HARTS: High-Threshold, Adaptively Secure, and Robust Threshold Schnorr Signatures

Renas Bacho 1,3, Julian Loss 1, Gilad Stern 2, Benedikt Wagner 1,3

February 19, 2024

1 CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{renas.bacho,loss,benedikt.wagner}@cispa.de
2 Tel Aviv University, Tel Aviv-Yafo, Israel
gilad.stern@mail.huji.ac.il
3 Saarland University, Saarbrücken, Germany

Abstract

Threshold variants of the Schnorr signature scheme have recently been at the center of attention due to their applications to Bitcoin, Ethereum, and other cryptocurrencies. However, existing constructions for threshold Schnorr signatures among a set of \( n \) parties with corruption threshold \( t_c \) suffer from at least one of the following drawbacks: (i) security only against static (i.e., non-adaptive) adversaries, (ii) cubic or higher communication cost to generate a single signature, (iii) strong synchrony assumptions on the network, or (iv) \( t_c + 1 \) are sufficient to generate a signature, i.e., the corruption threshold of the scheme equals its reconstruction threshold. Especially (iv) turns out to be a severe limitation for many asynchronous real-world applications where \( t_c < n/3 \) is necessary to maintain liveness, but a higher signing threshold of \( n - t_c \) is needed. A recent scheme, ROAST, proposed by Ruffing et al. (ACM CCS ’22) addresses (iii) and (iv), but still falls short of obtaining subcubic complexity and adaptive security.

In this work, we present HARTS, the first threshold Schnorr signature scheme to incorporate all these desiderata. More concretely:

- **HARTS** is adaptively secure and remains fully secure and operational even under asynchronous network conditions in the presence of up to \( t_c < n/3 \) malicious parties. This is optimal.
- **HARTS** outputs a Schnorr signature of size \( \lambda \) with a near-optimal amortized communication cost of \( O(\lambda n^3 \log n) \) bits and \( O(1) \) rounds per signature.
- **HARTS** is a high-threshold scheme: no fewer than \( t_r + 1 \) signature shares can be combined to yield a full signature, where \( t_r \geq 2n/3 > 2t_c \). This is optimal.

We prove our result in a modular fashion in the algebraic group model. At the core of our construction, we design a new simple, and adaptively secure high-threshold AVSS scheme which may be of independent interest.

Keywords: Threshold Signatures, Schnorr Signatures, Adaptive Security, Robustness, High-Threshold, Asynchronous Network

1 Introduction

A threshold signature [Des88, DF90] scheme is a special type of digital signature scheme that allows any set of \( t_r + 1 \) signers in a system of \( n \) parties to jointly generate a compact signature \( \sigma \) on a message \( m \). On the other hand, this should be infeasible for \( t_r \) or less signers. Over the last two decades, many threshold versions of the Schnorr signature scheme [Sch91] have been proposed [BP23, CGRS23, CKM23a, GJKR07, KG20, Lin22, RRJ+22, SS01, TZ23, WNR20]. Collectively, these schemes offer a great variety
We refer to Table 1 for a complete overview and comparison of our scheme’s properties with existing schemes from the literature.

We build HARTS, a novel threshold Schnorr signature scheme that improves significantly over prior works. Concretely, HARTS has the following properties:

- **Efficiency**: HARTS produces signatures with a (near-optimal) amortized communication cost of $O(\lambda n^2 \log n)$ bits and round complexity $O(1)$.

- **Asynchrony**: HARTS remains fully secure and operational against up to (optimal) $t_c < n/3$ corrupted parties in a fully asynchronous network where message delivery between honest parties can take longer than expected [LSP82].

- **Adaptive Security**: HARTS is secure against adaptive corruptions.

- **High-threshold**: HARTS is a high-threshold signature scheme satisfying the following features: (i) a signing session results in a valid signature even in the presence of up to $t_c$ malicious parties that try to prevent the other parties from generating a signature, (ii) a signature cannot be created given less than $t_r + 1 = n - t_c$ signature shares, where $t_r \geq 2n/3 > 2t_c$. This notion of enhanced security has found many applications and real-world significance in recent years, especially in the context of consensus and blockchain systems [GKKS+22, YMR+19].

We refer to Table 1 for a complete overview and comparison of our scheme’s properties with existing schemes from the literature.

### A Modular Approach

We build HARTS by following a modular approach, which is summarized in Figure 1 and outlined in more detail below.

- **Threshold Schnorr Signatures from ADKG**: Building upon the technique of Gennaro et al. [GJKR99], we construct a generic high-threshold and robust threshold Schnorr signature scheme. As a building block, we use a (packed) high-threshold asynchronous distributed key generation (ADKG) protocol. We prove unforgeability of this scheme against an adaptive adversary in the algebraic group model (AGM) [FKL18] based on the security of the (packed) ADKG protocol and the one-more discrete logarithm assumption.

- **Packed ADKG from AVSS**: We give a generic construction (cf. Figure 2) for an efficient packed high-threshold ADKG from a high-threshold asynchronous verifiable secret sharing (AVSS) scheme using the technique of superinvertible matrices [HN06]. We prove the adaptive security of this construction by reduction to the security of the AVSS scheme and the underlying consensus primitives.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Network</th>
<th>Robust</th>
<th>Corrupt</th>
<th>Reconst</th>
<th>Adapt</th>
<th>Commun</th>
<th>Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROST</td>
<td>sync</td>
<td>✓</td>
<td>$t_c &lt; n$</td>
<td>$t_r = t_c$</td>
<td>✓</td>
<td>$O(\lambda n^3)$</td>
<td>2</td>
</tr>
<tr>
<td>Sparkle</td>
<td>sync</td>
<td>✓</td>
<td>$t_c &lt; n$</td>
<td>$t_r = t_c$</td>
<td>✓</td>
<td>$O(\lambda n^2)$</td>
<td>3</td>
</tr>
<tr>
<td>GJKR</td>
<td>sync</td>
<td>✓</td>
<td>$t_r &lt; n/2$</td>
<td>$t_r = t_c$</td>
<td>✓</td>
<td>$O(\lambda n^3)$</td>
<td>3 BC</td>
</tr>
<tr>
<td>ROAST</td>
<td>async</td>
<td>✓</td>
<td>$t_c &lt; n^3$</td>
<td>$t_r &lt; n-t_c$</td>
<td>✓</td>
<td>$O(\lambda^3 + n^4)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>SPRINT</td>
<td>async</td>
<td>✓</td>
<td>$t_r &lt; n/3$</td>
<td>$t_r = t_c$</td>
<td>✓</td>
<td>$O(\lambda n^2)$</td>
<td>3 BC</td>
</tr>
<tr>
<td>GS23</td>
<td>async</td>
<td>✓</td>
<td>$t_r &lt; n/3$</td>
<td>$t_r = t_c$</td>
<td>✓</td>
<td>$O(\lambda n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

**Our work**

<table>
<thead>
<tr>
<th>Network</th>
<th>Robust</th>
<th>Corrupt</th>
<th>Reconst</th>
<th>Adapt</th>
<th>Commun</th>
<th>Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>async</td>
<td>✓</td>
<td>$t_c &lt; n/3$</td>
<td>$t_r &lt; n-t_c$</td>
<td>✓</td>
<td>✓</td>
<td>$O(\lambda n)$</td>
</tr>
</tbody>
</table>
We briefly elaborate on this. Each party to sign a message completed successfully using a consensus tools. Say in the presence of reconstructed independently from the others with reconstruction threshold ℓ such a protocol executed by to construct an context of multi-party computation (MPC) protocols, we use the technique of superinvertible (SI) matrices Regaining Efficiency: Superinvertible Matrices.

Section 5 Packed ADKG

Section 4 Threshold Schnorr

Figure 1: Overview of our framework to construct high-threshold, adaptively secure, and robust threshold Schnorr signatures.

- **New High-Threshold AVSS.** We design a simple high-threshold AVSS scheme and give an adaptive security proof. This gives the first pairing-free, adaptively secure AVSS scheme with quadratic communication cost (cf. Table 2 for a comparison with existing schemes). With our new AVSS scheme and building blocks from the literature, we instantiate our framework, yielding a threshold Schnorr signature scheme with (amortized) communication cost of $O(\lambda n^2 \log n)$ bits per signature.

1.1 Technical Overview

In the following, we provide a technical overview of our work.

**Starting Point: Robust Threshold Schnorr Signatures.** Our starting point is the construction for robust threshold Schnorr signatures by Gennaro et al. [GJKR99]. First, we recall the (single-party) Schnorr signature scheme. For this, let $G = \langle g \rangle$ be a group of prime order $p$ with generator $g$. The secret key is a random element $sk \leftarrow \mathbb{Z}_p$ and the public key is $pk := g^{sk}$. To sign a message $m$, the party samples a random element $r \leftarrow \mathbb{Z}_p$ and computes the signature on $m$ as $\sigma := (R, s)$ where $s = H(pk, R, m) \cdot sk + r \in \mathbb{Z}_p$ and $R = g^r$. Here, $H: \{0, 1\}^* \rightarrow \mathbb{Z}_p$ is a hash function (modeled as a random oracle). Verification of the signature $(R, s)$ is done by checking $g^s = R \cdot pk^c$ where $c := H(pk, R, m)$. Now let us switch to the multi-party setting in which $n$ parties $P_1, \ldots, P_n$ want to jointly create a signature $\sigma$ on $m$. For convenience we assume that parties have already established a $(t_r, n)$-threshold key setup, e.g., by running a distributed key generation (DKG) protocol. Concretely, this means that each party $P_i$ has a share $sk_i$ of the secret key $sk$ such that any set of $t_r + 1$ shares uniquely determine the secret key and the public key shares $pk_i := g^{sk_i}$ are known to all parties. In order to transform the Schnorr signature scheme into a $(t_r, n)$-threshold signature scheme, Gennaro et al.’s insight was to run a DKG protocol to generate shares $r_i$ of a secret nonce $r$ for parties along with associated public shares $R_i = g^{r_i}$ and $R = g^r$. To sign a message $m$, each party $P_i$ computes its share of the signature on $m$ as $\sigma_i := (R_i, s_i)$ where $s_i = H(pk, R, m) \cdot sk_i + r_i$. Verification of a signature share $\sigma_i$ is done with respect to the public key share $pk_i$ and the public nonce share $R_i$. It can be seen that any $t_r + 1$ valid signature shares recover the full signature $\sigma = (R, s)$. Now, a major drawback of this approach is its efficiency: parties need to run a DKG protocol each time they want to sign a new message. Using a state-of-the-art asynchronous DKG protocol in terms of efficiency [AJM+23, DXKLR23], this yields a communication cost of $O(\lambda n^3)$ bits per signature. On the other hand, assuming nonce shares have already been generated, each party can locally compute its signature share and send it to all other parties, which cost only a total of $O(\lambda n^2)$ bits communication.

**Regaining Efficiency: Superinvertible Matrices.** Introduced by Hirt and Nielsen [HN06] in the context of multi-party computation (MPC) protocols, we use the technique of superinvertible (SI) matrices to construct an $(\ell, t_c, t_r, n)$-packed ADKG protocol for more efficient multi-nonce generation. Informally, such a protocol executed by $n$ parties has the following property in the presence of up to $t_c$ malicious parties: it outputs $\ell$ independent keys distributed among the parties such that each of them can be reconstructed independently from the others with reconstruction threshold $t_r$. Our construction has the following parameters: it generates $\ell = t_c + 1$ keys with (arbitrary) reconstruction threshold $t_r < n - t_c$ in the presence of $t_c < n/3$ malicious parties. Our construction (cf. Figure 2) follows the usual flow of an ADKG protocol with some tweaks in the parameters in order to apply an SI matrix at the end. We briefly elaborate on this. Each party $P_i$ samples a random element $s_i \leftarrow \mathbb{Z}_p$ and shares it via an $(t_r, t_c, n)$-threshold asynchronous verifiable secret sharing (AVSS) scheme where $s_i$ lies on some polynomial $f_i \in \mathbb{Z}_p[X]$ of degree $t_r$. Then, parties agree on a set $I \subset [n]$ of $n - t_c$ dealers whose AVSS sharings completed successfully using a consensus tools. Say $I = \{1, \ldots, n - t_c\}$ so that after this phase, each
AVSS schemes are AVSS schemes with Bacho and Loss [BL22] who introduced a new security notion for DKG protocols called Handling Adaptive Corruptions. O bits and send their approval to all parties upon receiving a correct row, and echo other parties' approvals upon guarantee unanimous termination, we employ a Bracha-style termination gadget [Bra84] in which parties send every other party the column polynomials \( C_j \) along the row \( j \). Each party \( P_i \) sends each party \( C_j(i) \) for \( j \neq i \) along with the proofs sent by the dealer. Thus, we set \( \ell := t_c + 1 \). Similar constructions were recently introduced in [BHK+23, GS23, Sho23] for the same purpose of efficient multi-nonce generation. These constructions, however, employ low-threshold AVSS schemes with \( t_c = \frac{n}{3} - 1 \). To the best of our knowledge, the only existing high-threshold AVSS schemes are [AJM+23, AVZ21, KMS20], each of them with its own limitations. Kokoris-Kogias et al.'s AVSS [KMS20] has cubic communication cost, resulting in prohibitive \( \Omega(\lambda n^4) \) communication to share \( t_c + 1 \) nonces. Alhaddad et al. [AVZ21] provide a generic construction for AVSS with quadratic communication cost, but lacking a proof of adaptive security. Abraham et al.'s AVSS [AJM+23] relies on the KZG polynomial commitment scheme [KZG10] that requires pairings (and trusted setup) which is not suitable for Schnorr signatures.

HAVSS: New AVSS Scheme to the Rescue. We take insights from both protocols, Bingo [AJM+23] and HAVSS [AVZ21], and combine certain aspects to obtain a simple high-threshold AVSS scheme called HAVSS. On a high level, our AVSS scheme works as follows. The designated dealer \( P_d \) holds a secret \( s \in \mathbb{Z}_p \) as input that it wants to share among all parties. For this, it samples a bivariate polynomial \( S \in \mathbb{Z}_p[X,Y] \) of degree \( t_c \) in \( X \) and \( t_c \) in \( Y \) such that \( S(0,0) = s \). The goal is to let each party \( P_i \) receive the column polynomial \( C_j(Y) := S(i, Y) \) assigned to it so that it can recover the share \( s_i := S(i,0) \in \mathbb{Z}_p \) of the secret \( s \). Note that the shares \( s_i \) lie on a polynomial \( S(X,0) \in \mathbb{Z}_p[X] \) of degree \( t_r \). We follow a simple two-step approach which results in an \((n \times n)\)-dimensional matrix whose entry at coordinates \( a, b \in [n] \) is \( S(a, b) \). First, the dealer reliably broadcasts Pedersen commitments \( \{ \text{com}_1, \ldots, \text{com}_{t_r+1} \} \) on the column polynomials \( C_1(Y), \ldots, C_{t_r+1}(Y) \), from which parties can locally (by interpolation) derive the commitments \( \{ \text{com}_1, \ldots, \text{com}_n \} \) to all \( n \) column polynomials \( C_1(Y), \ldots, C_n(Y) \). Following this, \( P_d \) sends each party \( P_i \) shares \( \{ C_1(i), C_2(i), \ldots, C_n(i) \} \) on each other party's assigned column polynomial, along with proofs that the openings are correct. This can be thought of as sending to \( P_i \) the evaluations along the row \( R_i(X) := S(X,i) \). Whenever a party \( P_i \) receives a row with correct opening proofs, it sends the other party \( P_j \) the share \( C_j(i) \) (along with the proof sent by the dealer) on its column polynomial \( C_j(Y) \). In this way, it is guaranteed that each party \( P_i \) obtains at least \( t_c + 1 \) shares on its column polynomial \( C_j(Y) \) and can recover its share \( s_i = C_j(0) \in \mathbb{Z}_p \) of the dealer's initial secret \( s \). To guarantee unanimous termination, we employ a Bracha-style termination gadget [Bra84] in which parties send their approval to all parties upon receiving a correct row, and echo other parties' approvals upon seeing a total of \( n - t_c \) approvals. HAVSS has a near-optimal communication cost of \( O(\lambda n^2 \log n) \) per sharing (cf. Table 2). In combination with the aforementioned technique of superinvertible matrices, we are able to construct a packed ADKG protocol that outputs \( \ell = t_c + 1 \in O(n) \) nonces with \( O(\lambda n^2 \log n) \) bits and \( O(1) \) rounds of communication. As a result, we achieve an amortized communication cost of \( O(\lambda n^2 \log n) \) per generated Schnorr signature.

Handling Adaptive Corruptions. To prove adaptive security, our starting point is the recent work of Bacho and Loss [BL22] who introduced a new security notion for DKG protocols called oracle-aided
simulatability. Loosely speaking, this notion states the existence of an efficient simulator $Sim$ that on input $k$ group elements $\xi_1, \ldots, \xi_k \in G$ can simulate an execution of the DKG protocol under adaptive corruptions while having $(k-1)$-time access to a discrete logarithm oracle $DL_{G,g}$. With this notion of security for DKG, they show a reduction from the one-more discrete logarithm (OMDL) assumption [BNPS03] of degree $k$ to the unforgeability of the threshold BLS signature scheme against an adaptive adversary. Their reduction internally runs $Sim$ (on input the OMDL challenge $\xi_1, \ldots, \xi_k \in G$) in order to simulate an execution of the DKG protocol as part of the broader simulation of the unforgeability experiment. To emulate the oracle $DL_{G,g}$ for the simulator $Sim$, the reduction simply forwards any query $Sim$ makes to its own oracle. We want to employ a similar strategy to simulate the executions for multi-nonce and key generation. However, we encounter several challenges when trying to adopt this strategy naively. For the remainder of this overview, we assume for simplicity that parties employ a regular single-output ADKG protocol for nonce generation instead of a packed one.

**Challenges in Our Context.** Very recently, Crites et al. [CKM23a] gave an adaptive security proof for their threshold Schnorr signature scheme under the algebraic OMDL assumption. In their proof, corruption queries are simulated using the oracle $DL_{G,g}$ and signing queries are simulated using honest-verifier zero-knowledge and by programming the random oracle suitably. Omitting details, their simulator essentially samples random signature shares $\sigma_i \leftarrow \mathbb{Z}_p$ for honest parties and retroactively defines the public nonce shares $R_i$ by suitably programming the random oracle. To make this strategy work in our context, the (packed) ADKG protocol $NDKG$ for nonce generation would have to be fully secret in the sense of Gennaro et al. [GJKR09], i.e., there exists an efficient simulator that on input a group element $R \in G$ can simulate an execution of $NDKG$ that terminates with $R$ as public nonce. Unfortunately, without resorting to tools such as secure erasures or non-committing encryption, this notion of security seems to be hopeless to achieve in the context of adaptive corruptions [BL22, GJKR07, JL00]. Therefore, to deal with signing queries under adaptive corruptions, a new approach is required.

**Combining Different Proof Strategies.** In their work, Bellare et al. [BTZ22] provide a security reduction from the OMDL assumption to the security of the FROST1 and FROST2 schemes under a static adversary. Their reduction uses the discrete logarithm oracle $DL_{G,g}$ to answer certain signing queries. Essentially, the values for the nonce $R_j$, the challenge $c_j$, and the public key shares $pk_i$ are fixed and determine the signature shares $\sigma_{j,i}$ by the relation $g^{\sigma_{j,i}} = R_j \cdot pk_i^{c_j}$. We observe that a similar strategy could be useful for our scheme, in particular in combination with previously explained oracle-aided simulatability. And indeed, combining these two proof strategies [BLZ22, BTZ22] (almost!) succeeds: using oracle-aided simulators to simulate executions of $IDKG$ and $NDKG$ along with corruption queries, and at the same time using the oracle $DL_{G,g}$ separately to answer signing queries. However, trying to employ this approach as it currently stands, we exceed the number of allowed queries to the oracle $DL_{G,g}$ prescribed by the OMDL challenge: assume the oracle in question uses $DL_{G,g}$ on input $g^{\sigma_{j,i}}$ to answer a signing query for party $P_i$ and nonce $R_j$. If $P_i$ gets corrupted later on, the simulators for IDKG and the $j$-th execution of $NDKG$ that generated $R_j$ might make discrete logarithm queries such that they can internally compute the secret key share $sk_i$ and the secret nonce share $r_{j,i}$, respectively. Three discrete logarithm oracle queries have been made to return the values $\sigma_{j,i}, sk_i, r_{j,i}$, although by the identity $\sigma_{j,i} = c_j \cdot sk_i + r_{j,i}$ two queries would suffice. To resolve this issue we have to (i) adapt the original definition of oracle-aided simulatability delicately and (ii) cleverly design the reduction to limit the number of its queries to $DL_{G,g}$.

**1.2 More on Related Work**

We discuss related work on threshold signatures and DKG. In Supplementary Material Section A, we discuss related work on AVSS and threshold Schnorr signatures with a focus on robustness, high-threshold, and efficiency.

**Threshold Signatures.** Most of the threshold signature schemes [BCK+22, CKM+23b, KY02, LP01] focus on threshold DSA/ECDSA and threshold Schnorr [CGG+20, CM21, CM23a, DOK+20, GG18, KG20], mainly due to their significance in blockchain systems and cryptocurrency wallets. Among the threshold Schnorr signatures, only the work [CKM23a] provides adaptive security. Further, several protocols for threshold RSA signatures were proposed [ADN06, KY02, Sho00] from which only [ADN06]
we present our new high-threshold AVSS scheme and prove it adaptively secure. In Section 7, we give a generic construction for a high-threshold, robust, and efficient threshold Schnorr signature scheme and prove it adaptively secure in the AGM. In Section 5, we give a generic construction for an efficient packed ADKG protocol and prove it adaptively secure from its building blocks. In Section 6, we present our new high-threshold AVSS scheme and prove it adaptively secure. In Section 7, we instantiate our framework to obtain HARTS, and evaluate the communication and round complexity of it. In Supplementary Material Section A, we discuss related work on AVSS schemes and threshold Schnorr signatures. In Supplementary Material Section B, we cover additional preliminaries relevant for the paper. Due to space constraints, we defer security proofs to Supplementary Material Section C.

### 1.3 Outline of the Paper

The paper is organized as follows. In Section 2, we define relevant preliminaries. In Section 3, we define the model of syntax and security of a robust threshold signature scheme relevant for this work. In Section 4, we give a generic construction for a high-threshold, robust, and efficient threshold Schnorr signature scheme and prove it adaptively secure in the AGM. In Section 5, we give a generic construction for an efficient packed ADKG protocol and prove it adaptively secure from its building blocks. In Section 6, we present our new high-threshold AVSS scheme and prove it adaptively secure.
General Notation. Let $\lambda$ denote the security parameter. Throughout the paper, we assume that global parameters $\text{par} := (\mathbb{G}, p, g)$ implicitly parameterized by $\lambda$ are fixed and known to all parties. Here, $\mathbb{G}$ is a cyclic group of prime order $p$ generated by $g$. For two integers $a \leq b$, we define the set $[a, b] := \{a, \ldots, b\}$; if $a = 1$, we denote this set by $[b]$, and if $a = 0$, we denote it by $[0]$. For an element $x$ in a finite set $S$, we write $x \leftarrow S$ to denote that $x$ was sampled from $S$ uniformly at random. All our algorithms may be randomized, unless stated otherwise. We use the acronym PPT to mean probabilistic polynomial-time. By $x \leftarrow A(x_1, \ldots, x_n)$ we denote running algorithm $A$ on inputs $(x_1, \ldots, x_n)$ and uniformly random coins and then assigning its output to $x$.

Adversarial and Network Model. We consider a complete network of $n$ parties $P_1, \ldots, P_n$ (modeled as PPT machines) connected by bilateral private and authenticated channels\(^3\). We consider an asynchronous network model, i.e., any message can be delayed arbitrarily under the constraint that messages sent between correct parties must eventually be delivered. We consider an adversary who can corrupt up to $t_c < n/3$ parties maliciously and may cause them to deviate from the protocol arbitrarily. We refer to $t_c$ as the corruption threshold and to $t_r \in [t_c, n - t_c)$ as the reconstruction threshold. Further, the adversary is strongly adaptive and can choose its corruptions at any time during the protocol execution. When it corrupts a party, it can delete or substitute any undelivered messages that this party sent while being correct. We refer to the correct parties as honest and to the malicious parties as corrupt.

Public Key Infrastructure. We assume that parties have established a bulletin board public key infrastructure (PKI) before the protocol execution. Concretely, this means that every party $P_i$ has a verification-signing key pair $(vk_i, sik_i)$ for a digital signature scheme, where $vk_i$ is known to all parties but $sik_i$ is known only to $P_i$. For this, we assume that each party generates its keys locally (where corrupt parties may choose their keys arbitrarily) and then makes its verification key known to everybody using a public bulletin board. These keys are used to provide authentication. In particular, we assume that parties sign each message before they send it to other parties.

Algebraic Group Model. In the algebraic group model (AGM) [FKL18], all algorithms are treated as algebraic: whenever an algorithm outputs a group element, it must also provide a representation of that element with respect to all of the inputs the algorithm has received so far. Formally, an algorithm $A$ is called algebraic (over a group $\mathbb{G}$) if for all group elements $h \in \mathbb{G}$ that $A$ outputs, it additionally outputs a vector $\mathbf{z} = (z_1, \ldots, z_m)$ of integers such that $h = \prod_{i \in [m]} g_i^{z_i}$, where $\zeta = (g_1, \ldots, g_m) \in G^m$ is the list of group elements $A$ has received so far.

Computational Assumptions. We rely on the one-more discrete logarithm (OMDL) assumption [BNPS03] for our security proofs. Throughout the paper, we denote by $\text{DL}_{\mathbb{G}, g}$ an oracle that on input $\zeta := g^z \in \mathbb{G}$ returns the discrete logarithm $z \in \mathbb{Z}_p$ of $\zeta$ to base $g$.

**Definition 2.1** (OMDL Assumption). Let $\mathbb{G}$ be a cyclic group of prime order $p$ generated by $g$ and $\text{DL}_{\mathbb{G}, g}$ as defined above. For an algorithm $A$ and $k \in \mathbb{N}$, we consider the following experiment:

- **Offline Phase.** Sample $(z_1, \ldots, z_k) \leftarrow \mathbb{Z}_p^k$ and set $\xi_i := g^{z_i} \in G$ for all $i \in [k]$.
- **Online Phase.** Run $A$ on input $(\mathbb{G}, p, g)$ and $(\xi_1, \ldots, \xi_k)$. Here, $A$ gets access to the oracle $\text{DL}_{\mathbb{G}, g}$.
- **Winning Condition.** Let $(z'_1, \ldots, z'_k)$ denote the output of $A$. Return 1 if (i) $z'_i = z_i$ for all $i \in [k]$, and (ii) $\text{DL}_{\mathbb{G}, g}$ was queried at most $k - 1$ times during the online phase. Otherwise, return 0.

We say that the one-more discrete logarithm assumption of degree $k$ holds relative to $(\mathbb{G}, p, g)$ if for any PPT algorithm $A$, the probability that the above experiment outputs 1 is negligible in $\lambda$.

Further, the discrete logarithm assumption (DLOG) is the one-more discrete logarithm assumption of degree $k = 1$.

### 2.1 Cryptographic and Consensus Primitives

In this section, we formally define syntax and security notions of the cryptographic and consensus primitives used in the paper.

\(^3\)When implementing those channels, one has to make sure that they are secure in the presence of adaptive corruptions. For an example implementation, we refer to the early works [BH93, JL00].
Multivalued Validated Byzantine Agreement. A *multivalued validated Byzantine agreement (MVBA) protocol* [CKPS01] allows a set of parties, each holding an input \(v_i \in V\) from a value set \(V\) with \(|V| \geq 2\), to agree on a common output value \(v \in V\) satisfying a predefined external validity function \(\text{Val}: V \to \{0, 1\}\). A value \(v \in V\) is said to be *externally valid* if \(\text{Val}(v) = 1\). We formally define an MVBA protocol as follows.

**Definition 2.2 (MVBA Protocol).** Let \(\Pi\) be a protocol executed by \(n\) parties \(P_1, \ldots, P_n\), where each party \(P_i\) holds \(v_i \in V\) as input, and let \(\text{Val}: V \to \{0, 1\}\) be an external validity function. We say that \(\Pi\) is a \((t_c, n)\)-secure MVBA protocol if whenever at most \(t_c\) parties are corrupted the following properties hold:

- **External Validity.** If every honest party’s input is externally valid, then every honest party \(P_i\) outputs an externally valid value \(v_i\).
- **Consistency.** If every honest party’s input is externally valid, then all honest parties output the same value \(v\).
- **Termination.** If every honest party’s input is externally valid, then every honest party \(P_i\) terminates with an output value \(v_i\).

Hereafter, we write MVBA to denote a generic \((t_c, n)\)-secure multivalued validated Byzantine agreement protocol.

Reliable Broadcast. A *reliable broadcast (RBC) protocol* [Bra84] allows a designated party \(P_s\) (called the sender) to consistently distribute a message among all parties. In contrast to synchronous broadcast, reliable broadcast does not require full termination. We formally define a reliable broadcast protocol as follows.

**Definition 2.3 (RBC Protocol).** Let \(\Pi\) be a protocol executed by \(n\) parties \(P_1, \ldots, P_n\), where a designated sender \(P_s\) holds \(v \in V\) as input. We say that \(\Pi\) is a \((t_c, n)\)-secure RBC protocol if whenever at most \(t_c\) parties are corrupted the following properties hold:

- **Validity.** If the sender \(P_s\) is honest and holds \(v\) as input, then every honest party \(P_i\) outputs \(v_i = v\).
- **Consistency.** All honest parties that output a value output the same value \(v'\).
- **Totality.** If an honest party outputs a value, then every honest party eventually outputs a value.

Hereafter, we write RBC to denote a generic \((t_c, n)\)-secure RBC protocol.

Superinvertible Matrices. A *superinvertible (SI) matrix of dimension \((\ell, k)\) with \(k \geq \ell\)* [HN06] is a matrix \(A \in \mathbb{Z}_p^{\ell \times k}\) over some field \(\mathbb{Z}_p\) with the property that each of its \((\ell \times \ell)\)-dimensional square submatrix \(A_i\) is invertible. Large classes of superinvertible matrices are given in [BHK+23, GS23]. Looking ahead, each party \(P_i\) applies the SI matrix \(A\) of dimension \((\ell, k) := (n - 2t_c, n - t_c)\) to its \(k\) secret shares \(f_1(i), \ldots, f_k(i)\) that it received from different completed AVSS sharings. The result is \(\ell\) new secret shares \(r_1(i), \ldots, r_\ell(i)\) with the property that if at least \(\ell\) input secrets are independent and uniformly random, then the \(\ell\) output secrets are also guaranteed to be independent and uniformly random.

Asynchronous Verifiable Secret Sharing. An *asynchronous verifiable secret sharing (AVSS) scheme* [BCG93, CR93] consists of two protocols \text{Share} and \text{Rec} which allow a designated dealer to share a secret \(s\) over some field \(\mathbb{Z}_p\) among all parties using Shamir secret sharing. Here, the threshold \(t_s \in [t_c, n - t_c]\) specifies the degree of the shared polynomial \(f\). In our definition of an AVSS scheme, we require a reconstruction protocol in which parties reconstruct exponentiated evaluations of the polynomial \(f\) at the points \(\{0, 1, \ldots, n\}\). We formally define an AVSS scheme over the group \((\mathbb{G}, p, g)\) as follows.

**Definition 2.4 ((t_c, t_r, n)-Threshold AVSS Scheme).** Let \(\Pi = (\text{Share}, \text{Rec})\) be a pair of protocols executed by \(n\) parties \(P_1, \ldots, P_n\), where a designated dealer \(P_d\) holds a secret \(s \in \mathbb{Z}_p\) as input. Upon completion of \text{Share} parties only maintain a state and do not output anything. Parties can then call \text{Rec} with their state and output a tuple of \(n + 1\) elements in \(\mathbb{G}\) and an element in \(\mathbb{Z}_p\). We say that \(\Pi\) is a complete \((t_c, t_r, n)\)-threshold AVSS scheme if whenever at most \(t_c\) parties are corrupted the following properties hold:

8
Remark 2.5 In this definition, we leave out a notion of secrecy and postpone it to Section 6 instead. We were run in parallel. For the definition, we use the group

The basic idea is to realize the same functionality as if

coincide in synchrony.

3.1 Packed Asynchronous DKG

In our AVSS construction, we use non-interactive zero-knowledge (NIZK) proofs [BFM88]. Informally, a non-interactive proof system for an NP relation \( \mathcal{R} \) with respect to a random oracle \( \mathcal{H} \) is a pair of PPT algorithms \( PS = (PProve, PVer) \), where \( PProve^H \) takes a statement \( x \) and a witness \( w \) with \( (x, w) \in \mathcal{R} \) as input and outputs a proof \( \pi \), and \( PVer^H \) takes the statement \( x \) and the proof \( \pi \) as input and decides to accept or reject. Completeness requires that honestly computed proofs for \( (x, w) \in \mathcal{R} \) are accepted, whereas soundness requires that no malicious prover can find an accepting proof for a false statement \( x \), i.e., a statement such that \( (x, w) \notin \mathcal{R} \) for all \( w \). Further, zero-knowledge requires that there is a simulator that can simulate proofs without knowing \( w \) by programming the random oracle \( \mathcal{H} \). Finally, the system is a proof of knowledge, if there is an extractor that can extract the witness from any proof provided by the adversary. To do so, the extractor is allowed to observe the random oracle queries made by the adversary. Our definitions hence model online-extraction, which is reasonable in the algebraic group model. We postpone formal definitions to Supplementary Material Section B.

3 Packed Asynchronous DKG and Threshold Signatures

In this section, we introduce the notion of a packed asynchronous distributed key generation (ADKG) protocol and define our model of syntax and security of a threshold signature scheme.

3.1 Packed Asynchronous DKG

In a regular distributed key generation (DKG) protocol, a set of mutually distrusting parties securely establish a public-secret key pair without relying on a trusted dealer. At the end of the protocol, the public key is output in the clear, whereas the secret key is kept as a virtual secret distributed among all parties. This shared secret key can then be used for threshold cryptosystems, such as threshold signatures or threshold encryption, without ever being explicitly reconstructed. When the underlying network is asynchronous, we call it an asynchronous DKG (ADKG). In the following, we introduce and define the notion of an \((\ell, t_c, t_r, n)\)-packed ADKG protocol which allows \( n \) parties out of which at most \( t_c \) are corrupted to generate \( \ell \geq 1 \) independent shared keys each with reconstruction threshold \( t_r \in [t_c, n - t_c] \) in a way that is potentially more efficient than just executing \( \ell \) instances of an ADKG protocol in parallel. The basic idea is to realize the same functionality as if \( \ell \) independent instances of an ADKG protocol were run in parallel. For the definition, we use the group \( G \) specified by \( par = (G, p, g) \). Hereafter, fix the parameter \( \delta_c := t_r + 1 - t_c \). The subscript stands for \textit{asynchrony}, since the two thresholds \( t_r \) and \( t_c \) coincide in synchrony.
Definition 3.1 ((ℓ, t_c, t_r, n)-Packed ADKG Protocol). Let Π be a protocol executed by n parties
P_1, ..., P_n, where for each j ∈ [ℓ], P_j outputs a secret key share r_{j,i}, a vector of public key shares
(R_{j,1}, ..., R_{j,n}), and a public key R_j. We say that Π is an oracle-aided secure (ℓ, t_c, t_r, n)-packed ADKG
protocol if whenever at most t_c parties are corrupted the following properties hold:

- Consistency. For each j ∈ [ℓ], all honest parties output the same public key R_j = g^{r_{j,i}} and the same
  vector of public key shares (R_{j,1}, ..., R_{j,n}).

- Correctness. For each j ∈ [ℓ], there exists a unique polynomial f_j ∈ Z_p[X] of degree t_r such that
  for all i ∈ [n], r_{j,i} = f_j(i) and R_{j,i} = g^{r_{j,i}}. Moreover, R_j = g^{f_j(0)}.

- Termination. If all honest parties participate in the protocol execution, then all honest parties
  terminate with an output.

- Oracle-aided Simulatability. There exists k ∈ poly(λ) with k ≥ ℓ(t_r + 1) such that for any PPT
  algorithm A, there exists an algebraic PPT simulator Sim that on input 𝜖 := (g^{a_1}, ..., g^{a_k}) ∈ G^k
  makes k' := k − ℓδ_a queries to the oracle DL_{G,G} and such that:

  - Syntax. Sim simulates the role of the honest parties in an execution of Π. At the end of the
    simulation, Sim outputs the public keys R_1, ..., R_ℓ and public key shares (R_{j,1}, ..., R_{j,n}) for
    all j ∈ [ℓ].

  - Queries upon Corruption. Denote by C ⊂ [n] the dynamic set of corrupted parties. Once the
    first honest party outputs (R_{j,1}, ..., R_{j,n}) for all j ∈ [ℓ], the following holds. Upon corruption
    query i ∈ [n] \ C, Sim invokes DL_{G,G} on inputs (R_{j,i}) for all j ∈ [ℓ] among (possibly) other
    input elements. Conversely, it does not query R_{j,i} for any j ∈ [ℓ] before that corruption.

  - Query Independence. Let C be as before and H := [n] \ C. Assume that |C| = t_c after a simulation
    of Π. For i ∈ [k − ℓδ_a], denote by g_i ∈ G the i-th query to DL_{G,G} and let (a_1, a_1, ..., a_k)
    be the corresponding algebraic vector, i.e., g_i = g^{a_1} · ξ_1^{a_1} · ... · ξ_k^{a_k}. Further, denote by
    (b_1, b_1, ..., b_k) the algebraic vector of the public key share R_i for all i = (j, i) ∈ ℓ \ H. Then
    for all I := ℓ \ H with |I| = δ_a, the following matrix is invertible over Z_p

    \[ L(I, C) := \begin{pmatrix}
    a_{1,1} & a_{1,2} & \cdots & a_{1,k} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{k-\ell \delta_a,1} & a_{k-\ell \delta_a,2} & \cdots & a_{k-\ell \delta_a,k} \\
    b_{1,1} & b_{1,2} & \cdots & b_{1,k} \\
    \vdots & \vdots & \ddots & \vdots \\
    b_{k-\ell \delta_a,1} & b_{k-\ell \delta_a,2} & \cdots & b_{k-\ell \delta_a,k}
    \end{pmatrix} \in Z_p^{k \times k}, \]

  where the indices i_1 range over the set \bigcup_{j \in [\ell]}(\{j\} \times I). Whenever Sim completes a simulation
  of an execution of Π, we call L(I, C) the simulatability matrix of Sim (for this particular
  simulation and the set I). Further, we call k a simulatability factor of Π.

  - Bad Event. There is an event Bad, such that for any PPT algorithm A, the probability of Bad
    in an execution of Π with adversary A is negligible.

  - Indistinguishability. Denote by view_{A,Π} the view of A in an execution of Π. Denote by view_{A,ξ,Sim}
    the view of A when interacting with Sim on input ξ. Then, the distributions (ξ, view_{A,Π}) and
    (ξ, view_{A,ξ,Sim}) where ξ ← G^k and both distributions are conditioned on \neg Bad are statistically
    close.

For ℓ = 1 (when the packing is trivial), we simply call Π a (t_c, t_r, n)-threshold ADKG protocol (over
(G, p, g)). Further, we call ℓ ≥ 1 the packing parameter.

Remark 3.2 In our above definition of oracle-aided security, we do not require the simulator Sim to
terminate once it outputs the public keys, but only after it has made the required k − ℓδ_a calls to the oracle
DL_{G,G} (conditioned on the simulation of Π being completed). The reason for this being that adaptive
corruptions can happen even after termination of the DKG protocol, e.g., when the DKG is part of a
more complex protocol such as a threshold signature scheme.
Discussion. For simplicity, we consider only the case $\ell = 1$ in this discussion. First, note that consistency, correctness, and termination notions are in line with established definitions from the literature for DKG protocols. In addition, our definition is built upon the oracle-aided algebraic security (OAAS) notion from [BL22] which is defined for DKG protocols with a single threshold $t$. We adjust their definition in several ways. First, we extend it to the $(t_c, t_r, t_e)$-dual-threshold setting which is often relevant in asynchronous networks\footnote{A dual-threshold might also be interesting in the synchronous setting, e.g., when trading off liveness with enhanced security or for responsiveness [MCK20].}. Second, we state a more precise requirement on the behavior of the oracle-aided simulator $\text{Sim}$, which is explained below. This allows us to make the DKG definition amenable for a more general framework of adaptively secure threshold signatures, such as threshold Schnorr and threshold BLS.

We begin with our new property “Queries upon Corruption” that specifies $\text{Sim}$’s behavior when a corruption $i \in \mathcal{H}$ happens after the public key shares are defined from the protocol. Specifically, we require $\text{Sim}$ to call $\text{DL}_{c,g}$ on input $R_i = g^{f(i)}$ only upon that event and not before. The intuition for this being that any reasonable simulator should not know the secret key share $f(i)$ of that party $P_i$ before the corruption happens; not surprisingly, all the OAAS simulators constructed in [BL22] have this property. We proceed with the property “Query Independence” that upon [BL22] takes a dual-threshold $(t_c, t_r)$ into consideration. For this, we introduce the set $\mathcal{I}$. To understand this, we observe that the idea behind the invertibility of the matrix $L(C)$ as given in their paper is that the joint secret key $f(0)$ should not be known to the simulator even after $t_c$ corruptions happened. In the dual-threshold setting, we want this property to hold even if $t_c = t_e = \delta_a - 1$ additional secret key shares are leaked. That is why we require the algebraic vectors of any $|I| = \delta_a$ additional public key shares to be independent from already leaked data. Finally, note that [BL22] does not take computationally indistinguishable simulations into consideration. For better composability, we separate the computational and statistical aspects by introducing the property “Bad Event” and making the property “Indistinguishability” statistical.

3.2 Robust Threshold Signatures

In the following, we introduce the syntax and security notions for robust threshold signature schemes. These are in line with established definitions but adopted to the structure of our protocol.

Syntax and Completeness. In our model, a threshold signature scheme has the following structure. First, all parties $P_1, \ldots, P_n$ run a regular $(t_c, t_r, t_e)$-threshold ADKG protocol denoted by IDKG (called the initial ADKG). Having done this, each party $P_i$ holds a secret key share $sk_i$ and the public key shares $pk_1, \ldots, pk_n$ of other parties along with the public key $pk$. Following this, parties repeatedly run two parallel instances of an $(\ell, t_c, t_r, t_e)$-packed ADKG protocol denoted by NDKG in the background. The keys generated by these executions are interpreted as nonces. In particular, after each parallel execution of the Nonce-ADKG protocol NDKG, the parties obtain $2\ell$ new and independent nonces. To simplify matters, we assume that the nonces are output in pairs $(R_j, R'_j)$. For each such public nonce $R_j$ (respective $R'_j$), each party $P_i$ also obtains its secret nonce share $r_{j,i}$ (respective $r'_{j,i}$) along with the public nonce shares $(R_{j,1}, \ldots, R_{j,n})$ (respective $(R'_{j,1}, \ldots, R'_{j,n})$) of other parties. For signing, we adapt the double-nonce approach introduced by Komlo and Goldberg [KG20] in order to prevent concurrent session attacks [BLL*21, DEF+19]\footnote{Nick et al. [NRS21] introduced essentially the same technique at the same time to construct a two-round Schnorr multi-signature with a rigorous security analysis.]. That is, we assume that parties have agreement on a previously generated but never before used nonce pair $(R_j, R'_j)$ and use the effective nonce $R_j = R_jR'_j$ to sign a message $m$ where the scalar $b \in \mathbb{Z}_p$ is derived from a random oracle $H_{\text{nonce}}$ as $b = H_{\text{nonce}}(pk, R_j, R'_j, m)$. Upon such a signing request, each party derives the effective nonce shares $(\hat{R}_{j,1}, \ldots, \hat{R}_{j,n})$ and its effective secret nonce share $\hat{r}_{j,i}$ analogously. In this light, the protocol essentially becomes non-interactive: When party $P_i$ wants to sign message $m$ with respect to effective nonce $\hat{R}_j$, it runs an algorithm $SSign$ using its secret key $sk_i$, and its secret nonce share $\hat{r}_{j,i}$ on message $m$. As a result, the party obtains a signature share $\sigma$ that it sends to all other parties. This signature share can be verified with respect to the parties public key share $pk_i$ and the public nonce share $\hat{R}_{j,i}$. Upon receiving $t_e + 1$ valid signature shares, a party can now be verified with respect to the public key $pk$ only. From this explanation of the execution model, it is clear that we can define such a threshold signature scheme by specifying the initial ADKG protocol,
the packed ADKG protocol for nonce generation, and algorithms for signing and verification similar to a
non-interactive threshold signature scheme [BL22, LJY14].

Definition 3.3 (Threshold Signature Scheme). An \((\ell, t_c, t_r, n)\)-threshold signature scheme is a tuple of
PPT protocols and algorithms \(\Sigma = (\text{IDKG}, \text{NDKG}, \text{SSign}, \text{SVer}, \text{Comb}, \text{Ver})\) with the following syntax:

- IDKG: This is a \((t_c, t_r, n)\)-threshold asynchronous DKG protocol as specified in Definition 3.1.
- NDKG: This is an \((\ell, t_c, t_r, n)\)-packed asynchronous DKG protocol as specified in Definition 3.1.
- SSign: The signature share generation algorithm takes as input a secret key share \(sk_i \in \mathbb{Z}_p\), a
  public key \(pk \in \mathcal{G}\), two secret nonce shares \(r_i, r'_i \in \mathbb{Z}_p\), two public nonces \(R, R' \in \mathcal{G}\), and a message
  \(m \in \{0,1\}^*\). It outputs a signature share \(\sigma_i\).
- SVer: The signature share verification algorithm takes as input a public key \(pk \in \mathcal{G}\), a public
  key share \(pk_i \in \mathcal{G}\), two public nonces \(R, R' \in \mathcal{G}\), two public nonce shares \(R_i, R'_i \in \mathcal{G}\), a message
  \(m \in \{0,1\}^*\), and a signature share \(\sigma_i\). It outputs 1 (accept) or 0 (reject).
- Comb: The signature share combining algorithm takes as input two public nonces \(R, R' \in \mathcal{G}\), a
  message \(m \in \{0,1\}^*\), and a set \(S\) of \(t_r + 1\) signature shares \((\sigma_i, i)\) with corresponding indices. It
  outputs either a signature \(\sigma\) or \(\bot\).
- Ver: The signature verification algorithm takes as input a public key \(pk \in \mathcal{G}\), a message \(m \in \{0,1\}^*\),
  and a signature \(\sigma\). It outputs 1 (accept) or 0 (reject).

Further, we require the following correctness properties to hold:

- For any \(m \in \{0,1\}^*\), any \(sk_i, r_i, r'_i \in \mathbb{Z}_p\), and any \(pk, R, R' \in \mathcal{G}\), we have
  \[\Pr[SVer(pk, pk_i, R, R', R_i, R'_i, m, SSign(sk_i, pk, r_i, r'_i, R, R', m)) = 1] = 1,\]
  where \(R_i = g^{r_i}, R'_i = g^{r'_i}\), and \(pk_i = g^{sk_i}\).
- For all sets \(I \subset [n]\) with \(|I| = t_r + 1\), all messages \(m \in \{0,1\}^*\), all polynomials \(f, r, r' \in \mathbb{Z}_p[X]\) of
degree \(t_r\), and all possible sets \(S\) of the form \(\{(\sigma_i, i)\}_{i \in I}\),
  \[(\forall i \in I : SVer(pk, pk_i, R, R', R_i, R'_i, m, \sigma_i) = 1) \implies Ver(pk, m, Comb(R, R', m, S)) = 1,\]
  where \(pk = g^{r(0)}, R = g^{r(0)}, R' = g^{r'(0)}, pk_i = g^{f(i)}, R_i = g^{r(i)}\), and \(R'_i = g^{r'(i)}\) for all \(i \in I\).

We emphasize that our definition models a robust threshold signing protocol [GJKR07]. The reason for
this is that the protocol NDKG terminates with distributed nonces each having reconstruction threshold
\(t_r\). Since there are at least \(n - t_c \geq t_r + 1\) honest parties in the system, it is guaranteed for every honest
party to obtain enough valid signature shares (even if no corrupt party sends a valid signature share or
anything at all) and thus to compute the full signature.

Security Model. We define the security of a threshold signature scheme following our syntax. The
established security definition for non-interactive adaptively secure threshold signatures [BL22, LJY14]
allows the adversary to adaptively ask for signature shares and corruptions for up to \(t_c\) parties of its
choice. In the end, the adversary succeeds if it outputs a message \(m^*\) and a valid signature \(\sigma^*\) for it
such that it obtained at most \(t_c\) signature shares for \(m^*\). In the synchronous setting, the thresholds
for corruption and reconstruction coincide\(^6\). As we work in an asynchronous network, we adjust their
definition to a dual-threshold: the protocol should be resistant against \(t_c\) corruptions while providing
security for even up to \(t_r\) leaked signature shares. Additionally, we let the adversary freely decide when
parties execute a new (parallel) instance of the Nonce-ADKG protocol in which he also participates.
Finally, signature shares are generated with respect to a specific nonce pair that has been generated but
not used previously and is specified by the adversary.

\(^6\)To guarantee robustness (i.e., guaranteed output delivery) in synchrony, \(t < n/2\) is required. As a result, the optimal
thresholds for corruption and reconstruction have to coincide.
We say that SME is unforgeable under chosen message attacks (or UF-CMA secure) if for any PPT algorithm A, the probability that the above experiment outputs 1 is negligible in λ.

4 Robust Threshold Schnorr Signatures

In this section, we provide a generic construction for a high-threshold, robust, and efficient threshold Schnorr signature scheme and analyze its security.

4.1 Our Construction

In the following, we give a generic construction for a robust threshold Schnorr signature scheme (also refer to Figure 2). Our construction is based on the technique introduced by Gennaro et al. [GJKR99, GJKR07]. In our work, we observed that in order to obtain a robust threshold Schnorr signature, the nonce itself

\[\text{Definition 3.4 (Unforgeability Under Chosen Message Attack). Let } \Sigma = (\text{IDKG}, \text{NDKG}, \text{SSign}, \text{SVer}, \text{Comb}, \text{Ver}) \text{ be an } (t, l, c, t_r, n)\text{-threshold signature scheme. For an algorithm } A, \text{ we consider the following experiment:}

1. Setup. Initialize a corruption set \( C := \emptyset \) and a signing query set \( Q := \emptyset \). For each party \( P_i \), \( i \in [n] \), initialize an empty state \( St_i \). Run A on input \( \pi \). At any point throughout the experiment, A can issue corruption queries:

   - Corrupt Query. A corrupts a party by submitting an index \( i \in [n] \setminus C \). In this case, set \( C := C \cup \{ i \} \) and return the internal state \( St_i \) of party \( P_i \) to A. Henceforth, A has full control over \( P_i \).

2. Initial Asynchronous DKG. Initialize an execution of DKG among parties \( P_1, \ldots, P_n \), where at any point in time, A controls all parties \( P_i \) with \( i \in C \), and the experiment simulates all other parties following the protocol, and adds their respective state to \( St_i \). Denote by \( pk \) and \( (pk_1, \ldots, pk_n) \) the public key and public key shares determined by DKG. Denote by \( sk_i \) for all \( i \in [n] \setminus C \) the secret key shares of the honest parties. When the execution of DKG is terminated, add \( sk_i \) to \( St_i \) for all \( i \in [n] \setminus C \).

3. Online Phase. During this phase, A gets additional access to oracles that answer queries of the following types:

   - Nonce-ADKG Query. When A queries this oracle, a new parallel protocol execution\(^7\) of DKG among the parties \( P_1, \ldots, P_n \) is initiated. As for the initial distributed key generation, A controls all parties \( P_i \) with \( i \in C \), and the experiment simulates all other parties following the protocol, and adds their respective state to \( St_i \). When this protocol terminates for the \((k+1)\)-th time, let \((R_{k+1}, R_{k+1}'), \ldots, (R_{k+\ell}, R_{k+\ell}'))\) be the respective public nonce pairs, and let \( R_{k+\ell+1}, \ldots, R_{k+\ell+j}, n \) and \( R_{k+\ell+1}', \ldots, R_{k+\ell+j}', n \) for each \( j \in [\ell] \) be the respective public nonce shares. Further, for each party \( P_i \) with \( i \in [n] \setminus C \), let \((r_{k+1}, r_{k+1}'), \ldots, (r_{k+\ell}, r_{k+\ell}'))\) be the respective secret nonce share pairs that party \( P_i \) obtains. These secret nonce share pairs are added to \( St_i \).

   - Signing Query. When A submits a new tuple \((i, j, m) \notin Q\) for an \( i \in [n] \setminus C \) and nonce index \( j \) such that \((R_j, R_j')\) is defined, then: If there is an \( m' \neq m \) and an \( i' \in [n] \) such that \((i', j, m') \in Q\), then return \( \perp \). Otherwise, set \( Q := Q \cup \{(i, j, m)\} \) and return \( \sigma \leftarrow \text{SSign}(sk_i, pk, r_j, r_j', R_j, R_j', m) \).

4. Winning Condition. When A outputs a message \( m^\ast \) and a signature \( \sigma^\ast \), let \( S \subset [n] \) denote the subset of parties for which A made a signing query for \( m^\ast \), i.e., let

\[ S := \{ i \in [n] \mid \exists j \text{ s.t. } (i, j, m^\ast) \in Q \}. \]

Return 1 if \(|C| \leq t, |C \cup S| \leq t_r, \text{ and } \text{Ver}(pk, m^\ast, \sigma^\ast) = 1 \). Otherwise, return 0.

We say that \( \Sigma \) is unforgeable under chosen message attacks (or UF-CMA secure) if for any PPT algorithm A, the probability that the above experiment outputs 1 is negligible in \( \lambda \).
should be computed in a distributed threshold fashion realized via a DKG protocol. Building upon this idea, we implement the DKG protocol for nonce generation with a packed ADKG protocol. For this, let \( t, t_c, t_r, n \in \mathbb{N} \) be natural numbers such that \( t_c < n/3 \) and \( t_r \in [t_c, n - t_c] \).

**Construction.** Let IDKG be a \((t_c, t_r, n)\)-threshold ADKG protocol and let NDKG be an \((\ell, t_c, t_r, n)\)-packed ADKG protocol. Further, let \( H, H_{\text{non}} : \{0,1\}^* \to \mathbb{Z}_p \) be two hash functions (modeled as random oracles). Then, the threshold Schnorr signature scheme \( \text{SchnorrTS}[\text{IDKG}, \text{NDKG}] = (\text{IDKG}, \text{NDKG}, \text{SSign}, \text{SVer}, \text{Comb}, \text{Ver}) \) is defined as follows:

- **SSign**\((sk_i, pk, r_i, R_i, R', m)\): Compute \( b := H_{\text{non}}(pk, R, R', m) \) and the effective nonce \( \hat{R} := RR' \). Further, compute \( \hat{r}_i := r_i + br_i' \) and \( c := H(pk, \hat{R}, m) \). Return the signature share \( \sigma_i := c \cdot sk_i + \hat{r}_i \in \mathbb{Z}_p \).  

- **SVer**\((pk, pk_i, R, R', R_i, R_i', m, \sigma_i)\): Compute \( b := H_{\text{non}}(pk, R, R', m) \), the effective nonce \( \hat{R} := RR' \), the effective nonce share \( \hat{R}_i := R_iR'_i \), and further \( c := H(pk, \hat{R}, m) \). Return 1 if \( pk^c \cdot \hat{R}_i = g^{\sigma_i} \) and 0 otherwise.  

- **Comb**\((R, R', m, S)\): Parse \( S \) as a set of \( t_r + 1 \) signature shares \( \{\sigma_i, i\} \) corresponding to indices. Denote the set of these indices by \( I \). Compute \( s := \sum_{i \in I} L_{i, \hat{r}_i} \sigma_i \), where \( L_{i, \hat{r}_i} \) denotes the \( i \)-th Lagrange coefficient for the set \( I \). Further, compute the effective nonce \( \hat{R} := RR' \) where \( b := H_{\text{non}}(pk, R, R', m) \). Return the signature \( \sigma := (\hat{R}, s) \).  

- **Ver**\((pk, m, \sigma)\): Parse \( \sigma = (\hat{R}, s) \). Return 1 if \( pk^c \cdot \hat{R} = g^s \) and 0 otherwise.

### 4.2 Security Analysis

We proceed with the security proof of SchnorrTS[IDKG, NDKG] assuming oracle-aided security of IDKG and NDKG. For this, we give a security reduction from the hardness of the OMDL assumption to the unforgeability (cf. Definition 3.4) of our threshold signature scheme SchnorrTS[IDKG, NDKG].

**Proof Intuition.** We give here an intuition for our proof. The key idea of our reduction is to embed the OMDL challenge \( \xi \) into the public keys \( pk_1, \ldots, pk_\ell \) of parties that are output by IDKG and into the public nonces \( \{R_i, R'_i | i \in [\ell_0]\} \) that are output by the \( \ell_0 \) parallel executions of NDKG. Recall that each parallel execution outputs \( 2\ell \) nonces that we interpret as \( \ell \) nonce pairs. In order to do so, we employ the oracle-aided simulators \( \text{Sim}_0 \) for IDKG and \( \text{Sim}_j, \text{Sim}_j' \) for the \( j \)-th parallel execution of NDKG\(_j\) of NDKG. Corruption queries \( i \in \mathcal{H} \) are handled by \( \text{Sim}_0 \) to return its secret key share \( sk_i \) (along with other internal data generated from IDKG related to \( P_i \)) and by \( \text{Sim}_j, \text{Sim}_j' \) to return its \( 2\ell \) secret nonce shares from NDKG\(_j\) (along with other internal data generated from NDKG\(_j\) related to \( P_i \)). Signature share queries \((i,j,m)\) for an honest party \( P_i \) and (previously generated) nonce pair \((R_i, R'_i)\) are handled in one of two ways. (i) If the reduction already knows \( t_r + 1 \) signature shares for \((j,m)\), then it computes the remaining shares by Lagrange interpolation and returns the signature share \( \sigma_{j,i} \) of that party. (ii) If the reduction knows \( t_r \) or less signature shares for \((j,m)\), then it calls the discrete logarithm oracle \( DL_{\ell, m} \) on input \( pk^{r_j} \cdot R_{j,i} \) to obtain \( \sigma_{j,i} := c_j \cdot sk_i + r_{j,i} \) and returns it. Here, it can derive the values \( c_j := H(pk, R_{j,i}, m) \), \( R_{j,i} := R_jR'_j \), and \( b_j := H_{\text{non}}(pk, R_j, R'_j, m) \) by itself from local computations and consistent lazy sampling for random oracle outputs (if not yet defined).

However, this approach has the subtlety that it exceeds the number of allowed calls to \( DL_{\ell, m} \). If the adversary \( A \) makes a signature share query \((i,j,m)\) and later in the course of the protocol execution corrupts that same party \( P_i \), then the reduction would have used \( DL_{\ell, m} \) too often: once for \( \text{Sim}_0 \) to return the secret key share \( sk_i \), once for \( \text{Sim}_j \) to return the secret nonce share \( r_{j,i} \), once for \( \text{Sim}_j' \) to return the secret nonce share \( r_{j,i}' \), and once to compute the signature share \( \sigma_{j,i} \). On the other hand, the identity \( \sigma_{j,i} = c_j \cdot sk_i + r_{j,i} \) tells us that these calls are enough to derive those four values. To make use of this, we carefully leverage the “queries upon corruption” property of the simulators \( \text{Sim}_j, \text{Sim}_j' \) for \( j \geq 1 \). More precisely, as we know that \( \text{Sim}_j \) queries the discrete logarithm oracle on the element \( R_{j,i} \) upon corruption of party \( P_i \), we simply answer this query on \( R_{j,i} \) by computing \( r_{j,i} := \sigma_{j,i} - c_j \cdot sk_i \) and returning the value \( r_{j,i} = r_{j,i} - b_jr_{j,i}' \) directly instead of calling \( DL_{\ell, m} \). In particular, we avoid redundant calls to \( DL_{\ell, m} \). At the end of the game, we obtain a forgery \((m', \sigma^*)\) from \( A \) which we convert into a solution of the OMDL challenge \( \xi \); recall that \( \sigma^* \) is of the form \((R', s^*)\). This is done as follows. First, as \( A \) is an algebraic

---

\(^*\)Recall that in our model, a message \( m \in \{0,1\}^* \) is always signed with respect to a previously generated and agreed-upon nonce pair \((R_j, R'_j)\). That is, when message \( m \) is signed, the parties have agreement on which nonce index \( j \) to use for it.
adversary, it returns the random oracle query $H(pk, R^*, m^*)$ together with a representation of elements in $\mathbb{Z}_p$. Second, using the forgery $(m^*, \sigma^*)$, known signature shares $\{\sigma_{i,1}, \ldots, \sigma_{i,n}\}_j$, and known secret key shares $sk_i$, from $t_o$ parties, we can compute the secret key $sk$. Third, this allows us to compute all secret key shares $sk_1, \ldots, sk_n$ and thus using the signature shares also all secret nonce shares $\{r_{j,1}, \ldots, r_{j,n}\}_j$. Finally, by inverting the simulatability matrices of all oracle-aided simulators $Sim_0, Sim_1, \ldots$, we can translate the aforementioned values into an OMDL solution. We provide a full proof of the following theorem in Supplementary Material Section C.1.

**Theorem 4.1 (ADKG $\rightarrow$ Threshold Schnorr).** Let $\ell, t_c, t_r, n \in \mathbb{N}$ be natural numbers such that $t_c < n/3$ and $t_r \in [t_c, n - t_c]$. Let $IDKG$ be an algebraic oracle-aided secure $(t_c, t_r, n)$-threshold ADKG protocol and let $NDKG$ be an algebraic oracle-aided secure $(t_r, t_r, n)$-packed ADKG protocol. Further, let $H, H_{non} \colon \{0,1\}^* \rightarrow \mathbb{Z}_p$ be two random oracles. Then, the threshold Schnorr signature scheme $SchnorrTS[IDKG, NDKG]$ (cf. Section 4.1) is UF-CMA secure in the algebraic group model under the OMDL assumption.

5 Efficient Packed ADKG Protocol

In this section, we provide a generic construction for a packed ADKG protocol that is more efficient than naïvely executing many instances of a regular ADKG protocol in parallel.

5.1 Our Construction

In the following, we give a construction for an $(\ell, t_c, t_r, n)$-packed ADKG protocol over $(G, p, g)$ where $\ell = n - 2t_c$ and $t_r \in [t_c, n - t_c]$ is arbitrary. We start by describing the protocol informally.

**Packed ADKG Description.** A formal description of the protocol PADKG is given in Algorithm 1. Conceptually, parties agree on $n - t_c$ AVSS sharings and use a superinvertible (SI) matrix to extract as much randomness from these as is possible. In more detail, our protocol has the following four steps:

1. **Sharing.** In the first step, each party $P_i$ shares a secret $s_i \leftarrow \mathbb{Z}_p$ via a high-threshold AVSS. That means that $s_i$ lies on a polynomial $f_i \in \mathbb{Z}_p[X]$ of degree $t_r$. Afterwards, each party waits for $n - t_c$ AVSS sharings to complete locally and stores the corresponding indices of the dealers in a set $\text{dealers}_i$. Since the network is asynchronous, each party might have a different set $\text{dealers}_i$ of locally completed sharings. Therefore, parties need to agree on exactly one such set using an MVBA protocol $\text{MVBA}$. However, the problem is that these sets as are cannot be checked via an external validity function which is needed for the MVBA protocol. This issue is resolved as follows.

2. **MVBA Execution.** Once its set $\text{dealers}_i$ of completed sharings reaches size $n - t_c$, party $P_i$ sends it as a proposal $\text{prop}_i$ to all other parties with the aim to collect at least $t_c + 1$ signatures from other parties on it that it stores in a set $\text{signs}_i$. Conversely, a party $P_j$ only issues a signature on $\text{prop}_i$ once all AVSS sharings specified by $\text{prop}_i$ have completed at $P_j$ itself. Once the set $\text{signs}_i$ of collected signatures on $\text{prop}_i$ reaches size $t_c + 1$ (guaranteeing that these sharings completed at an honest party and thus by completeness of the AVSS eventually also at all other honest parties), party $P_i$ invokes $\text{MVBA}$ on input $(\text{prop}_i, \text{signs}_i)$ with external validity function $\text{checkValidity}$ given by:

$$\left( |\text{prop}_i| = n - t_c \right) \land \left( |\text{signs}_i| \geq t_c + 1 \right) \land \left( \forall (j, \sigma_j) \in \text{signs}_i : \text{Ver}(vk_j, \text{prop}_i, \sigma_j) = 1 \right).$$

3. **AVSS Reconstruction.** Once the MVBA terminates and parties have agreement on a set $\text{dealers}$ of $n - t_c$ dealers whose sharings completed, parties proceed with the (possibly interactive) reconstruction phase. The result is that each party $P_i$ obtains a vector $(S_{i,1}, S_{i,2}, \ldots, S_{i,n})$ of group elements in $G$ such that $S_{j,k} = g^{f_j(k)}$ for $k \in [n]$ along with its secret share $s_{j,i} = f_j(i)$ for all $j \in \text{dealers}$. While this phase comes for free for *fully committing* schemes like Feldman’s VSS, it is required for other schemes, e.g., those that build upon KZG commitments. We note that this phase comes for free with our AVSS scheme.

---

*That is, all parties behave algebraically and can be modeled as algebraic machines.*
4. **SI Matrix Application.** Having done this, each party $P_i$ locally applies (i.e., matrix multiplication from the left) the $(\ell, n - t_c)$-dimensional superinvertible matrix $\text{SI}$ to its secret shares arranged in a vector $(s_{j,i})_{j \in \text{dealers}}$ to obtain an $\ell$-dimensional vector $(r_{1,i}, \ldots, r_{\ell,i})$ of new private outputs. Additionally, $P_i$ applies $\text{SI}$ in the exponent to the matrix with rows $(S_j, S_{j,1}, \ldots, S_{j,n})$ for $j \in \text{dealers}$ to obtain an $n$-dimensional vector $(R_{1,i}, R_{1,j,1}, \ldots, R_{1,j,n})$ of new public outputs for each $j \in [\ell]$. Looking ahead, these vectors constitute the public nonces and public nonce shares. These operations are captured by the algorithm $\text{ApplySI}$. At the end of this phase, each party $P_i$ outputs a set $\{(j, r_{j,i}, (R_{j,1}, R_{j,1}, \ldots, R_{j,n}))\}_{j \in [\ell]}$. For each $j \in [\ell]$, $R_j$ is the $j$-th public nonce with corresponding public sharings $(R_{j,1}, \ldots, R_{j,n})$ of all parties and $r_{j,i}$ is party $P_i$'s secret share of the nonce $R_j$.

The idea of the final phase is the following. Only $\ell = n - 2t_c$ of the polynomials $f_j$ shared by the parties $j \in \text{dealers}$ are guaranteed to be chosen from honest parties and thus uniformly random. By applying the SI matrix also to public output related to $f_j$ (i.e., the public elements $S_j, S_{j,1}, \ldots, S_{j,n}$ for all $j \in \text{dealers}$), parties obtain regular Feldman commitments to the polynomials $r_1, \ldots, r_\ell$ (i.e., the public elements $R_{j,1}, R_{j,1}, \ldots, R_{j,n}$ for each polynomial $r_j$ where $j \in [\ell]$) which makes subsequent threshold signing for Schnorr signatures possible.

**Algorithm 1 PADKG from the view of $P_i$**

1. initialize $\text{prop}_i, \text{dealers}_i, \text{sigs}_i, \text{sharings}_i, \text{nonces}_i := \emptyset$
   
   // Share a random secret via AVSS, and wait to complete $n - t_c$ sharings
2. sample $s_i \leftarrow \mathbb{Z}_p$ uniformly at random
3. call $\text{Share}$, sharing the secret $s_i$
4. **upon** completing AVSS invocation with $P_j$ as dealer, do
5. $\text{dealers}_i := \text{dealers}_i \cup \{j\}$
6. if $|\text{dealers}_i| = n - t_c$ then
7. $\text{prop}_i := \text{dealers}_i$
8. send ("proposal", $\text{prop}_i$) to all parties
   
   // Gather proof that share completed, and agree on a set of dealers
9. **upon** receiving the first ("proposal", $\text{prop}_j$) message from party $P_j$, do
10. **upon** $\text{prop}_j \subseteq \text{dealers}_i$, do // Wait to complete $\text{Share}$ for dealers in $\text{prop}_j$
11. send ("signature", $\text{Sig}(s_{i,\text{prop}_j})$) to party $P_j$
12. **upon** receiving ("signature", $\sigma_j$) from $P_j$, do
13. if $\text{prop}_j \neq \emptyset$ and Ver$(vk_j, \text{prop}_j, \sigma_j) = 1$ then
14. $\text{sigs}_i := \text{sigs}_i \cup \{(j, \sigma_j)\}$
15. if $|\text{sigs}_i| = t_c + 1$ then
16. call MVBA on input (prop, sigs) and external validity function checkValidity // Agree on $n - t_c$ dealers with at least one honest approval
   
   // Reconstruct shared values
17. **upon** MVBA terminating with output (prop, sigs), do
18. **upon** prop $\subseteq$ dealers, do // Call $\text{Rec}$ for the AVSS instance with $P_j$ as dealer for all $j \in$ prop
19. call $\text{Rec}$ for the AVSS instance with $P_j$ as dealer for all $j \in$ prop
   
   // Save reconstructed sharings and apply SI matrix
20. **upon** Rec terminating for dealer $P_j$ with output $(S_j, S_{j,1}, \ldots, S_{j,n})$ and $s_{j,i}$, do
21. $\text{sharings}_i := \text{sharings}_i \cup \{(j, s_{j,i}, (S_j, S_{j,1}, \ldots, S_{j,n}))\}$
22. if $|\text{sharings}_i| = n - t_c$ then
23. $\text{nonces}_i := \text{ApplySI}(\text{sharings}_i)$
24. output nonces, and terminate
5.2 Security Analysis

We proceed with the security analysis of our generic packed ADKG protocol PADKG, described before (cf. Algorithm 1). For this, we first introduce a security notion for AVSS schemes that is very similar to the oracle-aided simulatability notion for packed ADKG (cf. Definition 3.1). Then, we show that this notion for the AVSS in combination with the (regular) security of the MVBA are sufficient to obtain an oracle-aided secure packed ADKG protocol à la Algorithm 1. Here, we emphasize that the proof crucially relies on the defining property of a superinvertible matrix which is that any square submatrix is invertible.

Definition 5.1 (Oracle-aided Simulatability for AVSS). Let \( \Pi = (\text{Share, Rec}) \) be a complete \((t_c, t_r, n)\)-threshold AVSS scheme (cf. Definition 2.4). We say that \( \Pi \) is an oracle-aided secure \((t_c, t_r, n)\)-threshold AVSS scheme if it additionally has oracle-aided simulatability: There exists \( k \in \text{poly}(\lambda) \) with \( k \geq t_c + 1 \) such that for any PPT algorithm \( A \) that corrupts at most \( t_c \) parties, there exists an algebraic PPT simulator \( \text{Sim} \) that on input \( \zeta := (g^{z_1}, \ldots, g^{z_k}) \in \mathbb{G}^k \) makes \( k' = \{k, k - \delta_a\} \) queries to the oracle \( DL_{\mathbb{G}, g} \) (recall that \( \delta_a := t_r + 1 - t_c \)) and such that:

- **Syntax.** \( \text{Sim} \) simulates the role of the honest parties in an execution of \( \Pi \). At the end of the simulation, \( \text{Sim} \) outputs the tuple \((S, S_1, \ldots, S_n)\).

- **Dealer Corruption.** If the dealer remains honest at the end of the simulation, \( \text{Sim} \) makes \( k' = k - \delta_a \) queries to \( DL_{\mathbb{G}, g} \). Otherwise, it makes \( k' = k \) queries.

- **Queries upon Corruption.** Denote by \( \mathcal{C} \subset [n] \) the dynamic set of corrupted parties. Once the first honest party outputs \((S_1, \ldots, S_n)\), the following holds. Upon corruption query \( i \in [n] \setminus \mathcal{C} \), \( \text{Sim} \) invokes \( DL_{\mathbb{G}, g} \) on input \( S_i = g^{f(i)} \) among (possibly) other input elements. Conversely, it does not query \( S_i \) before that corruption.

- **Query Independence.** Let \( \mathcal{C} \) be as before and \( \mathcal{H} := [n] \setminus \mathcal{C} \). Assume that \([\mathcal{C}] = t_c \) after a simulation of \( \Pi \). For all \( i \in [k'] \), denote by \( b_i \in \mathbb{G} \) the \( i \)-th query to \( DL_{\mathbb{G}, g} \) and let \((a_i, a_{i,1}, \ldots, a_{i,k})\) be the corresponding algebraic vector, i.e., \( b_i = g^{a_i} \cdot g^{a_{i,1}} \cdot \cdots \cdot g^{a_{i,k}} \). Further, denote by \((\hat{a}_i, \hat{a}_{i,1}, \ldots, \hat{a}_{i,k})\) the algebraic vector of \( S_i \) for all \( i \in \mathcal{H} \). Then for all \( I \subset \mathcal{H} \) with \( |I| = k - k^{10} \), the following matrix is invertible over \( \mathbb{Z}_p \):

\[
L(I, \mathcal{C}) := \begin{pmatrix}
    a_{1,1} & a_{1,2} & \cdots & a_{1,k} \\
    \vdots & \vdots & & \vdots \\
    a_{k',1} & a_{k',2} & \cdots & a_{k',k} \\
    b_{1,1} & b_{1,2} & \cdots & b_{1,k} \\
    \vdots & \vdots & & \vdots \\
    b_{i_{1},1} & b_{i_{1},2} & \cdots & b_{i_{1},k}
\end{pmatrix} \in \mathbb{Z}_p^{k \times k},
\]

where the indices \( i_{1,j} \) range over the set \( I \). Whenever \( \text{Sim} \) completes a simulation of an execution of \( \Pi \), we call \( L(I, \mathcal{C}) \) the simulatability matrix of \( \text{Sim} \) (for this particular simulation and the set \( I \)). Further, we call \( k \) a simulatability factor of \( \Pi \).

- **Bad Event.** There is an event \( \text{Bad} \), such that for any PPT algorithm \( A \), the probability of \( \text{Bad} \) in an execution of \( \Pi \) with adversary \( A \) is negligible.

- **Indistinguishability.** Denote by \( \text{view}_{A, \Pi} \) the view of \( A \) in an execution of \( \Pi \). Denote by \( \text{view}_{A, \zeta, \text{Sim}} \) the view of \( A \) when interacting with \( \text{Sim} \) on input \( \zeta \). Then, the distributions \((\zeta, \text{view}_{A, \Pi})\) and \((\zeta, \text{view}_{A, \zeta, \text{Sim}})\) where \( \zeta \leftrightarrow \mathbb{G}^k \) and both distributions are conditioned on \( \neg \text{Bad} \) are statistically close.

Proof Intuition. In the following, we describe how to construct an oracle-aided simulator \( \text{Sim} \) for PADKG given oracle-aided simulators for AVSS. For this informal overview, we omit the discussion on correctness, consistency, and termination for PADKG, since these follow from standard considerations. For each \( i \in [n] \), denote by \( \text{Sim}_i \) the simulator for the instance AVSS, with dealer \( P_i \). Assume that AVSS has simulatability factor \( k_i \). Then, our simulator \( \text{Sim} \) has simulatability factor \( kn \). Let \( \xi := (\xi_1, \ldots, \xi_k) \) be elements where \( \xi_i := (\xi_{i,1}, \ldots, \xi_{i,k}) \in G^k \) such that \( \xi_{i,j} = g^{z_{i,j}} \) for some \( z_{i,j} \in \mathbb{Z}_p \). On input \((\text{par}, \xi)\), our simulator \( \text{Sim} \)

\[10\] If the dealer gets corrupted, this set is empty and the matrix is defined accordingly. Otherwise, the set is of size \( \delta_a \).
does the following in an execution of PADKG described by its four phases (cf. Section 5.1). First, in the AVSS sharing phase, it runs Sim$_i$ on input $(par, \xi_i)$ for all $i \in [n]$. Second, in the MVBA execution phase, it runs the protocol faithfully on behalf of all honest parties. Third, the AVSS reconstruction phase is also handled by the simulators Sim$_i$ for $i \in [n]$. Finally, in the SI matrix application phase, Sim applies SI to the reconstructed vectors from the instances AVSS$_i$ for all $i \in dealers$ to obtain $(R_{j,1}, \ldots, R_{j,n})$ for $j \in \ell$.

As a result, Sim can output all necessary group elements in the group $G$ and terminate the protocol.

Throughout the simulation up until the point in which the first honest party outputs the elements $\{(R_{j,1}, \ldots, R_{j,n})\}_{j \in \ell}$, a corruption query $\ell \in \mathcal{H}$ is forwarded to all Sim$_i$, $i \in [n]$, simultaneously. In that case, to answer discrete logarithm oracle queries from Sim$_i$ on an element $h \in G$, Sim forwards it to its oracle DL$_{\ell,g}$ and returns the result to Sim$_i$. However, once the event happens in which the first honest party outputs $\{(R_{j,1}, \ldots, R_{j,n})\}_{j \in \ell}$, subsequent corruption queries have to be answered differently. The subtle reason for this is that Sim otherwise would make redundant calls to its oracle DL$_{\ell,g}$, thus violating the required notion of “query independence.” We see this as follows. Assuming party $P_\ell$ gets corrupted, the notion of oracle-aided simulatability for (packed) ADKG requires the simulator Sim to query DL$_{\ell,g}$ on input $R_{j,\ell}$ for all $j \in \ell$ at this point in time. On the other hand, by the notion of oracle-aided simulatability for AVSS we also know that all simulators Sim$_i$, for $i \in [n]$ will query the discrete logarithm oracle on input $S_{i,\ell}$ for all $i \in [n]$ at this point in time. By definition of $R_{j,\ell}$, we know that for all $j \in \ell$,

$$r_{j,\ell} = m_{j,1}s_{1,\ell} + \ldots + m_{j,n-t_\ell}s_{n-t_\ell,\ell},$$

where the $m_{j,i} \in \mathbb{Z}_p$ are the entries of the superinvertible matrix $SI := (m_{j,i})_{j,i}$ and we assume for simplicity that $dealers = \{1, \ldots, n - t_\ell\}$. Obviously, the required calls from Sim to its oracle DL$_{\ell,g}$ on input $R_{j,\ell}$, $j \in \ell$, cannot be independent from the ones the individual simulators Sim$_i$ would make on input $S_{i,\ell}$, $i \in [n]$. In order to deal with this issue, the simulator Sim has to return the values $s_{i,\ell} = DL_{\ell,g}(S_{i,\ell})$ to Sim$_i$ for all $i \in [n]$ differently. Concretely, it will first query DL$_{\ell,g}(R_{j,\ell})$ for all $j \in \ell$ to obtain the values $r_{j,\ell}$ for $\ell \in \ell$. Next, it will choose a random subset $S \subset dealers \setminus \ell$, of size $t_\ell - |C \cap dealers|$ and query DL$_{\ell,g}(S_{i,\ell})$ for all $i \in S$ to obtain a total of $t_\ell$ values $(s_{i,\ell})_{i \in S \cup (C \cap dealers)}$. From knowledge of these values, the identities in $(\triangledown)$, and the property of SI, Sim can compute the remaining values $s_{i,\ell}$ from $(\triangledown)$ by inverting a suitable submatrix of SI and return these values to the simulators Sim$_i$. Still, special care has to be taken as the set $C$ of corrupt parties is dynamically increasing and we have to make the counting argument of calls to DL$_{\ell,g}$ rigorous. We provide a full proof of the following theorem in Supplementary Material Section C.2.

**Theorem 5.2 (AVSS → ADKG).** Let $t_c, t_r, n \in \mathbb{N}$ be natural numbers such that $t_c < n/3$ and $t_r < [t_r, n - t_c]$. Let AVSS be an oracle-aided secure $(t_c, t_r, n)$-threshold AVSS scheme and let MVBA be a $(t_r, n)$-secure MVBA protocol. Further, let SI be a superinvertible matrix over $\mathbb{Z}_p$ of dimension $(n - 2t_c, n - t_c)$. Then, PADKG (cf. Algorithm 1) is an oracle-aided secure $(\ell, t_c, t_r, n)$-packed ADKG protocol with $\ell = n - 2t_c$.

**Remark 5.3** We note that our proof does not rely on the algebraic group model. However, if the AVSS scheme is algebraic (i.e., all parties behave algebraically) and the adversary is algebraic, then all our reductions are also algebraic.

### 6 High-Threshold AVSS Scheme

In this section, we design a new high-threshold AVSS scheme and show that it satisfies our notion of oracle-aided simulatability for AVSS under adaptive corruptions.

#### 6.1 Our Construction

We construct a simple high-threshold AVSS scheme $HAVSS = (HAVSS.Share, HAVSS.Rec)$, relying on bivariate polynomials and NIZK proofs for inner product relations. We refer to Algorithms 2 to 4 for formal descriptions as pseudocode from the perspective of a party $P_\ell$.

---

11Note that this can only happen under adaptive corruptions.
Building Blocks. For the construction, we assume additional $t_r + 1$ random generators $g_0, g_1, \ldots, g_{t_r} \leftarrow G$. These can for example be derived from a random oracle. We also assume a non-interactive proof system (cf. Definition B.2) $\mathcal{PS}_{\text{open}} = (\mathsf{PProve}_{\text{open}}, \mathsf{PVer}_{\text{open}})$ for the relation

$$\mathcal{R}_{\text{open}} := \left\{ ((g, g_0, \ldots, g_{t_r}, cm_i, \omega, y), C_i) \middle| \ cm_i = \prod_{j=0}^{t_r} g_j^{c_{j,i}} \land C_i(\omega) = y \right\},$$

and a non-interactive proof system $\mathcal{PS}_{\text{exp}} = (\mathsf{PProve}_{\text{exp}}, \mathsf{PVer}_{\text{exp}})$ for the relation

$$\mathcal{R}_{\text{exp}} := \left\{ ((g, g_0, \ldots, g_{t_r}, cm_i, \omega, Y), C_i) \middle| \ cm_i = \prod_{j=0}^{t_r} g_j^{c_{j,i}} \land Y = g^{C_i(\omega)} \right\}.$$

Note that both relations are inner-product relations, as evaluating a polynomial at a known location is an inner product. For simplicity, we write them using the same random oracle $H$, while formally we should understand this as two separate random oracles. Further, we omit the elements $g, g_0, \ldots, g_{t_r}$ from the statements to avoid clutter. They are always clear from the context.

Finally, we make use of two deterministic algorithms Interpolate and ExpInterpolate, which is Lagrange interpolation and Lagrange interpolation in the exponent, respectively. In more detail, these algorithms work as follows:

- **Interpolate**: This algorithm takes as input a set of $t_c + 1$ pairs $\{(x_i, y_i)\}_{i=1}^{t_c+1}$ of field elements where $x_i \neq x_j$ for $i \neq j$. It outputs the unique polynomial $C \in \mathbb{F}_p[X]$ of degree at most $t_c$ such that $C(x_i) = y_i$ for all $i \in [t_c + 1]$. This can be done by computing the coefficients of the polynomial using standard Lagrange interpolation.

- **ExpInterpolate**: This algorithm takes as input a vector of $t_r + 1$ group elements $(S_1, \ldots, S_{t_r+1})$. It outputs a vector of $n + 1$ group elements $T = (T_0, \ldots, T_n)$ where $T_j = \prod_{i \in [t_r+1]} S_i^{L_{i,j}}$ for all $j \in [n]$ and $L_{i,j}$ denotes the $i$-th Lagrange coefficient for the set $\{1, \ldots, t_r+1\}$ at the evaluation point $j$. Concretely, for all polynomials $F \in \mathbb{F}_p[X]$ of degree at most $t_r$, we have $F(j) = \sum_{i=1}^{t_r+1} L_{i,j} F(i)$.

Protocol Description. As said, the formal description of HAVSS from the perspective of a party $P_i$ is given in Algorithms 2 to 4. Conceptually, HAVSS has the following four steps:

1. **Dealer Committing Phase.** The dealer $P_d$ samples a uniform bivariate polynomial $S \in \mathbb{F}_p[X, Y]$ of degree $t_c$ in $X$ and $t_r$ in $Y$ such that $S(0, 0) = s$. It then generates commitments $cm_1, \ldots, cm_{t_r+1}$ to the (univariate) column polynomials $C_i(Y) := S(1, Y), \ldots, C_{t_r+1}(Y) := S(t_r+1, Y)$ of degree $t_c$. Concretely, these commitments are generalized Pedersen commitments and have the following form:

$$cm_i := \prod_{j=0}^{t_r} g_j^{c_{j,i}} \text{ where } C_i(Y) = \sum_{j=0}^{t_r} c_{j,i} Y^j \in \mathbb{F}_p[Y].$$

Additionally, the dealer $P_d$ computes for all $i \in [t_r+1]$ the exponentiated evaluations $S_i := g^{S(i,0)}$ of the polynomial $S(X,0)$ and NIZK proofs $\sigma_{i,\text{exp}}$ for the relation $\mathcal{R}_{\text{exp}}$. Having done this, the dealer reliably broadcasts the message $(CM, row_0)$ where $CM = (cm_1, \ldots, cm_{t_r+1})$ are the commitments and $row_0 := ((S_1, \pi_{1,\text{exp}}), \ldots, (S_{t_r+1}, \pi_{t_r+1,\text{exp}}))$ are the exponentiated evaluations along with the NIZK proofs of correctness. Upon receiving this message, parties can compute commitments $(cm_1, \ldots, cm_n)$ to all column polynomials $C_0(Y), \ldots, C_n(Y)$ and the exponentiated evaluations $S_i$ for all $i \in [n]$ using ExpInterpolate. Here, we rely on the homomorphic properties of the commitments (cf. Remark 6.1).

2. **Dealer Distributing Rows.** The dealer proceeds by sending each party $P_i$ the evaluations $C_1(i), \ldots, C_n(i)$ along the $i$-th row polynomial $S(X,i)$\footnote{Note that the identity $C_j(i) = S(j, i)$ holds by definition of $C_j$.}. The dealer also sends for all $j \in [n]$ proofs $\pi_{j,i}$ for the relation $\mathcal{R}_{\text{open}}$ attesting that the evaluation $C_j(i)$ is correct with respect to the commitment $cm_j$. Upon receiving such a row along with the evaluation proofs from the dealer, each party $P_i$ checks the correctness of the evaluations by verifying the proofs. Only in case all
Remark 6.1. We proceed with the security analysis of our high-threshold AVSS scheme (cf. Section 6.1). In the following, we give an intuition for the proof of oracle-aided simulatibility. We omit the correctness and termination properties, since these follow from standard considerations.

6.2 Security Analysis

We proceed with the security analysis of our high-threshold AVSS scheme HAVSS. In the following, we give an intuition for the proof of oracle-aided simulatibility. We omit the correctness and termination properties, since these follow from standard considerations.
Algorithm 3 HAVSS.Share,

1: $CM := \bot$, $S := \bot$, $C_i := \bot$, $points_{col,i} := \emptyset$
2: if $P_i$ is the dealer with input $r \in \mathbb{Z}_p$ then
3: HAVSS.Deal($r$)

// Check exponentiated openings, then store commitments
4: upon receiving a (“commits”, $CM', row_0$) broadcast from the dealer, do
5: parse $CM' = (cm_1, \ldots, cm_{r+1})$ and $row_0 = ((S_1, \pi_{1exp}), \ldots, (S_{t+1}, \pi_{t+1exp})$
6: if $\forall j \in [t+1] PVer^H((cm_j, 0, S_j), \pi_{jexp}) = 1$ then
7: $CM := \text{ExpInterpolate}(CM')$ \> $CM = (cm_0, cm_1, \ldots, cm_n)$
8: $S := \text{ExpInterpolate}((S_1, \ldots, S_{t+1}))$ \> $S = (S_0, \ldots, S_n)$

// Check all openings, then forward them
9: upon receiving a (“row”, row) message from the dealer, do
10: upon $CM = \bot$, do
11: parse row = $((C_1(i), \pi_{1,i}), \ldots, (C_n(i), \pi_{n,i}))$
12: if $\forall j \in [n] PVer^H((cm_j, C_j(i), \pi_{j,i})) = 1$ then
13: for every party $P_j$ send (“column”, $C_j(i), \pi_{j,i}$) to $P_j$
14: send (“vote”) to every party $P_i$

// Collect forwarded points and interpolate them
15: upon receiving a (“column”, $C_i(j), \pi_{j,i}$) from party $P_j$, do
16: upon $CM = \bot$, do
17: if $PVer^{open}_H((cm_j, C_i(j), \pi_{j,i})) = 1$ then
18: $points_{col,i} := points_{col,i} \cup \{(j, C_i(j))\}$
19: if $|points_{col,i}| = t_c + 1$ then \> Interpolate points after receiving enough
20: $C_i := \text{Interpolate}(points_{col,i})$

// Bracha-style termination gadget
21: upon receiving (“vote”) messages from $n - t_c$ different parties, do
22: send a (“done”) message to all parties
23: upon receiving (“done”) messages from $t_c + 1$ different parties, do
24: send a (“done”) message to all parties
25: upon receiving (“done”) from $n - t_c$ different parties and $S \neq \bot, C_i \neq \bot$, do
26: terminate

Algorithm 4 HAVSS.Rec$_i$

1: output $s_i := C_i(0)$ and $S = (S_0, S_1, \ldots, S_n)$ and terminate
Proof Intuition. The simulator Sim runs on an input of $k := t_r + 1$ group elements $\zeta := (\zeta_1, \ldots, \zeta_k) \in \mathbb{G}^k$. In order to simulate a sharing of a bivariate polynomial $S(X,Y) \in \mathbb{Z}_p[X,Y]$ of degree $t_r$ in $X$ and $t_c$ in $Y$, the simulator Sim embeds the given $t_r + 1$ elements $\zeta_1, \ldots, \zeta_k$ into exponentiated evaluations of the polynomial $S(X,0)$ of degree $t_r$ at the points $\{1, \ldots, t_r + 1\}$. Since $S(X,0)$ is of degree $t_r$, these $t_r + 1$ evaluations determine the remaining evaluations in the exponent (to base $g$). By Lagrange interpolation in the exponent, Sim obtains evaluations of $S(X,0)$ in the exponent at all the points $\{1, \ldots, n\}$. Next, it samples $t_r + 1$ commitments $c_1, \ldots, c_{t_r+1} \leftarrow \mathbb{G}$ to the first $t_r + 1$ column polynomials $C_i(Y) := S(i,Y)$ uniformly at random, and interpolates them in the exponent to obtain the commitments $c_1, \ldots, c_n$ to all column polynomials. From this point on, while simulating we make sure that parties’ messages are consistent with the commitments and with the polynomial $S(X,0)$. This mainly involves sending messages normally while carefully generating a corrupted party $P_i$’s view upon corruption. This is done by calling the discrete logarithm oracle (which is provided to Sim by definition of oracle-aided simulatability) on input element $S_i := g^{S(i,0)}$ to obtain $S(i,0)$, and sampling polynomials for $P_i$ that is consistent with these $S(i,0)$ and with the previously defined polynomials for all other corrupted parties. In this way, the simulator Sim makes at most $t_r = k - \delta_n$ calls to the discrete logarithm oracle which is the correct number of total calls according to our definition of oracle-aided simulatability. All opening proofs, along with the exponentiated opening proofs for the $S_i$ elements can produced by simulating the NIZKs for the relations $R_{\text{open}}$ and $R_{\text{exp}}$. We provide a full proof of the following theorem in Supplementary Material Section C.3.

Theorem 6.2 (AVSS). Let $t_c, t_r, n \in \mathbb{N}$ be natural numbers such that $t_c < n/3$ and $t_r \in [t_c, n - t_c)$. Further, let $PS_{\text{open}}$ and $PS_{\text{exp}}$ be zero-knowledge proofs of knowledge and let the DLOG assumption hold relative to $(\mathbb{G}, p, g)$. Then, assuming secure erasures, HAVSS (cf. Algorithms 2 to 4) is an oracle-aided secure $(t_c, t_r, n)$-threshold AVSS scheme.

Remark 6.3 We note that secure erasures are only used in the protocol to erase the randomness for generating the proofs (cf. Algorithm 2, Line 9). The reason for this is as follows: in our simulation, we simulate the proofs using the zero-knowledge property. Upon an adaptive corruption, we would have to provide the randomness used for generating the proofs if the protocol did not specify erasing it. Hence, we would need a stronger and non-standard kind of zero-knowledge in which one can efficiently generate consistent randomness for a simulated proof when learning the witness.

To illustrate that such an extension of zero-knowledge is natural, consider the Schnorr NIZK proof as an example. Here, we can compute the randomness $r$ from the witness $w$ and a simulated proof $\sigma$ as $r := \sigma - c \cdot w$. For generic NIZK proofs or existing constructions of inner product arguments, however, it is not directly clear to us how to achieve this property.

7 Instantiation and Efficiency

In this section, we instantiate our framework with concrete building blocks to obtain HARTS and evaluate its communication and round complexity.

Instantiation. For an overview of our instantiation, we refer to Figure 1. Concretely, we use an upper-triangular Pascal matrix [GS23] for the superinvertible matrix. Further, we use VABA [AJM$^+$23] for the MVBA protocol. Finally, we use our HAVSS (cf. Section 6) for the AVSS scheme with the following specifications: Bulletproofs [BBB$^+$18] for the inner product arguments, and the protocol from [DXR22] for the reliable broadcast.

Efficiency. We evaluate the communication and round complexity of our threshold Schnorr signature scheme HARTS. In our AVSS scheme HAVSS, the dealer reliably broadcasts $n$ group elements and proofs of correctness. Using the reliable broadcast protocol from [DXR22] and Bulletproofs [BBB$^+$18], this step has a communication complexity of $O(\lambda n^2 \log n)$. Further, the dealer privately sends $n$ field elements and evaluation proofs to each party, who then disperses these values among all parties. This step also has a communication complexity of $O(\lambda n^2 \log n)$. As a result, we see that HAVSS has log-quadratic communication complexity. As each party invokes it once, the overall cost of the AVSS sharing phase is log-cubic. Next, the MVBA protocol VABA [AJM$^+$23] has cubic communication complexity and terminates in expected constant rounds. Obviously, the application of the matrix $S_i$ in the subsequent phase is only local and does not affect the communication and round complexity of the protocol.

22
From this analysis, we see that the protocol PADKG (cf. Algorithm 1) for nonce generation generates $\ell = t_c + 1 \in O(n)$ nonces with a communication complexity of $O(\lambda n^3 \log n)$. Thus, we obtain an amortized communication cost of $O(\lambda n^2 \log n)$ per nonce. Finally, for signature generation, each party sends a threshold Schnorr signature share of size $O(\lambda)$ to all other parties. Since this step has a communication cost of $O(\lambda n^2)$, we find that a total of $O(\lambda n^3 \log n)$ communication is required to generate $O(n)$ signatures in expected constant rounds, as desired. We note that PADKG generates $\ell \in O(n)$ nonces in expected constant rounds, but only a single round is needed after that to sign a message or even a batch of up to $\ell$ messages if required.

**Acknowledgments.** CISPA authors are funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 507237585, and by the European Union, ERC-StG-2023-101116713. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

**References**


Supplementary Material

A Additional Related Work

We discuss further related work on verifiable secret sharing and threshold Schnorr signatures with a focus on robustness, high-threshold, and efficiency.

Verifiable Secret Sharing. Verifiable secret sharing (VSS) is an essential tool in threshold cryptography and there has been done a huge amount of research on VSS for different network models and different security levels (information-theoretic and computational security). Therefore, we only focus on VSS protocols in the asynchronous setting (AVSS). There has been a long line of research on AVSS protocols [AVZ21, BDK13, BCG93, CT05, CR93, KMS20]. The earlier works in the 1990s [BCG93, CR93] provide information-theoretic security, but at the expense of huge communication complexity. The first practical AVSS was given by Cachin et al. [CKLS02] with computational security and achieves (suboptimal) cubic communication complexity. Backes et al. [BDK13] give the first AVSS with (asymptotically optimal) quadratic communication complexity. To achieve this, they use the KZG polynomial commitment scheme [KZG10] which requires trusted setup. All these protocols have low-threshold reconstruction and the computationally secure ones are only proven against a static adversary. Kokoris-Kogias et al. [KMS20] provide the first AVSS that supports high-threshold reconstruction. Their construction has cubic communication complexity. Alhaddad et al. [AVZ21] then gave the first construction of a high-threshold AVSS with optimal quadratic communication complexity. Abraham et al. [AJM23] provide two constructions of AVSS with optimal quadratic communication complexity using KZG commitments: (i) a high-threshold AVSS that shares one secret, and (ii) a low-threshold AVSS that shares \( t+1 \) secrets while supporting individual reconstruction of the low-threshold secrets among other desirable properties. Further, their constructions are proven adaptively secure. Recently, Shoup and Smart [SS23] provide a (generic) low-threshold AVSS that can share \( t+1 \) secrets with quadratic communication complexity for a correct dealer while having cubic communication complexity for a dealer that provably misbehaves. Their AVSS works over any prime-power modulus ring and finite field.

Robustness. Many proposed schemes for threshold Schnorr signatures from the literature [CGRS23, CKM21, CKM23α, KG20, Lin22] fall short in providing the guarantee of signature output delivery even when only a single party misbehaves in a signing session. Despite many of these schemes providing identifiable aborts, i.e., the possibility to identify a signer responsible for failure in a signing session, they require strong synchrony assumptions on the network\(^{13}\) to restore the property of robustness. Concretely, each time a signing session fails, parties replace the identified malicious signers and start a new session with a different set of signers. Despite this requiring strong synchrony assumptions, it can take up to \( t_\epsilon + 1 \) sequential runs of signing sessions to generate a signature successfully, thus incurring a linear blowup of \( t_\epsilon + 1 \) in both the communication and round complexity. Since any of these schemes has at least quadratic communication complexity, this results in potentially cubic communication complexity for a single signature. More critically, however, the synchrony assumption has been criticized for being too strong [BJMS20, PS17]: if an honest party ever experiences even a short outage or some other kind of malfunction, this party is now considered malicious. As a result, these schemes fail to work under more realistic network conditions [CR93, PS17].

High-Threshold. Initiated by the work of Ruffing et al. [RRJ+22], the last two years have seen a lot of attention on robust threshold Schnorr signatures [BHK+23, GS23, RRJ+22, Sho23]. All of them, with the exception of ROAST [RRJ+22], follow the technique introduced by Gennaro et al. [GJKR99] in order to maintain robustness. Concretely, parties run a (packed) \((t_r, n)\)-threshold ADKG protocol with \( t_r = t_\epsilon < n/3 \) to generate the nonces used for Schnorr signatures. As a result, their protocols for threshold Schnorr signatures inherit this property and retain secrecy as long as at most \( t_\epsilon \) signature shares are revealed, despite there being \( n - t_\epsilon > 2n/3 \) honest parties in the system that keep the protocol alive. In modern systems [GKKS+22, YMR+19], however, one often demands security to hold even if up to \( t_r > t_\epsilon \) signature shares are revealed. More precisely, the protocol should guarantee robustness in the presence of up to \( t_\epsilon < n/3 \) malicious parties, while retaining unforgeability for up to \( t_r < n - t_\epsilon \) revealed signature shares. Unfortunately, all previously mentioned schemes for robust threshold Schnorr signatures [BHK+23, GS23, Sho23], with the exception of ROAST [RRJ+22], fall short in this regard

\(^{13}\) In a synchronous network, messages are sent in synchronized rounds and arrive within a given time bound \( \Delta > 0 \) that is known to every party.
and only provide low-threshold security with \( t_r = t_c < n/3 \). The reason for this is that their techniques and building blocks are not applicable to the high-threshold setting. The protocols in [GS23, Sho23], e.g., employ a low-threshold AVSS [SS23] to retain efficiency and there is no obvious way to use this AVSS in the high-threshold setting. Further, these works use online error correction [CP17] which inherently requires \( t_r < n/3 \). Similarly, the protocol in [BHK+23] employs packed secret sharing and certain consensus primitives, both of which require the low-threshold setting. Further, it only allows parties to sign batches of \( O(n) \) messages (and no individual messages).

**Efficiency.** While ROAST [RRJ+22] provides the desirable property of high-threshold reconstruction with \( t_r < n - t_c \), it significantly falls back in terms of efficiency. We briefly elaborate on that. ROAST (which is an acronym for “RObust ASynchronous Threshold signatures”) is a wrapper protocol that transforms a non-robust threshold signature scheme of a specific type (namely: semi-interactive, two-round, concurrently secure, identifiable aborts) into a protocol for robust and asynchronous threshold signatures. Essentially, the way it achieves this is by starting at most \( n - t_c + 1 \) concurrent signing sessions of the underlying threshold signature scheme in such a clever way that guarantees successful termination of at least one of these sessions. By applying the ROAST wrapper to the threshold Schnorr signature scheme FROST [KG20] (which has the required properties), this yields a robust threshold Schnorr signature scheme with \( t_c < n \) and \( t_r < n - t_c \). Without making any further restrictions on the model, such as the existence of a semi-trusted coordinator, the resulting protocol has a total per-signature communication complexity of \( O(\lambda n^3 + n^4) \) and a round complexity of \( O(n) \). In contrast, the robust low-threshold schemes in [BHK+23, GS23, Sho23] have per-signature communication complexity of \( O(\lambda n^2) \) and a round complexity of \( O(1) \). We emphasize that none of these protocols achieves adaptive security.

## B Additional Preliminaries

In this section, we provide additional preliminaries.

**Definition B.1 (NP Relation).** Let \( \mathcal{R} \) be a relation that contains pairs \((x, w)\) of statement and witnesses such that (i) there exists a polynomial \( p \) such that \(|w| \leq p(|x|)\) for all \((x, w) \in \mathcal{R}\), and (ii) the relation can be decided deterministically in polynomial time. In this case, we refer to \( \mathcal{R} \) as an NP relation. We allow \( \mathcal{R} \) to be parameterized implicitly by the security parameter and assume that \(|x| \leq \text{poly}(\lambda)|\) for all \((x, w) \in \mathcal{R}\).

We define non-interactive proof systems for such NP relation.

**Definition B.2 (Non-Interactive Proof System).** Let \( \mathcal{R} \) be an NP relation and \( H \) be a random oracle. A non-interactive proof system for \( \mathcal{R} \) with respect to \( H \) is defined to be a pair \( PS = (PProve, PVer) \) of PPT algorithms with oracle access to \( H \) and the following syntax:

- \( PProve^H(x, w) \rightarrow \pi \) takes as input a statement \( x \) and a witness \( w \), and outputs a proof \( \pi \).
- \( PVer^H(x, \pi) \rightarrow b \) is deterministic, takes as input a statement \( x \) and a proof \( \pi \), and outputs a bit \( b \in \{0, 1\} \).

Further, we require that the following completeness property holds: For any \((x, w) \in \mathcal{R}\), we have

\[
\Pr \left[ PVer^H(x, \pi) = 1 \mid \pi \leftarrow PProve^H(x, w) \right] = 1.
\]

In addition to completeness, we typically require proof systems to be sound (if no stronger notion is required). Namely, it should not be possible to come up with an accepting proof for a false statement.

**Definition B.3 (Soundness).** Let \( \mathcal{R} \) be an NP relation, \( H \) be a random oracle, and \( PS = (PProve, PVer) \) be a non-interactive proof system for \( \mathcal{R} \) with respect to \( H \). We say that \( PS \) is sound, if there is a PPT algorithm \( PEx \) such that for every PPT algorithm \( A \), the following advantage is negligible in \( \lambda \):

\[
\Pr \left[ b = 1 \land \forall w : (x, w) \notin \mathcal{R} \mid \begin{array}{l}
(x, \pi) \leftarrow A^H(1^\lambda), \\
b := PVer^H(x, \pi)
\end{array} \right] = 0.
\]

\(^{14}\)Their notion of robustness is weaker than ours and does not guarantee signature generation for \( t_r \geq n/3 \) (without assuming a trusted dealer).
The non-interactive proof systems that we need should be zero-knowledge, meaning that one can simulate (by programming the random oracle) proofs without using the witness.

**Definition B.4 (Zero-Knowledge).** Let \( \mathcal{R} \) be an NP relation, \( H \) be a random oracle, and \( PS = (PProve, PVer) \) be a non-interactive proof system for \( \mathcal{R} \) with respect to \( H \). We say that \( PS \) is zero-knowledge, if there is a potentially stateful PPT algorithm \( PSim \) such that for every (potentially unbounded) algorithm \( A \) issuing a polynomial number of queries to \( H \), the following advantage is negligible in \( \lambda \):

\[
\Pr[A^{H,O_0}(1^\lambda) = 1] - \Pr[A^{H^{PSim},O_1}(1^\lambda) = 1],
\]

where \( H^{PSim} \) denotes a random oracle simulated by \( PSim \), and \( O_b \) for \( b \in \{0,1\} \) takes as input pairs \((x,w) \in \mathcal{R} \) and outputs \( \pi \leftarrow PProve^b(x, w) \) if \( b = 0 \) and \( \pi \leftarrow PSim(x) \) if \( b = 1 \).

Next, we define what constitutes a proof of knowledge.

**Definition B.5 (Proof of Knowledge).** Let \( \mathcal{R} \) be an NP relation, \( H \) be a random oracle, and \( PS = (PProve, PVer) \) be a non-interactive proof system for \( \mathcal{R} \) with respect to \( H \). We say that \( PS \) is a proof of knowledge, if there is a PPT algorithm \( PExt \) such that for every PPT algorithm \( A \), the following advantage is negligible in \( \lambda \):

\[
\Pr\left[ b = 1 \land (x, w) \notin \mathcal{R} \mid (x, \pi) \leftarrow A^{H(1^\lambda)}, \ b := PVer^H(x, \pi), \ w \leftarrow PExt(x, \pi, Q) \right]\,
\]

where \( Q \) denotes the list of all random oracle queries and the resulting outputs.

Our syntax assumes proof systems based on random oracles and that we can extract just by inspecting the random oracle queries. An instantiation of such an argument system would be the online-extractable system by Fischlin [Fis05]. However, we emphasize that this is just for readability and ease of presentation, especially, we only extract once and do not rely on any witness indistinguishability which is known to cause problems in combination with rewinding [KLX22].

**C Security Proofs**

In this section, we provide the security proofs for the theorems given in the main body of the paper.

**C.1 Proof for Threshold Schnorr**

*Proof of Theorem 4.1.* Let \( A \) be an algebraic adversary against the unforgeability of the threshold Schnorr signature scheme SchnorrTS [IDKG, NDKG] under chosen message attacks (cf. Definition 3.4). We split our proof of the theorem into two parts. In the first part, we provide a simulation of the UF-CMA experiment to \( A \) via a sequence of games. In the second part, we bound \( A \)'s winning probability in the final game by providing an efficient reduction against the OMDL assumption.

Before we start with the proof, we make some simplifications that are without loss of generality. First, we assume that \( A \) makes exactly \( t_o \) corruption queries. Second, we assume that \( A \) does not make signature share queries for corrupt parties. These assumptions are without loss of generality, since one could build a wrapper adversary that internally runs \( A \), but makes enough corruption queries before producing the output forgery and provides signature shares of corrupt parties for \( A \). Clearly, none of these assumptions changes the advantage of \( A \), i.e., the wrapper adversary has the same advantage.

Additionally, the reader may recall that the security game assumes the adversary never issues the same signature share query twice.

**Game Go:** This is the real game. We recall the game to fix notation. The game samples system parameters \( par = (G, p, g) \) where \( G \) is a cyclic group of prime order \( p \) with a generator \( g \). It also initializes a corruption set \( C := \emptyset \) and sets \( H := [n] \setminus C \) throughout the game. Further, it initializes an empty state \( St_i \) for each party \( P_i \), \( i \in [n] \).

By abuse of notation, we will just write \( P_i \in [n] \) instead of \( "P_i, i \in [n]" \). Similarly, we do the same for any index set of parties such as \( C \) or \( H \) instead of \([n]\).
Whenever A decides to corrupt a party $P_i \in \mathcal{H}$, the game returns the internal state $S_{t_i}$ of party $P_i$ to A and sets $\mathcal{C} := \mathcal{C} \cup \{i\}$. Henceforth, A gets full control over $P_i$. Following the setup phase, the game runs IDKG on behalf of the honest parties. Upon termination of IDKG, let $pk$ and $(pk_1,\ldots,pk_n)$ denote the public key and public keys shares determined by IDKG. Moreover, let $sk_i$ for all $i \in \mathcal{H}$ denote the secret key shares of the honest parties. The game updates the state $S_{t_i}$ for all honest parties accordingly. Following the termination of IDKG, the game starts the online phase of the game in which A gets additional access to the random oracles $H, H_{\text{non}}$, to the Nonce-ADKG oracle, and to the signing oracle. As standard, the game provides the random oracles $H, H_{\text{non}}$ by lazy sampling using maps $H[\cdot], H_{\text{non}}[\cdot]$. The game initializes an empty Nonce-ADKG set $\mathcal{R} := \emptyset$ and runs a new parallel execution of NDKG on behalf of the honest parties for each Nonce-ADKG query A makes. Upon termination of the $(k + 1)$-th execution NDKG_{c+1} of NDKG, let $(R_{k+1}, R_{k+1}'), \ldots, (R_{k+t}, R_{k+t}')$ denote the respective public nonce pairs, and let $R_{k+1,j}, \ldots, R_{k+t,j,n}$ (respective $R_{k+1,j+1}^l, \ldots, R_{k+t,j,n}^l$) for each $j \in [t]$ denote the public nonce shares of $R_{k+t,j}$ (respective $R_{k+t,j}^l$). Moreover, let $(r_{k+1,i}, r_{k+t,1,i}, \ldots, r_{k+t,\ell,i})$ for $i \in \mathcal{H}$ denote the respective secret nonce share pairs of the honest party $P_i$. Recall that in our notation each NDKG_{c+1} is a parallel execution of two instances of the Nonce-ADKG protocol NDKG such that the nonces are output in pairs $(R_j, R_j^\ast)$. These nonce pairs are used for signing later to derive the effective nonce $R_j$ upon signing request for message $m$. The game updates the state $S_{t_i}$ for all honest parties accordingly.

Additionally, the game updates $\mathcal{R} := \mathcal{R} \cup \{(k + 1, \{R_{k+1,j}, R_{k+t,j}^l\})\}_{j \in [t]}$. The game initializes an empty signing query set $Q := \emptyset$. Whenever A submits a new query $(i, j, m) \notin Q$, the game first checks if $i \in \mathcal{H}$ and $j \in \pi_i(\mathcal{R})$ where $\pi_i : N \times \mathcal{G}^2 \to N$, $(x, y) \mapsto x$ is the projection onto the first coordinate; this checks if the $j$-th nonce pair is already defined. If one of the checks fails, the game returns ⊥ to A. Otherwise, it further checks if there is an $m' \neq m$ and an $i' \in [n]$ such that $(i', j, m') \in Q$: this checks if A already queried the signing oracle on a different message $m'$ for this particular nonce pair. If this nonce pair was indeed already used for a signature on a different message $m'$, then the game returns ⊥ to A. Otherwise, it updates $Q := Q \cup \{(i, j, m)\}$ and returns $\sigma_{j,i} = \mathsf{SSign}(sk_i, pk_i, r_{j,i}, R_j, R_j^l, m)$ to A. At the end of the game, A outputs a message $m^\ast$ and a signature $\sigma^\ast$. It wins the game if the following conditions are satisfied: $|\mathcal{C}| \leq t_c$, $|\mathcal{C} \cup \mathcal{S}| \leq t_r$, and $\mathsf{Ver}(pk, m^\ast, \sigma^\ast) = 1$ where $\mathcal{S} := \{i \in [n] \mid \exists j \text{ s.t. } (i, j, m^\ast) \in Q\}$ denotes the set of parties for which A already made a signing query for $m^\ast$. Note that we do not require the nonce $R^\ast$ of the forgery $\sigma^\ast = (R^\ast, s^\ast)$ to be among the set of already generated nonces $\mathcal{R}$, in which case $\mathcal{S}$ would be empty.

**Game G_1:** This game is identical to the game before, except that the game aborts when there is a collision $H[pk, \hat{R}_1, m_1] = H[pk, \hat{R}_2, m_2]$ among distinct random oracle queries $(pk, \hat{R}_1, m_1) \neq (pk, \hat{R}_2, m_2)$ from A (for convenience, we omit $pk$ from random oracle queries in our analysis hereafter). By a standard argument, we can bound the probability of this happening by $q_h^2/p$ where $q_h$ denotes an upper bound on the number of random oracle $H$ queries A makes throughout the game. As a reminder, $p$ denotes the order of the underlying system parameters group $\mathcal{G}$. As a consequence, we obtain the bound

$$\Pr [G_0 \Rightarrow 1] \leq \Pr [G_1 \Rightarrow 1] + \frac{q_h^2}{p} \leq \Pr [G_1 \Rightarrow 1] + \negl(\lambda).$$

**Game G_2:** This game is identical to the game before, except that we introduce a coin flip $\theta \leftarrow \{0, 1\}$ at the beginning of the game. In the following, we denote by $q_r$ the number of queries A makes to the Nonce-ADKG. For $i \in [q_r]$, we denote by NDKG$^\circ_i$ the $i$-th execution of NDKG. Let $\sigma^i = (m^i, R^i, s^i)$ be A’s forgery it outputs at the end of the game. The game aborts if (i) $\theta = 0$ and there exists an $\theta \in [q_r]$ and $m_\theta \in \{0, 1\}^*$ for which $R^\ast = R_\theta R_\theta^\circ \theta$ where $b_\theta = H_{\text{non}}(pk, (R_\theta, R_\theta^\circ), m_\theta)$ and $(R_\theta, R_\theta^\circ)$ has been output by NDKG$^\circ_i$, or (ii) $\theta = 1$ and there does not exist such a tuple $(\theta, m_\theta)^{16}$. Essentially, this means that the game correctly guesses whether the forgery is produced from a nonce pair along with a message that was previously output by some NDKG$^\circ_i$ or not. Since the view of A is independent of the choice of $\theta$, we obtain the bound

$$\Pr [G_2 \Rightarrow 1] \geq \frac{1}{2} \Pr [G_1 \Rightarrow 1].$$

**Game G_3:** In this game, we introduce another abort condition. By definition of oracle-aided security, there are events $\text{Bad}_0$ for IDKG and $\text{Bad}_i, \text{Bad}_i^\circ$ for NDKG$^\circ_i, i \in [q_r]$, that each happen with negligible

---

^{16} Hereafter, we will just write $R \leftarrow \text{NDKG}_i$ to mean that $R$ is output by NDKG$^\circ_i$. 

---
probability. We let the game abort if any of these events occur. By a union bound, we obtain
\[
\Pr [G_2 \Rightarrow 1] \leq \Pr [G_3 \Rightarrow 1] + \Pr [\text{Bad}_0] + \sum_{i=1}^{q_r} (\Pr [\text{Bad}_i] + \Pr [\text{Bad}_i'])
\]
\[
\leq \Pr [G_3 \Rightarrow 1] + \text{negl}(\lambda).
\]

More formally, we would build a reduction to bound the probability of event \text{Bad}_i (respectively \text{Bad}_i'). This reduction runs in an execution of a single DKG instance and simulates the game \(G_2\) for the adversary by forwarding between (one of the two instances of) NDKG_i and the game in which it runs. Using the assumption that the DKGs are algebraic, this reduction can easily translate algebraic representations into the required form.

**Game \(G_4\):** In this game, we change the way the signing oracle computes the signature shares. Specifically, on input a tuple \((i, j, m) \not\in \mathcal{Q}\) that is valid (recall that this means \(i \in \mathcal{H}, j \in \pi_1(\mathcal{R})\), and there is no \(m' \neq m\) and \(i' \in [n]\) such that \((i', j, m') \in \mathcal{Q}\), the game does the following. It computes \(g' := \hat{R}_{j,i} \cdot pk_i^{c_i}\) where \(c_i := H(pk, \hat{R}_j, m)\) and \(\hat{R}_j := R_j R_{j,\ell}^{mb} = \text{the effective nonce with } b_j := H_{\text{non}}(pk, \hat{R}_j, R_{j,i}^{mb}, m)\) (the effective nonce shares \(\{\hat{R}_{j,i}\}_{i \in [n]}\) are correspondingly defined) and returns the discrete logarithm value of \(g'\) to base \(g\). Note that the game is no longer efficient now. By definition of the signature share generation algorithm SSgn, this does not change \(A\)'s view and we obtain
\[
\Pr [G_3 \Rightarrow 1] = \Pr [G_4 \Rightarrow 1].
\]

**Game \(G_5\):** In this game, we change the simulation of IDKG. By definition of oracle-aided security, there exists an algebraic simulator \(\text{Sim}_0\) for IDKG with a simulatability factor \(k_0 \geq \ell_r + 1\) that has all the properties specified in Definition 3.1. At the onset, the game samples an element \(\xi(0) := (\xi_1, \ldots, \xi_{k_0}) \leftarrow \mathbb{G}^{k_0}\) uniformly at random. It then simulates IDKG by running \(\text{Sim}_0\) on input \(\xi(0)\). By definition of game \(G_3\) (event \text{Bad}_0 not happening), the view of \(A\) is statistically close to its view in the previous game. As a result, we obtain
\[
\Pr [G_4 \Rightarrow 1] \leq \Pr [G_5 \Rightarrow 1] + \text{negl}(\lambda).
\]

For all \(i \in [q_r]\), we define games \(G_{4+2i}, G_{5+2i}\) in sequence as follows.

**Game \(G_{4+2i}\):** In this game, we change the simulation of the first instance of NDKG_i. Recall that we write NDKG_i to denote a parallel execution of NDKG instances. By definition of oracle-aided security, there exists an algebraic simulator \(\text{Sim}_i\) for NDKG_i with a simulatability factor \(k_i \geq \ell(t_r + 1)\) that has all the properties specified in Definition 3.1. At the onset, the game samples an element \(\xi(i) \leftarrow \mathbb{G}^{k_i}\) uniformly at random. It then simulates the first instance of NDKG_i by running \(\text{Sim}_i\) on input \(\xi(i)\). By definition of game \(G_3\) (event \text{Bad}_i not happening), the view of \(A\) is statistically close to its view in the previous game. As a result, we obtain
\[
\forall i \in [q_r]: \Pr [G_{3+2i} \Rightarrow 1] \leq \Pr [G_{4+2i} \Rightarrow 1] + \text{negl}(\lambda).
\]

**Game \(G_{5+2i}\):** In this game, we change the simulation of the second instance of NDKG_i. Again by definition of oracle-aided security, there exists an algebraic simulator \(\text{Sim}_i'\) for NDKG_i with a simulatability factor \(k_i \geq \ell(t_r + 1)\). At the onset, the game samples an element \(\xi_i' \leftarrow \mathbb{G}^{k_i}\) uniformly at random. It then simulates the second instance of NDKG_i by running \(\text{Sim}_i'\) on input \(\xi_i'\). By definition of game \(G_3\) (event \text{Bad}_i not happening), the view of \(A\) is statistically close to its view in the previous game. As a result, we obtain
\[
\forall i \in [q_r]: \Pr [G_{4+2i} \Rightarrow 1] \leq \Pr [G_{5+2i} \Rightarrow 1] + \text{negl}(\lambda).
\]

Before we proceed, we summarize what we have achieved so far. We have ruled out collisions involving random oracle queries \(H\). We have changed the signing oracle on input \((i, j, m)\) to return the discrete logarithm value of \(\hat{R}_{j,i} \cdot pk_i^{c_i}\) to base \(g\). Further, we have introduced algebraic simulators \(\text{Sim}_0\) for the

\footnote{Hereafter, we will write \(\xi_{\ell}(j)\) to denote the \(j\)-th component of the tuple \(\xi_{\ell}\).}
Recall the evolving sets
In this final game, we change the way the corruption oracle works. This is done as follows.

The second term
This requires some bookkeeping for which we introduce a binary array

For the sake of simplicity, we assume that the adversary \( A \) corrupts \( t_c \) parties \( \mathcal{C} \subseteq [n] \) right before outputting its forgery. The hypothetical reduction would make the following number of calls to the oracle: (i) up to \( n \ell q_r \) calls to simulate signing for all \( n \) parties using the \( \ell q_r \) generated nonce pairs, (ii) \( k_0 - \delta_a \) calls to simulate \( \text{Sim}_0 \) where \( \delta_a = t_r + 1 - t_c \), and (iii) \( 2q_r(k_1 - \ell \delta_a) \) calls to simulate \( \text{Sim}_i, \text{Sim}'_i \) for all \( i \in [q_\ell] \). On the other hand, the degree of the OMDL challenge should be \( k_0 + 2q_r k_1 \) to obtain the elements \( \{\xi(0), \xi(i), \xi'_i\}_{i \in [q_\ell]} \). As a result, the total number of \( \text{DL}_{G, g} \) calls \( R_{\text{hyp}} \) would have to make is given by

\[
n \ell q_r + (k_0 - \delta_a) + 2q_r(k_1 - \ell \delta_a) = (k_0 + 2q_r k_1) \cdot (\ell q_r - 2q_r(\ell \delta_a - \delta_a)) = \ell q_r(n - 2\delta_a) - \delta_a = \ell q_r(4t_c - n) - (n - 2t_c) = \ell q_r(t_c - 1) - (t_c + 1),\]

which is non-negative for \( t_c > 1 \) and large enough \( q_r \), and thus \( R_{\text{hyp}} \) could not solve the given OMDL challenge without exceeding the number of allowed discrete logarithm oracle calls. To resolve this issue, we make two further changes to the game that target the signing and corruption oracles. First, when the game already provided \( t_r + 1 \) signature shares for a particular tuple message-nonce pair \((r, j, m)\), it computes the remaining shares via Lagrange interpolation (as the signature shares lie on a polynomial of degree \( t_r \) by definition). Second, the game exploits the linear equations \( \sigma_{j,i} = c_j \cdot S_{1i} + \tau_{j,i} \) and \( \tau_{j,i} = r_{j,i} + b_j r'_{j,i} \) in order to obtain the secret nonce shares \( r_{j,i}, r'_{j,i} \) upon corruption query for party \( P_i \) (instead of naively forwarding the corruption query to both simulatores \( \text{Sim}_i \) and \( \text{Sim}'_i \) independently). These changes allow us to correctly limit the number of discrete logarithm calls for our reduction later. This requires some bookkeeping for which we introduce a binary array \( S_i := \{0, \ldots, 0\} \subseteq [0, 1]^{q_\ell} \) for each party \( i \in [n] \). Its purpose is to keep track for which nonce \((R_j, R'_j)\) with \( j \in [\ell q_r] \), the game already computed a signature share for party \( P_i \) on that particular nonce. We proceed with the next game description.

**Game \( G_{6+2q_r} \):** Recall the evolving sets \( \mathcal{R} \) and \( \mathcal{Q} \) of public nonces output by \( \text{NDKG}_1, \text{NDKG}_2, \ldots \) and signing queries, respectively. Note that \( |\mathcal{R}| \leq \ell q_r \), since each Nonce-ADKG generates \( \ell \) public nonce pairs at once. At the onset, the game initializes a binary array \( S := \{0, \ldots, 0\} \subseteq [0, 1]^{q_\ell} \) for each \( i \in [n] \). Its purpose is to keep track for which nonce pairs \((R_j, R'_j)\), \( j \in [\ell q_r] \), the game already computed a signature share for \( P_i \) using that particular nonce (and some message bound to it). In this game, we further change the way the signing oracle works. This is done as follows. Whenever \( A \) makes a new signing query \((i, j, m) \notin \mathcal{Q} \), the game first checks if the query is valid. That is, if \( i \in \mathcal{H}, j \in \pi_1(\mathcal{R}) \), and there is no \( m' \neq m \) such that \((., j, m') \in \mathcal{Q} \). If the validity check verifies, the game checks if \( |\mathcal{C}| + \sum_{v \in \mathcal{H}} S_v[j] \geq t_r \). If in that case, the game retrieves \( c_j := H(pk, R_j, m) \), computes the discrete logarithm value of the element \( pk_{c_j} \cdot R_j \), and returns the output denoted by \( \sigma_{j,i} \) to \( A \). Following this, the game updates the binary array for that particular party as \( S_v[j] := 1 \) and also \( \mathcal{Q} := \mathcal{Q} \cup \{(i, j, m)\} \). Note that the game does not make redundant computations of discrete logarithm values, since it by definition only replies to new signing queries that are not already stored in the set \( \mathcal{Q} \). In the other case, i.e., if \( |\mathcal{C}| + \sum_{v \in \mathcal{H}} S_v[j] \geq t_r + 1 \), the game gathers the signature shares \( \{(v, \sigma_{j,v}) \mid v \in \mathcal{H}, S_v[j] = 1\} \cup \{(v, \sigma_{j,v}) \mid v \in \mathcal{C}\} \) that it already provided to \( A \) and computes the share \( \sigma_{j,i} \) by standard Lagrange interpolation. By the correctness of Lagrange interpolation, this does not change \( A \)'s view and we obtain

\[
\forall i \in [q_\ell]: \text{Pr}[G_{5+2q_r} \Rightarrow 1] = \text{Pr}[G_{6+2q_r} \Rightarrow 1].
\]

**Game \( G_{7+2q_r} \):** In this final game, we change the way the corruption oracle works. This is done as follows.

- Whenever \( A \) decides to corrupt a party \( P_i \) with \( i \in [n] \), the game first checks if the query is valid, i.e., if \( i \in \mathcal{H} \). If not, it returns \( \perp \) to \( A \).
• In case the query is valid, the game proceeds as follows. Let \( r \in \{q_r\} \) denote the number of completed and ongoing Nonce-ADKG executions up to this point. As before, the game provides discrete logarithm oracle access for \( \text{Sim}_0 \) by simulating this oracle for itself (directly computing the discrete logarithm value of the queried group element to base \( g \in \mathbb{G} \)). Since \( \text{Sim}_0 \) has to handle internal state data from the initial IDKG phase, this also reveals the secret key share \( \sk_i \) of the corrupted party \( P_i \) to the game.

• For the remaining simulators \( \{\text{Sim}_j,\text{Sim}_j'\}_{j \in [r]} \) (we define this set to be empty if \( r = 0 \)), the game provides discrete logarithm oracle access differently. First, the game scans through the array \( S_i[\cdot] \) and computes for all \( j \in [r] \) such that \( S_i[j] = 1 \) the value \( r_{j,i} := \sigma_{j,i} - \text{H}(pk, \hat{R}_{j,i}, m) \cdot \sk_i \) (note that \( c_j := \text{H}(pk, \hat{R}_{j,i}, m) \) is already defined if \( S_i[j] = 1 \)). By definition of oracle-aided simulatability, we know that \( \text{Sim}_j \) accesses its discrete logarithm oracle on input element \( R_{j,i} \) upon corruption query for \( P_i \). For all those \( j \in [r] \) where \( S_i[j] = 1 \), the game provides this oracle access on \( R_{j,i} \) for \( \text{Sim}_j \) by retrieving both values \( r_{j,i}, r_{j,i}' \), where it knows \( r_{j,i}' \) from the interaction with \( \text{Sim}_j' \), and returns the computed value \( r_{j,i} = \hat{r}_{j,i} - b_j r_{j,i}' \) to \( \text{Sim}_j \). Finally, the game updates the \( i \)-th binary array of the corrupted party \( P_i \) as \( S_i := [1, \ldots, 1] \).

Since the interface provided to the simulators is the same, this change is independent from \( A \)’s view and we therefore obtain

\[
\Pr[G_{7+2q_r} \Rightarrow 1] = \Pr[G_{6+2q_r} \Rightarrow 1].
\]

It remains to bound the probability that the final game \( G_{7+2q_r} \) outputs 1. For that, we build an efficient reduction \( R \) against the OMDL assumption of degree \( k := k_0 + 2q_r k_1 \). The way we have defined the sequence of games, \( R \)’s simulation of \( G_{7+2q_r} \) on input an \( k \)-OMDL instance \( \xi \in \mathbb{G}^k \) is straightforward. We will describe it in the following and then convert the adversary \( A \)’s forgery into a solution of the challenge \( \xi \).

**Building a reduction \( R \).** The reduction \( R \) gets as input \( \xi \in \mathbb{G}^k \) and simulates game \( G_{7+2q_r} \) for \( A \) as follows. First, it parses \( \xi \) as \( (\xi_{(0)}, \xi_{(1)}, \xi_{(2)}, \ldots, \xi_{(q_r)}, \xi_{(q_r)}) \) where \( \xi_{(0)} \in \mathbb{G}^{k_0} \) and \( \xi_{(i)} \in \mathbb{G}^{k_1} \) for all \( i \in [q_r] \). With that, it runs \( \text{Sim}_0 \) on \( \xi_{(0)} \), \( \text{Sim}_i \) on \( \xi_{(i)} \) and \( \text{Sim}_i' \) on \( \xi_{(i)}' \) for all \( i \in [q_r] \). Whenever the game simulates a discrete logarithm oracle for itself, the reduction \( R \) uses its oracle \( \text{DL}_{\mathbb{G},g} \) for this purpose. In the following, we describe the simulation \( R \) provides in more detail.

• **Initial ADKG Protocol.** The reduction \( R \) invokes \( \text{Sim}_0 \) on input \( \xi_{(0)} \) in order to simulate the initial distributed key generation protocol. Whenever \( \text{Sim}_0 \) queries the discrete logarithm oracle, the reduction simply forwards this query to its own oracle \( \text{DL}_{\mathbb{G},g} \). Corruption queries for this phase are completely handled by the simulator \( \text{Sim}_0 \).

• **Nonce-ADKG Protocols.** For all \( i \in [q_r] \), the reduction \( R \) invokes \( \text{Sim}_i \) on input \( \xi_{(i)} \) and likewise \( \text{Sim}_i' \) on input \( \xi_{(i)}' \) to simulate the \( i \)-th (parallel) execution of\( \text{NDKG}_i \). In general, the reduction does not know the entire internal states of the honest parties as it delegated parts of the simulation so far to the oracle-aided simulators.

• **Signing Query.** Upon a new signing query \( (i,j,m) \notin \mathcal{Q} \), the reduction checks if the query is valid. In that case, it checks if the sum \(|C| + \sum_{v \in \mathcal{H}} S_v[j]\) exceeds the value \( t_r \) or not. In the first case, \( R \) calls its oracle \( \text{DL}_{\mathbb{G},g} \) on input \( pk^{\xi_{(i)}} \cdot \hat{R}_{j,i} \), returns the output \( \sigma_{j,i} \) to \( A \), and updates the array as \( S_i[j] := 1 \) and \( \mathcal{Q} := \mathcal{Q} \cup \{ (i,j,m) \} \). In the second case, \( R \) gathers already computed signature shares \( \{ (v, \sigma_{j,v}) \} \) (where \( v \in \mathcal{C} \) or \( v \in \mathcal{H} \) such that \( S_v[j] = 1 \)) and computes \( \sigma_{j,i} \) via Lagrange interpolation.

• **Corruption Query.** Upon a new corruption query \( i \in \mathcal{H} \), the reduction does the following. Again, \( r \in \{ q_r \} \) denotes the number of completed \( \text{NDKG} \) executions. Corruption queries are handled by the simulators \( \{\text{Sim}_0,\text{Sim}_i,\text{Sim}_i'\}_{i \in [r]} \), whereby providing access to a discrete logarithm oracle by its own oracle \( \text{DL}_{\mathbb{G},g} \) except for queries in \( \{ R_{j,i} \mid j \in [r], S_i[j] = 1 \} \). For these elements, \( R \) computes \( \tilde{r}_{j,i} := \sigma_{j,i} - c_j \cdot \sk_i \) and returns \( r_{j,i} := \tilde{r}_{j,i} - b_j r_{j,i}' \). Note that it knows all the values on the right-hand side of the equations. Finally, it updates \( \mathcal{C}' := \mathcal{C} \cup \{i\} \) and the \( i \)-th binary array as \( S_i := [1, \ldots, 1] \).
It is clear that the reduction perfectly simulates the game for the adversary, and that its running time is dominated by the running time of the adversary.

**Counting calls to DL_{g, g}**. Before we proceed with the forgery \((m^*, R^*, s^*)\) output by the adversary, we count the number of queries the reduction makes to its discrete logarithm oracle. We need to show that it does not exceed the number of allowed queries for the given challenge \(\xi\). To this end, we assume without loss of generality that (i) \(A\) makes exactly \(t_c\) corruption queries \(C \subseteq [n]\), (ii) for all non-forgery indices \(j \in \pi_1(\mathcal{R}) \setminus \{\vartheta\}\), \(A\) makes at least \(t_r + 1 - |\mathcal{C}|\) signature share queries \((i, j, m_j)\) for that nonce index where \(i \in \mathcal{H} = [n] \setminus C^{18}\), and (iii) if \(\vartheta \neq 1\), \(A\) makes exactly \(t_r\) signature share queries for \(R^*\). Again, this can trivially be enforced by building a wrapper adversary that internally runs \(A\), but makes enough corruption and signing queries before outputting its forgery.

Recall that the given OMDL challenge \(\xi\) is of degree \(k = k_0 + 2q_kk_1\) and thus at most \(k - 1\) calls to the oracle are allowed. First, we observe that the simulator \(\text{Sim}_0\) for the initial ADKG protocol IDKG gets \(k_0\) elements as input, while the simulators \(\text{Sim}_i, \text{Sim}_i'\) for all \(i \in [q]\) for the Nonce-ADKG protocols NDKG, each gets \(k_1\) elements as input. The way \(R\) is built, these input elements \(\{\xi(i_0), \xi(i_0'), \xi(i')\} \in [\eta, \eta]\) are chosen disjointly from the OMDL challenge \(\xi\) of degree \(k = k_0 + 2q_kk_1\). Second, we observe that the reduction delegates corruption queries to the simulators \(\text{Sim}_j, \text{Sim}_j'\) for all \(i \in [q]\) that each handles the internal state data from NDKG, (by abuse of notation, we interpret NDKG as IDKG). By definition of oracle-aided simulatability, each simulator \(\text{Sim}_i, \text{Sim}_i'\) for \(i \in [q]\) makes \(k_1 - \ell_{\delta_a}\) calls to the discrete logarithm oracle where \(\delta_a = t_r + 1 - t_c\). On the other hand, the initial simulator \(\text{Sim}_0\) makes \(k_0 - \delta_a\) queries. Third, we observe that the reduction makes at most \(t_r + 1\) calls to its discrete logarithm oracle for each nonce index \(j \in \pi_1(\mathcal{R}) \setminus \{\vartheta\}\) to answer signing queries for \((R_j, R'_j, m)\). Having said that, we obtain the following:

- For each \(j \in [\ell q]\), denote by \(S_j \in [t_r + 1]\) the number of calls the reduction makes to \(\text{DL}_{g, g}\) in order to answer signing queries for \((R_j, R'_j)\). In case a party \(P_i \in \mathcal{H}\) gets corrupted after the signature share \(\sigma_{j,i}\) for that party and nonce index \(j\) was computed, by design the reduction saves one call to its discrete logarithm oracle. The reason for this is that by definition the oracle-aided simulator \(\text{Sim}_j\) queries the discrete logarithm value on input \(R_{j,i}\) upon corruption query for party \(P_i\). In case a party \(P_i \in \mathcal{H}\) gets corrupted before the signature share \(\sigma_{j,i}\) was computed, the reduction does not call its discrete logarithm oracle anymore to answer signing queries for that particular party \(P_i\) and nonce index \(j\). Further, we denote by \(C_j[+1]\) the subset of parties that got corrupted after their signature share \(\sigma_{j,i}\) was computed, the reduction does not call its discrete logarithm oracle anymore to answer signing queries for that particular party \(P_i\) and nonce index \(j\). Further, we denote by \(C_j[-1]\) the subset of parties that got corrupted before their signature share \(\sigma_{j,i}\) was computed and by \(C_j[+1]\) the remaining corrupted parties (i.e., the parties for which the reduction never explicitly computed a signature share for \((R_j, R'_j)\)). Then, we find that \(S_j + |C_j[-1]| = t_r + 1\) by assumption (ii) and \(|C_j[-1] \cup C_j[+1]| = t_c\) by assumption (i). Since the reduction by design saves one call to its discrete logarithm oracle for all parties in \(C_j[-1]\), the difference \(S_j - |C_j[-1]|\) exactly represents the number of calls the reduction would make to \(\text{DL}_{g, g}\) in order to compute the signature shares for all parties and nonce \((R_j, R'_j)\) under the assumption that all \(t_c\) corruptions happened before any signing query for \((R_j, R'_j)\). In combination, the above expression \(S_j - |C_j[+1]|\) is equal to \(S_j - (t_c - |C_j[-1]|) = S_j + C_j[-1] - t_c = t_r + 1 - t_c = \delta_a\).

Interpreted in another way, this value gives the number of additional calls the reduction makes for each nonce \((R_j, R'_j)\) besides the ones for corruptions. This calculation is true for all indices \(j \in [\ell q]\) in case \(\theta = 0\) (i.e., the forgery nonce \(R^*\) is not among previously queried signature shares for some nonce pair \((R_{\vartheta}, R_{\vartheta}')\)). In case \(\theta = 1\), however, this calculation is true for all \(j \neq \vartheta\). For \(j = \vartheta\), this number is equal to \(\delta_a - 1\), since \(A\) makes at most \(t_r\) signing queries for the nonce \(R^*\) (which is derived from \((R_{\vartheta}, R_{\vartheta}')\)) and a message \(m\). By summing up over all indices \(j \in [\ell q]\), we obtain for the total number of discrete logarithm oracle calls of this type \(S := \ell q \delta_a - \vartheta\).

- As a result, the total number of calls the reduction makes to its discrete logarithm oracle, including the queries from the oracle-aided simulators, is given by the following sum:

\[
(k_0 - \delta_a) + 2q_k \cdot (k_1 - \ell_{\delta_a}) + (\ell q, \delta_a - \theta) = k_0 - \delta_a - q_k \ell_{\delta_a} - \theta.
\]

The first summand is the number of calls the initial simulator \(\text{Sim}_0\) makes, the second summand is the number of calls the Nonce-ADKG simulators \(\text{Sim}_i, \text{Sim}_i'\) for \(i \in [q]\) make, and the third

---

18Effectively, this should reveal at least \(t_r + 1\) signature shares for each such index \(j\).
Finally, when $A$ outputs its forgery, we let the reduction additionally call its discrete logarithm oracle $\text{DL}_{G,g}$ on input $pk_i$ for all $i \in H$ such that $S_i[\theta] = 1$ (in the other case where $\theta = 0$, the reduction randomly chooses a set of indices $i \in H$ of size $t_r - |C|$). We denote this set of indices by $I_s$. Since we assume that $A$ makes $t_r$ signature share queries for $R^*$, this number is given by $t_r - |C| = t_r - t_c = \delta_a - 1$. Further, we let the reduction additionally call its discrete logarithm oracle $\text{DL}_{G,g}$ on input $R_{j,i}$ for all $j \in [\ell q_r]$ and $i \in I_s$ (note that these are $\delta_a - 1$ additional calls for each $j$). In case $\theta = 1$, we let the reduction make an additional discrete logarithm oracle call on input $R_\delta$. Adding this up with the computed number in the previous bullet point, we thus obtain

$$(k - \delta_a - q_r)\delta_a - (\delta_a - 1) + q_r \ell \cdot (\delta_a - 1) + \theta = k - 1 - q_r \ell$$

as the total number of calls the reduction has made so far to the discrete logarithm oracle provided by the OMDL challenge of degree $k$.

Now that we have counted queries, let us continue with a look at the bigger picture of the proof. Informally, what we have done so far is the following. During the game, we have obtained the secret key shares \{sk\}_{j \in C} and secret nonce shares \{r_{j,i}, r'_{j,i} | j \in [\ell q_r] \} \subset C$ of the $t_r$ corrupt parties $C \subset [n]$. We have also obtained $t_r + 1$ signature shares \{sk_{i,j} | j \in [\ell q_r], j \neq \theta\} \subset [n]$ (by Lagrange interpolation we obtain all $n$ shares) where $\theta \in [\ell q_r] \cup \{\bot\}$ denotes the index used for the forgery nonce (recall that $\theta = \bot$ in case $\theta = 0$, and $\theta \in [\ell q_r]$ otherwise). Finally, we have obtained the additional secret key shares \{sk_{i,j} | j \in [\ell q_r], j \neq \theta\} \subset [n]$ (thus in total $|C \cup I_s| = t_r$ secret key shares), the secret nonce shares \{r_{j,i} | j \in [\ell q_r], j \neq \theta\} \subset [n]$, and for the case $\theta = 1$ also the secret nonce $r_\delta$. In any case, the reduction can obtain from this data the additional secret nonce shares \{r'_{j,i} | j \in [\ell q_r], j \neq \theta\} \subset [n]$. This can be seen as follows. For $i \in I_s$ and $j \in [\ell q_r] \setminus \{\theta\}$, it can from the knowledge of $sk_i$ and $sk_{j,i}$ derive the value for $r_{j,i}$ and together with $r_{j,i}$ thus obtain $r'_{j,i}$. At this point, we note that the reduction still has $q_r \ell$ discrete logarithm oracle calls at its disposal. Later, the reduction will use these calls to compute all but one value from among the set \{sk_{j,i} | j \in [\ell q_r], j \neq \theta\} \subset [n]$ (via a case distinction on the algebraic equation obtained from the adversary’s random oracle query $H(pk, R^*, m^*)$) and derive the remaining value from the adversary’s forgery signature. As described before, we can then also derive the value $r'_j$. Invertibility of the simulatability matrices of the oracle-aided simulators then allows the reduction to obtain a solution to the OMDL challenge $\xi$. Having conveyed the intuition, we now make the final part of our proof formal.

**Converting forgery to solution of OMDL.** In the final part of our proof, we show how the reduction can efficiently convert the forgery produced by $A$ into a solution of the underlying OMDL challenge $\xi$. We give a brief intuition for this part. First, since $A$ is an algebraic adversary, it outputs the random oracle query $H(pk, R^*, m^*)$ together with an algebraic representation of elements in $\mathbb{Z}_q$. Second, using the forgery $(m^*, \sigma^*)$, the remaining signature shares \{sk_{j,1}, \ldots, sk_{j,n}\}_{j \in \ell q_r}$, and the secret key shares $sk_i$ for $i \in C \cup I_s$, we can solve for the secret key $sk$. Third, this enables us to compute all secret key shares $sk_1, \ldots, sk_n$ and thus by going back to the signature shares also all effective secret nonce shares \{r_{j,1}', \ldots, r_{j,n}'\}_{j \in \ell q_r}$ from which we can then derive the secret nonce shares \{r_{j,1}, r'_{j,1}, \ldots, r_{j,n}, r'_{j,n}\}_{j \in \ell q_r}$. Finally, by inverting the simulatability matrices of the oracle-aided simulators $\text{Sim}_0, \text{Sim}_1, \text{Sim}_1', \ldots, \text{Sim}_q, \text{Sim}'_q$, we can use the aforementioned values to obtain a solution to the OMDL challenge $\xi$. This ends our informal discussion on the proof strategy for converting the forgery into a solution to OMDL. We proceed with the actual analysis now.

Hereafter, let $C \subset [n]$ and $H = [n] \setminus C$ denote the set of corrupt and honest parties, respectively, right before $A$ outputs its forgery $(m^*, R^*, s^*)$. Let $I_s \subset H$ denote the set of parties for which the reduction made a call to its discrete logarithm oracle $\text{DL}_{G,g}$ on input $pk_i$ after obtaining the forgery from $A$. This set was already defined in the preceding paragraph. In the following, let $L = [\ell q_r]$ denote the index of the nonce after whose generation $A$ queried the random oracle on the triple $(pk, R^*, m^*)$. Since $A$ is an algebraic adversary, when it makes the query $H(pk, R^*, m^*)$, it also outputs an algebraic representation

$$a := \{a, a', a_1, \ldots, a_n, \{a_{1,1}, a'_{1,1}, \ldots, a_{1,n}, a'_{1,n}\}, \ldots, \{a_{L,1}, a'_{L,1}, \ldots, a_{L,n}, a'_{L,n}\}\}$$

37
of elements in $\mathbb{Z}_p$, such that

$$R^* = g^\alpha \cdot pk^{a_1} \cdot pk^{a_2^1} \cdot \ldots \cdot pk^{a_n} \cdot \prod_{j=1}^{L} R_{j,1}^{a_{j,1}} R_{j,2}^{a_{j,2}} \cdot \ldots \cdot R_{j,n}^{a_{j,n}} R_{j,n+1}^{a_{j,n+1}}.$$  

Here, the representation is split from left to right into powers of the generator $g$, the public key $pk$, the public key shares $pk_1, \ldots, pk_n$, and the public nonce shares $\{R_{j,1}, R_{j,2}, \ldots, R_{j,n+1}\}_{j \in [L]}$ for $j \in [L]$ returned by the Nonce-ADKG oracle. We omit the combined nonces $\{R_{j,1}, R_{j,2}\}_{j \in [L]}$ here, since these can be written as known linear combinations (Lagrange interpolation in the exponent) of the respective shares $\{R_{j,1,1}, \ldots, R_{j,1,n}\}$ (analogously for $R_{j,2}$). Verification of the forgery $(m^*, R^*, s^*)$ implies $R^* = g^{s^*} \cdot pk^{-c^*}$, where $c^* = H(pk, R^*, m^*)$. Further, recall that $R_{j,1} = g^{c_{j,1}}$ and $R_{j,2} = g^{c_{j,2}}$ for all $j \in [L]$ and $i \in [n]$. Recall by assumption (ii) that the reductions knows the entire set of signature shares $\{\sigma_j, | j \in [L], i \in [n]\}$ and that they satisfy $R_{j,1} = g^{c_{j,1}} \cdot pk^{\sigma_{j,i}}$. Here, we have $c_j := H(pk, R_j, m_j)$ and $R_j := R_j^\theta$, where $b_j := H_{\text{non}}(pk, R_j, R_j^\theta, m_j)$ and $m_j$ is the message bound to the nonce pair $(R_j, R_j^\theta)$. Having said that, the above equation is over $\mathbb{Z}_p$ equivalent to

$$s^* - c^* \cdot sk = \tilde{a} + a' \cdot sk + \sum_{i=1}^{n} a_i \cdot sk_i + \sum_{j=1}^{L} \sum_{i=1}^{n} a_{j,i} r_{j,i} + a'_{j} b_{j}^{-1}(\sigma_{j,i} - c_j \cdot sk_i - r_{j,i}).$$  

By construction, when $A$ outputs its forgery, the reduction calls its discrete logarithm oracle on input $pk_i$ for all $i \in I_*$ and thus knows $sk_i$ for all $i \in C \cup I_*$. Without loss of generality we assume that $C \cup I_* = \{1, \ldots, t_r\}$. Since the secret key shares $sk_1, \ldots, sk_n$ interpolate a polynomial $f \in \mathbb{Z}_p[X]$ of degree $t_r + 1$, there are coefficients $\alpha, \alpha' \in \mathbb{Z}_p$ (depending on the scalars $a_1, \ldots, a_n$ and $sk_1, \ldots, sk_n$) such that $\sum_{i \in [n]} a_i \cdot sk_i = \alpha \cdot sk + \alpha'$. These coefficients $\alpha, \alpha'$ can efficiently be obtained from Lagrange interpolation operations using knowledge of $sk_1, \ldots, sk_n$. Similarly, for every $j \in [L]$ there are coefficients $\alpha_j, \alpha_j' \in \mathbb{Z}_p$ (depending on the scalars $a_{j,1}, \ldots, a_{j,n}$ and $sk_1, \ldots, sk_n$) such that $\sum_{i \in [n]} a_{j,i} \cdot sk_i = \alpha_j \cdot sk + \alpha_j'$. As the known index set for the secret key shares $\{sk_1\}$, and the secret nonce shares $\{r_{j,i}\}_i$ is the same $t_r$-sized set $C \cup I_*$, the interpolation coefficient to obtain $sk_0 = \sum_{i \in [n]} a_{j,i} \cdot r_{j,i}$ is $\alpha_j \cdot r_{j,0} + \beta_j$ for some known scalars $\beta_j \in \mathbb{Z}_p$. In the above equation (.), we put together the known values for $\{s^*, \tilde{a}, \alpha', \sigma_{j,i}, \alpha_j', \beta_j, r_{j,i}, sk_i\}$ (where the set ranges over $j \in [L]$ and $i \in C \cup I_*$) and thus obtain

$$\tilde{s} - c^* \cdot sk = \tilde{a} - sk - \sum_{j \in [L]} c_j \alpha_j b_j^{-1} \cdot sk + \sum_{j \in [L]} (\tilde{a}_j - \alpha_j b_j^{-1}) \cdot r_j,$$  

where $\tilde{s}, \tilde{a},$ and $\tilde{a}_j$ are appropriately defined known values. We make the following observation. As $A$ queries the random oracle on $R^*$ before obtaining the value $c^*$, it fixes the algebraic coefficients $a$ before $c^*$ is chosen (uniformly at random) by the reduction. The same is true for all $(c_j, b_j)$, $j \in [L]$, values that were already fixed at the time $A$ made the random oracle query $H(pk, R^*, m^*)$; we denote the corresponding index set by $L_{\text{pre}} \subseteq L := [L]$. However, we do not apply to the pairs $(c_j, b_j)$ that were only defined after $c^*$ was chosen; we denote the corresponding index set by $L_{\text{post}} \subseteq L$ (note that $[L] = L_{\text{pre}} \cup L_{\text{post}}$). Having said that, we define the following event $\text{ROS}$ by: there exists an $j \in L_{\text{post}}$ such that $\alpha_j \neq 0$. The reduction proceeds as follows with the goal to compute all the secret shares $sk_i, r_{j,i}, r'_{j,i}$ for all $j \in [L]$ and all $i \in [n]$. We make a case distinction with two cases.

Case I: ROS occurs. In this case, there is an $j \in L_{\text{post}}$ such that $\alpha_j \neq 0$. Now we let the reduction call its discrete logarithm oracle $\mathcal{DL}_{g, q}$ on input $R_j$ for all $j \in \{\ell q_l \setminus \{j\}\}$ and additionally on input $pk$. As a result, the reduction obtains the secret key $sk$ and the secret nonces $\{r_{j,i}\}_{i \neq j}$. Therefore, the above equation (.) reduces to $(\tilde{a}_j - \alpha_j b_j^{-1}) \cdot r_j = (\ldots)$ where the right-hand side is some now known value. By assumption of $\tilde{j} \in L_{\text{post}}$, we know that the value for $b_{\tilde{j}}$ was chosen by the reduction uniformly at random and after the values for $\tilde{a}_j, \alpha_j$ were fixed by the adversary (note that these values are derived from the algebraic coefficients $a$ the adversary fixed at the time of its random oracle query $H(pk, R^*, m^*)$, the secret key shares $sk_i$ for $i \in C \cup I_*$, the secret nonce shares $r_{j,i}$ for $j \in [L]$ and $i \in C \cup I_*$, and fixed Lagrange interpolation coefficients). As a result, the expression $\tilde{a}_j - \alpha_j b_j^{-1}$ is only zero with negligible
Case II: ROS does not occur. In this case, for all \( j \in \mathcal{L}_{\text{post}} \) we have \( \alpha_j = 0 \). Then, the above equation (\( \diamond \)) reduces to

\[
\hat{s} + \left( \sum_{j \in \mathcal{L}_{\text{pre}}} c_j \alpha_j b_j^{-1} - \hat{a} - c^* \right) \cdot \mathbf{s}^k = \sum_{j \in \mathcal{L}} \left( \hat{a}_j - \alpha_j b_j^{-1} \right) \cdot r_j. \tag{\( \diamond \)}
\]

Similarly, we observe that the sum \( \sum_{j \in \mathcal{L}_{\text{pre}}} c_j \alpha_j b_j^{-1} - \hat{a} \) on the left-hand side of the above equation (\( \diamond \)) only contains values that were fixed by the adversary before the reduction chose \( c^* \) uniformly at random. As a consequence, the expression \( \sum_{j \in \mathcal{L}_{\text{pre}}} c_j \alpha_j b_j^{-1} - \hat{a} - c^* \) is only zero with negligible probability \( 1/p \).

In particular, we find from equation (\( \diamond \)) that

\[
\mathbf{s}^k = \left( \sum_{j \in \mathcal{L}} \left( \hat{a}_j - \alpha_j b_j^{-1} \right) \cdot r_j - \hat{s} \right) \cdot \left( \sum_{j \in \mathcal{L}_{\text{pre}}} c_j \alpha_j b_j^{-1} - \hat{a} - c^* \right)^{-1}
\]

is well-defined with probability \( 1 - 1/p \). Now, we let the reduction call its discrete logarithm oracle \( \text{DL}_{\mathbb{G}, g} \) on input \( R_i \) for all \( j \in \{q_r\} \) so that it derives \( \mathbf{s}^k \) from the preceding identity. As before, by knowledge of secret key shares \( \{s_k\}_{i \in [n]} \), signature shares \( \{\sigma_{j, i}\} \), and additionally all secret nonce shares \( \{r_{j, i}\}_{j \in \{q_r\}} \), the reduction can efficiently compute all secret nonce shares \( \{r_{j, i, 1}\} \). From this data it can then also compute all the secret nonce shares \( \{r_{j, i, 1, 2}\} \). Next, we consider the case where the event ROS does not occur.

Final Step. What we have achieved so far is that the reduction has obtained from the forgery all the secret shares \( \mathbf{s}^k, r_{j, i, 1}, r_{j, i, 2} \) for all \( j \in \{q_r\} \) and all \( i \in [n] \). From this, we want to derive the solution to the given OMDL challenge \( \xi \) using properties of the oracle-aided simulators \( \{\text{Sim}_0, \text{Sim}_1, \text{Sim}'_0, \text{Sim}'_1\}_{i \in [n]} \). In the following, denote by \( g_1, \ldots, g_{k_0 - \delta_a} \in \mathbb{G} \) the queries the initial simulator \( \text{Sim}_0 \) makes to the discrete logarithm oracle \( \text{DL}_{\mathbb{G}, g} \) and let \( (\hat{a}_i, a_{i, 1}, \ldots, a_{i, k_0}) \) for \( i \in [k_0 - \delta_a] \) be \( g_i \)'s corresponding algebraic vector. Further, for \( i \in [q_r] \) denote by \( g_{i, 1}, \ldots, g_{i, k_1 - \delta_a} \in \mathbb{G} \) the queries the simulator \( \text{Sim}_i \) makes to \( \text{DL}_{\mathbb{G}, g} \) and let \( (\hat{a}_i, a_{i, 1}, \ldots, a_{i, k_1}) \) for \( i \in [q_r] \times [k_1 - \delta_a] \) be \( g_i \)'s corresponding algebraic vector, i.e.,

\[
g_{i, j} = g^{a_{i, j}} \cdot \xi_{i, 1}^{a_{i, 1}} \cdots \xi_{i, k_1}^{a_{i, k_1}}.
\]

For \( i \in [q_r] \), denote by \( y_{i, 1}, \ldots, y_{i, k_1 - \delta_a} \in \mathbb{G} \) the queries the simulator \( \text{Sim}_i \) makes to \( \text{DL}_{\mathbb{G}, g} \) and let \( (\hat{a}_i, a_{i, 1}, \ldots, a_{i, k_1}) \) for \( i \in [q_r] \times [k_1 - \delta_a] \) be \( g_i \)'s corresponding algebraic vector, i.e.,

\[
y_{i, j} = g^{a_{i, j}} \cdot \xi_{i, 1}^{a_{i, 1}} \cdots \xi_{i, k_1}^{a_{i, k_1}}.
\]

Now let \( \mathcal{L}_0 \subset \mathcal{H} \) be any set of size \( \delta_a \), and for \( i \in [q_r] \) let \( \mathcal{I}_i := \mathcal{I}_i^f \subset \mathcal{H}^f \) be any set for which \( |\mathcal{I}_i| = \delta_a \). Further, let \( (\hat{a}_i, a_{i, 1}, \ldots, a_{i, k_0}) \) for \( i \in [k_0 - \delta_a + 1, k_0] \) be the algebraic vectors of the elements in \( \{pk_i\}_{i \in \mathcal{L}_0} \) (in some fixed order), and for \( i \in [q_r] \) let \( (\hat{a}_i, a_{i, 1}, \ldots, a_{i, k_1}) \) with \( i_j := (i, j) \in [q_r] \times [k_1 - \delta_a + 1, k_1] \) be the algebraic vectors of the elements in \( \{R_{i, j}\}_{j \in \mathcal{L}_0} \) (in some fixed order). Further, for \( i \in [q_r] \) let \( (\hat{a}_i, a_{i, 1}, \ldots, a_{i, k_1}) \) with \( i_j := (i, j) \in [q_r] \times [k_1 - \delta_a + 1, k_1] \) be the algebraic vectors of the elements in \( \{R_{i, j}\}_{j \in \mathcal{L}_0} \) (in some fixed order). Essentially, this means that we consider the \( \delta_a \) public key shares \( \{pk_i\}_{i \in \mathcal{L}_0} \) and their algebraic representation, and for each Nonce-ADKG index \( i \in [q_r] \) we consider \( \delta_n \) public nonce shares \( \{R_{i, j}^\ell\}_{j \in \mathcal{L}_0} \) (respectively the public nonce shares \( \{R_{i, j}^\ell\}_{j \in \mathcal{L}_0} \) and their algebraic representation where \( j \) ranges over the set \( \mathcal{I}_i \) (for each of the \( \ell \) public nonces \( \{R_{i, j}^\ell\}_{k \in [\ell]} \)) and their algebraic representation where \( j \) ranges over the set \( \mathcal{I}_i \). Collectively, this gives \( k_0 \) elements with known discrete logarithm values \( \{v_{i, j}\}_{i \in [k_0]} \) associated to \( \text{Sim}_0 \), \( k_1 \) elements with known discrete logarithm values \( \{v_{i, j}\}_{i \in [k_0]} \) associated to \( \text{Sim}_1 \) for all \( i \in [q_r] \), and \( k_1 \) elements with known discrete logarithm values \( \{v_{i, j}\}_{i \in [k_0]} \) associated to \( \text{Sim}'_0 \) for all \( i \in [q_r] \). We denote the set of these elements (with known value) by \( \mathcal{V} := \{v_{i, j, v_{i, j}} \mid l \in [k_0], i \in [q_r], j \in [k_1] \} \).
Analogous considerations for the Nonce-ADKG executions yield the identities

\[ g_i = g^{\hat{a}_i}, \xi_1^{a_i}, \ldots, \xi_{k_0}^{a_i} \Leftrightarrow g^{v_i} \cdot g^{-\hat{a}_i} = \xi_1^{a_i}, \ldots, \xi_{k_0}^{a_i} \]

from the initial ADKG execution, where the \( g_i = g^{v_i} \in G \) are the queries the simulator \( \text{Sim}_0 \) makes to the discrete logarithm oracle. Further, for all \( i \in [k_0 - \delta_{a} + 1, k_0] \) we have the identities

\[ pk_i = g^{\hat{a}_i}, \xi_1^{a_i}, \ldots, \xi_{k_0}^{a_i} \Leftrightarrow g^{v_i} \cdot g^{-\hat{a}_i} = \xi_1^{a_i}, \ldots, \xi_{k_0}^{a_i} \]

From these equations and by definition of the matrix \( L_0 \), the identity holds:

\[ L_0 \cdot \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_{k_0} \end{pmatrix} = \begin{pmatrix} v_1 - \hat{a}_1 \\ v_2 - \hat{a}_2 \\ \vdots \\ v_{k_0} - \hat{a}_{k_0} \end{pmatrix}, \]

where \( z_i \) denotes the discrete logarithm value of the \( i \)-th element \( \xi_i \) of the OMDL challenge \( \xi \in G^k \).

Analogous considerations for the Nonce-ADKG executions yield the identities

\[ L_i \cdot \begin{pmatrix} z_{i,1} \\ z_{i,2} \\ \vdots \\ z_{i,k_1} \end{pmatrix} = \begin{pmatrix} v_{i,1} - \hat{a}_{i,1} \\ v_{i,2} - \hat{a}_{i,2} \\ \vdots \\ v_{i,k_1} - \hat{a}_{i,k_1} \end{pmatrix}, \quad L'_i \cdot \begin{pmatrix} z'_{i,1} \\ z'_{i,2} \\ \vdots \\ z'_{i,k_1} \end{pmatrix} = \begin{pmatrix} v'_{i,1} - \hat{a}'_{i,1} \\ v'_{i,2} - \hat{a}'_{i,2} \\ \vdots \\ v'_{i,k_1} - \hat{a}'_{i,k_1} \end{pmatrix} \]

for all \( i \in [q_r] \). As before, we denote by \( z_{i,j} \) the discrete logarithm value of the OMDL element \( \xi_{i,j} \) and by \( z'_{i,j} \) the discrete logarithm value of \( \xi'_{i,j} \) for all \( i \in [q_r] \) and \( j \in [k_1] \). By definition of oracle-aided simulatability, the matrices \( L_0, L_i \) and \( L'_i \) for \( i \in [q_r] \) are invertible and explicitly known (defined by the algebraic vectors provided by \( \text{Sim}_0, \text{Sim}_i \) and \( \text{Sim}'_i \)). Moreover, the vectors on the right-hand side contain explicitly known values (from knowledge of the set \( V \) and the algebraic vectors provided by the simulators). As a result, by inversion of the simulatability matrices \( L_0, L_i \) and \( L'_i \) for \( i \in [q_r] \), we can efficiently compute \( (z_1, \ldots, z_{k_0 + 2q_rk_1}) \) and thus solve the OMDL challenge \( \xi \) of degree \( k = k_0 + 2q_rk_1 \) (with at most \( k - 1 \) calls to the discrete logarithm oracle). This concludes the proof. \( \square \)

**Remark C.1** We note that the proof above also applies to an algebraic version of the OMDL assumption as defined in [NRS21]. For this, we observe that our security reduction queries the discrete logarithm oracle \( \text{DL}_G,g \) only in the following two cases: (i) when an algebraic simulator \( \text{Sim}_i \) makes a discrete logarithm oracle query, and (ii) when it needs to compute a signature share \( \sigma_{j,i} \), it calls the oracle on input \( \Sigma_{j,i} := pk_{g}^{\sigma_{j,i}} \cdot \hat{R}_{j,i} \). By algebraicity of all simulators \( \{ \text{Sim}_0, \text{Sim}_i, \text{Sim}'_i \}_{i \in q_r} \), these elements are linear combinations of the OMDL elements on which the simulators are run. Since \( \hat{R}_{j,i} \) itself is also an algebraic linear combination of the nonces \( R_{j,i} \) and \( R'_{j,i} \), the reduction can provide (in an efficient manner) an algebraic representation for the elements \( \Sigma_{j,i} \) when calling the oracle \( \text{DL}_G,g \). As a result, the reduction itself calls \( \text{DL}_G,g \) only on algebraic linear combinations of the OMDL challenge it is given.

**C.2 Proof for Packed ADKG**

**Proof of Theorem 5.2.** We show the properties of an oracle-aided secure packed ADKG protocol (cf. Definition 3.1). We begin with the termination property. To this end, we also recall the protocol \( \text{PADKG} \) and go through each of its phases. Hereafter, we use the term *party* \( P_l \) *multcasts a message* \( m \) to mean that \( P_l \) sends the message \( m \) to all parties in the system. Further, we use the term \( (a,b) \)-dimensional matrix to mean a matrix of dimension \( a \times b \) specified over the field \( \mathbb{Z}_p \). For the following discussion, we split the
Termination. In the first phase of the protocol, each party \( P_i \) samples a secret element \( s_i \leftarrow_{\text{R}} \mathbb{Z}_p \) uniformly at random and shares it among all parties via an execution of the high-threshold AVSS scheme that we denote by AVSS, for \( i \in [n] \). Let \( f_i \in \mathbb{Z}_p[X] \) denote the corresponding polynomial of degree \( t_c \) that interpolates the secret \( s_i \). More specifically, parties execute the Share protocol of AVSS = (Share, Rec) at the beginning of this phase. By the correctness property of AVSS we know that if an instance AVSS\(_j\) of the AVSS scheme terminates at an honest party \( P_i \), then it will eventually terminate at all honest parties. Since each honest party executes its AVSS instance correctly, we know that there are at least \( n - t_c \) AVSS instances that will terminate at each honest party. Having said that, each party waits for \( n - t_c \) AVSS instances to terminate at it locally. By the observation made previously, this phase will eventually terminate for all honest parties, as there are at least \( n - t_c \) honest parties and these execute their AVSS instance correctly. For this, each party \( P_i \) locally maintains a set dealers, that is initially empty but throughout keeps track of the dealers whose AVSS instance AVSS\(_i\) terminates at \( P_i \).

Once this set has size \( n - t_c \), the second phase for the parties begins. This is the typical gather proof phase in asynchronous consensus protocols. However, we note here that the set dealers, is still maintained and being filled with completed AVSS instances even in subsequent phases so that it could even potentially reach a size of \( n \) (for instance in case each party behaves honestly and the whole network progresses fast with minimal delays). The goal of the second phase is to extend a message that is a priori not externally valid in such a way that it becomes externally valid. Here, the idea is to collect at least \( t_c + 1 \) signatures from other parties on its set of dealers to ensure there is at least one honest party that approves this set. As a consequence, parties can be sure that this set is a valid set of AVSS instances that will eventually terminate at each honest party. More concretely, each party \( P_i \) sends its set of dealers as a proposal to all parties. For this, it defines the proposal set \( \text{prop}_i := \text{dealers}_i \) and multicasts this set. Upon receiving such a set \( \text{prop}_j \) from another party \( P_j \), the party \( P_i \) waits until all AVSS instances declared in \( \text{prop}_j \) terminate at it locally, i.e., until \( \text{dealers}_i \supseteq \text{prop}_j \) (note that the local set \( \text{dealers}_i \) keeps growing in general). Only after this condition is satisfied, party \( P_i \) knows that each of the instances AVSS\(_j\) for \( j' \in \text{prop}_j \) will also eventually terminate at each other honest party and approves this set by providing a digital signature \( \text{Sig}(\text{sig}_i, \text{prop}_j) \) on it with its signing key \( \text{sig}_i \) and sending this signature back to party \( P_j \). To this end, each party \( P_i \) also maintains a local set of signatures \( \text{signs}_i \) that is initially empty but keeps track of exactly these signatures \( \text{Sig}(\text{sig}_j, \text{prop}_j) \) from other parties \( P_j \) that approve its proposal \( \text{prop}_j \). Since we know that each AVSS instance AVSS\(_j\) with \( j \in \text{prop}_i \) that terminated at \( P_i \), will eventually also terminate at any other honest party and there are at least \( n - t_c \geq t_c + 1 \) honest parties, \( P_i \) will eventually collect at least \( t_c + 1 \) signatures on its proposal and thus terminate the second phase. Once a party collects \( t_c + 1 \) signatures on its proposal (i.e., once \( |\text{signs}_i| = t_c + 1 \)), it progresses to the next phase.

In the third phase of the protocol, parties execute an instance of the MVBA protocol MVBA whose goal is to agree on such a set \( \text{prop} \) of \( n - t_c \) dealers along with the approval \( \text{signs} \) that satisfies the external validity predicate (i.e., the set \( \text{prop} \) of dealers is of size \( n - t_c \) and the approval \( \text{signs} \) consists of at least \( t_c + 1 \) valid signatures on \( \text{prop} \)). From the previous phase we know that each honest party \( P_i \) will invoke MVBA with an externally valid input \( (\text{prop}_i, \text{signs}_i) \). As a consequence, by the termination property of the MVBA protocol, each honest party will terminate MVBA with the same externally valid output \( (\text{prop}, \text{signs}) \) and progress to the next phase.

In the fourth phase of the protocol, parties execute the (exponentiated) reconstruction phase of the AVSS instance AVSS\(_j\) for all \( j \in \text{prop} \) for the agreed upon instances from the MVBA protocol. By the termination property of the MVBA protocol and the AVSS scheme, each honest party \( P_i \) will invoke the reconstruction protocol Rec of AVSS\(_j\) and terminate with the public output \( (S_j, S_{j1}, ..., S_{jn}) \) and private output \( s_{jj} \). In particular, each honest party will progress to the next phase. In the following, define \( \ell := n - 2t_c \) (which is equal to \( t_c + 1 \) in the optimal-resilience case). In the fifth phase of the protocol, each party \( P_i \) locally applies the \((\ell, n - t_c)\)-dimensional superinvertible matrix \( S \) to its \((n - t_c)\)-dimensional vector of private outputs to obtain an \( \ell \)-dimensional vector \( (x_{1j}, ..., x_{\ell j}) \) of new private outputs. Additionally, \( P_i \) locally applies the superinvertible matrix \( S \) in the exponent to the \((n - t_c, n)\)-dimensional matrix consisting of rows \( (S_{1j}, ..., S_{jn}) \) for \( j \in \text{prop} \) to obtain an \( n \)-dimensional vector \( (R_{j1}, ..., R_{jn}) \) of new public outputs for each \( j \in [\ell] \). Since all these operations are done locally and efficiently computable, each honest party will terminate this phase. Additionally, by the correctness
property of the AVSS scheme and the consistency property of the MVBA protocol, the honest parties have agreement on the public output data which is the resulting n-dimensional vectors \((R_{j,1},\ldots,R_{j,n})\) for all \(j \in [\ell]\). Finally, each party terminates the whole protocol with public output \((R_j,R_{j,1},\ldots,R_{j,n})\) for \(j \in [\ell]\) and private output \((x_{1,j},\ldots,x_{t,j})\) where \(R_j\) is also obtained from the \(R_{j,\cdot}\) by Lagrange interpolation in the exponent. Here, the elements \((R_j,R_{j,1},\ldots,R_{j,n})\) are the public nonce shares in the \(j\)-th slot of the packed ADKG protocol with corresponding secret nonce share \(x_{j,1}\) for party \(P_j\). This concludes the discussion on termination of the protocol.

**Correctness and Consistency.** We proceed with the consistency and correctness properties of the protocol. We do not consider these properties separately, since the correctness property will immediately follow from our discussion on consistency. First of all, we show that the public output data \((R_j,R_{j,1},\ldots,R_{j,n})\) for each slot \(j \in [\ell]\) corresponds to a polynomial \(r_j \in \mathbb{Z}_p[X]\) of degree at most \(t_r\). By the correctness property of the AVSS scheme, we know that each instance AVSS, for the agreed upon set of instances \(i \in \text{prop}\) output by MVBA corresponds to a polynomial \(f_i \in \mathbb{Z}_p[X]\) of degree \(t_r\); hereafter, we take w.l.o.g. \(\text{prop} = \{1,\ldots,n-t_c\}\). In particular, this property is preserved after the application of the superinvertible matrix \(\mathbb{S}\). The reason for this is that \(\mathbb{S}\) considered as an \((n-t_c,\ell)\)-dimensional matrix over the polynomial ring \(\mathbb{Z}_p[X]\) (via the natural embedding \(\mathbb{Z}_p \hookrightarrow \mathbb{Z}_p[X]\)) is just a linear transformation that does

\[
(f_1,\ldots,f_{n-t_c}) \mapsto (r_1,\ldots,r_{\ell}), \quad \forall i \in [\ell]: r_i := \lambda_i f_1 + \ldots + \mu_{i,n-t_c} f_{n-t_c},
\]

where \(\mathbb{S} = (\lambda_{i,j})_{i,j}\). As a result, the \(n\)-dimensional vector \((R_{j,1},\ldots,R_{j,n})\) for each \(j \in [\ell]\) corresponds to a polynomial \(r_j \in \mathbb{Z}_p[X]\) of degree at most \(t_r\). Additionally, there is agreement on the elements from the correctness properties of MVBA and AVSS. Additionally, by correctness of the AVSS scheme we know that \(R_{j,\cdot} = g^{s(i,t)}\) for all \((j, i) \in [\ell] \times [n]\). By Lagrange interpolation, a party reconstructs the full vector of elements \((R_j,R_{j,1},\ldots,R_{j,n})\). In particular, each honest party outputs the same public nonce \(R_j\) and the same vector of public nonce shares \((R_{j,1},\ldots,R_{j,n})\) for each \(j \in [\ell]\), and these elements come from polynomials \(r_j \in \mathbb{Z}_p[X]\) of degree at most \(t_r\). This concludes the discussion on correctness and consistency of the protocol.

**Oracle-aided Simulatability.** We proceed with the oracle-aided simulatability property of the protocol. Here, we have to be careful with how we choose the parameter \(\tilde{k}\) for the packed ADKG protocol and how we use the discrete logarithm oracle \(\text{DL}_{G,g}\) in order to not exceed the allowed number of queries to it. However, the basic idea of the simulator \(\text{Sim}\) for the packed ADKG protocol is quite simple. In essence, \(\text{Sim}\) is build upon the individual oracle-aided simulators \(\text{Sim}_i\) for each AVSS instance \(\text{AVSS}_i\) with \(P_i\) as dealer and corruptions are handled by each \(\text{Sim}_i\) individually. By carefully choosing when to query the discrete logarithm oracle and other subtle details, we can design \(\text{Sim}\) in such a way that it satisfies all the properties the oracle-aided simulatability property requires, including the maximal allowed number of queries to the oracle \(\text{DL}_{G,g}\).

In the following, we provide a full and detailed description. Let \(\text{AVSS}\) have oracle-aided simulatability as described in Definition 5.1. In particular, there exists a natural number \(k \in \text{poly}(\lambda)\) with \(k \geq t_r + 1\) and an algebraic simulator that on input \(k\) group elements with oracle access simulates the role of the honest parties in an execution of the AVSS instance and with additional specifications. For each \(i \in [n]\), let \(\text{Sim}_i\) be the algebraic simulator that simulates the instance \(\text{AVSS}_i\) with \(P_i\) as dealer. We build an oracle-aided simulator \(\text{Sim}\) for the packed ADKG protocol denoted by \(\text{PADKG}\) as follows. First, define the simulatability factor of \(\text{PADKG}\) to be \(\tilde{k} := kn \in \text{poly}(\lambda)\). In particular, the condition \(\tilde{k} \geq \ell(t_r + 1)\) is satisfied. Next, let \(\xi := (\xi_1,\ldots,\xi_n) \in G^{kn}\) be a tuple of \(kn\) group elements where \(\xi_i = (g^{z_{i,1}},\ldots,g^{z_{i,n}}) \in G^k\) for each \(i \in [n]\). We choose to write the input element \(\xi\) in this way for reasons that will be clear in a second. The element \(\xi \in G^{kn}\) is the one that \(\text{Sim}\) gets as input. Additionally, \(\text{Sim}\) gets access to a discrete logarithm oracle \(\text{DL}_{G,g}\) in the group \(G\) (to base \(g\)). To have a clear understanding of how the simulator works, we divide the protocol \(\text{PADKG}\) into two conceptual phases. The first phase begins with the invocation of the protocol and ends with the reconstruction of the elements \((S_{j,1},\ldots,S_{j,n})\) for all AVSS instances \(j \in \text{prop}\). The second phase begins with the application of the superinvertible matrix \(\mathbb{S}\) to public and private output data and ends after these local operations with public output data \((R_{j,1},\ldots,R_{j,n})\) for all \(j \in [\ell]\) and private output data.

---

\(^{19}\)In a security proof, these elements along with the discrete logarithm oracle are instantiated with the OMDL assumption of degree \(kn\).
We start with the description of the simulator in the first phase of the protocol. Let $A$ be the PPT adversary that corrupts at most $t_c$ parties. Further, let $C \subseteq [n]$ and $\mathcal{H} := [n] \setminus C$ denote the dynamically evolving sets of corrupted and honest parties, respectively. For each $i \in [n]$, the simulator $\text{Sim}$ invokes $\text{Sim}_i$ on input $\xi_i \in \mathbb{G}^k$ to handle the instance $\text{AVSS}$, with party $P_i$ as dealer. Whenever $A$ decides to corrupt a party $P_j$ with $j \in \mathcal{H}$, $\text{Sim}$ forwards that corruption query to $\text{Sim}_j$ for all $i \in [n]$. To simulate the discrete logarithm oracle $\text{DL}_{\mathcal{G},g}$ for $\text{Sim}_i$ on an element $g' \in \mathcal{G}$ (with algebraic representation), the simulator $\text{Sim}$ simply queries its own discrete logarithm oracle $\text{DL}_{\mathcal{G},g}$ on this element $g'$ and returns the result to $\text{Sim}_i$. The simulator $\text{Sim}$ works that way throughout the first phase. For the execution of the MVBA protocol MVBA, the simulator $\text{Sim}$ behaves on behalf of the honest parties correctly and according to the protocol instructions. Upon termination of the MVBA protocol the set $\text{prop}$ of $n - t_c$ AVSS instances to be used is fixed. Following this, parties start the exponentiated reconstruction phase of these AVSS instances and terminate the first conceptual phase of the protocol with the public elements $(S_{j,1}, \ldots, S_{j,n})$ for all $j \in \text{prop}$ along with some private output data. In the second conceptual phase of the protocol, the simulator $\text{Sim}$ applies the superinvertible matrix $\mathcal{S}$ to the $n$-dimensional public vectors $(S_{j,1}, \ldots, S_{j,n})$ for all $j \in \text{prop}$ to obtain the vectors $(R_{j,1}, \ldots, R_{j,n})$ for all $j \in [\ell]$ and terminate the second conceptual phase. Since all involved algorithms are algebraic and the application of the superinvertible matrix is a linear operation, $\text{Sim}$ can provide its output element with a corresponding algebraic representation. Having done this, $\text{Sim}$ can terminate the protocol simulation.

Throughout the simulation up until the point in which the first corrupt party outputs the elements $(R_{j,1}, \ldots, R_{j,n})$ for all $j \in [\ell]$ (by design of the protocol, we can always assume that all slots $j \in [\ell]$ of the packed ADKG protocol output simultaneously), the simulator $\text{Sim}$ delegates corruption queries entirely to the individual simulators $\text{Sim}_i$ for all $i \in [n]$ without interfering. However, once the aforementioned event (i.e., the first corrupt party outputs $(R_{j,1}, \ldots, R_{j,n})$ for all $j \in [\ell]$) happens and a party $P_i$ with $i \in \mathcal{H}$ gets corrupted, the simulator proceeds as follows: it queries its discrete logarithm oracle $\text{DL}_{\mathcal{G},g}$ on input $R_{j,i}$ for all $j \in [\ell]$ and then forwards this corruption query to all simulators $\text{Sim}_j$ with $i' \in [n]$. Discrete logarithm oracle queries from simulator $\text{Sim}_j$ on any input element $g' \neq S_{i',j}$ are answered as usual (querying its own oracle $\text{DL}_{\mathcal{G},g}$). However, queries on the elements $S_{i',j}$ for all $i' \in [n]$ are answered as follows$^{20}$. By definition of $R_{j,i}$, we know that

$$r_{j,i} = m_{j,1}s_{1,i} + \ldots + m_{j,n-t_c}s_{n-t_c,i} \quad \forall j \in [\ell],$$

where $\mathcal{S} = (m_{j,i})_{j,i}$. Recall that the indices $1, \ldots, n - t_c$ for $s_{(\cdot),i}$ are the labels for parties in dealers (we assumed w.l.o.g. that $\{\text{dealers}\} = \{1, \ldots, n - t_c\}$). Let $C' := C \cap \{\text{dealers}\}$ denote the dynamic set of corrupt parties that are at the same time among the agreed upon set of dealers. At this stage, $\text{Sim}$ already knows the internal data for all parties in $C'$ and in particular $s_{i',j}$ for all $i' \in C'$. Now, $\text{Sim}$ chooses some set $\mathcal{S} \subset (\{\text{dealers}\} \setminus C')$ of size $t_c - |C'|$ uniformly at random and queries its discrete logarithm oracle $\text{DL}_{\mathcal{G},g}$ on input elements $S_{i',j}$ for all $i' \in ([n] \setminus \{\text{dealers}\}) \cup (\mathcal{S} \cup C') := S_d$. In particular, this is the usual behavior as would be done by the simulators $\text{Sim}_j$ for these particular $i' \in S_d$. Note that the set $S_d$ is of size $t_c + t_c = 2t_c$ and thus its complement $S_d' := [n] \setminus S_d$ is of size $\ell = n - 2t_c$. Having done this, $\text{Sim}$ knows the discrete logarithm value $s_{i',j}$ of $S_{i',j}$ for the $n - \ell$ indices $i' \in S_d$. In particular, it knows the values for $i' \in S \cup C'$ which itself is of size $t_c$. By identities (1) and the properties of the superinvertible matrix $\mathcal{S}$, we know that knowledge of the elements $\{r_{j,i}\}_{j \in [\ell]}$ and any $t_c$ elements among the $\{s_{1,i}, \ldots, s_{n-t_c,i}\}$ is sufficient to recover the remaining $n - 2t_c = \ell$ elements $s_{(\cdot),i}$ in this set (since the equations written in matrix form exactly give an $(\ell \times \ell)$-dimensional submatrix of $\mathcal{S}$ which is invertible by definition). By taking the set $\mathcal{S} \cup C'$ of size $\ell$, for which $\text{Sim}$ knows the corresponding $s_{(\cdot),i}$ values, it can compute all values $s_{1,i}, \ldots, s_{n-t_c,i}$ and answer the corresponding discrete logarithm oracle queries from $\text{Sim}_j$ on input $S_{i',j}$ for $i' \in S_d'$ (note that this set is the complement of $\mathcal{S} \cup C'$ in dealers). In total, $\text{Sim}$ made $\ell$ calls to its discrete logarithm oracle for the $R_{j,i}$ with $j \in [\ell]$, but afterwards saved $(n - t_c) - t_c = \ell$ calls for those $S_{i',j}$ with $i' \in S_d'$. Finally, before $\text{Sim}$ terminates, it invokes the simulators $\text{Sim}_j$ for $j \notin $ dealers only in such a way as if $P_j$ was being corrupted.

One should think of the simulators $\text{Sim}_i$ as being responsible for the private data related to the instance $\text{AVSS}_i$, only, and not to $P_i$’s entire internal state from the data of all the different instances $\text{AVSS}_j$ of other parties it obtains. In particular, any party $P_i$’s internal state should be thought of as a disjoint union of data from each of the instances $\text{AVSS}_j$ with $j \in [n]$ that is handled by $\text{Sim}_j$ each.

---

$^{20}$By definition of oracle-aided simulatability we know that the simulators $\text{Sim}_j$, ask the discrete logarithm oracle on these input elements upon corruption of party $P_j$. 43
Having said all that, we begin with the analysis of the simulator Sim in order to show that it is indeed an oracle-aided simulator for PADKG.

Properties. For this, we step-by-step go through each property required for an oracle-aided simulator according to Definition 3.1.

- **Simulatability Factor.** In the beginning of the discussion, we defined the simulatability factor $k := kn$ for Sim. Since $k$ is the simulatability factor of the underlying AVSS, we know that $k \geq t_r + 1$. In particular, we have $kn \geq n(t_r + 1) \geq \ell(t_r + 1)$ since $\ell = n - 2t_c \leq n$ and thus also $k \in \text{poly}(\lambda)$. This shows that our simulatability factor is well-defined.

- **Syntax.** Since the simulators $\text{Sim}_i$ for all $i \in [n]$ have this property, and MVBA is secure, by design the simulator Sim outputs the correct public keys $R_1, \ldots, R_\ell$ and public key shares $(R_{j,1}, \ldots, R_{j,n})$ for all $j \in [\ell]$.

- **Queries upon Corruption.** By design, the simulator Sim forwards the corruption queries directly to the individual simulators $\text{Sim}_i$ for all $i \in [n]$. Since these have the property that they query the discrete logarithm oracle only upon corruption queries and Sim answers their discrete logarithm oracle queries by (i) calling its own oracle $DL_{G,g}$ or (ii) by calling the oracle on $R_{j,i}$ for all $j \in [\ell]$ (once these elements are defined from the protocol execution), it directly follows that Sim has this property. As already mentioned, by design we let Sim call the oracle $DL_{G,g}$ on input $R_{j,i}$ for all $j \in [\ell]$ (once they are defined) upon a corruption query.

- **Bad Event.** We define the bad event $\text{Bad}$ for an execution of PADKG with the simulator Sim as $\text{Bad} := \text{Bad}_1 \lor \ldots \lor \text{Bad}_n$, where $\text{Bad}_i$ is the existing bad event for the execution of instance AVSS, with the simulator $\text{Sim}_i$ for $i \in [n]$. The standard union bound theorem tells us that

$$\Pr[\text{Bad}] \leq \Pr[\text{Bad}_1] + \ldots + \Pr[\text{Bad}_n] \leq n \cdot \text{negl}(\lambda),$$

which itself is negligible.

- **Indistinguishability.** Conditioned on the previously defined event $\text{Bad}$ does not happen, we know that none of the events $\text{Bad}_i$ for $i \in [n]$ happens. In particular, each individual simulator $\text{Sim}_i$ generates a simulation for the adversary $A$ that is statistically indistinguishable from a real execution of AVSS. By a hybrid argument, it follows that the simulation $\text{Sim}$ provides to $A$ is also statistically indistinguishable from a real execution of PADKG, since it simulates the MVBA protocol honestly.

- **Number of $DL_{G,g}$ Queries.** We count the total number of call Sim makes to its discrete logarithm oracle $DL_{G,g}$. For this, we assume without loss of generality that the adversary $A$ corrupts exactly $t_c$ parties during an execution of PADKG. Again, let dealers be the agreed upon set of dealers after termination of the MVBA protocol. By construction, Sim calls its discrete logarithm oracle in one of two ways upon a corruption query on $Pr$. (i) To answer a discrete logarithm query from $\text{Sim}_i$ for $j \in [n]$, and (ii) to compute $r_{j,i}$ for all $j \in [\ell]$ which is the discrete logarithm value of $R_{j,i}$. We have designed Sim in such a way that it answers the discrete logarithm queries from $\text{Sim}_j$ for $j \notin \text{dealers}$ simply by calling its own oracle $DL_{G,g}$ on the queried element, even for $h' = S_{j,i}$ in that case. If the corruption query happened before the public key shares $(R_{j,1}, \ldots, R_{j,n})$ for all $j \in [\ell]$ are defined, the previous sentence is also true for any $j \in [n]$, and not only the once outside the set dealers. However, if the corruption query happened after the public key shares $(R_{j,1}, \ldots, R_{j,n})$ for all $j \in [\ell]$ are defined, then by design Sim calls its oracle $DL_{G,g}$ on input the elements $R_{1,i}, \ldots, R_{t_c,i}$ and picks a set $S \subset \{(\text{dealers}) \setminus C'\}$ of size $t_c - |C'|$. Having done this, it also answers the discrete logarithm queries from $\text{Sim}_j$ for $j \in (S \cup C')$ simply by calling its own oracle $DL_{G,g}$ on the queried element, even for the elements $h' = S_{j,i}$. For the remaining $\ell$ simulators with indices specified by the set $S' \setminus S$, it answers their discrete logarithm queries for all $h' \neq S_{j,i}$ as usual by calling its own oracle on the queried input. The $\ell$ values $s_{j,i}$ for $j \in S' \setminus S$ are computed by knowledge of other values and the property of the superinvertible matrix $S$ as specified before. That way it saved $\ell$ further discrete logarithm oracle calls and we can think of the $DL_{G,g}$ calls for the elements $R_{1,i}, \ldots, R_{t_c,i}$ as being a pulled back oracle call for $S_{j,i}$ with $j \in S' \setminus S$. This observation tells us that we can without loss of generality assume that all corruptions happened before the public key shares are defined from the protocol execution. With this in mind, we can count the number of total discrete logarithm oracle
calls Sim makes by summing up the number of calls from the individual simulators Sim$_{c}$. Since each simulator Sim$_{c}$ is run on input $k$ elements, we know that it makes exactly $k' = k$ calls to the discrete logarithm oracle if party $P_{i}$ gets corrupted and $k' = k - \delta_{a}$ calls otherwise. Assuming that exactly $t_{c}$ parties get corrupted and dealers contains these corrupt parties, we find that

$$2t_{c} \cdot k + (n - 2t_{c}) \cdot (k - \delta_{a}) = 2t_{c}k + nk - n\delta_{a} - 2t_{c}k + 2t_{c}\delta_{a}$$

$$= nk - n\delta_{a} + 2t_{c}\delta_{a}$$

$$= nk - (n - 2t_{c})\delta_{a}$$

$$= nk - \ell\delta_{a},$$

where the first summand comes from the queries for the $t_{c}$ corrupt parties and the queries for the $t_{c}$ parties outside the set dealers, and the second summand comes from the queries for the remaining $(n - 2t_{c})$ parties. This shows that on input $nk$ elements, our simulator Sim makes exactly $nk - \ell\delta_{a}$ queries to the discrete logarithm oracle which is in line with the requirement.

- **Query Independence.** In this final bullet point we show that the discrete logarithm oracle calls the simulator Sim makes are independent in the sense that the algebraic vectors from their representation in $\xi \in G^{kn}$ are independent. For this, we again assume without loss of generality that the adversary A corrupts exactly $t_{c}$ parties during an execution of PADKG. Again, let dealers be the agreed upon set of dealers after termination of the MVBA protocol. Further, we assume that $C = \{1, \ldots, t_{c}\}$ and dealers = $\{1, \ldots, n - t_{c}\}$. By construction, Sim calls its discrete logarithm oracle in one of two ways upon a corruption query on $P_{i}$. (i) To answer a discrete logarithm query from Sim$_{j}$ for $j \in [n]$, and (ii) to compute $r_{j,i}$ for all $j \in [\ell]$ which is the discrete logarithm value of $R_{j,i}$. If the adversary A corrupted all $t_{c}$ parties before the public key shares $(R_{1,i}, \ldots, R_{n,i})$ for all $j \in [\ell]$ are defined from the protocol execution, then Sim does not make any separate calls to any $R_{j,i}$, and it simply answers all discrete logarithm oracle queries from the simulators Sim$_{i}$ for all $i \in [n]$ by calling its own oracle DL$\theta$.g. Now we know that Sim$_{i}$ for all $i \in (C \cup \text{dealers})$ (where we denote $X := [n] \setminus X$ for a set $X \subseteq [n]$ to be the complement of $X$ in $[n]$) makes exactly $k$ discrete logarithm queries and that the corresponding $(k \times k)$-dimensional simulatability matrix $L_{i}(\emptyset, C)$ is invertible over $Z_{p}$. For the remaining $n - 2t_{c} = \ell$ indices, we know that the simulators Sim$_{i}$ each make $k - \delta_{a}$ discrete logarithm queries. Now let $I \subset \mathcal{H}$ be some set of size $\delta_{a}$. Hereafter, for convenience we assume that $t_{c} = n - t_{c} - 1$ and $I = \{t_{c} + 1, \ldots, n - t_{c}\} \subseteq \text{dealers}$. The cases $t_{c} < n - t_{c} - 1$ and $I \not\subseteq \text{dealers}$ work analogously. By definition of the simulatability matrix for Sim$_{i}$ with $i \in I$, we know that the $(k \times k)$-dimensional matrix $L_{i}(I, C)$ is invertible. Recall that the matrix $L_{i}(I, C)$ consists of the algebraic vectors of the discrete logarithm queries Sim$_{i}$ makes (which are in total $k - \delta_{a}$, since $i \notin C$) and the additional algebraic vectors corresponding to the elements $S_{i,j}$ for $j \in I$ (which are $|I| = \delta_{a}$ elements).

Subsequently, we write $L_{i}(0, C)$ to denote the $(k - \delta_{a}, k)$-dimensional matrix resulting from $L_{i}(I, C)$ by deleting the rows corresponding to the elements $S_{i,j}$ for $j \in I$ (i.e., the last $\delta_{a}$ rows). On the other hand, these additional elements (or better, their corresponding algebraic vectors) $S_{i,j}$ for $(i, j) \in I \times I$ (note that $I = n - 2t_{c} = t_{c} - t_{c} + 1 = \delta_{a}$) are not considered in the simulatability matrix for Sim, but instead the elements $R_{j,i}$ with $(j, i)$ ranging over $j \in [\ell]$ and $i \in I$. Since each simulator Sim$_{i}$ takes a disjoint part of the $kn$-wise vector $\xi$ as input, we can arrange the matrices $L_{i}(0, C)$ for $i \in I^{c}$ and $L_{i}(0, C)$ in a block diagonal matrix $L$ where each block has full rank and has size $(k \times k)$ for $i \in I^{c}$ and $(k - \delta_{a}, k)$ for $i \in I$. In order to make the matrix $L$ quadratic, we need to add additional $|I| \cdot \delta_{a} = \ell\delta_{a}$ rows of length $kn$ to it. This is done by the algebraic vectors for the elements $R_{j,i}$ with $(j, i)$ ranging over the set $[\ell] \times I$. By definition of $R_{j,i}$ and its counterpart $R_{j,i}^{\prime}$, we recall that $r_{j,i} = m_{j,1}s_{1,i} + \ldots + m_{j,n-t_{c}}s_{n-t_{c},i}$ for all $j \in [\ell]$, where the $m_{j,i}$ are the coefficients of the superinvertible matrix $S_{i}$. As a result, the algebraic representation of $R_{j,i}$ is a long vector with the first $k$ coordinates being $m_{j,1}$ multiplied by the representation for $S_{1,i}$, the next $k$ coordinates being $m_{j,2}$ multiplied by the representation for $S_{2,i}$, and so on. Adding these long vectors as additional rows into the matrix $L$, we obtain an $(kn, kn)$-dimensional square matrix whose upper part $L_{u}$ consists of blocks and its lower part $L_{l}$ of long vectors as described before. The first block column (by this we mean the first $k$ columns) of the lower part $L_{l}$ are the vectors $\{m_{j,1}\text{rep}(S_{1,i}), \ldots, m_{j,\ell}\text{rep}(S_{1,i})\}_{i \in I}$. The second up to $(n - t_{c})$-th block columns are framed in a similar way with respective factors $m_{j,i}$ and vectors $\text{rep}(S_{j,i})$. Since we want to show that this
This game is the game in the lemma. By definition, we have

This concludes the proof of Theorem 5.2.

C.3 Proof for AVSS

Before we start with a proof of Theorem 6.2, we show that the Pedersen commitment used in our construction of HAVSS (cf. Algorithms 2 to 4) satisfies three properties that we show in Lemmata C.2 to C.4. In Lemma C.2, we want to show the property of interpolation-binding as defined in [AJM+23]. Informally, this notion tells us that if an adversary generates a commitment for a polynomial of degree \( t_c \) and opens \( t_c + 1 \) evaluations of it, then these evaluations fully determine the committed polynomial. More concretely, interpolating these points and computing the commitment to the interpolated polynomial will yield the same commitment.

**Lemma C.2** Consider parameters \((G, p, g)\). Assume that \(PS_{open} \) is a proof of knowledge and that the DLOG assumption (i.e., the one-more discrete logarithm assumption of degree 1) holds relative to \((G, p, g)\).

Further, consider an algorithm \(A\) and the following experiment:

1. Sample \( g_0, \ldots, g_{t_c} \leftarrow G \) and run \( A \) on input \((G, p, g, g_0, \ldots, g_{t_c})\).
2. Obtain from \( A \) a group element \( cm \in G \) and \( t_c + 1 \) triples \(((x_i, y_i, \pi_i))_{i \in [t_c+1]}\) where \( x_i, y_i \in \mathbb{Z}_p\).
3. Output 1 if and only if the following properties hold, otherwise output 0:

   (a) All \( x_i \) are distinct, i.e., for all \( i \neq j \), we have \( x_i \neq x_j \).
   (b) All proofs verify, i.e., \( PVer_{open}^H(cm, x_i, y_i, \pi_i) = 1 \).
   (c) We have \( cm \neq \prod_{j=0}^{t_c} g_j^{a_j} \), where \( A(X) = \sum_{j=0}^{t_c} a_j X^j \in \mathbb{Z}_p[X] \) is the unique polynomial of degree at most \( t_c \) such that \( A(x_i) = y_i \) for all \( i \in [t_c + 1] \).

Then, for every PPT algorithm \( A \), the probability that the game outputs 1 is negligible.

**Proof.** Let \( A \) be a PPT algorithm and let \( \varepsilon \) be the probability that the game in the lemma outputs 1. Our goal is to upper bound \( \varepsilon \). To do so, we provide a sequence of games.

**Game \( G_0 \):** This game is the game in the lemma. By definition, we have

\[
\varepsilon = Pr[G_0 \Rightarrow 1].
\]

**Game \( G_1 \):** This game is as \( G_0 \), but when \( G_0 \) would output 1, the new game \( G_1 \) additionally runs \((b_0, \ldots, b_{t_c}) \leftarrow PExt(cm, x_1, y_1, \pi_1, Q)\). Here, \( PExt \) is the extractor from the proof of knowledge property of \( PS_{open} \) and \( Q \) is the list of random oracle queries with respect to \( H \). That is, the game tries to extract a witness \((b_0, \ldots, b_{t_c}) \in \mathbb{Z}_p^{t_c+1} \) specifying a polynomial \( B(X) = \sum_{j=0}^{t_c} b_j X^j \) from the proof \( \pi_1 \). If either \( cm \neq \prod_{j=0}^{t_c} g_j^{b_j} \) or \( y_1 \neq B(x_1) \), the game aborts and we say the event \( \text{Bad}_1 \) occurs. Otherwise, it outputs 1 as \( G_0 \) does. Clearly, \( G_0 \) and \( G_1 \) differ only if \( \text{Bad}_1 \) occurs. We can easily bound the probability

46
of event $\text{Bad}_1$ using a reduction breaking the proof of knowledge property of $\text{PS}_{\text{open}}$. The reduction forwards random oracle queries and responses between $A$ and the proof of knowledge game and outputs the statement $(\text{cm}, x_1, y_1)$ and the proof $\pi_1$. We get

$$\Pr[\text{G}_0 \Rightarrow 1] \leq \Pr[\text{G}_1 \Rightarrow 1] + \Pr[\text{Bad}_1] \leq \Pr[\text{G}_1 \Rightarrow 1] + \negl(\lambda).$$

**Game $\text{G}_2$:** This game is as $\text{G}_1$, but when $\text{G}_1$ would output 1, the new game $\text{G}_2$ additionally identifies the minimal index $\hat{i} \in [t_c + 1]$ such that $A(x_i) \neq B(x_i)$. We can see that such an index exists as follows: first, since $\prod_{j=0}^{t_c} g_j^{b_j} = \text{cm} \neq \prod_{j=0}^{t_c} g_j^{b_j}$, it must be the case that $B \neq A$. As both are of degree $t_c$, $B(x_i) = A(x_i)$ can only hold for at most $t_c$ indices $i \in [t_c + 1]$. Once this minimal index $\hat{i}$ is identified, the game computes a witness $(c_0, \ldots, c_{\hat{i}}) \leftarrow \text{PExt}((\text{cm}, x_{\hat{i}}, y_{\hat{i}}), \pi_{\hat{i}}, Q)$, i.e., it computes a polynomial $C = \sum_{j=0}^{t_c} c_j x_j$ by running the proof of knowledge extractor another time. If either $\text{cm} \neq \prod_{j=0}^{t_c} g_j^{b_j}$ or $y_{\hat{i}} \neq C(x_{\hat{i}})$, the game aborts and we say the event $\text{Bad}_2$ occurs. Otherwise, it outputs 1 if $\text{G}_1$ does. Again, $\text{G}_1$ and $\text{G}_2$ differ only if $\text{Bad}_2$ occurs and we can bound the probability of $\text{Bad}_2$ using a reduction breaking the proof of knowledge property of $\text{PS}_{\text{open}}$. We get

$$\Pr[\text{G}_1 \Rightarrow 1] \leq \Pr[\text{G}_2 \Rightarrow 1] + \Pr[\text{Bad}_2] \leq \Pr[\text{G}_2 \Rightarrow 1] + \negl(\lambda).$$

Note what we have achieved so far: if $\text{G}_2$ outputs 1, then we know $A(x_i) \neq B(x_i)$ and $A(x_i) = y_i = C(x_i)$. In combination, we get $B(x_i) \neq C(x_i)$ and especially $B \neq C$. Thus, there has to exists at least one $j^*$ such that $b_{j^*} \neq c_j$. But $B$ and $C$ commit to the same element $\text{cm}$. We will now use this to break DLOG.

**Game $\text{G}_3$:** This game is as $\text{G}_2$, but in the beginning it samples $j^* \leftarrow [t_c]$. Then, if $\text{G}_2$ would output 1 but $j^* \neq j^*$, where $j^*$ is as above, the game aborts. By the discussion above, and as $A$’s view is independent from $j^*$, we get

$$\Pr[\text{G}_3 \Rightarrow 1] \geq \frac{1}{t_c + 1} \cdot \Pr[\text{G}_2 \Rightarrow 1].$$

Finally, we bound the probability that $\text{G}_3$ outputs 1. For that, we sketch a reduction that breaks the DLOG assumption whenever $\text{G}_3$ outputs 1:

1. The reduction gets as input parameters $(G, p, g)$ and an element $h = g^x \in G$. The goal is to compute the discrete logarithm $x$ of $h$ to base $g$.
2. The reduction samples $j^* \leftarrow [t_c]$ as $\text{G}_3$ does and sets $g_{j^*} := h$. For every $j \in [t_c]$ with $j \neq j^*$ the reduction samples $\delta_j \leftarrow \mathbb{Z}_p$ and sets $g_j := g^{\delta_j}$.
3. The reduction continues running $\text{G}_3$, which includes extracting the polynomials $B$ and $C$.
4. If $\text{G}_3$ outputs 1, we know that

$$x \cdot b_{j^*} + \sum_{j \neq j^*} b_j \delta_j = x \cdot c_{j^*} + \sum_{j \neq j^*} c_j \delta_j.$$

Isolating $x$, we get

$$x = \frac{\sum_{j \neq j^*} \delta_j (b_j - c_j)}{c_{j^*} - b_{j^*}}.$$

The reduction now computes $x$ in this way and outputs it.

It is clear that the reduction perfectly simulates $\text{G}_3$ for $A$ and finds the correct $x$ whenever $\text{G}_3$ outputs 1. Using the DLOG assumption, we get

$$\Pr[\text{G}_3 \Rightarrow 1] \leq \negl(\lambda).$$

In Lemma C.3, we prove that if an adversary generates a commitment $\text{cm}$, a polynomial $A$ whose commitment is $\text{cm}$, and a valid opening of the commitment, then the opening is consistent with $A$. This can be seen as a relaxed version of the well-known evaluation-binding property of polynomial commitments.
Lemma C.3 Consider parameters \((G, p, g)\). Assume that \(PS_{\text{open}}\) is a proof of knowledge and that the DLOG assumption (i.e., the one-more discrete logarithm assumption of degree 1) holds relative to \((G, p, g)\). Further, consider an algorithm \(A\) and the following experiment:

1. Sample \(g_0, \ldots, g_c \leftarrow G\) and run \(A\) on input \((G, p, g, g_0, \ldots, g_c)\).
2. Obtain from \(A\) a group element \(cm \in G\), a polynomial \(A \in \mathbb{Z}_p[x]\), and a triple \((x, y, \pi)\) where \(x, y \in \mathbb{Z}_p\).
3. Output 1 if and only if the following properties hold, otherwise output 0:
   (a) \(A\) is of degree at most \(t_c\).
   (b) The proof verifies, i.e., \(PVer^H_{\text{open}}((cm, x, y), \pi) = 1\).
   (c) We have \(cm = \prod_{j=0}^{t_c} g_j^{a_j}\), where \(A(X) = \sum_{j=0}^{t_c} a_j X^j\).
   (d) We have \(A(x) \neq y\).

Then, for every PPT algorithm \(A\), the probability that the game outputs 1 is negligible.

Proof. Let \(A\) be a PPT algorithm and let \(\varepsilon\) be the probability that the game in the lemma outputs 1. Our goal is to upper bound \(\varepsilon\). To do so, we provide a sequence of games.

**Game \(G_0\):** This game is the game in the lemma. By definition, we have

\[
\varepsilon = \Pr[G_0 \Rightarrow 1].
\]

**Game \(G_1\):** This game is as \(G_0\), but when \(G_0\) would output 1, the new game \(G_1\) additionally runs \((b_0, \ldots, b_c) \leftarrow \text{PExt}((cm, x, y), \pi, \mathcal{Q})\). Here, \(\text{PExt}\) is the extractor from the proof of knowledge property of \(PS_{\text{open}}\) and \(Q\) is the list of random oracle queries with respect to \(H\). That is, the game tries to extract a witness \((b_0, \ldots, b_c) \in \mathbb{Z}_p^{t_c+1}\) specifying a polynomial \(B(X) = \sum_{j=0}^{t_c} b_j X^j\) from the proof \(\pi\). If either \(cm \neq \prod_{j=0}^{t_c} g_j^{b_j}\) or \(y \neq B(x)\), the game aborts and we say the event \(\text{Bad}_1\) occurs. Otherwise, it outputs 1 as \(G_0\) does. Clearly, \(G_0\) and \(G_1\) differ only if \(\text{Bad}_1\) occurs. We can easily bound the probability of event \(\text{Bad}_1\) using a reduction breaking the proof of knowledge property of \(PS_{\text{open}}\). The reduction forwards random oracle queries and responses between \(A\) and the proof of knowledge game and outputs the statement \((cm, x, y)\) and the proof \(\pi\). We get

\[
\Pr[G_0 \Rightarrow 1] \leq \Pr[G_1 \Rightarrow 1] + \Pr[\text{Bad}_1] \leq \Pr[G_1 \Rightarrow 1] + \text{negl}(\lambda).
\]

Note that if \(G_1\) outputs 1, then we know that \(A(x) \neq y = B(x)\), and thus \(A \neq B\). As a consequence, there exists an index \(j^* \in \llbracket t_c \rrbracket\) such that \(a_{j^*} \neq b_{j^*}\). We will now use this to break DLOG, similarly to what we have done in Lemma C.2.

**Game \(G_2\):** This game is as \(G_1\), but in the beginning it samples \(j^* \leftarrow \llbracket t_c \rrbracket\). Then, if \(G_2\) would output 1 but \(j^* \neq j^*\), where \(j^*\) is as above, the game aborts. By the discussion above, and as \(A\)'s view is independent from \(j^*\), we get

\[
\Pr[G_2 \Rightarrow 1] \geq \frac{1}{t_c + 1} \cdot \Pr[G_1 \Rightarrow 1].
\]

Finally, we bound the probability that \(G_2\) outputs 1. For that, we sketch a reduction that breaks the DLOG assumption whenever \(G_2\) outputs 1:

1. The reduction gets as input parameters \((G, p, g)\) and an element \(h = g^x \in G\). The goal is to compute the discrete logarithm \(x\) of \(h\) to base \(g\).
2. The reduction samples \(j^* \leftarrow \llbracket t_c \rrbracket\) as \(G_2\) does and sets \(g_{j^*} := h\). For every \(j \in \llbracket t_c \rrbracket\) with \(j \neq j^*\) the reduction samples \(\delta_j \leftarrow \mathbb{Z}_p\) and sets \(g_j := g^{\delta_j}\).
3. The reduction continues running \(G_2\), which includes extracting \(B\).
4. If $G_2$ outputs 1, we know that
\[ x \cdot a_j + \sum_{j \neq j^*} a_j \delta_j = x \cdot b_j + \sum_{j \neq j^*} b_j \delta_j. \]

Isolating $x$, we get
\[ x = \frac{\sum_{j \neq j^*} \delta_j (a_j - b_j)}{b_j - a_j}. \]

The reduction now computes $x$ in this way and outputs it.

It is clear that the reduction perfectly simulates $G_2$ for $A$ and finds the correct $x$ whenever $G_2$ outputs 1. Using the DLOG assumption, we get
\[ \Pr [G_2 \Rightarrow 1] \leq \text{negl}(\lambda). \]

\[ \square \]

In Lemma C.4, we prove a variation of the property shown in Lemma C.3. Namely, we show that if an adversary generates a commitment $cm$, a polynomial $A$ whose commitment is $cm$, and a valid exponentiated opening of $cm$, then the exponentiated opening is consistent with $A$. That is, if it provides a group element $Y$ and a valid proof $\pi$ of an exponentiated opening at a point $x$, then $Y = g^{A(x)}$.

**Lemma C.4** Consider parameters $(\mathbb{G}, p, g)$. Assume that $\text{PS}_{\text{exp}}$ is a proof of knowledge and that the DLOG assumption (i.e., the one-more discrete logarithm assumption of degree 1) holds relative to $(\mathbb{G}, p, g)$.

Further, consider an algorithm $A$ and the following experiment:

1. Sample $g_0, \ldots, g_t \leftarrow \mathbb{G}$ and run $A$ on input $(\mathbb{G}, p, g, g_0, \ldots, g_t)$.
2. Obtain from $A$ a group element $cm \in \mathbb{G}$, a polynomial $A \in \mathbb{Z}_p[x]$, and a triple $(x, Y, \pi)$ where $x \in \mathbb{Z}_p$ and $y \in \mathbb{G}$.
3. Output 1 if and only if the following properties hold, otherwise output 0:
   (a) $A$ is of degree at most $t_c$.
   (b) The proof verifies, i.e., $\text{PVer}_\text{exp}^H((cm, x, Y), \pi) = 1$.
   (c) We have $cm = \prod_{j=0}^{t_c} g_j^{a_j}$, where $A(X) = \sum_{j=0}^{t_c} a_j X^j$.
   (d) We have $g^{A(x)} \neq Y$.

Then, for every PPT algorithm $A$, the probability that the game outputs 1 is negligible.

**Proof.** The proof is very similar to that of Lemma C.3. Let $A$ be a PPT algorithm and let $\varepsilon$ be the probability that the game in the lemma outputs 1. Our goal is to upper bound $\varepsilon$. To do so, we provide a sequence of games.

**Game $G_0$:** This game is the game in the lemma. By definition, we have
\[ \varepsilon = \Pr [G_0 \Rightarrow 1]. \]

**Game $G_1$:** This game is as $G_0$, but when $G_0$ would output 1, the new game $G_1$ additionally runs $(b_0, \ldots, b_t) \leftarrow \text{PExt}((cm, x, Y), \pi, Q)$. Here, $\text{PExt}$ is the extractor from the proof of knowledge property of $\text{PS}_{\text{exp}}$ and $Q$ is the list of random oracle queries with respect to $H$. That is, the game tries to extract a witness $(b_0, \ldots, b_t) \in \mathbb{Z}_p^{t+1}$ specifying a polynomial $B(X) = \sum_{j=0}^{t} b_j X^j$ from the proof $\pi$. If either $cm \neq \prod_{j=0}^{t_c} g_j^{b_j}$ or $Y \neq g^{B(x)}$, the game aborts and we say the event $\text{Bad}_1$ occurs. Otherwise, it outputs 1 as $G_0$ does. Clearly, $G_0$ and $G_1$ differ only if $\text{Bad}_1$ occurs. We can easily bound the probability of event $\text{Bad}_1$ using a reduction breaking the proof of knowledge property of $\text{PS}_{\text{exp}}$. The reduction forwards random oracle queries and responses between $A$ and the proof of knowledge game and outputs the statement $(cm, x, Y)$ and the proof $\pi$. We get
\[ \Pr [G_0 \Rightarrow 1] \leq \Pr [G_1 \Rightarrow 1] + \Pr [\text{Bad}_1] \leq \Pr [G_1 \Rightarrow 1] + \text{negl}(\lambda). \]
Note that if $G_1$ outputs 1, then we know that $g^{A(x)} \neq Y = g^{B(x)}$, and thus $A \neq B$. As a consequence, there exists an index $j^* \in [t_c]$ such that $a_{j^*} \neq b_{j^*}$. But $A$ and $B$ commit to the same element $cm$. We will now use this to break DLOG, similarly to what we have done in previous lemmata.

**Game $G_2$:** This game is as $G_1$, but in the beginning it samples $j' \leftarrow [t_c]$. Then, if $G_2$ would output 1 but $j' \neq j^*$, where $j^*$ is as above, the game aborts. By the discussion above, and as A’s view is independent from $j'$, we get
\[
\Pr [G_2 \Rightarrow 1] \geq \frac{1}{t_c + 1} \cdot \Pr [G_1 \Rightarrow 1].
\]

Finally, we bound the probability that $G_2$ outputs 1. For that, we sketch a reduction that breaks the DLOG assumption whenever $G_2$ outputs 1:

1. The reduction gets as input parameters $(G, p, g)$ and an element $h = g^r \in G$. The goal is to compute the discrete logarithm $x$ of $h$ to base $g$.
2. The reduction samples $j' \leftarrow [t_c]$ as $G_2$ does and sets $g_{j'} := h$. For every $j \in [t_c]$ with $j \neq j'$ the reduction samples $\delta_j \leftarrow \mathbb{Z}_p$ and sets $g_j := g^{\delta_j}$.
3. The reduction continues running $G_2$, which includes extracting $B$.
4. If $G_2$ outputs 1, we know that
\[
x \cdot a_{j^*} + \sum_{j \neq j^*} a_j \delta_j = x \cdot b_{j^*} + \sum_{j \neq j^*} b_j \delta_j.
\]

Isolating $x$, we get
\[
x = \frac{\sum_{j \neq j^*} \delta_j (a_j - b_j)}{b_{j^*} - a_{j^*}}.
\]

The reduction now computes $x$ in this way and outputs it.

It is clear that the reduction perfectly simulates $G_2$ for $A$ and finds the correct $x$ whenever $G_2$ outputs 1. Using the DLOG assumption, we get
\[
\Pr [G_2 \Rightarrow 1] \leq \text{negl}(\lambda).
\]

In Lemma C.5, we show that once $t_c + 1$ honest parties send their “column” messages as in the protocol specification (cf. Algorithm 3), it is possible to efficiently extract a polynomial for the AVSS instance from their views. By this, we mean that it is possible to compute a bivariate polynomial $S(X, Y)$ of correct degrees in $X$ and $Y$ such that all opened evaluations are consistent with $S$ and all of the exponentiated openings $S_i$ are such that $S_i = g^{S(i, 0)}$ holds. Informally, we do this by taking the evaluations of these $t_c + 1$ honest parties along each of the columns. Each set of $t_c + 1$ evaluations should fully define the committed polynomial, as shown in Lemma C.2. All opened evaluations and exponentiated evaluations should be consistent with these polynomials, as shown in Lemmata C.3 and C.4. Once this is established, it will be easy to show that all honest parties’ outputs will be consistent with $S$ because they compute their outputs as functions of the openings they received. Note that this process can actually be done with any set of $t_c + 1$ honest parties that sent “column” messages, but we choose the first $t_c + 1$ parties that do so. If we construct a polynomial in this way from any other set of $t_c + 1$ parties, the evaluations must also be consistent with $S$ (as shown in the following lemma), and thus we will also construct $S$.

**Lemma C.5** Consider parameters $(G, p, g)$ and $g_0, \ldots, g_{t_c} \leftarrow \mathbb{G}$. Assume that $\text{PS}_{\text{open}}$ and $\text{PS}_{\text{exp}}$ are proofs of knowledge and that the DLOG assumption holds relative to $(G, p, g)$. Consider the experiment of running HAVSS.Share as specified in Algorithm 3. Assume that in this experiment $t_c + 1$ honest parties send “column” messages at some point throughout the protocol after they receive a (“commits”, $CM'$, $(S_i, \pi_i^{\text{exp}})_{i \in [t_c + 1]}$) broadcast from the dealer.

Then, it is possible to efficiently compute a bivariate polynomial $S(X, Y)$ of degree $t_r$ in $X$ and $t_c$ in $Y$ from the views of the first $t_c + 1$ honest parties that do so (i.e., send “column” messages) such that for every PPT adversary corrupting at most $t_c$ parties, the following hold with all but negligible probability:
(a) For all \( i \in [t_r + 1] \), we have \( S_i = g^{S(i,0)} \), and

(b) if any honest \( P_i \) adds \((j, C((j)))\) to \( \text{points}_{col,i} \), then \( C_i(j) = S(i,j) \).

Further, if the dealer is honest, then \( S(X,Y) \) is the polynomial it sampled in HAVSSDeal (cf. Algorithm 2).

Proof. Assume that \( t_r + 1 \) honest parties sent such messages, and let \( I \subseteq [n] \) be the indices of the first honest parties to do so. Before doing so, they must have had \( CM \neq \perp \), and thus they received a “commits” message from the dealer, verified it, and updated both \( CM \) and \( S \) by using the ExpInterpolate algorithm on the received values. In addition, each \( P_i \) received a “row” message containing a vector \( \{(C_j(i), \pi_{j,i})\}_{j \in [n]} \) and verifying it before sending the “column” message. For each \( i \in I \), define \( R_i(X) \) to be the unique polynomial of degree \( t_r \) or less such that \( R_i(j) = C_j(i) \) for every \( i \in [t_r + 1] \). In addition, let \( S(X,Y) \) be the unique polynomial of degree at most \( t_r \) in \( X \) and at most \( t_c \) in \( Y \) such that \( S(X,i) = R_i(X) \) for every \( i \in I \). Note that it is indeed possible to efficiently interpolate both of these polynomials from the views of the aforementioned parties. Also, if the dealer is honest, then it sampled a polynomial \( S'(X,Y) \) and sent each party \( P_i \) “row” messages with evaluations on the polynomial \( R'_i(X) = S'(X,i) \). These means that the interpolated \( R_i \) polynomials are the \( R'_i \) polynomials and thus \( S(X,Y) \) is the polynomial \( S'(X,Y) \) sampled by the dealer.

Now that this is established, we need to bound the probability that one of the two properties in the lemma do not hold, with respect to \( S(X,Y) \). To this end, let \( A \) be a PPT adversary running in the protocol. We denote by \( \varepsilon \) the probability that there is an \( i \in [t_r + 1] \) with \( S_i \neq g^{S(i,0)} \) or some honest party \( P_i \) adds \((j, C((j)))\) to \( \text{points}_{col,i} \) but \( C_i(j) \neq S(i,j) \). We have

\[
\varepsilon = \Pr[\mathcal{G}_0 \Rightarrow 1].
\]

Game \( \mathcal{G}_0 \): This game is the game that runs the protocol with the adversary. It outputs 1 if there is an \( i \in [t_r + 1] \) with \( S_i \neq g^{S(i,0)} \) or some honest party \( P_i \) adds \((j, C((j)))\) to \( \text{points}_{col,i} \) but \( C_i(j) \neq S(i,j) \). We have

\[
\varepsilon = \Pr[\mathcal{G}_0 \Rightarrow 1].
\]

Game \( \mathcal{G}_1 \): This game is the same as \( \mathcal{G}_0 \), but if at any point \( t_r + 1 \) honest parties send “column” messages after accepting a “commits” message from the dealer, it computes \( R_i(X) \) for every such party and \( S(X,Y) \) as defined above. It then computes \( C_i(Y) = S(i,Y) = \sum_{k=0}^{t_r} a_{k,i} Y^k \) for every \( i \in [t_r + 1] \). If \( cm_i \neq \prod_{k=0}^{t_r} g_{a_{k,i}} \) for any \( i \in [t_r + 1] \), then \( \mathcal{G}_1 \) aborts and we say that event \( \text{Bad}_1 \) occurred. Otherwise, it acts identically to \( \mathcal{G}_0 \) and outputs 1 when it does.

Clearly, \( \mathcal{G}_0 \) and \( \mathcal{G}_1 \) only differ if the event \( \text{Bad}_1 \) occurs. We can bound the probability that the event occurs by the following reduction to Lemma \( C.2 \):

1. The reduction gets as input parameters \( (G,p,g,g_0,\ldots,g_n) \).

2. The reduction runs the protocol as \( \mathcal{G}_1 \) does, and if the event \( \text{Bad}_1 \) ever occurs, it finds an index \( i \in [t_r + 1] \) such that \( cm_i \neq \prod_{k=0}^{t_r} g_{a_{k,i}} \). It then outputs \( cm_i \) and the triples \( ((x_i,y_i,\pi_i))_{i \in I} \) where \( I \) is the set of \( t_c + 1 \) honest parties that sent “row” messages first.

Note that before sending a “column” message, every honest party makes sure that all of the openings and proofs it received in its “row” message verify. This means that if the event \( \text{Bad}_1 \) occurred, the reduction wins in the game described in Lemma \( C.2 \). Therefore:

\[
\Pr[\mathcal{G}_0 \Rightarrow 1] \leq \Pr[\mathcal{G}_1 \Rightarrow 1] + \Pr[\text{Bad}_1] \leq \Pr[\mathcal{G}_1 \Rightarrow 1] + \negl(\lambda).
\]

Game \( \mathcal{G}_2 \): This game is the same as \( \mathcal{G}_1 \), and it computes \( S \) in the same way as \( \mathcal{G}_1 \). However, it stores all “column” messages received by honest parties throughout the protocol. If at any point, an honest party \( i \) receives a (“column”, \( C_i(j), \pi_{i,j} \)) message such that \( C_i(j) \neq S(i,j) \) and \( \text{PVeropen}(cm_i, j, C_i(j), \pi_{i,j}) = 1 \), \( \mathcal{G}_2 \) aborts and we say that event \( \text{Bad}_2 \) occurred. Otherwise, \( \mathcal{G}_2 \) continues as the same as \( \mathcal{G}_1 \) and outputs 1 in the same situations. Clearly, \( \mathcal{G}_1 \) and \( \mathcal{G}_2 \) only differ if the event \( \text{Bad}_1 \) does not occur and the event \( \text{Bad}_2 \) does occur. We can bound the probability that \( \text{Bad}_2 \) takes place given that \( \text{Bad}_1 \) did not take place by the following reduction to Lemma \( C.3 \):

1. The reduction gets as input parameters \( (G,p,g,g_0,\ldots,g_n) \).

2. The reduction runs the protocol as \( \mathcal{G}_2 \) does, and if the event \( \text{Bad}_2 \) ever occurs without \( \text{Bad}_1 \) taking place, it find the indices \( i,j \) described above such that \( \text{PVeropen}(cm_i, j, C_i(j), \pi_{i,j}) = 1 \), but \( C_i(j) \neq S(i,j) \). It then outputs \( cm_i \), the polynomial \( C_i(Y) = S(i,Y) \) and \( (j, C_i(j), \pi_{i,j}) \).
Given that Bad1 does not occur, cmi is indeed a commitment to Ci. That is, if \( C_i(Y) = \sum_{k=0}^{t_c} a_k X^k \), then \( cm_i = \prod_{k=0}^{t_c} g_k^{a_k} \). Therefore, the reduction outputs a commitment, a consistent polynomial and an inconsistent opening that verifies, winning in the game described in Lemma C.3. Note that G1 and G2 only differ if Bad1 occurs, and thus:

\[
\Pr[G_1 \Rightarrow 1] \leq \Pr[G_2 \Rightarrow 1] + \Pr[\text{Bad}_2 \Rightarrow \neg \text{Bad}_1] \leq \Pr[G_2 \Rightarrow 1] + \negl(\lambda).
\]

Finally, we bound the probability of G2 outputting 1. First of all, the game never outputs 1 if the events Bad1 or Bad2 take place. This means that no honest party i received a "column" message with and verifying proof that is inconsistent with \( C_i(Y) = S(i, Y) \). Since these are the only values it adds to points, G2 cannot output 1 as a result of the second bullet described in the lemma. Therefore, G2 only outputs 1 if there exists some \( i \in [t_c + 1] \) such that \( S_i \neq g^{S(i,0)} \). We denote this event Bad3. Note that honest parties only send their "column" messages if they receive a "commits" broadcast from the dealer with verifying proofs. Thus, we can construct the following reduction to Lemma C.4:

1. The reduction gets as input parameters \((G, p, g, g_0, \ldots, g_n)\).
2. The reduction runs the protocol as G2 does, and if the event Bad3 ever occurs without Bad1 taking place, it finds the index \( i \) described above such that \( \text{PVer}_\text{exp}(cm_i, 0, S_i, \pi^{\text{exp}}_i) = 1 \), but \( S_i \neq g^{S(i,0)} \). It then outputs cmi, the polynomial \( C_i(Y) = S(i, Y) \) and \((0, S_i, \pi^{\text{exp}}_i)\).

As before, G2 only outputs 1 if the event Bad3 occurs and the event Bad1 does not occur and thus:

\[
\Pr[G_2 \Rightarrow 1] \leq \Pr[\text{Bad}_3 \Rightarrow \neg \text{Bad}_1] \leq \negl(\lambda).
\]

To prove security of our AVSS (Theorem 6.2), we need to prove correctness and termination as defined in Definition 2.4 and oracle-aided simulatability as defined in Definition 5.1. We do so in the following lemmata. We begin with the correctness property.

**Lemma C.6** Assume that PS_{open} and PS_{exp} are proofs of knowledge and that the DLOG assumption holds relative to the parameters of the protocol \((G, p, g)\). Then, HAVSS = (HAVSS.Share, HAVSS.Rec) has the Correctness property.

**Proof.** To recall, we need to show that – in presence of at most \( t_c \) corruptions – when the first honest party completes HAVSS.Share, then there is a unique polynomial \( f \in \mathbb{Z}_p[X] \) of degree at most \( t_c \) such that every honest party completing HAVSS.Rec outputs \((S, S_1, \ldots, S_n)\) and \( f(i) \in \mathbb{Z}_p \) with \( S = g^{f(0)} \) and \( S_j = g^{f(j)} \) for all \( j \in [n] \). If the dealer is honest, it has to hold that \( f(0) \) is equal to the dealer's input.

We now give the proof. Assume some honest party completes the protocol. Before it does so, it receives “done” messages from \( n - t_c \) different parties. We know that \( n - t_c \geq 2t_c + 1 \) and at least one of these messages was sent by an honest party. Consider the first honest party to send a “done” message. At that time, it did not receive a “done” message from another honest party, so it received at least \( t_c \) “done” messages. This means that it must have sent its “done” message after receiving “vote” messages from at least \( n - t_c \) parties. Out of those, at least \( n - 2t_c \geq t_c + 1 \) are honest. Honest parties only send such messages if they have CM \( \neq \bot \), and they only update CM after receiving a “commits” broadcast from the dealer. In addition, before sending “vote” messages, these parties also send “column” message to everybody. Therefore we have successfully argued that the conditions of Lemma C.5 hold. Now, recall Lemma C.5 implies the existence of a bivariate polynomial \( S(X, Y) \) with certain properties. We will show that the Correctness property holds with respect to \( f(X) = S(X, 0) \).

First, \( S \) is of degree \( t_c \) in \( X \) and thus \( f \) is also of degree \( t_c \). Let row0 = \((S_t, \pi^{\text{exp}}_t)\)\( \in [t_c + 1] \) be the value broadcast by the dealer in its “commits” message. By Lemma C.5, we have \( S_t = g^{S(t,0)} = g^{f(0)} \) for every \( i \in [t_c + 1] \). Parties then interpolate the \( S_t \) values and evaluate them at 0 to get \( S_0 \) and at \( t_c + 2, \ldots, n \) to get \( S_{t_c+2}, \ldots, S_n \). Since the discrete logarithms of the \( S_t \) elements lie on \( f(X) \) for every \( i \in [t_c + 1] \), it is also true that \( S_t = g^{f(i)} \) for every one of the interpolated values. In addition, assume some honest party \( P_i \) completed the HAVSS.Share protocol. It only did so after having \( C_i \neq \bot \), which it updated by interpolating the pairs \((j, C_i(j))\) in points. Once there are \( t_c + 1 \) such pairs. By Lemma C.5, for every such pair \((j, C_i(j))\), we have \( C_i(j) = S(i, j) \). This means that the interpolated polynomial \( C_i(Y) \) must be
$S(i, Y)$ the unique polynomial of degree $t_c$ or less that is consistent with all of these evaluations. When parties output values in HAVSS.Rec, they output $(S_0, \ldots, S_n)$ and $C_i(0)$. As shown above, $S_i = g^{f(i)}$ for every $i \in [0, n]$. In addition, $C_i(0) = S(i, 0) = f(i)$, as required. Finally, if the dealer is honest, then $S$ is the polynomial sampled by it, meaning that $f(0) = S(0, 0) = s$.

**Lemma C.7** Assume that $PS_{open}$ is a proof of knowledge and that the DLOG assumption holds relative to the parameters of the protocol $(G, p, g)$. Then, HAVSS = (HAVSS.Share, HAVSS.Rec) has the Termination property.

**Proof.** We have to prove three properties, namely (1) if the dealer is honest and all honest parties call HAVSS.Share, then all honest parties complete HAVSS.Share, (2) if all honest parties call HAVSS.Share and an honest party completes HAVSS.Share, then all honest parties complete HAVSS.Share, and (3) if all honest parties call HAVSS.Rec, then all honest parties complete HAVSS.Rec.

We start with (1). So, assume that the dealer is honest and that all honest parties called HAVSS.Share. Recall that the dealer starts by sampling a polynomial $S(X, Y)$ and computing $S_i = g^{S(i, 0)}$, $C_i(Y) = \sum_{k=0}^{t_c} a_k Y^k$ and $cm_i = \prod_{k=0}^{t_c} g_k^{a_k i}$ for every $i \in [t_c + 1]$. It then reliably broadcasts a “commits” message containing these $S_i$ and $cm_i$, values along with proofs that the discrete log of $S_i$ is indeed $C_i(0)$. It similarly computes $cm_i$ and $C_i(Y)$ for every other $i \in [t_c + 2, n]$, and sends every $P_i$ the values $C_i(i)$ for every $j \in [n]$ along with proofs that these are the correct values in “row” messages. Since all of these proofs are generated honestly, all honest parties will accept these proofs (by completeness of the non-interactive proof system), update their $CM$ and $S$ variables and send “vote” messages. They will also accept the “row” message and send “column” messages to all parties with the same values and proofs. Every honest party will accept “column” messages from every other honest party, for a total of at least $n - t_c$ messages. After accepting $t_c + 1$ such messages, every honest party $P_i$ will interpolate a polynomial $C_i$. In addition, after receiving “vote” messages from $n - t_c$ parties, every honest party will send a “done” message to all parties. After receiving those messages from all honest parties, every party will have received “done” messages from $n - t_c$ parties and will have $S \neq \perp$ and $C_i \neq \perp$. Consequently, it will terminate.

We now turn to proving (2). To this end, assume some honest party terminates. This means that it received “done” messages from $n - t_c$ parties. Out of those, at least $n - 2t_c \geq t_c + 1$ are honest, so every honest party will receive $t_c + 1$ such messages and also send a “done” message. This means that every party will receive “done” messages from at least $n - t_c$ parties. Consider the first honest party to send a “done” message. It could not have done so as a result of receiving “done” messages from $t_c + 1$ different parties, because that would mean it received one of those messages from an honest party, contradicting the fact that it is the first to do so. This means that it sent the message after receiving “vote” messages from $n - t_c$ different parties. Out of those parties, at least $n - 2t_c \geq t_c + 1$ are honest. Before sending their “vote” messages, every one of those parties receives a “commits” broadcast with verifying proofs. Accordingly, it updates $CM$ and $S$. Following that, they also receive a “row” message with a vector $\text{row}_i = ((C_i(i), \pi_{i,j}))_{i \in [n]}$ with verifying proofs. Every one of those parties $P_i$ sends a (“column”, $C_i(i)$, $\pi_{i,j}$) to every $P_j$. Every honest $P_j$ receives the same “commits” broadcast, sees that the proofs verify and updates $CM$ and $S$. They also receive the “column” messages sent by the honest parties that sent “vote” messages, see that the proofs verify, and add a tuple to $\text{points}_{\text{col}, j}$ each time. After adding $t_c + 1$ different tuples, $P_j$ interpolates $\text{points}_{\text{col}, j}$ and stores the result in $C_i$. In total, every honest party $P_j$ eventually receives “done” messages from at least $n - t_c$ parties and has $S \neq \perp$ and $C_i \neq \perp$. At that point, every honest party terminates.

Finally, (3) follows directly from the fact that parties immediately terminate upon starting HAVSS.Rec. This completes the proof.

In order to prove that the protocol has the Oracle-Aided Simulatability property, we first define a bad event for the Bad Event property, and show that there is a negligible property of it taking place.

**Lemma C.8** Assume that $PS_{open}$ is a proof of knowledge and that the DLOG assumption holds relative to the values $(G, p, g)$ used in the public parameters of the protocol. Then, HAVSS = (HAVSS.Share, HAVSS.Rec) has the Bad Event property.

**Proof.** First, we will define two separate sub-events as follows:

- **BadOpen**: This is the event in an execution of the real protocol in which an honest dealer sampled the polynomial $S$ and some honest party $P_i$ received a (“column”, $C_i(j)$, $\pi_{i,j}$) message from a corrupt party $P_j$ such that $C_i(j) \neq S(j, i)$ and $P_{\text{Ver}}^\text{H}(cm_i, j, C_i(j)) = 1$.
• \textbf{Bad}_M: Define the parameters to be \((G, p, g, g_0, \ldots , g_{t_c})\) and let \(\delta_0, \ldots , \delta_{t_c} \in \mathbb{Z}_p\) be field elements such that \(\forall i \in [t_c], g_i = g^{\delta_i}\). Observe the set of corrupted parties \(C \subseteq [n]\) at any time throughout the protocol and let \(I \subseteq [n]\) be the set of \(t_c - |C|\) minimal indices that aren’t in \(C\). Let \(M\) be the matrix whose first row is \((\delta_0, \ldots , \delta_{t_c})\) and the next \(t_c\) rows are Vandermonde rows \((j^0, \ldots , j^{t_c})\) for \(j \in I \cup C\). We define \(\emptyset^0\) to be the value 1 in all of these computations. Define the event \textbf{Bad}_M as the event that at any time throughout the run, the matrix \(M\) is not invertible.

Finally, define \textbf{Bad} as \(\textbf{Bad}_\text{Open} \lor \textbf{Bad}_M\). Let \(A\) be some PPT adversary for the protocol. Our goal is to bound the probability of the event \textbf{Bad} taking place. We will do so by bounding the probability of each one of the events taking place.

\textbf{Bounding \textbf{Bad}_\text{Open}}. We will bound the probability of \textbf{Bad}_\text{Open} by constructing a reduction to Lemma C.3 that acts as follows:

1. The reduction gets as input parameters \((G, p, g, g_0, \ldots , g_{t_c})\).
2. The reduction runs the protocol by running the adversary and controlling all honest parties, instructing them to act honestly. It stores the polynomial \(S\) sampled by an honest dealer and all “column” messages received by honest parties. If it sees that at any point \textbf{Bad}_\text{Open} occurs by checking if at any point an honest party receives a verifying “column” message with an evaluation that is inconsistent with \(S\). If that happens, some honest party received a \("\text{column}\", C_i(j), \pi_{i,j}\) message from a corrupt party \(P_i\) such that \(C_i(j) \neq S(j, i)\) and \(P_{\text{Ver}}(\text{cm}_i, j, C_i(j)), \pi_{i,j}) = 1\). It then outputs \(\text{cm}_i, S(i, Y), (j, C_i(j), \pi_{i,j})\).

An honest dealer does compute the polynomial \(S(i, Y) = \sum_{k=0}^{t_c} a_k Y^k\) and send the commitment \(\text{cm}_i = \prod_{k=0}^{t_c} g_k^{a_k}\). Therefore, if the event \textbf{Bad}_\text{Open} takes place, the reduction outputs a commitment \(\text{cm}_i\), a corresponding polynomial \(S(i, Y)\) and a verifying opening \((j, C_i(j), \pi_{i,j})\) such that \(C_i(j) \neq S(i, j)\), meaning that it wins in the game described in Lemma C.3. As shown in Lemma C.3, the probability of this taking place is negligible and thus:

\[
\Pr[\textbf{Bad}_\text{Open}] \leq \text{negl}(\lambda).
\]

\textbf{Bounding \textbf{Bad}_M}. We will bound the probability that \textbf{Bad}_M takes place by the following reduction to the DLOG assumption:

1. The reduction gets as input parameters \((G, p, g, h)\) and attempts to find the discrete log of \(h\) with respect to \(g\).
2. For every \(i \in [t_c - 1]\) the reduction samples a field element \(\delta_i \leftarrow \mathbb{Z}_p\) and defines \(g_i = g^{\delta_i}\). In addition, it defines \(g_{t_c} = h\).
3. The reduction runs the protocol with parameters \((G, p, g, g_0, \ldots , g_{t_c})\) by controlling all honest parties and running the adversary \(A\).
4. Whenever a party is corrupted, the reduction adds its index to \(C\) In the beginning of the protocol and after each such update, the reduction defines \(I \subseteq [n]\) to be the \(t_c - |C|\) minimal indices not in \(C\). Let \(V\) be a matrix with \(t_c\) Vandermonde rows \((j^0, \ldots , j^{t_c})\) for different values \(j \in I \cup C\) and let its columns be \(V^0, \ldots , V^{t_c}\). The reduction finds the unique coefficients \(a_0, \ldots , a_{t_c-1}\) such that \(V^{t_c} = \sum_{k=0}^{t_c-1} a_k V^k\) by Gaussian elimination. It then computes \(x = \sum_{k=0}^{t_c-1} a_k b_k\). If \(h = g^x\), the reduction outputs \(x\) and otherwise it continues running the protocol.

The adversary’s view is identical in the reduction and in a real execution, since the parameters are sampled identically. If the event \textbf{Bad}_M takes place, then after adding some index to \(C\), the matrix \(M\) was not invertible. The matrix has one row of the form \((\delta_0, \ldots , \delta_{t_c-1}, x)\) where \(h = g^x\), and the rest of the rows are the rows of \(V\). Since the rows of \(V\) are Vandermonde rows, the matrix whose columns are \(V^0, \ldots , V^{t_c-1}\) is invertible. This means that the vectors \(V^0, \ldots , V^{t_c-1}\) are a basis for \(\mathbb{Z}_p^{t_c}\), and thus there is a unique choice of coefficients \(a_0, \ldots , a_{t_c-1}\) such that \(V^{t_c-1} = \sum_{k=0}^{t_c-1} a_k V^k\). The matrix \(M\) is not invertible if and only if this linear relationship holds for the first row as well. In other words, it is not
invertible if and only if $x = \sum_{k=0}^{t_{c}-1} a_k \delta_k$. In this case, the reduction outputs $x$ and succeeds in finding the discrete log of $h$. Therefore:

$$\Pr[\text{Bad}_M] \leq \text{negl}(\lambda).$$

Combining these two results, we find that:

$$\Pr[\text{Bad}] = \Pr[\text{Bad}_\text{Open} \lor \text{Bad}_M] \leq \Pr[\text{Bad}_\text{Open}] + \Pr[\text{Bad}_M] \leq \text{negl}(\lambda).$$

\[\square\]

**Lemma C.9** Assume that $PS_{\text{open}}$ and $PS_{\text{exp}}$ are zero-knowledge and that the DLOG assumption holds relative to the parameters of the protocol $(G, p, g)$. Then, assuming secure erasures, HAVSS = (HAVSS.Share, HAVSS.Rec) has the Oracle-Aided Simulatibility property.

**Proof.** We will start by constructing a simulator $\text{Sim}$. The simulator receives an OMDL instance $g^{z_1}, \ldots, g^{z_{t_{c}+1}}$ of elements in $G$. It starts by defining $C = \emptyset$. It then uniformly samples discrete logs $(\delta_0, \ldots, \delta_{t_c}) \leftarrow \mathbb{Z}_p^{t_{c}+1}$ and defines the parameters $g_0 = g^{\delta_0}, \ldots, g_{t_c} = g^{\delta_{t_c}}$. $\text{Sim}$ then runs the protocol.

**Honest dealer activation.** When the dealer is first activated, if it is not corrupted, $\text{Sim}$ sets $S_i = g^{z_i}$, for every $i \in \{t_c+1\}$. $\text{Sim}$ then interpolates the set $\{(i, S_i)\}_{i \in \{t_{c}+1\}}$ in the exponent and evaluates the exponentiated polynomial at 0 to get $S_0$ and at $t_c + 2, \ldots, n$ to get the values $S_{t_c+2}, \ldots, S_n$. Following that, $\text{Sim}$ uniformly samples $t_{c} + 1$ values $(r_1, \ldots, r_{t_{c}+1}) \leftarrow \mathbb{Z}_p^{t_{c}+1}$. Note that $\text{ExpInterpolate}$ performs a constant linear operation in the exponent on $t_{c} + 1$ group elements to generate $n$ group elements. $\text{Sim}$ performs the same linear operation on $r_1, \ldots, r_{t_{c}+1}$ to generate $n$ field elements $r_1, \ldots, r_{n}$ and computes $c_m = g^{r_i}$ for every $i \in \{n\}$. Finally, $\text{Sim}$ simulates proofs $\pi_{x_{j}^{\text{exp}}}$ for every $i \in \{t_{c}+1\}$ showing that $(c_m, 0, S_i)$ is in the relation $\mathcal{R}_{\exp}$. $\text{Sim}$ adds messages to the queue as if it broadcasted a “commits” message with $CM = (c_{m_1}, \ldots, c_{m_{t_{c}+1}})$ and $row_0 = (\langle S_1, \pi_{x_{1}^{\text{exp}}}, \ldots, (S_{t_{c}+1}, \pi_{x_{t_{c}+1}^{\text{exp}}}) \rangle)$, as well as “row” messages for all parties.

**Honest party receiving a message.** Whenever an honest party (i.e., a party not in $C$) gets a message, $\text{Sim}$ accepts it as a correct message if it was sent by a party that was honest at the time, or checks if it was sent by a corrupt party. $\text{Sim}$ then adds any messages that are sent as a result of processing messages. In more detail, whenever an honest party $P_i$ receives a message from party $P_j$, $\text{Sim}$ acts as follows:

- If the message is a “commits” message the dealer $P_j$, then if the dealer is honest, $\text{Sim}$ considers $P_i$ as having received a commitment $CM$. Otherwise, $\text{Sim}$ checks that the message contains a commitment $CM' = (c_{m_1}, \ldots, c_{m_{t_{c}+1}})$ and a row $row_0 = (\langle S_1, \pi_{x_{1}^{\text{exp}}}, \ldots, (S_{t_{c}+1}, \pi_{x_{t_{c}+1}^{\text{exp}}}) \rangle)$ such that $\forall j \in \{t_{c}+1\} \text{PVer}_{\text{exp}}^i((c_{m_j}, 0, S_j), \pi_{x_{j}^{\text{exp}}}) = 1$. If that is the case, $\text{Sim}$ considers $P_i$ as having received the commitment $CM = \text{ExpandCom}(CM')$. $\text{Sim}$ delays the execution of any following bullets for $P_i$ until $P_i$ has received a commitment.

- If the message is a “row” message from the dealer, then if the dealer is honest, $\text{Sim}$ considers $P_i$ as receiving a correct polynomial. If the dealer is corrupt, and the message contains a row $row_i = (\langle C_i(1), \pi_{i,1}, \ldots, (C_i, \pi_{i,n}) \rangle)$, then $P_i$ is said to have received a correct polynomial only if for all $j \in \{n\}$, we have $\text{PVer}_{\text{exp}}^i((c_{m_j}, i, C_j(i)), \pi_{j,i}) = 1$. In both of these cases, if $P_i$ received a correct polynomial, then $\text{Sim}$ adds “column” and “vote” messages to the queue from $i$ to all parties if it hasn’t done so earlier.

- If the message is a “column” message with the values $y_j, \pi_{ij}$ from $P_j$, then if $P_j$ is honest, $\text{Sim}$ considers $P_i$ as receiving a correct message from $P_j$. If $P_j$ is corrupt, then $P_i$ checks that $\text{PVer}_{\text{open}}^i((c_{m_j}, i, y_j), \pi_{ij}) = 1$ and if that is the case, it considers $P_i$ as receiving a correct message from $P_j$. If the received message is the $t_c + 1$’th correct “column” message received by $P_i$, then $\text{Sim}$ considers $P_i$ as having interpolated $C_i$.

- If $P_i$ received “vote” messages from $n - t_c$ different parties or “done” messages from $t_c + 1$ different parties, add “done” messages to the queue from $i$ to all parties.

- If $P_i$ received “done” messages from $n - t_c$ parties, and it has received a commitment from the dealer and interpolated a polynomial $C_i$, $\text{Sim}$ considers it as having completed the HAVSS.Share protocol.
If at any time an honest party $P_i$ is activated in the HAVSS.Share protocol, $Sim$ waits until it completes the HAVSS.Share protocol, and then immediately considers it as having completed the protocol.

**Party corruption.** Whenever a party $P_i$ is corrupted, $Sim$ starts by defining its polynomials $C_i$ and $R_i$ under the condition that $C_i$ is also consistent with the commitment $cm_i$ and that $C_i(j) = R_i(j)$ for every corrupt $P_j$. First, it defines a polynomial $C_i$ for $P_i$. It does so by calling its discrete log oracle on $S_i$ and set the response to be the value $R_0(i)$. The algebraic representations of $S_i$ is the one defined by the linear transformation for interpolating and evaluating the $S_i$. $Sim$ defines the set $I \subseteq [n]$ to be the $t_c - |C|$ minimal indices that aren’t in $C$. Note that $0$ is always in $I$. For every $j \in I \setminus (C \cup \{0\})$, $Sim$ uniformly samples a value $R_j(i) \leftarrow \mathbb{Z}_p$. It then constructs a matrix $M$ of dimensions $(t_c + 1) \times (t_c + 1)$ where the first row is the vector $(\delta_0, \ldots, \delta_{t_c})$ and the next $t_c$ rows are Vandermonde rows of form $(j^0, j^1, \ldots, j^{t_c})$ for different values $j \in I \cup C$. We define $0^0$ to be $1$ in all of these vectors. If the resulting matrix is not invertible, $Sim$ aborts. $Sim$ then defines the vector $v = (r_i, R_0(i), \ldots, R_{b_i}(i))^T$ with $j_1, \ldots, j_{t_r}$ being the indices of $I \cup C$ in the same order as above. Finally, $Sim$ computes the vector $(a_i, 0, \ldots, a_i, t_r)^T = M^{-1}v$ and defines $C_i(Y) = \sum_{k=0}^{t_r} a_{i,k}Y^k$. Note that by construction, $r_i = \sum_{k=0}^{t_r} \delta_{i,k}$ and $C_i(j) = R_i(i)$ for every $j \in I$. This also means that $C_i(0) = R_0(i)$ and thus $S_i = g^{C_i(0)}$. Following that, $Sim$ adds to $i$ to $C$ and turns to define $R_i(X)$. It does so by simply uniformly sampling a polynomial of degree $t_c$ under the condition that for every $j \in C$, $R(j) = C_i(j)$.

$Sim$ then saves $C_i$ and $R_i$. Note that there are at most $t_c$ corruptions. Since $I$ is defined before adding a newly corrupted party’s index to $C$, at that time $C$ is of size $t_c - 1$ at most, and thus it is possible find a set $I$ as defined above, and it always includes at least one index, namely $0$. In addition, there is indeed at least one polynomial $R_i$ of degree $t_c \geq t_c$ for which the required equalities hold.

**Delivery of message to corrupt party.** After sampling these polynomials, whenever $Sim$ delivers a message to $P_i$ from an honest party $P_j$, it does so in a consistent way with $C_i$ and $R_i$. Similarly, before providing $P_i$’s state to the adversary, it updates it as if it received similar messages from honest parties. In more detail whenever $P_i$ receives a message from an honest $P_j$:

- If the message is a “commits” message from the dealer, then $Sim$ delivers the message containing the commitments $CM$ and the row $row_0$ defined above.
- If the message is a “row” message from the dealer, $Sim$ delivers the message containing the row $row_i = ((R_i(1), \pi_{i,1}), \ldots, (R_i(n), \pi_{i,n}))$, with $\pi_{i,j}$ being simulated proofs of the statement $(cm_i, i, R_i(j))$ with respect to the relation $R_{open}$.
- If the message is a “column” message, then $Sim$ delivers message (“column”, $C_i(j), \pi_{i,j}$) with $\pi_{i,j}$ being a simulated proof of the statement $(cm_i, j, C_i(j))$ with respect to the relation $R_{open}$.
- If the message is a “vote” or a “done” message, $Sim$ delivers it normally.

In addition, whenever such a corrupt party $P_i$ receives a message from a corrupt $P_j$, $Sim$ delivers that message to $P_i$.

**Corrupted party’s state.** Before completing $P_i$’s corruption, $Sim$ constructs $P_i$’s state by going through the messages $P_i$ received, and computing the state it would have had after receiving those messages if it were honest. Note that by this we mean that $Sim$ constructs its state as if it received the messages described above from honest parties and the actual messages sent by corrupt parties. Finally, after the first honest party completes the HAVSS.Rec protocol, $Sim$ outputs $(S_0, S_1, \ldots, S_n)$ with the same algebraic representation described above for each $R$ value.

**Corruption of the dealer.** If at any time the dealer is corrupted, $Sim$ chooses a set of indices $I \subseteq [n]$ such that $I \cap C = \emptyset$, $|I \cup C| = t_c + 1$ (for example, the minimal indices for which this holds). It then calls the discrete log oracle on $S_i$ for every $i \in I$. Following that, $Sim$ generates polynomials $C_i$ of degree $t_c$ for every $i \in I$ in the same manner as the one described above. Finally, $Sim$ defines $S$ to be the unique bivariate polynomial of degree $t_c$ in $X$ and $t_c$ in $Y$ such that for every $i \in I \cup C S(i, Y) = C_i(Y)$. It then generates the state of all honest parties by simulating them receiving messages consistent with the dealer sampling the bivariate polynomial $S$ by defining the polynomials $R_i(X) = S(X, i)$ and $C_i(Y) = S(i, Y)$. This can be done in the same way as generating the newly corrupted parties’ state. Following that, whenever an honest party receives a message, it is processed as an honest party would in HAVSS.Share, and whenever a corrupt party receives a message it is simply delivered. At the end of the simulation, $Sim$ outputs $(S_0, \ldots, S_n)$. 

56
Simply following the description above, it is clear that the Syntax, Dealer Corruption and Queries upon Corruption properties hold. In addition, only $S_i$ elements are queries, which are generated by interpolating and evaluating the OMDL challenge. These are generated by by multiplying by a Vandermonde matrix and its inverse, which are both invertible. Therefore, the Query Independence property holds. As shown in Lemma C.8, the protocol has the Bad Event property. It is only left to show that the protocol has the Indistinguishability property. We proceed with that.

**Indistinguishability.** We will show a series of games that are indistinguishable from each other if the event $\text{Bad}$ does not take place such that the first game is an execution of the protocol, and the last is the simulated execution of the protocol. Let $A$ be some PPT adversary for the protocol.

**Game $G_0$:** This game is a normal execution of the protocol given that the event $\text{Bad}$ does not take place with the adversary $A$ controlling the corrupt parties, and all other parties acting honestly.

**Game $G_1$:** This game is identical to $G_1$, except all parties abort if the event $\text{Bad}$ ever takes place. As shown in the Bad Event property, the probability of the parties ever aborting is negligible. Further, this event is efficiently detectable. It is possible to check whether the event $\text{Bad}_{\text{open}}$ took place by simply checking if at any point a verifying “column” message was received that is inconsistent with an honest dealer’s sampled polynomial $S$. In addition, it is possible to check whether the event $\text{Bad}_{\text{open}}$ took place by computing the same Vandermonde rows described in the Bad Event property, computing the linear transformation from the first $t_e$ columns of these rows to the final column, and then computing the same transformation in the exponent on $g_0, \ldots, g_{t_e-1}$. The matrix is not invertible if and only if the result is $g_{t_e}$, in which case all honest parties can abort.

**Game $G_2$:** This game is identical to $G_1$, all proofs of knowledge computed by honest parties are simulated instead of being computed honestly. $G_1$ and $G_2$ are indistinguishable because of the zero-knowledge property of the proofs of knowledge.

**Game $G_3$:** This game is identical to $G_2$, except that if the dealer is honest, it sends $t_e + 1$ uniformly sampled commitments $c_m$ and uniform values $S_i$ in its “commits” message. Additionally, whenever an honest party $P_i$ gets corrupted, two polynomials $C_i, R_i$ of degrees $t_e$ and $t_e$ are chosen for it uniformly under the condition that $C_i(j) = R_i(i)$ and $C_i(i) = R_i(j)$ for every corrupt $P_j$ and under the condition that $S_i = g^{C_i(0)}$ and that $c_m = g^{\sum_{k=0}^{t_e} \delta_{i,k} a_k}$, where $C_i(Y) = \sum_{i=0}^{k} a_k Y^k$ and $g_0 = g^{\delta_k}$ for every $k$. Then, whenever $P_i$ receives a “row” message from the dealer, it contains the values $((R_i(j), \pi_{i,j})))_{j \in [n]}$ with $\pi_{i,j}$ being simulated proofs. Similarly, whenever it receives a “column” message from $P_j$, it contains the values $C_i(j), \pi_{i,j}$ with $\pi_{i,j}$ being a simulated proof. If at any point the dealer is corrupted, the same process takes place, but $C_i$ polynomials are sampled for $t_e + 1 - |C|$ additional parties as well. A polynomial $S$ is defined to be the bivariate polynomial of degrees $t_e$ in $X$ and $t_e$ in $Y$ consistent with these $C_i$ polynomials (i.e., $S(i, Y) = C_i(Y)$ for every such $i$), and the dealer’s view is generated as if it originally sampled $S$. In addition, all honest parties act as if messages that they received from the dealer are consistent with $S$, as well as messages from other honest parties. From this point on, honest parties continue acting as they would normally in the protocol.

First we will show that the adversary’s view is identical in $G_2$ and $G_3$. In both games the dealer does indeed uniformly sample a bivariate polynomial $S(X, Y)$ of degree $t_e$ in $X$ and $t_e$ in $Y$. As long as the event $\text{Bad}$ does not take place, all honest parties only receive points and proofs consistent with $S$, and send consistent messages to corrupt parties. This means that the corrupt parties’ views consist of random values sent in the dealer’s “commits” message, and of values on the polynomials $C_i(Y) = S(i, Y), R_i(X) = S(X, i)$. These are simply uniformly sampled polynomials which are consistent with all other $C_j, R_j$ polynomials and with the broadcasted $S_i$ values. That is, $C_i(j) = R_j(i)$ and $C_i(i) = R_j(j)$ and $S_i = g^{C_i(0)}$. Since the adversary only has access to the polynomials $C_j, R_j$ held by corrupt parties $P_j$, all it sees are $C_j$ and $R_j$ polynomials consistent with each other and with the $S_i$ values in both $G_2$ and $G_3$. If at any point the dealer is corrupted, then its view should be consistent with that same $S$, and all honest parties simply act normally under the condition that messages sent up until that point are consistent with $S$.

In order complete the proof, it is now left to argue that the view the adversary has in $G_3$ is identical to the view it has while interacting with $\text{Sim}$. The “commits” message is indeed generated by using the uniform OMDL instance $g^{r_1}, \ldots, g^{r_{t_e+1}}$ as $S_1, \ldots, S_{t_e+1}$ and uniformly generating commitments. Following that, whenever a party $P_i$ is corrupted, two polynomials $C_i, R_i$ are defined for it. These polynomials are generated in a way consistent with $S_i$ and with the other corrupted parties’ $C_j, R_j$.
polynomials All messages sent to \( P \) then contain values consistent with \( C_i \) and \( R_i \) and with simulated proofs. If the dealer is corrupted, then \( t_r + 1 - |C| \) polynomials \( C_i \) are defined. These polynomials are used to define \( S \) and the dealer’s and honest parties’ views are generated in a way consistent with \( S \), after which they continue acting honestly. This means that it is enough to show that the distribution of the \( C_i \) and \( R_i \) polynomials in \( G_3 \) is identical to the distribution while interacting with \( \text{Sim} \). Note that the \( R_i \) polynomials are simply uniformly sampled by \( \text{Sim} \) under the condition that they are consistent with \( C_j \) for every \( j \in C \). This is the same distribution as the one in \( G_3 \).

The proof for the distribution of the \( C_i \) polynomials is a little more intricate. Given that the event \( \text{Bad} \) does not take place, \( M \) is invertible at any point throughout the protocol. This means that \( M \) defines a bijection between coefficients \( a = (a_{i,0}, \ldots, a_{i,t_c})^T \) and evaluations \( v = (r_i, R_{j_1}(i), \ldots, R_{j_k}(i))^T \) such that \( Ma = v \) and \( a = M^{-1}v \). By construction, \( r_i \) is the discrete log of \( cm_i \). In addition, the set \( I \) always includes the index 0, and \( R_0(i) \) is defined to be the discrete log of \( S_i \). Finally, for every \( j_k \in C \), \( R_{j_k}(i) \) is the evaluation of \( j_k \)'s \( R \) polynomial at \( i \). The rest of the \( R_{j_k}(i) \) values are sampled uniformly. Therefore, because \( M^{-1} \) is a bijection, the coefficient vector \( a \) is sampled uniformly under the condition that the resulting polynomial \( C_i(Y) = \sum_{k=0}^{t_c} a_{i,k}Y^k \) has \( C_i(j) = R_j(i) \) for every \( j \in C \), \( S_i = g^{C_i(0)} \), and \( r_i = \sum_{k=0}^{t_c} \delta_k a_{i,k} \). In other words, the view generated when interacting with \( \text{Sim} \) is identical to the view in game \( G_3 \). \( \square \)