Jackpot: Non-Interactive Aggregatable Lotteries

Nils Fleischhacker [1] Mathias Hall-Andersen [*2] Mark Simkin [3]
Benedikt Wagner [† 3]

August 30, 2024

[1] Ruhr University Bochum
mail@nilsfleischhacker.de
[2] ZkSecurity
mathias@zksecurity.xyz
[3] Ethereum Foundation
{mark.simkin,benedikt.wagner}@ethereum.org

Abstract

In proof-of-stake blockchains, liveness is ensured by repeatedly selecting random groups of parties as leaders, who are then in charge of proposing new blocks and driving consensus forward. The lotteries that elect those leaders need to ensure that adversarial parties are not elected disproportionately often and that an adversary can not tell who was elected before those parties decide to speak, as this would potentially allow for denial-of-service attacks. Whenever an elected party speaks, it needs to provide a winning lottery ticket, which proves that the party did indeed win the lottery. Current solutions require all published winning tickets to be stored individually on-chain, which introduces undesirable storage overheads.

In this work, we introduce *non-interactive aggregatable lotteries* and show how these can be constructed efficiently. Our lotteries provide the same security guarantees as previous lottery constructions, but additionally allow any third party to take a set of published winning tickets and aggregate them into one short digest. We provide a formal model of our new primitive in the universal composability framework.

As one of our technical contributions, which may be of independent interest, we introduce aggregatable vector commitments with simulation-extractability and present a concretely efficient construction thereof in the algebraic group model in the presence of a random oracle. We show how these commitments can be used to construct non-interactive aggregatable lotteries. We have implemented our construction, called *Jackpot*, and provide benchmarks that underline its concrete efficiency.

Keywords: Lotteries, Aggregation, Vector Commitments, Simulation-Extractability, KZG Commitments

[*] This work was done while this author was at Aarhus University. Funded by the Concordium Foundation.

[†] This work was done while the author was at CISPA Helmholtz Center for Information Security. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 507237585.

Contents

1 Introduction **3**
1.1 Our Contribution . . . 3
1.2 Related Work . . . 4
1.3 Technical Overview . . . 5
1.4 Paper Organization . . . 7

2 Preliminaries **7**

3 Aggregatable Vector Commitments **8**
3.1 Syntax of Our Vector Commitments . . . 8
3.2 Simulation-Extractability . . . 9
3.3 Modularizing Simulation-Extractability . . . 10
3.4 Simulation-Extractable Vector Commitments from KZG . . . 16

4 Aggregatable Lotteries **26**
4.1 Definition of Aggregatable Lotteries . . . 26
4.2 Our Construction . . . 27

5 Discussion and Efficiency **33**
5.1 Practical Considerations . . . 33
5.2 Efficiency Evaluation . . . 33

A UC Proof of Our Lottery **39**

1 Introduction

Blockchains rely on lottery mechanisms for repeatedly electing one or multiple leaders at random from the pool of all participants. These leaders are then in charge of proposing new blocks and driving the protocol's consensus forward, thereby ensuring liveness of the blockchain. In proof-of-stake blockchains, the participants' probabilities of being elected are tied to their stake, i.e., to the amount of money they have put into the system. In Ethereum, each participant deposits a fixed amount of money to participate in the lotteries and thus everybody has the same probability of being elected. In Algorand [GHM+17], on the other hand, participants may have deposited different amounts of money and therefore have different probabilities of being elected.

In the context of proof-of-stake blockchains a lottery mechanism needs to satisfy several properties. From a security perspective, lotteries should not allow corrupt parties to be elected disproportionately often. Lotteries should hide who the elected leaders are, as an adversary could otherwise prevent the chain from growing by taking the leaders off the network right after they have been elected, but before they have had a chance to speak. Leaders should privately learn whether they won the lottery and obtain a publicly verifiable winning ticket. When a leader is ready to speak, they can attach the winning ticket to their message, so that everybody can verify that they are indeed one of the leaders.

From an efficiency perspective, lotteries should aim to minimize both the network bandwidth and storage overheads that they incur, since new leaders may need to be elected frequently among a large number of participants. In terms of bandwidth overhead we would like to minimize the amount of communication needed to run each lottery. In terms of storage overhead we would like to minimize the amount of memory needed to store all published winning tickets. Ideally, we would like the storage overhead to grow sublinearly in the number of published winning tickets.

Various constructions of lotteries schemes have already been proposed in the literature, but all of them either do not keep the lottery output secret [BD84, BK14, BZ17], require a trusted party [CHYC05, LBM20], or have storage overheads that are linear in the number of published winning tickets [GHM+17, DGKR18] per election.

1.1 Our Contribution

In this work, we introduce *non-interactive aggregatable lotteries.* In this setting we have a set of parties where each party is identified by a short verification key and holds a corresponding secret key. We assume the existence of a randomness beacon functionality which broadcasts uniformly random values to all parties in regular intervals. We will associate the randomness beacon output at time t with the t-th lottery execution.

Whenever the randomness beacon outputs a lottery seed, every party can, without interacting with the other parties, check whether they have won the current lottery. Each party will win each lottery independently with probability $1/k$ for some fixed parameter[1] k. Maliciously generated keys do not allow the adversary to increase their winning probabilities or to coordinate which corrupt parties win which lotteries at which times. The adversary is not able to determine which honest parties are winning which elections with probability noticeably better than guessing. Each winning party can locally compute a publicly verifiable proof, the winning ticket, that allows them to convince other parties that they won a lottery. Finally, and most importantly, the lotteries are aggregatable. By this we mean that all published winning tickets belonging to the same lottery execution can be compressed into one short ticket by any (possibly untrusted) third party. Given the public keys of all winning parties and the compressed lottery ticket anybody can still be convinced of the fact that each individual party won the lottery. We formally model these lotteries in the universal composability (UC) framework of Canetti [Can01].

Lotteries from Simulation-Extractable Vector Commitments. We introduce the notion of aggregatable vector commitments with a strong simulation-extractability property and show that these commitments can be used to instantiate our non-interactive aggregate lotteries. On an intuitive level, a vector commitment is said to be aggregatable if openings belonging to different commitments can be compressed into one short opening. A vector commitment is said to be simulation-extractable if it satisfies the following two properties: in security proofs, knowing a trapdoor, we can issue dummy

[1] We also show how to generalize our notion of lotteries and our constructions to the setting where parties have different winning probabilities.

commitments and later open those to arbitrary messages at arbitrary positions. Additionally we can extract the committed messages from any valid but adversarially chosen commitment. While our notion effectively requires the commitments to be "non-malleable", the openings of such a commitment scheme can still have homomorphic properties, which is of crucial importance for being able to aggregate them.

Simulation-Extractable Vector Commitments from KZG. We present a construction of such an aggregatable vector commitment with simulation-extractability proven secure in the algebraic group model (AGM) [FKL18]. Our construction is a modification of the polynomial commitment scheme of Kate, Zaverucha, and Goldberg (KZG) [KZG10] and uses the exact same trusted setup. While KZG itself is malleable and can therefore not be simulation-extractable, we show that our construction is simulation-extractable. At the same time it preserves the homomorphic properties of KZG needed for aggregation. The proof turns out to be rather involved and we present it in a modular way. We believe that our construction, our notion of simulation-extractability, and our modular proof may be of independent interest beyond their applications in this work.

Implementation and Benchmarks. To show the practicality of our construction, called *Jackpot*, we have implemented it and provide benchmarks for various parameter settings. For instance, Jackpot allows for aggregating 2048 winning tickets in less than 15 milliseconds and verifying the aggregated ticket takes less than 17 milliseconds on a regular Macbook Pro. Storing the 2048 winning tickets in aggregated form is 1228.8 times more efficient than storing a list of all tickets of a state-of-the-art lottery based on VRFs explicitly. The main bottleneck of our construction is the time it takes to generate the public keys. For generating a public key that is good for 2^{20} lotteries, i.e., for one lottery every 5 minutes for 10 years nonstop, our protocol takes around 8 seconds. The corresponding public key is 160 bytes large.

1.2 Related Work

Lotteries have appeared throughout cryptographic research in various shapes and forms. In the following we discuss a few of those research works and highlight how they differ from ours.

Lotteries Without Secrecy. The problem of allowing a group of parties to select a random set of leaders among them has already been addressed by Broder and Dolev [BD84] over 40 years ago. Their work, however, requires a large amount of interaction during each election and does not hide who is elected. The works of Bentov and Kumaresan [BK14] and of Bartoletti and Zunino [BZ17] allow parties to run financial lotteries that enjoy certain fairness properties on top of cryptocurrencies like Bitcoin or Ethereum. Here each party can deposit a coin and a random parties is elected to be the winner that obtains all deposited coins. Neither of those protocols provides any privacy guarantees and their techniques do not seem applicable to our setting.

Lotteries Without Aggregation. A lottery that satisfies all of our desired properties apart from aggregation was proposed by Gilad et al. [GHM+17]. In their construction each party is identified via a public key for a verifiable random functions (VRF) [MRV99]. The public key of party i can be viewed as a commitment to a secret random function f_i and, using their corresponding secret key, party i is able to output pairs (x, y) and prove that $y = f_i(x)$. Whenever a randomness beacon provides lseed, party i can check whether they won the corresponding lottery by computing whether $f_i(\mathsf{lseed}) < k$ for some parameter k. Since the function is random, nobody can predict whether party i wins a lottery. At the same time the verifiability property of the random function allows party i to claim the win. In subsequent work David et al. [DGKR18] properly formalized this approach and showed that the VRF actually needs to satisfy an additional property ensuring that high entropy inputs produce high entropy outputs even if the VRF keys were chosen by a malicious party.

Both works [GHM+17, DGKR18] show different ways of how their lotteries can then be used to select committees that then drive consensus forward in their respective blockchain designs. Both works would benefit from being able to aggregate lottery tickets as it would allow them to reduce their storage complexities.

Single Secret Leader Elections. A recent work by Boneh et al. [BEHG20] introduces the problem of secret leader elections and shows how it can be solved using cryptographic tools like indistinguishability obfuscation [GGH+13], threshold fully homomorphic encryption [BGG+18], or proofs of correct shuffles [BG12]. Whereas our work focuses on electing a certain number of leaders *in expectation*, they focus on computing an ordered list of an *exact* number of leaders. As their problem is significantly harder to solve, their protocols are significantly more expensive computationally and require large amounts of interaction for each lottery.

Aggregatable Vector Commitments. We mentioned above that our main technical tool is an aggregatable vector commitment that satisfies a strong form of simulation-extractability. Various aggregatable or linearly homomorphic vector commitments [LY10, CF13, PSTY13, LLNW16, TAB$^+$20, GRWZ20, LPR22, FSZ22, FHSZ22] have previously been proposed but all of these works fail to achieve simulation-extractability which is of crucial importance for our application.

On a technical level a recent result by Faonio et al. [FFK$^+$23] uses some observations similar to ours. They construct simulation-extractable succinct non-interactive arguments of knowledge (SNARKs) [Sah99, GM17]. To this end they show that the KZG polynomial commitment scheme satisfies a weak notion of simulation-extractability in the AGM. Indeed, there is no hope of proving full simulation-extractability for KZG commitments as both commitments and openings are homomorphic. Conceptually, both our work and theirs show that opening KZG at a random point chosen after the commitment is fixed makes the commitment simulation-extractable. However, we highlight three important differences: firstly, the notion that they show for the original KZG construction is tailored to their specific use-case in SNARKs. Contrary to that, we define a new scheme and show a full simulation-extractability notion that is more self-contained. Secondly, their notion states that we can extract a preimage from a KZG commitment (under certain restrictions), whereas our notion additionally guarantees that any future (aggregated) opening provided by the adversary is consistent with the extracted preimage. We can view this as a new form of binding for aggregated openings that composes with extraction. Interestingly, Faonio et al. also need a binding property, but only implicitly show it during the compilation to a SNARK. Finally, our analysis is more modular: we manage to generically separate simulation, extractability, and binding aspects.

In a concurrent and independent work, Libert [Lib24] also constructs a simulation-extractable version of KZG commitments. However, the goals of our work and Libert's work are orthogonal: Libert's construction allows to commit to multivariate polynomials and can be used in HyperPlonk [CBBZ23]. At the same time, openings can not be aggregated, which is an essential feature of our construction. Indeed, openings in Libert's construction contain a non-interactive Schnorr-style proof [Sch91]. While such proofs can be batched while they are created, it is not possible to aggregate given proofs publicly.

1.3 Technical Overview

One way of instantiating VRFs for lotteries that rely on them, e.g. [GHM$^+$17, DGKR18], is to use the unique signature scheme of Boneh, Lynn, and Shacham (BLS) [BLS01] as a verifiable unpredictable function and then apply a random oracle to the signature to make the output pseudorandom. More concretely, whenever the randomness beacon outputs the unpredictable lottery seed lseed, each participant j signs lseed (as well as potentially additional context such as their own identity) using their BLS signing key sk_j resulting in a unique signature σ_j. Participant j wins the lottery iff $\mathsf{H}(\sigma_j) < t$, where H is a random oracle and t is an appropriate threshold to achieve the desired winning probability. To prove that they won, the party presents σ_j as their winning ticket. Anyone can verify that they won by verifying the signature using the BLS public key pk_j and checking that indeed $\mathsf{H}(\sigma_j) < t$.

When considering the possibility of aggregating winning tickets, the use of BLS might seem promising at first glance. After all, BLS signatures are known to be aggregatable [BGLS03] even in the presence of rogue keys [BDN18] by computing a random linear combination of the signatures. One might thus be tempted to store this short aggregated signature σ instead of a long list of all individual signatures. Alas, this does not work. Although we could still verify that all aggregated σ_j were valid, the exact values of the individual signatures would be lost. We therefore could not recompute their individual hash values to check that all aggregated tickets were winning tickets.

The first idea to solve this dilemma is to try to avoid using the random oracle and directly look for a VRF with nice linearity properties. Specifically, let $(\mathsf{pk}_j, \mathsf{sk}_j)$ be key pairs of a VRF and let $y_j = \mathsf{VRF}(\mathsf{sk}_j, x)$. Further, let τ_j be proofs of the former equality. Then, we want that the following holds for arbitrary weights ξ_j:

$$\mathsf{VRF.Ver}\Big(\sum_{j=1}^{n} \xi_j \mathsf{pk}_j, x, \sum_{j=1}^{n} \xi_j y_j, \sum_{j=1}^{n} \xi_j \tau_j\Big) = 1 \qquad (1)$$

The ith round of the lottery could now proceed as follows: given lseed, derive per party challenges x_j. Party j wins the lottery iff $\mathsf{VRF}(\mathsf{sk}_j, (i, \mathsf{lseed})) = x_j$. The corresponding winning ticket is the proof τ_j. Using the linearity of the VRF, we could aggregate the proofs by computing a random linear combination of the winning tickets and weights $(\xi_1, \ldots, \xi_n)$, which are obtained by hashing the set of public keys. The aggregated ticket $\tau = \sum_{i=1}^{n} \xi_j \tau_j$ allows full verification of all proofs via Equation (1) simultaneously.

For this construction to be sensible we would, however, require a linearly homomorphic VRF with small codomain. Specifically, to achieve a winning probability of $1/k$, the VRF needs a codomain of size exactly k. There are currently no known constructions of such VRFs for usefully small values of k. Fortunately, we can still make the above approach work, if we are willing to make some concessions, namely that a public key will only be valid for a limited number T of successive lotteries. Since T can be chosen sufficiently large for practical purposes and because we can simply generate fresh keys after T lotteries, the concession we make is rather small.

Naive Homomorphic VRFs via Vector Commitments. If we use a vector commitment to commit to a uniformly random vector $\mathbf{v} \in [k]^T$, it can in many ways be viewed as a VRF with domain $[T]$ and codomain $[k]$. The public key is now the commitment and the secret key is the vector $\mathbf{v}$ as well as the randomness used to commit. To participate in T lotteries each party j initially commits to a random vector $\mathbf{v}^{(j)} \in [k]^T$. In the ith lottery round we again derive per party challenges x_j from lseed and party j wins iff $\mathbf{v}_i^{(j)} = x_j$. Each party can *prove* that they won by revealing an opening for position i of their commitment. If the vector commitment has the required homomorphic properties of Equation (1), we can verify all openings using only the aggregated opening. Luckily for us, such linearly homomorphic vector commitments do exist, with KZG [KZG10] being the most prominent among them.

The Woes of Universal Composability. For our lottery scheme to be useful as part of more complex protocols, it is neccessary that it composes securely with itself and other protocols. To this end, we define the security of a lottery scheme in the universal composability (UC) framework [Can01]. This, however, causes issues with the proof of the construction sketched above. Namely, in the security proof the simulator would need to both equivocate commitments for honest participants and extract from commitments of corrupted participants. This implies that the vector commitment requires some kind of simulation-extractability, i.e., a guarantee that it is possible to extract preimages from any valid commitment produced by an adversary, even if the adversary was previously given equivocal commitments (from which extraction would not be possible).

Unfortunately, not only does KZG not have this property, the required simulation-extractability and the linear homomorphism described above in fact contradict each other. Let com be a valid *simulated* commitment and let τ be an opening proving that com contained x at position i. Then by the linear homomorphism $\mathsf{com}' = \mathsf{com} + \mathsf{com}$ is also a valid commitment and $\tau' = \tau + \tau$ could be used to prove that com' contained $x + x$ at position i. However, it would not be possible to extract a preimage from com'. We thus need to depart from using a regular linear homomorphism for aggregation.

Making KZG Simulation-Extractable. To get around this problem, we make the commitments non-malleable, while maintaining the linear homomorphism on the openings (and a part of the commitments). An expensive black-box way of achieving this might be to add a simulation-extractable proof of knowledge of the secret vector to the commitment. Instead, we can leverage the fact that KZG is not just a vector commitment, but a polynomial commitment. When KZG is used to commit to a vector $\mathbf{v} \in [k]^T$, we are actually committing to the polynomial f of degree $T - 1$ over a large field $\mathbb{F}$ that is uniquely defined by the points $(j, \mathbf{v}_j)$. While we have only explicitly defined f on $[T] \subset \mathbb{F}$, we can still open the commitment at any position in $\mathbb{F}$. Now, the idea is to force anyone presenting a fresh commitment to also open their commitment at a random position. If the commitment is derived from simulated commitments, then providing such an opening should not be possible. Since this is an additional opening we need to increase the degree of the polynomial to T and they will turn out that a technicality in the proof actually requires the degree to be $T + 1$. The actual construction of our simulation-extractable vector commitment will work as follows: to commit to a vector $\mathbf{v} \in \mathbb{F}^T$ we uniformly choose a polynomial f of degree $T + 1$ conditioned on $f(j) = \mathbf{v}_j$ for $j \in [T]$ and commit to it using a regular KZG commitment $\mathsf{com}_{\mathsf{KZG}}$. The full commitment then consists of $\mathsf{com}_{\mathsf{KZG}}$ as well as an opening of the commitment at position $\mathsf{H}(\mathsf{com}_{\mathsf{KZG}})$ where H is a random oracle mapping to $\mathbb{F}$. The idea is that whenever an adversary would derive a commitment from existing commitments, they would need to open their commitment at a new random position, which the hiding property of KZG should prevent them from doing. At the same time, aggregation of openings can still be done using a random linear combination, just as with regular

KZG. Aggregated openings can be verified given the list of commitments by verifying that each individual commitment is indeed valid and then using the linear combination of the KZG part of the commitments to verify the aggregated opening. Finally, we note that while our commitment is conceptually simple, the proof that it provides simulation-extractability is far from it.

On the Necessity of Randomness Beacons. Throughout our paper, we assume that all parties have access to a randomness beacon. It is sensible to ask how necessary this assumption is. Intuitively, we would like our lotteries to ensure that no party can predict when they will win a lottery. For this to be feasible, there needs to be a source of entropy associated with each lottery execution, which is exactly what a randomness beacon provides. From a practical perspective, assuming the existence of a randomness beacon is also not too problematic, as they are deployed and running already. In the context of Ethereum, for example, the randomness beacon is known as Randao[2].

On Simulation-Based Security. In this work, we have chosen to define aggregatable lotteries through ideal functionalities in the UC framework. An alternative approach could have been to give game-based definitions. We believe that ideal functionalities are the right approach here for two reasons. Firstly, it is not at all clear what equivalent "clean" game-based notions would look like. Designing game-based notions that, for example, ensure that the adversary does not win disproportionally often or that the winning probabilities in each lottery are independent is non-trivial and would result in complex definitions. This would then make using our primitive in other contexts more cumbersome. Secondly, we would like to guarantee that our lotteries remain secure, even if composed arbitrarily with other protocols. Ideal functionalities in the UC model provide us with this guarantee, whereas game-based notions do not in general.

Parallels with Multi-Signatures. On a conceptual level, our contribution in this work has some strong parallels to multi-signatures [Ita83, MOR01]. These allow for aggregating many individual signatures for the same message into one short aggregate signature. Using multi-signatures one can significantly reduce the on-chain storage in blockchains like Ethereum, as each block only needs to store a small value, which simultaneously vouches for many different signers having approved the block's contents. Similarly, our aggregate lotteries allow for storing a short digest, which simultaneously vouches for all elected committee members within one election. Apart from our technical realization of such lotteries, we view our conceptual idea of compressing this lotteries as one of our important contributions.

1.4 Paper Organization

The main part of the paper is organized as follows. In Section 3, we introduce syntax and game-based security notions for aggregatable vector commitments. We also present our construction of this primitive. Then, in Section 4 we define aggregatable lotteries in the UC framework, present our construction from any aggregatable vector commitment. We show UC security of our lottery assuming the vector commitment meets the game-based notions we have defined. Finally, in Section 5, we discuss practical aspects and the efficiency of our construction.

2 Preliminaries

In this section, we fix notation and recall relevant cryptographic preliminaries.

Notation. For a finite set S, writing $s \leftarrow\!\!\$\, S$ means that s is sampled uniformly at random from S. For a probabilistic algorithm $\mathcal{A}$, we write $s := \mathcal{A}(x;\rho)$ to state that $\mathcal{A}$ is run on input x with random coins ρ, and the result is assigned to the variable s. If the coins ρ are sampled uniformly at random, we write $s \leftarrow \mathcal{A}(x)$. If we write $s \in \mathcal{A}(x)$, we mean that there are random coins such that when $\mathcal{A}$ is run on input x with these random coins, it outputs s. The security parameter λ is given implicitly to all algorithms (in unary). We denote the running time of an algorithm $\mathcal{A}$ by $\mathbf{T}(\mathcal{A})$. We use standard cryptographic notions, e.g., PPT, negligible. We define $[L] := \{1,\dots,L\} \subseteq \mathbb{N}$. We let $\mathcal{B}(p)$ denote a Bernoulli distribution with $\Pr[b=1] = p$ for b sampled from $\mathcal{B}(p)$ (written as $b \leftarrow \mathcal{B}(p)$).

Pairings and Assumptions. We rely on the ℓ-DLOG assumption and the ℓ-SDH assumption [BB08, KZG10]. For this and the remainder of this paper, let PGGen be an algorithm that on input 1^λ outputs the description of prime order groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of order p, generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$, and the

[2] https://eth2book.info/capella/part2/building_blocks/randomness/

description of a pairing, i.e., a non-degenerate bilinear map $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ for which $e(g_1, g_2)$ is a generator of $\mathbb{G}_T$. That is, PGGen outputs $\mathsf{par} = (\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, p, e)$. Then, informally, the ℓ-DLOG assumption states that it is hard to output α given $(g_1^{\alpha^i})_{i=1}^{\ell}, g_2^{\alpha}$ for a random $\alpha \leftarrow\!\!\$\, \mathbb{Z}_p$, and the ℓ-SDH assumption states that it is hard to output $(c, g_1^{1/(\alpha+c)})$ for some c on the same input. Clearly, ℓ-DLOG is implied by ℓ-SDH.

Definition 1 (ℓ-DLOG Assumption)**.** We say that the ℓ-DLOG assumption holds relative to PGGen, if for any PPT algorithm $\mathcal{A}$, the following advantage is negligible:

$$\mathsf{Adv}^{\ell\text{-}\mathsf{DLOG}}_{\mathcal{A},\mathsf{PGGen}}(\lambda) := \Pr\left[\mathcal{A}(\mathsf{par}, \mathsf{In}) = \alpha \;\middle|\; \begin{array}{l} \mathsf{par} \leftarrow \mathsf{PGGen}(1^{\lambda}), \\ \alpha \leftarrow\!\!\$\, \mathbb{Z}_p,\ \mathsf{In} := ((g_1^{\alpha^i})_{i=1}^{\ell}, g_2^{\alpha}) \end{array}\right].$$

Definition 2 (ℓ-SDH Assumption)**.** We say that the ℓ-SDH assumption holds relative to PGGen, if for any PPT algorithm $\mathcal{A}$, the following advantage is negligible:

$$\mathsf{Adv}^{\ell\text{-}\mathsf{SDH}}_{\mathcal{A},\mathsf{PGGen}}(\lambda) := \Pr\left[\begin{array}{l} \exists c \in \mathbb{Z}_q \setminus \{-\alpha\} : \\ \mathcal{A}(\mathsf{par}, \mathsf{In}) = \left(c, g_1^{1/(\alpha+c)}\right) \end{array} \;\middle|\; \begin{array}{l} \mathsf{par} \leftarrow \mathsf{PGGen}(1^{\lambda}), \\ \alpha \leftarrow\!\!\$\, \mathbb{Z}_p, \\ \mathsf{In} := ((g_1^{\alpha^i})_{i=1}^{\ell}, g_2^{\alpha}) \end{array}\right].$$

Universal Composability. We define an ideal functionality for aggregatable lotteries and prove security of our construction in the universal composability (UC) framework [Can01] in the presence of static corruptions. Our construction relies on synchronous broadcast and a synchronous randomness beacon, for which we provide ideal functionalities later. When specifying functionalities and simulators, we write $\mathsf{msg} \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{port}$ to denote the event of receiving the message msg on the (possibly emulated) port port. Correspondingly, we use $\mathsf{port} \overset{\mathsf{send}}{\hookleftarrow} \mathsf{msg}$ to denote the sending of the message msg on the (possibly emulated) port port.

Random Oracle Model. For some of our proofs, we use the (programmable) random oracle model (ROM) [BR93]. To recall, in the ROM, hash functions are modeled by oracles implementing perfectly random functions via lazy sampling. For our UC proof, we use the standard ROM, which is sometimes known as the local ROM as opposed to the global ROM [CDG+18].

Algebraic Group Model. For some of our proofs and extractors, we leverage the algebraic group model (AGM) [FKL18]. In this model, we only consider so called algebraic algorithms. This means that whenever such an algorithm outputs a group element Y in some cyclic group $\mathbb{G}$ of prime order p, it also outputs a so called algebraic representation, which is a vector $(c_1, \dots, c_k) \in \mathbb{Z}_p^k$ such that $Y = \prod_{i=1}^{k} X_i^{c_i}$. Here, $X_1, \dots, X_k$ are all group elements that the algorithm received so far. We emphasize that we analyze the game-based security of some of our building blocks in the AGM, and then use this security in a black-box manner for our UC proof.

3 Aggregatable Vector Commitments

In this section, we define and instantiate a special class of vector commitments that we will use to construct aggregatable lotteries.

3.1 Syntax of Our Vector Commitments

A vector commitment allows a party to commit to a vector $\mathbf{m} \in \mathcal{M}^{\ell}$ over some alphabet $\mathcal{M}$, resulting in a commitment com. Later, the committer can open com at any position $i \in [\ell]$ by revealing $\mathbf{m}_i$ and a corresponding opening (proof) τ. One can then publicly verify the pair $(\mathbf{m}_i, \tau)$ with respect to com and i. Our definition of vector commitments is special in two ways. First, it should be possible to publicly aggregate several openings for different commitments with respect to the same position. Precisely, we require the existence of an algorithm $\mathsf{Aggregate}$ that takes a list of L openings $\tau_j, j \in [L]$ (all for the same position $i \in [\ell]$) and outputs an aggregated opening τ. One can then verify τ with respect to a list of L commitments. For non-triviality, the aggregated τ should ideally be as large as one single τ_j. Note that a similar aggregation feature for openings of different commitments has been defined in [GRWZ20]. The second non-standard part of our definition is that we explicitly model an algorithm VerCom that verifies whether commitments (not openings) are well-formed. For our security notions, this will be convenient.

Definition 3 (Vector Commitment Scheme)**.** A vector commitment scheme (VC) is a tuple $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$ of PPT algorithms with the following syntax:

- $\mathsf{Setup}(1^\lambda, 1^\ell) \to \mathsf{ck}$ takes as input the security parameter and a message length ℓ, and outputs a commitment key ck. We assume that ck specifies a message alphabet $\mathcal{M}$, opening space $\mathcal{T}$, and commitment space $\mathcal{C}$.
- $\mathsf{Com}(\mathsf{ck}, \mathbf{m}) \to (\mathsf{com}, St)$ takes as input a commitment key ck and a vector $\mathbf{m} \in \mathcal{M}^\ell$, and outputs a commitment $\mathsf{com} \in \mathcal{C}$ and a state St.
- $\mathsf{VerCom}(\mathsf{ck}, \mathsf{com}) \to b$ is deterministic, takes as input a commitment key ck and a commitment com, and outputs a bit $b \in \{0, 1\}$.
- $\mathsf{Open}(\mathsf{ck}, St, i) \to \tau$ takes as input a commitment key ck, a state St, and an index $i \in [\ell]$, and outputs an opening $\tau \in \mathcal{T}$.
- $\mathsf{Aggregate}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, (\tau_j)_{j=1}^L) \to \tau$ is deterministic, takes as input a commitment key ck, an index $i \in [\ell]$, a list of commitments $\mathsf{com}_j \in \mathcal{C}$, a list of symbols $m_j \in \mathcal{M}$, and a list of openings $\tau_j \in \mathcal{T}$, and outputs an opening $\tau \in \mathcal{T}$.
- $\mathsf{Ver}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau) \to b$ is deterministic, takes as input a commitment key ck, an index $i \in [\ell]$, a list of commitments $\mathsf{com}_j \in \mathcal{C}$, a list of symbols $m_j \in \mathcal{M}$, and an opening $\tau \in \mathcal{T}$, and outputs a bit $b \in \{0, 1\}$.

Further, we require that the following properties holds:

1. **Commitment Completeness.** For any $\ell \in \mathbb{N}$, any $\mathsf{ck} \in \mathsf{Setup}(1^\lambda, 1^\ell)$, and any $\mathbf{m} \in \mathcal{M}^\ell$, we have

$$\Pr\left[\mathsf{VerCom}(\mathsf{ck}, \mathsf{com}) = 1 \mid (\mathsf{com}, St) \leftarrow \mathsf{Com}(\mathsf{ck}, \mathbf{m})\right] = 1.$$

2. **Opening Completeness.** For any $\ell \in \mathbb{N}$, any $\mathsf{ck} \in \mathsf{Setup}(1^\lambda, 1^\ell)$, any $\mathbf{m} \in \mathcal{M}^\ell$, and any $i \in [\ell]$, we have

$$\Pr\left[\mathsf{Ver}(\mathsf{ck}, i, \mathsf{com}, \mathbf{m}_i, \tau) = 1 \;\middle|\; \begin{array}{c} (\mathsf{com}, St) \leftarrow \mathsf{Com}(\mathsf{ck}, \mathbf{m}), \\ \tau \leftarrow \mathsf{Open}(\mathsf{ck}, St, j) \end{array}\right] = 1.$$

3. **Aggregation Completeness.** For any $\ell \in \mathbb{N}$, any $\mathsf{ck} \in \mathsf{Setup}(1^\lambda, 1^\ell)$, any $L \in \mathbb{N}$, any index $i \in [\ell]$, any list $(m_j)_{j=1}^L \in \mathcal{M}^L$, any list $(\mathsf{com}_j)_{j=1}^L \in \mathcal{C}^L$, any list $(\tau_j)_{j=1}^L \in \mathcal{T}^L$, we have

$$\begin{aligned} & \forall j \in [L] : \mathsf{Ver}(\mathsf{ck}, i, \mathsf{com}_j, m_j, \tau_j) = 1 \land \tau = \mathsf{Aggregate}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, (\tau_j)_{j=1}^L) \\ \Longrightarrow \quad & \mathsf{Ver}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau) = 1. \end{aligned}$$

3.2 Simulation-Extractability

We define a strong simulation-extractability property for vector commitments. This property captures all properties that we will need for our UC proof, including both hiding and binding properties. Beyond that, it may be interesting in itself. The notion states that no adversary can distinguish between two games in which it is running, where one game models the real world, and the other game models an ideal world. The first property that our notion models is a strong form of *hiding*. Namely, we require that there is a way to set up the commitment key with a trapdoor, and this trapdoor allows a simulator to compute commitments without knowing the message, and later open these commitments at arbitrary positions to arbitrary symbols. This is modeled in our notion as follows. In the real world game, the adversary gets an honest commitment key. It also gets access to an oracle GetCom that outputs honestly computed commitments to messages of the adversary's choice. Another oracle GetOp provides openings for these commitments when the adversary asks for them. In the ideal world game, the commitment key is set up with a trapdoor and both commitments and openings are simulated. In addition to this hiding property, our notion models a strong form of *binding*. Namely, the adversary gets access to oracles SubCom and SubOp that allow it to submit commitments and openings for them. While the commitments and openings are simply verified in the real world game, there are additional checks in the ideal world game.

Concretely, when the adversary submits a commitment com that is not output by GETCOM, the game not only verifies it, but also tries to extract a preimage $(\mathbf{m}, \varphi)$ from it, such that $\mathbf{m}$ with randomness φ commits to com. If this extraction fails but com verifies, SUBCOM outputs 0 in the ideal world game, whereas it would output 1 in the real world game. In other words, indistinguishability of the games ensures that we can always extract preimages of commitments. In addition, our notion ensures that openings are consistent: (1) whatever we extracted in SUBCOM is consistent with any valid opening that the adversary submits later, and (2) if the adversary opens a commitment output by GETCOM($\mathbf{m}$) at position i, then (2a) it opens to the respective $\mathbf{m}_i$, and (2b) it queried GETOP for this commitment at position i before. Our notion ensures this because in the ideal game, SUBOP outputs 0 if one of the inconsistencies (1, 2a, 2b) occurs, whereas in the real game the output of SUBOP only depends on whether the opening verifies.

Definition 4 (Simulation-Extractability of VC)**.** Consider a vector commitment scheme $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$. For any algorithm $\mathcal{A}$, any $\ell \in \mathbb{N}$, any algorithm Ext, and any triple of algorithms $\mathsf{Sim} = (\mathsf{TSetup}, \mathsf{TCom}, \mathsf{TOpen})$, consider the game ℓ-$\mathbf{SIM\text{-}EXT}^{\mathcal{A}}_{\mathsf{VC},0}(\lambda)$ and the game ℓ-$\mathbf{SIM\text{-}EXT}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},\mathsf{Ext},1}(\lambda)$ defined in Figure 1. We say that VC is simulation-extractable, if there are PPT algorithms Ext and $\mathsf{Sim} = (\mathsf{TSetup}, \mathsf{TCom}, \mathsf{TOpen})$ such that for any polynomial $\ell \in \mathbb{N}$ and any PPT algorithm $\mathcal{A}$, the following advantage is negligible:

$$\mathsf{Adv}^{\mathsf{sim\text{-}ext}}_{\mathcal{A},\mathsf{VC},\mathsf{Sim},\mathsf{Ext},\ell}(\lambda) := \left|\Pr\left[\ell\text{-}\mathbf{SIM\text{-}EXT}^{\mathcal{A}}_{\mathsf{VC},0}(\lambda) \Rightarrow 1\right] - \Pr\left[\ell\text{-}\mathbf{SIM\text{-}EXT}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},\mathsf{Ext},1}(\lambda) \Rightarrow 1\right]\right|.$$

Then, we say VC is simulation-extractable with extractor Ext and simulator Sim.

3.3 Modularizing Simulation-Extractability

Our simulation-extractability notion is well-suited for our UC proof. However, it models several distinct properties of the vector commitment simultaneously, which renders a direct proof of simulation-extractability complicated. Thus, we define three less complex security notions and show that in combination they imply simulation-extractability. The first notion, *equivocality*, is the hiding part of our simulation-extractability notion.

Definition 5 (Equivocal VC)**.** Consider a vector commitment scheme $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$. For any algorithm $\mathcal{A}$, any $\ell \in \mathbb{N}$, and any triple of algorithms $\mathsf{Sim} = (\mathsf{TSetup}, \mathsf{TCom}, \mathsf{TOpen})$ consider the games ℓ-$\mathbf{EQUIV}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},b}(\lambda)$ for $b \in \{0,1\}$ defined in Figure 2. We say that VC is equivocal, if there are PPT algorithms $\mathsf{Sim} = (\mathsf{TSetup}, \mathsf{TCom}, \mathsf{TOpen})$ such that for any polynomial $\ell \in \mathbb{N}$ and any PPT algorithm $\mathcal{A}$, the following advantage is negligible:

$$\mathsf{Adv}^{\mathsf{equiv}}_{\mathcal{A},\mathsf{VC},\mathsf{Sim},\ell}(\lambda) := \left|\Pr\left[\ell\text{-}\mathbf{EQUIV}^{\mathcal{A}}_{\mathsf{VC},0}(\lambda) \Rightarrow 1\right] - \Pr\left[\ell\text{-}\mathbf{EQUIV}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},1}(\lambda) \Rightarrow 1\right]\right|.$$

In this case, we say that VC is equivocal with simulator Sim.

The second and third notion focus on binding. Namely, the notion of *augmented extractability* states that we can extract preimages of commitments from any opening that the adversary outputs, even if it sees some honest commitments and openings. Notably, we do not allow the extractor to inspect the internal state of the oracles that output these honest commitments and openings, which is crucial for making this notion compose with equivocality.

Definition 6 (Augmented Extractability of VC)**.** Let $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$ denote a vector commitment scheme. For any algorithm $\mathcal{A}$, any algorithm Ext, any $\ell \in \mathbb{N}$, consider the game ℓ-$\mathbf{AUG\text{-}EXT}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Ext}}(\lambda)$ defined in Figure 3. We say that VC satisfies augmented extractability, if there is a PPT algorithm Ext such that for any polynomial $\ell \in \mathbb{N}$ and any PPT algorithm $\mathcal{A}$, the following advantage is negligible:

$$\mathsf{Adv}^{\mathsf{aug\text{-}ext}}_{\mathcal{A},\mathsf{VC},\mathsf{Ext},\ell}(\lambda) := \Pr\left[\ell\text{-}\mathbf{AUG\text{-}EXT}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Ext}}(\lambda) \Rightarrow 1\right].$$

In this case, we say that VC satisfies augmented extractability with extractor Ext.

```
Game $\ell$-SIM-EXT$^{\mathcal{A}}_{\mathsf{VC},0}(\lambda)$
$c := 0,\ \mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell)$
$\mathrm{O}_G := (\textsc{GetCom}_0, \textsc{GetOp}_0)$
$\mathrm{O}_S := (\textsc{SubCom}_0, \textsc{SubOp}_0)$
return $\mathcal{A}^{\mathrm{O}_G, \mathrm{O}_S}(\mathsf{ck})$

Oracle $\textsc{GetCom}_0(\mathbf{m})$
$c := c + 1,\ \ \mathsf{Msgs}[c] := \mathbf{m}$
$(\mathsf{com}, St) \leftarrow \mathsf{Com}(\mathsf{ck}, \mathbf{m})$
$\mathsf{Coms}[c] := \mathsf{com},\ \ \mathsf{St}[c] := St$
$\mathsf{Ops}[c] := \emptyset$
return $\mathsf{com}$

Oracle $\textsc{GetOp}_0(k, i)$
if $\mathsf{Coms}[k] = \bot$ : return $\bot$
if $i \in \mathsf{Ops}[k]$ : return $\bot$
$\mathsf{Ops}[k] := \mathsf{Ops}[k] \cup \{i\}$
$\tau \leftarrow \mathsf{Open}(\mathsf{ck}, \mathsf{St}[k], i)$
return $\tau$

Oracle $\textsc{SubCom}_0(\mathsf{com})$
if $\exists k$ s.t. $\mathsf{Coms}[k] = \mathsf{com}$ :
    return $0$
if $\mathsf{VerCom}(\mathsf{ck}, \mathsf{com}) = 0$ :
    return $0$
$\mathsf{Sub} := \mathsf{Sub} \cup \{\mathsf{com}\}$
return $1$
```

```
Game $\ell$-SIM-EXT$^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},\mathsf{Ext},1}(\lambda)$
$c := 0,\ (\mathsf{ck}, \mathsf{td}) \leftarrow \mathsf{TSetup}(1^\lambda, 1^\ell)$
$\mathrm{O}_G := (\textsc{GetCom}_1, \textsc{GetOp}_1)$
$\mathrm{O}_S := (\textsc{SubCom}_1, \textsc{SubOp}_1)$
return $\mathcal{A}^{\mathrm{O}_G, \mathrm{O}_S}(\mathsf{ck})$

Oracle $\textsc{GetCom}_1(\mathbf{m})$
$c := c + 1,\ \ \mathsf{Msgs}[c] := \mathbf{m}$
$(\mathsf{com}, St) \leftarrow \mathsf{TCom}(\mathsf{ck})$
$\mathsf{Coms}[c] := \mathsf{com},\ \ \mathsf{St}[c] := St$
$\mathsf{Ops}[c] := \emptyset$
return $\mathsf{com}$

Oracle $\textsc{GetOp}_1(k, i)$
if $\mathsf{Coms}[k] = \bot$ : return $\bot$
if $i \in \mathsf{Ops}[k]$ : return $\bot$
$\mathsf{Ops}[k] := \mathsf{Ops}[k] \cup \{i\}$
$\tau \leftarrow \mathsf{TOpen}(\mathsf{td}, \mathsf{St}[k], i, \mathsf{Msgs}[k]_i)$
return $\tau$

Oracle $\textsc{SubCom}_1(\mathsf{com})$
if $\exists k$ s.t. $\mathsf{Coms}[k] = \mathsf{com}$ :
    return $0$
if $\mathsf{VerCom}(\mathsf{ck}, \mathsf{com}) = 0$ :
    return $0$
$(\mathbf{m}, \varphi) \leftarrow \mathsf{Ext}(\mathsf{td}, \mathsf{com})$
$(\mathsf{com}', St) := \mathsf{Com}(\mathsf{ck}, \mathbf{m}; \varphi)$
if $\mathsf{com}' \neq \mathsf{com}$ : return $0$
$\mathsf{MsgsExt}[\mathsf{com}] := \mathbf{m}$
$\mathsf{Sub} := \mathsf{Sub} \cup \{\mathsf{com}\}$
return $1$
```

```
Oracle $\textsc{SubOp}_b(i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau)$
if $b = 1$ : for $j \in [L]$ :
    if $\mathsf{com}_j \in \mathsf{Sub} \wedge m_j \neq \mathsf{MsgsExt}[\mathsf{com}]_i$ : return $0$
    if $\exists k$ s.t. $\mathsf{com}_j = \mathsf{Coms}[k] \wedge i \in \mathsf{Ops}[k] \wedge m_j \neq \mathsf{Msgs}[k]_i$ : return $0$
    if $\exists k$ s.t. $\mathsf{com}_j = \mathsf{Coms}[k] \wedge i \notin \mathsf{Ops}[k]$ : return $0$
return $\mathsf{Ver}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau)$
```

Figure 1: The simulation-extractability games ℓ-**SIM-EXT** for a vector commitment $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$, an adversary $\mathcal{A}$, an extractor Ext, and a simulator $\mathsf{Sim} = (\mathsf{TSetup}, \mathsf{TCom}, \mathsf{TOpen})$. In the random oracle model, Ext gets as additional input the list of random oracle queries of $\mathcal{A}$. In the algebraic group model, Ext gets as additional input the algebraic representation of all group elements contained in the commitment com submitted by $\mathcal{A}$.

```
Game $\ell$-EQUIV$^{\mathcal{A}}_{\mathsf{VC},0}(\lambda)$
$c := 0$
$\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell)$
return $\mathcal{A}^{\textsc{GetCom}_0, \textsc{GetOp}_0}(\mathsf{ck})$
```

```
Game $\ell$-EQUIV$^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},1}(\lambda)$
$c := 0$
$(\mathsf{ck}, \mathsf{td}) \leftarrow \mathsf{TSetup}(1^\lambda, 1^\ell)$
return $\mathcal{A}^{\textsc{GetCom}_1, \textsc{GetOp}_1}(\mathsf{ck})$
```

Figure 2: The equivocality games ℓ-**EQUIV** for a vector commitment $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$, an adversary $\mathcal{A}$, and a simulator $\mathsf{Sim} = (\mathsf{TSetup}, \mathsf{TCom}, \mathsf{TOpen})$. Oracles $\textsc{GetCom}_b$ and $\textsc{GetOp}_b$ are as in Figure 1.

```
Game $\ell$-AUG-EXT$^{\mathcal{A}}_{\mathsf{VC},\mathsf{Ext}}(\lambda)$
$\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell)$
$\mathsf{com} \leftarrow \mathcal{A}^{\textsc{GetCom}_0,\textsc{GetOp}_0}(\mathsf{ck})$
$(\mathbf{m}, \varphi) \leftarrow \mathsf{Ext}(\mathsf{ck}, \mathsf{com}), \quad (\mathsf{com}', St) := \mathsf{Com}(\mathsf{ck}, \mathbf{m}; \varphi)$
if $\nexists k$ s.t. $\mathsf{Coms}[k] = \mathsf{com} \wedge \mathsf{VerCom}(\mathsf{ck}, \mathsf{com}) = 1 \wedge \mathsf{com} \neq \mathsf{com}'$ : return 1
return 0
```

Figure 3: The augmented extractability game ℓ-**AUG-EXT** for a vector commitment $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$, an extractor Ext, and an adversary $\mathcal{A}$. Oracles $\textsc{GetCom}_0$ and $\textsc{GetOp}_0$ are as in Figure 1. In the random oracle model, Ext gets as additional input the list of random oracle queries of $\mathcal{A}$. In the algebraic group model, Ext gets as additional input the algebraic representation of all group elements contained in the commitment com submitted by $\mathcal{A}$. Notably, Ext does not share any internal state with the rest of the game.

```
Game $\ell$-A-POS-BIND$^{\mathcal{A}}_{\mathsf{VC}}(\lambda)$
$\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell)$
$(\mathbf{m}, \varphi, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau) \leftarrow \mathcal{A}(\mathsf{ck})$
$(\mathsf{com}, St) := \mathsf{Com}(\mathsf{ck}, \mathbf{m}; \varphi)$
if $\mathsf{Ver}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau) = 0$ : return 0
if $\exists j^* \in [L]$ s.t. $\mathsf{com}_{j^*} = \mathsf{com} \wedge m_{j^*} \neq \mathbf{m}_i$ : return 1
return 0
```

Figure 4: The aggregation position-binding game ℓ-**A-POS-BIND** for a vector commitment $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$ and an adversary $\mathcal{A}$.

Augmented extractability states that we can extract some preimage of adversarially submitted commitments. It does not state that what we extract is consistent with whatever the adversary opens later. For that, we define *aggregation position-binding*. Intuitively, we want that any two lists of commitments and openings that an adversary outputs are consistent, i.e., if they share a commitment, then the opened symbols for that commitment are the same. It turns out that we can further simplify this by assuming that one of the lists contains exactly one honestly computed commitment (with potentially biased randomness).

Definition 7 (Aggregation Position-Binding of VC)**.** Let $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$ be a vector commitment scheme. For any algorithm $\mathcal{A}$ and any $\ell \in \mathbb{N}$, consider the game ℓ-**A-POS-BIND**$^{\mathcal{A}}_{\mathsf{VC}}(\lambda)$ defined in Figure 4. We say that VC is aggregation position-binding, if for any polynomial $\ell \in \mathbb{N}$ and any PPT algorithm $\mathcal{A}$, the following advantage is negligible:

$$\mathsf{Adv}^{\mathsf{a\text{-}pos\text{-}bind}}_{\mathcal{A},\mathsf{VC},\ell}(\lambda) := \Pr\left[\ell\text{-}\mathbf{A\text{-}POS\text{-}BIND}^{\mathcal{A}}_{\mathsf{VC}}(\lambda) \Rightarrow 1\right].$$

Next, we show that our notions imply simulation-extractability.

Lemma 1. *Let* VC *be a vector commitment scheme such that* $|\mathcal{M}| \geq \omega(\log \lambda)$*. Assume that* VC *is equivocal with simulator* Sim*, and that it is aggregation position-binding, and satisfies augmented extractability with extractor* Ext*. Then,* VC *is simulation-extractable with extractor* Ext *and simulator* Sim*. Concretely, for any PPT algorithm* $\mathcal{A}$ *that makes at most* Q *queries to oracle* $\textsc{GetCom}$ *there are PPT algorithms* $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$ *with* $\mathbf{T}(\mathcal{B}_i) \approx \mathbf{T}(\mathcal{A})$ *for* $i \in \{1, 2, 3\}$*, and*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{sim\text{-}ext}}_{\mathcal{A},\mathsf{VC},\mathsf{Sim},\mathsf{Ext},\ell}(\lambda) \leq \frac{\ell Q}{|\mathcal{M}|} &+ (\ell Q + 1) \cdot \mathsf{Adv}^{\mathsf{a\text{-}pos\text{-}bind}}_{\mathcal{B}_2,\mathsf{VC},\ell}(\lambda) \\ &+ \mathsf{Adv}^{\mathsf{aug\text{-}ext}}_{\mathcal{B}_1,\mathsf{VC},\mathsf{Ext},\ell}(\lambda) + (2\ell Q + 1) \cdot \mathsf{Adv}^{\mathsf{equiv}}_{\mathcal{B}_3,\mathsf{VC},\mathsf{Sim},\ell}(\lambda).\end{aligned}$$

Moreover, if $\mathcal{A}$ *is algebraic and every algorithm of* VC *is algebraic, then so are the* $\mathcal{B}_i$ *for* $i \in \{1, 2, 3\}$*.*

Proof. We first give a proof intuition and then provide the full proof. We need to show that the real game and the ideal game of simulation-extractability are indistinguishable. For that, we start with the real game. In a first step, we change the game by extracting from all commitments that the adversary submits via SubCom, and let the oracle return 0 if extraction does not yield a valid preimage. The games are indistinguishable by augmented extractability. Note that now oracle SubCom is as in the ideal game. In the second step, we change oracle SubOp to be as in the ideal world game as well. The adversary can only distinguish this, if one of the three conditions on which the implementations of oracle SubOp in the real game and the ideal game differ occurs. It turns out that we can bound this probability using aggregation position-binding. Now, it remains to change the implementation of oracles GetCom and GetOp to be as in the ideal game. This change can be done using equivocality of the commitment. Here, it is essential that we defined our extractor in an appropriate way, see Figure 3: the extractor does not rely on any internals of the oracles GetCom and GetOp and just sees their input and output behavior. Otherwise, a reduction for this final change would not be able to run the extractor correctly. Next, we turn to the formal proof. We present a sequence of games, where the initial game $\mathbf{G}_0$ is ℓ-$\mathbf{SIM\text{-}EXT}^{\mathcal{A}}_{\mathsf{VC},0}(\lambda)$, and the final game $\mathbf{G}_3$ is ℓ-$\mathbf{SIM\text{-}EXT}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},\mathsf{Ext},1}(\lambda)$.

Game $\mathbf{G}_0$: We define $\mathbf{G}_0$ to be the game ℓ-$\mathbf{SIM\text{-}EXT}^{\mathcal{A}}_{\mathsf{VC},0}(\lambda)$. We briefly recall it. In the game, the adversary $\mathcal{A}$ is run on input $\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell)$ with access to four oracles, which are as follows:

- Oracle GetCom($\mathbf{m}$) honestly commits to message $\mathsf{m} \in \mathcal{M}^\ell$ by running $(\mathsf{com}, St) \leftarrow \mathsf{Com}(\mathsf{ck}, \mathbf{m})$. It saves St and outputs com to $\mathcal{A}$.
- Oracle GetOp(k, i) honestly opens the kth commitment output by GetCom at the ith position, given that this commitment exists. That is, it returns $\tau \leftarrow \mathsf{Open}(\mathsf{ck}, St, i)$, where St has been saved before.
- Oracle SubCom(com) verifies commitment com by running $\mathsf{VerCom}(\mathsf{ck}, \mathsf{com})$, given that it is fresh, i.e., not an output of GetCom. It returns 1 if and only if the commitment is fresh and verifies.
- Oracle SubOp($i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau$) verifies the given opening. That is, it runs and returns $\mathsf{Ver}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau)$.

Finally, the game outputs whatever $\mathcal{A}$ outputs. In the following games, we will gradually change these oracles. By definition, we have

$$\Pr\left[\ell\text{-}\mathbf{SIM\text{-}EXT}^{\mathcal{A}}_{\mathsf{VC},0}(\lambda) \Rightarrow 1\right] = \Pr\left[\mathbf{G}_0 \Rightarrow 1\right].$$

Game $\mathbf{G}_1$: This game is as $\mathbf{G}_0$, but we change oracle SubCom. Roughly, we switch it from being implemented as in ℓ-$\mathbf{SIM\text{-}EXT}^{\mathcal{A}}_{\mathsf{VC},0}(\lambda)$ to being implemented as in ℓ-$\mathbf{SIM\text{-}EXT}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},\mathsf{Ext},1}(\lambda)$. More concretely, when $\mathcal{A}$ queries SubCom(com), the oracle first checks if com is fresh and if $\mathsf{VerCom}(\mathsf{ck}, \mathsf{com}) = 1$. If not, it returns 0 as before. Otherwise, however, it now additionally runs extractor Ext, namely, it runs $(\mathbf{m}, \varphi) \leftarrow \mathsf{Ext}(\mathsf{ck}, \mathsf{com})$ and returns 0 if $\mathsf{com}' \neq \mathsf{com}$ for $(\mathsf{com}', St) := \mathsf{Com}(\mathsf{ck}, \mathbf{m}; \varphi)$. That is, it returns 0 if the extractor does not compute a valid preimage of the submitted commitment. Otherwise, it returns 1. Here, we want to remark that all group elements that $\mathcal{A}$ may get in the algebraic group model are contained in ck and the outputs of GetCom and GetOp. This is exactly as in the augmented extractability game for which Ext is originally defined, meaning that the extractor "understands" the algebraic representation submitted by $\mathcal{A}$ with com, and the game does not need to do any modification on it before running Ext. This will be important later. Note that $\mathcal{A}$'s view only changes if it submits a commitment com for which SubCom would return 1 in $\mathbf{G}_0$ but 0 in $\mathbf{G}_1$. In this case, the commitment is fresh, it verifies via algorithm VerCom, but extraction fails. We can easily bound this event using a straight-forward reduction $\mathcal{B}$ breaking augmented extractability. The reduction gets as input ck and gets access to oracles GetCom and GetOp. It runs $\mathcal{A}$ on input ck, while simulating GetCom and GetOp by simply forwarding to its own oracles and simulating SubOp honestly as in $\mathbf{G}_0$ and $\mathbf{G}_1$. It also simulates SubCom as in $\mathbf{G}_1$, and if the event we want to bound occurs, it outputs the submitted commitment com (including the algebraic representation) to its game and terminates. Clearly, the simulation is perfect, and $\mathcal{B}$ breaks augmented extractability if the event we want to bound occurs. We get

$$\left|\Pr\left[\mathbf{G}_0 \Rightarrow 1\right] - \Pr\left[\mathbf{G}_1 \Rightarrow 1\right]\right| \leq \mathsf{Adv}^{\mathsf{aug\text{-}ext}}_{\mathcal{B},\mathsf{VC},\mathsf{Ext},\ell}(\lambda).$$

Game $\mathbf{G}_2$: In this game, we change oracle SubOp. Namely, we implement it as in the ideal game ℓ-$\mathbf{SIM\text{-}EXT}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},\mathsf{Ext},1}(\lambda)$ from now. To recall, in $\mathbf{G}_1$ the oracle takes an input $(i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau)$ and runs and returns $\mathsf{Ver}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau)$. Now, it returns 0, if one of the following events occur for this query, whereas it returns $\mathsf{Ver}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau)$ otherwise.

- Event $\mathsf{BadOpMal}$: This event occurs, if there is a $j \in [L]$ such that (1) com_j has been submitted to SubCom before, and (2) extraction on com_j succeeded, i.e., $(\mathbf{m}, \varphi) \leftarrow \mathsf{Ext}(\mathsf{ck}, \mathsf{com}_j)$ and $\mathbf{m}$ commits to com_j with randomness $\mathbf{m}$, but (3) the submitted m_j is not the consistent with the extracted $\mathbf{m}$, i.e., $\mathbf{m}_i \neq m_j$.
- Event $\mathsf{BadOpHon}$: This event occurs, if there is a $j \in [L]$ and a k such that (1) com_j has been output as the kth commitment by GetCom($\mathbf{m}$) before, and (2) GetOp(k, i) has been called, i.e., com_j has been opened at position i by the game, but (3) the submitted m_j is not the consistent with $\mathbf{m}$, i.e., $\mathbf{m}_i \neq m_j$.
- Event $\mathsf{BadOpEarly}$: This event occurs, if there is a $j \in [L]$ and a k such that (1) com_j has been output as the kth commitment by GetCom($\mathbf{m}$) before, and (2) GetOp(k, i) has not been called before, i.e., com_j has not been opened at position i by the game before.

Note that the three events correspond to the three conditions on which the implementations of oracle SubOp in the real game and the ideal game differ. Clearly, $\mathcal{A}$'s view only changes if makes a query to SubOp for which the oracle would return 1 in $\mathbf{G}_1$ but 0 in $\mathbf{G}_2$. In other words, $\mathcal{A}$'s view only changes if one of the three events occurs. To bound the probability, we first consider the event $\mathsf{BadOpMal} \vee \mathsf{BadOpHon}$. If this event occurs, it is easy to see that a reduction $\mathcal{B}$ can break aggregation position-binding of VC. Namely, if this happens for an input $(i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau)$, then the reduction can simply return $(\mathbf{m}, \varphi, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau)$, where in the case of $\mathsf{BadOpHon}$, φ is the randomness that has been used in oracle GetCom($\mathbf{m}$). We get

$$\Pr[\mathsf{BadOpMal} \vee \mathsf{BadOpHon}] \leq \mathsf{Adv}^{\mathsf{a\text{-}pos\text{-}bind}}_{\mathcal{B},\mathsf{VC},\ell}(\lambda).$$

Further, we claim that

$$\Pr[\mathsf{BadOpEarly}] \leq \frac{\ell Q}{|\mathcal{M}|} + 2\ell Q \cdot \mathsf{Adv}^{\mathsf{equiv}}_{\mathcal{B},\mathsf{VC},\mathsf{Sim},\ell}(\lambda) + \ell Q \cdot \mathsf{Adv}^{\mathsf{a\text{-}pos\text{-}bind}}_{\mathcal{B}',\mathsf{VC},\ell}(\lambda).$$

We will show this at the end of the proof. In combination, we get

$$|\Pr[\mathbf{G}_1 \Rightarrow 1] - \Pr[\mathbf{G}_2 \Rightarrow 1]| \leq \frac{\ell Q}{|\mathcal{M}|} + 2\ell Q \cdot \mathsf{Adv}^{\mathsf{equiv}}_{\mathcal{B},\mathsf{VC},\mathsf{Sim},\ell}(\lambda) + (\ell Q + 1) \cdot \mathsf{Adv}^{\mathsf{a\text{-}pos\text{-}bind}}_{\mathcal{B}',\mathsf{VC},\ell}(\lambda).$$

Game $\mathbf{G}_3$: In this game, we change ck and the oracles GetCom and GetOp. Namely, we no longer generate $\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell)$, but instead generate it with a trapdoor via $(\mathsf{ck}, \mathsf{td}) \leftarrow \mathsf{TSetup}(1^\lambda, 1^\ell)$. In oracle GetCom($\mathbf{m}$), we compute the commitment com without using the message $\mathbf{m}$ by running $(\mathsf{com}, St) \leftarrow \mathsf{TCom}(\mathsf{ck})$. Later, when we have to open this commitment at the ith position during a query to GetOp, we compute τ by running $\tau \leftarrow \mathsf{TOpen}(\mathsf{td}, St, i, \mathbf{m}_i)$. The difference between these two games can easily be bounded by a reduction $\mathcal{B}$ against equivocality of VC, which just forwards the queries to GetCom and GetOp and there responses between the adversary and its own oracles. In the algebraic group model, what is important here is that the reduction does not need to change the algebraic representation for the extractor, as noted in $\mathbf{G}_1$, and that the extractor Ext does not rely on any internal states of oracles GetCom and GetOp. If this were the case, the reduction could not provide this internal state to Ext and thus not simulate the game. We get

$$|\Pr[\mathbf{G}_2 \Rightarrow 1] - \Pr[\mathbf{G}_3 \Rightarrow 1]| \leq \mathsf{Adv}^{\mathsf{equiv}}_{\mathcal{B},\mathsf{VC},\mathsf{Sim},\ell}(\lambda).$$

Finally, note that $\mathbf{G}_3$ is exactly ℓ-$\mathbf{SIM\text{-}EXT}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},\mathsf{Ext},1}(\lambda)$, and we get

$$\Pr[\mathbf{G}_3 \Rightarrow 1] = \Pr\left[\ell\text{-}\mathbf{SIM\text{-}EXT}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},\mathsf{Ext},1}(\lambda) \Rightarrow 1\right].$$

Bounding Event $\mathsf{BadOpEarly}$. It remains to bound the probability of $\mathsf{BadOpEarly}$ in $\mathbf{G}_1$. For that, we first introduce sub-events $\mathsf{BadOpEarly}_{k,i}$. Namely, for every $k \in [Q]$ and every $i \in [\ell]$, we define $\mathsf{BadOpEarly}_{k,i}$ as to be the event that $\mathsf{BadOpEarly}$ occurs on a query $(i, (\mathsf{com}_j)_{j=1}^{L}, (m_j)_{j=1}^{L}, \tau)$ to $\textsc{SubOp}$ and the com_j triggering $\mathsf{BadOpEarly}$ is the kth commitment output by $\textsc{GetCom}$. With that, a union bound gives us

$$\Pr\left[\mathsf{BadOpEarly}\right] \leq \sum_{k\in[Q],i\in[\ell]} \Pr\left[\mathsf{BadOpEarly}_{k,i}\right].$$

In the following, we fix a $k^* \in [Q]$ and $i^* \in [\ell]$, and bound the probability of $\mathsf{BadOpEarly}_{k^*,i^*}$. Again, we do so by providing a sequence of games.

<u>Game $\mathbf{H}_0$:</u> This game is exactly as game $\mathbf{G}_1$, but it terminates and outputs 1 on immediately when $\mathsf{BadOpEarly}_{k^*,i^*}$ occurs. Also, it terminates and outputs 0 immediately when $\mathcal{A}$ queries $\textsc{GetOp}(k^*, i^*)$ and the k^*th commitment has already been output by $\textsc{GetCom}(\mathbf{m}^*)$, i.e., if $\mathsf{BadOpEarly}_{k^*,i^*}$ can no longer occur. To fix notation, we let com^* be this k^*th commitment output by $\textsc{GetCom}$ and $\mathbf{m}^* \in \mathcal{M}^\ell$ be the respective oracle input. By definition, we have

$$\Pr\left[\mathsf{BadOpEarly}_{k^*,i^*}\right] = \Pr\left[\mathbf{H}_0 \Rightarrow 1\right].$$

<u>Game $\mathbf{H}_1$:</u> This game is the same as $\mathbf{H}_0$, but all commitments and openings output by $\textsc{GetCom}$ and $\textsc{GetOp}$ are simulated. That is, we no longer generate $\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell)$, but instead generate it via $(\mathsf{ck}, \mathsf{td}) \leftarrow \mathsf{TSetup}(1^\lambda, 1^\ell)$. In oracle $\textsc{GetCom}(\mathbf{m})$, we compute the commitment com by running $(\mathsf{com}, St) \leftarrow \mathsf{TCom}(\mathsf{ck})$. When we have to open this commitment at the ith position during a query to $\textsc{GetOp}$, we compute τ by running $\tau \leftarrow \mathsf{TOpen}(\mathsf{td}, St, i, \mathbf{m}_i)$. In other words, this change is similar to the change from $\mathbf{G}_2$ to $\mathbf{G}_3$. Using a similar argument as we did there, we get

$$\left|\Pr\left[\mathbf{H}_0 \Rightarrow 1\right] - \Pr\left[\mathbf{H}_1 \Rightarrow 1\right]\right| \leq \mathsf{Adv}^{\mathsf{equiv}}_{\mathcal{B},\mathsf{VC},\mathsf{Sim},\ell}(\lambda).$$

<u>Game $\mathbf{H}_2$:</u> This game is the same as $\mathbf{H}_1$, but we sample $m^* \leftarrow\!\!\$\, \mathcal{M}$ in the beginning of the game. Later, when event $\mathsf{BadOpEarly}_{k^*,i^*}$ occurs on a query $(i^*, (\mathsf{com}_j)_{j=1}^{L}, (m_j)_{j=1}^{L}, \tau)$ to oracle $\textsc{SubCom}$, the game outputs 0 if for the smallest $j \in [L]$ with $\mathsf{com}_j = \mathsf{com}^*$ we have $m_j = m^*$. Otherwise, it outputs 1 as before. Clearly, as $\mathcal{A}$ obtains no information about m^*, we have

$$\left|\Pr\left[\mathbf{H}_1 \Rightarrow 1\right] - \Pr\left[\mathbf{H}_2 \Rightarrow 1\right]\right| \leq \frac{1}{|\mathcal{M}|}.$$

<u>Game $\mathbf{H}_3$:</u> In this game, we essentially undo our change from $\mathbf{H}_0$ to $\mathbf{H}_1$, with a small but important twist. Namely, we generate $\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell)$ and compute all commitments and openings output by $\textsc{GetCom}$ and $\textsc{GetOp}$ honestly via algorithms Com and Open. Importantly, we compute com^* not as $\mathsf{com}^* := \mathsf{Com}(\mathsf{ck}, \mathbf{m}^*; \varphi^*)$, but instead as

$$\mathsf{com}^* := \mathsf{Com}(\mathsf{ck}, \tilde{\mathbf{m}}; \varphi^*), \text{ with } \tilde{\mathbf{m}}_i = \begin{cases} \mathbf{m}^*_i, & \text{if } i \in [\ell] \setminus \{i^*\} \\ m^*, & \text{if } i = i^* \end{cases}.$$

Now, in case $\mathcal{A}$ issues a query $\textsc{GetOp}(k^*, i^*)$ and the k^*th commitment has already been output by $\textsc{GetCom}(\mathbf{m}^*)$, we see that both $\mathbf{H}_2$ and $\mathbf{H}_3$ output 0 by definition. So, we can focus on the case where $\mathcal{A}$ does not make such a query. We can easily bound the difference in this case using a reduction $\mathcal{B}$ against equivocality of VC as in the change from $\mathbf{G}_2$ to $\mathbf{G}_3$. The only modification is that it forwards $\tilde{\mathbf{m}}$ on behalf of $\mathbf{m}^*$. If $\mathcal{B}$ runs in the real world game of equivocality, then it perfectly simulates $\mathbf{G}_3$ for $\mathcal{A}$. On the other hand, if it runs in the ideal world game, it perfectly simulates $\mathbf{G}_2$ because of the assumption that $\mathcal{A}$ does not make such a query. We get

$$\left|\Pr\left[\mathbf{H}_2 \Rightarrow 1\right] - \Pr\left[\mathbf{H}_3 \Rightarrow 1\right]\right| \leq \mathsf{Adv}^{\mathsf{equiv}}_{\mathcal{B},\mathsf{VC},\mathsf{Sim},\ell}(\lambda).$$

Finally, we bound the probability that $\mathbf{H}_3$ outputs 1. Namely, we sketch a reduction that breaks aggregation position-binding of VC if $\mathbf{H}_3$ outputs 1. To this end, observe that when $\mathbf{H}_3$ outputs 1, then event $\mathsf{BadOpEarly}_{k^*,i^*}$ occurs on a query $(i^*, (\mathsf{com}_j)_{j=1}^{L}, (m_j)_{j=1}^{L}, \tau)$ to oracle $\textsc{SubCom}$. We also know that in this case $m_j \neq m^*$ for the smallest $j \in [L]$ such that $\mathsf{com}_j = \mathsf{com}^*$. Thus, the reduction can return $(\tilde{\mathbf{m}}, \varphi^*, i^*, (\mathsf{com}_j)_{j=1}^{L}, (m_j)_{j=1}^{L}, \tau)$ to break aggregation position-binding of VC. We get

$$\Pr\left[\mathbf{H}_3 \Rightarrow 1\right] \leq \mathsf{Adv}^{\mathsf{a\text{-}pos\text{-}bind}}_{\mathcal{B},\mathsf{VC},\ell}(\lambda).$$

□

3.4 Simulation-Extractable Vector Commitments from KZG

We present an instantiation of vector commitments with suitable properties based on the KZG commitment scheme [KZG10]. To recall, we let PGGen be an algorithm that on input 1^λ outputs the description of prime order groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of order p, generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$, and the description of a pairing $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. We first recall the KZG commitment scheme [KZG10]. Then, we modify it to get our vector commitment scheme with suitable properties.

- $\mathsf{KZG.Setup}(1^\lambda, 1^d) \to \mathsf{ck}$:
  1. Run $\mathsf{par} := (\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, p, e) \leftarrow \mathsf{PGGen}(1^\lambda)$.
  2. Sample $\alpha \leftarrow\!\!\$\, \mathbb{Z}_p$ and $\beta \leftarrow\!\!\$\, \mathbb{Z}_p^*$, and set $h_1 := g_1^\beta \in \mathbb{G}_1$.
  3. Set $u_i := g_1^{\alpha^i}$ and $\hat{u}_i := h_1^{\alpha^i}$ for each $i \in \{0, \ldots, d\}$. Set $R := g_2^\alpha$
  4. Return $\mathsf{ck} := (\mathsf{par}, h_1, R, (u_i)_{i=0}^d, (\hat{u}_i)_{i=0}^d)$.
- $\mathsf{KZG.Com}(\mathsf{ck}, f \in \mathbb{Z}_p[X]) \to (\mathsf{com}, St)$
  1. If the degree of f is larger than d, abort.
  2. Sample a polynomial $\hat{f} \in \mathbb{Z}_p[X]$ of degree d uniformly at random.
  3. Compute $\mathsf{com} = g_1^{f(\alpha)} \cdot h_1^{\hat{f}(\alpha)}$. Note that com can be computed efficiently.
  4. Return com and $St := (f, \hat{f})$.
- $\mathsf{KZG.Open}(\mathsf{ck}, St = (f, \hat{f}), z) \to \tau$
  1. Let $\psi := (f - f(z))/(X - z) \in \mathbb{Z}_p[X]$ and $\hat{\psi} := (\hat{f} - \hat{f}(z))/(X - z) \in \mathbb{Z}_p[X]$.
  2. Set $v := g_1^{\psi(\alpha)} \cdot h_1^{\hat{\psi}(\alpha)}$. Note that v can be computed efficiently.
  3. Return $\tau := (\hat{f}(z), v)$.
- $\mathsf{KZG.Ver}(\mathsf{ck}, \mathsf{com}, z, y, \tau = (\hat{y}, v)) \to b$
  1. If $e\left(\mathsf{com} \cdot g_1^{-y} \cdot h_1^{-\hat{y}}, g_2\right) = e\left(v, R \cdot g_2^{-z}\right)$, return $b = 1$. Else, return $b = 0$.

Let $\mathsf{H}\colon \{0,1\}^* \to \mathbb{Z}_p$ and $\mathsf{H}'\colon \{0,1\}^* \to \mathbb{Z}_p$ be random oracles. We now define the vector commitment scheme $\mathsf{VC}_{\mathsf{KZG}} = (\mathsf{VC}_{\mathsf{KZG}}.\mathsf{Setup}, \mathsf{VC}_{\mathsf{KZG}}.\mathsf{Com}, \mathsf{VC}_{\mathsf{KZG}}.\mathsf{Open}, \mathsf{VC}_{\mathsf{KZG}}.\mathsf{Ver})$. Roughly, we use the linear properties of KZG to make aggregation work. To add non-malleability at the same time, we include an opening at a random position z_0 in the commitments. Typically, to commit to a vector of ℓ elements, one would work with polynomials of degree $d = \ell - 1$. As we give out one additional point $f(z_0)$ for non-malleability and still need hiding, it is natural to increase d by one to $d = \ell$. It turns out that for a technical reason in our extractability proof (see paragraph "Proof Strategy" in the proof of Lemma 4), we have to choose $d = \ell + 1$.

- $\mathsf{VC}_{\mathsf{KZG}}.\mathsf{Setup}(1^\lambda, 1^\ell) \to \mathsf{ck}$:
  1. Run $\mathsf{ck}_{\mathsf{KZG}} \leftarrow \mathsf{KZG.Setup}(1^\lambda, 1^d)$, where $d := \ell + 1$. The parameters specify message alphabet $\mathcal{M} := \mathbb{Z}_p$, opening space $\mathcal{T} := \mathbb{Z}_p \times \mathbb{G}_1$, and commitment space $\mathcal{C} := \mathbb{G}_1 \times \mathbb{Z}_p \times \mathcal{T}$.
  2. Let $\iota\colon [\ell] \to \mathbb{Z}_p$ be a fixed injection and $z_{\mathsf{out}} \in \mathbb{Z}_p$ such that 0 and z_{out} are not in the image of ι.
  3. Return $\mathsf{ck} := (\mathsf{ck}_{\mathsf{KZG}}, z_{\mathsf{out}}, \iota)$.
- $\mathsf{VC}_{\mathsf{KZG}}.\mathsf{Com}(\mathsf{ck}, \mathbf{m}) \to (\mathsf{com}, St)$
  1. Sample $\delta_0, \delta_1 \leftarrow\!\!\$\, \mathbb{Z}_p$ and let $f \in \mathbb{Z}_p[X]$ be the unique polynomial of degree $d := \ell + 1$ such that $f(0) = \delta_0$, $f(z_{\mathsf{out}}) = \delta_1$ and $f(\iota(i)) = \mathbf{m}_i$ for all $i \in [\ell]$.
  2. Run $(\mathsf{com}_{\mathsf{KZG}}, St) \leftarrow \mathsf{KZG.Com}(\mathsf{ck}, f \in \mathbb{Z}_p[X])$.
  3. Compute $z_0 := \mathsf{H}(\mathsf{com}_{\mathsf{KZG}})$ and set $y_0 := f(z_0)$.
  4. Run $\tau_0 \leftarrow \mathsf{KZG.Open}(\mathsf{ck}_{\mathsf{KZG}}, St, z_0)$.

5. Return $\mathsf{com} := (\mathsf{com}_{\mathsf{KZG}}, y_0, \tau_0)$ and St.

- $\mathsf{VC}_{\mathsf{KZG}}.\mathsf{VerCom}(\mathsf{ck}, \mathsf{com}) \to b$
  1. Parse $\mathsf{com} = (\mathsf{com}_{\mathsf{KZG}}, y_0, \tau_0)$ and set $z_0 := \mathsf{H}(\mathsf{com}_{\mathsf{KZG}})$.
  2. Return $\mathsf{KZG}.\mathsf{Ver}(\mathsf{ck}_{\mathsf{KZG}}, \mathsf{com}_{\mathsf{KZG}}, z_0, y_0, \tau_0)$.
- $\mathsf{VC}_{\mathsf{KZG}}.\mathsf{Open}(\mathsf{ck}, St, i) \to \tau$
  1. Return $\mathsf{KZG}.\mathsf{Open}(\mathsf{ck}_{\mathsf{KZG}}, St, \iota(i))$.
- $\mathsf{Aggregate}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, (\tau_j)_{j=1}^L) \to \tau$
  1. For each $j \in [L]$, parse $\tau_j = (\hat{y}_j, v_j) \in \mathbb{Z}_p \times \mathbb{G}_1$.
  2. Set $\xi := \mathsf{H}'(i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L)$.
  3. Set $\hat{y} := \sum_{j=1}^L \xi^{j-1}\hat{y}_j$ and $v := \prod_{j=1}^L v_j^{\xi^{j-1}}$.
  4. Return $\tau = (\hat{y}, v)$.
- $\mathsf{VC}_{\mathsf{KZG}}.\mathsf{Ver}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau) \to b$
  1. For each $j \in [L]$, parse $\mathsf{com}_j = (\mathsf{com}_{\mathsf{KZG},j}, y_{0,j}, \tau_{0,j}) \in \mathbb{G}_1 \times \mathbb{Z}_p \times \mathcal{T}$.
  2. Set $\xi := \mathsf{H}'(i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L)$.
  3. Compute $m := \sum_{j=1}^L \xi^{j-1} m_j$ and $\mathsf{com} := \prod_{j=1}^L \mathsf{com}_{\mathsf{KZG},j}^{\xi^{j-1}}$.
  4. Return $\mathsf{KZG}.\mathsf{Ver}(\mathsf{ck}_{\mathsf{KZG}}, \mathsf{com}, \iota(i), m, \tau)$.

In the following, we show that $\mathsf{VC}_{\mathsf{KZG}}$ satisfies equivocality, aggregation position-binding, and augmented extractability. By Lemma 1, this implies that it is simulation-extractable.

Lemma 2. *Let* $\mathsf{H}\colon \{0,1\}^* \to \mathbb{Z}_p$ *be a random oracle. Then,* $\mathsf{VC}_{\mathsf{KZG}}$ *is equivocal. Namely, there are PPT algorithms* $\mathsf{Sim} = (\mathsf{TSetup}, \mathsf{TCom}, \mathsf{TOpen})$ *such that for any polynomial* $\ell \in \mathbb{N}$ *and any algorithm* $\mathcal{A}$ *that makes at most* Q_C *queries to oracle* GetCom*, we have*

$$\mathsf{Adv}^{\mathsf{equiv}}_{\mathcal{A},\mathsf{VC},\mathsf{Sim},\ell}(\lambda) = \frac{\ell + Q_C + 1 + \ell Q_C}{p}.$$

Proof. We prove the statement by defining a sequence of hybrid games. Namely, we start with game ℓ-$\mathbf{EQUIV}^{\mathcal{A}}_{\mathsf{VC},0}(\lambda)$. Then, we provide a sequence of hybrid games until we end up at a game that we can write as ℓ-$\mathbf{EQUIV}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},1}(\lambda)$ by appropriately defining Sim.

Game $\mathbf{G}_0$: We start with $\mathbf{G}_0$, which is defined to be ℓ-$\mathbf{EQUIV}^{\mathcal{A}}_{\mathsf{VC},0}(\lambda)$. We recall the game to fix some notation. Initially, the game sets $c := 0$ and runs $\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell)$. Here, ck has the form $\mathsf{ck} = (\mathsf{ck}_{\mathsf{KZG}}, z_{\mathsf{out}}, \iota)$, where ι is an injective mapping from $[\ell]$ to $\mathbb{Z}_p$ and $\mathsf{ck}_{\mathsf{KZG}} = (\mathsf{par}, h_1, R, (u_i)_{i=0}^d, (\hat{u}_i)_{i=0}^d)$ is a commitment key for the KZG polynomial commitment of degree $d = \ell + 1$, with group parameters $\mathsf{par} = (\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, p, e)$. Specifically, we have $h_1 = g_1^\beta$ for some random $\beta \in \mathbb{Z}_p$, and $u_i = g_1^{\alpha^i}$ and $\hat{u}_i = h_1^{\alpha^i}$ for each $i \in \{0, \ldots, d\}$ and some random $\alpha \in \mathbb{Z}_p$. Further, $R = g_2^\alpha$. Then, $\mathcal{A}$ is run on input ck with access to oracles GetCom, GetOp, which are as follows:

- On input $\mathbf{m} \in \mathbb{Z}_p^\ell$, oracle GetCom increments c and computes $(\mathsf{com}, St) \leftarrow \mathsf{Com}(\mathsf{ck}, \mathbf{m})$. It stores $\mathsf{Coms}[c] := \mathsf{com}$, $\mathsf{St}[c] := St$, $\mathsf{Msgs}[c] := \mathbf{m}$ and sets $\mathsf{Ops}[c] := \emptyset$. Then, it returns com to $\mathcal{A}$. To recall com consists of a KZG commitment $\mathsf{com}_{\mathsf{KZG}}$ and an opening τ_0 for value y_0 at position $z_0 := \mathsf{H}(\mathsf{com}_{\mathsf{KZG}})$.
- On input (k, i) for which $\mathsf{Coms}[k] \neq \bot$ and $i \notin \mathsf{Ops}[k]$, oracle GetOp inserts k into $\mathsf{Ops}[k]$ and returns $\tau \leftarrow \mathsf{Open}(\mathsf{ck}, \mathsf{St}[k], i)$ to $\mathcal{A}$.

By definition, we have

$$\Pr\left[\ell\text{-}\mathbf{EQUIV}^{\mathcal{A}}_{\mathsf{VC},0}(\lambda) \Rightarrow 1\right] = \Pr[\mathbf{G}_0 \Rightarrow 1].$$

<u>**Game $\mathbf{G}_1$:**</u> In this game, we change how openings are computed. This includes both openings τ_0 that are part of commitments returned by oracle GetCom and openings τ that are output by oracle GetOp. Namely, recall that in $\mathbf{G}_0$, openings for a KZG commitment $\mathsf{com}_{\mathsf{KZG}} = g_1^{f(\alpha)} h_1^{\hat{f}(\alpha)}$ at a position $z \in \mathbb{Z}_p$ to value $y := f(z)$ are computed as $\tau := (\hat{y}, v)$, where $\hat{y} = \hat{f}(z)$ and

$$v = g_1^{\psi_i(\alpha)} h_1^{\hat{\psi}_i(\alpha)} \text{ for } \psi_i = \frac{f - y}{X - z}, \quad \hat{\psi}_i = \frac{\hat{f} - \hat{y}}{X - z}.$$

From now on, we change how v is computed. Namely, it is computed directly using α and $\mathsf{com}_{\mathsf{KZG}}$ as

$$v = \left(\mathsf{com}_{\mathsf{KZG}} \cdot g_1^{-y} h_1^{-\hat{y}}\right)^{\frac{1}{\alpha - z}}.$$

In the case that $\alpha = z$, we simply set v to be a random element in $\mathbb{G}_1$. By expanding the equation defining v in $\mathbf{G}_0$, one can see that the distribution is not changed unless $\alpha = z$. To see that the probability that $\alpha = z$ is negligible, the reader may recall at which positions $z \in \mathbb{Z}_p$ opening proofs are provided during the game. Namely, either z is in the image of ι, which is fixed independently of α and has size ℓ, or z is an output of random oracle H during a query to GetCom. There are Q_C of these queries. With a union bound, we get

$$|\Pr[\mathbf{G}_0 \Rightarrow 1] - \Pr[\mathbf{G}_1 \Rightarrow 1]| \leq \frac{\ell + Q_C}{p}.$$

<u>**Game $\mathbf{G}_2$:**</u> In this game, we change how commitments com output by GetCom are computed. Recall that in $\mathbf{G}_1$, the commitments have the form $\mathsf{com} = (\mathsf{com}_{\mathsf{KZG}}, y_0, \tau_0)$, where $\mathsf{com}_{\mathsf{KZG}} = g_1^{f(\alpha)} h_1^{\hat{f}(\alpha)}$ is a KZG commitment to a polynomial f of degree $d = \ell + 1$ with randomness $\hat{f}$, where $\hat{f}$ is a random polynomial of degree $d = \ell + 1$. Further, $y_0 = f(z_0)$ for $z_0 = \mathsf{H}(\mathsf{com}_{\mathsf{KZG}})$, and τ_0 is an opening for $\mathsf{com}_{\mathsf{KZG}}$ to value y_0 at position z_0. From now on, whenever the game needs to generate such a commitment, it samples $\hat{f}(\alpha) \leftarrow\!\!\$\, \mathbb{Z}_p$ and sets $\mathsf{com}_{\mathsf{KZG}} = g_1^{f(\alpha)} h_1^{\hat{f}(\alpha)}$. Moreover, it defines $\hat{f}$ lazily by sampling $\hat{f}(z)$ uniformly at random from $\mathbb{Z}_p$ whenever it is needed to compute an opening as in $\mathbf{G}_1$. This is the case for $z = z_0$ and for z in the image of ι. To argue that this does not change the view of $\mathcal{A}$, we only need to argue that the joint distribution of the values

$$\hat{f}(\alpha),\ \hat{f}(z_0),\ \left(\hat{f}(\iota(i))\right)_{i \in [\ell]}$$

does not change from $\mathbf{G}_1$ to $\mathbf{G}_2$. Clearly, in $\mathbf{G}_2$, these values are uniformly distributed and independent. This is the same in $\mathbf{G}_1$, as these are at most $\ell + 2$ values and $\hat{f}$ is sampled as a uniform polynomial of degree $d = \ell + 1$. We have

$$\Pr[\mathbf{G}_1 \Rightarrow 1] = \Pr[\mathbf{G}_2 \Rightarrow 1].$$

<u>**Game $\mathbf{G}_3$:**</u> In this game, we change the commitments $\mathsf{com}_{\mathsf{KZG}} = g_1^{f(\alpha)} h_1^{\hat{f}(\alpha)}$ again. Concretely, from now on, we compute $\mathsf{com}_{\mathsf{KZG}}$ by sampling a random $\gamma \leftarrow\!\!\$\, \mathbb{Z}_p$ and setting $\mathsf{com}_{\mathsf{KZG}} := g_1^{\gamma}$. To argue indistinguishability, we use the fact that, as long as h_1 is a generator of $\mathbb{G}_1$, the term $h_1^{\hat{f}(\alpha)}$ acts as a one-time pad. With probability at most $1/p$, h_1 is not a generator, and so we get

$$|\Pr[\mathbf{G}_2 \Rightarrow 1] - \Pr[\mathbf{G}_3 \Rightarrow 1]| \leq \frac{1}{p}.$$

<u>**Game $\mathbf{G}_4$:**</u> In this game, we change the commitments $\mathsf{com} = (\mathsf{com}_{\mathsf{KZG}}, y_0, \tau_0)$ output by GetCom again. However, this time, we do not change the KZG commitment $\mathsf{com}_{\mathsf{KZG}}$. Instead, we change the value y_0. Recall that until now, on a query GetCom($\mathbf{m}$) for a message $\mathbf{m} \in \mathbb{Z}_p^{\ell}$, the game sampled $\delta_0, \delta_1 \leftarrow\!\!\$\, \mathbb{Z}_p$ and defined $f \in \mathbb{Z}_p[X]$ to be the unique polynomial of degree $d = \ell + 1$ such that $f(0) = \delta_0$, $f(z_{\mathsf{out}}) = \delta_1$ and $f(\iota(i)) = \mathbf{m}_i$ for all $i \in [\ell]$. Then, it sets $\mathsf{com}_{\mathsf{KZG}} = g_1^{\gamma}$ for a random $\gamma \leftarrow\!\!\$\, \mathbb{Z}_p$, computes $z_0 := \mathsf{H}(\mathsf{com}_{\mathsf{KZG}})$ and sets $y_0 := f(z_0)$. Finally, it computed an opening τ_0 for f at z_0 as explained in $\mathbf{G}_1$. From now on, the game no longer samples δ_0 and δ_1 and no longer defines f right away. Instead, the game samples $y_0 \leftarrow\!\!\$\, \mathbb{Z}_p$ at random, sets $f(z_0) := y_0$ and defines the rest of f in a lazy way. Namely, whenever oracle

GetOp is called for this particular commitment, and the game needs $f(\iota(i))$, it sets $f(\iota(i)) := \mathbf{m}_i$ and continues as before. To argue that this does not change the view of $\mathcal{A}$, we first assume that z_0 is not in the image of ι. For this assumption, we have to account a term of $\ell Q_C/p$. If this is the case, then the joint distribution of the values

$$f(z_0),\ (f(\iota(i)))_{i\in[\ell]}$$

does not change from $\mathbf{G}_3$ to $\mathbf{G}_4$. This is because f has degree $d > \ell - 1$ and therefore $f(z_0)$ is uniform in $\mathbf{G}_3$ given all $f(\iota(i)) = \mathbf{m}_i$. We get

$$|\Pr[\mathbf{G}_3 \Rightarrow 1] - \Pr[\mathbf{G}_4 \Rightarrow 1]| \leq \frac{\ell Q_C}{p}.$$

Now, we argue that $\mathbf{G}_4$ can be written as $\ell\text{-}\mathbf{EQUIV}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},1}(\lambda)$ by appropriately defining a PPT simulator $\mathsf{Sim} = (\mathsf{TSetup}, \mathsf{TCom}, \mathsf{TOpen})$. This is because TSetup can simply output α as the trapdoor and TCom can define the commitment $\mathsf{com}_{\mathsf{KZG}}$ to be $\mathsf{com}_{\mathsf{KZG}} = g_1^{\gamma}$ for a random γ, sample $y_0 \leftarrow_\$ \mathbb{Z}_p$, and compute τ_0 as explained in $\mathbf{G}_1$. Further, TCol can use the trapdoor α to open commitments as explained in $\mathbf{G}_1$ and $\mathbf{G}_2$. □

Lemma 3. *If the d-DLOG assumption holds relative to PGGen and $\mathsf{H}'\colon \{0,1\}^* \to \mathbb{Z}_p$ is modeled as a random oracle, then $\mathsf{VC}_{\mathsf{KZG}}$ is aggregation position-binding in the algebraic group model. Concretely, for any polynomial $\ell \in \mathbb{N}$ and any algebraic PPT algorithm $\mathcal{A}$ that makes at most $Q_{\mathsf{H}'}$ queries to random oracle H', there are PPT algorithms $\mathcal{B}_1, \mathcal{B}_2$ with $\mathbf{T}(\mathcal{B}_1) \approx \mathbf{T}(\mathcal{B}_2) \approx \mathbf{T}(\mathcal{A})$ and*

$$\mathsf{Adv}^{\mathsf{a\text{-}pos\text{-}bind}}_{\mathcal{A},\mathsf{VC}_{\mathsf{KZG}},\ell}(\lambda) \leq 2 \cdot \mathsf{Adv}^{\mathsf{1\text{-}DLOG}}_{\mathcal{B}_1,\mathsf{PGGen}}(\lambda) + 2 \cdot \mathsf{Adv}^{(\ell+1)\text{-}\mathsf{DLOG}}_{\mathcal{B}_2,\mathsf{PGGen}}(\lambda) + \frac{Q_{\mathsf{H}'} L_{max}}{p},$$

where L_{max} is the maximum length of lists $(m_j)_j$ submitted by $\mathcal{A}$ as part of the input to H'.

Proof. We first recall the aggregation position-binding game for an algebraic adversary $\mathcal{A}$ and a dimension ℓ to fix some notation. Set $d := \ell + 1$ as in the scheme. First, a commitment key $\mathsf{ck} = (\mathsf{ck}_{\mathsf{KZG}}, z_{\mathsf{out}}, \iota)$ is generated, where ι is a mapping from $[\ell]$ to $\mathbb{Z}_p$ and $\mathsf{ck}_{\mathsf{KZG}} = (\mathsf{par}, h_1, R, (u_i)_{i=0}^d, (\hat{u}_i)_{i=0}^d)$ is a commitment key for the KZG polynomial commitment, with group parameters $\mathsf{par} = (\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, p, e)$. That is, $h_1 = g_1^{\beta}$ for some $\beta \in \mathbb{Z}_p$, and there is some $\alpha \in \mathbb{Z}_p$ such that $u_i = g_1^{\alpha^i}$ and $\hat{u}_i = h_1^{\alpha^i}$ for each $i \in \{0, \dots, d\}$. Further, $R = g_2^{\alpha}$. Then, when $\mathcal{A}$ terminates, it outputs the following:

- A message $\mathbf{m} \in \mathbb{Z}_p^{\ell}$ and randomness φ. Here φ has the form $\varphi = (\delta_0, \delta_1, \hat{f}') \in \mathbb{Z}_p \times \mathbb{Z}_p \times \mathbb{Z}_p[X]$, where $\hat{f}'$ is of degree ℓ. Based on this output, the aggregation position-binding game honestly computes a commitment com. More concretely, let $f' \in \mathbb{Z}_p[X]$ be the polynomial of degree $d = \ell + 1$ with $f'(0) = \delta_0$, $f'(z_{\mathsf{out}}) = \delta_1$, and $f'(\iota(i)) = \mathbf{m}_i$ for every $i \in [\ell]$. Then, the commitment com has the form $\mathsf{com} = (\mathsf{com}_{\mathsf{KZG}}, y_0, \tau_0)$, where $\mathsf{com}_{\mathsf{KZG}} = g_1^{f'(\alpha)} h_1^{\hat{f}'(\alpha)}$.
- An index $i^* \in [\ell]$. We will denote $z := \iota(i^*)$. Further, we will denote by ψ', $\hat{\psi}' \in \mathbb{Z}_p[X]$ the polynomials
$$\psi' := \frac{f' - f'(z)}{X - z},\quad \hat{\psi}' := \frac{\hat{f}' - \hat{f}'(z)}{X - z}$$
as in an honest KZG opening for f' at position z. The game can efficiently compute these polynomials.
- Lists $(\mathsf{com}_j)_{j=1}^L$ and $(m_j)_{j=1}^L$, and an opening $\tau = (\hat{y}, v) \in \mathbb{Z}_p \times \mathbb{G}_1$. Concretely, each com_j has the form $\mathsf{com}_j = (\mathsf{com}_{\mathsf{KZG},j}, y_{0,j}, \tau_{0,j})$, where $\mathsf{com}_{\mathsf{KZG},j} \in \mathbb{G}_1$. As $\mathcal{A}$ is algebraic, it also outputs an algebraic representation for each $\mathsf{com}_{\mathsf{KZG},j}$ and for τ. Due to the structure of the commitment key, this is equivalent to saying that $\mathcal{A}$ outputs polynomials $f_j, \hat{f}_j \in \mathbb{Z}_p[X]$ and $\psi, \hat{\psi} \in \mathbb{Z}_p[X]$ all of degree at most $d = \ell + 1$ such that
$$\tau = g_1^{\psi(\alpha)} \cdot h_1^{\hat{\psi}(\alpha)} \wedge \forall j \in [L] : \mathsf{com}_{\mathsf{KZG},j} = g_1^{f_j(\alpha)} \cdot h_1^{\hat{f}_j(\alpha)}.$$

We denote the event that $\mathcal{A}$ breaks aggregation position-binding by Win. Assuming that Win occurs, we know the following: There is an index $j^* \in [L]$ such that $\mathsf{com}_{j^*} = \mathsf{com}$ and $m_{j^*} \neq \mathbf{m}_{i^*}$. In particular, this means that $\mathsf{com}_{\mathsf{KZG}} = \mathsf{com}_{\mathsf{KZG},j^*}$ and $m_{j^*} \neq f'(z)$. We have $\mathsf{VC}_{\mathsf{KZG}}.\mathsf{Ver}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau) = 1$. In particular, by reading the verification equation in the exponent, we have

$$\sum_{j=1}^{L} \xi^{j-1}(f_j(\alpha) + \beta\hat{f}_j(\alpha) - m_j) - \beta\hat{y} = (\psi(\alpha) + \beta\hat{\psi}(\alpha))(\alpha - z)$$

for $\xi := \mathsf{H}'(i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L)$. Defining polynomials $f := \sum_{j=1}^L f_j \xi^{j-1} \in \mathbb{Z}_p[X]$ and $\hat{f} := \sum_{j=1}^L \hat{f}_j \xi^{j-1} \in \mathbb{Z}_p[X]$, and the element $m := \sum_{j=1}^L m_j \xi^{j-1}$ simplifies this equation to

$$f(\alpha) + \beta\hat{f}(\alpha) - m - \beta\hat{y} = (\psi(\alpha) + \beta\hat{\psi}(\alpha))(\alpha - z).$$

By construction of $f', \hat{f}'$ and $\psi', \hat{\psi}'$, we also have

$$f'(\alpha) + \beta\hat{f}'(\alpha) - f'(z) - \beta\hat{f}'(z) = (\psi'(\alpha) + \beta\hat{\psi}'(\alpha))(\alpha - z).$$

Subtracting the two equations, we get our *core equation*, namely,

$$\begin{aligned} &f(\alpha) - f'(\alpha) + \beta(\hat{f}(\alpha) - \hat{f}'(\alpha)) - (m - f'(z)) - \beta(\hat{y} - \hat{f}'(z)) \\ &= (\psi(\alpha) - \psi'(\alpha) + \beta(\hat{\psi}(\alpha) - \hat{\psi}'(\alpha)))(\alpha - z). \end{aligned}$$

Our goal will be to simplify the structure of this core equation. To do so, our first step is to eliminate the terms containing β. We define the following event:

- Event $\mathsf{Complex}$: This event occurs, if $\eta := \hat{f}(\alpha) - \hat{f}'(\alpha) - (\hat{y} - \hat{f}'(z)) - (\hat{\psi}(\alpha) - \hat{\psi}'(\alpha))(\alpha - z) \neq 0$.

Claim. There is a PPT algorithm $\mathcal{B}$ with $\Pr[\mathsf{Win} \wedge \mathsf{Complex}] \leq \mathsf{Adv}_{\mathcal{B},\mathsf{PGGen}}^{1\text{-}\mathsf{DLOG}}(\lambda)$.

Proof of Claim. Reduction $\mathcal{B}$ is as follows. It gets as input the group parameters, the generator g_1 and the element $h_1 = g_1^\beta$. We show that it can simulate the game for $\mathcal{A}$ and compute β if $\mathsf{Win} \wedge \mathsf{Complex}$ occurs. For that, $\mathcal{B}$ first samples $\alpha \leftarrow\!\!\$\, \mathbb{Z}_p$ and computes the commitment key ck as in algorithm Setup. Observe that for that, β is not needed. Now, if Win occurs, then we know that the core equation holds. For convenience, we group together the β-terms in our core equation, getting

$$\beta \cdot \eta = f'(\alpha) - f(\alpha) + (m - f'(z)) + (\psi(\alpha) - \psi'(\alpha))(\alpha - z).$$

Clearly, if $\mathsf{Complex}$, then reduction $\mathcal{B}$ can compute β by multiplying the right hand-side with η^{-1}.

From now on, we condition on $\neg\mathsf{Complex}$. In other words, we assume that $\eta = 0$, which implies that the *simplified core equation*

$$f(\alpha) - m - (f'(\alpha) - f'(z)) = (\psi(\alpha) - \psi'(\alpha))(\alpha - z)$$

holds. Our goal will be to show that if this equation holds, we can build a reduction breaking the d-DLOG assumption. For that, we define the following events:

- Event NoColl : This event occurs, if $f'(\alpha) \neq f_{j^*}(\alpha)$.
- Event Ambig : This event occurs, if $f' \neq f_{j^*}$ over $\mathbb{Z}_p[X]$.
- Event $\mathsf{AggFail}$: This event occurs, if $m = f(z)$.

In the next claims, we bound the probability that one of these event occurs. Informally, if NoColl occurs, then one can use $\mathsf{com}_{\mathsf{KZG}} = \mathsf{com}_{\mathsf{KZG},j^*}$ solve for β to break DLOG. If Ambig occurs but NoColl does not, then we can find α efficiently as a root of the non-zero polynomial $f' - f_{j^*}$. We will bound the case that $\mathsf{AggFail}$ occurs using a statistical argument using the fact that ξ is chosen after the f_j and m_j are fixed.

Claim. There is a PPT algorithm $\mathcal{B}$ with $\Pr[\mathsf{Win} \wedge \mathsf{NoColl}] \leq \mathsf{Adv}_{\mathcal{B},\mathsf{PGGen}}^{1\text{-}\mathsf{DLOG}}(\lambda)$.

Proof of Claim. Note that if Win occurs, then we have $f'(\alpha) + \beta\hat{f}'(\alpha) = f_{j^*}(\alpha) + \beta\hat{f}_{j^*}(\alpha)$, because $\mathsf{com}_{\mathsf{KZG}} = \mathsf{com}_{\mathsf{KZG},j^*}$. If NoColl occurs at the same time, then we know that $\hat{f}'(\alpha) - \hat{f}_{j^*}(\alpha) \neq 0$ and one can efficiently solve for β. It is not hard to turn that into a formal reduction that determines β given $h_1 = g_1^{\beta}$.

Claim. There is a PPT algorithm $\mathcal{B}$ with $\Pr[\mathsf{Ambig} \wedge \neg\mathsf{NoColl}] \leq \mathsf{Adv}^{d\text{-}\mathsf{DLOG}}_{\mathcal{B},\mathsf{PGGen}}(\lambda)$.

Proof of Claim. Note that if $\neg\mathsf{NoColl}$ occurs, then we have $f'(\alpha) \neq f_{j^*}(\alpha)$. If Ambig occurs at the same time, we know that α is a root of the non-zero polynomial $f' - f_{j^*}$, which has degree at most $d = \ell + 1$. Hence, α can be found in polynomial time by a reduction. We leave details to the reader.

Claim. We have $\Pr[\mathsf{Win} \wedge \mathsf{AggFail} \wedge \neg\mathsf{Ambig}] \leq Q_{\mathsf{H}'} L_{max}/p$.

Proof of Claim. By definition of m and f, event $\mathsf{AggFail}$ is equivalent to the equation

$$\sum_{j=1}^{L} m_j \xi^{j-1} = \sum_{j=1}^{L} f_j(z) \xi^{j-1}.$$

In other words, if $\mathsf{AggFail}$ occurs, then the polynomial

$$\zeta = \sum_{j=1}^{L} (f_j(z) - m_j) X^{j-1} \in \mathbb{Z}_p[X]$$

has a root at ξ. Observe that ζ has degree $L \leq L_{max}$ and is fixed before[3] ξ is chosen at random by the random oracle H'. Thus, for any fixed random oracle query where $\zeta \neq 0$, this event occurs with probability at most L_{max}/p. It remains to argue that $\zeta \neq 0$ if Win occurs and Ambig does not. This can be seen by observing that the j^*th coefficient of ζ is non-zero, i.e., $m_{j^*} \neq f_{j^*}(z)$. Namely, we know that $f' = f_{j^*}$ due to $\neg\mathsf{Ambig}$. Thus, if we had $m_{j^*} = f_{j^*}(z)$, then we had $m_{j^*} = f'(z)$, contradicting Win.

Concluding the Proof. To conclude the proof, we can now assume that Win occurs, but neither of the events defined above occurs. We come back to our simplified core equation. The equation tells us that α is a root of the polynomial Ψ of degree at most $d = \ell + 1$, which is given as

$$\Psi = f - m - (f' - f'(z)) - (\psi - \psi')(X - z) \in \mathbb{Z}_p[X].$$

Now, if we can argue that Ψ is non-zero, then one can efficiently find α based on $\mathcal{A}$'s output, leading to our final reduction. To argue that Ψ is non-zero, note that $\Psi = 0$ implies that

$$f - m = (f' - f'(z)) + (\psi - \psi')(X - z).$$

While the right hand-side is a multiple of $X - z$, the left hand-side is not, as we assume $\neg\mathsf{AggFail}$. Therefore, this equality can not hold, which means that Ψ is non-zero. In combination, setting $\mathsf{Bad} := \mathsf{NoColl} \vee \mathsf{Ambig} \vee \mathsf{AggFail} \vee \mathsf{Complex}$ we get a reduction $\mathcal{B}$ with

$$\Pr[\mathsf{Win} \wedge \neg\mathsf{Bad}] \leq \mathsf{Adv}^{d\text{-}\mathsf{DLOG}}_{\mathcal{B},\mathsf{PGGen}}(\lambda).$$

□

Lemma 4. *If the d-DLOG assumption holds relative to PGGen and $\mathsf{H}\colon \{0,1\}^* \to \mathbb{Z}_p$ is modeled as a random oracle, then $\mathsf{VC}_{\mathsf{KZG}}$ satisfies augmented extractability in the algebraic group model. Concretely, there is a PPT algorithm Ext such that for any polynomial $\ell \in \mathbb{N}$ and any algebraic PPT algorithm $\mathcal{A}$, there are PPT algorithms $\mathcal{B}_1, \mathcal{B}_2$ with $\mathbf{T}(\mathcal{B}_1) \approx \mathbf{T}(\mathcal{B}_2) \approx \mathbf{T}(\mathcal{A})$ and*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{aug\text{-}ext}}_{\mathcal{A},\mathsf{VC}_{\mathsf{KZG}},\mathsf{Ext},\ell}(\lambda) \leq\ & \frac{Q_{\mathsf{H}}^2 + Q_{\mathsf{H}} + \ell + Q_{\mathsf{H}} Q_C (\ell+1)}{p} \\ & + (Q_C + 2) \cdot \mathsf{Adv}^{1\text{-}\mathsf{DLOG}}_{\mathcal{B},\mathsf{PGGen}}(\lambda) + 2 \cdot \mathsf{Adv}^{(\ell+1)\text{-}\mathsf{DLOG}}_{\mathcal{B},\mathsf{PGGen}}(\lambda).\end{aligned}$$

[3] It could be that the adversary submitted a different algebraic representation to the random oracle. In this case, we just define the f_j's to be this representation.

Proof. We first recall the augmented extractability game to fix notation and define extractor Ext. Then, we provide a proof intuition. Finally, we proceed with the details of the proof.

Game, Extractor and Notation. In the augmented extractability game for $\mathsf{VC}_{\mathsf{KZG}}$, the adversary $\mathcal{A}$ gets as input a commitment key ck and access to a commitment oracle GetCom and an opening oracle GetOp. These are as follows:

- The commitment key ck has the form $\mathsf{ck} = (\mathsf{ck}_{\mathsf{KZG}}, z_{\mathsf{out}}, \iota)$ where $\iota \colon [\ell] \to \mathbb{Z}_p$ is injective and $\mathsf{ck}_{\mathsf{KZG}} = (\mathsf{par}, h_1, R, (u_i)_{i=0}^d, (\hat{u}_i)_{i=0}^d)$ is a KZG commitment key with group parameters $\mathsf{par} = (\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, p, e)$. We have $h_1 = g_1^{\beta}$ for some random $\beta \in \mathbb{Z}_p$, and $u_i = g_1^{\alpha^i}$ and $\hat{u}_i = h_1^{\alpha^i}$ for each $i \in \{0, \dots, d\}$ for some random $\alpha \in \mathbb{Z}_p$. We also have $R = g_2^{\alpha}$.
- On input $\mathbf{m} \in \mathbb{Z}_p^{\ell}$, the commitment oracle returns a commitment com for $\mathbf{m}$. We use the subscript k to refer to the kth commitment returned by the oracle. That is, $\mathsf{com}_k = (\mathsf{com}_{\mathsf{KZG},k}, f_k(z_{k,0}), \tau_{k,0})$ is the kth commitment returned by the oracle, where $\mathsf{com}_{\mathsf{KZG},k} = g_1^{f_k(\alpha)} h_1^{\hat{f}_k(\alpha)}$ is a KZG commitment to a polynomial f_k of degree d with randomness $\hat{f}_k$, and $\tau_{k,0} = (\hat{f}_k(z_{k,0}), v_{k,0})$ is a KZG opening for f_k at position $z_{k,0} = \mathsf{H}(\mathsf{com}_{\mathsf{KZG},k})$ to value $f_k(z_{k,0})$. We denote the number of queries to GetCom by Q_C and assume without loss of generality that $Q_C \geq 1$.
- On input (k, i) such that com_k is defined, the opening oracle GetOp opens com_k at position i. To recall, such an opening is a KZG openings for commitment $\mathsf{com}_{\mathsf{KZG},k}$ at position $z_{k,i} := \iota(i)$. We denote the opening by $\tau_{k,i} = (\hat{f}_k(z_{k,i}), v_{k,i})$ for $v_{k,i} = g_1^{\psi_{k,i}(\alpha)} h_1^{\hat{\psi}_{k,i}(\alpha)}$, where $\psi_{k,i} = (f_k - f_k(z_{k,i}))/(X - z_{k,i}) \in \mathbb{Z}_p[X]$ and $\hat{\psi}_{k,i} := (\hat{f}_k - \hat{f}_k(z_{k,i}))/(X - z_{k,i})$. Without loss of generality, we can assume that for every commitment com_k returned by the commitment oracle, $\mathcal{A}$ queries the opening oracle for every $i \in [\ell]$, and thus it obtained all of these openings $\tau_{k,i}$ for $i \in \{0, \dots, \ell\}$.

When $\mathcal{A}$ terminates, it outputs a commitment outputs $\mathsf{com} = (\mathsf{com}_{\mathsf{KZG}}, y_0, \tau_0)$ with $\tau_0 = (\hat{y}_0, v_0)$. As $\mathcal{A}$ is algebraic, it also outputs the algebraic representation of all group elements in com. Due to the structure of ck and the group elements that $\mathcal{A}$ obtained from the commitment and opening oracles, we can assume that this representation is given by polynomials $f, \hat{f}, \psi, \hat{\psi} \in \mathbb{Z}_p[X]$ of degree $d = \ell + 1$ and lists of exponents $(w_k)_k, (r_k)_k$ and $(t_{k,i})_{k,i}, (s_{k,i})_{k,i}$ over $\mathbb{Z}_p$ such that

$$\mathsf{com}_{\mathsf{KZG}} = g_1^{f(\alpha)} \cdot h_1^{\hat{f}(\alpha)} \cdot \underbrace{\prod_{k=1}^{Q_C} \mathsf{com}_{\mathsf{KZG},k}^{w_k} \cdot \prod_{k=1}^{Q_C} \prod_{i=0}^{\ell} v_{k,i}^{t_{k,i}}}_{=:L}$$

and

$$v_0 = g_1^{\psi(\alpha)} \cdot h_1^{\hat{\psi}(\alpha)} \cdot \prod_{k=1}^{Q_C} \mathsf{com}_{\mathsf{KZG},k}^{r_k} \cdot \prod_{k=1}^{Q_C} \prod_{i=0}^{\ell} v_{k,i}^{s_{k,i}}.$$

Without loss of generality, we assume that $\mathcal{A}$ queried $\mathsf{H}(\mathsf{com}_{\mathsf{KZG}})$, and it did so with the same algebraic representation for $\mathsf{com}_{\mathsf{KZG}}$ as the one that it outputs in the end. Given the output of the adversary, the extractor returns the message $\mathbf{m} \in \mathbb{Z}_p^{\ell}$ defined by $\mathbf{m}_i := f(\iota(i))$ for all $i \in [\ell]$ and the randomness $\varphi := (\delta_0, \delta_1, \hat{f})$, where $\delta_0 := f(0)$ and $\delta_1 := f(z_{\mathsf{out}})$. Now, $\mathcal{A}$ wins the game, if the following three properties hold:

- The commitment com is fresh, i.e., it was not output by the commitment oracle GetCom.
- Committing to $\mathbf{m}$ with randomness φ does not yield com. It is easy to see that this can only happen if $L \neq g_1^0$.
- The commitment com verifies, i.e., $\mathsf{VC}_{\mathsf{KZG}}.\mathsf{VerCom}(\mathsf{ck}, \mathsf{com}) = 1$. This is equivalent to saying that τ_0 is a valid KZG opening for $\mathsf{com}_{\mathsf{KZG}}$ at position $z_0 = \mathsf{H}(\mathsf{com}_{\mathsf{KZG}})$ to value y_0, i.e., that

$$e\left(\mathsf{com}_{\mathsf{KZG}} \cdot g_1^{-y_0} \cdot h_1^{-\hat{y_0}}, g_2\right) = e\left(v_0, g_2^{\alpha} \cdot g_2^{-z_0}\right).$$

Proof Strategy. In a preparatory phase (see $\mathbf{G}_0$ to $\mathbf{G}_3$), we rule out some simple bad events and simplify some equations. Namely, we rule out that there are collisions among the z's, e.g., that $z_0 = z_{k,i}$ for some i and k. We also rule out that α is equal to one of those z's. Further, we ensure that not only com is fresh, but instead $\mathsf{com}_{\mathsf{KZG}}$ is fresh. For that, we need to rule out that the adversary reuses a $\mathsf{com}_{\mathsf{KZG},k}$ with a different opening. We also expand the verification equation using the algebraic representation, and simplify it by ensuring that the exponent of h_1 is zero. Indeed, if this were not the case, one could compute the discrete logarithm β of $h_1 = g_1^{\beta}$. After this preparatory phase, our main argument follows (see $\mathbf{G}_4$ to $\mathbf{G}_7$). Namely, we show that from the adversary's output, a reduction can efficiently compute $f_{k^*}(z_0)$ for some k^*, while it only provided the $\ell + 1 = d$ evaluations $f_{k^*}(z_{k^*,i})$ for $i \in \{0, \ldots, \ell\}$ to the adversary. With this additional evaluation point $f_{k^*}(z_0)$, the reduction can compute f_{k^*} entirely. Roughly, this observation can be used as follows: The reduction guesses k^*, interprets a DLOG instance $X^* = g_1^{x^*}$ as $g_1^{f_{k^*}(\alpha)}$, and embeds it into the commitment $\mathsf{com}_{\mathsf{KZG},k^*}$. With the output of the adversary, it can efficiently recover f_{k^*} and therefore the discrete logarithm $x^* = f_{k^*}(\alpha)$.

Proof Details. Now, we turn to the details of the proof. We bound the probability that $\mathcal{A}$ wins the augmented extractability game using a sequence of games.

Game $\mathbf{G}_0$: This game is the augmented extractability game as presented above. To recall, $\mathcal{A}$ wins if the commitment com that it outputs is fresh, i.e., not output by GetCom, $L \neq g_1^0$, and the KZG verification equation holds for commitment $\mathsf{com}_{\mathsf{KZG}}$ at position z_0 with opening τ_0 to value y_0. By definition, we have

$$\mathsf{Adv}^{\mathsf{aug\text{-}ext}}_{\mathcal{A},\mathsf{VC},\mathsf{Ext},\ell}(\lambda) = \Pr[\mathbf{G}_0 \Rightarrow 1].$$

Game $\mathbf{G}_1$: This game is as $\mathbf{G}_0$, but it outputs 0 if there is a collision for random oracle H or if α is output by H, or if α is in the image of ι. The probability of the first event is at most Q_{H}^2/p, the probability of the second is at most Q_{H}/p, and the probability of the third is at most ℓ/p so we get

$$|\Pr[\mathbf{G}_0 \Rightarrow 1] - \Pr[\mathbf{G}_1 \Rightarrow 1]| \leq \frac{Q_{\mathsf{H}}^2 + Q_{\mathsf{H}} + \ell}{p}.$$

Game $\mathbf{G}_2$: This game is as $\mathbf{G}_2$, but we change the freshness condition which is part of the winning condition. Namely, we now require that $\mathsf{com}_{\mathsf{KZG}}$ is fresh, meaning that there is no $k \in [Q_C]$ such that $\mathsf{com}_{\mathsf{KZG}} = \mathsf{com}_{\mathsf{KZG},k}$. Clearly, the games only differ if the KZG verification equation holds for commitment $\mathsf{com}_{\mathsf{KZG}}$ at position z_0 with opening τ_0 to value y_0, and $\mathsf{com}_{\mathsf{KZG}} = \mathsf{com}_{\mathsf{KZG},k}$ (and thus $z_0 = z_{k,0}$) for some $k \in [Q_C]$, but $(y_0, \tau_0) \neq (f_k(z_0), \tau_{k,0})$. To bound the probability of this event, we assume the event occurs and consider three cases. In the first case, we have $y_0 = f_k(z_0)$ and $\hat{y}_0 = \hat{f}_k(z_0)$. It is easy to see that the verification equation now implies that $v_0 = v_{k,0}$, and so $(y_0, \tau_0) = (f_k(z_0), \tau_{k,0})$, a contradiction. In the second case, we have $(y_0, \hat{y}_0) \neq (f_k(z_0), \hat{f}_k(z_0))$, but $y_0 + \beta\hat{y}_0 = f_k(z_0) + \beta\hat{f}_k(z_0)$. In this case, it is easy to build a reduction that gets as input g_1^{β} and breaks DLOG by finding β. In the third case, we have $y_0 + \beta\hat{y}_0 \neq f_k(z_0) + \beta\hat{f}_k(z_0)$. Set $\bar{y} := y_0 + \beta\hat{y}_0$ and $\bar{y}' := f_k(z_0) + \beta\hat{f}_k(z_0)$. In this case, intuitively, we can break the standard position-binding of KZG. More precisely, we sketch a reduction that breaks d-SDH. Namely, it gets as input group parameters and $g_1^{\alpha^i}$ for all $i \in [d]$, and g_2^{α}, and uses this to define the commitment key for $\mathcal{A}$. It simulates $\mathbf{G}_1$ for $\mathcal{A}$. Assuming that the event we want to bound occurs and we are in the third case, we know from the verification equation that

$$g_1^{\bar{y}} \cdot v_0^{\alpha - z_0} = \mathsf{com}_{\mathsf{KZG}} = \mathsf{com}_{\mathsf{KZG},k} = g_1^{\bar{y}'} \cdot v_{k,0}^{\alpha - z_{k,0}}.$$

Using $z_0 = z_{k,0}$ and $\bar{y} \neq \bar{y}'$, this implies that

$$g_1^{\frac{1}{\alpha - z_0}} = \left(v_0 \cdot v_{k,0}^{-1}\right)^{\frac{1}{\bar{y}' - \bar{y}}},$$

which can be computed efficiently by the reduction. In combination, we get

$$|\Pr[\mathbf{G}_1 \Rightarrow 1] - \Pr[\mathbf{G}_2 \Rightarrow 1]| \leq \mathsf{Adv}^{1\text{-}\mathsf{DLOG}}_{\mathcal{B},\mathsf{PGGen}}(\lambda) + \mathsf{Adv}^{d\text{-}\mathsf{DLOG}}_{\mathcal{B},\mathsf{PGGen}}(\lambda).$$

Game $\mathbf{G}_3$: The game is as $\mathbf{G}_2$, but it outputs 0 if a bad event $\mathsf{Complex}$ occurs, which is defined as follows.

- Event $\mathsf{Complex}$: This event occurs, if $\eta_L \neq \eta_R$, where

$$\eta_L = \hat{f}(\alpha) + \sum_{k=1}^{Q_C} w_k \hat{f}_k(\alpha) + \sum_{k=1}^{Q_C}\sum_{i=0}^{\ell} t_{k,i}\hat{\psi}_{k,i}(\alpha) - \hat{y}_0,$$
$$\eta_R = \left(\hat{\psi}(\alpha) + \sum_{k=1}^{Q_C} r_k \hat{f}_k(\alpha) + \sum_{k=1}^{Q_C}\sum_{i=0}^{\ell} s_{k,i}\hat{\psi}_{k,i}(\alpha)\right)(\alpha - z_0).$$

Roughly, this event occurs if when we expand the verification equation using the algebraic representation of $\mathsf{com}_{\mathsf{KZG}}$ and v_0, the combined exponent of h_1 is non-zero. It is easy to see that if this event occurs and $\mathbf{G}_2$ outputs 1, then one can efficiently compute the discrete logarithm β of $h_1 = g_1^{\beta}$. We leave the details of a reduction $\mathcal{B}$ to the reader, and get

$$|\Pr[\mathbf{G}_2 \Rightarrow 1] - \Pr[\mathbf{G}_3 \Rightarrow 1]| \leq \Pr[\mathsf{Complex}] \leq \mathsf{Adv}^{1\text{-}\mathsf{DLOG}}_{\mathcal{B},\mathsf{PGGen}}(\lambda).$$

Now, given that $\mathbf{G}_3$ outputs 1, we know that

$$f(\alpha) + \bar{f}(\alpha) + \bar{\psi}(\alpha) - y_0 = \left(\psi(\alpha) + \tilde{f}(\alpha) + \tilde{\psi}(\alpha)\right)(\alpha - z_0)$$

for the polynomials

$$\bar{f} := \sum_{k=1}^{Q_C} w_k f_k, \quad \bar{\psi} := \sum_{k=1}^{Q_C}\sum_{i=0}^{\ell} t_{k,i}\psi_{k,i}, \quad \tilde{f} := \sum_{k=1}^{Q_C} r_k f_k, \quad \tilde{\psi} := \sum_{k=1}^{Q_C}\sum_{i=0}^{\ell} s_{k,i}\psi_{k,i}.$$

We call this our *simplified core equation.*

Game $\mathbf{G}_4$: This game is as $\mathbf{G}_3$, but we introduce another bad event $\mathsf{WrongValue}$ and let the game output 0 if this event occurs.

- Event $\mathsf{WrongValue}$: This event occurs, if $\bar{f}(z_0) + \bar{\psi}(z_0) \neq y_0 - f(z_0)$.

Next, we bound the probability that $\mathsf{WrongValue}$ occurs and $\mathbf{G}_3$ outputs 1. For that, we see that $\mathsf{WrongValue}$ implies that the polynomial $f + \bar{f} + \bar{\psi} - y_0$ does not evaluate to 0 at z_0. Therefore, we have

$$f + \bar{f} + \bar{\psi} - y_0 \neq \left(\psi + \tilde{f} + \tilde{\psi}\right)(X - z_0)$$

over $\mathbb{Z}_p[X]$, which implies that

$$\zeta := f + \bar{f} + \bar{\psi} - y_0 - \left(\psi + \tilde{f} + \tilde{\psi}\right)(X - z_0) \neq 0 \in \mathbb{Z}_p[X].$$

However, it follows from our simplified core equation that α is a root of ζ, which has degree at most $d+1$ by definition. Therefore, a reduction that gets as input $g_1^{\alpha^i}$ for $i \in \{0, \ldots, d\}$ and g_2^{α} can efficiently compute α as a root of ζ (which is known to the reduction) if event $\mathsf{WrongValue}$ occurs and $\mathbf{G}_3$ would output 1. We get

$$|\Pr[\mathbf{G}_3 \Rightarrow 1] - \Pr[\mathbf{G}_4 \Rightarrow 1]| \leq \Pr[\mathsf{WrongValue}] \leq \mathsf{Adv}^{d\text{-}\mathsf{DLOG}}_{\mathcal{B},\mathsf{PGGen}}(\lambda).$$

Before we continue, we introduce more notation and make an observation. Namely, we want to leverage the fact that we know $\bar{f}(z_0) + \bar{\psi}(z_0) = y_0 - f(z_0)$ by relating this term to the polynomials f_k. For that, we recall the definition of $\bar{f}$ and $\bar{\psi}$ and get

$$\begin{aligned}
y_0 - f(z_0) &= (\bar{f} + \bar{\psi})(z_0) \\
&= \sum_{k=1}^{Q_C} \Big(w_k f_k(z_0) + \sum_{i=0}^{\ell} t_{k,i} \psi_{k,i}(z_0) \Big) \\
&= \sum_{k=1}^{Q_C} \Big(w_k f_k(z_0) + \sum_{i=0}^{\ell} t_{k,i} \frac{f_k(z_0) - f_k(z_{k,i})}{z_0 - z_{k,i}} \Big) \\
&= \sum_{k=1}^{Q_C} \Big(\underbrace{\Big(w_k + \sum_{i=0}^{\ell} \frac{t_{k,i}}{z_0 - z_{k,i}} \Big)}_{=:c_k(z_0)} f_k(z_0) - \sum_{i=0}^{\ell} \frac{t_{k,i} f_k(z_{k,i})}{z_0 - z_{k,i}} \Big) \\
&= \sum_{k=1}^{Q_C} \Big(c_k(z_0) f_k(z_0) - \sum_{i=0}^{\ell} \frac{t_{k,i} f_k(z_{k,i})}{z_0 - z_{k,i}} \Big).
\end{aligned}$$

We call this equation our *helper equation*. Roughly, our crucial observation is that if $c_k(z_0) \neq 0$, then we can compute $f_k(z_0)$ from all $f_k(z_{k,i})$ and y_0, $f(z_0)$.

Game $\mathbf{G}_5$: This game is as $\mathbf{G}_4$, but we introduce another bad event $\mathsf{CoeffZero}$ and let the game output 0 if this event occurs.

- Event $\mathsf{CoeffZero}$: This event occurs, if $c_k(z_0) = 0$ for all $k \in [Q_C]$.

We bound the probability that event $\mathsf{CoeffZero}$ occurs and $\mathbf{G}_4$ outputs 1 as follows: By multiplying with $\prod_{j=0}^{\ell}(z_0 - z_{k,j})$, we see that $c_k(z_0) = 0$ implies that $p_k(z_0) = 0$ for the polynomial

$$p_k := w_k \prod_{i=0}^{\ell}(X - z_{k,i}) + \sum_{i=0}^{\ell} t_{k,i} \prod_{j=0, j\neq i}^{\ell} (X - z_{k,j}).$$

Note that p_k has degree at most $\ell + 1$ and is fixed entirely when $\mathcal{A}$ queried $\mathsf{H}(\mathsf{com}_{\mathsf{KZG}})$ for the first time, i.e., before z_0 is sampled uniformly from $\mathbb{Z}_p$. Therefore, if we can argue that there is a $k \in [Q_C]$ such that $p_k \neq 0$, then event $\mathsf{CoeffZero}$ occurs with probability at most $Q_{\mathsf{H}} Q_C(\ell+1)/p$. To argue that there is such a k, assume towards contradiction that $p_k = 0$ for all $k \in [Q_C]$. It is easy to see that the coefficient of $X^{\ell+1}$ in p_k is w_k, and so $w_k = 0$ for all $k \in [Q_C]$. Now, if $p_k = 0$ for all $k \in [Q_C]$, then especially $p_k(z_{k,i}) = 0$ for all $i \in \{0, \dots, \ell\}$ and all $k \in [Q_C]$. So we get

$$0 = p_k(z_{k,i}) = \sum_{i'=0}^{\ell} t_{k,i'} \prod_{j=0, j\neq i'}^{\ell} (z_{k,i} - z_{k,j}) = t_{k,i} \prod_{j=0, j\neq i} (z_{k,i} - z_{k,j}),$$

and thus $t_{k,i} = 0$ for all $k \in [Q_C]$ and all $i \in \{0, \dots, \ell\}$. However, if all w_k and all $t_{k,i}$ are zero, then $L = g_1^0$, contradicting $L \neq g_1^0$. With this, we get

$$|\Pr[\mathbf{G}_4 \Rightarrow 1] - \Pr[\mathbf{G}_5 \Rightarrow 1]| \leq \Pr[\mathsf{CoeffZero}] \leq \frac{Q_{\mathsf{H}} Q_C(\ell+1)}{p}.$$

Game $\mathbf{G}_6$: In this game, we sample a $k^* \leftarrow\!\!\$\, [Q_C]$ in the beginning of the game. Then, we run the game exactly as in $\mathbf{G}_5$. In the end, we let the game output 0 if $c_{k^*}(z_0) = 0$, and output whatever $\mathbf{G}_5$ would have output otherwise. Clearly, the view of $\mathcal{A}$ is independent of k^*, and if $\mathbf{G}_5$ outputs 1, then there must be at least one k such that $c_k(z_0) \neq 0$. Therefore, we have

$$\Pr[\mathbf{G}_5 \Rightarrow 1] \leq Q_C \cdot \Pr[\mathbf{G}_6 \Rightarrow 1].$$

Game $\mathbf{G}_7$: In this game, we change how the openings $v_{k^*,i}$ are computed. Recall that until now, these are computed as $v_{k^*,i} = g_1^{\psi_{k^*,i}(\alpha)} h_1^{\hat{\psi}_{k^*,i}(\alpha)}$, where $\psi_{k^*,i} = (f_{k^*} - f_{k^*}(z_{k*,i}))/(X - z_{k^*,i}) \in \mathbb{Z}_p[X]$ and $\hat{\psi}_{k^*,i} := (\hat{f}_{k^*} - \hat{f}_{k^*}(z_{k*,i}))/(X - z_{k^*,i})$. Now, we compute them directly via

$$v_{k^*,i} := \Big(\mathsf{com}_{\mathsf{KZG},k^*} \cdot g_1^{-f_{k^*}(z_{k*,i})} h_1^{-\hat{f}_{k^*}(z_{k*,i})} \Big)^{\frac{1}{\alpha - z_{k^*,i}}},$$

where we used that $\alpha \neq z_{k^*,i}$, see $\mathbf{G}_1$. It is easy to see that the distribution of the $v_{k^*,i}$ does not change, and so we get

$$\Pr[\mathbf{G}_6 \Rightarrow 1] = \Pr[\mathbf{G}_7 \Rightarrow 1].$$

The reader should observe the consequence of this change: we no longer need $f_{k^*}(\alpha)$ to simulate $\mathsf{com}_{\mathsf{KZG},k^*}$ and its openings. Instead, it is sufficient to know $g_1^{f_{k^*}(\alpha)}$. We will use this insight in the following to bound the probability that $\mathbf{G}_7$ outputs 1 using a reduction $\mathcal{B}$ against DLOG. The reduction is as follows:

1. $\mathcal{B}$ gets as input group parameters and $X^* = g_1^{x^*}$. Its goal is to find x^*.
2. $\mathcal{B}$ samples $k^* \leftarrow\!\!\$\, [Q_C]$ and computes a commitment key ck as in $\mathbf{G}_7$ honestly. Especially, $\mathcal{B}$ knows α and β.
3. $\mathcal{B}$ runs $\mathcal{A}$ with input ck and access to a commitment oracle GetCom and an opening oracle GetOp:
   - For $k \neq k^*$, $\mathcal{B}$ simulates GetCom and GetOp honestly as in $\mathbf{G}_7$. Especially, $\mathcal{B}$ knows all $f_k \in \mathbb{Z}_p[X]$ for $k \neq k^*$, and all $z_{k,i}$ for all k and i.
   - For the k^*th query to GetCom, $\mathcal{B}$ gets as input a message $\mathbf{m} \in \mathbb{Z}_p^\ell$. It does not sample δ_0, δ_1 to define f_{k^*} as in $\mathbf{G}_7$. Instead, it implicitly sets $f_{k^*}(\alpha) = x^*$ by defining $\mathsf{com}_{\mathsf{KZG},k^*} := X^* h_1^{\hat{f}_{k^*}(\alpha)}$. Then, it hashes $z_{k^*,0} := \mathsf{H}(\mathsf{com}_{\mathsf{KZG},k^*})$ and sets $f_{k^*}(z_{k^*,0})$ to be a random element in $\mathbb{Z}_p$, and $f_{k^*}(z_{k^*,i}) := \mathbf{m}_i$ for all $i \in [\ell]$. Note that f_{k^*} is fully defined now, and $\mathcal{B}$ knows $f_{k^*}(z_{k^*,i})$ for all $i \in \{0, \ldots, \ell\}$. Any opening $v_{k^*,i}$ for $\mathsf{com}_{\mathsf{KZG},k^*}$ is computed as explained in $\mathbf{G}_7$.
4. When $\mathcal{A}$ terminates and outputs its commitment, $\mathcal{B}$ computes a guess for $f_{k^*}(z_0)$ by solving the helper equation for $f_{k^*}(z_0)$. If this works and the set $\{z_0\} \cup \{z_{k^*,i}\}_{i=0}^{\ell}$ has size $\ell + 2$, then $\mathcal{B}$ interpolates f_{k^*} from the corresponding evaluations $f_{k^*}(z_0), f_{k^*}(z_{k^*,i})$ and outputs $f_{k^*}(\alpha)$.

First, it is clear that $\mathcal{B}$ simulates the commitment key ck and the commitment and opening oracles perfectly as in $\mathbf{G}_7$, and that $\mathcal{B}$'s running time is dominated by that of $\mathcal{A}$. While $\mathcal{B}$ can not efficiently check all bad events that we introduced throughout the games, we can still argue that if none of these events occurs, i.e., if $\mathbf{G}_7$ outputs 1, then $\mathcal{B}$'s output is x^*. To see this, note that if $\mathbf{G}_7$ outputs 1, then especially $c_{k^*}(z_0) \neq 0$ and so $\mathcal{B}$ can solve the helper equation to get $f_{k^*}(z_0)$. Further, by the change introduced in $\mathbf{G}_1$, we know that all $z_0, z_{k^*,i}$ are different, and so $\mathcal{B}$ knows $\ell + 2$ evaluations of f_{k^*}, which means it can correctly compute $f_{k^*}(\alpha) = x^*$. We get

$$\Pr[\mathbf{G}_7 \Rightarrow 1] \leq \mathsf{Adv}^{\mathsf{1\text{-}DLOG}}_{\mathcal{B},\mathsf{PGGen}}(\lambda).$$

□

4 Aggregatable Lotteries

In this section, we discuss how our lotteries are defined and how they can be constructed from our notion of vector commitments.

4.1 Definition of Aggregatable Lotteries

We formally present our ideal functionality $\mathcal{F}_{\text{lottery}}(p, T)$ for *non-interactive aggregatable lotteries* in Figures 5 and 6. It is parameterized by an upper bound on the winning probability p and the number of lotteries T. The probability p specifies how likely it is that a party wins in a lottery round. As described previously, our lotteries should intuitively prevent an adversary from winning the lotteries a disproportionate amount of times and the adversary should also not be able to tell which honest parties win the lotteries when. We do, however, allow the adversary to reduce its winning probability, i.e., the adversary can misbehave in a way that makes them win the lottery less often. We model this by allowing the adversary to specify their own winning probabilities for each lottery, capped at p, upon registration.

Our ideal functionality also relies on a party called the $\mathsf{Croupier}$, which we did not mention so far. It is in charge of initiating lottery rounds and registering participants. Note that this model allows the adversary to corrupt $\mathsf{Croupier}$, meaning that security is guaranteed even when the adversary can arbitrarily control the lottery, i.e., by registering players or initiating new lotteries.

Parties can be registered by Croupier via the Register interface. A lottery execution is run by Croupier via the NextLottery interface. Upon calling this interface, the functionality flips a biased coin for every currently registered party and stores whether they won the currently executed lottery. Parties can call the Participate interface to see whether they won a specific lottery. If they did, then they obtain a lottery ticket label, otherwise they receive nothing. Parties can call the Aggregate interface with a set of winning ticket labels and party identifiers to obtain an aggregate ticket label. Lastly, the Verify interface takes an aggregate ticket label and the corresponding winning parties' identifiers as input and checks whether the ticket is valid.

Functionality $\mathcal{F}_{\text{lottery}}(p, T)$

The functionality for T lotteries with winning probability p interacts with parties Player_j and a dedicated party $\mathsf{Player}_{\mathsf{Croupier}}$. It also interacts with the ideal world adversary Sim.

Interface Initialize:

```
$\mathsf{Tickets} := \emptyset$,  $\mathsf{AggTickets} := \emptyset$
$\mathsf{Win} := \emptyset$,  $\mathsf{Registered} := \emptyset$,  $\mathsf{LotCnt} := 1$
```

Interface Register: On $(\text{Register}, j, \mathsf{pid}) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_{\mathsf{Croupier}}$

```
// Check if pid already registered
if $\exists(\_, \mathsf{pid}, \_) \in \mathsf{Registered}$ : return
$\mathsf{Sim} \overset{\mathsf{send}}{\hookleftarrow} (\text{Register}, j, \mathsf{pid})$

// Corrupted parties can win with smaller probability
if $\mathsf{Player}_j$ is corrupted:
    $\mathbf{p} \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_j$
    for $t \in [T]$ : $\mathbf{p}_t := \min\{\mathbf{p}_t, p\}$
else :
    $\mathbf{p} := (p, \dots, p) \in \{p\}^T$

// Add to registrations
$\mathsf{Registered} := \mathsf{Registered} \cup \{(j, \mathsf{pid}, \mathbf{p})\}$
$\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} (\text{Register}, j, \mathsf{pid})$
```

Interface NextLottery: On $(\text{NextLottery}) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_{\mathsf{Croupier}}$

```
// Obtain a label for this lottery
if $\mathsf{LotCnt} > T$ : return
$l \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Sim}$

// Sample lottery outputs for all registered parties
for $(j, \mathsf{pid}, \mathbf{p}) \in \mathsf{Registered}$ :
    $w \leftarrow \mathcal{B}(\mathbf{p}_{\mathsf{LotCnt}})$  // Sample from Bernoulli distribution
    if $w = 1$ : $\mathsf{Win} := \mathsf{Win} \cup \{(l, j, \mathsf{pid})\}$
    $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} (\text{NextLottery}, \mathsf{pid}, l, w)$

// Advance lottery counter
$\mathsf{LotCnt} := \mathsf{LotCnt} + 1$
```

Figure 5: Ideal functionality $\mathcal{F}_{\text{lottery}}(p, T)$ for an T-time aggregatable lottery with winning probability p. The remaining interfaces are given in Figure 6.

4.2 Our Construction

Let us now proceed with our construction of aggregatable lotteries from vector commitments. For that, let $\mathsf{VC} = (\mathsf{VC.Setup}, \mathsf{VC.Com}, \mathsf{VC.VerCom}, \mathsf{VC.Open}, \mathsf{VC.Aggregate}, \mathsf{VC.Ver})$ be a vector commitment scheme according to Definition 3. Let $T, k \in \mathbb{N}$ be arbitrary parameters. We construct a T-time aggregatable lottery with winning probability $p = 1/k$ using a random oracle $\mathsf{H}\colon \{0,1\}^* \to [k]$. The main idea is that each player commits to a vector $\mathbf{v} \in [k]^T$ upfront, and wins the ith lottery if and only if its *personal challenge* x is equal to $\mathbf{v}_i$. Crucially, the challenge x has to be independent for different players and over different lotteries, and should be unpredictable before the lottery seed lseed is known. Thus, we

Functionality $\mathcal{F}_{\text{lottery}}(p, T)$ (continued)

Interface Participate: On $(\textsc{Participate}, l, \mathsf{pid}) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_j$

```
// Obtain fresh ticket label from Sim
$\mathsf{Sim} \overset{\mathsf{send}}{\hookleftarrow} (\textsc{Participate}, l, \mathsf{pid})$
$\mathsf{ticket} \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Sim}$

// If player won add ticket to table
if $(l, \mathsf{pid}, j) \in \mathsf{Win}$ :
    $\mathsf{Tickets} := \mathsf{Tickets} \cup \{(l, \mathsf{pid}, \mathsf{ticket})\}$
    $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \mathsf{ticket}$
else $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \bot$
```

Interface Aggregate: On $(\textsc{Aggregate}, l, I = (\mathsf{pid}_{j'})_{j' \in [L]}, T = (\mathsf{ticket}_{j'})_{j' \in [L]}\})$ from Player_j

```
// Check that the tickets are winning
for $(\mathsf{pid}, \mathsf{ticket}) \in T$ :
    if $(l, \mathsf{pid}, \mathsf{ticket}) \notin \mathsf{Tickets}$ :
        $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \bot$
        return

// Obtain aggregated ticket label from Sim
$\mathsf{Sim} \overset{\mathsf{send}}{\hookleftarrow} (\textsc{Aggregate}, l, I, T)$
$\mathsf{agg} \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Sim}$

// Store aggregated ticket label
$\mathsf{AggTickets} := \mathsf{AggTickets} \cup \{(l, \{\mathsf{pid} : \mathsf{pid} \in I\}, \mathsf{agg})\}$
$\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \mathsf{agg}$
```

Interface Verify: $(\textsc{Verify}, l, \mathsf{agg}, I) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_j$

```
// Only tickets generated by the functionality are valid
$\mathsf{Sim} \overset{\mathsf{send}}{\hookleftarrow} (\textsc{Verify}, l, \mathsf{agg}, I)$
if $(l, I, \mathsf{agg}) \in \mathsf{AggTickets}$ :
    $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \top$
else
    $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \bot$
```

Figure 6: Remaining interfaces of ideal functionality $\mathcal{F}_{\text{lottery}}(p, T)$ for T aggregatable lotteries with winning probability p. The other interfaces are given in Figure 5.

define $x := \mathsf{H}(\mathsf{pk}, \mathsf{pid}, i, \mathsf{lseed})$, where pid is the identifier of the player and pk is its public key, i.e., its commitment to $\mathbf{v}$.

Algorithms. To define our lottery protocol, we first define algorithms $\mathsf{Setup}, \mathsf{Gen}, \mathsf{VerKey}, \mathsf{Participate}$, $\mathsf{Aggregate}, \mathsf{Ver}$ that will be used in our protocol:

- $\mathsf{Setup}(1^\lambda) \to \mathsf{par}$:
  1. Run $\mathsf{ck} \leftarrow \mathsf{VC.Setup}(1^\lambda, 1^T)$. Recall that ck defines message alphabet $\mathcal{M}$, opening space $\mathcal{T}$, and commitment space $\mathcal{C}$. We assume that $[k] \subseteq \mathcal{M}$.
  2. Set and return $\mathsf{par} := \mathsf{ck}$.
- $\mathsf{Gen}(\mathsf{par}) \to (\mathsf{pk}, \mathsf{sk})$:
  1. Sample $\mathbf{v} \leftarrow\!\!\$\, [k]^T$ and run $(\mathsf{com}, St) \leftarrow \mathsf{VC.Com}(\mathsf{ck}, \mathbf{v})$.
  2. Set and return $\mathsf{pk} := \mathsf{com}$ and $\mathsf{sk} := (\mathbf{v}, St)$.
- $\mathsf{VerKey}(\mathsf{pk}) \to b$
  1. Return $b := \mathsf{VC.VerCom}(\mathsf{ck}, \mathsf{pk})$.
- $\mathsf{Participate}(t, \mathsf{lseed}, \mathsf{pid}, \mathsf{sk}) \to \mathsf{ticket}/\bot$:
  1. Set $x := \mathsf{H}(\mathsf{pk}, \mathsf{pid}, t, \mathsf{lseed})$. If $\mathbf{v}_i \neq x$, return $\bot$.
  2. Otherwise, set $\tau \leftarrow \mathsf{VC.Open}(\mathsf{ck}, St, i)$ and return $\mathsf{ticket} := \tau$.
- $\mathsf{Aggregate}(t, \mathsf{lseed}, (\mathsf{pid}_j, \mathsf{pk}_j)_{j=1}^L, (\mathsf{ticket}_j)_{j=1}^L) \to \mathsf{agg}$:
  1. For each $j \in [L]$, write $\mathsf{pk}_j = \mathsf{com}_j$ and $\mathsf{ticket}_j = \tau_j$.
  2. For each $j \in [L]$, set $x_j := \mathsf{H}(\mathsf{pk}_j, \mathsf{pid}_j, i, \mathsf{lseed})$.
  3. Return $\mathsf{agg} := \mathsf{VC.Aggregate}(\mathsf{ck}, t, (\mathsf{com}_j)_{j=1}^L, (x_j)_{j=1}^L, (\tau_j)_{j=1}^L)$.
- $\mathsf{Ver}(t, \mathsf{lseed}, (\mathsf{pid}_j, \mathsf{pk}_j)_{j=1}^L, \mathsf{agg} = \tau) \to b$:
  1. For each $j \in [L]$, write $\mathsf{pk}_j = \mathsf{com}_j$ and set $x_j := \mathsf{H}(\mathsf{pk}_j, \mathsf{pid}_j, t, \mathsf{lseed})$.
  2. Return $b := \mathsf{VC.Ver}(\mathsf{ck}, t, (\mathsf{com}_j)_{j=1}^L, (x_j)_{j=1}^L, \tau)$.

Protocol. We informally explain how to turn these algorithms into a protocol Π_{lottery} for aggregatable lotteries using a randomness beacon $\mathcal{F}_{\text{random}}$ (see Figure 8) and a broadcast channel $\mathcal{F}_{\text{broadcast}}$ (see Figure 7). Parties register for the lottery by running $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(\mathsf{par})$ and broadcasting their public key pk to other parties who verify it using VerKey. To commence the next lottery, the random beacon samples $\mathsf{lseed} \leftarrow\!\!\$\, \{0,1\}^\lambda$ and distributes it to all the parties, each party then locally computes a ticket ticket as $\mathsf{ticket} \leftarrow \mathsf{Participate}(t, \mathsf{lseed}, \mathsf{pid}, \mathsf{sk})$ where t is the index of the current lottery, to observe whether they obtained a winning ticket for the current lottery. These tickets can be verified and aggregated by any party. Given a set of tickets $(\mathsf{ticket}_j)_{j=1}^L$ a party can locally use $\mathsf{Aggregate}$ to compute an aggregated ticket agg, similarly it can locally use Ver to verify an aggregated ticket agg. The full formal formal description can be found in Figures 9 to 12.

Theorem 1. *Let $k, T \in \mathbb{N}$ be arbitrary and $p = 1/k$. Further, let* VC *be a simulation-extractable vector commitment scheme (see Definition 4). Then, the protocol $\Pi_{lottery}(p, T)$ realizes the functionality $\mathcal{F}_{lottery}(p, T)$ in the $(\mathcal{F}_{random}, \mathcal{F}_{broadcast})$-hybrid model against PPT adversaries for any polynomially bounded T and k.*

We give the proof of Theorem 1 in Appendix A.

Functionality $\mathcal{F}_{\text{broadcast}}$

The functionality for a totally ordered broadcast channel with n parties. It interacts with parties Player_j and the ideal world adversary Sim.

Interface Initialize:

```
$\mathsf{MsgReq} := \emptyset$
```

Interface Broadcast: On $(\textsc{Broadcast}, \mu) \overset{\text{recv}}{\hookleftarrow} \mathsf{Player}_j$

```
// Leak the message to the simulator
// Get fresh label for the message
$\mathsf{Sim} \overset{\text{send}}{\hookleftarrow} (\textsc{Broadcast}, \mu, j)$
$l \overset{\text{recv}}{\hookleftarrow} \mathsf{Sim}$
$\mathsf{MsgReq} := \mathsf{MsgReq} \cup \{(l, j, \mu)\}$
```

Interface Deliver: On $(\textsc{Deliver}, l) \overset{\text{recv}}{\hookleftarrow} \mathsf{Sim}$

```
// Deliver the next message to every player
if $\exists j', \mu$ s.t. $(l, j', \mu) \in \mathsf{MsgReq}$
  $\mathsf{MsgReq} := \mathsf{MsgReq} \setminus \{(l, j', \mu)\}$
  for each $\mathsf{Player}_j$ :
    $\mathsf{Player}_j \overset{\text{send}}{\hookleftarrow} (\textsc{Deliver}, j', \mu)$
```

Figure 7: Ideal functionality $\mathcal{F}_{\text{broadcast}}$ for broadcast.

Functionality $\mathcal{F}_{\text{random}}$

The functionality enabling the sampling of public randomness. The functionality interacts with parties Player_j and a special party $\mathsf{Player}_{\mathsf{Croupier}}$ which initiates the sampling of randomness.

Interface Random: On $(\textsc{Random}) \overset{\text{recv}}{\hookleftarrow} \mathsf{Player}_{\mathsf{Croupier}}$

```
$\mathsf{rnd} \leftarrow\!\!\$\ \{0,1\}^{\lambda}$
$\mathsf{Sim} \overset{\text{send}}{\hookleftarrow} (\textsc{Random}, \mathsf{rnd})$.
for each $\mathsf{Player}_j$ :
  $\mathsf{Player}_j \overset{\text{send}}{\hookleftarrow} (\textsc{Random}, \mathsf{rnd})$
```

Figure 8: Ideal functionality $\mathcal{F}_{\text{random}}$ for randomness beacon.

Protocol $\Pi_{\text{lottery}}(p, T)$

The protocol for T lotteries with winning probability p which is run parties Player_j and a dedicated party $\mathsf{Player}_{\mathsf{Croupier}}$. It uses the algorithms $\mathsf{Setup}, \mathsf{Gen}, \mathsf{VerKey}, \mathsf{Participate}, \mathsf{Aggregate}, \mathsf{Ver}$ defined in Section 4.2.

Interface Initialize:

```
// Generate global params for the lottery scheme
$\mathsf{par} \leftarrow \mathsf{Setup}(1^{\lambda})$

// Each player initializes their local state
Each player $\mathsf{Player}_j$ locally sets:
  $\mathsf{Player}_j.\mathsf{LotCnt} := 1$   // Local lottery counter
  $\mathsf{Player}_j.\mathsf{Registered} := \emptyset$   // Local allocation of key registrations
  $\mathsf{Player}_j.\mathsf{PidAlloced} := \emptyset$   // Local allocation of players to ids
  $\mathsf{Player}_j.\mathsf{Tickets} := \emptyset$   // Locally computed tickets
  $\mathsf{Player}_j.\mathsf{Keys} := \emptyset$   // Local secret keys
```

Figure 9: Initialization of protocol $\Pi_{\text{lottery}}(p, T)$ to implement $\mathcal{F}_{\text{lottery}}(p, T)$. The remaining parts of the protocol are given in Figures 10 to 12.

Protocol $\Pi_{\text{lottery}}(p, T)$

```
Interface Register: On $(\textsc{Register}, j, \mathsf{pid}) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_{\mathsf{Croupier}}$
// $\mathsf{Player}_{\mathsf{Croupier}}$ broadcasts the allocation $\mathsf{pid} \mapsto j$
$\mathcal{F}_{\text{broadcast}}.\mathsf{Player}_{\mathsf{Croupier}} \overset{\mathsf{send}}{\hookleftarrow} (\mathsf{ReqKey}, j, \mathsf{pid})$

Interface Register: On $(\textsc{Deliver}, \mathsf{Croupier}, \mu = (\mathsf{ReqKey}, j', \mathsf{pid})) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j$
// Ignore if id already allocated
if $\exists(\_, \mathsf{pid}) \in \mathsf{Player}_j.\mathsf{PidAlloced}$ : return
$\mathsf{Player}_j.\mathsf{PidAlloced} := \mathsf{Player}_j.\mathsf{PidAlloced} \cup (j', \mathsf{pid})$

// If addressed to this player, sample fresh key and broadcast
if $j' = j$ :
    $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(\mathsf{par})$
    $\mathsf{Player}_j.\mathsf{Keys} := \mathsf{Player}_j.\mathsf{Keys} \cup \{(\mathsf{pk}, \mathsf{sk})\}$   // Store the secret key locally
    $\mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} (\mathsf{SetKey}, \mathsf{pid}, \mathsf{pk})$   // Broadcast public key

Interface Register: On $(\textsc{Deliver}, j' \neq \mathsf{Croupier}, \mu = (\mathsf{SetKey}, \mathsf{pid}, \mathsf{pk})) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j$
// Check that the player is authorized
if $(j', \mathsf{pid}) \notin \mathsf{Player}_j.\mathsf{PidAlloced}$ : return
if $\exists(\mathsf{pid}, \_, \_) \in \mathsf{Player}_j.\mathsf{Registered}$ : return

// Check validity/uniqueness of the commitment
if $\exists(\_, \_, \mathsf{pk}) \in \mathsf{Player}_j.\mathsf{Registered}$ : $\mathsf{pk} := \bot$
if $\mathsf{VerKey}(\mathsf{par}, \mathsf{pk}) = 0$ : $\mathsf{pk} := \bot$

// Locally mark the player as registered
$\mathsf{Player}_j.\mathsf{Registered} :=$
      $\mathsf{Player}_j.\mathsf{Registered} \cup \{(\mathsf{pid}, \mathsf{Player}_j.\mathsf{LotCnt}, \mathsf{pk})\}$
```

Figure 10: Procedures of $\Pi_{\text{lottery}}(p, T)$ implementing the Register interface of $\mathcal{F}_{\text{lottery}}(p, T)$. The dedicated player $\mathsf{Player}_{\mathsf{Croupier}}$ broadcasts an "id allocation" (j, pid) on $\mathcal{F}_{\text{broadcast}}.\mathsf{Player}_{\mathsf{Croupier}}$, the addressed party Player_j samples a fresh key and broadcasts $(\mathsf{pid}, \mathsf{pk})$ on $\mathcal{F}_{\text{broadcast}}.\mathsf{Player}_{\mathsf{pid}}$. Upon receiving the public key, all honest parties store it (in $\mathsf{Registered}$) along with the pid and the index of the first lottery for which it is marked registered. The remaining parts of the protocol are given in Figures 9, 11 and 12.

Protocol $\Pi_{\text{lottery}}(p, T)$

```
Interface NextLottery: On $(\textsc{NextLottery}) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_{\mathsf{Croupier}}$
// Run $\mathsf{Croupier}$ request the next random output from the beacon
$\mathcal{F}_{\text{random}}.\mathsf{Player}_{\mathsf{Croupier}} \overset{\mathsf{send}}{\hookleftarrow} (\textsc{Random})$

Interface NextLottery: On $(\textsc{Random}, \mathsf{lseed}) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{random}}.\mathsf{Player}_j$
// Check if any lotteries left
if $\mathsf{Player}_j.\mathsf{LotCnt} > T$ : return

// Check for which $\mathsf{pid}$'s the party won
Define $l = (\mathsf{Player}_j.\mathsf{LotCnt}, \mathsf{lseed})$
for $(\mathsf{pid}, t, \mathsf{pk}) \in \mathsf{Player}_j.\mathsf{Registered} \wedge (\mathsf{pk}, \mathsf{sk}) \in \mathsf{Player}_j.\mathsf{Keys}$ :
    $\mathsf{ticket} \leftarrow \mathsf{Participate}(\mathsf{Player}_j.\mathsf{LotCnt}, \mathsf{lseed}, \mathsf{pid}, \mathsf{sk})$
    if $\mathsf{ticket} \neq \bot$ :
        $\mathsf{Player}_j.\mathsf{Tickets} = \mathsf{Player}_j.\mathsf{Tickets} \cup \{(l, \mathsf{pid}, \mathsf{ticket})\}$
        $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} (\textsc{NextLottery}, \mathsf{pid}, l, 1)$
    else
        $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} (\textsc{NextLottery}, \mathsf{pid}, l, 0)$
$\mathsf{Player}_j.\mathsf{LotCnt} := \mathsf{Player}_j.\mathsf{LotCnt} + 1$
```

Figure 11: Procedures of $\Pi_{\text{lottery}}(p, T)$ implementing the NextLottery interface of $\mathcal{F}_{\text{lottery}}(p, T)$. The remaining parts of the protocol are given in Figures 9, 10 and 12.

Protocol $\Pi_{\text{lottery}}(p, T)$ (local methods)

Interface PARTICIPATE: On $(\textsc{Participate}, l, \mathsf{pid}) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_j$

```
// Locally lookup the ticket
if $\exists(l, \mathsf{pid}, \mathsf{ticket}) \in \mathsf{Player}_j.\mathsf{Tickets} : \mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \mathsf{ticket}$
else $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \bot$
```

Interface AGGREGATE: On $(\textsc{Aggregate}, l = (t, \mathsf{lseed}), T, I) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_j$

```
// Lookup public keys, check registered before t
$P := \emptyset$
for $\mathsf{pid} \in I$ :
  if $\exists(\mathsf{pid}, t', \mathsf{pk}) \in \mathsf{Player}_j.\mathsf{Registered} \wedge t' \leq t : P := P \cup \{(\mathsf{pid}, \mathsf{pk})\}$
  else $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \bot$   // No such pid in lottery $t$
// Run the aggregation algorithm and return the result to $\mathsf{Player}_j$
$\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \mathsf{Aggregate}(t, \mathsf{lseed}, P, T)$
```

Interface VERIFY: $(\textsc{Verify}, l = (t, \mathsf{lseed}), \mathsf{agg}, I) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_j$

```
// Lookup public keys, check registered before t
$P := \emptyset$
for $\mathsf{pid} \in I$ :
  if $\exists(\mathsf{pid}, t', \mathsf{pk}) \in \mathsf{Player}_j.\mathsf{Registered} \wedge t' \leq t : P := P \cup \{(\mathsf{pid}, \mathsf{pk})\}$
  else $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \bot$   // No such pid in lottery $t$
// Run the verification algorithm and return the result to $\mathsf{Player}_j$
$\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \mathsf{Ver}(t, \mathsf{lseed}, P, \mathsf{agg})$
```

Figure 12: Protocol implementing the local operations (interfaces PARTICIPATE, AGGREGATE, VERIFY) of $\mathcal{F}_{\text{lottery}}(p, T)$. The remaining parts of the protocol are given in Figures 9 to 11.

5 Discussion and Efficiency

In the final section of our paper, we present some practical thoughts about our construction and evaluate its concrete efficiency.

5.1 Practical Considerations

In practice, one can make some natural adjustments to our lottery, which we discuss here.

Weighted Lotteries. One may assign a weight p_j to participant j (e.g., based on its stake) such that j wins independently with probability p_j. We can adjust our lottery to support this: we simply let a participant with weight $p_j = 1/k_j$ commit to vectors over the range $[k_i]$ instead of $[k]$, and let the hash function defining j's challenge $x_j = \mathsf{H}(\mathsf{pk}_j, \mathsf{pid}_j, i, \mathsf{lseed})$ map to the range $[k_i]$.

Late Registration. Consider the case of ℓ lotteries and assume that a user registers late, say after the ith and before the $(i+1)$st lottery. In the extreme case, we have $i = \ell - 1$. As written, the user would have to sample a random vector of length ℓ and commit to it, while only the coordinates from $i+1$ to ℓ would be relevant. After the ℓth lottery, the entire system restarts and the user would have to generate a new key and register again. This is wasteful. Fortunately, there are ways to deal with this: (1) the user could implicitly set the first i coordinates to 0, which makes committing much more efficient (in evaluation form, see below). A similar solution applies when the user only wants to take part in any small subset of lotteries; (2) the user could keep its key for the next lifecycle of the lottery system until after the ith lottery, where for the i'th lottery ($i' \leq i$) in the next lifecycle, it would use the i'th coordinate.

High Entropy. Our proof only relies on the fact that the lottery seed output by the randomness beacon has high entropy. It is actually not necessary that it is uniformly random.

Evaluation Form. When KZG [KZG10] is used as a vector commitment (as in our case), we should avoid explicitly computing the interpolating polynomial. Instead, we can compute the KZG commitments and openings more efficiently in the Lagrange basis. For that, we need to assume that the KZG setup contains elements $g_1^{\lambda_i(\alpha)}$, where λ_i is the ith Lagrange polynomial with respect to our evaluation domain. Note that this can be publicly pre-computed from a standard KZG setup. Optimal for efficiency is the case where we choose our evaluation domain to be the group of roots of unity and the polynomials we work with have degree d such that $d+1$ is a power of two, meaning that $\ell + 2$ has to be a power of two. In this case, we can benefit from several tricks to compute commitments and openings efficiently [Fei21]. We emphasize that this changes the way we compute commitments and openings, but not their final value, meaning that this has no negative impact on security.

Pre-computing Openings. Note that for participants of a lottery, the most time-critical part is not key generation, but rather the time it takes to participate and compute winning tickets (algorithm $\mathsf{Participate}$). In our scheme, a winning ticket is just a KZG opening proof within our evaluation domain. While computing a single proof takes linear time in ℓ (the number of lotteries), we can instead pre-compute all KZG opening proofs right after key generation and before lotteries take place. This can be done efficiently [FK23].

5.2 Efficiency Evaluation

With these considerations in mind, we evaluate the efficiency of our aggregatable lottery scheme $\mathsf{Jackpot}$. We focus on communication/storage and computation costs.

Lottery Schemes. We have implemented the following lottery schemes in Rust using the arkworks[4] framework. $\mathsf{VRF\text{-}BLS}$: The VRF-based lottery [GHM+17, DGKR18] instantiated with BLS signatures [BLS01] over curve BLS12-381 and SHA-256; more precisely, a party with secret key sk wins in lottery i with seed lseed if $\mathsf{H}(\sigma) < t$ for appropriate $t = t(k)$, where σ is a BLS signature of i and lseed and H is a hash function; σ is the winning ticket; To verify L tickets, we use BLS batch verification and L individual hash operations; Note that batch verification is only possible if all parties sign the same message, which is why can not include the party's identifier in the signed message. This also means that two parties with the same public key do not win independently, which has to be taken care of by other means. $\mathsf{Jackpot}$: Our lottery scheme, using curve BLS12-381 to implement KZG; we follow the practical considerations

[4] http://arkworks.rs

Tickets L	VRF-BLS [B]	Jackpot [B]	Ratio VRF-BLS/Jackpot
1	48	80	0.6
16	768	80	9.6
256	12288	80	153.6
1024	49152	80	614.4
2048	98304	80	1228.8

Table 1: Comparison of the memory consumption for storing or communicating L winning tickets for Jackpot with VRF-BLS. Memory is given in Bytes, and the column "Ratio" describes the ratio between the two schemes. We do not count player identifiers and their public keys, as they have to be stored on registration independent of winning tickets.

		$L = 1$	$L = 16$	$L = 256$	$L = 1024$	$L = 2048$
Jackpot	Aggregate [ms]	0.038	0.390	2.377	6.899	14.242
Jackpot	Ver [ms]	1.413	1.959	3.948	8.875	15.422
VRF-BLS	Ver [ms]	1.663	2.990	7.959	19.010	33.838

Table 2: Benchmarked running times for aggregation and verification of L tickets for Jackpot and VRF-BLS. All times are given in milliseconds.

discussed above; we have also implemented the technique from [FK23] to optionally pre-compute all openings. For all of our benchmarks, we assume $k = 512$ and $\mathsf{lseed} \in \{0,1\}^{256}$. Our code is available at

https://github.com/b-wagn/jackpot.

Bandwidth and Memory Consumption. Public keys for Jackpot have size $2 \cdot 48 + 2 \cdot 32 = 160$ Bytes, whereas they have size 48 Bytes for VRF-BLS. Now, assume that L winning tickets have to be stored or communicated. For VRF-BLS, this requires a storage of (ignoring public keys and identifiers of winning parties) $48 \cdot L$ Bytes, whereas for Jackpot each ticket has size $48 + 32 = 80$ Bytes, but the L tickets can be aggregated into one. This means that for L tickets, the relative improvement of Jackpot in comparison to VRF-BLS is $48 \cdot L/80 = 0.6 \cdot L$, which is a significant improvement even for small L. Table 1 shows some example numbers.

System Setup. For the following benchmarks, we have used a Macbook Pro (2020) with an Apple M1 processor, 16 GB of RAM, and MacOS Ventura 13.4. We benchmark our Rust implementation using the Criterion benchmark crate[5].

Aggregation and Verification. As our first benchmark in terms of running time, we evaluate the running times of aggregation (for Jackpot) and verification (for Jackpot and VRF-BLS) for different numbers L of tickets in Table 2. The running time is independent of the total number of lotteries. For VRF-BLS, we use BLS batch verification to verify multiple tickets with only one pairing equation. In this way, both schemes require L many hash evaluations and one pairing equation. Remarkably, our results demonstrate a increase in efficiency by a factor of 2 for Jackpot in comparison to VRF-BLS. This advantage arises from the fact that BLS batch verification necessitates operations over $\mathbb{G}_2$, whereas Jackpot's verification exclusively relies on operations over $\mathbb{G}_1$.

Key Generation and Pre-computation. In Table 3, we evaluate the parts of our scheme Jackpot for which the running time and memory depend on the total number of lotteries ℓ. For VRF-BLS, the running time is independent of ℓ and there is no preprocessing, and thus VRF-BLS is not listed in this table. We focus on three parameter settings $\ell \approx 2^z$ for $z \in \{10, 15, 20\}$. The table also shows the lifetime of keys for such settings, assuming one lottery every 5 minutes. In this case, 2^{20} lotteries are sufficient for 10 years. For these three parameter sets, we evaluate the running time of key generation (algorithm Gen) and the pre-computation of all KZG openings (algorithm Precompute). The table also shows the memory consumption for storing the result of the pre-computation. Even for 2^{20} lotteries, running times and memory consumption remain within practical bounds. Especially, the pre-computation's duration

[5] https://github.com/bheisler/criterion.rs

Number of Lotteries Lifetime	$\ell \approx 2^{10}$ 3.5 days	$\ell \approx 2^{15}$ 4 months	$\ell \approx 2^{20}$ 10 years
Time Gen	14.83 ms	317.82 ms	8.27 s
Time Precompute	2.20 s	65.45 s	45 min
Memory Precompute	147 KB	5 MB	151 MB
Time Participate	1.26 μs	1.27 μs	1.31 μs
Time ComputeTicket	9.45 ms	215.80 ms	6.36 s

Table 3: Benchmark results of running times for our lottery scheme Jackpot for key generation (Gen), pre-computing all openings (Precompute), and participating (Participate and ComputeTicket) in a lottery for different numbers of lotteries $\ell = 2^z - 2$, $z \in \{10, 15, 20\}$. Lifetimes are estimated by assuming one lottery every 5 minutes.

of approximately 45 minutes is acceptable, given that it can be run in the background at the user's convenience, and it is a one-time task for the entire lifespan of a key.

Participation. To participate in a lottery round, a party determines whether it won, and it computes a winning ticket in case it did. While we combined these two tasks in our modeling (algorithm Participate), we separate them for our benchmarks (algorithms Participate and ComputeTicket, respectively). We show the results in Table 3. For all parameter sets, the running time of Participate (i.e., checking if the party won) is negligibly small, namely, within microseconds. The running time to compute a ticket scales linearly with ℓ, but even for 2^{20} lotteries it is within a practical range: waiting 7 seconds for a ticket is fine, as one can compute a ticket in our scheme even before the lottery round started. Also, assuming we did a pre-computation as discussed above, this cost is completely avoided.

References

[BB08] Dan Boneh and Xavier Boyen. Short signatures without random oracles and the SDH assumption in bilinear groups. *Journal of Cryptology*, 21(2):149–177, April 2008. (Cited on page 7.)

[BD84] Andrei Z. Broder and Danny Dolev. Flipping coins in many pockets (byzantine agreement on uniformly random values). In *25th FOCS*, pages 157–170. IEEE Computer Society Press, October 1984. (Cited on page 3, 4.)

[BDN18] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, pages 435–464. Springer, Heidelberg, December 2018. (Cited on page 5.)

[BEHG20] Dan Boneh, Saba Eskandarian, Lucjan Hanzlik, and Nicola Greco. Single secret leader election. In *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020*, pages 12–24. ACM, 2020. (Cited on page 4.)

[BG12] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 263–280. Springer, Heidelberg, April 2012. (Cited on page 4.)

[BGG$^+$18] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 565–596. Springer, Heidelberg, August 2018. (Cited on page 4.)

[BGLS03] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 416–432. Springer, Heidelberg, May 2003. (Cited on page 5.)

[BK14] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 421–439. Springer, Heidelberg, August 2014. (Cited on page 3, 4.)

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 514–532. Springer, Heidelberg, December 2001. (Cited on page 5, 33.)

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993. (Cited on page 8.)

[BZ17] Massimo Bartoletti and Roberto Zunino. Constant-deposit multiparty lotteries on bitcoin. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y. A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *FC 2017 Workshops*, volume 10323 of *LNCS*, pages 231–247. Springer, Heidelberg, April 2017. (Cited on page 3, 4.)

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001. (Cited on page 3, 6, 8.)

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. HyperPlonk: Plonk with linear-time prover and high-degree custom gates. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part II*, volume 14005 of *LNCS*, pages 499–530. Springer, Heidelberg, April 2023. (Cited on page 5.)

[CDG$^+$18] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 280–312. Springer, Heidelberg, April / May 2018. (Cited on page 8.)

[CF13] Dario Catalano and Dario Fiore. Vector commitments and their applications. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 55–72. Springer, Heidelberg, February / March 2013. (Cited on page 5.)

[CHYC05] Sherman SM Chow, Lucas CK Hui, Siu-Ming Yiu, and KP Chow. An e-lottery scheme using verifiable random function. In *International Conference on Computational Science and its Applications*, pages 651–660. Springer, 2005. (Cited on page 3.)

[DGKR18] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 66–98. Springer, Heidelberg, April / May 2018. (Cited on page 3, 4, 5, 33.)

[Fei21] Dankrad Feist. PCS multiproofs using random evaluation. `https://dankradfeist.de/ethereum/2021/06/18/pcs-multiproofs.html`, 2021. Accessed: 2023-09-28. (Cited on page 33.)

[FFK$^+$23] Antonio Faonio, Dario Fiore, Markulf Kohlweiss, Luigi Russo, and Michal Zajac. From polynomial IOP and commitments to non-malleable zksnarks. *IACR Cryptol. ePrint Arch.*, page 569, 2023. (Cited on page 5.)

[FHSZ22] Nils Fleischhacker, Gottfried Herold, Mark Simkin, and Zhenfei Zhang. Chipmunk: Better synchronized multi-signatures from lattices. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1109–1123. ACM, 2022. (Cited on page 5.)

[FK23] Dankrad Feist and Dmitry Khovratovich. Fast amortized KZG proofs. Cryptology ePrint Archive, Report 2023/033, 2023. `https://eprint.iacr.org/2023/033`. (Cited on page 33, 34.)

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018. (Cited on page 4, 8.)

[FSZ22] Nils Fleischhacker, Mark Simkin, and Zhenfei Zhang. Squirrel: Efficient synchronized multi-signatures from lattices. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 1109–1123. ACM Press, November 2022. (Cited on page 5.)

[GGH$^+$13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th FOCS*, pages 40–49. IEEE Computer Society Press, October 2013. (Cited on page 4.)

[GHM$^+$17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68. ACM, 2017. (Cited on page 3, 4, 5, 33.)

[GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 581–612. Springer, Heidelberg, August 2017. (Cited on page 5.)

[GRWZ20] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating proofs for multiple vector commitments. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 2007–2023. ACM Press, November 2020. (Cited on page 5, 8.)

[Ita83] Kazuharu Itakura. A public-key cryptosystem suitable for digital multisignature. *NEC research and development*, 71:1–8, 1983. (Cited on page 7.)

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, December 2010. (Cited on page 4, 6, 7, 16, 33.)

[LBM20] Bei Liang, Gustavo Banegas, and Aikaterini Mitrokotsa. Statically aggregate verifiable random functions and application to e-lottery. *Cryptography*, 4(4):37, 2020. (Cited on page 3.)

[Lib24] Benoît Libert. Simulation-extractable KZG polynomial commitments and applications to HyperPlonk. In *PKC 2024*, LNCS. Springer, Heidelberg, May 2024. to appear. (Cited on page 5.)

[LLNW16] Benoît Libert, San Ling, Khoa Nguyen, and Huaxiong Wang. Zero-knowledge arguments for lattice-based accumulators: Logarithmic-size ring signatures and group signatures without trapdoors. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 1–31. Springer, Heidelberg, May 2016. (Cited on page 5.)

[LPR22] Benoît Libert, Alain Passelègue, and Mahshid Riahinia. PointProofs, revisited. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part IV*, volume 13794 of *LNCS*, pages 220–246. Springer, Heidelberg, December 2022. (Cited on page 5.)

[LY10] Benoît Libert and Moti Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 499–517. Springer, Heidelberg, February 2010. (Cited on page 5.)

[MOR01] Silvio Micali, Kazuo Ohta, and Leonid Reyzin. Accountable-subgroup multisignatures: Extended abstract. In Michael K. Reiter and Pierangela Samarati, editors, *ACM CCS 2001*, pages 245–254. ACM Press, November 2001. (Cited on page 7.)

[MRV99] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *40th FOCS*, pages 120–130. IEEE Computer Society Press, October 1999. (Cited on page 4.)

[PSTY13] Charalampos Papamanthou, Elaine Shi, Roberto Tamassia, and Ke Yi. Streaming authenticated data structures. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 353–370. Springer, Heidelberg, May 2013. (Cited on page 5.)

[Sah99] Amit Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In *40th FOCS*, pages 543–553. IEEE Computer Society Press, October 1999. (Cited on page 5.)

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, January 1991. (Cited on page 5.)

[TAB$^+$20] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 45–64. Springer, Heidelberg, September 2020. (Cited on page 5.)

Appendix

A UC Proof of Our Lottery

Proof of Theorem 1. To prove the theorem, we have to describe a simulator (aka ideal world adversary) $\mathsf{Sim}(\Pi_{\text{lottery}})$ such that the ideal world with functionality $\mathcal{F}_{\text{lottery}}(p, T)$ and $\mathsf{Sim}(\Pi_{\text{lottery}})$ is indistinguishable from the real world with the protocol $\Pi_{\text{lottery}}(p, T)$ for any environment E. The simulator $\mathsf{Sim}(\Pi_{\text{lottery}})$ internally emulates functionalities $\mathcal{F}_{\text{random}}$ and $\mathcal{F}_{\text{broadcast}}$: maintaining the same state as the functionalities and interacts with the environment E via the leakage/influence ports like for the real functionalities. When a party is corrupted in $\mathcal{F}_{\text{lottery}}$, the corresponding party is corrupted in the emulated functionalities $\mathcal{F}_{\text{random}}/\mathcal{F}_{\text{broadcast}}$ and its emulated ports are observed/controlled by the enviroment via the corresponding leakage/influence ports exposed by the simulator. The simulator may inspect/alter the state of these emulated functionalities and for honest players emulates control of the player ports. To recap and provide context, recall that:

- For honest Player_j:
  - $\mathcal{F}_{\text{lottery}}.\mathsf{Player}_j$ is controlled by E
  - Emulated port $\mathcal{F}_{\text{random}}.\mathsf{Player}_j$ is controlled by $\mathsf{Sim}(\Pi_{\text{lottery}})$ internally.
  - Emulated port $\mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j$ is controlled by $\mathsf{Sim}(\Pi_{\text{lottery}})$ internally.
- For corrupt Player_j:
  - $\mathcal{F}_{\text{lottery}}.\mathsf{Player}_j$ is controlled by $\mathsf{Sim}(\Pi_{\text{lottery}})$.
  - Emulated port $\mathcal{F}_{\text{random}}.\mathsf{Player}_j$ is controlled by E via $\mathcal{F}_{\text{random}}.\mathsf{Infl}$.
  - Emulated port $\mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j$ is controlled by E via $\mathcal{F}_{\text{broadcast}}.\mathsf{Infl}$.

Now, we describe the behavior of the simulator $\mathsf{Sim}(\Pi_{\text{lottery}})$ for different events:

On Initialize :

```
    $\mathsf{LotCnt} := 1$    // Lottery counter
    $\mathsf{PidAlloced} := \emptyset$    // Set of registered pid's
    $\mathsf{Registered} := \emptyset$    // Set of registered commitments (any type)
    $(\mathsf{ck}, \mathsf{td}) \leftarrow \mathsf{TSetup}(1^\lambda, 1^T)$    // Trapdoor setup for commitment scheme
```

```
// Simulate the registration. When an honest $\mathsf{Player}_{\mathsf{Croupier}}$
// invokes $(\textsc{Register}, j, \mathsf{pid})$ emulate sending $(\mathsf{ReqKey}, j, \mathsf{pid})$.
// Regardless of whether $\mathsf{Player}_{\mathsf{Croupier}}$ is corrupted, when an honest
// party receives $(\textsc{Deliver}, \mathsf{Croupier}, \mu = (\mathsf{ReqKey}, j', \mathsf{pid}))$
// in the simulation, add $(j', \mathsf{pid})$ to the allocated ids, if addressed to
// an honest player sample a trapdoor commitment, if instead
// a corrupted party registers the key
// then extract from the commitment.
```

On $(\textsc{Register}, j, \mathsf{pid}) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{lottery}}.\mathsf{Leak}$:

```
    // $\mathsf{Player}_{\mathsf{Croupier}}$ is honest
    $\mathcal{F}_{\text{broadcast}}.\mathsf{Player}_{\mathsf{Croupier}} \overset{\mathsf{send}}{\hookleftarrow} (\mathsf{ReqKey}, j, \mathsf{pid})$
```

On $(\textsc{Deliver}, \mathsf{Croupier}, \mu = (\mathsf{ReqKey}, j', \mathsf{pid})) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j$

```
    // $\textit{Allocate the id to the player}$
    if $\exists(\_, \mathsf{pid}) \in \mathsf{PidAlloced}$ : return $\bot$
    $\mathsf{PidAlloced} := \mathsf{PidAlloced} \cup \{(j', \mathsf{pid})\}$

    // $\textit{Generate trapdoor commitment for honest party}$
    if $j' = j$:
        $(\mathsf{com}, St) \leftarrow \mathsf{VC.TCom}(\mathsf{td})$
        $\mathsf{St}[\mathsf{com}] := St$
        $\mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} (\mathsf{SetKey}, \mathsf{pid}, \mathsf{com})$
```

```
On $(\textsc{Deliver}, j' \neq \mathsf{Croupier}, \mu = (\mathsf{SetKey}, \mathsf{pid}, \mathsf{com})) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j$
    // $\textit{Ignore unauthorized registration or double registration}$
    if $\exists(j', \mathsf{pid}) \notin \mathsf{PidAlloced}$ : return
    if $\exists(\mathsf{pid}, \_, \_) \in \mathsf{Registered}$ : return

    // $\textit{Extract from corrupt keys received by honest players}$
    if $\mathsf{Player}_{j'}$ is corrupt:
        // If $\mathsf{com}$ is invalid or has already been registered, replace with$\bot$
        if $\mathsf{VC.VerCom}(\mathsf{ck}, \mathsf{com}) = 0 \vee \exists(\_, \_, \mathsf{com}) \in \mathsf{Registered}$ :
            $\mathsf{Registered} := \mathsf{Registered} \cup \{(\mathsf{pid}, \mathsf{LotCnt}, \bot)\}$
            Define $\mathbf{p} := \mathbf{0}$
            $\mathcal{F}_{\text{lottery}}.\mathsf{Player}_{j'} \overset{\mathsf{send}}{\hookleftarrow} \mathbf{p}$
            return
        // Otherwise extract
        $(\mathbf{m}, \varphi) \leftarrow \mathsf{VC.Ext}(\mathsf{td}, \mathsf{com})$
        if $\mathbf{m} = \bot$ : abort
        $\mathsf{MsgsExt}[\mathsf{com}] = \mathbf{m}$
        // Translate corrupt party behavior to winning probabilities
        for $i \in [T] : \mathbf{m}_i \in [k] : \mathbf{p}_i := p$ else $\mathbf{p}_i := 0$
        $\mathcal{F}_{\text{lottery}}.\mathsf{Player}_{j'} \overset{\mathsf{send}}{\hookleftarrow} \mathbf{p}$ (finish registration).

    // $\textit{Register}$ $\mathsf{pid}$ $\textit{from next lottery onwards}$
    $\mathsf{Registered} := \mathsf{Registered} \cup \{(\mathsf{pid}, \mathsf{LotCnt}, \mathsf{com})\}$
```

// *Corrupt* $\mathsf{Player}_{\mathsf{Croupier}}$ *invoking* $\textit{Random}$ *on the emulated*

// *random beacon is equivalent to the* $\mathsf{Player}_{\mathsf{Croupier}}$

// *inputting* $\textsc{NextLottery}$ *into* $\mathcal{F}_{lottery}$*:*

```
On $(\textsc{Random}) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{random}}.\mathsf{Player}_{\mathsf{Croupier}}$ when $\mathsf{Player}_{\mathsf{Croupier}}$ corrupt:

    $\mathcal{F}_{\text{lottery}}.\mathsf{Player}_{\mathsf{Croupier}} \overset{\mathsf{send}}{\hookleftarrow} (\textsc{NextLottery})$
```

// *When the next lottery is started, we sample the random beacon,*

// *invoke the* $\mathcal{F}_{lottery}$ *to learn the outcomes for the corrupt parties,*

// *then program the random oracle to ensure that their tickets*

// *win iff. they do in the ideal world*

```
On $(\textsc{NextLottery}) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{lottery}}.\mathsf{Leak}$:
    // Sample the outcome of the random beacon
    Sample $\mathsf{lseed} \leftarrow_{\$} \{0,1\}^{\lambda}$.
    Define label $l := (\mathsf{LotCnt}, \mathsf{lseed})$
    $\mathsf{LotCnt} := \mathsf{LotCnt} + 1$

    // Let $\mathcal{F}_{lottery}$ sample outcomes. At this point
    // the simulator learns outcomes for corrupted parties.
    Pass $l$ to $\mathcal{F}_{\mathrm{lottery}}$

    // Emulate $\mathcal{F}_{random}$:
    // the players learn the output of the random beacon
    Emulate $\mathcal{F}_{\mathrm{random}}$ with $\mathsf{lseed}$ as output.

On $(\textsc{NextLottery}, \mathsf{pid}, l = (t, \mathsf{lseed}), w) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\mathrm{lottery}}.\mathsf{Player}_j$ when $\mathsf{Player}_j$ is corrupt:
    // Make corrupted party win/lose as in the ideal world
    if $\exists(\mathsf{pid}, \_, \mathsf{com}) \in \mathsf{Registered} \wedge \mathsf{com} \neq \perp$:
        if $\mathsf{H}(\mathsf{com}, \mathsf{pid}, t, \mathsf{lseed})$ has already been queried: abort
        if $w = 1$ :
            Program $\mathsf{H}(\mathsf{com}, \mathsf{pid}, t, \mathsf{lseed}) := \mathsf{MsgsExt}[\mathsf{com}]_t$.
        if $w = 0$ :
            $R \leftarrow_{\$} [k] \setminus \{\mathsf{MsgsExt}[\mathsf{com}]_t\}$
            Program $\mathsf{H}(\mathsf{com}, \mathsf{pid}, t, \mathsf{lseed}) := R$.

// When an honest party invokes PARTICIPATE we always generate
// an opening for the trapdoor commitment. Recall that $\mathcal{F}_{lottery}$ only
// passes this to $\mathsf{Player}_j$ if the player won.

On $(\textsc{Participate}, l = (t, \mathsf{lseed}), \mathsf{pid}) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\mathrm{lottery}}.\mathsf{Leak}$:
    // Generate opening for honest party invoking PARTICIPATE
    if $\exists(\mathsf{pid}, \_, \mathsf{com}) \in \mathsf{Registered} \wedge \mathsf{com} \in \mathsf{St}$:
        $\mathbf{m}_t := \mathsf{H}(\mathsf{com}, \mathsf{pid}, t, \mathsf{lseed})$
        $\tau \leftarrow \mathsf{TOpen}(\mathsf{td}, \mathsf{St}[\mathsf{com}], t, \mathbf{m}_t)$
        return $\tau$
    return $\perp$

// The simulation of the remaining interfaces
// simply run the protocol $\Pi_{lottery}$

On $(\textsc{Aggregate}, l = (t, \mathsf{lseed}), I, T) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\mathrm{lottery}}.\mathsf{Leak}$:
    // Retrieve the public keys
    for $\mathsf{pid}_i \in I$ :
        if $\exists(\mathsf{pid}_i, t', \mathsf{com}) \in \mathsf{Registered} \wedge t' \leq t : \mathsf{com}_i = \mathsf{com}$
        else return $0$

    // Run the aggregation algorithm for honest party
    $\mathsf{agg} := \mathsf{Aggregate}(t, \mathsf{lseed}, (\mathsf{pid}_i, \mathsf{com}_i)_{i=1}^{L}, T)$
    return $\mathsf{agg}$

On $(\textsc{Verify}, l = (t, \mathsf{lseed}), \mathsf{agg}, I) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\mathrm{lottery}}.\mathsf{Leak}$:
```

```
// Retrieve the public keys for the pid's
for $\mathsf{pid}_i \in I$ :
  if $\exists(\mathsf{pid}_i, t', \mathsf{com}) \in \mathsf{Registered} \wedge t' \leq t : \mathsf{com}_i = \mathsf{com}$
  else return $0$

// Run the verification algorithm
$b := \mathsf{Ver}(t, \mathsf{lseed}, (\mathsf{pid}_i, \mathsf{com}_i)_{i=1}^{L}, \mathsf{agg})$
return $b$
```

To see that the distribution of the real protocol Π_{lottery} and the simulation $\mathcal{F}_{\text{lottery}} \diamond \mathsf{Sim}(\Pi_{\text{lottery}})$ are indistinguishable, consider the following sequence of hybrids starting from the ideal world $\mathcal{F}_{\text{lottery}} \diamond \mathsf{Sim}(\Pi_{\text{lottery}})$:

$\mathbf{H}_0$. The Ideal World: $\mathcal{F}_{\textbf{lottery}} \diamond \mathsf{Sim}(\Pi_{\textbf{lottery}})$ Simulation

Let $\mathrm{E}_{\mathrm{FailProgram}}$ be the event that the simulator attempts to program H at a point where it has already been queried by the enviroment. Note that the simulation of NextLottery aborts iff $\mathrm{E}_{\mathrm{FailProgram}}$ occurs. Claim:

$$\Pr\left[\mathrm{E}_{\mathrm{FailProgram}}\right] \leq T \cdot \mathsf{poly}(\lambda)/2^{\lambda},$$

where $\mathsf{poly}(\lambda)$ is the running time of the enviroment. To see this observe that in the simulation lseed is chosen uniformly at random from $\{0,1\}^{\lambda}$ and unknown to the enviroment before the point of programming. Therefore, for $\mathrm{E}_{\mathrm{FailProgram}}$ to occur the enviroment must have queried H on some input $q = (\mathsf{com}, \mathsf{pid}, t, \mathsf{lseed})$ prior to $\mathsf{lseed} \leftarrow\!\!\mathrm{s}\, \{0,1\}^{\lambda}$ being sampled. In each lottery the probability that lseed matches with *any* previous query $q = (\mathsf{com}, \mathsf{pid}, t, \mathsf{lseed})$ made by the enviroment is at most $\mathsf{poly}(\lambda)/2^{\lambda}$; since the enviroment makes at most $\mathsf{poly}(\lambda)$ queries. A union bound over all T lotteries yields the claim.

$\mathbf{H}_1$. Don't Sample Lottery Outputs for Corrupted Players.

Hybrid $\mathbf{H}_1$ is the same as hybrid $\mathbf{H}_0$, except during NextLottery rather than having $\mathcal{F}_{\text{lottery}}$ sample the bits w from a Bernoulli distribution and programming the random oracle, the simulator computes the winning bit for *corrupted players* by checking the output of the random beacon against the extracted value $\mathbf{m}$. The behavior for honest players is unchanged, the important changes are highlighted:

On $(\textsc{Register}, j, \mathsf{pid}) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{lottery}}.\mathsf{Player}_{\mathsf{Croupier}}$

```
// Check if pid already registered
if $\exists(\_, \mathsf{pid}, \_) \in \mathcal{F}_{\text{lottery}}.\mathsf{Registered}$ : abort
$\mathsf{Sim} \overset{\mathsf{send}}{\hookleftarrow} (\textsc{Register}, j, \mathsf{pid})$
// Ignore the probability vector from corrupted parties
if $\mathsf{Player}_j$ is corrupted:
  Ignore $\mathbf{p}$ from $\mathsf{Player}_j$
// Add to registrations (but don't store $\mathbf{p}$ inside $\mathcal{F}_{lottery}$)
$\mathcal{F}_{\text{lottery}}.\mathsf{Registered} := \mathcal{F}_{\text{lottery}}.\mathsf{Registered} \cup \{(j, \mathsf{pid}, \perp)\}$
$\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} (\textsc{Register}, j, \mathsf{pid})$
```

On $(\textsc{NextLottery}) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_{\mathsf{Croupier}}$

```
// Obtain a label for this lottery
if $\mathsf{LotCnt} > T$ : abort
Obtain $l = (\mathsf{LotCnt}, \mathsf{lseed})$ from $\mathsf{Sim}$
// Sample lottery outputs for all registered parties
```

```
for $(j, \mathsf{pid}, \mathbf{p}) \in \mathcal{F}_{\text{lottery}}.\mathsf{Registered}$ :
  if $\mathsf{Player}_j$ is corrupt :
    if $\mathsf{H}(\mathsf{com}, \mathsf{pid}, \mathsf{LotCnt}, \mathsf{lseed}) = \mathsf{MsgsExt}[\mathsf{com}]_{\mathsf{LotCnt}}$ :
      $w := 1$
    else
      $w := 0$
  if $\mathsf{Player}_j$ is honest :
    $w \leftarrow \mathcal{B}(p)$
  if $w = 1$ :
    $\mathsf{Win} := \mathsf{Win} \cup \{(l, j, \mathsf{pid})\}$
  $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} (\textsc{NextLottery}, \mathsf{pid}, l, w)$
// Advance lottery counter
$\mathsf{LotCnt} := \mathsf{LotCnt} + 1$
```

$\underline{\text{On } (\textsc{NextLottery}, \mathsf{pid}, l = (t, \mathsf{lseed}), w) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{lottery}}.\mathsf{Player}_j\text{:}}$

```
// Do nothing (do not program the random oracle $\mathsf{H}$ anymore)
```

Indistinguishability. Conditioned on $\neg \mathrm{E}_{\text{FailProgram}}$, the distribution of w is equal in hybrid $\mathbf{H}_0$ and hybrid $\mathbf{H}_1$:

- From the definition of $\mathcal{F}_{\text{lottery}}$ in $\mathbf{H}_0$ it is clear that:

$$\Pr[w = 1] = \mathbf{p}_t$$

- In $\mathbf{H}_1$: Since none of the inputs $(\mathsf{com}, \mathsf{pid}, t, \mathsf{lseed})$ has been queried by E the outputs $m'_t = \mathsf{H}(\mathsf{com}, \mathsf{pid}, t, \mathsf{lseed})$ for distinct pid's are sampled uniformly and independently in the view of the enviroment E. Therefore any value in $m'_t \in [k]$ is equiprobable with probability $1/k$. Thus, if $\mathbf{m}_t \in [k]$, $\Pr[w = 1] = 1/k$, otherwise $\Pr[w = 1] = 0$ – which is how $\mathbf{p}_t$ is defined during Register in $\mathbf{H}_0$.

Therefore, the statistical distance between the distribution of w in $\mathbf{H}_0$ and $\mathbf{H}_1$ is at most $\Pr[\mathrm{E}_{\text{FailProgram}}] = T \cdot \mathsf{poly}(\lambda)/2^\lambda$, where $\mathsf{poly}(\lambda)$ is the running time of the enviroment.

$\mathbf{H}_2$. Don't Sample Lottery Outputs for Honest Players.

Hybrid $\mathbf{H}_2$ is the same as hybrid $\mathbf{H}_1$, except rather than $\mathcal{F}_{\text{lottery}}$ sampling $w \leftarrow \mathcal{B}(p)$ for *honest players*, the simulator picks $\mathbf{m} \leftarrow\!\!\$\, [k]^T$ independent of the commitment upon registration, checks whether the honest player wins checking if $\mathsf{H}(\mathsf{com}, \mathsf{pid}, t, \mathsf{lseed}) = \mathbf{m}_t$ and then defines the bit w correspondingly, i.e., the following changes are made:

The registration simulation is changed to sample $\mathbf{m}$ for honest players:

$\underline{\text{On } (\textsc{Deliver}, \mathsf{Croupier}, \mu = (\mathsf{ReqKey}, j', \mathsf{pid})) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j}$

```
// Allocate the id to the player
$\mathsf{PidAlloced} := \mathsf{PidAlloced} \cup \{(j', \mathsf{pid})\}$
// Generate trapdoor commitment for honest party
if $j' = j$:
  $\mathbf{m} \leftarrow\!\!\$\, [k]^T$
  $(\mathsf{com}, St) \leftarrow \mathsf{VC.TCom}(\mathsf{ck})$
  $\mathsf{Msgs}[\mathsf{com}] := \mathbf{m}$
  $\mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} (\mathsf{SetKey}, \mathsf{pid}, \mathsf{com})$
```

$\mathcal{F}_{\text{lottery}}$ is changed to check $\mathsf{H}(\mathsf{com}, \mathsf{pid}, t, \mathsf{lseed})$ against $\mathbf{m}$:

$\underline{\text{On } (\textsc{NextLottery}) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_{\mathsf{Croupier}}}$

```
// Obtain a label for this lottery
if $\mathsf{LotCnt} > T$ : return
Obtain $l = (\mathsf{LotCnt}, \mathsf{lseed})$ from Sim
// Sample lottery outputs for all registered parties
for $(j, \mathsf{pid}, \mathbf{p}) \in \mathcal{F}_{\text{lottery}}.\mathsf{Registered}$ :
  if $\mathsf{Player}_j$ is corrupt :
    if $\mathsf{H}(\mathsf{com}, \mathsf{pid}, \mathsf{LotCnt}, \mathsf{lseed}) = \mathsf{MsgsExt}[\mathsf{com}]_{\mathsf{LotCnt}}$ :
      $w := 1$
    else
      $w := 0$
  if $\mathsf{Player}_j$ is honest :
    if $\mathsf{H}(\mathsf{com}, \mathsf{pid}, t, \mathsf{lseed}) = \mathsf{Msgs}[\mathsf{com}]_{\mathsf{LotCnt}}$ :
      $w := 1$
    else
      $w := 0$
  if $w = 1$ : $\mathsf{Win} := \mathsf{Win} \cup \{(l, j, \mathsf{pid})\}$
  $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} (\textsc{NextLottery}, \mathsf{pid}, l, w)$
// Advance lottery counter
$\mathsf{LotCnt} := \mathsf{LotCnt} + 1$
```

Indistinguishability. Honest parties clearly win independently with probability $p = 1/k$ in each lottery in both hybrids.

$\mathbf{H}_3$. Rewrite $\mathbf{H}_2$ Using the Definition of SIM-EXT$_{\mathsf{VC},\mathsf{Sim},\mathsf{Ext},1}$.

The hybrid $\mathbf{H}_3$ is exactly the same as hybrid $\mathbf{H}_2$, except rewritten using the oracles in the simulation-extractability game (Definition 4) when the challenge bit $b = 1$. For clarity, the rewritten simulator is provided below:

On $\textsc{Initialize}$:

```
$\mathsf{LotCnt} := 1$
$\mathsf{PidAlloced} := \emptyset$
$\mathsf{Registered} := \emptyset$
```

On $(\textsc{Register}, j, \mathsf{pid}) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{lottery}}.\mathsf{Leak}$

```
// $\mathsf{Player}_{\mathsf{Croupier}}$ is honest
$\mathcal{F}_{\text{broadcast}}.\mathsf{Player}_{\mathsf{Croupier}} \overset{\mathsf{send}}{\hookleftarrow} (\mathsf{ReqKey}, j, \mathsf{pid})$
```

On $(\textsc{Deliver}, \mathsf{Croupier}, \mu = (\mathsf{ReqKey}, j', \mathsf{pid})) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j$

```
// Allocate the id to the player
if $\exists(\_, \mathsf{pid}) \in \mathsf{PidAlloced}$ : return $\perp$
$\mathsf{PidAlloced} := \mathsf{PidAlloced} \cup \{(j', \mathsf{pid})\}$

// Use $\textsc{GetCom}_1$ oracle for honest player commitments
if $j' = j$:
  $\mathbf{m} \leftarrow\!\!\$\; [k]^T$
  $\mathsf{com} := \textsc{GetCom}_1(\mathbf{m})$.
  Input $(\mathsf{SetKey}, \mathsf{pid}, \mathsf{com})$ on emulated $\mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j$
```

```
On $(\textsc{Deliver}, j' \neq \mathsf{Croupier}, \mu = (\mathsf{SetKey}, \mathsf{pid}, \mathsf{com})) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j$
    // Ignore unauthorized registration or double registration
    if $\exists (j', \mathsf{pid}) \notin \mathsf{PidAlloced}$ : return
    if $\exists (\mathsf{pid}, \_, \_) \in \mathsf{Registered}$ : return

    // Submit commitments from corrupt players
    if $\mathsf{Player}_{j'}$ is corrupt :
        if $\textsc{SubCom}_1(\mathsf{com}) = 0 : \mathsf{com} := \bot$

    // Register pid from next lottery onwards
    $\mathsf{Registered} := \mathsf{Registered} \cup \{(\mathsf{pid}, \mathsf{LotCnt}, \mathsf{com})\}$
```

```
On $(\textsc{Random}) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{random}}.\mathsf{Player}_{\mathsf{Croupier}}$
  when $\mathsf{Player}_{\mathsf{Croupier}}$ is corrupt:
    Input $(\textsc{NextLottery})$ into $\mathcal{F}_{\text{lottery}}.\mathsf{Player}_{\mathsf{Croupier}}$
```

```
On $(\textsc{NextLottery}) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{lottery}}.\mathsf{Leak}$:
    // Emulate the random beacon
    Emulate $\mathcal{F}_{\text{random}}$ obtain $\mathsf{lseed}$ as output.
    $\mathsf{LotCnt} := \mathsf{LotCnt} + 1$

    // Compute the lottery outputs for all registered parties
    Pass $(\mathsf{LotCnt}, \mathsf{lseed})$ to $\mathcal{F}_{\text{lottery}}$
```

```
On $(\textsc{Participate}, l = (t, \mathsf{lseed}), \mathsf{pid}) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{lottery}}.\mathsf{Leak}$:
    // Generate opening for honest party using $\textsc{GetOp}_1$
    if $\exists (\mathsf{pid}, t', \mathsf{com}) \in \mathsf{Registered} \wedge t' \leq t$ :
        return $\textsc{GetOp}_1(\mathsf{com}, t)$
    return $\bot$
```

```
On $(\textsc{Aggregate}, l = (t, \mathsf{lseed}), I, T) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{lottery}}.\mathsf{Leak}$:
    // Retrieve the public keys
    for $\mathsf{pid}_i \in I$ :
        if $\exists (\mathsf{pid}_i, t', \mathsf{com}) \in \mathsf{Registered} \wedge t' \leq t$ :
            $\mathsf{com}_i = \mathsf{com}$
        else return $0$

    // Run the aggregation algorithm for honest party
    $\mathsf{agg} := \mathsf{Aggregate}(t, \mathsf{lseed}, (\mathsf{pid}_i, \mathsf{com}_i)_{i=1}^{L}, T)$
    return $\mathsf{agg}$
```

```
On $(\textsc{Verify}, l = (t, \mathsf{lseed}), \mathsf{agg}, I) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{lottery}}.\mathsf{Leak}$:
    // Retrieve the public keys for the pid's
    for $\mathsf{pid}_i \in I$ :
        if $\exists (\mathsf{pid}_i, t', \mathsf{com}) \in \mathsf{Registered} \wedge t' \leq t$ :
            $\mathsf{com}_i = \mathsf{com}$
        else return $0$

    // Use the $\textsc{SubOp}_1$ oracle to run verification
    for $i \in [L] : m_i := \mathsf{H}(\mathsf{com}_i, \mathsf{pid}_i, t, \mathsf{lseed})$.
    $b := \textsc{SubOp}_1(t, (\mathsf{com}_i)_{i=1}^{L}, (m_i)_{i=1}^{L}, \mathsf{agg})$.
    return $b$
```

Indistinguishability. Since the simulation is simply rewritten, indistinguishability is trivial: the two hybrids are identical except one is written with the algorithms in $\textbf{SIM-EXT}_{\mathsf{VC},\mathsf{Sim},\mathsf{Ext},1}$.

$\mathbf{H}_4$. Move to $b = 0$ using Simulation-Extractability of VC.

Hybrid $\mathbf{H}_4$ is the same as hybrid $\mathbf{H}_3$, except moving from $b = 1$ in the simulation extractability game to $b = 0$, i.e., replacing every instance of:

- $\textsc{GetCom}_1$ with $\textsc{GetCom}_0$.
- $\textsc{GetOp}_1$ with $\textsc{GetOp}_0$.
- $\textsc{SubCom}_1$ with $\textsc{SubCom}_0$.
- $\textsc{SubOp}_1$ with $\textsc{SubOp}_0$.

Indistinguishability. $\mathbf{H}_4$ and $\mathbf{H}_3$ are indistinguishable by the simulation-extractability of VC (Definition 4).

$\mathbf{H}_5$. Rewriting $\mathbf{H}_4$ Using the Definition of $\textbf{SIM-EXT}_{\mathsf{VC},0}$.

Hybrid $\mathbf{H}_5$ is exactly the same as hybrid $\mathbf{H}_4$ except expanding the definitions of the oracles in the simulation extractability game for the $b = 0$ case. In hybrid $\mathbf{H}_4$ the opening state for honest commitments resides inside the state of the $\textbf{SIM-EXT}_{\mathsf{VC},0}$ oracles, in $\mathbf{H}_5$ we simply remove this boundary. For clarity, the full rewrite of the simulation is again provided below:

```
On INITIALIZE :
    $\mathsf{LotCnt} := 1$
    $\mathsf{PidAlloced} := \emptyset$
    $\mathsf{Registered} := \emptyset$

On $(\textsc{Register}, j, \mathsf{pid}) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{lottery}}.\mathsf{Leak}$:
    // $\mathsf{Player}_{\mathsf{Croupier}}$ is honest
    Input $(\mathsf{ReqKey}, j, \mathsf{pid})$ on emulated $\mathcal{F}_{\text{broadcast}}.\mathsf{Player}_{\mathsf{Croupier}}$

On $(\textsc{Deliver}, \mathsf{Croupier}, \mu = (\mathsf{ReqKey}, j', \mathsf{pid})) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j$
    // Allocate the id to the player
    if $\exists(\_, \mathsf{pid}) \in \mathsf{PidAlloced}$ : return $\perp$
    $\mathsf{PidAlloced} := \mathsf{PidAlloced} \cup \{(j', \mathsf{pid})\}$

    // For honest players generate commitment as $\textsc{GetCom}_0$
    if $j' = j$:
        $\mathbf{m} \leftarrow\!\!\$ \, [k]^T$
        $(\mathsf{com}, St) \leftarrow \mathsf{VC.Com}(\mathsf{ck}, \mathbf{m})$
        $\mathsf{St}[\mathsf{com}] := St$
        $\mathsf{Msgs}[\mathsf{com}] := \mathbf{m}$
        Input $(\mathsf{SetKey}, \mathsf{pid}, \mathsf{com})$ on emulated $\mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j$

On $(\textsc{Deliver}, j' \neq \mathsf{Croupier}, \mu = (\mathsf{SetKey}, \mathsf{pid}, \mathsf{com})) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\text{broadcast}}.\mathsf{Player}_j$
    // Ignore unauthorized registration or double registration
    if $\exists(j', \mathsf{pid}) \notin \mathsf{PidAlloced}$ : return
    if $\exists(\mathsf{pid}, \_, \_) \in \mathsf{Registered}$ : return

    // Submit commitments from corrupt players
    if $\mathsf{Player}_{j'}$ is corrupt :
      if $\exists(\_, \_, \mathsf{com}) \in \mathsf{Registered} : \mathsf{com} := \bot$
      if $\mathsf{VC.VerCom}(\mathsf{ck}, \mathsf{com}) = 0 : \mathsf{com} := \bot$

    // Register pid from next lottery onwards
    $\mathsf{Registered} := \mathsf{Registered} \cup \{(\mathsf{pid}, \mathsf{LotCnt}, \mathsf{com})\}$

On $(\textsc{Random}) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\mathrm{random}}.\mathsf{Player}_{\mathsf{Croupier}}$
  where $\mathsf{Player}_{\mathsf{Croupier}}$ is corrupt:
    Input $(\textsc{NextLottery})$ into $\mathcal{F}_{\mathrm{lottery}}.\mathsf{Player}_{\mathsf{Croupier}}$

On $(\textsc{NextLottery}) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\mathrm{lottery}}.\mathsf{Leak}$:
    // Emulate the random beacon
    Emulate $\mathcal{F}_{\mathrm{random}}$ obtain $\mathsf{lseed}$ as output.
    $\mathsf{LotCnt} := \mathsf{LotCnt} + 1$

    // Compute the lottery outputs for all registered parties
    Pass $(\mathsf{LotCnt}, \mathsf{lseed})$ to $\mathcal{F}_{\mathrm{lottery}}$

On $(\textsc{Participate}, l = (t, \mathsf{lseed}), \mathsf{pid}) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\mathrm{lottery}}.\mathsf{Leak}$:
    // Generate opening for honest party using $\textsc{GetOp}_0$
    if $\exists(\mathsf{pid}, t', \mathsf{com}) \in \mathsf{Registered} \land t' \le t$ :
      return $\mathsf{VC.Open}(\mathsf{ck}, \mathsf{St}[\mathsf{com}], t)$

On $(\textsc{Aggregate}, l = (t, \mathsf{lseed}), I, T) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\mathrm{lottery}}.\mathsf{Leak}$:
    // Retrieve the public keys
    for $\mathsf{pid}_i \in I$ :
      if $\exists(\mathsf{pid}_i, t', \mathsf{com}) \in \mathsf{Registered} \land t' \le t$ :
        $\mathsf{com}_i = \mathsf{com}$
      else return $0$

    // Run the aggregation algorithm for honest party
    $\mathsf{agg} := \mathsf{Aggregate}(t, \mathsf{lseed}, (\mathsf{pid}_i, \mathsf{com}_i)_{i=1}^{L}, T)$
    return $\mathsf{agg}$

On $(\textsc{Verify}, l = (t, \mathsf{lseed}), \mathsf{agg}, I) \overset{\mathsf{recv}}{\hookleftarrow} \mathcal{F}_{\mathrm{lottery}}.\mathsf{Leak}$:
    // Retrieve the public keys for the pid's
    for $\mathsf{pid}_i \in I$ :
      if $\exists(\mathsf{pid}_i, t', \mathsf{com}) \in \mathsf{Registered} \land t' \le t$ :
        $\mathsf{com}_i = \mathsf{com}$
      else return $0$

    // Expand $\textsc{SubOp}_0$ in verification
    for $i \in [L] : m_i := \mathsf{H}(\mathsf{com}_i, \mathsf{pid}_i, t, \mathsf{lseed})$.
    $b := \mathsf{VC.Ver}(\mathsf{ck}, t, (\mathsf{com}_i)_{i=1}^{L}, (m_i)_{i=1}^{L}, \mathsf{agg})$.
    return $b$
```

Indistinguishability. Hybrid $\mathbf{H}_5$ is exactly the same as hybrid $\mathbf{H}_4$ except expanding the definitions of the oracles in the simulation extractability game for the $b = 0$ case.

$\mathbf{H}_6$. The Real-World: The $\Pi_{\mathbf{lottery}}$ Protocol.

Indistinguishability. To see that $\mathbf{H}_6$ and $\mathbf{H}_5$ are indistinguishable, make the following sequence of observations:

- Because the $\mathcal{F}_{\text{broadcast}}$ functionality delivers messages simultaneously to all honest parties, the individual sets $\mathsf{Player}_j.\mathsf{Registered}$ maintained by any honest party Player_j are identical at all times and equal to the global $\mathsf{Registered}$ in $\mathbf{H}_5$, i.e., for all pairs of honest players Player_j and $\mathsf{Player}_{j'}$:
$$\mathsf{Player}_j.\mathsf{Registered} = \mathsf{Player}_{j'}.\mathsf{Registered} = \mathsf{Registered}$$

- Because the $\mathcal{F}_{\text{random}}$ functionality delivers messages simultaneously to all honest parties, the individual counters $\mathsf{Player}_j.\mathsf{LotCnt}$ maintained by any honest party Player_j are identical at all times and equal to the global LotCnt in $\mathbf{H}_5$, i.e., for all pairs of honest players Player_j and $\mathsf{Player}_{j'}$:
$$\mathsf{Player}_j.\mathsf{LotCnt} = \mathsf{Player}_{j'}.\mathsf{LotCnt} = \mathsf{LotCnt}$$

- The definition of Verify in $\mathbf{H}_5$ is identical to $\mathbf{H}_6$, except that $\mathbf{H}_5$ uses a globally defined $\mathsf{Registered}$ and LotCnt. By the previous two claims this is equivalent to using the locally defined $\mathsf{Player}_j.\mathsf{Registered}$ in $\mathbf{H}_6$.

- The definition of Aggregate in $\mathbf{H}_5$ is identical to $\mathbf{H}_6$, since the only distinction is the use of a the global $\mathsf{Registered}$ as oppose to the local $\mathsf{Player}_j.\mathsf{Registered}$.

- Note that in hybrid $\mathbf{H}_5$ we still have the global maps Msgs and St. At this point, however, these do not need to be global as each Player_j only accesses parts of $\mathsf{St}/\mathsf{Msgs}$ that it also knows locally.

- The behavior upon $(\text{Deliver}, \mathsf{Croupier}, \mu = (\mathsf{ReqKey}, j', \mathsf{pid}))$ is exactly identical in $\mathbf{H}_5$ and $\mathbf{H}_6$ by inspection and the observations above.

- The behavior upon $(\text{Deliver}, j' \neq \mathsf{Croupier}, \mu = (\mathsf{SetKey}, \mathsf{pid}, \mathsf{com}))$ is exactly identical in $\mathbf{H}_5$ and $\mathbf{H}_6$ by inspection and the observations above.

- For participate observe that in $\mathbf{H}_6$ (Π_{lottery}) the output is computed as:

```
Set $x := \mathsf{H}(\mathsf{pk}, \mathsf{pid}, i, \mathsf{lseed})$. If $\mathbf{v}_i \neq x$, return $\bot$.
Otherwise, set $\tau \leftarrow \mathsf{VC.Open}(\mathsf{ck}, St, i)$ and return $\mathsf{ticket} := \tau$.
```

  Where as in the hybrid $\mathbf{H}_5$ the computation of the output is as follows:

```
// In modified $\mathcal{F}_{lottery}$ (hybrids $\mathbf{H}_1$ and $\mathbf{H}_2$):
if $\mathsf{H}(\mathsf{com}, \mathsf{pid}, t, \mathsf{lseed}) = \mathsf{Msgs}[\mathsf{com}]_{\mathsf{LotCnt}}$ :
  $w := 1$
else
  $w := 0$
if $w = 1$ : $\mathsf{Win} := \mathsf{Win} \cup \{(l, j, \mathsf{pid})\}$

// In modified $\mathsf{Sim}(\mathcal{F}_{lottery})$ (hybrids $\mathbf{H}_3$, $\mathbf{H}_4$ and $\mathbf{H}_5$):
if $(l, \mathsf{pid}, j) \in \mathsf{Win}$ :
  $\mathsf{ticket} := \mathsf{VC.Open}(\mathsf{ck}, \mathsf{St}[\mathsf{com}], t)$
  return ticket
else return $\bot$
```

  These two computations have identical behavior.

- For Aggregate in $\mathbf{H}_5$, observe that the simulation strategy is simply to execute Aggregate as defined Π_{lottery}: since the operation is completely local. Hence the behavior of $\mathbf{H}_5$ and $\mathbf{H}_6$ is trivially seen to be identical.

- For VERIFY, observe that the definition of VERIFY in $\mathbf{H}_5$ is identical to VERIFY in $\mathbf{H}_6$, except that $\mathbf{H}_5$ uses a globally defined $\mathsf{Registered}$. As argued above this is equivalent to using the locally defined $\mathsf{Player}_j.\mathsf{Registered}$ in $\mathbf{H}_6$.

Since the implementations of each interface are perfectly indistinguishable, it follows that $\mathbf{H}_5$ and $\mathbf{H}_6$ are perfectly indistinguishable.

□