FlexiRand: Output Private (Distributed) VRFs and Application to Blockchains

Aniket Kate
Supra Research/Purdue University, USA
aniket@purdue.edu

Easwar Vivek Mangipudi
Supra Research, USA
e.mangipudi@supraoracles.com

Siva Maradana
Indian Statistical Institute, India
msivakumar.1431@gmail.com

Pratyay Mukherjee
Supra Research, India
p.mukherjee@supraoracles.com

ABSTRACT

Web3 applications based on blockchains regularly need access to randomness that is unbiased, unpredictable, and publicly verifiable. For Web3 gaming applications, this becomes a crucial selling point to attract more users by providing credibility to the "random reward" distribution feature. A verifiable random function (VRF) protocol satisfies these requirements naturally, and there is a tremendous rise in the use of VRF services. As most blockchains cannot maintain the secret keys required for VRFs, Web3 applications interact with external VRF services via a smart contract where a VRF output is exchanged for a fee. While this smart contract-based plain-text exchange offers the much-needed public verifiability immediately, it severely limits the way the requester can employ the VRF service: the requests cannot be made in advance, *and* the output cannot be reused. This introduces significant latency and monetary overhead.

This work overcomes this crucial limitation of the VRF service by introducing a novel privacy primitive Output Private VRF (Pri-VRF) and thereby adds significantly more flexibility to the Web3-based VRF services. We call our framework FlexiRand. While maintaining the pseudo-randomness and public verifiability properties of VRFs, FlexiRand ensures that the requester alone can observe the VRF output. The smart contract and anybody else can only observe a blinded-yet-verifiable version of the output. We formally define Pri-VRF, put forward a practically efficient design, and provide provable security analysis in the universal composability (UC) framework (in the random oracle model) using a variant of one-more Diffie-Hellman assumption over bilinear groups.

As the VRF service, with its ownership of the secret key, becomes a single point of failure, it is realized as a distributed VRF with the key secret-shared across distinct nodes in our framework. We develop our distributed Pri-VRF construction by combining approaches from Distributed VRF and Distributed Oblivious PRF literature. We provide provable security analysis (in UC), implement it and compare its performance with existing distributed VRF schemes. Our distributed Pri-VRF only introduces a minimal computation and communication overhead for the VRF service, the requester, and the contract.

CCS CONCEPTS

• **Security and privacy** → *Public key (asymmetric) techniques.*

KEYWORDS

Verifiable Random Functions, Distributed VRFs, Privacy

1 INTRODUCTION

Randomness is a precious resource in computing. Its utility ranges from generating cryptographic keys to performing simulations to facilitating online gaming. With the gigantic rise of blockchain technology and Web3-based applications such as decentralized finance and GameFi [15, 21], the demand for reliable sources of randomness has increased enormously. In many of these applications involving multiple parties, it is important to ensure that the employed randomness is not predictable to, or not biased towards, any particular party. However, given that the secure on-chain randomness generation within a smart contract is inefficient, if not infeasible, for most blockchains, a natural approach is to delegate this to off-chain computation. Off-chain computations, nevertheless, must be verified on-chain to ensure the integrity of computation. Verifiable random functions (VRFs) enable such functionality.

A Verifiable Random Function, V is a keyed deterministic function which, on an input tag/string x, outputs a string $y = V_{sk}(x)$. The secret-key sk is selected uniformly at random. Intuitively, the VRF provides two main security guarantees: (i) *pseudorandomness*, which implies that, as long as the secret-key is hidden, the output is *indistinguishable* from a uniform random string; (ii) *verifiability*, which implies that given x, y and a proof π, anyone can *publicly verify* that y is indeed computed correctly as $V_{sk}(x)$ – such proof is produced using the secret-key sk. Thanks to these guarantees, VRFs are sought after in blockchains, online gaming, and online lotteries: the use of a VRF allows the service providers to demonstrate to anyone interested that they are running their services unbiasedly.

VRF Services via Smart Contract. A few firms [14, 47] in the blockchain industry offer VRF as a service for a fee, in that *VRF service* and a *randomness requester*, such as a gaming platform, communicate via a *smart contract*. Here, as shown in Figure 1, the requester makes a randomness request to the VRF service via a smart contract. The smart contract then forms an input tag (INP) of a specific format (for more details on the input formation, see Appendix A) and sends it to the VRF service. Upon receiving the response from the VRF service, the smart contract verifies the response, records the VRF output, invokes the callback function provided by the requester, and pays the VRF service.

We observe a couple of key practical issues with this approach: the VRF output appears on the public blockchain via the smart contract interaction immediately upon the protocol completion. This public nature of the VRF output puts significant restrictions on how the requester can employ it: (i) the requester cannot make its request in advance towards having the randomness ready when the

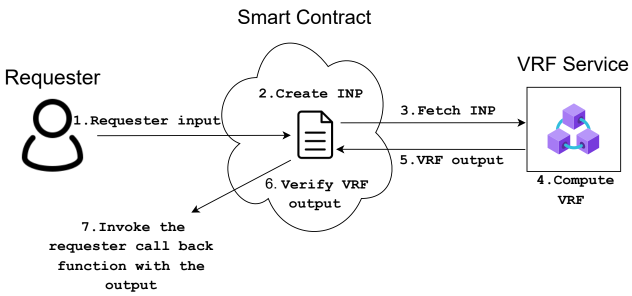

Figure 1: Flow of messages for computing (distributed) VRF via Smart Contract

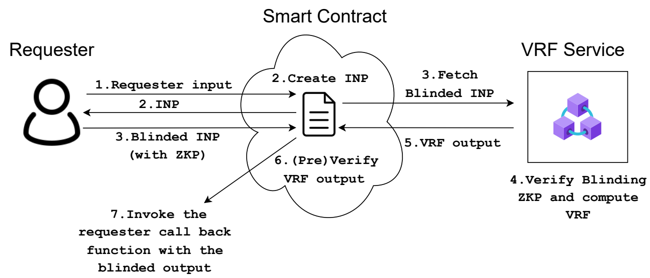

Figure 2: FlexiRand: Flow of messages for computing Output-private (distributed) VRF via Smart Contract

play begins. The request has to be *synchronized* with the application. As a result, the use of publicly verifiable external randomness introduces a significant *latency* overhead for the requester: It has to put the play on hold as it initiates and completes the VRF request. (ii) as the output is public, it *cannot be re-used* by the requester in the future (for example using a PRG to generate multiple random values to be used at different times when needed), which also results in significant overhead w.r.t. gas cost and VRF service fee as the requester has to make individual requests each time new randomness is required.[1] In a nutshell, this compels the requesting platforms to carefully design their games/services such that their players/clients cannot exploit the public VRF outputs, and furthermore, the latency and monetary overheads stay affordable. This limits the utility of smart-contract-based VRF services significantly.

Introducing Output-Private VRF. Towards overcoming the issues with the existing VRF services in the blockchain ecosystem, we introduce a new primitive called *Output-Private VRF* (Pri-VRF) and provide an efficient construction. The design is supported by *our provable security* analysis with respect to *our newly formalized definitions* in the universal composability (UC) framework [10]. In Pri-VRF, only the requester can obtain the output $y = V_{sk}(x)$. Everybody else can only see a blinded (a.k.a. masked) output, which only the requester can unblind. Crucially, anyone can still *publicly verify* that the requester's request was legitimate and ensure the legitimacy of the response (the final VRF output, when revealed, can still be verified as usual).

Output-private VRF allows the requester to overcome the above-mentioned restrictions as follows. As the public value is blinded, the requester can compute the necessary randomness *asynchronously* (ahead of time) to be used at any later point as needed – this resolves the first issue.[2] Furthermore, due to privacy, one may extend the private output $y = V_{sk}(x)$ to generate multiple pseudorandom values $z_1 = \mathrm{PRG}(y, 1), z_2 = \mathrm{PRG}(y, 2), \ldots$ using a pseudorandom generator. The randomnesses z_1, z_2 can be used (asynchronously) at a later point when needed. It thus can offer a cost-efficient randomness generation mechanism.[3]

Output-Private Distributed VRF. For VRFs, the computing node, which knows the secret key and computes the VRF output, becomes a *single point of failure* for secrecy as well as liveness: VRF outputs are completely predictable to this node and the VRF computation discontinues if the specific node crashes. Therefore, instead of using a centralized VRF, we can opt for a distributed VRF (DVRF), an extension of VRF in the decentralized setting.

In contrast to a centralized VRF, no single node has access to the entire secret key in the DVRF framework. In particular, the secret-key is shared among many parties (let us denote them by $P_1, P_2, \ldots, P_n$ and together call them the VRF committee), for example, using Shamir's secret sharing scheme [46], implemented using an appropriate Distributed Key-generation (DKG) protocol [29]. On an input x, each party P_i computes a *partial* evaluation-proof pair (y_i, π_i) using their shares of secret-key sk_i. An aggregator, who (possibly one of the servers in the VRF committee) may not hold any secret-key, can publicly gather $t+1 \leq n$ such partial evaluations to aggregate them into the final output y and an accompanying proof π – this procedure is public.

A t out of n distributed (threshold) procedure is, in fact, resilient to $f \leq t$ malicious corruptions, who may collude. Therefore, this setting, compared to the centralized setting, provides a number of enhanced guarantees: (i) *consistency*, which guarantees that, *any* $t + 1$ parties may collaborate to produce a unique and consistent output y; (ii) *robustness*, that ensures that if there is a wrongly computed partial evaluation, it must be detected before aggregation; (iii) *liveness* (alternatively *availability*), which ensures that corrupt parties can not prevent the output from being computed. Furthermore, the *pseudorandomness* guarantee is now much stronger as that must be achieved in the presence of $\leq t$ malicious corruptions.

We extend our Pri-VRF notion to the t out of n distributed setting, which we call Pri-DVRF – the extension is analogous to distributed VRFs, albeit with added privacy guarantee. In particular, in addition to the above guarantees, we need (iv) *output privacy*, even when $\leq t$ servers are malicious. So, each server now computes a partially blinded value, that are aggregated publicly (possible only if there are $\geq t + 1$ legitimate responses), and then the final blinded y is

[1]Chainlink's VRF service [14] actually supports generating multiple randomnesses by using the VRF output as a PRG seed. However, all randomnesses must be used together at the same time to remain unpredictable. This domain extension strategy maybe helpful in specific applications, but it does not address the issue fundamentally.

[2]We stress that the requester gains no advantage by obtaining output y early because of public verifiability with respect to a well-formatted (along with a timestamp) x, which was already made public. E.g., this ensures that the requester may not reject this y in favor of a y' as that requires querying with another x'.

[3]Let us stress that each such randomness z_i can only be verified using y. Given that one can compute all z_is, this approach can only be useful in a setting where the verifiability can be deferred to a later point when all z_i's were already used; after that, a new VRF request must be made.

sent to the requester, which then unblinds it to obtain y. All DVRF guarantees must hold on the blinded values. Our UC definition captures all these informal guarantees formally. Our design combines approaches from distributed VRFs (DVRF) [28] and Distributed Oblivious PRFs [35], and easy to extend from our centralized construction as well (we stress on the ease of decentralization while designing the centralized version).

Implementation. We implement the Pri-DVRF construction by extending the GLOW-DVRF [27, 28], written in C++. Our reference implementation indicates that the construction is highly practical, taking less than 0.5msec for the partial evaluation on each VRF node in a single-threaded implementation. The time taken is not too high compared to the non-private DVRF – 400μsec (vs 253μsec) for the BN256 curve using the mcl library [2]. The requester takes $\sim 300\mu$ sec for blinding and generating the zero-knowledge proof of correct blinding before forwarding it to the VRF service.

Contribution. This work is motivated by a contemporary real-world problem currently most visible in the blockchain game sector (see Section 2). In this work we adapt the existing techniques carefully in order to resolve that practically while providing rigorous theoretical analysis. We summarize our contributions here:

– We introduce the notions of (distributed) Output-private VRF that guarantees the privacy of the VRF output. Our formalization is based on UC framework and thus provides a strong security guarantee.

– We provide a Pri-VRF construction and a Pri-DVRF construction, both based on bilinear pairing. We give provable security analysis within our UC-based definitions. Our constructions borrow idea from the DVRF construction by Galindo et al. [28] and the Oblivious (D)PRF construction of Jarecki et al. [35]. We outline an enhanced smart contract based Pri-(D)VRF framework that we call FlexiRand, which incorporates the flexibility of our constructions.

– We show the practical efficiency of our constructions by providing simple implementation on top of GLOW-DVRF [28]; the VRF committee nodes incur an overhead of about 1.6x in computation time compared to GLOW-DVRF. This is a very reasonable trade-off compared to the benefits offered by FlexiRand.

– We provide a concrete real-world use-case where using FlexiRand instead of standard DVRF service is significantly beneficial. Our use-case stems from a real-world requirement of DeFi gaming platform such as DeFi Kingdoms [21], which crucially requires distributing random rewards (e.g. NFTs) in Blockchain Games [15].

2 USE CASE

In this section we describe a specific use-case, which is the primary motivation for this work. DeFi gaming (GameFi) platforms are increasingly becoming more popular in the Web3 world. They need access to randomness that is sensitive with respect to latency as well as cost. Precisely, cross-chain gaming platforms such as DeFi Kingdoms [21] attract customers by providing virtual assets (such as NFTs) to the players randomly at regular intervals within a game. Its philosophy is to offer the gamer an experience of "fun with surprises".

As explained in [15], a platform that engages in random reward distribution acquires significantly more credibility by providing publicly verifiable proof that the randomness is generated correctly. A smart-contract-based VRF service (as depicted in Figure 1) exactly provides that. Nevertheless, based on our discussion with DeFi Kingdoms [22], the service suffers from the aforementioned problems: (i) since the request can not be made in advance, there is a latency involved which is beyond the control of the platform; (ii) the overall fees to avail such a service become prohibitively expensive. In particular, they are interested in a service, in that the VRF output can be made in advance to avoid an unpredictable latency during the game and can be (re-)used as a seed to generate multiple random values over a period of time because a gamer is rewarded with a randomly chosen virtual asset after completing a few steps. Of course, one may think about re-using the VRF output, exposed on the contract, to generate random values, but then all those values are predictable defying the "surprise" aspect. Instead, in our framework, the VRF output remains hidden/blinded on-chain, and the verifiability can be deferred until all the random values, derived from a VRF request, are exhausted.

3 TECHNICAL OVERVIEW

Our framework and input formatting. In our non-private DVRF framework (cf. Figure 1), a requester sends a request to the smart contract, which then crafts an input INP carefully – the input is composed of many parameters, important for practical deployment of both DVRF framework and FlexiRand (for more details, we refer to Appendix A) and crucially prevent attacks as explained below. The input is sent to the VRF committee (consider the centralized version as a special case of DVRF where $n = 1$, $t = 0$). The servers interact to produce an output, which is then sent to the contract. The contract verifies the output and on success, forwards that to the requesting platform. The main change in the FlexiRand framework comes in the initial phase when the smart contract sends back the input INP to the requester (Step-2 of Figure 2), who then sends a blinded input (along with a NIZK proof of knowledge of exponent) to the contract. From this point onwards, the rest of the flow is pretty much the same, except that the contract now runs a verification over the blinded values.

Repeating input attack. One easy way to break the privacy might be to observe the input x, and then make the same request pretending to be the "owner" of x and legitimately derive y. The framework would prevent this by carefully crafting the input INP such that it has a component reflecting the owner's identity (for example, the requester's public key) – this can be checked at the server's end to avoid such an attack. This is incorporated in our construction by assuming a unique owner for input and is captured within our UC definition explicitly.

Our Pri-VRF construction. Our constructions combine techniques from the Oblivious PRF by Jarecki et al. [35] with the GLOW-DVRF by Galindo et al. [28]. We first describe how our Pri-VRF construction works. Consider a bilinear pairing group structure $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, each a cyclic group of prime order p. The VRF secret-key sk is chosen at random from $\mathbb{Z}_p$, whereas the public verification key vk is pair of group elements ($vk_1 = g_1^{sk}, vk_2 = g_2^{sk}$) where g_1, g_2 are random generators of groups $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. A requester with an input x first blinds her input to generate $\psi = \mathsf{H}_1(x)^\rho \in \mathbb{G}_1$ for a hash function $\mathsf{H}_1(\cdot)$, random blind/mask ρ in $\mathbb{Z}_p$ and produces a NIZK proof of knowledge of ρ (Schnorr's proof for knowledge of exponent [45]) and sends that over to the smart-contract, which forwards it to the VRF server. The server first verifies the NIZK proof, and if that succeeds, sends back $\widetilde{y} = (\mathsf{H}_1(x)^{\rho})^{sk}$. The contract verifies the response by using bilinear pairing $e(\psi, g_2^{sk}) = e(\widetilde{y}, g_2)$ (exactly the same as BLS signatures [8]) and if that succeeds, then it forwards $\widetilde{y}$ to the requester, who then unblinds to get $\pi = \widetilde{y}^{1/\rho} = \mathsf{H}_1(x)^{sk}$; then derives y as $\mathsf{H}_2(\pi)$. The final VRF verification is again running the BLS verification, but now with different components: $e(\mathsf{H}_1(x), g_2^{sk}) = e(\pi, g_2)$ plus the hash $\mathsf{H}_2(\pi) = y$. Note that there are three verifications in total: one for the requester's message, one for the server's message, and finally, one for the VRF triple – all of them can be done publicly. Furthermore, using these three verifications together, one could verify not only the output y is correct but the entire flow of communication with the input x is associated with the output y – this might be desirable in some applications.

Our Pri-DVRF construction. We extend our centralized solution to a t out of n setting by using a VRF committee consisting of n nodes, each of which holds a secret-key share sk_i of sk (this can be typically achieved through a distributed key-generation (DKG) protocol such as [30]). The verification key now is of the form $vk = (pk = g_2^{sk}, vk_1 = g_1^{sk_1}, vk_2 = g_1^{sk_i}, \ldots)$. The requester's steps are identical – in fact, the requester may be completely agnostic of whether a centralized or a decentralized service is being used (or what n, t are being used). The smart-contract now sends the blinded request to each server in the committee. So, once a VRF server within VRF committee receives a blinded request ψ along with x and a NIZK proof, each of them checks the proof as before, and if that succeeds, now uses its share sk_i to compute a partial value $\widetilde{w}_i = \psi^{sk_i}$. Additionally, it computes another NIZK for equality of exponent (we use Chaum-Pederson [16]) between $\widetilde{w}_i$ and vk_i. Then it sends over the partial evaluation $\widetilde{w}_i$ plus the proof to the aggregator, who verifies each NIZK proof, and if at least $t + 1$ of them succeeds, then combines the corresponding partial evaluations via Lagrange interpolation in the exponent to compute $\widetilde{y} = \mathsf{H}_1(x)^{\rho \cdot sk}$. Given $x, \psi, \widetilde{y}$, the contract verifies using the bilinear map and then sends that over to the requester on success. The overall computational overhead for the VRF committee servers is less than 2x compared to GLOW-DVRF, and is incurred due to the blinded-input NIZK verification by each server. Importantly, in our framework the smart-contract's work is exactly the same (a single pairing computation) and hence the gas cost remains the same.

Security Analysis. *Consistency* is guaranteed easily using Shamir's secret sharing. *Robustness* is guaranteed by the soundness of NIZK proof of equality computed during partial evaluations. Then we restrict our setting such that $n \geq 2t + 1$ – this, combined with *robustness* immediately gives *liveness*. The *pseudorandomness* of our constructions require that, if the server is not corrupt, no one else can predict the output y unless explicitly obtained from the server. For the distributed setting, the same should hold even if at most t servers are maliciously corrupt additionally. This is no different from the same scenario in the Jarecki et al.'s [35] oblivious PRF construction. So, our proof closely follows theirs and relies on a similar assumption, namely a variant of (threshold) one-more DH assumptions. However, since we are in bilinear pairing groups, we require a version that holds in a pairing source group. Nevertheless, in contrast to them, we do not need a gap version due to the presence of pairing. The *output-privacy* part is new to our setting and is carefully handled using the CDH assumption over bilinear groups (known as Co-CDH). Intuitively, this part works because of the unpredictability of $\mathsf{H}_1(x)^{sk}$, given g_1^{sk}, $\mathsf{H}_1(x)^{sk \cdot \rho}$ and $\mathsf{H}_1(x)^{\rho}$ (in the centralized case), which is somewhat similar to the proof of BLS signature unpredictability but requires more care due to exposure of many exponents of $\mathsf{H}_1(x)$. When the server is compromised (in the decentralized case, that is equivalent to corruption of $> t$ servers), then the only guarantee one may hope for is that the output y is correct, although not unpredictable – this is not hard to see because of the soundness of the NIZK proof (in the decentralized setting) or correctness of bilinear pairing (for centralized case). We model all hash functions as random oracles and carefully program them in the proofs.

Alternative approaches: Encryption plus NIZK. One way to generically convert any (D)VRF to a Pri-(D)VRF is to use (fully homomorphic) encryption and any non-interactive zero-knowledge proof (NIZK): the requester simply sends the input to the VRF committee (via the smart-contract) who computes the partial evaluations and provides a NIZK proof of correct evaluations. The aggregation can be done using homomorphism (for some constructions, additive homomorphism may suffice) plus by producing another succinct NIZK (such as SNARKs) of correct verification of at least $t + 1$ ciphertexts. While this may be a potential solution, the efficiency of this may be significantly worse than our approach. In particular, producing NIZK proof of a specific encryption scheme (even efficient ones such as ElGamal [32]) already adds significant overhead; on top of that, producing an aggregated proof during aggregation seems to incur even more computational inefficiency. Of course, this approach may be reasonably efficient (though still probably much behind our centralized version) in the centralized setting, but since we prefer a scheme that supports easy decentralization, we do not follow this.

Alternative without the bilinear pairing?. One may wonder whether the bilinear pairing is necessary here. In particular, what happens if we replace the pairing verification with a NIZK verification: the server would send π as above, plus a NIZK proof of the equality of exponent with g_1^{sk} (Chaum-Pederson's proof [45]) – basically adding output-privacy on top of [31]. The issue here is that the requester can not have a publicly verifiable triple (x, y, π), as the NIZK proof does not immediately support the "homomorphism" in bilinear pairing like above. Furthermore, such an approach would not be easy to decentralize because the NIZK proofs must be aggregated using, for example, a SNARK proof, leading to a significant challenge in terms of constructing an efficient SNARK for that specific language. In contrast, our approach is readily extendable into a decentralized setting.

Bilinear vs NIZK in Pre-verification. As shown by Galindo et al. [28], an alternative to using Chaum-Pederson's NIZK could be to use bilinear pairing for verifying server's response akin to our centralized construction. However, as shown in the same work, this would incur a computational overhead of about 2.5x compared to the NIZK proof – as in the case for Dfinity-DVRF vs GLOW-DVRF. This is because the NIZK proof works in the group $\mathbb{G}_1$ and supports faster operation than bilinear pairing. We remark that the idea of using Chaum-Pederson's proof for verifying the server's partial response during aggregation can also be incorporated in the centralized setting (albeit the final verification of (x, y, π) should still be done using pairing). However, in practice, the benefit is much less as only one pre-verification is done compared to at least $t + 1$ pre-verification performed in the distributed setting. So we choose to leave the centralized construction simple and use the NIZK-based optimization only in the distributed version (though it can be realized as a special case: $n = 1$, $t = 0$).

4 RELATED WORK

Verifiable Random Functions. The concept of VRF was introduced by Micali, Rabin and Vadhan [40]. They first noticed the similarities between VRFs and *unique* signatures (produces a unique signature for each message). Their construction is based on RSA signatures. Later, this was improved by the work of Dodis and Yampolskiy [24] – this construction is based on bilinear pairing and collision-resistant hash functions and is more efficient than Micali et al.'s construction. Feasibility of "theoretically optimal"[4] VRFs was settled by Hofheinz and Jager [34] – as expected, the design is not practical. This was later improved by Kohl [38] and very recently by Niehues [42]. Nir Bitansky [7] explores the relations between VRFs and other cryptographic concepts such as non-interactive zero-knowledge proofs. Post-quantum secure VRFs were explored by Esgin et al. [25].

In the practical regime, the most relevant construction was proposed by Goldberg et al. [31], which is being used by many enterprises such as Algorand and is now in the process of IETF standardization. The VRF design combines a pseudorandom function and a simple zero-knowledge proof of exponent (namely Schnorr's [45]). The designs elaborated on in this paper are conceptually related to this approach.

Distributed VRF. Distributed VRF was first considered by the work of Dodis [23], which requires a trusted dealer. Kuchta and Manulis [39] proposed a generic construction based on aggregate signatures. However, the most relevant to us is the work by Galindo et al. [27, 28] who formalized the security properties and analyzed three constructions. The first construction is a variant of distributed PRF [4, 41], which is essentially a distributed counterpart of the Goldberg et al. [31] construction with appropriately adjusted zero-knowledge proofs and a specific distributed key-generation protocol (a variant of Gennaro et al. [30]) – this is termed as DDH-DVRF. While the computation is very efficient, the size of the final proof is proportional to the number of participants. The second construction they considered is the one that was proposed and also used by Dfinity [33] – this is similar to DDH-DVRF, but uses bilinear pairing to enable a compact proof. However, the use of bilinear groups comes with a cost over discrete log groups (as mentioned later). The construction is very similar to BLS signatures [9] and is used in many places [17, 18, 20, 44]. Their final construction is called GLOW-DVRF – this was proposed in that paper. GLOW-DVRF uses bilinear pairing for final verification, but Schnorr's proof of exponent for partial verification – as a result not only is the security improved but the computation time is also improved by about 2.5x. The only cost is in terms of the size of partial proofs, which increases a little, but still stays well within the allowed bandwidth. Our Pri-DVRF construction is based on this.

VRFs in Blockchain. Many blockchain services use VRFs internally as a crucial source of randomness. For example, Cardano [13] and Polkadot [43] implement VRFs for block production. Dfinity [33] uses a DVRF (namely Dfinity-DVRF, as mentioned above) for producing a decentralized random beacon. Chainlink offers a popular VRF service that employs the VRF algorithm from Goldberg et al. [31] along with some optimizations. However, from their description [14], it seems that their VRF secret-key is not decentralized (in other words, they do not use a DVRF), and therefore is susceptible to a *single point of failure*.

Oblivious PRF (OPRF). The notion of Oblivious PRF (OPRF) is quite pertinent to our notion of Pri-VRF. OPRF is an extension of PRF to two-party setting where a server holds the secret key and a client holds an input – the notion was introduced in [26] and found numerous interesting applications, such as in key-word search, private set intersections etc. The main guarantees provided by OPRF are twofold: (i) the server should not learn the input (which we do not require); (ii) a client should not be able to break the pseudorandomness of the output (which we also need). Our Pri-VRF instantiations are similar to the (Distributed) OPRF used in the Jarecki et al. [35] and [3]. And we use a very similar BOMDH (T-BOMDH for Pri-DVRF) assumptions to prove the pseudorandomness of our construction. However output-privacy part is a new addition and requires new analysis.

5 PRELIMINARIES

Notation. We use $\mathbb{N}$ to denote the set of positive integers, $\mathbb{Z}$ to denote the set of all integers and $[n]$ to denote the set $\{1, 2, \ldots, n\}$ (for $n \in \mathbb{N}$). A tuple of values is denoted by the vector notation $\mathbf{v} = (v_1, v_2 \ldots)$. For a boolean vector $\mathbf{v} \in \{0, 1\}^n$, its hamming weight is given by the number of 1s in $\mathbf{v}$. For any set S, |S| denotes its cardinality.

We denote the security parameter by κ. We assume that every algorithm takes κ as an implicit input, and all definitions work for any sufficiently large choice of $\kappa \in \mathbb{N}$. We will omit mentioning the security parameter explicitly except in a few places. We use $\mathsf{negl}(\kappa)$ to denote a negligible function in the security parameter; a function $f : \mathbb{N} \to \mathbb{N}$ is considered negligible if for every polynomial p, it holds that $f(n) < 1/p(n)$ for all large enough values of n. Similarly, we use $\mathsf{poly}(\kappa)$ to denote a polynomial function of the security parameter κ.

We use $\mathsf{D}(x) =: y$ or $y := \mathsf{D}(x)$ to denote the evaluation of a specifically deterministic algorithm D on input x to produce output y. Often we use $x := \mathsf{val}$ to denote the assignment of a value val to the variable x. We write $\mathsf{R}(x) \to y$ or $y \leftarrow \mathsf{R}(x)$ to denote evaluation of a probabilistic algorithm R on input x to produce output y. We mostly consider probabilistic polynomial time (PPT) algorithms, which are randomized and run in polynomial time.

For a boolean condition $b := (x = y)$, we denote that if $x = y$ is satisfied, b gets the value 1, otherwise, if $x \neq y$ and the check fails, it gets the value 0.

[4] By theoretically optimal we mean that the design was proposed only to satisfy theoretical interest with the minimal assumption, standard model (for example, not in random oracle model), adaptive security etc.

Computational Hardness. When we say a problem is **computationally hard**, we mean that given a problem instance, generated using the security parameter κ, for any probabilistic algorithm $\mathcal{A}$ that runs in $O(\mathsf{poly}(\kappa))$ time, the probability that $\mathcal{A}$ can solve the given problem instance is upper bounded by $\leq \mathsf{negl}(\kappa)$.

Polynomial Interpolation. A polynomial $P(x)$ over a finite field $\mathbb{F}$ of degree t can be expressed as $P(x) = c_0 + c_1x + c_2x^2 \ldots c_tx^t$, where each coefficient is in $\mathbb{F}$. Given any $\ell \geq t+1$ evaluation points $P(j_1), \ldots, P(j_\ell)$, where $S = \{j_1, \ldots, j_\ell\}$ there are scalars $\lambda_{i,j,S}$ such that for any $i \in \mathbb{N}$: $P(i) = \sum_j \lambda_{i,j,S}P(j)$ Importantly, the Lagrange coefficient $\lambda_{i,j,S}$ corresponding to j depends *only* on the set S and the evaluation point at i.

The function $\mathsf{Rand}(\cdot)$**.** For compact presentation we use a on-the-fly random function, denoted $\mathsf{Rand}(\cdot)$, in our Pri-VRF and Pri-DVRF definitions. For a given domain Dom and Rng, the function, has a table T containing pairs (x, y) where $x \in$ Dom and is initialized to $\emptyset$. It works as follows:

$\underline{\mathsf{Rand}(x \in \mathsf{Dom})}$.
- If there exists $(x, y) \in T$, return y.
- Else return a uniform random $y \leftarrow_\$$ Rng and append (x, y) to T.

5.1 Universal Composability

In the UC framework, a PPT algorithm called the environment (which is adversarial) is trying to distinguish between a real and an ideal world. The adversary in the protocol can corrupt parties in the real world, whereas an ideal adversary, called the simulator, simulates the adversarial behavior in the ideal world. The ideal world comprises an ideal functionality (a.k.a. trusted third party) that is directly connected to all the parties, among which the simulator fully controls the corrupt ones. The honest ideal world parties are called dummy parties because they are interfaces between the environment and the ideal functionality. The objective is to design a simulator in the ideal world such that no environment providing inputs to and observing the outputs from the computing entities can distinguish between the real world and the ideal world, given the adversary's view of both worlds. The simulator typically simulates the real world to an instance of the real-world adversary by providing messages on behalf of the honest parties while accessing the ideal functionality and finally outputs whatever the adversary outputs. The simulator can schedule messaging and outputs in the ideal world to prevent trivial distinctions by timing. All entities are formally modeled as instances of an interactive Turing machine, or ITI. For a detailed formalization, we refer to [10, 12].

Discrete Log, CDH, DDH. For a cyclic group $\mathbb{G}$ of prime order p (where $|p| = O(\kappa)$) with any elements g and $h = g^x$, we denote $x = \mathsf{DLOG}_g(h)$ to denote the discrete logarithm of h to the base g. We assume that given (g, h), computing $\mathsf{DLOG}_g(h)$ is computationally hard – this is called **Discrete Log assumption** over $\mathbb{G}$. Furthermore, for random $g, h \leftarrow_\$ \mathbb{G}$ and random $\alpha \leftarrow_\$ \mathbb{Z}_p$, we say that the Computational Diffie-Hellman (**CDH**) assumption holds when it is computationally hard to compute h^α, given (g, h, g^α). The corresponding decisional version (**DDH**) states that it is computationally hard to distinguish between the tuples $(g, h, g^\alpha, h^\alpha)$ and (g, h, g^α, h') for a uniform random $h \leftarrow_\$ \mathbb{G}$.

Bilinear Pairing, Co-CDH, XDH. Our constructions rely on bilinear pairing. We consider three groups $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$, among which the source groups $\mathbb{G}_0$, $\mathbb{G}_1$ and $\mathbb{G}_T$ all are multiplicative groups of prime order p. The corresponding generators are denoted by g_0, g_1, and g_T. There is an efficiently computable map $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$ which is:

- **bilinear:** for any $a, b \in \mathbb{Z}_p$:
$$e(g_0^a, g_1^b) = e(g_0, g_1^b)^a = e(g_0^a, g_1)^b = e(g_0, g_1)^{ab}$$
- **non-degenerate:** $e(g_0, g_1) \neq 1_T$ where 1_T is the (multiplicative) identity of group $\mathbb{G}_T$.

We require the **Co-CDH assumption** over bilinear groups. The assumption states that: for uniform random $g_1, h_1 \leftarrow_\$ \mathbb{G}_1$ and $g_2 \leftarrow_\$ \mathbb{G}_2$ and uniform random $\alpha \leftarrow_\$ \mathbb{Z}_p$: given $(g_1, h_1, g_1^\alpha, g_2, g_2^\alpha)$ it is computationally hard to compute h_1^α. The corresponding decisional assumption, which requires the adversary to distinguish between h_1^α and a random $h_1' \leftarrow_\$ \mathbb{G}_1$ given $g_1, g_1^\alpha, g_2, g_2^\alpha$ as above is called the **XDH assumption** and is used to build the NIZK proofs (see Sec 5.3).

Unless mentioned otherwise we assume **Type-3 pairings** where not only the source groups $\mathbb{G}_0$ and $\mathbb{G}_1$ are distinct, but also there is *no efficiently computable isomorphism* between them.

5.2 Shamir's Secret Sharing [46].

We use Shamir's secret sharing scheme. Let p be a prime, and n, t be positive integers such that $t < n$. An (n, t, p)-Shamir's Secret Sharing ((n, t, p)-SSS for short) scheme is a pair of algorithms (Share, Recon) that work as follows.

- $\mathsf{Share}(s) \to (s_1, \ldots, s_n)$. This randomized algorithm takes any field element $s \in \mathbb{Z}_p$ as input. Then it works as follows:
  - Sample a uniform random polynomial $P(x) = s + c_1x + \ldots c_tx^t$ of degree t. This is done by sampling each of the coefficients $c_1, \ldots, c_t$ uniformly at random from $\mathbb{Z}_p$. Note that $P(0) = s$.
  - Output shares $s_1, \ldots, s_n$ where $s_i = P(i)$. The tuple $(s_1, \ldots, s_n)$ is also denoted by $\mathsf{Sharing}(s, n, t)$.
- $\mathsf{Recon}(s_{j_1}, \ldots, s_{j_\ell}) =: s/\bot$. The reconstruction is a deterministic procedure which takes a bunch of shares $s_{j_1}, \ldots, s_{j_\ell}$ each from the field $\mathbb{Z}_p$ as input and then executes the following steps:
  - If $\ell \leq t$, then output $\bot$;
    - Otherwise, if $\ell > t$, use the Lagrange coefficients to compute: $s = P(0) := \sum_k \lambda_{0,k,S}s_{j_k}$;
    - Finally output s.

  Security: The scheme provides the following security guarantee: For any uniform random secret $s \leftarrow_\$ \mathbb{Z}_p$, if $(s_1, \ldots, s_n) \leftarrow \mathsf{Share}(s)$, then any $\leq t$ shares $\{s_i\}_{i \in S}$ such that $|S| \leq t$ do not reveal any information about the secret s. More formally, given any $\leq t$ shares, s is still distributed uniformly at random. This is an information theoretic fact.

5.3 NIZK proofs

We require two simple and efficient non-interactive zero-knowledge proof (NIZK) systems. Both were proven to be *complete, sound, and zero-knowledge* based on the DDH assumption on the underlying cyclic group $\mathbb{G}$ of prime order p in the random oracle model. However, in this paper, we use these in one of the source groups of a triple of Type-3 bilinear pairing groups, and hence the corresponding assumption we need is XDH. A NIZK proof system satisfying all these properties is called a **secure NIZK** proof system.

NIZK for Knowledge of Exponent [45]. Our construction uses non-interactive zero-knowledge (NIZK) proof for knowledge of exponents. In particular, given an instance $\mathsf{inst} = (g, h) \in \mathbb{G}^2$ and witness $\mathsf{wit} = k \in \mathbb{Z}_p$ such that $k = \mathsf{DLOG}_g(h)$. Also, consider a hash function $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_p$. So the set of public parameters is defined as $\mathsf{pp} := (\mathsf{H}, \mathbb{G})$, which is provided as an input to all algorithms below. Then the proof system consists of the following two algorithms and a simulator:

- $\mathsf{KExpProve}(\mathsf{inst}, \mathsf{wit}) \to \pi$. This randomized algorithm takes an instance-witness pair $(\mathsf{inst}, \mathsf{wit}) = ((g, h), k)$ as input. Then it executes the following steps:
  - randomly choose $r \leftarrow_\$ \mathbb{Z}_p$;
  - compute $\alpha := g^r \in \mathbb{G}$;
  - compute $c := \mathsf{H}(g, h, \alpha) \in \mathbb{Z}_p$ and $s := r + k \cdot c \in \mathbb{Z}_p$.
  - output the NIZK proof $\pi = (c, s)$
- $\mathsf{KExpVer}(\mathsf{inst}, \pi) =: 1/0$. This deterministic algorithm takes an instance $\mathsf{inst} = (g, h)$ and a candidate proof $\pi = (c, s)$ as input. Then:
  - compute $\alpha := g^s \cdot (x^c)^{-1} \in \mathbb{G}$;
  - output $(c = \mathsf{H}(g, h, \alpha)) \in \{0, 1\}$.
- $\mathsf{KepSimu}(\mathsf{inst}) \to \pi$. The simulator samples $s \leftarrow_\$ \mathbb{Z}_p$ and $c \leftarrow_\$ \mathbb{Z}_p$, then compute $\alpha := g^s \cdot (x^c)^{-1}$ then program the random oracle as: $c := \mathsf{H}(g, h, \alpha)$.

NIZK for Equality of Discrete Log [16]. Our construction uses non-interactive zero-knowledge (NIZK) proof for equality of discrete logarithms, which is quite similar to the above proof. We highlight the crucial differences in black. In particular, consider an instance $\mathsf{inst} = (g, h, x, y) \in \mathbb{G}^4$ and witness $\mathsf{wit} = k \in \mathbb{Z}_p$ such that $k = \mathsf{DLOG}_g(x) = \mathsf{DLOG}_h(y)$ and a hash function $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_p$. So the set of public parameters is defined as $\mathsf{pp} := (\mathsf{H}, \mathbb{G})$, which is provided as an input to all algorithms implicitly. Then, the proof system consists of the following two algorithms and a simulator:

- $\mathsf{EqProve}(\mathsf{inst}, \mathsf{wit}) \to \pi$. This randomized algorithm takes a statement-witness pair $(\mathsf{inst}, \mathsf{wit}) = ((g, h, x, y), k)$ as input. Then it executes the following steps:
  - randomly choose $r \leftarrow_\$ \mathbb{Z}_p$;
  - compute $\alpha := g^r \in \mathbb{G}$; $\beta := h^r \in \mathbb{G}$;
  - compute $c := \mathsf{H}(g, h, x, y, \alpha, \beta) \in \mathbb{Z}_p$ and $s := r + k \cdot c \in \mathbb{Z}_p$.
  - output the NIZK proof $\pi = (c, s)$
- $\mathsf{EqVer}(\mathsf{inst}, \pi) =: 1/0$. This deterministic algorithm takes a statement $\mathsf{inst} = (g, h, x, y)$ and a candidate proof $\pi = (c, s)$ as input. Then:
  - compute $\alpha := g^s \cdot (x^c)^{-1} \in \mathbb{G}$;
  - compute $\beta := h^s \cdot (y^c)^{-1} \in \mathbb{G}$;
  - output $(c = \mathsf{H}(g, h, x, y, \alpha, \beta)) \in \{0, 1\}$.
- $\mathsf{EqSimu}(\mathsf{inst}) \to \pi$. The simulator samples $s \leftarrow_\$ \mathbb{Z}_p$ and $c \leftarrow_\$ \mathbb{Z}_p$, then compute $\alpha := g^s \cdot (x^c)^{-1}$ and $\beta := h^s \cdot (y^c)^{-1}$. Finally, program the random oracle as $c := \mathsf{H}(g, h, \alpha)$.

5.4 Our Model

We follow the Universal Composability Framework [10], in that a real-world multi-party protocol realizes an ideal functionality. Similar to the simplified UC framework [12] we assume the existence of a *default authenticated channel* in the real world. This significantly simplifies our definitions and can easily be removed using an ideal authenticated channel functionality [11].

We consider a *fixed number of parties* in the system and a **static corruption** model, that is, neither the set of participants nor the set of corrupt parties can change during the execution. The corrupt parties can behave in a completely **malicious** manner and may collude with each other.

For more details on the UC framework see Section 5.1.

5.5 (Threshold) One-More Diffie-Hellman Assumptions

We use a variant of threshold one-more Diffie-Hellman assumptions used in [3, 35]. In particular, our assumption will be over *bilinear pairing groups*, and for that, we also do not need the gap-versions. A proof in the generic group model is included in Appendix B.3.

Notations. We use notations from Agrawal et al. [3]. For $t, f, n \in \mathbb{N}$ (where $f \leq t < n$) and $\mathbf{q} = (q_1, \ldots, q_n) \in \mathbb{N}^n$, define $\mathsf{Max}_{t,f}(\mathbf{q})$ to be the largest value of ℓ for which there exists binary vectors $\mathbf{u}_1, \ldots, \mathbf{u}_\ell \in \{0,1\}^n$ such that each $\mathbf{u}_i$ has hamming wight $\geq t - f$ and $\mathbf{q}$ satisfies $\mathbf{q} \geq \sum_{i=1}^{\ell} \mathbf{u}_i$. Next, we define the T-BOMDH – Threshold-Bilinear One-more Diffie Hellman assumption.

Definition 1 (T-BOMDH). *Consider polynomial (in κ) size integers n, t, f, N such that $f \leq t < n$ and consider bilinear pairing groups $\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ where each group has prime order p. Let g_1 and g_2 be two random generators of the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Then we say that the T-BOMDH assumption holds, if for all PPT adversary $\mathcal{A}$ the probability of the following game returning* 1 *is* $\leq \mathsf{negl}(\kappa)$.

- *Sample uniform random secret $\alpha \leftarrow_\$ \mathbb{Z}_p$.*
- *Sample random group elements $\widehat{g}_1, \ldots \widehat{g}_N \in \mathbb{G}_1$.*
- *Provide $g_1, g_1^\alpha, g_2, g_2^\alpha, (\widehat{g}_1, \ldots, \widehat{g}_N)$ to $\mathcal{A}$.*
- *On receiving $\{(i, \alpha_i)\}_{i \in [f]}$ from $\mathcal{A}$ choose an t-degree polynomial D uniformly at random such that for all $i \in [f]$: $D(i) = \alpha_i$ and $D(0) = \alpha$.*
- *Set $\mathbf{q} := 0^n$.*
- *Give the following oracle access $O(i, x)$ to the adversary:*
  $\underline{O(i, x \in \mathbb{G})}$
  - *Increment q_i by* 1.
  - *Output x^{α_i} where $\alpha_i := D(i)$.*
- *On receiving $\{(\bar{g}, \bar{h})\}_{i \in [\ell]}$ from $\mathcal{A}$, return* 1 *if and only if all of the following conditions are met:*
  - *All $\bar{g}_i$ are distinct and $\ell > \mathsf{Max}(\vec{q})$.*
  - *For all $i \in [\ell] : \bar{g} \in \{\widehat{g}_1, \ldots, \widehat{g}_N\}$ and $\bar{h}_i = \bar{g}_i^\alpha$.*

Discussion and BOMDH assumption. The main difference of our assumption with the versions used in [3, 35] is that instead of a *gap-version* we use a bilinear pairing group. Intuitively it has a similar effect because one can use bilinear pairing to check (a specific form of) DDH across source groups. The basic intuition of the above assumption is to give the adversary the oracle access to individual polynomial points in such a manner that, unless the adversary gathers enough, that is $(t+1)$, evaluation points on a certain input, it can not compute that evaluation point in the exponent of a randomly chosen element. The complexities in notation arise as the oracle has no way to distinguish whether the adversary is hiding the actual input with some known randomizer (such as instead of x the adversary can query on x^r to obtain the same result for a known r). For more intuition, we refer to [3, 35]. We also use a specific version of the above assumption, when $n = 1$ and $f = t = 0$, which has found more usage in the literature (e.g. [6, 37]) and is called simply the **BOMDH assumption**.

6 OUTPUT PRIVATE VRF (Pri-VRF)

In this section, we put forward the formal definition of Output Private VRFs (Pri-VRF). Our definition follows the UC-framework [10] and is based on ideas from the UC-based VRF definitions provided by Coretti et al. [19]. We then present our construction and security analysis with respect to the proposed definition.

6.1 Definition: Pri-VRF

We consider a general setting, in that many instances of Pri-VRF protocols are executed among multiple parties connected by point-to-point authenticated (but public) channels. In a specific execution of a VRF, a party (called client) with an input x interacts with another party (called server) with a public verification key vk. The server holds a long-term secret key sk corresponding to vk, and at the end, the client obtains $y = V_{sk}(x)$ and a proof π, where $V : \{0,1\}^* \times \{0,1\}^\kappa \rightarrow \{0,1\}^\gamma$ denotes the VRF function. The output should be *pseudorandom* to the client. There is *public verifiability*, which means that given the triple (x, y, π), anyone can verify whether y is indeed equal to $V_{sk}(x)$. Furthermore, the protocol should satisfy *uniqueness*, which guarantees that, there does not exist another $y' \neq V_{sk}(x)$ and a proof π' such that (x, y', π') verifies successfully. These are the standard properties offered by any VRF. In Pri-VRF, we additionally require *output-privacy*, which guarantees that only the client and the server knows the output y in this case. Moreover, this should be guaranteed while maintaining *public verifiability with respect to the transcript* of the communication – if the client's message to server is $\widetilde{x}$ and server's response is $\widetilde{y}$, then the pair $(\widetilde{x}, \widetilde{y})$ must be publicly verifiable as well. We call this crucial property *public pre-verifiability*.

Ideal Functionality $\mathcal{F}_{\mathsf{pvrf}}$. All guarantees are captured by the ideal functionality $\mathcal{F}_{\mathsf{pvrf}}$, which is detailed in Figure 3. The ideal functionality interacts with parties, generally denoted by P and a simulator $\mathcal{S}$. The phrase "any ITI", denoted by M, refers to either a party P or the simulator $\mathcal{S}$. The ideal functionality keeps track of the following variables, all of which are initialized to $\perp$ (or $\emptyset$) implicitly.

(1) $Keys[M]$: contains the verification keys owned by any ITI M. We say that a verification key vk is unique if there exists a unique M, for which $vk \in Keys[M]$.

(2) $T[vk, x]$: contains entries of the form (y, Π, B) corresponding to a verification key vk and an input x. Each entry contains an output y, and sets $\Pi = \{\pi_1, \pi_2, \ldots\}$, $\mathrm{B} = \{\beta_1, \beta_2 \ldots\}$ etc. The set Π contains all proofs for the tuple (vk, x, y), whereas the set B contains the corresponding server messages. We say that a proof π is *unique* whenever there exists a unique pair (vk, x) such that $\pi \in T[vk, x]$. Similarly, *uniqueness* of transcript and a pair (π, β) are defined. When we say *append* (π, β) to a list T it means that updating the sets $\Pi := \Pi \cup \{\pi\}$ and $\mathrm{B} := \mathrm{B} \cup \{\beta\}$. When we say that the list $T[vk, x]$ is *defined*, that implies $T[vk, x] \neq \perp$.

(3) $Inp[vk, x]$: Contains identity of a party, who is the sender/client for the execution specific to (vk, x). If $Inp[vk, x] = Q$, that implies Q holds the input x in the execution.

Some intuitions on $\mathcal{F}_{\mathsf{pvrf}}$. We follow the overall approach taken by Coretti et al. and therefore do not include session id for simplicity – note that an execution session can be indeed uniquely identified by a pair (vk, x) due to the uniqueness criteria, which makes a

---

Ideal Functionality $\mathcal{F}_{\mathsf{pvrf}}$

Key-registration. Upon $(\mathtt{RegKey}, vk, P)$ from $\mathcal{S}$: If vk is unique add it to $Keys[P]$, send $(\mathtt{Key}, vk)$ to P, else exit.

Input: Upon $(\mathtt{Input}, vk, x)$ from any client Q:

(1) If $Inp[vk, x] = \perp$, and there is a P such that $vk \in Keys[P]$, then set $Inp[vk, x] := Q$ and forward the message to $\mathcal{S}$; when $\mathcal{S}$ returns the same message, then send it to P.
(2) Else exit.

Evaluation: Upon $(\mathtt{Eval}, x, vk)$ from any server P: If $Inp[vk, x] = \perp$ or $vk \notin Keys[P]$ then exit; otherwise let Q be such that $Q := Inp[vk, x]$ and forward the request to $\mathcal{S}$. If $\mathcal{S}$ returns $\perp$, then send $(\mathtt{Val}, vk, x, \perp)$ to Q and P. Otherwise:

(1) When $\mathcal{S}$ returns (π, β), if each of them is unique append the triple $(y := \mathtt{Rand}(vk, x), \pi, \beta)$ to $T[vk, x]$, otherwise exit.
(2) Send $(\mathtt{Val}, vk, x, y, \pi, \beta)$ to P and Q.

Pre-Verification: Upon $(\mathtt{Pre\text{-}Verify}, vk, x, \beta)$ from any ITI M: Send $(\mathtt{Pre\text{-}Verify}, vk, x, \beta)$ to $\mathcal{S}$, and upon receiving ϕ from $\mathcal{S}$:

(1) If there is a P for which $vk \in Keys[P]$ *and* $T[vk, x] = (y, \Pi, \mathrm{B})$ is defined then do as follows:
  (a) If $\beta \in \mathrm{B}$ set $f := 1$.
  (b) Else, if $\phi = 1$ and β is unique: then append β into $T[vk, x]$ and set $f := 1$.
  (c) Else, set $f := 0$.
(2) Else, set $f := 0$.
(3) Finally return f to M.

Reveal: Upon $(\mathtt{Reveal}, vk, x)$ from any client Q: send this to $\mathcal{S}$, when $\mathcal{S}$ returns the message, then mark (vk, x) as $\mathtt{Revealed}$.

Unblind: Upon $(\mathtt{Unblind}, x, \beta)$ from any ITI M: Only if there is a triple (Q, P, vk) such that $\beta \in T[vk, x]$ and $vk \in Keys[P]$ and $Q = Inp[vk, x]$ then go to the next step, otherwise exit:

(1) If either (vk, x) is marked $\mathtt{Revealed}$ or $M = Q$ then return (y, π) to M where $(y, \pi) \in T[vk, x]$. Else exit.

Verification: Upon $(\mathtt{Verify}, vk, x, y, \pi)$ from any M forward this to $\mathcal{S}$, and upon receiving ϕ from $\mathcal{S}$:

(1) If there is a P for which $vk \in Keys[P]$ *and* $T[v, x]$ is defined then do as follows:
  (a) If $(y, \pi) \in T[vk, x]$ set $f := 1$.
  (b) Else, if $\phi = 1$ and π is unique: then append π to $T[vk, x]$ and set $f := 1$.
  (c) Else, set $f := 0$.
(2) Else, set $f := 0$.
(3) Finally return f to M.

Figure 3: Ideal Functionality of Pri-VRF

session id redundant. The major difference with their definition comes obviously from the output-privacy requirement. We capture that through replacing the output with another variable β, which works as a placeholder for the server's message and is used in the pre-verification. For the same purpose, we also introduce a `Reveal` and `Unblind` phase. A minor difference with their approach is that we merge the `Eval` and `RegKey` queries from the simulator and any other party as the simulator controls the corrupt parties and can make those queries through them.

Also note that in the ideal functionality $\mathcal{F}_{\mathsf{pvrf}}$ the output is given to both P and Q – at a first glance, it may appear to be a violation of output-privacy. However, we stress that, this is not the case. This phenomenon is specific in the centralized setting where P holds the whole secret-key. So, given x it can locally compute y. The output-privacy in this case would guarantee secrecy of y from eavesdroppers. Looking ahead, in the distributed setting (i.e. $\mathcal{F}_{\mathsf{pdvrf}}$) y is not given to anyone but Q as long as at most t parties are compromised. However, if there are more than t corruptions it does give away y to the simulator – this becomes analogous to giving y to P in the centralized setting.

Real-world for Pri-VRF. In the real world we assume a structured protocol execution. Towards that, first consider the following set of algorithms:

- $\mathsf{Keygen}(1^\kappa) \to (sk, vk)$: The key-generation algorithm outputs a pair of keys (sk, vk) – sk is the secret key and vk is the verification key.
- $\mathsf{Blind}(1^\kappa, x) \to (\mathsf{st}, \widetilde{x})$: This algorithm processes the input x to offer a secret state st and a public output $\widetilde{x}$.
- $\mathsf{InpVer}(1^\kappa, (x, \widetilde{x})) =: 1/0$. The input verification algorithm verifies whether the pair $(x, \widetilde{x})$ is correctly computed, and returns 1 if and only if the check succeeds.
- $\mathsf{Eval}(vk, sk, \widetilde{x}) \to \widetilde{y}$: The evaluation algorithm uses the secret key sk (and possibly also the verification key vk) on the blinded input $\widetilde{x}$ to produce a blinded output $\widetilde{y}$.
- $\mathsf{PreVer}(vk, (x, \widetilde{x}, \widetilde{y})) =: 1/0$: The pre-verification algorithm verifies whether the computed blinded value $\widetilde{y}$ is correct for the pair $(x, \widetilde{x})$ and verification key vk.
- $\mathsf{Unblind}(x, \widetilde{y}, \mathsf{st}) =: (y, \pi)$: The deterministic unblinding algorithm takes a blinded output $\widetilde{y}$ and a secret-state st (typically generated during the blinding procedure) plus an input x as inputs and outputs an output-proof pair (y, π).
- $\mathsf{Verify}(vk, (x, y, \pi)) =: 1/0$: The verification algorithm takes the public verification key vk and a pair (x, y) as input and outputs a decision bit.

In a real-world with parties connected by pairwise authenticated channels, any party P_S may run Keygen and publish a verification key vk while keeping the secret key sk private – P_S will be called a server. Any other party P_C may have an input x and is called a client – she wants to derive a VRF $y = V_{sk}(x)$. A party can be a server or a client in different executions. The client runs Blind and sends over the pair $(x, \widetilde{x})$ to the server, which first checks whether the blinded input was correctly computed using InpVer on the pair $(x, \widetilde{x})$. In fact, anyone else can perform this check. If the verification succeeds, then the server runs Eval on $\widetilde{x}$ to produce $\widetilde{y}$ and subsequently sends that over to P_C. Anyone (may or may not be the same as P_C) can run PreVer on $(vk, \widetilde{x}, \widetilde{y})$ to publicly verify whether the server's computation was correct. At any point, the client may unblind by running Unblind on $(x, \widetilde{y})$ to get the final output-proof pair (y, π). Once the pair (y, π) is made public, anyone can check whether y was correctly computed from x by publicly running Verify on (x, y, π). Importantly a combination of the checks Verify, PreVer, and InpVer together allow public verification of the full transcript $(x, \widetilde{x}, \widetilde{y}, y)$.

The real world execution is described in a protocol Π (Figure 4)

Pri-VRF Protocol Π

- **Key Generation:** Any party P_S, who would be a server runs key-generation $(sk, vk) \leftarrow \mathsf{Keygen}(1^\kappa)$.
- **Request:** Any client P_C with an input x runs $(\mathsf{st}, \widetilde{x}) \leftarrow \mathsf{Blind}(1^\kappa, x)$. Then it sends $\widetilde{x}$ to a particular server P_S.
- **Response:** The server P_S, on receiving a request on a blinded input $\widetilde{x}$ executes the following steps:
  - Run $\mathsf{InpVer}(1^\kappa, (x, \widetilde{x}))$, if it returns 0, then do nothing. Else go to the next step.
  - Run $\widetilde{y} \leftarrow \mathsf{Part.Eval}(vk, sk, \widetilde{x})$ and then send $\widetilde{y}$ back to P_C.
- **Pre-verification:** P_C, once gets $\widetilde{y}$ runs $\mathsf{PreVer}(vk, (\widetilde{x}, \widetilde{y}))$ – if it outputs 0, then discard $\widetilde{y}$, otherwise unblind $(y, \pi) := \mathsf{Unblind}(\widetilde{y}, \mathsf{st})$. When necessary it publishes (x, y, π).
- **Verification:** Anyone, on input (x, y, π) can run $\mathsf{Verify}(vk, (x, y, \pi))$ and if and only if that returns 1 concludes that the triple is legitimate.

Figure 4: A real world Pri-VRF protocol Π.

Definition 2 (UC-security of Pri-VRF). *Let Π be a protocol that works as above and provides the algorithm specifications. Then we say that Π UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{pvrf}}$ if for any real-world static, malicious PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ in the ideal world, such that for all environment $\mathcal{E}$:*

$$\textsc{Real}_{\Pi,\mathcal{A},\mathcal{E}} \approx_c \textsc{Ideal}_{\mathcal{F}_{\mathsf{pvrf}},\mathcal{S},\mathcal{E}}$$

Unique Input Ownership. We assume that in the protocol each input x is unique to a the client who provides it – therefore if client Q provides input x, we call Q the *owner*. Without this, one may think about the following attack. Another client Q' observes the input x and separately executes a legitimate VRF protocol to compute y – this is not desirable. This can be ensured simply by appending the unique party identity to the input, which would be checked during evaluation by each server – this is possible due to the presence of pairwise authenticated channels.[5] We also note that, in our ideal functionality, the ownership is with respect to the session, defined by (vk, x), and formalized by $Inp[vk, x]$. So, our protocol would provide a slightly stronger guarantee than what is required by the definition.

6.2 Our Pri-VRF Construction

We now present our Pri-VRF construction. (See Figure 5.) This construction is based on a non-threshold version of the BLS-based

[5]Note that, a similar issue arises in Distributed Encryption setting, as mentioned in Agrawal et al. [4]. In fact, this was resolved exactly by appending the party identity in presence of pairwise authenticated channels.

Ingredients

Public parameters: The security parameter κ. An efficiently computable Type-3 bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where the groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are multiplicative groups and each of prime order p. g_1 and g_2 are randomly chosen generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Without loss of generality we assume that all algorithms have the public parameters as input.

Hash functions $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{G}_1$; $\mathsf{H}_2 : \mathbb{G}_1 \to \{0,1\}^\gamma$; $\mathsf{H}_3 : \{0,1\}^* \to \mathbb{Z}_p$.

A secure NIZK proof system (KExpProve, KExpVer) for knowledge of exponent in group $\mathbb{G}_1$. The public parameter for this proof system is $\{\mathsf{H}_3, \mathbb{G}_1\}$.

Construction

- $\mathsf{Keygen}(1^\kappa) \to (sk, vk)$: Sample $sk \leftarrow_\$ \mathbb{Z}_p$ and set $vk = (vk_1, vk_2) := (g_1^{sk}, g_2^{sk})$.
- $\mathsf{Blind}(1^\kappa, x) \to (\mathsf{st}, \widetilde{x})$: Sample a uniform random $\rho \leftarrow_\$ \mathbb{Z}_p$ and set $\psi := \mathsf{H}_1(x)^\rho$. Then:
  - Produce the proof μ using KExpProve on instance $(\mathsf{H}_1(x), \psi)$ and witness ρ.
  - Set $\mathsf{st} := \rho$ and $\widetilde{x} := (\psi, \mu)$.
- $\mathsf{InpVer}(x, \widetilde{x}) =: 1/0$:
  - Parse $(\psi, \mu) := \widetilde{x}$.
  - Then run KExpVer on the instance $(\mathsf{H}_1(x), \psi)$ – if it fails output 0; otherwise output 1.
- $\mathsf{Eval}(vk, sk, \widetilde{x}) \to \widetilde{y}$:
  - Parse $(\psi, \mu) := \widetilde{x}$.
  - Compute $\widetilde{y} := \psi^{sk}$.
- $\mathsf{PreVer}(vk, (\widetilde{x}, \widetilde{y})) \to 1/0$: Return the check:
  - $e(\widetilde{y}, g_2) = e(\widetilde{x}, vk_2)$
- $\mathsf{Unblind}(\widetilde{y}, \mathsf{st}) =: (y, \pi)$.
  - Parse $\rho := \mathsf{st}$
  - Compute $\pi := \widetilde{y}^{\rho^{-1}}$.
  - Compute $y := \mathsf{H}_2(\pi)$.
- $\mathsf{Verify}(vk, (x, y, \pi)) =: 1/0$: Return the check:
  - $(e(\mathsf{H}_1(x), vk_2) = e(\pi, g_2)) \wedge (\mathsf{H}_2(\pi) = y))$.

Figure 5: Our Pri-VRF construction

DVRF proposed in [28]. We argue it satisfies our Pri-VRF definition as captured by our ideal functionality $\mathcal{F}_{\mathsf{pvrf}}$. Formally we state the following theorem, which is proven in Appendix B.1.

THEOREM 1. *Our Pri-VRF construction, described in Fig. 5, UC-realizes $\mathcal{F}_{\mathsf{pvrf}}$ with overwhelming probability as long as the one-more BDH assumption (BOMDH) and the Co-CDH assumption hold over the underlying bilinear groups; the hash functions are modeled as random oracles; and the underlying NIZK proof is secure (which requires XDH over the same groups).*

Here we provide some intuitions. We consider a simpler setting comprising of three parties: a client, a server and an eavesdropper. And, correspondingly we consider three scenarios for a particular execution with a fixed (vk, x), in each of which there is exactly one corrupt party (as we argue in the analysis that this is without loss of generality). Now, when only the eavesdropper is corrupt, we want to guarantee exactly "output-privacy". We show that in this case the simulator is able to simulate the communication between the honest server and the honest in a way which is computationally indistinguishable from the real world as long as Co-CDH holds in the underlying bilinear pairing group and the NIZK proof is zero-knowledge. We argue this by providing an explicit reduction to Co-CDH (plus the zero-knowledge property of the NIZK). The second case, in which the client is the only corrupt party, we want to guarantee pseudorandomness of the VRF output – this case is quite similar to the pseudorandomness of the Oblivious PRF and is reduced similarly to the BOMDH assumption. The third case considers the server to the only corrupt party – in this case since sk is leaked, the only guarantee we can hope for is the output y is still computed correctly (that is "unbiased"). We provide a simulation strategy involving careful programming of the random oracles in this case.

7 DISTRIBUTED Pri-VRF (Pri-DVRF)

In this section we introduce the Distributed variant of Pri-VRF, which we call Pri-DVRF in short. First, we present our UC-based definition, first describing the ideal functionality and later providing a specifically structured real-world execution. Later in this section, we provide our Pri-DVRF construction.

7.1 Definition: Pri-DVRF

In the distributed setting, no server alone holds the entire key. Instead, the VRF secret-key sk is distributed among multiple parties. Let us call the set of n servers $\mathsf{S} = \{P_1, \ldots, P_n\}$ who jointly hold a VRF key sk jointly in a t out of n fashion, for example using a secret-sharing scheme.[6] Now, even if t servers are compromised (and potentially collude with each other), the key is hidden from the adversary. Any client then can interact with $t+1$ servers to evaluate $y = V_{sk}(x)$ and an associated proof π privately, such that no one except the client knows y or π. More concretely, the client sends a message containing the input x to all servers in the set S. As long as $t+1$ replies correctly with blinded responses, the client should be able to aggregate the responses to compute an aggregated blinded response. The client later can unblind to obtain the output-proof pair (y, π), where y must be pseudorandom and publicly verifiable (and remains so even if sk is completely leaked) even when up to t parties are controlled by a malicious adversary. However, in addition to these standard VRF properties, we need more properties in the distributed setting. First, we need *consistency* which means that the final output y is independent of the participating set. We also need *availability/liveness* which means that no matter what the malicious parties do, the protocol will execute correctly (a.k.a. guaranteed output delivery). These two requirements are easy to achieve, the first one by using a t out of n secret sharing scheme, such as Shamir's [46] (which we use in our constructions) and the second one by assuming $n \geq 2t+1$, which is ensured within our ideal functionality $\mathcal{F}_{\mathsf{pdvrf}}$. Another requirement, considered in prior works [28], is *robustness*, which guarantees that if the aggregation is successful, then the final verification would also be successful – this is captured within the ideal functionality by a *partial* pre-verification mechanism which ensures that any incorrect response from a server can be caught during aggregation. Like in the Pri-VRF

[6] The access structure can be generalized to other settings, but in this paper, we stick to t out of n threshold access structure.

setting, we assume multiple parties, any of which can play the role of server or client for any particular execution. A group of parties can collaborate to execute a key-generation to have a common (public) verification key vk and shares $sk_1, sk_2, \ldots$ of a secret-key sk.

Ideal Functionality $\mathcal{F}_{\mathsf{pdvrf}}$. All guarantees, informally described above, are captured by the ideal functionality $\mathcal{F}_{\mathsf{pdvrf}}$ in Figure 6. The ideal functionality interacts with parties, denoted generally by P or Q and a simulator $\mathcal{S}$. Sometimes, to stress on to the distributed aspect, servers are denoted as P_i. A set of n servers $P_1, P_2 \ldots, P_n$ is denoted by S, which plays a similar role to that of a single server in $\mathcal{F}_{\mathsf{pvrf}}$. Sometimes to distinguish a client is denoted by Q. The phrase "any ITI", denoted by M, refers to either a party or the simulator.

The ideal functionality keeps track of the following variable, all of which are initialized to $\perp$ (or $\emptyset$) implicitly.

(1) $Keys[M], Keys[\mathrm{S}]$: contains the public verification keys owned by any entity M or a set of servers $\mathrm{S} = \{P_1, \ldots, P_n\}$. We note that, if $vk \in Keys[\mathrm{S}]$ then $vk \in Keys[P_i]$ for each $P_i \in \mathrm{S}$. We say that a verification key vk is unique if there exists a unique set of servers S, for which $vk \in Keys[\mathrm{S}]$ – this is extended from $\mathcal{F}_{\mathsf{pvrf}}$, which considers uniqueness corresponding to parties.

(2) $T[vk, x]$: contains entries of the form $(y, (\pi, \beta), (\pi', \beta'), \ldots)$ corresponding to a verification key vk and an input x exactly like in the case for Pri-VRF. The uniqueness is also defined exactly in the same manner.

(3) $T_{\mathsf{part}}[vk, x, P_i]$: extends the above definition to the partial setting, where each partial list corresponds to a server P_i. A list T_{part} contains entries $\beta, \beta', \ldots$, which is slightly different from lists T. Uniqueness of β is defined naturally with respect to the triple (vk, x, P_i). Note that, since vk is unique to a set of servers S, we do not need to specify the set of servers.

(4) $Inp[vk, x]$: denotes the party (client) who sent the pair (vk, x) for evaluation. This contains exactly one element, unless marked $\perp$ (while undefined) by default.

The real world execution. Consider the following set of algorithms exclusive to the distributed setting:

- $\mathsf{Keygen}(1^\kappa, n, t) \rightarrow (vk, sk_1, \ldots, sk_n)$: The key-generation algorithm (implemented by a DKG protocol) outputs a verification key vk and n shares $sk_1, \ldots, sk_n$ of the secret-key sk where the sharing is t out of n threshold.
- $\mathsf{Blind}(1^\kappa, x) \rightarrow (\mathsf{st}, \widetilde{x})$: This algorithm processes an input x to offer a secret state st and a public output $\widetilde{x}$.
- $\mathsf{InpVer}(1^\kappa, (x, \widetilde{x})) =: 1/0$. The input verification algorithm verifies whether a pair $(x, \widetilde{x})$ is correctly formed and returns 1 if and only if the check succeeds.
- $\mathsf{Part.Eval}(vk, sk_i, \widetilde{x}) \rightarrow \widetilde{y}_i$: The partial evaluation algorithm uses the partial secret-key sk_i on the blinded input $\widetilde{x}$ to produce a blinded partial output $\widetilde{y}_i$.
- $\mathsf{PartPreVer}(vk, (x, \widetilde{y}_i)) =: 1/0$: There is a partial pre-verification algorithm which verifies whether the computed blinded partial value $\widetilde{y}_i$ is correct for the input x and verification key vk.
- $\mathsf{Aggregate}(vk, \{(\widetilde{y}_i)\}_{i \in S}) =: \widetilde{y}$. The aggregation algorithm gathers a set of blinded values to produce an aggregated blinded value $\widetilde{y}$.
- $\mathsf{PreVer}(vk, (\widetilde{x}, \widetilde{y})) =: 1/0$: There is a pre-verification algorithm, similar to PVRF, which verifies whether the computed blinded value $\widetilde{y}$ is correct for the blinded input $\widetilde{x}$ and verification key vk.
- $\mathsf{Unblind}(\widetilde{y}, \mathsf{st}) =: (y, \pi)$. The deterministic unblinding algorithm takes a blinded output $\widetilde{y}$ and a secret-state st (typically generated during the blinding procedure) and then outputs an output-proof pair (y, π).
- $\mathsf{Verify}(vk, (x, y, \pi)) =: 1/0$: The verification algorithm takes the public verification key vk and a pair (x, y) as input and outputs a decision bit.

In the real-world, parties are connected by pairwise authenticated channels. A set of n parties $P_1, P_2, \ldots, P_n$ successfully run a distributed key-generation protocol,[7] that securely implements Keygen such that the verification key vk is made public and each P_i gets a secret key share sk_i. Let us denote this set of parties by $\mathrm{S} := \{P_1, \ldots, P_n\}$. At any point, a client Q with an input x may run Blind to generate $\widetilde{x}$ and subsequently sends over $(x, \widetilde{x})$ for evaluation to the servers in S (with verification key vk). Server P_i in set S first runs the input-verification InpVer on $(x, \widetilde{x})$, and if that succeeds, runs Part.Eval on $(x, \widetilde{x})$ with (vk, sk_i) to generate a blinded partial output $\widetilde{y}_i$, which it sends back. The values $(\widetilde{y}_1, \widetilde{y}_2, \ldots)$ are supposed to be collected by an aggregator A (which may or may not be the same as Q or any P_i), who then runs PartPreVer on each $\widetilde{y}_i$ with respect to (vk, x), and if there are at least t many correct such values, then it may produce a blinded output $\widetilde{y}$ (otherwise it outputs $\perp$). The client, when obtaining $\widetilde{y}$, may first run PreVer to check whether the aggregation was done correctly (in particular when $A \neq Q$), and if that succeeds, it may unblind using Unblind to obtain (y, π). The triple (x, y, π) can be publicly verified at any point by anyone to confirm that y was correctly produced. Furthermore, combining this with InpVer, PartPreVer, and PreVer anyone can verify whether this value is computed via a particular interaction defined by the entire transcript $(x, \widetilde{x}, \{\widetilde{y}_i\})i, \widetilde{y}, y, \pi)$. The real world execution is described in a protocol Π (Figure 7)

Definition 3 (Distributed Pri-VRF (Pri-DVRF)). Let $\mathbf{\Pi}$ be a protocol that works as above and provide the algorithm specifications. We say that $\mathbf{\Pi}$ UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{pdvrf}}$ if for any static, malicious PPT adversary $\mathcal{A}$ in the real world, there exists a PPT simulator $\mathcal{S}$ in the ideal world, such that for all environment $\mathcal{E}$:

$$\textsc{Real}_{\Pi, \mathcal{A}, \mathcal{E}} \approx_c \textsc{Ideal}_{\mathcal{F}_{\mathsf{dpvrf}}, \mathcal{S}, \mathcal{E}}$$

Remark 2. *Note that our aggregation is a public procedure, and therefore can be done by any of the nodes – this is similar to* all *threshold protocols. Consequently, compromising the aggregator does not allow one to break any security property (in particular, public verifiability ensures that a malicious aggregation is not possible). However, if we rely on a single aggregator node, that may hurt the*

[7] In the description, we do not present a distributed key-generation (DKG) formally. We stress that it would be straightforward to extend the construction in a hybrid model that uses an ideal DKG functionality, for example, a variant of the one provided in [35]. The changes in the proof will also be analogous to theirs. We avoid this for simplicity of the exposition.

Ideal Functionality $\mathcal{F}_{\text{pdvrf}}$

Key Generation. Upon $(\texttt{KeyGen}, vk, \mathrm{S})$ where $\mathrm{S} \subseteq \{P_1, \ldots, P_n\}$ from $\mathcal{S}$ when vk is unique:

(1) Define $\mathrm{C_S} := \mathrm{C} \cap \mathrm{S}$ and $\mathrm{H_S} := \mathrm{S} \setminus \mathrm{C_S}$ and set $n_\mathrm{S} := |\mathrm{S}|$
(2) If $n_\mathrm{S} < 2t + 1$, then exit the procedure.
(3) Append vk to $Keys[\mathrm{S}]$ and for each $P_i \in \mathrm{S}$ $Keys[P_i]$.
(4) If $|\mathrm{C_S}| \geq t + 1$, then mark S as `Corrupt`.
(5) Send $(\texttt{KeyGen}, vk, \mathrm{S})$ to each $P_i \in \mathrm{H_S}$.

Input: Upon $(\texttt{Input}, vk, x)$ from any client Q:

(1) If $Inp[vk, x] = \perp$, and there is a P such that $vk \in Keys[P]$, then set $Inp[vk, x] := Q$ and forward the message to $\mathcal{S}$; when $\mathcal{S}$ returns the same message, then send it to P.
(2) Else exit.

Partial Evaluation. Upon $(\texttt{PartEval}, vk, x)$ from any P_i: If $vk \notin Keys[P_i]$ or $Inp[vk, x] = \perp$ then exit; otherwise send $(\texttt{PartEval}, vk, x, P_i)$ to $\mathcal{S}$. If $\mathcal{S}$ returns $\perp$ then send $\perp$ to P_i; otherwise:

(1) When $\mathcal{S}$ returns β_i, then if it is unique then append it to $T_{\text{part}}[vk, x, P_i]$; otherwise exit.
(2) Send β_i to P_i.

Partial Pre-Verification: Upon $(\texttt{PartPreVerify}, vk, x, \beta_i)$ from any M, forward this to $\mathcal{S}$, and when $\mathcal{S}$ returns ϕ, then do as follows:

(1) If there is a party P_i such that $vk \in Keys[P_i]$ and $T_{\text{part}}[vk, x, P_i]$ is defined then:
  (a) If $\beta_i \in T_{\text{part}}[vk, x, P_i]$ then set $f = 1$
  (b) Else if $\phi = 1$ and β_i is unique, then append β_i into $T_{\text{part}}[vk, x, P_i]$ and set $f := 1$.
  (c) Else set $f := 0$
(2) Otherwise set $f := 0$.
(3) Finally return f to M.

Aggregation: Upon $(\texttt{Aggregate}, vk, x, \beta_1, \ldots, \beta_\ell)$ from any ITI M: if $\ell < t + 1$, then return $\perp$ to M, else forward the message to $\mathcal{S}$, when $\mathcal{S}$ returns β and π, if either of β or π is not unique, then exit, otherwise:

(1) Initialize a temporary list $J := \emptyset$ and append β_i into J only if there is a P_i for which $\beta_i \in T_{\text{part}}[vk, x, P_i]$. If $|J| \leq t$, then append $(y := \texttt{Rand}(vk, x), \pi, \beta)$ into $T[vk, x]$.
(2) Return β to M.
(3) If $vk \in Keys[\mathrm{S}]$ such that S is marked `Corrupt`, then return (y, π) to $\mathcal{S}$.

Pre-Verification: Upon $(\texttt{PreVerify}, vk, x, \beta)$ from any M, forward this to $\mathcal{S}$, and when $\mathcal{S}$ returns ϕ, do:

(1) If $T[vk, x]$ is defined then:
  (a) If $\beta \in T[vk, x]$ then set $f = 1$
  (b) Else if $\phi = 1$ and β is unique, then append β into $T[vk, x]$ and set $f := 1$.
  (c) Else set $f := 0$
(2) Otherwise set $f := 0$.
(3) Finally return f to M.

Reveal: Upon $(\texttt{Reveal}, vk, x)$ from any client Q: send this to $\mathcal{S}$, when $\mathcal{S}$ returns the message, mark (vk, x) as `Revealed`.
Unblind: Upon $(\texttt{Unblind}, x, \beta)$ from any ITI M: Only if there is a triple (Q, S, vk) such that $\beta \in T[vk, x]$ and $vk \in Keys[\mathrm{S}]$ and $Q = Inp[vk, x]$ then go to the next step, otherwise exit:

(1) If either (vk, x) is marked `Revealed` or $M = Q$ then return (y, π) to M where $T[vk, x] = (y, \cdots)$ and $Prv[\beta] = \pi$. Else exit.

Verification: Upon $(\texttt{Verify}, vk, x, y, \pi)$ from any M forward this to $\mathcal{S}$, and upon receiving ϕ from $\mathcal{S}$:

(1) If there is a S for which $vk \in Keys[\mathrm{S}]$ *and* $T[v, x]$ is defined then do as follows:
  (a) If $(y, \pi) \in T[vk, x]$ set $f := 1$.
  (b) Else, if $\phi = 1$ and π is unique: then append π to $T[vk, x]$ and set $f := 1$.
  (c) Else, set $f := 0$.
(2) Else, set $f := 0$.
(3) Finally return f to M.

Figure 6: Ideal Functionality of Pri-DVRF

liveness/availability. To remedy that, one may either deploy t+1 aggregator nodes (to ensure at least one honest aggregator node) or design a simple reward mechanism to incentivize aggregation. We do not formalize this here.

7.2 Our Pri-DVRF construction

We present our Pri-DVRF construction in this section. The construction is a natural extension to our centralized Pri-VRF construction (cf. Figure 5) except that the partial evaluation now produces a zero-knowledge proof of correct partial computation, which is verified by the partial pre-verification algorithm. Our construction is presented in Figure 8. The construction is based on the GLOW-DVRF, proposed in [28]. This construction is based on a non-threshold version

Pri-DVRF Protocol Π

- **DKG:** Parties in set S, that are $P_1, \ldots, P_n$ run a distributed key-generation, after which each party P_i obtains a secret key sk_i and everyone gets a public key vk as a output of $\mathsf{Keygen}(1^\kappa, n, t)$.
- **Request:** Any party Q (which may or may not be part of S) with an input x runs $(\mathsf{st}, \widetilde{x}) \leftarrow \mathsf{Blind}(1^\kappa, x)$. Then it sends $\widetilde{x}$ to all parties $P_1, \ldots, P_n$ in set S.
- **Response:** Each party P_i, on receiving a request on a blinded input $\widetilde{x}$ executes the following steps:
  - Run $\mathsf{InpVer}(1^\kappa, (x, \widetilde{x}))$, if it returns 0, then do nothing. Else go to the next step.
  - Run $\widetilde{y}_i \leftarrow \mathsf{Part.Eval}(vk, sk_i, \widetilde{x})$ and then send $\widetilde{y}$ to an aggregator A (which may or may not be the same as Q or any P_i).
- **Aggregation:** The aggregator A, once collects the values $(\widetilde{y}_1, \widetilde{y}_2, \ldots)$ executes the following steps:
  - Initiate a set $S := \emptyset$.
  - For each i run $\mathsf{PartPreVer}(vk, (x, \widetilde{y}_i))$ – if it returns 1, then append i into S, else do nothing.
  - If S contains at least $t+1$ elements, then run $\widetilde{y} := \mathsf{Aggregate}(vk, \{\widetilde{y}_i\}_{i \in S})$.
  - If $A \neq Q$ then send $\widetilde{y}$ to Q, otherwise store $\widetilde{y}$.
- **Pre-verification:** Q, once gets $\widetilde{y}$ runs $\mathsf{PreVer}(vk, (\widetilde{x}, \widetilde{y}))$ – if it outputs 0, then discard $\widetilde{y}$, otherwise unblind $(y, \pi) := \mathsf{Unblind}(\widetilde{y}, \mathsf{st})$. When necessary it publishes (x, y, π).
- **Verification:** Anyone, on input (x, y, π) can run $\mathsf{Verify}(vk, (x, y, \pi))$ and if and only if that returns 1 concludes that the triple is legitimate.

Figure 7: A real world Pri-DVRF protocol Π.

of the BLS-based DVRF proposed in [28]. We argue our Pri-DVRF construction satisfies our Pri-DVRF definition as captured by our ideal functionality $\mathcal{F}_{\mathsf{pdvrf}}$. Formally we state the following theorem, which is proven in the Appendix B.2 .

Theorem 3. *Our Pri-DVRF construction, described in Fig. 8, UC-realizes $\mathcal{F}_{pdvrf}$ with overwhelming probability as long as the threshold one-more BDH assumption (T-BOMDH) and the co-CDH assumption hold over the underlying bilinear groups; the hash functions are modeled as random oracles; and the NIZK proof systems are secure (that, in turn, require XDH).*

The proof extends naturally from the centralized case. However, each time we need to deal with up to t malicious servers. However, since they do not possess the secret-key, this case essentially becomes analogous to the scenario in the centralized setting, when the server is honest. For example, when the client is honest and there is at most t server corruption, output privacy must be guaranteed. To argue that, now we reduce this to a threshold variant of the BOMDH problem, called T-BOMDH. Analyses of the other cases are similar to the centralized setting.

Remark 4. *We stress that the VRF servers do not need to maintain states. To ensure uniqueness of the input, the smart contract crafts an input (INP as detailed in Appendix A) which is used by the VRF servers – this is done precisely to avoid this sort of "statefulness", because among other things, this input contains the identity of the requester. Hence, unique ownership is easily ensured by a signature (or, more generally, an authenticated channel a la DiSE [4]) which can be checked by the VRF servers.*

Ingredients

Public parameters: The security parameter κ, the total number of parties n, a threshold $t < \lceil n/2 \rceil$. An efficiently computable Type-3 bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where the groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are multiplicative groups and each of prime order p. g_1 and g_2 are randomly chosen generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively.
Hash functions $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{G}_1$; $\mathsf{H}_2 : \mathbb{G}_1 \to \{0,1\}^\gamma$; $\mathsf{H}_3 : \{0,1\}^* \to \mathbb{Z}_p$.
A **Shamir's secret sharing** scheme (that has two algorithms Share and Recon).
A **secure NIZK proof** system $(\mathsf{EqProve}, \mathsf{EqVer})$ for **equality of discrete log** over group $\mathbb{G}_1$. The public parameter for this NIZK system is given by $\{\mathsf{H}_3, \mathbb{G}_1\}$.
A **secure NIZK proof** system $(\mathsf{KExpProve}, \mathsf{KExpVer})$ for **knowledge of exponent** in group $\mathbb{G}_1$. The public parameter for this proof system is $\{\mathsf{H}_3, \mathbb{G}_1\}$.

Construction

- $\mathsf{Keygen}(1^\kappa, n, t) \to (vk, sk_1, \ldots, sk_n)$:
  - Sample $sk \leftarrow_\$ \mathbb{Z}_p$; generate $(sk_1, \ldots, sk_n) \leftarrow_\$ \mathsf{Share}_{n,t,p}(sk)$.
  - Set $pk := g_2^{sk}$, and $\forall\ i \in [n]$: $vk_i := g_1^{sk_i}$.
  - Set $vk := (pk, vk_1, \ldots, vk_n)$.
- $\mathsf{Blind}(1^\kappa, x) \to (\mathsf{st}, \widetilde{x})$: Sample a uniform random $\rho \leftarrow_\$ \mathbb{Z}_p$ and set $\psi := \mathsf{H}_1(x)^\rho$. Then:
  - Produce the proof μ using KExpProve on instance $(\mathsf{H}_1(x), \psi)$ and witness ρ.
  - Set $\mathsf{st} := \rho$ and $\widetilde{x} := (\psi, \mu)$.
- $\mathsf{InpVer}(x, \widetilde{x}) =: 1/0$:
  - Parse $(\psi, \mu) := \widetilde{x}$.
  - Then run KExpVer on the instance $(\mathsf{H}_1(x), \psi)$ – if it fails output 0; otherwise output 1.
- $\mathsf{Part.Eval}(vk, sk_i, \widetilde{x}) \to \widetilde{y}_i$: Parse $(\psi, \mu) := \widetilde{x}$ and:
  - Compute $\widetilde{w}_i := \psi^{sk_i}$.
  - Run EqProve on the instance $(\psi, \widetilde{w}_i, g_1, vk_i)$ with witness sk_i to produce proof of equal exponent $\widetilde{\pi}_i$.
  - Set $\widetilde{y}_i := (\widetilde{w}_i, \widetilde{\pi}_i)$.
- $\mathsf{PartPreVer}(vk, (\widetilde{x}, \widetilde{y})) =: 1/0$: Output the result of EqVer on the instance $(\psi, \widetilde{w}_i, g_1, vk_i)$ and proof $\widetilde{\pi}$ where $(\psi, \mu) := \widetilde{x}$ and $(\widetilde{w}, \widetilde{\pi}) := \widetilde{y}$.
- $\mathsf{Aggregate}(vk, \{(\widetilde{y}_i)\}_{i \in S}) =: z$. If $|S| < t+1$ then output $\perp$, otherwise run the Lagrange interpolation in the exponent on $\widetilde{w}_i$'s where $(\widetilde{w}_i, \widetilde{\pi}_i) := \widetilde{y}_i$:
  - Compute $\widetilde{y} := \prod_{i \in S} \widetilde{w}_i^{\lambda_{i,S}}$
- $\mathsf{PreVer}(vk, (\widetilde{x}, \widetilde{y})) \to 1/0$: Return the check:
  - $e(\widetilde{y}, g_2) = e(\widetilde{x}, vk_2)$
- $\mathsf{Unblind}(\widetilde{y}, \mathsf{st}) =: (y, \pi)$.
  - Parse $\rho := \mathsf{st}$
  - Compute $\pi := \widetilde{y}^{\rho^{-1}}$.
  - Compute $y := \mathsf{H}_2(\pi)$.
- $\mathsf{Verify}(vk, (x, y, \pi)) =: 1/0$: Return the check:
  - $(e(\mathsf{H}_1(x), vk_2) = e(\pi, g_2)) \wedge (\mathsf{H}_2(\pi) = y))$.

Figure 8: Our Pri-DVRF Construction

	Input-Generation	Partial-Eval.
GLOW-DVRF (MCL)	-	253.304 μsec
Pri-DVRF (MCL)	307.079 μsec	403.059 μsec
GLOW-DVRF (RELIC)	-	1.30304 msec
Pri-DVRF (RELIC)	1.67658 msec	2.5978 msec

Table 1: Average time taken for each step for GLOW-DVRF and Pri-DVRF for the BN256 curve, over 100 iterations. In the Pri-DVRF construction, the partial evaluation includes verifying the ZKP forwarded by the requester.

Another construction based on Dfinity-DVRF. We note that the NIZK proof of equality computed in the partial evaluation could just be omitted, if each $g_2^{sk_i}$ was publicly available, and PartPreVer was performed using bilinear pairing $e(\widetilde{x}, g_2^{sk_i}) = e(\widetilde{y}, g_2)$. However, this would incur concrete computation overhead because verifying a NIZK proof of equality amounts to 4 exponentiations in the group $\mathbb{G}_1$, and that is about $2.5x$ more efficient than a single bilinear pairing verification. A DVRF scheme constructed using this alternative approach was deployed by Dfinity [33] and was analyzed by Galindo et al. [28]. We stress that adding output-privacy to that construction is straightforward. Moreover, the issue of strong vs weak pseudorandomness for the DVRF constructions does not seem to appear for the respective Pri-DVRF constructions. Recall that, in the same paper the authors show that Dfinity-DVRF can not be proven strongly pseudorandom, which allows an adversary to make honest partial evaluation queries on the challenge input (the weaker notion was called simply pseudorandomness and does not allow those queries). And they showed that GLOW-DVRF can actually be proven to satisfy the stronger notion. For the corresponding Pri-DVRF construction, this does not seem to be the case, because our approach relies on T-BOMDH oracles for simulating partial evaluation queries.

8 PERFORMANCE ANALYSIS

We evaluate the performance of our Pri-DVRF construction and compare it with the GLOW-DVRF [28] construction. We implement [1] our Pri-DVRF by extending the GLOW-DVRF framework [27, 28] written in C++. The framework supports mcl [2] and RELIC [5] cryptographic libraries.

In our Pri-DVRF construction, for a given input, the requester generates a random blinding value and a NIZK proof of the correctness of the blinded input. The proof is a Schnorr signature-based proof of knowledge of the DLog exponent. The proof consists of two elements, one scalar and one group $\mathbb{G}_1$ element. After receiving the blinded input, each VRF node verifies the zero-knowledge proof before computing the partial evaluation of the VRF. The requester receives the aggregated evaluation and unblinds the output private VRF using the pre-computed blinding value to obtain the final VRF output. The NIZK proof is the only additional input forwarded to the VRF nodes in Pri-DVRF protocol when compared to the non-private version of the protocol. The proof amounts to an overhead of 513 bits.

We benchmark the different steps of the VRF computation using mcl [2] and RELIC libraries [5] for the BN256 curve. We run our single-threaded implementation on Mac OSX 2015 with an Intel i7-3.1GHz processor with 16GB RAM. With the MCL library, the requester takes $\sim 307\mu$ sec on an average for computing $H(x)^r$ (for the input x and the blinding factor r) and the zero-knowledge proof of exponent r. Each VRF node verifies the zero-knowledge proof (ZKP) and then computes the $H(x)^{r \cdot sk_i}$ for the secret share sk_i. The partial evaluation, including verifying the ZKP per node, takes $\sim 403\mu$ sec. Unblinding by the requester involves one exponentiation and takes on an average $\sim 146\mu$sec. The GLOW-DVRF, which is non-private, does not involve any input blinding, and the input message x is forwarded to the VRF nodes. Each VRF node computes the partial evaluation $H(x)^{sk_i}$, which takes $\sim 253\mu$ sec per node on average.

The computation times for input-blinding at the requester and partial evaluation at the VRF node have been presented in table 1; the table provides the timings for the operations using both the mcl and the RELIC libraries. The reported values are taken as a mean over 100 iterations over each operation. A smart contract would verify the VRF output; though we do not deploy the smart contract, our estimates indicate that the gas cost for the VRF verification on the BN256 curve would be $\sim 250k$ gwei (more on the gas cost below).

We also benchmark the average time taken to generate one PVRF value for varying VRF committee sizes. Table 2 indicates the average total time taken to generate one PRVF value. The time for partial evaluation by each VRF node is constant, irrespective of the committee size. The table also indicates the time taken to combine the partial evaluations. The total time without network delays is the summation of the partial evaluation time and the time taken to aggregate the evaluations of the VRF committee nodes. That involves verifying each proof of the correctness of the evaluation and then combining the partial outputs through Lagrange interpolation. This process is similar to both the Pri-DVRF and the non-private GLOW-DVRF protocols. Only the partial evaluation of the nodes differs as far as the VRF committee is concerned. Since the overhead is just checking a Schnorr-based zero-knowledge proof, the time difference between the two approaches is minor.

To simulate a real-network deployment, we also induce network delays of $\sim$ 120 msec between each pair of nodes and compute the total time taken to generate the aggregate VRF output. Table 3 denotes the average time taken to evaluate Pri-DVRF for different committee sizes. Each VRF node forwards the partial evaluation to all the other committee nodes, and each produces the aggregated output value.

Estimate of the gas cost. The base cost for creating and deploying a smart contract is $32k$ gwei on Ethereum. In the FlexiRand protocol, the partial evaluations by each VRF node are combined, and the blinded VRF output along with the proof, is published through the smart contract. The smart contract verifies the proof before publishing it, which is the pre-verification step of the protocol. It performs a pairing-based verification of the blinded VRF output. Since the blinded value is of the form $H(x)^r$, the verification simply involves the equality of two pairing computations: $e(H(x)^r, g_2^{sk}) == e(H(x)^{r \cdot sk}, g_2)$. It must be noted that in the non-private version, while performing the pairing check $(e(H(x), g_2^{sk}) == e(H(x)^{sk}, g_2))$, the hash value $H(x)$ is computed on the smart contract using x. In Pri-DVRF, this hash is not computed, as the (blinded) input value is

	n	Total
GLOW-DVRF (MCL)	8	1.71 msec
Pri-DVRF (MCL)	8	1.89 msec
GLOW-DVRF (RELIC)	8	10.39 msec
Pri-DVRF (RELIC)	8	11.71 msec
GLOW-DVRF (MCL)	16	2.97 msec
Pri-DVRF (MCL)	16	3.12 msec
GLOW-DVRF (RELIC)	16	18.42 msec
Pri-DVRF (RELIC)	16	19.89 msec
GLOW-DVRF (MCL)	32	5.47 msec
Pri-DVRF (MCL)	32	5.66 msec
GLOW-DVRF (RELIC)	32	35.63 msec
Pri-DVRF (RELIC)	32	36.77 msec
GLOW-DVRF (MCL)	64	10.46 msec
Pri-DVRF (MCL)	64	10.66 msec
GLOW-DVRF (RELIC)	64	72.87 msec
Pri-DVRF (RELIC)	64	74.34 msec

Table 2: Average time taken to evaluate Pri-DVRF and GLOW-DVRF for varying n. The time is indicated by the summation of the partial evaluation time and the time to combine the evaluations of the VRF nodes. Network communication delays are not considered here.

	n	Total
MCL	8	0.159 sec
RELIC	8	0.171 sec
MCL	16	0.244 sec
RELIC	16	0.277 sec
MCL	32	0.399 sec
RELIC	32	0.695 sec
MCL	64	1.08 sec
RELIC	64	2.33 sec

Table 3: Average time taken to evaluate Pri-DVRF for varying n with artificial network delay of 120 msec added to the communication.

$H(x)^r$, and the verification involves just two pairings. Each pairing operation costs 108K gwei; hence, the pairing-based verification which involves two pairing operations, costs $\sim 250K$ gwei including 20K gwei for storing a 256 bit value. The requester forwards the input message to the smart contract and, after obtaining the formatted INP (141 bytes as per the description given in the full version [36]), blinds it and forwards it along with the proof of correctness. Compared to the GLOW-based non-private case, this constitutes two additional transactions amounting to $\sim$ 42K gwei. The storage of the additional bit-length of the proof amounts to an additional gas cost of 40K gwei. The total gas cost for each request in the GLOW-based non-private version would be $\sim$ 410K gwei which would amount to roughly \$0.77 USD (as of April 2023). Compared to this, the cost of Pri-DVRF request would be $\sim$ 450K gwei amounting to \$0.84 USD. However, as Pri-DVRF enables reusability without breaking predictability (as explained in Section 1, for example, by using PRGs), the amortized cost turns out to be significantly cheaper – re-using just twice is already cheaper than the non-private counter-part, and re-using, say ten times would make the amortized cost \$0.084 USD, which becomes significantly cheaper.

9 CONCLUSION

Randomness is an indispensable resource in Web3 gaming. With a growing demand for on-chain verifiable randomness, new problems are arising. This work addresses one such problem and proposes a practical solution with formal analysis. We expect more problems to arise in this space in the near future with more innovation happening. Also, as the first work, in this paper, we only formalize the core primitive, namely output-private (distributed) VRF, and leave the formalization of the entire smart-contract-based framework for future work.

REFERENCES

[1] [n. d.]. FlexiRand - Code. https://github.com/easwarvivek/FlexiRand.git.

[2] [n. d.]. mcl - A portable and fast pairing-based cryptography library. . https://github.com/herumi/mcl.

[3] Shashank Agrawal, Peihan Miao, Payman Mohassel, and Pratyay Mukherjee. 2018. PASTA: PASsword-based Threshold Authentication. In *ACM CCS 2018: 25th Conference on Computer and Communications Security*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, Toronto, ON, Canada, 2042–2059. https://doi.org/10.1145/3243734.3243839

[4] Shashank Agrawal, Payman Mohassel, Pratyay Mukherjee, and Peter Rindal. 2018. DiSE: Distributed Symmetric-key Encryption. In *ACM CCS 2018: 25th Conference on Computer and Communications Security*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, Toronto, ON, Canada, 1993–2010. https://doi.org/10.1145/3243734.3243774

[5] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. [n. d.]. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[6] Renas Bacho and Julian Loss. 2022. On the Adaptive Security of the Threshold BLS Signature Scheme. Cryptology ePrint Archive, Paper 2022/534. https://eprint.iacr.org/2022/534 https://eprint.iacr.org/2022/534.

[7] Nir Bitansky. 2020. Verifiable Random Functions from Non-interactive Witness-Indistinguishable Proofs. *Journal of Cryptology* 33, 2 (April 2020), 459–493. https://doi.org/10.1007/s00145-019-09331-1

[8] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short Signatures from the Weil Pairing. In *Advances in Cryptology — ASIACRYPT 2001*, Colin Boyd (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 514–532.

[9] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short Signatures from the Weil Pairing. In *Advances in Cryptology – ASIACRYPT 2001 (Lecture Notes in Computer Science, Vol. 2248)*, Colin Boyd (Ed.). Springer, Heidelberg, Germany, Gold Coast, Australia, 514–532. https://doi.org/10.1007/3-540-45682-1_30

[10] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Las Vegas, NV, USA, 136–145. https://doi.org/10.1109/SFCS.2001.959888

[11] Ran Canetti. 2004. Universally Composable Signature, Certification, and Authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*. IEEE Computer Society, Los Alamitos, CA, USA, 219. https://doi.org/10.1109/CSFW.2004.24

[12] Ran Canetti, Asaf Cohen, and Yehuda Lindell. 2015. A Simpler Variant of Universally Composable Security for Standard Multiparty Computation. In *Advances in Cryptology – CRYPTO 2015, Part II (Lecture Notes in Computer Science, Vol. 9216)*, Rosario Gennaro and Matthew J. B. Robshaw (Eds.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 3–22. https://doi.org/10.1007/978-3-662-48000-7_1

[13] Cardano. [n. d.]. Ouroboros Protocol. https://cardano-foundation.gitbook.io/stake-pool-course/lessons/introduction/ouroboros.

[14] Chainlink. [n. d.]. Chainlink VRF: On-Chain Verifiable Randomness. https://developer.wax.io/en/tutorials/create-wax-rng-smart-contract/rng_basics.html.

[15] Chainlink Lab. [n. d.]. Random Rewards in Blockchain Games. https://blog.chain.link/random-rewards-in-blockchain-games/.

[16] David Chaum and Torben P. Pedersen. 1993. Wallet Databases with Observers. In *Advances in Cryptology – CRYPTO'92 (Lecture Notes in Computer Science, Vol. 740)*, Ernest F. Brickell (Ed.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 89–105. https://doi.org/10.1007/3-540-48071-4_7

[17] Cloudflare. [n. d.]. Decentralized Verifiable Randomness Beacon. https://developers.cloudflare.com/randomness-beacon/.

[18] Corestar. [n. d.]. Corestar Arcade: Tendermint-based Byzantine Fault Tolerant (BFT) middleware with an embedded BLS-based random beacon. https://github.com/corestario/tendermint.

[19] Sandro Coretti, Aggelos Kiayias, Cristopher Moore, and Alexander Russell. 2022. The Generals' Scuttlebutt: Byzantine-Resilient Gossip Protocols. In *ACM CCS 2022: 29th Conference on Computer and Communications Security*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM Press, Los Angeles, CA, USA, 595–608. https://doi.org/10.1145/3548606.3560638

[20] DAOBet (ex — DAO.Casino). [n. d.]. To Deliver On-Chain Ran- dom Beacon Based on BLS Cryptography. https://daobet.org/blog/on-chain-random-generator/.

[21] DeFi Kingdom. [n. d.]. Official DeFi Kingdoms Whitepaper. https://docs.defikingdoms.com/.

[22] Defi Kingdom. 2023. Unaffordability of existing VRF service framework. Personal Communication.

[23] Yevgeniy Dodis. 2003. Efficient Construction of (Distributed) Verifiable Random Functions. In *PKC 2003: 6th International Workshop on Theory and Practice in Public Key Cryptography (Lecture Notes in Computer Science, Vol. 2567)*, Yvo Desmedt (Ed.). Springer, Heidelberg, Germany, Miami, FL, USA, 1–17. https://doi.org/10.1007/3-540-36288-6_1

[24] Yevgeniy Dodis and Aleksandr Yampolskiy. 2005. A Verifiable Random Function with Short Proofs and Keys. In *PKC 2005: 8th International Workshop on Theory and Practice in Public Key Cryptography (Lecture Notes in Computer Science, Vol. 3386)*, Serge Vaudenay (Ed.). Springer, Heidelberg, Germany, Les Diablerets, Switzerland, 416–431. https://doi.org/10.1007/978-3-540-30580-4_28

[25] Muhammed F. Esgin, Veronika Kuchta, Amin Sakzad, Ron Steinfeld, Zhenfei Zhang, Shifeng Sun, and Shumo Chu. 2021. Practical Post-quantum Few-Time Verifiable Random Function with Applications to Algorand. In *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part II (Lecture Notes in Computer Science, Vol. 12675)*, Nikita Borisov and Claudia Diaz (Eds.). Springer, 560–578. https://doi.org/10.1007/978-3-662-64331-0_29

[26] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. 2005. Keyword Search and Oblivious Pseudorandom Functions. In *Theory of Cryptography*, Joe Kilian (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 303–324.

[27] David Galindo. [n. d.]. Distributed Verifiable Random Functions: an Enabler of Decentralized Random Beacons. https://github.com/fetchai/research-dvrf.

[28] David Galindo, Jia Liu, Mihai Ordean, and Jin-Mann Wong. 2021. Fully Distributed Verifiable Random Functions and their Application to Decentralised Random Beacons. In *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, 88–102. https://doi.org/10.1109/EuroSP51992.2021.00017

[29] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. 2003. Secure Applications of Pedersen's Distributed Key Generation Protocol. In *Topics in Cryptology – CT-RSA 2003 (Lecture Notes in Computer Science, Vol. 2612)*, Marc Joye (Ed.). Springer, Heidelberg, Germany, San Francisco, CA, USA, 373–390. https://doi.org/10.1007/3-540-36563-X_26

[30] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. 2007. Secure Distributed Key Generation for Discrete-Log Based Cryptosystems. *Journal of Cryptology* 20, 1 (Jan. 2007), 51–83. https://doi.org/10.1007/s00145-006-0347-3

[31] Sharon Goldberg, Jan Vcelak, Dimitrios Papadopoulos, and Leonid Reyzin. 2018. Verifiable random functions (VRFs). (2018).

[32] Jens Groth. 2021. Non-interactive distributed key generation and key resharing. Cryptology ePrint Archive, Paper 2021/339. https://eprint.iacr.org/2021/339 https://eprint.iacr.org/2021/339.

[33] Timo Hanke, Mahnush Movahedi, and Dominic Williams. 2018. DFINITY Technology Overview Series, Consensus System. *CoRR* abs/1805.04548 (2018). arXiv:1805.04548 http://arxiv.org/abs/1805.04548

[34] Dennis Hofheinz and Tibor Jager. 2016. Verifiable Random Functions from Standard Assumptions. In *TCC 2016-A: 13th Theory of Cryptography Conference, Part I (Lecture Notes in Computer Science, Vol. 9562)*, Eyal Kushilevitz and Tal Malkin (Eds.). Springer, Heidelberg, Germany, Tel Aviv, Israel, 336–362. https://doi.org/10.1007/978-3-662-49096-9_14

[35] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. 2017. TOPPSS: Cost-Minimal Password-Protected Secret Sharing Based on Threshold OPRF. In *ACNS 17: 15th International Conference on Applied Cryptography and Network Security (Lecture Notes in Computer Science, Vol. 10355)*, Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi (Eds.). Springer, Heidelberg, Germany, Kanazawa, Japan, 39–58. https://doi.org/10.1007/978-3-319-61204-1_3

[36] Aniket Kate, Easwar Vivek Mangipudi, Siva Mardana, and Pratyay Mukherjee. 2023. FlexiRand: Output Private (Distributed) VRFs and Application to Blockchains. Cryptology ePrint Archive, Paper 2023/1339. https://eprint.iacr.org/2023/1339 https://eprint.iacr.org/2023/1339.

[37] Neal Koblitz and Alfred Menezes. 2008. Another look at non-standard discrete log and Diffie-Hellman problems. *J. Math. Cryptol.* 2, 4 (2008), 311–326. https://doi.org/10.1515/JMC.2008.014

[38] Lisa Kohl. 2019. Hunting and Gathering - Verifiable Random Functions from Standard Assumptions with Short Proofs. In *PKC 2019: 22nd International Conference on Theory and Practice of Public Key Cryptography, Part II (Lecture Notes in Computer Science, Vol. 11443)*, Dongdai Lin and Kazue Sako (Eds.). Springer, Heidelberg, Germany, Beijing, China, 408–437. https://doi.org/10.1007/978-3-030-17259-6_14

[39] Veronika Kuchta and Mark Manulis. 2013. Unique Aggregate Signatures with Applications to Distributed Verifiable Random Functions. In *CANS 13: 12th International Conference on Cryptology and Network Security (Lecture Notes in Computer Science, Vol. 8257)*, Michel Abdalla, Cristina Nita-Rotaru, and Ricardo Dahab (Eds.). Springer, Heidelberg, Germany, Paraty, Brazil, 251–270. https://doi.org/10.1007/978-3-319-02937-5_14

[40] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. 1999. Verifiable Random Functions. In *40th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, New York, NY, USA, 120–130. https://doi.org/10.1109/SFFCS.1999.814584

[41] Moni Naor, Benny Pinkas, and Omer Reingold. 1999. Distributed Pseudo-random Functions and KDCs. In *Advances in Cryptology – EUROCRYPT'99 (Lecture Notes in Computer Science, Vol. 1592)*, Jacques Stern (Ed.). Springer, Heidelberg, Germany, Prague, Czech Republic, 327–346. https://doi.org/10.1007/3-540-48910-X_23

[42] David Niehues. 2021. Verifiable Random Functions with Optimal Tightness. In *Public-Key Cryptography - PKC 2021 - 24th IACR International Conference on Practice and Theory of Public Key Cryptography, Virtual Event, May 10-13, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12711)*, Juan A. Garay (Ed.). Springer, 61–91. https://doi.org/10.1007/978-3-030-75248-4_3

[43] Polkadot. [n. d.]. Polkadot Wiki – Randomness. https://wiki.polkadot.network/docs/learn-randomness.

[44] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. 2019. ETHDKG: Distributed Key Generation with Ethereum Smart Contracts. Cryptology ePrint Archive, Report 2019/985. https://eprint.iacr.org/2019/985.

[45] Claus-Peter Schnorr. 1990. Efficient Identification and Signatures for Smart Cards. In *Advances in Cryptology – CRYPTO'89 (Lecture Notes in Computer Science, Vol. 435)*, Gilles Brassard (Ed.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 239–252. https://doi.org/10.1007/0-387-34805-0_22

[46] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[47] Wax. [n. d.]. WAX RNG Basics. https://blog.chain.link/chainlink-vrf-on-chain-verifiable-randomness/.

A SMART-CONTRACT BASED VRF SERVICE FRAMEWORK

We depict the message flow in the VRF service framework in Figure 1. To avail of the service any user first forwards their own input to the smart contract along with the callback function to be called with the VRF output; this is indicated by step 1 in Figure 1. The smart contract may be running on any blockchain service like Ethereum. When input from the user is sent to the smart contract, after verifying the input format and checking that the same value has not been requested previously, the smart contract combines it with additional information (detailed below) forming the VRF input INP. The VRF service fetches the formatted request from the smart contract. Each of the nodes of the service computes a partial evaluation of the user input by running the $\mathsf{Part.Eval}_{sk_i}(\mathsf{INP})$ and also generates the (zero-knowledge) proof of correctness of the computation. At least $t+1$ partial evaluations are aggregated (typically using aggregator nodes) by running the $\mathsf{Aggregate}(\cdot)$ algorithm after verifying the zero-knowledge proofs. The final VRF output and an accompanying proof are sent to the smart contract, which then verifies the correctness of the VRF output. If that succeeds, it invokes the user-specified callback function with the VRF value as the input. Below we summarize the steps of Figure 1:

(1) The user forwards its own input to the smart contract.
(2) The smart contract combines user input with other values and produces the VRF input INP.
(3) The VRF service nodes fetch the input, and verify the legitimacy of INP (for example, by verifying the signature provided by the contract), and whether it was previously used.
    - Each node in the VRF committee computes the partial evaluation on INP with the zero-knowledge proof of

correct evaluation. They send them to the aggregator nodes of the VRF service.

(4) When more than t partial evaluations are obtained at an aggregator node, they are aggregated to compute the VRF output and accompanying proof of correctness. The pair is then sent to the smart contract as a response.
(5) The smart contract verifies the VRF output.
(6) If the verification succeeds, it invokes the user-specified callback function.

Output-private VRF. For the Pri-VRFcomputation, the framework stays similar to the above non-private case. However, the workflow changes slightly. In particular, initially, when the user forwards its input to the smart contract, the smart contract creates the VRF input INP and sends it back to the user, who then blinds INP and sends a pair consisting of blinded INP and an accompanying zero-knowledge proof of correct blinding. The VRF service nodes fetch this zero-knowledge proof and the blinded input. The rest of the workflow is similar to before; the smart contract will run Pre-verification PreVer instead of Verification Verify now. This is depicted in Figure 2. The callback function should run the Unblind algorithm on the blinded output inside it to obtain the VRF output. Here we describe all the fields included in the VRF input INP. **Constructing the VRF input,** INP**..** The VRF input is produced by the smart contract. Each input INP is a concatenation of the following values:

- User input – this is the user's chosen input and may be empty.
- Block-hash – this is included to ensure that no one can request the input before the block-hash is computed. This prevents one from pre-computing a VRF output to be used at a later time.
- Unique nonce –a unique nonce generated at the specific smart contract each time a VRF is called. This ensures that each VRF input is different. For this, the smart contract must keep a state (for example, a counter).
- Chain id – this distinguishes inputs generated at two different blockchains (for example, Ethereum and Solana).
- User address – this is user-specific information to distinguish between requests from different users.
- Callback function name – this is included to distinguish between two different functions coming from the same user at about the same time.
- VRF or Pri-VRF– this is a flag distinguishing between a Pri-VRFand VRF. Without this, a PVRF request may be maliciously processed as a DVRF, leading to exposure of the output.

A.1 GLOW-DVRF Framework [27, 28]

The Distributed Verifiable Random Functions implementation [27], which we call GLOW-framework, realizes the three DVRFs, Dfinity-DVRF, the DDH-DVRF, and the GLOW-DVRF [28]. The framework is written in C++ and provides implementations of the pairing-based GLOW-DVRF and Dfinity-DVRF protocols with curves BN256, BN384, and BLS12-381, and DDH-DVRF with curve Ristretto255. The pairing-based protocols are implemented using mcl [2] and RELIC [5] cryptographic libraries and the DDH-DVRF protocol with Libsodium. The code compares the performance of the DVRFs for three curves, BN256, BN384 and BL12-381. It realizes distributed key generation protocol of Gennaro et al.[30] along the consensus layer for reliable broadcast.

B MISSING PROOFS

In this section, we present the proofs that are missing from the main body.

B.1 Proof of Theorem 1

We consider a simpler case consisting of three parties, who are performing specific tasks: a client P_C who's sending/receiving inputs, a server P_S who's holding VRF keys and is performing the evaluations and an eavesdropper P_E who has no input, and is just observing the communications (we assume authenticated but no secure channels). All three parties may perform the public operations such as verifications based on the publicly available values. We build three distinct simulators $\mathcal{S}_E$, $\mathcal{S}_S$ and $\mathcal{S}_C$ for three distinct cases

- Case-1: $\mathcal{S}_E$: a corrupt P_E, when P_C and P_S are honest.
- Case-2: $\mathcal{S}_C$: a corrupt P_C, when P_E and P_S are honest.
- Case-3: $\mathcal{S}_S$: a corrupt P_S, when P_E and P_C are honest.

We argue that this is without loss of generality because, in a multiplayer scenario, any corruption can be simulated by a combination of these when a party can in fact act as any of the three roles or a combination of them. Nevertheless, for a particular execution, defined by (vk, x), a party can have exactly one of the three roles – the input provider, who provides x, is a client, the VRF-evaluator, who owns the key vk, is a server and everyone else is an eavesdropper. Therefore, for any scenario, the generic simulation strategy would be to identify the role of each corrupt party corresponding to an execution and then use the corresponding simulation strategy from above as a sub-routine. Therefore, it is sufficient to describe each simulator and argue why the simulations work, which we present next.

Case-1. P_E corrupt. In this case, we need to ensure that no eavesdropper can learn the VRF output until it is revealed, even if it can access the input x, the verification-key vk plus the entire transcript. Essentially this case specifically captures the output-privacy property we formalize in this paper. For any standard VRF scheme without output-privacy, this step can not be simulated.

The main idea here is that $\mathcal{S}_E$ simulates honest client P_C and honest server P_S just as honest parties and also simulates all random oracle (RO) queries. For the server, it runs Keygen to generate (sk, vk) and then registers $vk \in Keys[P_S]$. When it receives the message $(\mathtt{Input}, vk, x)$ from the ideal functionality, it generates uniform random ρ and correctly computes the blinded input $\widetilde{x}$. It is then given to the adversary (corrupt P_E). From the server side, the simulator computes $\widetilde{y}$ correctly and knows sk. So the corrupt eavesdropper obtains the following values before unblinding:

- public key $vk = (vk_1 = g_1^{sk}, vk_2 = g_2^{sk})$;
- input x, and subsequently $\mathsf{H}_1(x)$ through RO query;
- blinded input $\psi = \mathsf{H}_1(x)^\rho$ and the NIZK proof μ;
- blinded output $\widetilde{y} = \mathsf{H}_1(x)^{\rho sk}$.

After the unblinding phase P_E additionally gets $y = \mathsf{H}_2(\pi)$ and $\pi = \mathsf{H}_1(x)^{sk}$. The simulator only gets a uniform random y after making an explicit unblinding query on $\beta = \widetilde{y}$, namely $(\mathtt{Unblind}, x, \beta)$ to the ideal functionality. So it needs to program y as $\mathsf{H}_2(\pi)$. This is easily done as long as the adversary makes a random oracle query *after* the unblinding phase. However, if the simulator receives a random oracle query on $\mathsf{H}_2(\pi)$ before the unblinding phase, then it fails. This is because the only way for the simulator to obtain (y, π) is through an explicit Unblind query. In particular, since both P_C and P_S are honest, the simulator does not get the output during Eval or by any other means. So, before the unblind phase, there was no $(\mathtt{Reveal}, vk, x)$ query from the honest client, and hence at this point the pair (vk, x) is not marked Revealed, and consequently, the simulator can not obtain y. So, for a successful simulation, we need to prove that the probability that the eavesdropper can predict the value $\pi = \mathsf{H}_1(x)^{sk}$ (and subsequently make a RO query with that) must be negligible. We argue that, unless it is so, we can construct a PPT algorithm to break the co-CDH assumption over bilinear groups with non-negligible probability. The reduction works as follows:

Given a co-CDH instance:

$$g_1, g_1^{sk}, h_1 \in \mathbb{G}_1;\ g_2, g_2^{sk} \in \mathbb{G}_2$$

for uniform random generators g_1, h_1, g_2 and a uniform random field element $sk \in \mathbb{Z}_p$. The reduction's goal is to compute h_1^{sk}. For that, the reduction simulates as follows:

- Let sk be the secret-key of the scheme (implicitly), then $vk_1 = g_1^{sk}$ and $vk_2 = g_2^{sk}$ and $vk := (vk_1, vk_2)$ is the verification key. Note that the reduction can not mimic the simulator as it does not know sk.
- Program the RO query $\mathsf{H}_1(x) := h_1$. However, for $q = \mathsf{poly}(\kappa)$ many queries, this x must be guessed by the reduction, which is correct with probability $1/q$, incurring a loss by the same factor.
- Choose uniform random $r \leftarrow_\$ \mathbb{Z}_p$, and compute $\widehat{g}_1 = g_1^r$ and $\widehat{g}_1^{sk} := g_1^{rsk}$. Then implicitly define $\psi := \widehat{g}_1$ and $\widetilde{y} := \widehat{g}_1^{sk}$.
- Finally NIZK proof μ is simulated using the simulator KepSimu on the instance $(h_1, \psi = \widehat{g}_1)$.

We argue that the above simulation is correct. Most part of this is straightforward to see. However, the simulation of ψ is done in a manner such that the blind state ρ for which $\psi = \mathsf{H}_1(x)^\rho$ remains unknown, though $\mathsf{H}_1(x)$ is known. This is possible because the client is honest and therefore ρ must come from a uniform random distribution. By setting $\widehat{g} = g_1^r = \mathsf{H}_1(x)^\rho$, the simulator is implicitly setting $r = \rho\omega$ where $h_1 = g_1^\omega$ knowing neither ω (which is basically $\mathsf{DLOG}_{h_1}(\widehat{g}_1)$) nor ρ, but only r, which is again distributed uniformly at random. Now, clearly if the adversary makes a RO query $\mathsf{H}_2(\pi)$ where $\pi = h_1^{sk}$, the reduction checks whether $e(h_1, vk_2) = e(\pi, g_2)$, and if it satisfies the reduction output π as the answer to the Co-CDH challenger.

So, we have that:

$$\Pr[\mathcal{E}_1] \geq \frac{1}{q} \cdot \Pr[\mathcal{E}_2]$$

where the probabilities are over the randomnesses of the reduction and the adversary and the events $\mathcal{E}_1$ and $\mathcal{E}_2$ are defined as:

- $\mathcal{E}_1$: The reduction breaks Co-CDH.
- $\mathcal{E}_2$: The adversary (corrupt P_E) makes a RO query $\mathsf{H}_2(h_1^{sk})$.

and the loss q was introduced due to guessing in programming the correct challenge. This concludes the proof of this case, because $q = \mathsf{poly}(\kappa)$.

Case-2: P_C **corrupt..** In this case, the simulator simulates the honest server to corrupt client. There are two main objectives of the corrupt client: (i) to produce a malformed pair (ψ, μ); (ii) to distinguish y from a uniform random string. The first attack is prevented easily by the soundness of NIZK used. Handling the second scenario is more involved. Nevertheless, it can be proven using techniques similar to Jarecki et al. [35] and Agrawal et al. [3] who provide proofs of pseudorandomness of a very similar OPRF construction. In particular, we need to prove that if a corrupt client makes $q = \mathsf{poly}(\kappa)$ many complete evaluation queries to the server, it is unable to produce more than q "valid" triples $(x_1, y_1, \pi_1), \ldots, (x_q, y_q, y_q)$. We will argue, unless this is true, there is a PPT reduction which would break the Bilinear One-more DH (BOMDH) problem in the underlying pairing-supported groups.

The simulator simulates the honest server by sampling a key pair (sk, vk) using Keygen and registering vk for P_S as $vk \in Keys[P_S]$. In this case, the adversary sends client's message (x, ψ, μ). The simulator sends back $\widetilde{y} = \widetilde{x}^{sk}$ and set $\beta := \widetilde{y}$. Now, due to the soundness of the zero-knowledge proof, the adversary is bound to send $\psi = (x, \mathsf{H}_1(x)^\rho)$ with overwhelming probability. So, in the end it obtains $\pi = \mathsf{H}_1(x)^{sk}$ from the server interaction and then subsequently $\mathsf{H}_2(\pi)$. The hash functions H_1 and H_2 are simulated as random oracles on-the-fly in a straightforward manner. This is repeated for $q = O(\mathsf{poly}(\kappa))$ many times, after which the adversary obtains q triples $X = \{(x_i, y_i, \pi_i)\}_{i \in [q]}$. We want to bound the following probability by the probability of breaking BOMDH.

$$\Pr[\mathcal{E}_1 | \mathcal{E}_2]$$

where the events are defined as:

- $\mathcal{E}_1$: $\mathsf{Verify}(vk, (x^*, y^*, \pi^*)) = 1 \land\ (x^*, y^*, \pi^*) \notin X$
- $\mathcal{E}_2$: P_C outputs (x^*, y^*, π^*)

The reduction to BOMDH works as follows: Given an Bilinear OMDH instance

$$g_1, g_1^{sk}, \widehat{g}_1, \widehat{g}_2, \ldots, \widehat{g}_k \in \mathbb{G}_1;\ g_2, g_2^{sk} \in \mathbb{G}_2$$

for uniform generators $g_1, \{\widehat{g}_i\}_{i \in [q]}, g_2$ and uniform random field element $sk \in \mathbb{Z}_p$, and an exponentiation oracle, which on input any element $g \in \mathbb{G}_1$ returns g^{sk} (let us call this sk-exp oracle), the reduction's goal is to compute $q + 1$ pairs

$$(\bar{g}_1, \bar{g}_1^{sk}), \ldots, (\bar{g}_{q+1}, \bar{g}_{q+1}^{sk})$$

by making at most q $(k \geq q + 1)$ queries to the sk-exp oracle such that for all $i \in [q + 1]$, $\bar{g}_i \in \{\widehat{g}_1, \ldots, \widehat{g}_m\}$.

Towards that the reduction simulates our setting to the adversary (corrupt P_C) as follows:

(1) Let the secret-key be (implicitly) sk, and then the verification key becomes $vk := (vk_1, vk_2)$ where $vk_1 = g_1^{sk}$ and $vk_2 = g_2^{sk}$.
(2) Each RO query $\mathsf{H}(x_i)$ is responded with $\widehat{g}_i$. Store such x_i into a list L.
(3) When the client sends (x_i, ψ_i, μ_i), then first verify the proof using KExpVer, and if it succeeds then use the sk-exp oracle to obtain $\widetilde{y}_i := \psi_i^{sk}$ and return that to the adversary. Keep a

counter cnt to keep track of the number of distinct access to sk-exp oracle.

(4) Each RO query $\mathsf{H}_2(\alpha)$ is responded with $\mathsf{Rand}(\alpha)$. For each such query, check if there exists any $x_i \in L$ such that $e(\mathsf{H}_1(x_i), vk_2) = e(\alpha, g_2)$. If yes, then store the triple (x_i, y_i, π_i) to a list F, where $y_i := \mathsf{Rand}(\alpha)$ and $\pi_i := \alpha$. At any time $|F| > \mathsf{cnt}$, output F as the answer to the BOMDH challenger.

We argue that the above simulation is correct despite the fact that the reduction, unlike the simulator, does not have access to the secret-key sk. However, this was resolved using the sk-exp oracle. The random oracles are simulated perfectly too by plugging in the values from the challenge. Now, since the counter cnt is incremented only when a evaluation query is completed, whenever the adversary is able to produce one more valid triple, $|F| > |\mathsf{cnt}|$ and the reduction wins the BOMDH game.

So, we can claim that:

$$\Pr[\mathcal{E}] \geq \Pr[\mathcal{E}_1 \mid \mathcal{E}_2]$$

where $\mathcal{E}$ defines the event when the reduction wins the BOMDH game. This concludes the proof of this case.

Case-3. P_S corrupt. When the server is corrupt, the "unpredictability aspect" of the construction is off the table. However, even in that case, the public verifiability guarantees that the server can not produce an output that is incorrect, for example, biased towards a specific value. In other words, though unpredictability can not be guaranteed, the so-called "unbiasability" would continue to hold.

The simulator, in this case, receives the verification key vk from the adversarial server P_S and registers it with the ideal functionality within $Keys[P_S]$ while controlling the ideal server P_S. Then it simulates the honest client to the corrupt server as follows:

(1) The simulator maintains two lists I and L, where I contains pairs (x, ψ), that is the information with respect to the input and corresponding client's message (generated by the simulator); and L contains tuples (x, π, β, y), that is information from the entire evaluation, including server's message and the output with respect to an input.

(2) On receiving $(\mathtt{Input}, vk, x)$ sample a uniform random $\rho \leftarrow_\$ \mathbb{Z}_p$ and then construct $(x, \widetilde{x})$ just like an honest party, where $\widetilde{x} = (\psi := \mathsf{H}_1(x)^\rho, \mu)$. Append (x, ψ) to a list I.

(3) On receiving a message $\widetilde{y}$ from the server:
    (a) If there is an $(x, \psi) \in I$ such that $e(\widetilde{y}, g_2) = e(\psi, vk_2)$:
        (i) If $(x, *, *, *) \notin L$: then issue an evaluation query $(\mathtt{Eval}, x, vk)$ to the ideal functionality, and when the ideal functionality returns the same query, reply with $(\pi := \widetilde{y}^{1/\rho}, \beta := \widetilde{y})$. Finally, on receiving an output y from the ideal functionality store (x, π, β, y) into L.
        (ii) If there is $(x, \pi, \beta, y) \in L$: reply the evaluation query with (π, β).
    (b) Otherwise, on receiving the evaluation query from the ideal functionality, reply with $\perp$.

(4) On receiving an RO query $\mathsf{H}_2(\pi)$:
    (a) If there is an $(x, \psi) \in I$, such that $e = (\pi, g_2) = e(\mathsf{H}_1(x), vk_2)$ but $(x, *, *, *) \notin L$: then make a $(\mathtt{Eval}, vk, x)$ query to the ideal functionality and respond with $(\pi, \beta := \pi^\rho)$. On the completion of the evaluation query, receive y from the ideal functionality which it programs as an answer $\mathsf{H}_2(\pi) := y$. Append (x, π, β, y) into L.
    (b) If there exists a tuple $(x, \pi, \beta, y) \in L$, then answer with $\mathsf{H}_2(\pi) := y$.
    (c) Otherwise just respond with $\mathsf{Rand}(\pi)$.

Other queries are straightforward to handle in this case. We argue that the above simulation is correct with overwhelming probability. In particular, unless the adversarial server can guess $\mathsf{H}_1(x)^{sk}$ before observing x, the simulation would be perfect. Note that, once the server obtains x, a pair (x, ψ) gets listed in I. And then there are two cases: (i) The adversary makes a RO query $\mathsf{H}_2(\pi)$, with a valid π for which the verification equation holds, before it returns $\widetilde{y}$: in this case, the simulator first executes Step 4a. Later when it receives $\widetilde{y}$, it executes Step 3(a)ii. Clearly, in this case, the simulator is able to consistently program the random oracle and then subsequently finish the evaluation using ideal functionality. (ii) In the other case, the adversary first sends $\widetilde{y}$ and later makes a RO query $\mathsf{H}_2(\pi)$: the simulator now first executes Step 3(a)i, and later Step 4c. In this case, since the pre-verification must satisfy, the simulation would be perfect. Of course, in case the pre-verification does not satisfy, the simulator would not allow to complete the evaluation, which it ensures by sending $\perp$ to the ideal functionality. However, if the adversary can correctly predict the output of $\mathsf{H}_1(x)$ without explicitly making RO query $\mathsf{H}_1(x)$, the simulation would fail, as it could not have y without making an evaluation query to the ideal functionality – this clearly happens only with negligible probability. This concludes the proof.

B.2 Proof of Theorem 3

Similar to the proof of the non-threshold case (Theorem 1) we consider a simpler setting consisting of a single client P_C who has inputs, n servers $\mathsf{S} = \{P_1, \ldots, P_n\}$ each of whom holds a partial VRF secret-key (that is, P_i holds sk_i) after a successful DKG execution and they perform the evaluations jointly, an aggregator P_A who observes all communications and performs any verification just like the eavesdropper in the proof of Theorem 1, but additionally aggregates the partial responses, and sends the aggregated value to the client, and an eavesdropper P_E. Note that, neither the eavesdropper's functionality is a subset of the aggregator's functionality and hence we can often consider them as a single entity. We again note that for each execution corresponding to a specific (vk, x) a party plays exactly one role, although across different executions that can change. Again we argue that considering this specific setting is without loss of generality. To see that fix a specific (vk, x). Then, the overall objectives of different entities can be described as follows for the above setting:

- The honest P_C and an uncorrupted set S (that has no more than t corrupt servers) intend to compute a VRF $y = V_{sk}(x)$ correctly and securely, so that no one else can recover/predict y, given the entire transcripts that include x, without querying explicitly on x (which is prevented as (vk, x) is unique to each party.
- If the client is corrupt, and colludes with up to t malicious servers in S then she tries to break the pseudorandomness of y. In this case, the protocol should guarantee that unless

the client derives y explicitly by interacting with honest servers, the value y remains pseudorandom.

- If only the client is honest, and everyone else in the system is corrupt, then the client's objective would be to ensure that, the value y is, nevertheless, computed correctly by the server (and forwarded by the aggregator), where the adversarial server would try to produce an incorrect (and potentially biased) value $y' \neq V_{sk}(x)$ such that it appears legitimate to the client. Obviously, the unpredictability of y is impossible to guarantee in this case.

It is not hard to see that this exhausts the objectives of all parties in the system. In a more complex system, for each execution (defined by (vk, x)), the strategy would be to assign roles to each party, and then deal with them separately by different simulation strategies corresponding to each case respectively as described below:

- Case-1: $\mathcal{S}_E$: P_A, P_E and a set $\mathrm{C} \subset \mathrm{S}$ of servers are corrupt such that $|\mathrm{C}| \leq t$ (recall, t is the threshold of the system). The client and other servers in $\mathrm{H} = \mathrm{S} \setminus \mathrm{C}$ are honest.
- Case-2: $\mathcal{S}_C$: P_C and the set $\mathrm{C} \subset \mathrm{S}$ of servers are corrupt such that $|\mathrm{C}| \leq t$. The aggregator (and the eavesdropper) and rest of the servers in H are honest.
- Case-3: $\mathcal{S}_S$: $|\mathrm{C}| \geq t+1$ and the aggregator P_A (plus P_E) are corrupt. The client P_C and the servers in H ($|\mathrm{H}| < t$) honest.

Case-1: Corrupt P_E, P_A and severs in C with $|\mathrm{C}| \leq t$. We remark that this case is analogous to Case-1 in the non-threshold setting (Theorem 1), because since less than t servers are corrupted, the secret-key sk is hidden information theoretically and hence the adversary should not see the output before the `Reveal` phase even if it is provided with (vk, x) and the entire transcripts – this case specifically captures the "output-privacy" property introduced in this work.

The simulation strategy can be extended straightforwardly from the centralized PVRF analysis, except for the following two things:

- In contrast to the centralized case, here the blinded output is sent by a potentially corrupt aggregator. However, this is rather easy to simulate due to the pre-verification check. In particular, once the simulator receives an aggregated value from the corrupt aggregator, it uses pre-verification to determine the correctness of that, and if the check fails, return $\perp$ to the request ($\texttt{Aggregate}, vk, x, \ldots$).
- Since there are $f \leq t$ corrupt servers, the simulator needs to provide them the key shares $\{sk_i\}_{i\in[f]}$ (for simplicity denote the corrupt servers by $P_1, \ldots, P_f$). The simulator does this by computing each share using Shamir's secret sharing. Furthermore, the corrupt server's response can be checked using partial pre-verification.

The adversary obtains the following values in total before the `Reveal` phase:

- public key $vk = (pk = g_2^{sk}, \{pk_i = g_1^{sk_i}\}_{f<i\leq n})$;
- corrupt secret-keys $\{sk_i\}_{i\in[f]}$
- input x, and subsequently $\mathsf{H}_1(x)$ through RO query;
- blinded input $\psi = \mathsf{H}_1(x)^\rho$ and the NIZK proof of knowledge of exponent μ;
- blinded partial outputs $\widetilde{y_i} = (\widetilde{w_i} = \psi^{sk_i}, \widetilde{\pi_i})$ for $i \in \{f+1, \ldots, n\}$, where $\widetilde{\pi_i}$ is a NIZK proof of equal exponent with respect to vk_i.
- blinded output $\widetilde{y} = \mathsf{H}_1(x)^{\rho sk}$.

Again, we need to ensure that the probability that the adversary can ask an RO query to $\mathsf{H}_2(\cdot)$ on $\pi = \mathsf{H}_1(sk)$ is negligible. We reduce this again to Co-CDH akin to the centralized case. Given a Co-CDH instance:

$$g_1, g_1^{sk}, h_1 \in \mathbb{G}_1;\ g_2, g_2^{sk} \in \mathbb{G}_2$$

for uniform random generators g_1, h_1, g_2 and a uniform random field element $sk \in \mathbb{Z}_p$. The reduction's goal is to compute h_1^{sk}. For that, the reduction simulates as follows:

- Let sk be the secret-key of the scheme (implicitly), then $pk = g_2^{sk}$. Let the corrupt set be $\{P_1, \ldots, P_f\}$. Then choose t random $sk_i \leftarrow_\$ \mathbb{Z}_p$ and set for each $i \in [t]$: $vk_i := g_1^{sk_i}$. For $\in \{t+1, \ldots, n\}$ use the Lagrange interpolation in the exponent to construct $vk_i := g_1^{sk_i}$ where implicitly using sk at point 0 and for each $i \in [t]$: sk_i as the i-th polynomial output. Set $vk := (pk, \{vk_i\}_{i\in[n]})$ as the verification key.
- Program the RO query $\mathsf{H}_1(x) := h_1$.
- Choose uniform random $r \leftarrow_\$ \mathbb{Z}_p$, and compute $\widehat{g_1} = g_1^r$ and $\widehat{g}_1^{sk} := g_1^{rsk}$. Define $\psi := \widehat{g}$. Then for $i \in [n]$ define for each $i \in [n]$ $\widetilde{w}_i := \widehat{g}_1^{sk_i}$. The associated NIZK proof $\widetilde{\pi_i}$ is simulated using EqSimu if sk_i is unknown, otherwise, it is computed correctly.
- Finally NIZK proof μ is simulated using the simulator $\mathsf{KepSimu}$ on the instance $(h_1, \psi = \widehat{g_1})$.

Similar to the centralized case, we can argue that the above simulation is correct. Note that, since no partial evaluation $\mathsf{H}_1(x)^{sk_i}$ is given to the adversary, the issue (which comes up in [4, 28]) in simulating the honest partial evaluations on the challenge input does not arise. The rest of the proof for this case mimics that of the centralized case.

Case-2: Corrupt client P_C plus servers in C with $|\mathrm{C}| \leq t$. This case is again analogous to the Case-2 in the non-threshold setting (Theorem 1), as $\leq t$ corrupt servers essentially implies the secret-key is unknown to the adversary. Therefore, although the output is revealed immediately through an explicit query (no output privacy is guaranteed), the pseudorandomness of the output should still hold. In particular, we need to argue that, unless the client explicitly queries on x, it does not know $y = V_{sk}(x)$. The simulation strategy, in this case, can be adapted from the centralized case.

In this case, the simulator simulates the honest server to corrupt client and corrupt servers in C. Again, there are two main objectives of the adversary: (i) to produce a malformed pair (ψ, μ); (ii) to distinguish y from a uniform random string, while controlling up to t servers. The first attack is prevented easily by the soundness of NIZK used. To handle the second attack, we use techniques similar to Jarecki et al. [35] and Agrawal et al. [3] provide proofs of pseudorandomness of a very similar Distributed OPRF construction. In fact, we use a proof technique similar to the centralized case, but now in the threshold setting. In particular, we need to prove that if a corrupt client makes $q = \mathsf{poly}(\kappa)$ many complete (here it means for each x_i it completes at least $t+1-f$ honest partial evaluation queries, and the aggregation query subsequently) evaluation queries to the honest servers, it is unable to produce more than q "valid" triples $(x_1, y_1, \pi_1), \ldots, (x_q, y_q, y_q)$. We will argue unless this is true, there is a PPT reduction that would break the *Threshold* Bilinear One-more DH (T-BOMDH) problem in the underlying pairing-based groups.

So, following the footsteps of the proof of Theorem 1, we want to bound the following probability by the probability of breaking BOMDH.

$$\Pr[\mathcal{E}_1|\mathcal{E}_2]$$

where the events are defined as:

- $\mathcal{E}_1$: $\mathsf{Verify}(vk, (x^*, y^*, \pi^*)) = 1 \wedge (x^*, y^*, \pi^*) \notin X$ where X lists all q triples that are generated after making q "complete evaluation" queries.
- $\mathcal{E}_2$: P_C outputs a new triple (x^*, y^*, π^*)

The reduction to BOMDH works as follows: Given a T-BOMDH instance

$$g_1, g_1^{sk}, \widehat{g}_1, \widehat{g}_2, \ldots, \widehat{g}_k \in \mathbb{G}_1;\ g_2, g_2^{sk} \in \mathbb{G}_2$$

for uniform generators $g_1, \{\widehat{g}_i\}_{i\in[q]}, g_2$ and uniform random field element $sk \in \mathbb{Z}_p$, and an oracle that, on query (j, g) for any element $g \in \mathbb{G}_1$ and any index $j \in [n]$ returns $g^{D(j)}$ where D is a t-degree polynomial such that $D(0) = sk$, when any $f \le t$ points are fixed by the adversary (let us call this D-poly-exp oracle). In this case, since $|\mathsf{C}| = \{P_1, \ldots, P_f\}$, the reduction chooses f uniform random sk_j and set $D(j) := sk_j$. The reduction's goal is to compute $q + 1$ pairs

$$(\bar{g}_1, \bar{g}_1^{sk}), \ldots, (\bar{g}_{q+1}, \bar{g}_{q+1}^{sk})$$

by making at most q ($k \ge q + 1$) "complete queries" to the D-poly-exp oracle such that for all $i \in [q + 1]$, $\bar{g}_i \in \{\widehat{g}_1, \ldots, \widehat{g}_m\}$ and each complete query means at least querying for $\ge t' := t+1-f$ distinct j. Towards that the reduction simulates our setting to the adversary as follows:

(1) Let sk be the secret-key of the scheme (implicitly), then $pk = g_2^{sk}$. For the corrupt set $\mathsf{C} = \{P_1, \ldots, P_f\}$ choose f random $sk_j \leftarrow_\$ \mathbb{Z}_p$ and set for each $j \in [f]$: $vk_j := g_1^{sk_j}$. For $j \in \{f + 1, \ldots, t\}$ use the D-poly-exp oracle to get $vk_j := g_1^{D(j)}$ where implicitly using $D(0) = sk$ and for each $j \in [f]$: $D(j) = sk_j$. Note that for g_1 the oracle was queries only $t' - 1$ times, it will not be counted towards the budget. Set $vk := (pk, \{vk_j\}_{j\in[n]})$ as the verification key.
(2) Each RO query $\mathsf{H}(x_i)$ is responded with $\widehat{g}_i$. Store such x_i into a list L.
(3) When the corrupt client sends $(x_i, \psi_i, \mu_i, P_j)$, then first verify the proof using $\mathsf{KExpVer}$, and if it succeeds then use the D-poly-exp oracles to obtain $\widetilde{w}_j := \psi_i^{sk_j}$ for honest P_j and return that to the adversary. Keep a counter $\mathsf{cnt}[x_i]$ to keep track of the number of distinct access to D-poly-exp oracle. Also keep another counter cnt to track the total number of completed query. The NIZK proof of equality of exponent is simulated.
(4) Each RO query $\mathsf{H}_2(\alpha)$ is responded with $\mathsf{Rand}(\alpha)$. For each such query, check if there exists any $x_i \in L$ such that $e(\mathsf{H}_1(x_i), vk_2) = e(\alpha, g_2)$. If yes, then store the triple (x_i, y_i, π_i) to a list F, where $y_i := \mathsf{Rand}(\alpha)$ and $\pi_i := \alpha$. At any time $|F| > \mathsf{cnt}$, output F as the answer to the T-BOMDH challenger.

We argue that the above simulation is correct despite the fact that the reduction, unlike the simulator, does not have access to the secret-key sk. However, this was resolved using the D-poly-exp oracle. The random oracles are simulated perfectly too by plugging in the values from the challenge. Now, since the counter cnt is incremented only when an evaluation query is completed, that is whenever the adversary has acquired sufficient information to produce one more valid triple, $|F| > |\mathsf{cnt}|$ and the reduction wins the BOMDH game.

So, we can claim that:

$$\Pr[\mathcal{E}] \ge \Pr[\mathcal{E}_1 \mid \mathcal{E}_2]$$

where $\mathcal{E}$ defines the event when the reduction wins the BOMDH game. This concludes the proof of this case.

Case-3. P_A corrupt, $|\mathsf{C}| \ge t + 1$. In this case, (analogous to Case-2 in the centralized setting) the "unpredictability aspect" of the construction is off the table. However, even in this case the "public verifiability" guarantees that the server can not produce an output that is incorrect, for example, biased towards a specific value. In other words, though unpredictability can not be guaranteed, the so-called "unbiasability" would continue to hold.

The simulator, in this case, receives the verification key vk from the adversary and registers it with the ideal functionality within $Keys[\mathsf{S}]$ while controlling the servers in the C. Then it simulates the honest client to the servers in C as follows:

(1) The simulator maintains two lists I and L, where I contains pairs (x, ψ), that is the information with respect to the input and corresponding client's message (generated by the simulator); and L contains tuples (x, π, β, y), that is information from the entire evaluation, including server's message and the output with respect to an input.
(2) On receiving $(\mathtt{Input}, vk, x)$ sample a uniform random $\rho \leftarrow_\$ \mathbb{Z}_p$ and then construct $(x, \widetilde{x})$ just like an honest party, where $\widetilde{x} = (\psi := \mathsf{H}_1(x)^\rho, \mu)$. Append (x, ψ) to a list I.
(3) On receiving a message $\widetilde{y}$ from the aggregator P_A:
    (a) If there is an $(x, \psi) \in I$ such that $e(\widetilde{y}, g_2) = e(\psi, vk_2)$:
        (i) If $(x, *, *, *) \notin L$: then issue an evaluation query $(\mathtt{Eval}, x, vk)$ to the ideal functionality, and when the ideal functionality returns the same query, reply with $(\pi := \widetilde{y}^{1/\rho}, \beta := \widetilde{y})$. Finally, on receiving an output y from the ideal functionality store (x, π, β, y) into L.
        (ii) If there is $(x, \pi, \beta, y) \in L$: reply the evaluation query with (π, β).
    (b) Otherwise, on receiving the evaluation query from the ideal functionality, reply with $\perp$.
(4) On receiving an RO query $\mathsf{H}_2(\pi)$:
    (a) If there is an $(x, \psi) \in I$, such that $e = (\pi, g_2) = e(\mathsf{H}_1(x), vk_2)$ but $(x, *, *, *) \notin L$: then make a $(\mathtt{Eval}, vk, x)$ query to the ideal functionality and respond with $(\pi, \beta := \pi^\rho)$. On the completion of the evaluation query, receive y from the ideal functionality which it programs as an answer $\mathsf{H}_2(\pi) := y$. Append (x, π, β, y) into L.
    (b) If there exists a tuple $(x, \pi, \beta, y) \in L$, then answer with $\mathsf{H}_2(\pi) := y$.
    (c) Otherwise just respond with $\mathsf{Rand}(\pi)$.

Other queries are straightforward to handle in this case. We argue that the above simulation is correct with overwhelming probability. In particular, unless the adversarial server can guess $\mathsf{H}_1(x)^{sk}$ before observing x, the simulation would be perfect. Note that, once the server obtains x, a pair (x, ψ) gets listed in I. And then there are two cases: (i) The adversary makes a RO query $\mathsf{H}_2(\pi)$, with a valid π for which the verification equation holds, before it returns $\widetilde{y}$: in this case, the simulator first executes Step 4a. Later when it receives $\widetilde{y}$, it executes Step 3(a)ii. Clearly, in this case, the simulator is able to consistently program the random oracle and then subsequently finish the evaluation using ideal functionality. (ii) In the other case, the adversary first sends $\widetilde{y}$ and later makes a RO query $\mathsf{H}_2(\pi)$: the simulator now first executes Step 3(a)i, and later Step 4c. In this case, since the pre-verification must satisfy, the simulation would be perfect. Of course, in case the pre-verification does not satisfy, the simulator would not allow to complete the evaluation, which it ensures by seding $\perp$ to the ideal functionality. However, if the adversary can correctly predict the output of $\mathsf{H}_1(x)$ without explicitly making RO query $\mathsf{H}_1(x)$, the simulation would fail, as it could not have y without making an evaluation query to the ideal functionality – this clearly happens only with negligible probability. This concludes the proof.

B.3 (Threshold) One-More Diffie-Hellman Assumptions in Generic Group Model

We use a variant of threshold one-more Diffie-Hellman assumptions used in [3, 35]. In particular, our assumption will be over *bilinear pairing groups*, and for that, we also do not need the gap-versions.

Notations. We use notations from Agrawal et al. [3]. For $t, f, n \in \mathbb{N}$ (where $f \le t < n$) and $\mathbf{q} = (q_1, \ldots, q_n) \in \mathbb{N}^n$, define $\mathsf{Max}_{t,f}(\mathbf{q})$ to be the largest value of ℓ for which there exists binary vectors $\mathbf{u}_1, \ldots, \mathbf{u}_\ell \in \{0,1\}^n$ such that each $\mathbf{u}_i$ has hamming wight $\ge t - f$ and $\mathbf{q}$ satisfies $\mathbf{q} \ge \sum_{i=1}^{\ell} \mathbf{u}_i$. Next, we define the T-BOMDH – Threshold-Bilinear One-more Diffie Hellman assumption.

Definition 4. *(f, t, n, N, l) - T-BOMDH. Consider polynomial (in κ) size integers n, t, f, N such that $f \le t < n$ and consider bilinear pairing groups $\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ where each group has prime order p. Let g_1 and g_2 be two random generators of the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Then we say that the T-BOMDH assumption holds, if for all PPT adversary $\mathcal{A}$ the probability of the following game returning 1 is $\le \mathsf{negl}(\kappa)$.*

- *Sample uniform random secret $\alpha \leftarrow_\$ \mathbb{Z}_p$.*
- *Sample random group elements $\widehat{g}_1, \ldots \widehat{g}_N \in \mathbb{G}_1$.*
- *Provide $g_1, g_1^\alpha, g_2, g_2^\alpha, (\widehat{g}_1, \ldots, \widehat{g}_N)$ to $\mathcal{A}$.*
- *On receiving $\{(i, \alpha_i)\}_{i\in[f]}$ from $\mathcal{A}$ choose an t-degree polynomial D uniformly at random such that for all $i \in [f]$: $D(i) = \alpha_i$ and $D(0) = \alpha$.*
- *Set $\mathbf{q} := 0^n$.*
- *Give the following oracle access $O(i, x)$ to the adversary:*
  $\underline{O(i, x \in \mathbb{G})}$
  - *Increment q_i by 1.*
  - *Output x^{α_i} where $\alpha_i := D(i)$.*
- *On receiving $\{(\bar{g}, \bar{h})\}_{i\in[\ell]}$ from $\mathcal{A}$, return 1 if and only if all of the following conditions are met:*
  - *All $\bar{g}_i$ are distinct and $\ell > \mathsf{Max}(\vec{q})$.*
  - *For all $i \in [\ell] : \bar{g} \in \{\widehat{g}_1, \ldots, \widehat{g}_N\}$ and $\bar{h}_i = \bar{g}_i^\alpha$.*

Theorem 5. *[35] (f, t, n, N, l) – T-BOMDH is equivalent to (f, t, n, l, l)-T-BOMDH*

Proof. Given an (f, t, n, N, l) – T-BOMDH adversary, $\mathcal{A}$, let us construct $\mathcal{B}$, a (f, t, n, l, l) – T-BOMDH adversary using $\mathcal{A}$.

Let $\alpha \xleftarrow[\$]{} \mathbb{Z}_p$ and $g_1, g_1^\alpha, g_2, g_2^\alpha, (\hat{g}_1, \hat{g}_2, \cdots, \hat{g}_l) \xleftarrow[\$]{} \mathbb{G}_1^l$ be a challenge vector for $\mathcal{B}$. Now $\mathcal{B}$ chooses N random vectors $(\beta_{i,1}, \beta_{i,2}, \cdots, \beta_{i,l}) \xleftarrow[\$]{} \mathbb{Z}_p^l$ for all $1 \leqslant i \leqslant N$ (Note that choosing N random vectors is nothing but randomly selecting a $N \times l$ matrix over $\mathbb{Z}_p$ so let's call the matrix formed by these β vectors as M.) and defines $\hat{g}'_i := \Pi_{k=1}^{l} \hat{g}_k^{\beta_{i,k}}$ and sends $g_1, g_1^\alpha, g_2, g_2^\alpha, (\hat{g}'_1, \hat{g}'_2, \cdots, \hat{g}'_N)$ as a challenge vector to $\mathcal{A}$.

$$M \times \begin{pmatrix} \hat{g}_1 \\ \hat{g}_2 \\ \vdots \\ \hat{g}_l \end{pmatrix} = \begin{pmatrix} \hat{g}'_1 \\ \hat{g}'_2 \\ \vdots \\ \hat{g}'_N \end{pmatrix} \tag{1}$$

First $\mathcal{A}$ chooses a corrupt set of parties along with the corrupt values $F = \{(i, \alpha_i)\}$ of size f. Then $\mathcal{A}$ passes F to $\mathcal{B}$, then $\mathcal{B}$ passes it to it's challenger. Then the challenger fixes a random t-degree polynomial $D(x) \in \mathbb{Z}_p[x]$ such that $D(i) = \alpha_i$ for all indices i in F, $D(0) = \alpha$ and rest of the evaluations are all randomly fixed.

$\mathcal{B}$ answers a T-BOMDH oracle query to $O_D(.,.)$ of $\mathcal{A}$ by making the same query to itself.

Finally, if $\mathcal{A}$ outputs some $l-$ element $J \subseteq [N]$ and $v_j = (\hat{g}'_j)^\alpha$ for all $j \in J$.

Now let U be the submatrix formed by the rows corresponding to the indices in J, so U is an $l \times l$ random matrix over $\mathbb{Z}_p$. It is invertible with high probability.

Now, $\mathcal{B}$ computes

$$U^{-1} \times \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_l \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_l \end{pmatrix} \tag{2}$$

and outputs $(w_1, w_2, \cdots, w_l)$. Note $w_i = \hat{g}_i^\alpha$ with high probability iff $\mathcal{A}$ wins, so $\mathcal{B}$ wins the T-BOMDH game with high proibability iff $\mathcal{A}$ wins. □

Lemma 1. *[35] Let t be any positive integer. Then there doesn't exist $\mathbf{q} \in \mathbb{Z}_p^n$ s.t.*
(1) $||\mathbf{q}||_1 \geqslant Qt$, and
(2) $||\mathbf{q}||_{sup} \leqslant Q$,
where $Q = \mathsf{Max}_{t,0}(\mathbf{q}) + 1$.

Proof. Proof by induction on Q. If $Q = 1$ then $\mathsf{Max}_{t,f}\mathbf{q}) = 0$, which implies that there are at most $t - 1$ non zero entries in $\mathbf{q}$. So if satisfies (2), then w is at most $t - 1$, so (1) can't be satisfied. So there doesn't exist $\mathbf{q}$ where both the inequalities are satisfied simultaneously.

Now suppose that the lemma holds for $Q - 1$. Now let's check for Q. If the lemma doesn't hold, then there exists $\mathbf{q}$ which satisfies both (1) and (2). Such a $\mathbf{q}$ will have at most $t - 1$ entries that are greater than or equal to Q (otherwise those t entries can be decreased Q times, so $\mathsf{Max}_{t,0}(\mathbf{q}) \geqslant Q$). Let $\mathbf{q}'$ be $\mathbf{q}$ with largest t entries decreased. Then this $\mathbf{q}'$ satisfies both (1) and (2) as $||\mathbf{q}'||_1 =$

$(||\mathbf{q}||_1 - t) \geqslant (Q-1)t$ and $||\mathbf{q}'||_{sup} \leqslant (Q-1)$ and $\mathsf{Max}_{t,0}(\mathbf{q}') = Q-2$. □

Lemma 2. *[35] Let t be any non negative integer, n be any positive integer, $\mathbf{q}$ be an $n-$dimensional vector, $w = ||\mathbf{q}||_1, Q = \mathsf{Max}_{t+1,0}(\mathbf{q})+1$, and $\mathbf{k}$ be a $w-$dimensional vector where there are $\mathbf{q}_i$ i's as its entries $(i = 1, 2, \ldots, n)$. Then for any $w-$ dimensional vectors $\mathbf{b}_1, \ldots, \mathbf{b}_Q$, the set $V = \{\mathbf{k}^{(j)} \odot \mathbf{b}_i\}_{j\in\{0,\ldots,t\}, i\in[Q]}$ is linearly dependent, where $\mathbf{k} = (k_1, \cdots, k_w)^T$ $\mathbf{k}^{(j)} = (k_1^j, \cdots, k_w^j)^T$.*

Proof. Let M be the $w \times Q(t+1)$ dimensional matrix whose columns are vectors in V. To prove the lemma, enough to show that $rank(M) < Q(t+1)$.

Let $\mathbf{k} = (k_1 \ldots k_w)^T$

$$M = \begin{bmatrix} b_{11} & b_{12} \cdots b_{1Q} \cdots k_1^t b_{11} & k_1^t b_{12} \cdots k_1^t b_{1Q} \\ b_{21} & b_{22} \cdots b_{2Q} \cdots k_2^t b_{21} & k_2^t b_{22} \cdots k_2^t b_{2Q} \\ \vdots & \vdots \ddots \vdots \ddots \vdots & \vdots \ddots \vdots \\ b_{w1} & b_{w2} \cdots b_{wQ} \cdots k_w^t b_{w1} & k_w^t b_{w2} \cdots k_w^t b_{wQ} \end{bmatrix} \tag{3}$$

Let M_i be the sub matrix formed by rows where $k_j = i$, note that $rank(M_i) \leqslant Q$, as M_i has 1st, $Q+1 \ldots, (t-1)Q+1$ columns are just multiples of the first column, and similarly for 2nd and $Q+2, \ldots, (t-1)Q+2$ are multiples of 2nd column and so on. If $q_i > Q$, then one can select Q rows of M_i forming the matrix M_i' s.t $rank(M_i') = rank(M_i)$. For all other q_i's let $M_i' = M_i$, let q_i' be the no of rows of the matrix M_i'. Note that $q_i' = Q$ if $q_i > Q$, otherwise $q_i' = q_i$. Let $\mathbf{q}' = (q_1' \ldots q_w')^T$ and let $w' = ||\mathbf{q}'||_1$.

Now let

$$M' = \begin{bmatrix} M_1' \\ M_2' \\ \vdots \\ M_Q' \end{bmatrix} \tag{4}$$

i.e M' is the concatenation of M_i''s.

And observe that $rank(M') = rank(M)$ as we removed some rows from M which are linearly dependent on the rows of M'. But we have $\mathsf{Max}_{t+1,0}(\mathbf{q}') = \mathsf{Max}_{t+1,0}(\mathbf{q}) = Q-1$. ($q_i' \leqslant q_i$ for all i implies $\mathsf{Max}_{t+1,0}(\mathbf{q}') \leqslant Q-1$. On the other hand let $\mathbf{v}_1 + \cdots + \mathbf{v}_{Q-1}$ be the sum of binary vectors with hamming weight being $t+1$, then each co-ordinate of such a sum is at most $Q-1$, so $\mathsf{Max}_{t+1,0} \geqslant Q-1$.) Now by lemma 1, we have $w' < Q(t+1)$. So $rank(M') = rank(M) \leqslant w' < Q(t+1)$. □

Lemma 3. *[35] Let $t, n, \mathbf{q}, w, Q$ be same as in lemma 2. Then there doesn't exist a $Q \times w$ matrix A and $w \times Q$ matrix B and an invertible $w \times w$ diagonal matrix K s.t $AB = I$ and $AKB = AK^2B = \cdots = AK^tB = O$, where K is the matrix which has q_i i's as diagonal entries (for all $i = 1$ to n), I and O are $Q \times Q$ identity and zero matrix respectively.*

Proof. Suppose such a K, A and B exists

$$K = \begin{bmatrix} k_1 & & \\ & \ddots & \\ & & k_w \end{bmatrix} \tag{5}$$

and let $\mathbf{a}_1^T, \ldots, \mathbf{a}_Q^T$ be the rows of A, $\mathbf{b}_1, \ldots, \mathbf{b}_Q$ be the columns of B, and $\mathbf{k} = (k_1 \ldots k_w)^T$. Then all these $\mathbf{a}_i$'s , $\mathbf{b}_j$'s and $\mathbf{k}$ are w-dimensional column vectors.

Let $\mathbf{k}^j = (k_1^j \ldots k_w^j)^T, \mathbf{k}^j \odot \mathbf{b}_i = K^j \mathbf{b}_i$ and $V = \{\mathbf{k}^j \odot \mathbf{b}_i\}_{j=0\ldots t, i=1\ldots Q}$. Then the conditions $AB = I$ and $AKB = AK^2B = \cdots = AK^tB = O$ can be rewritten as

$$\mathbf{a}_i^T \mathbf{b} = \begin{cases} 1 & \mathbf{b} = \mathbf{b}_i \\ 0 & \mathbf{b} \in V \setminus \{\mathbf{b}_i\} \end{cases} \tag{6}$$

for all $1 \leqslant i \leqslant Q$. Therefore $\mathbf{b}_i$ doesn't belong to linear span of $V \setminus \{\mathbf{b}_i\}$. Suppose that $W = V \setminus \{\mathbf{b}_1, \ldots, \mathbf{b}_Q\}$ is a linearly independent set. Then $W \cup \{\mathbf{b}_1\}$ is also linearly independent set, similarly $W \cup \{\mathbf{b}_1, \mathbf{b}_2\}$ is a linearly independent set as $\mathbf{b}_2$ is independent of $W \cup \{\mathbf{b}_1\}$. And finally V is a linearly independent set. Contradicts lemma 2. So $W = \{\mathbf{k}^j \odot \mathbf{b}_i\}_{1\leqslant j\leqslant t, 1\leqslant i\leqslant Q}$ is a linearly dependent set.

Then there exists x_{ij} for all $1 \leqslant j \leqslant t, 1 \leqslant i \leqslant Q$ at least one of them is non zero s.t

$$\sum_{j=1}^{t} \sum_{i=1}^{Q} x_{ij} \mathbf{k}^j \odot \mathbf{b}_i = 0. \tag{7}$$

Since non of the $\mathbf{k}$'s entries are zeroes,

$$\begin{aligned} \sum_{j=1}^{t} \sum_{i=1}^{Q} x_{ij} \mathbf{k}^j \odot \mathbf{b}_i &= \sum_{j=1}^{t} \sum_{i=1}^{Q} \sum_{l=1}^{w} x_{ij} k_l^j b_{li} \\ &= k_l \Big(\sum_{j=1}^{t} \sum_{i=1}^{Q} \sum_{l=1}^{w} x_{ij} k_l^{j-1} b_{li}\Big) = 0 \text{ for all l.} \end{aligned} \tag{8}$$

.

So

$$\sum_{i=1}^{Q} x_{i1} \mathbf{b}_i + \sum_{j=0}^{t-1} \sum_{i=1}^{Q} x_{ij+1} \mathbf{k}^j \odot \mathbf{b}_i = 0. \tag{9}$$

Since $\mathbf{b}_i$s are linearly independent of W, so $x_{i1} = 0$ for all $1 \leqslant i \leqslant Q$.

So

$$\sum_{j=0}^{t-1} \sum_{i=1}^{Q} x_{ij+1} \mathbf{k}^j \odot \mathbf{b}_i = 0. \tag{10}$$

Again repeating the same trick as above(i.e $k_i \neq 0$), we get all $x_{ij} = 0$ for all $0 \leqslant j \leqslant t, 1 \leqslant i \leqslant Q$. So V can't be linearly dependent either. So such an A, B and K doesn't exist. □

Theorem 6 (Generic group hardness of (0, t, n, l, l)T-BOMDH). *Let $\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be generic bilinear pairing map, where each group is of prime order p. We use $\xi^{(1)}(a), \xi^{(2)}(b)$ and $\xi^{(T)}(c)$ for $a, b, c \in \mathbb{Z}_p$ to denote elements in $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ respectively, where $\xi^{(1)}(.), \xi^{(2)}(.)$ and $\xi^{(T)}(.)$ are random injective mappings from $\mathbb{Z}_p$ to bit strings of sufficient size(Note: for all the three mappings domain and codomain are same).*

- *Group operation oracle, which on input $(\xi^{(i)}(a), \xi^{(i)}(b))$, outputs $\xi^{(i)}(a+b)$ for all $i \in \{1, 2, T\}$.*
- *Bilinear pairing oracle, which on input $(\xi^{(1)}(a), \xi^{(2)}(b))$, outputs $\xi^{(T)}(ab)$.*
- *T-BOMDH oracle, which on input $(i, \xi^{(1)}(a))$, outputs*

$$O_D(i, \xi^{(1)}(a)) = \xi^{(1)}(D(i)a)$$

.

If $\mathsf{Adv}_{\mathcal{A}}^{O_D(.,.)}(\mathsf{t,n,l,r,s,p})$ *is the probability that* $\mathcal{A}(\xi^{(1)}(1), \xi^{(1)}(D(0)), \xi^{(2)}(1), \xi^{(2)}(D(0)), \xi^{(1)}(u_1), \cdots, \xi^{(1)}(u_l))$ *outputs*

$$(\xi^{(1)}(D(0)u_1), \cdots, \xi^{(1)}(D(0)u_l))$$

after making r group operation queries to all three groups combined, e bilinear pairing queries and q_i *queries to* $O_D(i,.)$ $(w = \sum_{i=1}^{n} q_i)$ *s.t* $\mathsf{Max}_{t,0}(\mathbf{q}) < l$*, then*

$$\mathsf{Adv}_{\mathcal{A}}^{O_D(.,.)}(\mathsf{t,n,l,r,e},\mathbf{q},\mathsf{p}) \leqslant \frac{(\mathsf{ewl})+2}{\mathsf{p}} + \frac{(\mathsf{l+e+r+w})^2\mathsf{ew}}{2\mathsf{p}}$$

Proof. Let's construct $\mathcal{B}$ which simulates the real challenger while interacting with $\mathcal{A}$. It maintains a list $\mathscr{L} := \{(F_s, \xi_s^{(i)})\}_{s=1,\ldots,\sigma}$, where $F_s(U_1, \ldots, U_l, A_0, A_1, \ldots, A_t)$ is a polynomial of degree at most ew, and $\xi_s^{(i)}$'s are random elements in $\mathbb{G}_i$. Initially $\mathcal{B}$ sets $\sigma = l+2$ and initializes list $\mathscr{L}$ by setting $F_1 = 1, F_2 = D(0) = a_0, F_3 = u_1, \cdots, F_{l+2} = u_l$, and picks

$$\xi_1^{(1)}, \xi_1^{(2)}, \xi_2^{(1)}, \xi_2^{(2)}, \xi_3^{(1)}, \cdots, \xi_{l+2}^{(1)}$$

as random elements in respective groups corresponding to upper indices and $a_0, \cdots, a_t \xleftarrow{\$} \mathbb{Z}_p$. $\mathcal{B}$ sends

$$\xi_1^{(1)}, \xi_1^{(2)}, \xi_2^{(1)}, \xi_2^{(2)}, \xi_3^{(1)}, \cdots, \xi_{l+2}^{(1)}$$

to $\mathcal{A}$ as

$$\xi^{(1)}(1), \xi^{(2)}(1), \xi^{(1)}(D(0)), \xi^{(2)}(D(0)), \xi^{(1)}(u_1), \cdots, \xi^{(1)}(u_l).$$

Then $\mathcal{A}$ makes following three types of oracle queries to $\mathcal{B}$ on values that are previously obtained from $\mathcal{B}$:

- Group operation query: $\mathcal{A}$ inputs (i, s_1) and (i, s_2). Then $\mathcal{B}$ computes $F_{\sigma+1} = F_{s_1} + F_{s_2}$, if there exists $t \leqslant \sigma$ such that $(F_t, \xi_t^{(i)}) \in \mathscr{L}$, then $\mathcal{B}$ outputs $\xi_t^{(i)}$ to $\mathcal{A}$. Otherwise $\mathcal{B}$ picks random group element $\xi_{\sigma+1}^{(i)}$ from $\mathbb{G}_i$ which is different from the previously chosen ones and sends it to $\mathcal{A}$ and sets $\sigma++, r++$.
- Bilinear pairing operation query: $\mathcal{A}$ inputs $(1, s_1)$ and $(2, s_2)$. Then $\mathcal{B}$ computes $F_{\sigma+1} = F_{s_1}F_{s_2}$. If there exists $t \leqslant \sigma$ such that $(F_t, \xi_t^{(T)}) \in \mathscr{L}$, then $\mathcal{B}$ outputs $\xi_t^{(T)}$ to $\mathcal{A}$. Otherwise $\mathcal{B}$ picks random group element $\xi_{\sigma+1}^{(T)}$ from $\mathbb{G}_T$ which is different from the previously chosen ones and sends it to $\mathcal{A}$ and sets $\sigma++, e++$.
- $O_D(.,.)$ oracle query: $\mathcal{A}$ inputs $k \in [n]$ and $s \in [\sigma]$. Then $\mathcal{B}$ computes $F_{\sigma+1} = (\mathbf{L}_k^T\mathbf{a})F_s$. If there exists $t \leqslant \sigma$ such that $(F_t, \xi_t^{(1)}) \in \mathscr{L}$, then $\mathcal{B}$ outputs $\xi_t^{(1)}$ to $\mathcal{A}$. Otherwise $\mathcal{B}$ picks random group element $\xi_{\sigma+1}^{(1)}$ from $\mathbb{G}_T$ which is different from the previously chosen ones and sends it to $\mathcal{A}$ and sets $\sigma++$, q_k++. Where $\mathbf{L}_k^T = [1\ k \cdots k^t]^T$ and $\mathbf{a} = [a_0 \cdots a_t]^T$.

$\mathcal{A}$ finally outputs $(F_{s_1}, \cdots, F_{s_l})$, and it wins if $F_{s_i} = u_iD(0)$ for all $i \in [l]$.

Now we analyze the probability that $\mathcal{A}$ succeeds for a random assignment of $(u_1, \cdots, u_l, a_0, \cdots, a_t)$.

Note that output of $\mathcal{A}$ comes from three types of oracle queries mentioned above. Therefore F_s is a linear combination of $v_1, \cdots, v_w$, u_1, u_l 1 and $D(0)$. Where $v_i (i = 1, \ldots, w)$ is the value obtained from $O_D(.,.)$ oracle queries and other u_j's are obtained from group oerations and bilinear pairing operations. So

$$F_s = \sum_{i=1}^{w} \alpha_i^{(s)} v_i + \sum_{i=0}^{r+e} \gamma_i^{(s)} u_i,$$

where

$$v_i = \sum_{Z \subseteq [i] \text{ s.t } i \in Z} [(\sum_{j=0}^{r+e} \beta_{jZ}^{(i)} u_j) \prod_{l \in Z} (\mathbf{L}_{k_l}^T \mathbf{a})]$$

where α_i's, γ_i's and β_{jZ}'s are all field elements specified by $\mathcal{A}$. (And suppose that in the ith $O_D(.,.)$ oracle query, $\mathcal{A}$'s second is k_i; then $\mathbf{L}_{k_i}^T\mathbf{a}$ must appear, so $i \in Z$ holds in the expression of v_i.)

$\mathcal{A}$ wins iff $F_{s_i} = D(0)u_i$ for all $i = 1, \ldots, l$. Suppose that there exists an $i \in [l]$ s.t. $deg(F_{s_i}) > 1$, but $\mathcal{A}$ still wins, i.e. as the $deg(D(0)u_i) = 1$, then the above case happens only when the both the polynomials evaluates to the same values on the randomness over u_i's and a_j's (abuse of notation , we are using same variables for polynomial variables and field elements u_i and a_j). So $1 \leqslant deg(F_{s_i} - D(0)u_i) \leqslant ew)$ for a fixed chosen random u_i's random $D(0)$ will be solution with probability $\frac{ew}{p}$. Since there are l possible values of i, so the probability $deg(F_{s_i}) > 1$ but $\mathcal{A}$ still wins is $\frac{ewl}{p}$.

Now consider the case where $deg(F_{s_i}) \leqslant 1$ for all i. Let v_i' be v_i with degree greater than 1 eliminated and similarly for u_i's. But in the case of v_i's only single term is left, where $Z = \{1\}$, i.e.,

$$v_i' = (\sum_{j=0}^{l} \beta_{j\{i\}}^{(i)} u_j)(\mathbf{L}_{k_i}\mathbf{a}).$$

Then

$$F_s = \sum_{i=1}^{w} \alpha_i^{(s)} v_i' + \sum_{i=0}^{l} \gamma_i^{(s)} u_i.$$

We can rewrite the above expression F_s in matrix form below. Note that all the terms in the above expression are of degree $\leqslant 1$. And denote $\beta_{j\{1\}}^{(i)}$ as $\beta_j^{(i)}$.

$$\begin{bmatrix} F_{s_1} \\ \vdots \\ F_{s_l} \end{bmatrix} = A \begin{bmatrix} v_1' \\ \vdots \\ v_w' \end{bmatrix} + C\mathbf{u} + \begin{bmatrix} \gamma_0^{(s_1)} \\ \vdots \\ \gamma_0^{(s_l)} \end{bmatrix}, \tag{11}$$

$$\begin{bmatrix} v_1' \\ \vdots \\ v_w' \end{bmatrix} = (B\mathbf{u} + \mathbf{b}_0) \odot \begin{bmatrix} \mathbf{L}_{k_1}^T\mathbf{a} \\ \vdots \\ \mathbf{L}_{k_w}^T\mathbf{a} \end{bmatrix}, \tag{12}$$

where

$$A = \begin{bmatrix} \alpha_1^{(s_1)} \cdots \alpha_w^{(s_1)} \\ \vdots \ddots \vdots \\ \alpha_1^{(s_l)} \cdots \alpha_w^{(s_l)} \end{bmatrix}_{l \times w}, \quad B = \begin{bmatrix} \beta_1^{(1)} \cdots \alpha_l^{(1)} \\ \vdots \ddots \vdots \\ \alpha_1^{(w)} \cdots \alpha_l^{(w)} \end{bmatrix}_{w \times l}$$

$$C = \begin{bmatrix} \gamma_1^{(1)} \cdots \gamma_l^{(1)} \\ \vdots \ddots \vdots \\ \gamma_1^{(l)} \cdots \gamma_l^{(l)} \end{bmatrix}_{l \times l}, \mathbf{u} = \begin{bmatrix} u_1 \\ \vdots \\ u_l \end{bmatrix}, \mathbf{b}_0 = \begin{bmatrix} \beta_0^{(1)} \\ \vdots \\ \beta_0^{(w)} \end{bmatrix}$$

Let $\mathbf{b} = B\mathbf{u} + \mathbf{b}_0$. Now substitute eq. (12) into eq. (11), then we get

$$\begin{bmatrix} F_{s_1} \\ \vdots \\ F_{s_l} \end{bmatrix} = A \left(\mathbf{b} \odot \begin{bmatrix} \mathbf{L}_{k_1}^T\mathbf{a} \\ \vdots \\ \mathbf{L}_{k_w}^T\mathbf{a} \end{bmatrix} \right) + C\mathbf{u} + \begin{bmatrix} \gamma_0^{(s_1)} \\ \vdots \\ \gamma_0^{(s_l)} \end{bmatrix}. \tag{13}$$

Note that

$$\begin{bmatrix} \mathbf{L}_{k_1}^T \mathbf{a} \\ \vdots \\ \mathbf{L}_{k_w}^T \mathbf{a} \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^t a_i k_1^i \\ \vdots \\ \sum_{i=0}^t a_i k_w^i \end{bmatrix} = a_0 \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} + a_1 \begin{bmatrix} k_1 \\ \vdots \\ k_w \end{bmatrix} + \cdots + a_t \begin{bmatrix} k_1^t \\ \vdots \\ k_w^t \end{bmatrix}$$

So

$$\mathbf{b} \odot \begin{bmatrix} \mathbf{L}_{k_1}^T \mathbf{a} \\ \vdots \\ \mathbf{L}_{k_w}^T \mathbf{a} \end{bmatrix} = a_0 K^0 \mathbf{b} + \cdots + a_t K^t \mathbf{b}, \tag{14}$$

where

$$K = \begin{bmatrix} k_1 & & \\ & \ddots & \\ & & k_w \end{bmatrix}$$

so now substituting eq. (14) in eq. (13) we get

$$\begin{aligned} \begin{bmatrix} F_{s_1} \\ \vdots \\ F_{s_l} \end{bmatrix} &= A(a_0 K^0 \mathbf{b} + \cdots + a_t K^t \mathbf{b}) + C\mathbf{u} + \begin{bmatrix} Y_0^{(s_1)} \\ \vdots \\ Y_0^{(s_l)} \end{bmatrix} \\ &= a_0 A K^0 \mathbf{b} + \cdots + a_t A K^t \mathbf{b} + C\mathbf{u} + \begin{bmatrix} Y_0^{(s_1)} \\ \vdots \\ Y_0^{(s_l)} \end{bmatrix}. \end{aligned} \tag{15}$$

substituting back $\mathbf{b} = B\mathbf{u} + \mathbf{b}_0$ in eq. (15) we get

$$\begin{bmatrix} F_{s_1} \\ \vdots \\ F_{s_l} \end{bmatrix} = a_0 A K^0 B\mathbf{u} + a_0 A K^0 \mathbf{b}_0 + \cdots + a_0 A K^t B\mathbf{u} + a_t A K^t \mathbf{b}_0 + C\mathbf{u} + \begin{bmatrix} Y_0^{(s_1)} \\ \vdots \\ Y_0^{(s_l)} \end{bmatrix}$$

For a fixed random $\mathbf{u}$, the right side of the above equation is a linear function of $\mathbf{a}$. If $\mathcal{A}$ wins, then

$$\begin{bmatrix} F_{s_1} \\ \vdots \\ F_{s_l} \end{bmatrix} = a_0 \mathbf{u},$$

then upon comparing the last two equations, we get

$$(AB - I)\mathbf{u} + A\mathbf{b}_0 = AKB\mathbf{u} + AK\mathbf{b}_0 = \cdots = AK^t B\mathbf{u} + AK^t \mathbf{b}_0 = 0 \tag{16}$$

For fixed random $\mathbf{u}$ the above equations evaluate to 0 with probability $\frac{1}{p}$. The other case where the above equations become zero is when $AB - I = AKB = \ldots AK^t B = 0$. But according to lemma 3, at least one of $AB - I, AKB, \ldots, AK^t B$ isn't zero. Then that particular matrix equation evaluates to zero with probability for a fixed $\mathbf{u}$ is again $\frac{1}{p}$.

So in both the cases where $deg \leqslant 1$ and $deg > 1$ advantage of the adversary interacting $\mathcal{B}$ is $\frac{(ewl)+2}{p}$.

So far the above computations are for a fixed random $\mathbf{u}$, now if we fix both $\mathbf{u}$ and $\mathbf{a}$, then we have to account for two distinct polynomials evaluating to the same value for random $\mathbf{u}$ and $\mathbf{a}$. This event happens with probability $\binom{\sigma}{2} \frac{ew}{p}$, where $\sigma \leqslant l + e + r + w$. Thus

$$\mathsf{Adv}_{\mathcal{A}}^{O_{\mathsf{D}}(.,.)}(\mathsf{t}, \mathsf{n}, \mathsf{l}, \mathsf{r}, \mathsf{e}, \mathbf{q}, \mathsf{p}) \leqslant \frac{(\mathsf{ewl}) + 2}{\mathsf{p}} + \frac{(\mathsf{l} + \mathsf{e} + \mathsf{r} + \mathsf{w})^2 \mathsf{ew}}{2\mathsf{p}}$$

$\square$

Now the generic group hardness of (f, t, n, l, l)T-BOMDH follows from theorem 6 as (f, t, n, l, l)T-BOMDH is same as $(0, t - f, n, l, l)$T-BOMDH. And from theorem 6, theorem 5 generic group hardness of (f, t, n, N, l)T-BOMDH follows.