude faster.

As seen in Figure 5.7, the computation time for the $(n, |C|, t) = (25, 25, 11)$ parameter combination, where $\delta = \binom{24}{10} = 1961256$ is around 940 ms, almost all of which is spent in Shine.Gen computing Equation 5.3. However, we observe that Equation 5.3 computes the

[4] Our code is available at https://git-crysp.uwaterloo.ca/iang/arctic/.

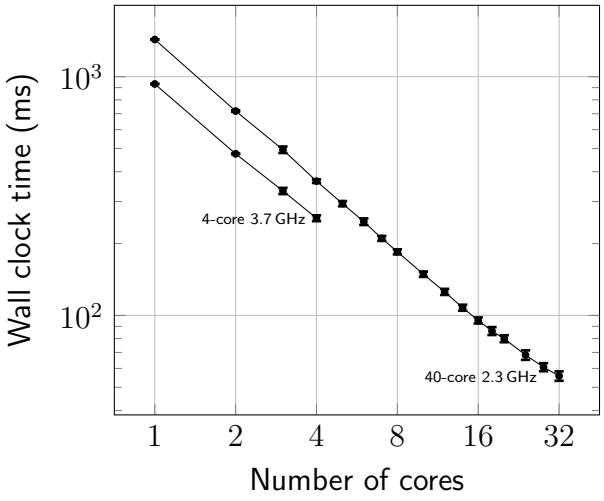


Figure 5.8: Scaling experiment, showing the wall clock time of $(n, |C|, t) = (25, 25, 11)$ shown in Figure 5.7 relative to CPU cores.

sum of δ independent terms, and so is highly amenable to parallelization, which we also implemented and measured. We ran this scaling experiment both on the above machine, and also on a 40-core 2.3 GHz Intel 8380 CPU. The results are shown in Figure 5.8. Although the slower clock speed of the 40-core CPU puts it at a disadvantage for smaller numbers of cores, we can see almost linear scaling for both CPUs. Using all 4 cores, the 4-core CPU sees a speedup of 3.69× for Shine.Gen and 3.65× in total time, while using 32 cores, the 40-core CPU sees a speedup of 27.9× for Shine.Gen and 25.7× in total time. Beyond 32 cores, we observed diminishing returns.

5.8.1 Comparison to Related Work

For comparison, we next expand on our performance estimates for two related schemes given in Table 5.1. We do not give exact computation estimates, but instead give estimates for computation that dominate. For bandwidth, we estimate for a 256-bit elliptic curve, and so estimate scalers as 32 bytes and group elements as 33 bytes.

MuSig-DN [NRSW20]. Nick et al. [NRSW20] present the proof size as 1124 bytes, which agrees with our measurements. Further, each participant sends one group element in round one, and outputs one field element in round three, yielding the total of 1189 bytes we report in Table 5.1.

While Nick et al. give total computational overhead for MuSig-DN for a 256-bit curve, we give more detail as to computation that dominates when performing signing operations.

The computational overhead of MuSig-DN is dominated by the cost of generating and verifying Bulletproof [BBB+18] proofs. For each signing operation, each participant must generate one proof and verify $(n-1)$ other proofs. While the computational overhead of Bulletproofs scales linearly relative to the size of the circuit, batch verification and multi-scalar multiplication allow for improved performance.

Proofs in MuSig-DN require a circuit of 2030 multiplication gates [NRSW20], and estimates are given with respect to a 256-bit elliptic curve group. Naively, this overhead would translate into roughly $2030n$ operations for each signer, due to the requirement that each signer generates a proof and verifies $n-1$ other proofs.

However, when considering speedups offered by multi-scalar multiplications and batch verification, the overhead for a signer becomes roughly 2030 operations, plus additional (small) terms linear in n. When generating a MuSig-DN proof using Bulletproofs, we estimate that the dominant computational overhead is roughly $6 \cdot 2030$ multi-scalar multiplications, to commit to the circuit constraints and additional factors. For batch verifying $n-1$ MuSig-DN proofs, we estimate the dominant computational overhead to the verifier is $2030 + 24(n-1)$ scalar multiplications.

GKMN21 [GKMN21]. Garillot et al. [GKMN21] give computational and bandwidth estimates for a single sender and receiver, but each player will be involved in $t-1$ concurrent sending and receiving sessions with all other players. As such, the estimate we give in Table 5.1 is for the total sending and receiving overhead for each signer in a signing session with t participants.

Arctic. For bandwidth, each party sends one group element and one field element as the output to the first signing query, and then outputs a single field element in the second signing round.

For computation, each party performs Shine.Gen once for each signing round, resulting in $\binom{n-1}{t-1}$ field and hash operations, and one group multiplication, for each execution. In the second round of signing, each party performs Shine.Gen again, for another $\binom{n-1}{t-1}$ field and hash operations, and one group multiplication, and also Shine.Verify and Shine.Agg, which require $2t^2 - t$ group multiplications.

Chapter 6

Conclusion

In this thesis, we have investigated the design of practical threshold key generation and signing schemes. We have presented several constructions that minimize barriers to implementation, including reducing network latency and communication overhead, and limiting state-keeping.

In Chapter 3, we presented FROST, a Flexible Round-Optimized Schnorr Threshold Signature. We showed how FROST can be used either as a two-round protocol, or optimized to a single-round online signing protocol with a batched preprocessing phase. As discussed in Section 1.2, FROST has already found use in practice, and will likely become the scheme of choice in the future for thresholdizing Schnorr signatures in practical applications.

In Chapter 4, we presented new game-based definitions for distributed key generation schemes. We then presented a generic construction that employs as building blocks a non-interactive key-exchange (NIKE) scheme, an aggregated verifiable secret sharing (AgVSS) scheme, and a secure hash function. We then presented Storm, a concrete instantiation of our generic construction that is secure assuming the hardness of the Computational Diffie-Hellman (CDH) assumption.

Finally, in Chapter 5, we presented Arctic, a deterministic and stateless threshold Schnorr signature scheme. Arctic builds upon pseudorandom secret sharing as a building block, and allows all participants to perform a two-round threshold signing operation, without requiring participants to maintain state in between or across signing rounds. Arctic requires the assumption of an honest majority, and is efficient for small signing groups (i.e., when the total number of possible signers is less than 25).

The thesis statement, as given in Chapter 1, is that it is possible to construct efficient threshold key generation and signature schemes that are practically useful. The three constructions we present in this thesis are optimized for efficiency, simplicity, and low round complexity, and are among the most efficient in the literature, while maintaining backwards compatibility with existing centralized schemes, often a key priority in practice. The real-world deployment of FROST in particular underscores that this thesis statement is true.

6.1 Future Work

While this thesis presents advancements in the design of practical threshold key generation and signatures, many problems remain. For example, this thesis largely considers the problem of efficient and practical threshold Schnorr signatures. However, it is possible that the techniques employed in this work are equally applicable in the setting of other signature schemes, such as threshold ECDSA. While randomized threshold ECDSA signatures under honest-majority assumptions exist that similarly build upon pseudorandom secret sharing [DJN+22, DKLS24, DOK+20], the techniques we employ in FROST may likewise potentially be applicable to improving the efficiency of threshold ECDSA signatures.

Another avenue for future research is the investigation of the minimal security assumptions required for two-round threshold Schnorr signature schemes. While Arctic can be proven under standard assumptions, FROST requires the stronger Algebraic One-More Discrete Logarithm (ℓ-AOMDL) assumption. It is interesting to consider whether this difference is inherent, or if it is possible to prove two-round randomized Schnorr threshold signature secure under weaker assumptions.

6.2 Reflections

While threshold signatures have been extensively studied in the literature for decades, it is only more recently that threshold signatures have enjoyed adoption in practice. With the increase in real-world deployments, additional requirements have been introduced for the design of threshold signatures, including simplicity, minimizing round complexity, and backwards compatibility with single-party signature schemes, such as Schnorr signatures. These new requirements in practice have led to the research presented in this thesis.

What does the future hold for threshold signatures and distributed key generation? Clearly, investigating how to define quantum-secure (and practical) versions of these

schemes is an immediate priority. However, as threshold signatures gain more widespread adoption, it will be interesting to see what additional practical requirements emerge. In particular, using threshold cryptography beyond the core current use case of securing cryptocurrencies is an important next step. Hopefully, threshold cryptography will continue to prove its relevance in the real world by distributing trust, in a wider range of use cases in the future.

References

[AAM20] Aysajan Abidin, Abdelrahaman Aly, and Mustafa A. Mustafa. Collaborative Authentication Using Threshold Cryptography. In *Emerging Technologies for Authorization and Authentication*, pages 122–137, 2020.

[AJM+21] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching Consensus for Asynchronous Distributed Key Generation. In *40th ACM PODC*, pages 363–373, 2021.

[AJM+22] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, and Gilad Stern. Bingo: Adaptively Secure Packed Asynchronous Verifiable Secret Sharing and Asynchronous Distributed Key Generation. Cryptology ePrint Archive, Report 2022/1759, 2022.

[Arc25] Arch Network. Arch's Signature Scheme Model (FROST + ROAST). https://arch-network.gitbook.io/arch-documentation/fundamentals/introducing-arch/archs-signature-scheme-model-frost-+-roast, 2025. Accessed Dec 2025.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short Proofs for Confidential Transactions and More. In *IEEE Symposium on Security and Privacy*, pages 315–334, 2018.

[BBG+21] Fabrice Benhamouda, Elette Boyle, Niv Gilboa, Shai Halevi, Yuval Ishai, and Ariel Nof. Generalized Pseudorandom Secret Sharing and Efficient Straggler-Resilient Secure Computation. In *TCC*, volume 13043 of *LNCS*, pages 129–161, 2021.

[BBLW22] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher Wood. Hybrid Public Key Encryption, 2022. Available at https://www.rfc-editor.org/info/rfc9180.

[BCG+20] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient Pseudorandom Correlation Generators from Ring-LPN. In *CRYPTO*, volume 12171 of *LNCS*, pages 387–416, 2020.

[BCK+22] Mihir Bellare, Elizabeth C. Crites, Chelsea Komlo, Mary Maller, Stefano Tessaro, and Chenzhi Zhu. Better than Advertised Security for Non-interactive Threshold Signatures. In *CRYPTO*, volume 13510 of *LNCS*, pages 517–550, 2022.

[BD22] Luís Brandão and Michael Davidson. Notes on Threshold EdDSA/Schnorr Signatures. https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8214B.ipd.pdf, 2022.

[BDN18] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact Multi-signatures for Smaller Blockchains. In *ASIACRYPT*, pages 435–464, 2018.

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie–Hellman Speed Records. In *PKC*, volume 3958 of *LNCS*, pages 207–228, 2006.

[BFP21] Balthazar Bauer, Georg Fuchsbauer, and Antoine Plouviez. The One-More Discrete Logarithm Assumption in the Generic Group Model. In *ASIACRYPT*, volume 13093 of *LNCS*, pages 587–617, 2021.

[BHLS25] Dan Boneh, Iftach Haitner, Yehuda Lindell, and Gil Segev. Exponent-VRFs and Their Applications. In *EUROCRYPT*, volume 15607 of *LNCS*, pages 195–224, 2025.

[Bit24] Bitcoin Wiki. Multi-Signatures. https://en.bitcoin.it/wiki/Multi-signature, 2024. Accessed Dec 2025.

[BKP13] Rikke Bendlin, Sara Krehbiel, and Chris Peikert. How to Share a Lattice Trapdoor: Threshold Protocols for Signatures and (H)IBE. In *ACNS*, volume 7954 of *LNCS*, pages 218–236, 2013.

[BL22] Renas Bacho and Julian Loss. On the Adaptive Security of the Threshold BLS Signature Scheme. In *CCS*, pages 193–207, 2022.

[BLL+21] Fabrice Benhamouda, Tancrède Lepoint, Julian Loss, Michele Orrù, and Mariana Raykova. On the (in)security of ROS. In *EUROCRYPT*, volume 12696 of *LNCS*, pages 33–53, 2021.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. *Journal of Cryptology*, 17(4):297–319, 2004.

[BN06] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-Key model and a general forking lemma. In *CCS*, pages 390–399, 2006.

[BNPS03] Mihir Bellare, Chanathip Namprempre, David Pointcheval, and Michael Semanko. The One-More-RSA-Inversion Problems and the Security of Chaum's Blind Signature Scheme. *Journal of Cryptology*, 16(3):185–215, 2003.

[Bol03] Alexandra Boldyreva. Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme. In *PKC*, volume 2567 of *LNCS*, pages 31–46, 2003.

[BP04] Mihir Bellare and Adriana Palacio. The Knowledge-of-Exponent Assumptions and 3-Round Zero-Knowledge Protocols. In *CRYPTO*, volume 3152 of *LNCS*, pages 273–289, 2004.

[BP23] Luís Brandão and Rene Peralta. NIST First Call for Multi-Party Threshold Schemes. https://nvlpubs.nist.gov/nistpubs/ir/2023/NIST.IR.8214C.ipd.pdf, 2023.

[BR06] Mihir Bellare and Phillip Rogaway. The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs. In *EUROCRYPT*, volume 4004 of *LNCS*, pages 409–426, 2006.

[BS20] Dan Boneh and Victor Shoup. A Graduate Course in Applied Cryptography, 2020. Available at http://toc.cryptobook.us/book.pdf. Accessed Dec 2025.

[BST21] Charlotte Bonte, Nigel P. Smart, and Titouan Tanguy. Thresholdizing HashEdDSA: MPC to the Rescue. *Int. J. Inf. Sec.*, 20(6):879–894, 2021.

[Can01] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *FOCS*, pages 136–145, 2001.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation. In *TCC*, volume 3378 of *LNCS*, pages 342–362, 2005.

[CG22] Deirdre Connolly Conrado Gouvea, Chelsea Komlo. FROST for Spend Authorization Multisignatures. https://zips.z.cash/zip-0312, 2022. Accessed Dec 2025.

[CGG+20] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In *CCS*, pages 1769–1787, 2020.

[CGRS23] Hien Chu, Paul Gerhart, Tim Ruffing, and Dominique Schröder. Practical Schnorr Threshold Signatures Without the Algebraic Group Model. In *CRYPTO*, volume 14081 of *LNCS*, pages 743–773, 2023.

[CKGW24] Deirdre Connolly, Chelsea Komlo, Ian Goldberg, and Chris Wood. The Flexible Round-Optimized Schnorr Threshold (FROST) Protocol for Two-Round Schnorr Signatures, 2024. Available at https://www.rfc-editor.org/info/rfc9591.

[CKM23] Elizabeth C. Crites, Chelsea Komlo, and Mary Maller. Fully Adaptive Schnorr Threshold Signatures. In *CRYPTO*, volume 14081 of *LNCS*, pages 678–709, 2023.

[CKS08] David Cash, Eike Kiltz, and Victor Shoup. The Twin Diffie–Hellman Problem and Applications. In *EUROCRYPT*, volume 4965 of *LNCS*, pages 127–145, 2008.

[CS04] John F. Canny and Stephen Sorkin. Practical Large-Scale Distributed Key Generation. In *EUROCRYPT*, volume 3027 of *LNCS*, pages 138–152, 2004.

[Dam91] Ivan Damgård. Towards Practical Public Key Systems Secure Against Chosen Ciphertext Attacks. In *CRYPTO*, volume 576 of *LNCS*, pages 445–456, 1991.

[DEF+19] Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. On the Security of Two-Round Multi-Signatures. In *IEEE Symposium on Security and Privacy*, pages 1084–1101, 2019.

[DJN+22] Ivan Damgård, Thomas P. Jakobsen, Jesper Buus Nielsen, Jakob Illeborg Pagter, and Michael Bæksvang Østergaard. Fast threshold ECDSA with honest majority. *J. Comput. Secur.*, 30(1):167–196, 2022.

[DKLS24] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Threshold ECDSA in Three Rounds. In *IEEE Symposium on Security and Privacy*, pages 3053–3071, 2024.

[DM10] Ivan Damgård and Gert Læssøe Mikkelsen. Efficient, Robust and Constant-Round Distributed RSA Key Generation. In *TCC*, volume 5978 of *LNCS*, pages 183–200, 2010.

[Dod03] Yevgeniy Dodis. Efficient Construction of (Distributed) Verifiable Random Functions. In *PKC*, volume 2567 of *LNCS*, pages 1–17, 2003.

[DOK+20] Anders P. K. Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrishak, and Haya Schulmann. Securing DNSSEC Keys via Threshold ECDSA from Generic MPC. In *ESORICS*, volume 12309 of *LNCS*, pages 654–673, 2020.

[dra25] drand. League of Entropy. https://www.drand.love/loe, 2025. Accessed Dec 2025.

[DSQ+22] Alex Davidson, Peter Snyder, E. B. Quirk, Joseph Genereux, Benjamin Livshits, and Hamed Haddadi. STAR: Secret Sharing for Private Threshold Aggregation Reporting. In *CCS*, pages 1569–1583, 2022.

[DYX+22] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew K. Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical Asynchronous Distributed Key Generation. In *IEEE Symposium on Security and Privacy*, pages 2518–2534, 2022.

[Fel87] Paul Feldman. A Practical Scheme for Non-interactive Verifiable Secret Sharing. In *FOCS*, pages 427–437, 1987.

[FHKP13] Eduarda S. V. Freire, Dennis Hofheinz, Eike Kiltz, and Kenneth G. Paterson. Non-Interactive Key Exchange. In *PKC*, volume 7778 of *LNCS*, pages 254–271, 2013.

[Fis05] Marc Fischlin. Communication-Efficient Non-interactive Proofs of Knowledge with Online Extractors. In *CRYPTO*, volume 3621 of *LNCS*, pages 152–168, 2005.

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The Algebraic Group Model and its Applications. In *CRYPTO*, volume 10992 of *LNCS*, pages 33–62, 2018.

[FMY98] Yair Frankel, Philip D. MacKenzie, and Moti Yung. Robust Efficient Distributed RSA-Key Generation. In *STOC*, pages 663–672, 1998.

[FPS20] Georg Fuchsbauer, Antoine Plouviez, and Yannick Seurin. Blind Schnorr Signatures and Signed ElGamal Encryption in the Algebraic Group Model. In *EUROCRYPT*, volume 12106 of *LNCS*, pages 63–95, 2020.

[FS86] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO*, volume 263 of *LNCS*, pages 186–194, 1986.

[GG18] Rosario Gennaro and Steven Goldfeder. Fast Multiparty Threshold ECDSA with Fast Trustless Setup. In *CCS*, pages 1179–1194, 2018.

[GG20] Rosario Gennaro and Steven Goldfeder. One Round Threshold ECDSA with Identifiable Abort. Cryptology ePrint Archive, Report 2020/540, 2020.

[GGK+15] Steven Goldfeder, Rosario Gennaro, Harry Kalodner, Joseph Bonneau, Joshua A. Kroll, Edward W. Felten, and Arvind Narayanan. Securing Bitcoin wallets via a new DSA/ECDSA threshold signature scheme. http://stevengoldfeder.com/papers/threshold_sigs.pdf, 2015. Accessed Dec 2025.

[GJKR96] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Robust Threshold DSS Signatures. In *EUROCRYPT*, volume 1070 of *LNCS*, pages 354–371, 1996.

[GJKR01] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Robust Threshold DSS Signatures. *Inf. Comput.*, 164(1):54–84, 2001.

[GJKR03] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure Applications of Pedersen's Distributed Key Generation Protocol. In *CT-RSA*, volume 2612 of *LNCS*, pages 373–390, 2003.

[GJKR07] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure Distributed Key Generation for Discrete-Log Based Cryptosystems. *Journal of Cryptology*, 20(1):51–83, 2007.

[GJM+21] Kobi Gurkan, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Aggregatable Distributed Key Generation. In *EUROCRYPT*, pages 147–176, 2021.

[GKMN21] François Garillot, Yashvanth Kondi, Payman Mohassel, and Valeria Nikolaenko. Threshold Schnorr with Stateless Deterministic Signing from Standard Assumptions. In *CRYPTO*, volume 12825 of *LNCS*, pages 127–156, 2021.

[GLOW21] David Galindo, Jia Liu, Mihai Ordean, and Jin-Mann Wong. Fully Distributed Verifiable Random Functions and their Application to Decentralised Random Beacons. In *EuroS&P*, pages 88–102, 2021.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-message Attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[GRJK07] Rosario Gennaro, Tal Rabin, Stanislaw Jarecki, and Hugo Krawczyk. Robust and Efficient Sharing of RSA Functions. *Journal of Cryptology*, 20(3):393, 2007.

[Gro10] Jens Groth. Short Pairing-Based Non-interactive Zero-Knowledge Arguments. In *ASIACRYPT*, volume 6477 of *LNCS*, pages 321–340, 2010.

[Gro21] Jens Groth. Non-interactive distributed key generation and key resharing. Cryptology ePrint Archive, Report 2021/339, 2021.

[ISN89] Mitsuru Ito, Akira Saito, and Takao Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *CCS*, pages 955–966, 2013.

[JKR18] Stanislaw Jarecki, Hugo Krawczyk, and Jason Resch. Threshold Partially-Oblivious PRFs with Applications to Key Management. Cryptology ePrint Archive, Report 2018/733, 2018.

[Kat24] Jonathan Katz. Round-Optimal, Fully Secure Distributed Key Generation. In *CRYPTO*, volume 14926 of *LNCS*, pages 285–316, 2024.

[KG20] Chelsea Komlo and Ian Goldberg. FROST: Flexible Round-Optimized Schnorr Threshold Signatures. In *SAC*, volume 12804 of *LNCS*, pages 34–65, 2020.

[KG25] Chelsea Komlo and Ian Goldberg. Arctic: Lightweight and Stateless Threshold Schnorr Signatures. In *PKC*, volume 15678 of *LNCS*, pages 234–267, 2025.

[KGS23] Chelsea Komlo, Ian Goldberg, and Douglas Stebila. A Formal Treatment of Distributed Key Generation, and New Constructions. Cryptology ePrint Archive, Report 2023/292, 2023.

[KHG12] Aniket Kate, Yizhou Huang, and Ian Goldberg. Distributed Key Generation in the Wild. Cryptology ePrint Archive, Report 2012/377, 2012.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

[KMS20] Eleftherios Kokoris-Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous Distributed Key Generation for Computationally-Secure Randomness, Consensus, and Threshold Signatures. In *CCS*, pages 1751–1767, 2020.

[Kom22] Chelsea Komlo. On Security Assumptions Underpinning Recent Schnorr Threshold Schemes. https://crypto.ethereum.org/blog/schnorr-threshold-blogpost, 2022. Accessed Dec 2025.

[Kon24] Yashvanth Kondi. Personal Communication, 2024.

[KOR23] Yashvanth Kondi, Claudio Orlandi, and Lawrence Roy. Two-Round Stateless Deterministic Two-Party Schnorr Signatures from Pseudorandom Correlation Functions. In *CRYPTO*, volume 14081 of *LNCS*, pages 646–677, 2023.

[KZe19] KZen Networks. Multi Party Schnorr Signatures. https://github.com/KZen-networks/multi-party-schnorr, 2019. Accessed Dec 2025.

[LdV20] Isis Lovecruft and Henry de Valence. The Ristretto Group. https://doc.dalek.rs/curve25519_dalek/, 2020. Accessed Dec 2025.

[Lin24] Yehuda Lindell. Simple Three-Round Multiparty Schnorr Signing with Full Simulatability. *IACR Commun. Cryptol.*, 1(1):25, 2024.

[LJY16] Benoît Libert, Marc Joye, and Moti Yung. Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. *Theoretical Computer Science*, 645:1–24, 2016.

[LN18] Yehuda Lindell and Ariel Nof. Fast Secure Multiparty ECDSA with Practical Distributed Key Generation and Applications to Cryptocurrency Custody. In *CCS*, pages 1837–1854, 2018.

[Mak22] Nikolaos Makriyannis. On the Classic Protocol for MPC Schnorr Signatures. Cryptology ePrint Archive, Paper 2022/1332, 2022. https://eprint.iacr.org/2022/1332.

[MOT+11] Prateek Mittal, Femi G. Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval. In *USENIX Security*, 2011.

[MRV99] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable Random Functions. In *FOCS*, pages 120–130, 1999.

[MYG24] Nikolaos Makriyannis, Oren Yomtov, and Arik Galansky. Practical Key-Extraction Attacks in Leading MPC Wallets. In *CCS*, pages 3053–3064, 2024.

[Nic20] Jonas Nick. Switch bulletproof example to purify. https://github.com/jonasnick/secp256k1-zkp/blob/bulletproof-musig-dn-benches/examples/bulletproof.c, 2020. Accessed Dec 2025.

[NPR99] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed Pseudo-random Functions and KDCs. In *EUROCRYPT*, volume 1592 of *LNCS*, pages 327–346, 1999.

[NRS21] Jonas Nick, Tim Ruffing, and Yannick Seurin. MuSig2: Simple Two-Round Schnorr Multi-signatures. In *CRYPTO*, volume 12825 of *LNCS*, pages 189–221, 2021.

[NRSW20] Jonas Nick, Tim Ruffing, Yannick Seurin, and Pieter Wuille. MuSig-DN: Schnorr Multi-Signatures with Verifiably Deterministic Nonces. In *CCS*, pages 1717–1731, 2020.

[NSR16] Wafa Neji, Kaouther Blibech Sinaoui, and Narjes Ben Rajeb. Distributed key generation protocol with a new complaint management strategy. *Secur. Commun. Networks*, 9(17):4585–4595, 2016.

[OSY21] Claudio Orlandi, Peter Scholl, and Sophia Yakoubov. The Rise of Paillier: Homomorphic Secret Sharing and Public-Key Silent OT. In *EUROCRYPT*, volume 12696 of *LNCS*, pages 678–708, 2021.

[Ped91] Torben P. Pedersen. A Threshold Cryptosystem without a Trusted Party (Extended Abstract). In *EUROCRYPT*, volume 547 of *LNCS*, pages 522–526, 1991.

[Pos25] Jesse Posner. BIP340 compatible implementation of Flexible Round-Optimized Schnorr Threshold Signatures (FROST). https://github.com/jesseposner/FROST-BIP340, 2025. Accessed Dec 2025.

[PS96] David Pointcheval and Jacques Stern. Security Proofs for Signature Schemes. In *EUROCRYPT*, volume 1070 of *LNCS*, pages 387–398, 1996.

[PS00] David Pointcheval and Jacques Stern. Security Arguments for Digital Signatures and Blind Signatures. *Journal of Cryptology*, 13(3):361–396, 2000.

[RN25] Tim Ruffing and Jonas Nick. ChillDKG: Distributed Key Generation for FROST. https://github.com/BlockstreamResearch/bip-frost-dkg, 2025. Accessed Dec 2025.

[RRJ+22] Tim Ruffing, Viktoria Ronge, Elliott Jin, Jonas Schneider-Bensch, and Dominique Schröder. ROAST: Robust Asynchronous Schnorr Threshold Signatures. In *CCS*, pages 2551–2564, 2022.

[RY10] Thomas Ristenpart and Scott Yilek. When Good Randomness Goes Bad: Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography. In *NDSS 2010*, 2010.

[Sch91] Claus-Peter Schnorr. Efficient Signature Generation by Smart Cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[Sch01] Claus-Peter Schnorr. Security of Blind Discrete Log Signatures against Interactive Attacks. In *Information and Communications Security, Third International Conference, ICICS*, volume 2229 of *LNCS*, pages 1–12, 2001.

[Sch06] Claus-Peter Schnorr. Enhancing the security of perfect blind DL-signatures. *Inf. Sci.*, 176(10):1305–1320, 2006.

[Sha79] Adi Shamir. How to Share a Secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, 1979.

[SS01] Douglas R. Stinson and Reto Strobl. Provably Secure Distributed Schnorr Signatures and a (t, n) Threshold Scheme for Implicit Certificates. In *ACISP*, volume 2119 of *LNCS*, pages 417–434, 2001.

[SSL25] SSLMate. Timeline of Certificate Authority Failures. https://sslmate.com/resources/certificate_authority_failures, 2025. Accessed Dec 2025.

[SU17] Dominique Schröder and Dominique Unruh. Security of Blind Signatures Revisited. *Journal of Cryptology*, 30(2):470–494, 2017.

[Wag02] David A. Wagner. A Generalized Birthday Problem. In *CRYPTO*, volume 2442 of *LNCS*, pages 288–303, 2002.

[YLF+22] Thomas Yurek, Licheng Luo, Jaiden Fairoze, Aniket Kate, and Andrew Miller. hbACSS: How to Robustly Share Many Secrets. In *NDSS*, 2022.

[Zca25] Zcash Foundation. ZF FROST (Flexible Round-Optimised Schnorr Threshold signatures). https://github.com/ZcashFoundation/frost, 2025. Accessed Dec 2025.

[ZXH+22] Jiaheng Zhang, Tiancheng Xie, Thang Hoang, Elaine Shi, and Yupeng Zhang. Polynomial Commitment with a One-to-Many Prover and Applications. In *USENIX Security 2022*, pages 2965–2982, 2022.