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
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. 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 $y = V_k(x)$. 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. Furthermore, due to privacy, one may extend the private output $y = V_k(x)$ to generate multiple pseudorandom values $z_1 = \text{PRG}(y, 1)$, $z_2 = \text{PRG}(y, 2)$, ... 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.

**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 liveliness: 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 $(y_i, \pi_i)$ 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 $\pi$ – 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 verified. Let us stress that each such randomness $z_i$ can only be verified using $y$. Given that one can compute all $z_i$, 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 DVRFs (DVRF) [28] and Distributed Oblivious PRFs [35], and easily 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 0.5ms for the partial evaluation on each VR 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 $\sim 300$µ sec for blinding and generating the zero-knowledge proof of correct blinding before forwarding it to the VR 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 Kingdoms [21], which crucially requires distributing random rewards (e.g. NFTs) in Blockchain Games [15].

**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 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 $e : G_1 \times G_2 \rightarrow G_T$, each a cyclic group of prime order $p$. The VRF secret-key $sk$ is chosen at random from $Z_p$, whereas the public verification key $vk$ is pair of group elements $(zk_1 = g_2^{tk_1}, zk_2 = g_2^{tk_2})$ where $g_1, g_2$ are random generators of groups $G_1$ and $G_2$ respectively. A requester with an input $x$ first blinded her input to generate $y = H_1(x)^\rho \in G_1$ for a hash function $H_1(\cdot)$, random blind/mask $\rho$ in $Z_p$ and produces a NIZK proof of knowledge of $\rho$ (Schnorr’s
proof for knowledge of exponent [45]) and sends that to the 
smart-contract, which forwards it to the VRF server. The server first verifies the NIZK proof, and if that succeeds, sends back \( \tilde{y} = (H_1(x))^{s_k} \). The contract verifies the response by using bilinear 
pairing \( e(\tilde{y}, g_2^{s_k}) = e(y, g_2) \) (exactly the same as BLS signatures [8]) and if 
that succeeds, then it forwards \( \tilde{y} \) to the requester, who then unblinds to get \( \pi = \tilde{y}^{\frac{1}{r}} = H_1(x)^{s_k} \); then derives \( y \) as \( H_2(\pi) \). The final VRF verification is again running the BLS verification, but now 
with different components: \( e(H_1(x), g_2^{s_k}) = e(\pi, g_2) \) plus the hash \( H_2(\pi) = y \). 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 \( s_k \); of \( s_k \) (this can be 
typically achieved through a distributed key-generation (DKG) 
protocol such as [30]). The verification key now is of the form 
\( \omega_k = (pk = g_2^{s_k}, \omega_k = g_1^{s_k}, \omega_k = g_1^{s_k}, \ldots) \). 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 \( \tilde{y} \) along 
with \( x \) and a NIZK proof, each of them checks the proof as before, 
and if that succeeds, now uses its share \( s_k \) to compute a partial 
value \( \tilde{w}_i = \psi^{s_k} \). Additionally, it computes another NIZK for equality 
of exponent (we use Chaum-Pederson [16]) between \( \tilde{w}_i \) and 
\( \omega_k \). Then it sends over the partial evaluation \( \tilde{w}_i \) plus the proof 
to the aggregator, who verifies each NIZK proof, and if at least 
\( t + 1 \) of them succeed, then combines the corresponding partial 
evaluations via Lagrange interpolation in the exponent to compute 
\( y = H_1(x)^{s_k} \). Given \( x, \psi, \tilde{y} \), 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 \( H_1(x)^{s_k} \), given \( g_1^{s_k}, H_1(x)^{s_k \cdot \rho} \) and \( H_1(x)^{\rho} \) (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 \( H_1(x) \). 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)DVRF to a Pri-(D)DVRF 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 veri-
fication: the server would send \( \pi \) as above, plus a NIZK proof 
of the equality of exponent with \( \psi^{s_k} \) (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 (\( x, y, \pi \)), 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 aggre-
gated using, for example, a SNARK proof, leading to a significant 
challenge in terms of constructing an efficient SNARK for that spe-
cific 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 Dinfinity-DVRF vs GLOW-
DVRF. This is because the NIZK proof works in the group \( G_1 \) 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 signature. Later, this was improved by the work of Dodis and Yam-polskiy [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” 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 \(v = (v_1, v_2, \ldots)\). For a boolean vector \(v \in \{0, 1\}^n\), its hamming weight is given by the number of 1s in \(v\). For any set \(S\), \(|S|\) denotes its cardinality.

We denote the security parameter by \(\kappa\). We assume that every algorithm takes \(\kappa\) 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 \(\negl(\kappa)\) 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 \(\poly(\kappa)\) to denote a polynomial function of the security parameter \(\kappa\).

We use \(D(x) = y\) or \(y = D(x)\) to denote the evaluation of a specifically deterministic algorithm \(D\) on input \(x\) to produce output \(y\). Often we use \(x := val\) to denote the assignment of a value \(val\) to the variable \(x\). We write \(R(x) \rightarrow y\) or \(y \leftarrow R(x)\) 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.

---

\(\text{Footnote:} \) 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 $\kappa$, for any probabilistic algorithm $\mathcal{A}$ that runs in $O(\text{poly}(\kappa))$ time, the probability that $\mathcal{A}$ can solve the given problem instance is upper bounded by $\leq \text{negl}(\kappa)$.

Polynomial Interpolation. A polynomial $P(x)$ over a finite field $\mathbb{F}$ of degree $t$ can be expressed as $P(x) = c_0 + c_1 x + c_2 x^2 \ldots c_t x^t$, 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 $\text{Rand}(\cdot)$. For compact presentation we use a one-way random function, denoted $\text{Rand}(\cdot)$, in our Pri-VRF and Pri-DVRF definitions. For a given domain $\text{Dom}$ and $\text{Rng}$, the function has a table $T$ containing pairs $(x, y)$ where $x \in \text{Dom}$ and is initialized to $\emptyset$. It works as follows:

\[
\text{Rand}(x \in \text{Dom}).
\]

- If there exists $(x, y) \in T$, return $y$.
- Else return a uniform random $y \leftarrow \text{Rng}$ 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 ITT. For a detailed formalization, we refer to [10, 12].

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

Bilinear Pairing, Co-CDH, XDH. Our constructions rely on bilinear pairing. We consider three groups $\mathcal{G}_0, \mathcal{G}_1, \mathcal{G}_T$, among which the source groups $\mathcal{G}_0, \mathcal{G}_1$ and $\mathcal{G}_T$ 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 $e : \mathcal{G}_0 \times \mathcal{G}_1 \rightarrow \mathcal{G}_T$ which is:

- **bilinear:** for any $a, b \in \mathbb{Z}_p$,
  
  \[ e(g_0^a, g_1^b) = e(g_0, g_1)^{ab} \]

- **non-degenerate:** $e(g_0, g_1) \neq 1_T$ where $1_T$ is the (multiplicative) identity of group $\mathcal{G}_T$.

We require the Co-CDH assumption over bilinear groups. The assumption states that: for uniform random $g_1, h_1 \leftarrow \mathcal{G}_1$ and $g_2 \leftarrow \mathcal{G}_2$ and uniform random $e \leftarrow \mathbb{Z}_p$: given $(g_1, h_1, g_2^e, g_2^f)$ it is computationally hard to compute $h_1^e$. The corresponding decisional assumption, which requires the adversary to distinguish between $h_1^e$ and a random $h^e \leftarrow \mathcal{G}_1$ given $g_1, g_2^e, g_2^f$ 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 $\mathcal{G}_0$ and $\mathcal{G}_1$ 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.

- **Share($s$) $\rightarrow$ $(s_1, \ldots, s_n)$**: This randomized algorithm takes any field element $s \in \mathbb{Z}_p$ as input. Then it works as follows:
  - Sample a uniform random polynomial $P(x) = s + c_1 x + \ldots + c_t x^t$ 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 Share($s$).
- **Recon($s_1, \ldots, s_n$) $\rightarrow$ $s$: $\leftarrow L$**: The reconstruction is a deterministic procedure which takes a bunch of shares $s_1, \ldots, s_t$ each from the field $\mathbb{Z}_p$ as input and then executes the following steps:
  - If $t \leq t$, then output $\perp$;
  - Otherwise, if $t > t$, use the Lagrange coefficients to compute: $s = P(0) = \sum_{i=1}^{n} \lambda_{0,k,s}s_k$;
  - Finally output $s$.

Security: The scheme provides the following security guarantee: For any uniform random secret $s \leftarrow \mathbb{Z}_p$, if $(s_1, \ldots, s_n) \leftarrow$ 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 \( \text{inst} = (g, h) \in \mathbb{G}^2 \) and witness \( w = k \in \mathbb{Z}_p \) such that \( k = \text{DLOG}_g(h) \). Also, consider a hash function \( H : \{0,1\}^* \rightarrow \mathbb{Z}_p \). So the set of public parameters is defined as \( pp := (H, \mathbb{G}) \), which is provided as an input to all algorithms below. Then the proof system consists of the following two algorithms and a simulator:

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

**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 consider a hash function \( H : \{0,1\}^* \rightarrow \mathbb{Z}_p \). So the set of public parameters is defined as \( pp := (H, \mathbb{G}) \), which is provided as an input to all algorithms implicitly. Then, the proof system consists of the following two algorithms and a simulator:

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

### 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 \( f, n, t, \in \mathbb{N} \) (where \( f \leq t < n \)) and \( q = (q_1, \ldots, q_n) \in \mathbb{N}^n \), define \( \text{Max}_{1 \leq f} (q) \) to be the largest value of \( f \) for which there exists binary vectors \( u_1, \ldots, u_t, u \in \{0,1\}^n \) such that each \( u_i \) has hamming wight \( \geq t - f \) and \( q \) satisfies \( q \geq \sum_{i=1}^t u_i \). Next, we define the T-BOMDH – Threshold-Bilinear One-more Diffie Hellman assumption.

**Definition 1** (T-BOMDH). Consider polynomial \((\alpha, k)\) size integers \( n, t, f, N \) such that \( f \leq t < n \) and consider bilinear pairing groups \( \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T \) 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 \( A \) the probability of the following game returning 1 is \( \leq \text{neg}(\alpha) \).

<table>
<thead>
<tr>
<th>( q = 0^n )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sample uniform random secret</strong> ( a \leftarrow \mathbb{Z}_p )</td>
</tr>
<tr>
<td><strong>Sample random group elements</strong> ( \tilde{g}_1, \tilde{g}_2, \tilde{g}_3, \tilde{g}_4 \in \mathbb{G}_1 )</td>
</tr>
<tr>
<td><strong>Provide</strong> ( g_1, g_2, g_3, g_4, \tilde{g}_1, \tilde{g}_2, \tilde{g}_3, \tilde{g}_4 ) to ( A )</td>
</tr>
<tr>
<td><strong>On receiving</strong> ( { (i, a_i) }_{i \in {f }} ) from ( A ), choose ( a ) to be of degree polynomial ( D ) uniformly at random such that for all ( i \in {f } ): ( D(i) = a_i ) and ( D(0) = a )</td>
</tr>
<tr>
<td><strong>Set</strong> ( q = 0^n )</td>
</tr>
<tr>
<td><strong>Give the following oracle access</strong> ( O(i, x) ) to the adversary: ( O(i, x) \in \mathbb{G} )</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>On receiving</strong> ( { (\tilde{g}<em>i) }</em>{i \in {f }} ) from ( A ), return 1 if and only if all of the following conditions are met:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**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 ex-
ponent 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 \(x'\) 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 \(t = 0\),
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 Pri-
vate 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
verify whether \(y\) is output \(\bot\) of the following variables, all of which are initialized to
\(P\) functionality interacts with parties, generally denoted by
ideal functionality \(F\) of the communica-
tion should satisfy pseudorandom
output should be \(y\) at the end, the client obtains
server holds a long-term secret key
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.

Ideal Functionality \(\mathcal{F}_\text{perf}\). All guarantees are captured by the
ideal functionality \(\mathcal{F}_\text{perf}\), which is detailed in Figure 3. The ideal
functionality interacts with parties, generally denoted by \(P\) and a
simulator \(S\). The phrase “any ITI”, denoted by \(M\), refers to either
a party \(P\) or the simulator \(S\). The ideal functionality keeps track of
the following variables, all of which are initialized to \(\bot\) (or 0)
implicitly.

1. \(\text{Keys}(M)\): contains the verification keys owned by any ITI \(M\).
We say that a verification key \(ok\) is unique if there exists a unique
\(M\), for which \(ok \in \text{Keys}(M)\).

2. \(T[ok, x]\): contains entries of the form \((y, \Pi, B)\) corresponding to
a verification key \(ok\) and an input \(x\). Each entry contains an
output \(y\), and sets \(\Pi = \{x_1, x_2, \ldots\}, B = \{b_1, b_2, \ldots\} etc.
The set \(\Pi\) contains all proofs for the tuple \((ok, x, y)\), whereas the set \(B\)
contains the corresponding server messages. We say that a proof
\(\pi\) is unique whenever there exists a unique pair \((ok, x)\) such that

\[
\pi \in T[ok, x].
\]

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

Some intuitions on \(\mathcal{F}_\text{perf}\). 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 \((ok, x)\) due to the uniqueness criteria, which makes a

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 $\beta$, 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 $\text{Reveal}$ and $\text{Unblind}$ phase. A minor difference with their approach is that we merge the $\text{Eval}$ and $\text{RegKey}$ 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 $F_{\text{perf}}$ 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., $F_{\text{perf}}$) $y$ 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:

- **Key Generation** $(1^\kappa) \rightarrow (sk, ok)$: The key-generation algorithm outputs a pair of keys $(sk, ok)$ – $sk$ is the secret key and $ok$ is the verification key.

- **Blind** $(1^\kappa, x) \rightarrow (st, \bar{x})$: This algorithm processes the input $x$ to offer a secret state $st$ and a public output $\bar{x}$.

- **InpVer** $(1^\kappa, (x, \bar{x})) = 1/0$: The input verification algorithm verifies whether the pair $(x, \bar{x})$ is correctly computed, and returns 1 if and only if the check succeeds.

- **Eval** $(ok, sk, \bar{x}) \rightarrow \bar{y}$: The evaluation algorithm uses the secret key $sk$ (and possibly also the verification key $ok$) on the blinded input $\bar{x}$ to produce a blinded output $\bar{y}$.

- **PreVer** $(ok, (x, \bar{x}, \bar{y})) = 1/0$: The pre-verification algorithm verifies whether the computed blinded value $\bar{y}$ is correct for the pair $(x, \bar{x})$ and verification key $ok$.

- **Unblind** $(x, \bar{y}, st) = (y, \pi)$: The deterministic unblinding algorithm takes a blinded output $\bar{y}$ and a secret state $st$ (typically generated during the blinding procedure) plus an input $x$ as inputs and outputs an output-proof pair $(y, \pi)$.

- **Verify** $(ok, (x, y, \pi)) = 1/0$: The verification algorithm takes the public verification key $ok$ 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 $ok$ 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 $y = \text{Vrf}(x)$. A party can be a server or a client in different executions. The client runs Blind and sends over the pair $(x, \bar{x})$ to the server, which first checks whether the blinded input was correctly computed using InpVer on the pair $(x, \bar{x})$. In fact, anyone else can perform this check. If the verification succeeds, then the server runs Eval on $\bar{x}$ to produce $\bar{y}$ and subsequently sends that over to $P_C$. Anyone (may or may not be the same as $P_C$) can run PreVer on $(ok, \bar{x}, \bar{y})$ to publicly verify whether the server’s computation was correct. At any point, the client may unblind by running Unblind on $(x, \bar{y})$ to get the final output-proof pair $(y, \pi)$. Once the pair $(y, \pi)$ is made public, anyone can check whether $y$ was correctly computed from $x$ by publicly running Verify on $(x, y, \pi)$. Importantly a combination of the checks Verify, PreVer, and InpVer together allow public verification of the full transcript $(x, \bar{x}, \bar{y}, y)$.

The real world execution is described in a protocol $\Pi$ (Figure 4).

**Figure 4: A real world Pri-VRF protocol $\Pi$.**

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

$$\text{REAL}_{A,E} \approx_{\text{IDEAL}} \text{F}_{\text{perf}}, S, E$$

**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 $(ok, x)$, and formalized by $\text{Inp}[ok, x]$. 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 5Note 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 $\kappa$. An efficiently computable Type-3 bilinear pairing $e: G_1 \times G_2 \rightarrow G_T$, where the groups $G_1, G_2, G_T$ are multiplicative groups and each of prime order $p$, $g_1$ and $g_2$ are randomly chosen generators of $G_1$ and $G_2$ respectively. Without loss of generality we assume that all algorithms have the public parameters as input.

**Hash functions**: $H_1 : \{0,1\}^* \rightarrow G_1$; $H_2 : G_1 \rightarrow \{0,1\}^*$; $H_3 : \{0,1\}^* \rightarrow \mathbb{Z}_p$.

**A secure NIZK proof system** (KExProve, KExpVer) for knowledge of exponent in group $G_1$. The public parameter for this proof system is $\{H_3, G_1\}$.

**Construction**
- $\text{KeyGen}(\kappa) \rightarrow (sk, ok)$: Sample $sk \leftarrow \mathbb{Z}_p$ and set $ok = (ok_1, ok_2) := (g_1^{sk}, g_2^{sk})$.
- $\text{Blind}(1^n, x) \rightarrow (st, x)$: Sample a uniform random $\rho \leftarrow \mathbb{Z}_p$ and set $\psi := H_1(x)^\rho$.
  - $\text{Compute } e(\psi, \rho)^\rho = e(\psi, \psi)$.
- $\text{InpVer}(\psi, \rho, \psi) = 1/0$:
  - Parse $\psi, \rho, \psi$.
  - Then run KExpVer on the instance $(H_1(x), \psi) \rightarrow \text{it fails output } 0; \text{otherwise output } 1$.
- $\text{Eval}(ok, sk, x) \rightarrow \tilde{y}$:
  - Parse $\psi, \rho, \psi$.
  - Compute $\tilde{y} := \psi^{sk}$.
- $\text{PreVer}(ok, \tilde{y}, g_2) = 1/0$ : Return the check:
  - $e(\tilde{y}, g_2) = e(x, ok_2)$.
- $\text{Unblind}(\tilde{y}, st) = (y, \pi)$:
  - Parse $\rho = st$.
  - Compute $\pi := \tilde{y}^{\rho^{-1}}$.
  - Compute $y := H_1(\pi)$.
- $\text{Verify}(ok, y, \pi, y) = 1/0$ : Return the check:
  - $(e(H_1(x), ok_2) = e(\pi, g_2)) \land (H_2(\pi) = y)$.

**Figure 5: Our Pri-VRF construction**

DVRF proposed in [28]. We argue it satisfies our Pri-VRF definition as captured by our ideal functionality $\mathcal{F}_\text{perf}$. 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 $\mathcal{F}_\text{perf}$ 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 $(ok, x)$, 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 have 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 $S = \{P_1, \ldots, P_n\}$ who jointly hold a VRF key $sk$ jointly in a $t$ out of $n$ fashion, for example using a secret-sharing scheme.\footnote{The access structure can be generalized to other settings, but in this paper, we stick to $t$ out of $n$ threshold access structure.} 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 $\pi$ privately, such that no one except the client knows $y$ or $\pi$. 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 $(y, \pi)$, 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 $\mathcal{F}_\text{perf}$. 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...
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 \( ok \) and shares \( sk_1, sk_2, \ldots \) of a secret-key \( sk \).

**Ideal Functionality** \( F_{pdvrf} \). All guarantees, informally described above, are captured by the ideal functionality \( F_{pdvrf} \) in Figure 6. The ideal functionality interacts with parties, denoted generally by \( P = P_1, P_2, \ldots, P_n \) and a simulator \( S \). Sometimes, to stress on 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 \( F_{prof} \). 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 \( 0 \)) implicitly.

1. **Keys**\[M], Keys\[S\]: contains the public verification keys owned by any entity \( M \) or a set of servers \( S = \{ P_1, \ldots, P_n \} \).

We note that, if \( ok \in Keys[S] \) then \( ok \in Keys[P_i] \) for each \( P_i \in S \). We say that a verification key \( ok \) is unique if there exists a unique set of servers \( S \), for which \( ok \in Keys[S] \) — this is extended from \( F_{prof} \), which considers uniqueness corresponding to parties.

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

3. **T_{part}**\([ok, x, P_i] \): 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 \( \beta \) is defined naturally with respect to the triple \((ok, x, P_i) \).

Note that, since \( ok \) is unique to a set of servers \( S \), we do not need to specify the set of servers.

4. **Inp**\([ok, x] \): denotes the party (client) who sent the pair \((ok, x) \) 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:

- **Keygen**\((1^s, n, t) \rightarrow (ok, sk_1, \ldots, sk_n) \): The key-generation algorithm (implemented by a DKG protocol) outputs a verification key \( ok \) and \( n \) shares \( sk_1, \ldots, sk_n \) of the secret-key \( sk \) where the sharing is \( t \) out of \( n \) threshold.
- **Blind**\((1^s, x) \rightarrow (st, \tilde{x}) \): This algorithm processes an input \( x \) to offer a secret state \( st \) and a public output \( \tilde{x} \).
- **InpVer**\((1^s, (x, \tilde{x})) = 1/0 \). The input verification algorithm verifies whether a pair \((x, \tilde{x}) \) is correctly formed and returns 1 if and only if the check succeeds.
- **Part.Eval**\((ok, sk_i, \tilde{x}) \rightarrow \tilde{y}_i \): The partial evaluation algorithm uses the partial secret-key \( sk_i \) on the blinded input \( \tilde{x} \) to produce a blinded partial output \( \tilde{y}_i \).
- **PartPreVer**\((ok, (x, \tilde{y}_i)) = 1/0 \): There is a partial pre-verification algorithm which verifies whether the computed blinded partial value \( \tilde{y}_i \) is correct for the input \( x \) and verification key \( ok \).
- **Aggregate**\((ok, \{(\tilde{y}_i)\}_{E,S}) = \tilde{y} \). The aggregation algorithm gathers a set of blinded values to produce an aggregated blinded value \( \tilde{y} \).
- **PreVer**\((ok, (x, \tilde{y})) = 1/0 \): There is a pre-verification algorithm, similar to PVRF, which verifies whether the computed blinded value \( \tilde{y} \) is correct for the blinded input \( x \) and verification key \( ok \).
- **Unblind**\((\tilde{y}, st) = (y, \pi) \). The deterministic unblinding algorithm takes a blinded output \( \tilde{y} \) and a secret-state \( st \) (typically generated during the blinding procedure) and then outputs an output-proof pair \((y, \pi) \).
- **Verify**\((ok, (x, y, \pi)) = 1/0 \): The verification algorithm takes the public verification key \( ok \) 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,\(^1\) that securely implements Keygen such that the verification key \( ok \) is made public and each \( P_i \) gets a secret key share \( sk_i \). Let us denote this set of parties by \( S = \{ P_1, \ldots, P_n \} \). At any point, a client \( Q \) with an input \( x \) may run Blind to generate \( \tilde{x} \) and subsequently sends over \((x, \tilde{x}) \) for evaluation to the servers in \( S \) (with verification key \( ok) \). Server \( P_i \) in set \( S \) first runs the input-verification InpVer on \((x, \tilde{x}) \), and if that succeeds, runs Part.Eval on \((x, \tilde{x}) \) with \((ok, sk_i) \) to generate a blinded partial output \( \tilde{y}_i \), which it sends back. The values \((\tilde{y}_1, \tilde{y}_2, \ldots) \) 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 \( \tilde{y}_i \) with respect to \((ok, x) \), and if there are at least \( t \) many correct such values, then it may produce a blinded output \( \tilde{y} \) (otherwise it outputs \( \perp \)). The client, when obtaining \( \tilde{y} \), 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 \((y, \pi) \). The triple \((x, y, \pi) \) 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 \((x, \tilde{x}, (\tilde{y}_i)), i, \tilde{y}, y, \pi) \). The real world execution is described in a protocol \( \Pi \) (Figure 7).

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

\[
\text{REAL}_{\Pi, A, E} \approx_{\text{c}} \text{IDEAL}_{F_{pdvrf}, S, E}
\]

**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

\(^1\) 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.
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

---

**Ideal Functionality** $F_{\text{ideal}}$

**Key Generation.** Upon $(\text{KeyGen}, ok, S)$ where $S \subseteq \{P_1, \ldots, P_n\}$ from $S$ when $ok$ is unique:

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

**Input:** Upon $(\text{Input}, ok, x)$ from any client $Q$:

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

**Partial Evaluation.** Upon $(\text{PartEval}, ok, x)$ from any $P_i$; if $ok \notin \text{Keys}[P_i]$ or $\text{Inp}[ok, x] = \perp$ then exit; otherwise send $(\text{PartEval}, ok, x, P_i)$ to $S$.

If $S$ returns $\perp$ then send $\perp$ to $P_i$; otherwise:

1. When $S$ returns $\beta_i$, then $\beta_i$ is unique and append it to $T_{\text{part}}[ok, x, P_i]$; otherwise exit.
2. Send $\beta_i$ to $P_i$.

**Partial Pre-Verification:** Upon $(\text{PartPreVerify}, ok, x, \beta_i)$ from any $M$, forward this to $S$ and when $S$ returns $\phi$, then do as follows:

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

**Aggregation:** Upon $(\text{aggregate}, ok, x, \beta_1, \ldots, \beta_l)$ from any ITI $M$: if $t < t+1$, then return $\perp$ to $M$, else forward the message to $S$ when $S$ returns $\beta$ and $\pi$, if either of $\beta$ or $\pi$ is not unique, then exit, otherwise:

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

**Pre-Verification:** Upon $(\text{PreVerify}, ok, x, \beta)$ from any $M$, forward this to $S$, and when $S$ returns $\phi$, do:

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

**Reveal:** Upon $(\text{Reveal}, ok, x)$ from any client $Q$: send this to $S$, when $S$ returns the message, mark $(ok, x)$ as Revealed.

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

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

**Verification:** Upon $(\text{Verify}, ok, x, y, \pi)$ from any $M$ forward this to $S$, and upon receiving $\phi$ from $S$:

1. If there is a $S$ for which $ok \in \text{Keys}[S]$ and $T[v, x]$ is defined then do as follows:
   - a) If $(y, \pi) \in T[ok, x]$ set $f = 1$
   - b) Else, if $\phi = 1$ and $\pi$ is unique: then append $\pi$ to $T[ok, x]$ 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
FlexiRand: Output Private (Distributed) VRFs and Application to Blockchains

Pri-DVRF Protocol II

- **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 $pk$ as a output of Keygen($1^k$, $n$, $t$).
- **Request**: Any party $Q$ (which may or may not be part of $S$) with an input $x$ runs $(st, \tilde{x}) \leftarrow$ Blind($1^k$, $x$). Then it sends $\tilde{x}$ to all parties $P_1, \ldots, P_n$ in set $S$.
- **Response**: Each party $P_i$, on receiving a request on a blinded input $\tilde{x}$ executes the following steps:
  - Run InpVer ($1^k$, $(x, \tilde{x})$), if it returns $0$, then do nothing. Else go to the next step.
  - Run $y_i \leftarrow$ Part.Eval($ok, sk_i, \tilde{x}$) and then send $y_i$ 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 $(y_1, y_2, \ldots)$ executes the following steps:
  - Initiate a set $S = \emptyset$.
  - For each $i$ run PartPreVer($ok, (x, y_i)$) - if it returns $1$, then append $i$ into $S$, else do nothing.
  - If $S$ contains at least $t+1$ elements, then run $y :=$ Aggregate($ok, (y_i)_{i \in S}$).
  - If $A \not\in Q$ then send $y$ to $Q$, otherwise store $y$.
- **Pre-verification**: $Q$, once gets $y$ runs PreVer($ok, (\tilde{x}, y)$) - if it outputs $0$, then discard $\tilde{y}$, otherwise unblind $(y, \pi) :=$ Unblind($\tilde{y}$, $st$). When necessary it publishes $(x, y, \pi)$.
- **Verification**: Anyone, on input $(x, y, \pi)$ can run Verify($ok, (x, y, \pi)$) and if and only if that returns 1 concludes that the triple is legitimate.

Theorem 3. Our Pri-DVRF construction, described in Fig. 8, UC-realizes $F_{pdvrf}$ 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 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 $F_{pdvrf}$. Formally we state the following theorem, which is proven in the Appendix B.2.

Figure 7: A real world Pri-DVRF protocol II.
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 $g_{sk_i}^r$ was publicly available, and PartPreVer was performed using bilinear pairing $e(x, g_{sk_i}^r) = e(u, g_2)$. However, this would incur concrete computation overhead because verifying a NIZK proof of equality amounts to 4 exponentiations in the group $G_1$, 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


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 $G_1$ 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 ~307μsec on an average for computing $H(x)^y$ (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 ~403μsec. Unblinding by the requester involves one exponentiation and takes on an average ~146μsec. 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 ~253μsec 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 ~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 ~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.

**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.**

<table>
<thead>
<tr>
<th></th>
<th>Input-Generation</th>
<th>Partial-Eval.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLOW-DVRF (MCL)</td>
<td>-</td>
<td>253.304 μsec</td>
</tr>
<tr>
<td></td>
<td>307.079 μsec</td>
<td>403.059 μsec</td>
</tr>
<tr>
<td>Pri-DVRF (MCL)</td>
<td>-</td>
<td>2.5978 μsec</td>
</tr>
<tr>
<td>GLOW-DVRF (RELIC)</td>
<td>1.67658 msec</td>
<td>1.30304 msec</td>
</tr>
<tr>
<td>Pri-DVRF (RELIC)</td>
<td>2.5978 μsec</td>
<td>2.5978 μsec</td>
</tr>
</tbody>
</table>

**Table 2: Average total time taken to generate one PRVF value.**
### 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


### 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.

<table>
<thead>
<tr>
<th></th>
<th>n</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLOW-DVRF (MCL)</td>
<td>8</td>
<td>1.71 msec</td>
</tr>
<tr>
<td>Pri-DVRF (MCL)</td>
<td>8</td>
<td>1.89 msec</td>
</tr>
<tr>
<td>GLOW-DVRF (RELIC)</td>
<td>16</td>
<td>2.97 msec</td>
</tr>
<tr>
<td>Pri-DVRF (RELIC)</td>
<td>16</td>
<td>3.12 msec</td>
</tr>
<tr>
<td>GLOW-DVRF (RELIC)</td>
<td>16</td>
<td>18.42 msec</td>
</tr>
<tr>
<td>Pri-DVRF (RELIC)</td>
<td>16</td>
<td>19.89 msec</td>
</tr>
<tr>
<td>GLOW-DVRF (MCL)</td>
<td>32</td>
<td>5.47 msec</td>
</tr>
<tr>
<td>Pri-DVRF (MCL)</td>
<td>32</td>
<td>5.66 msec</td>
</tr>
<tr>
<td>GLOW-DVRF (RELIC)</td>
<td>32</td>
<td>35.63 msec</td>
</tr>
<tr>
<td>Pri-DVRF (RELIC)</td>
<td>32</td>
<td>36.77 msec</td>
</tr>
<tr>
<td>GLOW-DVRF (MCL)</td>
<td>64</td>
<td>10.46 msec</td>
</tr>
<tr>
<td>Pri-DVRF (MCL)</td>
<td>64</td>
<td>10.66 msec</td>
</tr>
<tr>
<td>GLOW-DVRF (RELIC)</td>
<td>64</td>
<td>72.87 msec</td>
</tr>
<tr>
<td>Pri-DVRF (RELIC)</td>
<td>64</td>
<td>74.34 msec</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>n</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCL</td>
<td>8</td>
<td>0.159 sec</td>
</tr>
<tr>
<td>RELIC</td>
<td>8</td>
<td>0.171 sec</td>
</tr>
<tr>
<td>MCL</td>
<td>16</td>
<td>0.244 sec</td>
</tr>
<tr>
<td>RELIC</td>
<td>16</td>
<td>0.277 sec</td>
</tr>
<tr>
<td>MCL</td>
<td>32</td>
<td>0.399 sec</td>
</tr>
<tr>
<td>RELIC</td>
<td>32</td>
<td>0.695 sec</td>
</tr>
<tr>
<td>MCL</td>
<td>64</td>
<td>1.08 sec</td>
</tr>
<tr>
<td>RELIC</td>
<td>64</td>
<td>2.33 sec</td>
</tr>
</tbody>
</table>

$H(x^r)$, and the verification involves just two pairings. Each pairing operation costs 108K gwei; hence, the pairing-based verification which involves two pairing operations, costs ~250K gwei including 20K gwei for storing a 256 bit value. The requestor 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 ~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 ~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 ~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.
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 (described below) forming the VRF input. 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 Part.Eval vk,i (INP) and also generates the (zero-knowledge) proof of correctness of the computation. At least t + 1 partial evaluations are aggregated (typically using aggregator gates) by running the Aggregate() 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 output; this is indicated by step 1 in Figure 1.

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 (described below) forming the VRF input. 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 Part.Eval vk,i (INP) and also generates the (zero-knowledge) proof of correctness of the computation. At least t + 1 partial evaluations are aggregated (typically using aggregator gates) by running the Aggregate() 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-VRF computation, 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 \( \text{INP} \) and sends it back to the user, who then blinks \( \text{INP} \) 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 \( \text{INP} \). **Constructing the VRF input, \( \text{INP} \).** The VRF input is produced by the smart contract. Each input \( \text{INP} \) 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-VRF and 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 \( S_E, S_S \) and \( S_C \) for three distinct cases

- **Case-1**: \( S_E \): a corrupt \( P_E \), when \( P_C \) and \( P_S \) are honest.
- **Case-2**: \( S_S \): a corrupt \( P_S \), when \( P_C \) and \( P_E \) are honest.
- **Case-3**: \( S_C \): a corrupt \( P_C \), when \( P_S \) and \( P_E \) 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 \((R, x)\), 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 \( v \), 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 \( v \) and 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 be simulated.

The main idea here is that \( S_E \) 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 \( vk \in \text{Keys}[P_S]\). When it receives the message \((\text{Input}, x, y)\) from the ideal functionality, it generates uniform random \( \rho \) and correctly computes the blinded input \( \tilde{x} \). It is then given to the adversary (corrupt \( P_E \)). From the server side, the simulator computes \( \tilde{y} \) correctly and knows \( sk \). So the corrupt eavesdropper obtains the following values before unblinding:

- **public key** \( vk = (sk_1 = g_1^{sk}, sk_2 = g_2^{sk}) \);
- **input** \( x \), and subsequently \( H_1(x) \) through RO query;
- **blinded input** \( \psi = H_1(x)\mu \) and the NIZK proof \( \mu \);
- **blinded output** \( \tilde{y} = H_1(x)^{vsk} \).

After the unblinding phase \( P_E \) additionally gets \( y = H_2(\pi) \) and \( \pi = H_1(x)^{\psi} \). The simulator only gets a uniform random \( y \) after making
an explicit unblinding query on $\beta = \tilde{y}$, namely (Unblind, $x, \beta$) to the ideal functionality. So it needs to program $y$ as $H_2(\pi)$. 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 $H_2(\pi)$ before the unblinding phase, then it fails. This is because the only way for the simulator to obtain $(y, \pi)$ 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 unblinding phase, there was no (Reveal, $sk, x$) query from the honest client, and hence at this point the pair $(sk, x)$ is not marked Revealed, and consequently, the simulator cannot obtain $y$. So, for a successful simulation, we need to prove that the probability that the eavesdropper can predict the value $\pi = H_1(x)^{sk}$ (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 $ok_1 = g_1^{sk}$ and $ok_2 = g_2^{sk}$ and $ok := (ok_1, ok_2)$ is the verification key. Note that the reduction can not mimic the simulator as it does not know $sk$.
- Program the RO query $H_1(x) := h_1$. However, for $q = \text{poly}(x)$ 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 $\tilde{g}_1 = g_1^r$ and $\tilde{g}_2^{sk} = g_2^{sk \cdot r}$. Then implicitly define $\psi := \tilde{g}_1$ and $\tilde{y} := \tilde{g}_2^{sk}$. Finally NIZK proof $\mu$ is simulated using the simulator KepSimu on the instance $(h_1, \psi, \tilde{y})$.

We argue that the above simulation is correct. Most part of this is straightforward to see. However, the simulation of $\psi$ is done in a manner such that the blind state $\rho$ for which $\psi = H_1(x)^{\rho}$ remains unknown, though $H_1(x)$ is known. This is possible because the client is honest and therefore $\rho$ must come from a uniform random distribution. By setting $\tilde{g} = g_1^r = H_1(x)^\rho$, the simulator is implicitly setting $r = \rho \omega$ where $h_1 = g_1^r$ knowing neither $\omega$ (which is basically DLOG$_h(\tilde{g}_1)$) nor $\rho$, but only $r$, which is again distributed uniformly at random. Now, clearly if the adversary makes a RO query $H_2(\pi)$ where $\pi = h_1^{sk}$, the reduction checks whether $\epsilon(h_1, sk) = \epsilon(\pi, g_2)$, and if it satisfies the reduction output $\pi$ 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 randomness 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 $H_2(h_1^{sk})$.

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

**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 $(\psi, \mu)$; (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 $q = \text{poly}(x)$ many complete evaluation queries to the server, it is unable to produce more than $q$ “valid” triples $(x_1, y_1, \pi_1), \ldots, (x_q, y_q, \pi_q)$. 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 $(sk, ok)$ using Keygen and registering $ok$ for $P_S$ as $ok \in \text{Keys}(P_S)$. In this case, the adversary sends client’s message $(x, \psi, \mu)$. The simulator sends back $\tilde{y} = x^k$ and set $\beta := \tilde{y}$. Now, due to the soundness of the zero-knowledge proof, the adversary is bound to send $\psi = (x, H_1(x)^{\rho})$ with overwhelming probability. So, in the end it obtains $\pi = H_1(x)^{sk}$ from the server interaction and then subsequently $H_2(\pi)$. The hash functions $H_1$ and $H_2$ are simulated as random oracles on-the-fly in a straightforward manner. This is repeated for $q = O(\text{poly}(x))$ 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:

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

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

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

for uniform generators $g_1, g_1^{sk}, \tilde{g}_i \in \mathbb{G}_1$, $g_2, g_2^{sk} \in \mathbb{G}_2$. Each RO query $H(x_i)$ is responded with $\tilde{g}_i$. Store such $x_i$ into a list $L$.

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 $ok := (ok_1, ok_2)$ where $ok_1 = g_1^{sk}$ and $ok_2 = g_2^{sk}$.
2. Each RO query $H(x_i)$ is responded with $\tilde{g}_i$. Store such $x_i$ into a list $L$.
3. When the client sends $(x_i, \psi, \mu_i)$, then first verify the proof using $\text{KExpVer}$, and if it succeeds then use the $sk$-exp oracle to obtain $\tilde{y}_i := \psi_i^{sk}$ 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 $H_2(\alpha)$ is responded with Rand($\alpha$). For each such query, check if there exists any $x_i \in L$ such that $e(H_1(x_i), \text{ok}_2) = e(\alpha, g_2)$. If yes, then store the triple $(x_i, y_i, \pi_i)$ to a list $F$, where $y_i := \text{Rand}(\alpha)$ and $\pi_i := \alpha$. At any time $|F| > cnt$, 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 specific value. In other words, though unpredictability can not be satisfied, the “unpredictability aspect” of the construction is off the table. However, even in this case, the reduction would not allow to complete the evaluation, which it ensures by sending $\bot$ to the ideal functionality. However, if the adversary can correctly predict the output of $H_1(x)$ without explicitly making a query to the ideal functionality – this clearly happens only with negligible probability. This concludes the proof.

**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 cannot produce an output that is incorrect, for example, biased towards a specific value. In other words, though unpredictability cannot be guaranteed, the so-called “unbiasability” would continue to hold.

The simulator, in this case, receives the verification key $\text{vk}$ from the adversarial server $P_2$ and registers it with the ideal functionality within $\text{Keys}_i[P_S]$ 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 $L$ and $I$, where $L$ contains pairs $(x, \psi)$, that is the information with respect to the input and corresponding client’s message (generated by the simulator); and $I$ contains tuples $(x, \pi, \beta, y)$, that is information from the entire evaluation, including server’s message and the output with respect to an input.

(2) On receiving $(\text{Input}, \text{ok}, x)$ sample a uniform random $\rho \in \mathbb{Z}_p$ and then construct $(x, \bar{x})$ just like an honest party, where $\bar{x} = (\psi := H_1(x)^\rho, \mu)$. Append $(x, \psi)$ to a list $I$.

(3) On receiving a message $\gamma$ from the server:

   (a) If there is an $(x, \psi) \in I$ such that $e(\gamma, g_2) = e(\psi, \text{ok}_2)$:

      (i) If $(x, \pi, \beta, y) \notin L$: then issue an evaluation query $(\text{Eval}_1, x, \text{ok})$ to the ideal functionality, and when the ideal functionality returns the same query, reply with $(\pi := \gamma^{1/\rho}, \beta := y)$. Finally, on receiving an output $y$ from the ideal functionality store $(x, \pi, \beta, y)$ into $L$.

      (ii) If there is $(x, \pi, \beta, y) \in L$: reply the evaluation query with $(\pi, \beta)$.

   (b) Otherwise, on receiving the evaluation query from the ideal functionality, reply with $\bot$.

(4) On receiving an RO query $H_2(\alpha)$:

   (a) If there is an $(x, \psi) \in I$, such that $e = (\pi, g_2) = e(H_1(x), \text{ok}_2)$ but $(x, \pi, \beta, y) \notin L$: then make a $(\text{Eval}_1, \text{ok}, x)$ query to the ideal functionality and respond with $\pi^\rho$. On the completion of the evaluation query, receive $y$ from the ideal functionality which it programs as an answer $H_2(\pi) := y$. Append $(x, \pi, \beta, y)$ into $L$.

   (b) If there exists a tuple $(x, \pi, \beta, y) \in L$, then answer with $H_2(\pi) := y$.

   (c) Otherwise just respond with 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 $H_1(x)^{sk}$ before observing $x$, the simulation would be perfect. Note that, once the server obtains $x$, a pair $(x, \psi)$ gets listed in $I$. And then there are two cases: (i) The adversary makes a RO query $H_2(\pi)$, with a valid $\pi$ for which the verification equation holds, before it returns $\gamma$: in this case, the simulator first executes Step 4a. Later when it receives $\gamma$, it executes Step 3(aii). 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 $\gamma$ and later makes a RO query $H_2(\pi)$: the simulator now first executes Step 3(aii), 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 $\bot$ to the ideal functionality. However, if the adversary can correctly predict the output of $H_1(x)$ without explicitly making a 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 $S = \{P_1, \ldots, P_n\}$ each of whom holds a partial VRF secret-key (that is, $P_i$ holds $s_{ki}$) 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 $(\text{ok}, x)$ 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 $(\text{ok}, x)$.

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 $y = V_{\text{sk}}(x)$ 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 $(\text{ok}, x)$ 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 $(sk, x)$), 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:** $\text{SE}: P_A, P_S$ and a set $C \subseteq S$ of servers are corrupt such that $|C| \leq t$ (recall, $t$ is the threshold of the system). The client and other servers in $H = S \setminus C$ are honest.

- **Case-2:** $\text{SC}: P_C$ and the set $C \subseteq S$ of servers are corrupt such that $|C| \leq t$. The aggregator (and the eavesdropper) and rest of the servers in $H$ are honest.

- **Case-3:** $\text{SS}: |C| = t + 1$ and the aggregator $P_A$ (plus $P_S$) are corrupt. The client $P_C$ and the servers in $H (|H| < t)$ are honest.

**Case-1: Corrupt $P_E$, $P_A$ and servers in $C$ with $|C| \leq t$.** 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 RevEave phase even if it is provided with $(sk, x)$ 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 $\bot$ to the request (aggregate, $sk, x, \ldots$).

- 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 RevEave phase:

- public key $ok = (pk = g_2^{sk}, (pk_i = g_1^{sk_i})_{f < i \leq n})$;
- corrupt secret-keys $\{sk_i\}_{i \in [f]}$;
- input $x$, and subsequently $H_1(x)$ through RO query;
- blinded input $\tilde{y} = H_1(x)^{\rho}$ and the NIZK proof of knowledge of exponent $\rho$;
- blinded partial outputs $\tilde{y}_i = (\tilde{w}_i = \tilde{y}^{sk_i}, \tilde{\pi}_i)$ for $i \in \{f + 1, \ldots, n\}$, where $\tilde{\pi}_i$ is a NIZK proof of equal exponent with respect to $sk_i$.

- blinded output $\tilde{y} = H_1(x)^{\rho}\cdot g_k$.

Again, we need to ensure that the probability that the adversary can ask an RO query to $H_2(\cdot)$ on $\pi = H_1(sk)$ is negligible. We reduce this again to Co-CDH akin to the centralized case. Given a Co-CDH instance:

$$g_1, g_1^{sk}, h_1 \in G_1; g_2, g_2^{sk} \in G_2$$

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 $i \in [t]: \omega_i := x^{sk_i}$. For $i \in \{t + 1, \ldots, n\}$ use the Lagrange interpolation in the exponent to construct $x_{sk_i} := g_1^{sk_i}$ where implicitly using $sk$ at point $0$ and for each $i \in [t]: sk_i$ as the $i$-th polynomial output. Set $\omega := (pk, [\omega_i]_{i \in [n]}$) as the verification key.

- Program the RO query $H_1(x) := h_1$.

- Choose uniform random $r \leftarrow \mathbb{Z}_p$, and compute $\tilde{\omega}_1 = g_1^{-r}$ and $\tilde{\omega}_sk := g_1^{sk}$. Define $\tilde{\psi} := \tilde{\omega}$. Then for $i \in [n]$ define for each $i \in [n]: \tilde{\omega}_i := \omega_i^{sk_i}$. The associated NIZK proof $\tilde{\pi}_i$ is simulated using EqSimu if $sk_i$ is unknown, otherwise, it is computed correctly.

- Finally NIZK proof $\mu$ is simulated using the simulator KepSimu on the instance $(h_1, \tilde{\psi} = \tilde{\omega}_1)$.

Similar to the centralized case, we can argue that the above simulation is correct. Note that, since no partial evaluation $H_1(x)^{sk_i}$ 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 $|C| \leq t$.** 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 $(\tilde{\psi}, \mu)$; (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 $q = \text{poly}(x)$ many complete (here it means for each $x$, 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 $(x_1, y_1, \pi_1), \ldots , (x_q, y_q, \pi_q)$. We will argue unless this is true, there
The reduction to BOMDH works as follows: Given a T-BOMDH complete query means at least querying for by making at most \[ |D| \text{-poly such that} \]

\[ s_k \text{-element} \]

is a PPT reduction that would break the so-called “unbiasability” would continue to hold.

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 \( s_k \). 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 \( \text{cnt} \) is incremented only when an evaluation query is completed, that is whenever the adversary has acquired sufficient information to produce one more valid triple, \( |F| > |\text{cnt}| \) 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.** \( P_A \) corrupt, \( |C| \geq t + 1 \). 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 called “unbiasability” would continue to hold.

The simulator, in this case, receives the verification key \( \psi \) from the adversary and registers it with the ideal functionality within \( \text{Keys}[S] \) 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 \( I, L \), where \( I \) contains pairs \( x, \psi \), that is the information with respect to the input and corresponding client’s message (generated by the simulator); and \( L \) contains tuples \((x, \pi, \beta, y)\), that is information from the entire evaluation, including server’s message and the output with respect to an input.
2. On receiving \((\text{Input}, \psi, x)\) sample a uniform random \( \rho \in \mathbb{Z}_p \) and then construct \((x, \bar{x})\) just like an honest party, where \( \bar{x} = (\psi := H_1(x)^\rho, \mu) \). Append \((x, \bar{x})\) to a list \( I \).
3. On receiving a message \( \bar{y} \) from the aggregator \( P_A \):
   a. If there is an \((x, \bar{y}) \in I\) such that \( e(\bar{y}, g_2) = e(\psi, g_{k_2}) \):
      i. If \((x, *, *, *) \notin L\) then issue an evaluation query (Eval, \( x, \bar{y} \)) to the ideal functionality, and when the ideal functionality returns the same query, reply with \((\pi := \bar{y}^1/\rho, \beta := \bar{y})\). Finally, on receiving an output \( y \) from the ideal functionality store \((x, \pi, \beta, y)\) into \( L \).
   b. Otherwise, on receiving the evaluation query from the ideal functionality, reply with \( \bot \).
4. On receiving an RO query \( H_2(x) \):
   a. If there is an \((x, \psi) \in I\), such that \( e = (\pi, g_2) = e(H_1(x), \bar{y}) \) but \((x, *, *, *) \notin L\) then make a (Eval, \( \psi, x \)) 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 \( H_2(\pi) := y \). Append \((x, \pi, \beta, y)\) into \( L \).
   b. Otherwise just respond with \( H_2(\pi) := y \).

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.
Other queries are straightforward to handle in this way. We argue that the above simulation is correct with overwhelming probability. In particular, unless the adversarial server can guess $H_1(x)^{sk}$ before observing $x$, the simulation would be perfect. Note that, once the server obtains $x$, a pair $(x, \tilde{y})$ gets listed in $L$. And then there are two cases: (i) The adversary makes a RO query $H_2(\tilde{y})$, with a valid $\pi$ for which the verification equation holds, before it returns $\tilde{y}$ in this case, the simulator first executes Step 4a. Later when it receives $\tilde{y}$, 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 $\tilde{y}$ and later makes a RO query $H_2(\pi)$; the simulator now first executes Step 3(a)(i), and later Step 4c. In this case, since the pre-verification must satisfy the simulation, the verification 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 $\bot$ to the ideal functionality. However, if the adversary can correctly predict the output of $H_1(x)$ without explicitly making RO query $H_1(x)$, 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 $t, f, n, \ell, l, l', N \in \mathbb{N}$ (where $f \leq t < n$) and $\mathbf{q} = (q_1, \ldots, q_\ell) \in \mathbb{N}^\ell$, define $\text{Max}_{t, f}(\mathbf{q})$ to be the largest value of $t$ for which there exists binary vectors $u_1, \ldots, u_\ell \in \{0, 1\}^\ell$ such that each $u_i$ has hamming weight $\geq t - f$ and $\mathbf{q}$ satisfies $\mathbf{q} \geq \sum_{i=1}^\ell u_i$. Next, we define the T-BOMDH - Threshold-Bilinear One-more Diffie Hellman assumption.

**Definition 4.** $(f, t, n, \ell, l) \text{- T-BOMDH.}$ Consider polynomial (in $k$) size integers $t, f, n$ such that $f \leq t < n$ and consider bilinear pairing groups $G_1 \times G_2 \rightarrow G_T$ where each group has prime order $p$. Let $g_1, g_2$ be two random generators of the groups $G_1, G_2$ 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 $\leq \text{neg}(\kappa)$.

- Sample uniform random secret $\alpha \leftarrow \mathbb{Z}_p$.
- Sample random group elements $\tilde{g}_1, \ldots, \tilde{g}_N \in G_1$.
- Provide $g_1, g_1^{\alpha^r}, g_2, g_2^{\alpha^r}, (\tilde{g}_1, \ldots, \tilde{g}_N)$ 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 $\mathbf{q} := \alpha^r$.
- Give the following oracle access $O(i, x)$ to the adversary:
  - $O(i, x \in \mathbb{G})$
    - Increment $\tilde{g}_i$ by 1.
  - Output $x^{\tilde{g}_i}, \pi$ where $\pi := D(i)$.
  - On receiving $((\tilde{g}_i, \tilde{h}_i))_{i \in [t]}$ from $\mathcal{A}$, return 1 if and only if all of the following conditions are met:
    - All $\tilde{g}_i$ are distinct and $t > \text{Max}(\tilde{g})$.
    - For all $i \in [t]: \tilde{g}_i \in \langle \tilde{g}_1, \ldots, \tilde{g}_N \rangle$ and $\tilde{h}_i = \tilde{g}_i^{\alpha^r}$.

**Theorem 5.** $(f, t, n, \ell, l) \text{- T-BOMDH}$

**Proof.** Given an $(f, t, n, \ell, l) \text{- T-BOMDH}$ adversary $\mathcal{A}$, let us construct $\mathcal{B}$, a $(f, t, n, l, l') \text{- T-BOMDH}$ adversary using $\mathcal{A}$.

Let $\alpha \leftarrow \mathbb{Z}_p$ and $g_1, g_1^{\alpha^r}, g_2, g_2^{\alpha^r}, (\tilde{g}_1, \tilde{g}_2, \ldots, \tilde{g}_N) \leftarrow \mathbb{G}_1$ be a challenge vector for $\mathcal{B}$. Now $\mathcal{B}$ chooses $N$ random vectors $(\beta_1, \beta_2, \ldots, \beta_N) \leftarrow \mathbb{Z}_p$ for all $1 \leq i \leq N$ (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 $\beta$ vectors as $M$). And defines $g'_1 := \Pi_{k=1}^N g_{1, \beta_k}$ and sends $g_1, g_1^{\alpha^r}, g_2, g_2^{\alpha^r}, (g_1^{\alpha^r}, g_2^{\alpha^r}, \ldots, g_N^{\alpha^r})$ as a challenge vector to $\mathcal{A}$.

\[
M \times \begin{pmatrix} \tilde{g}_1 \\ \vdots \\ \tilde{g}_N \end{pmatrix} = \begin{pmatrix} g'_1 \\ \vdots \\ g'_N \end{pmatrix} 
\]  

First $\mathcal{A}$ chooses a corrupt set of parties along with the corrupt values $F = \{(a_i, \alpha_i)\}$ 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 $O_D(\cdot)$ of $\mathcal{A}$ by making the same query to itself.

Finally, if $\mathcal{A}$ outputs some $l$-element $j \subseteq [N]$ and $\alpha_j = (g_j')^\alpha$ for all $j \in F$.

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

\[
U^{-1} \times \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_l \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_l \end{pmatrix}
\]

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

**Lemma 1.** [35] Let $t$ be any positive integer. Then there doesn’t exist $\mathbf{q} \in \mathbb{Z}_p^\ell$ s.t.

(1) $||\mathbf{q}||_1 \geq Qt$, and

(2) $||\mathbf{q}||_{\text{sup}} \leq Q$, where $Q = \text{Max}_{t, 0}(\mathbf{q}) + 1$.

**Proof.** Proof by induction on $Q$. If $Q = 1$ then $\text{Max}_{t, 0}(\mathbf{q}) = 0$, which implies that there are at most $t - 1$ non zero entries in $\mathbf{q}$. So if satisfies (2), then $\mathbf{q}$ 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 $\text{Max}_{t, 0}(\mathbf{q}) > Q$). Let $q'$ be $\mathbf{q}$ with largest $t$ entries decreased. Then this $q'$ satisfies both (1) and (2) as $||q'||_1 = Q$. 

Aniket Kate, Earwar Vivek Mangipudi, Siva Maradana, and Pratyay Mukherjee
Now by lemma 1, we have 

and let \( a_1^T, \ldots, a_Q^T \) be the rows of \( A, b_1, \ldots, b_Q \) be the columns of \( B \), and \( k = (k_1, \ldots, k_M) \). Then all these \( a_i \)'s , \( b_j \)'s and \( k \) are \( w \)-dimensional column vectors.

Let \( k^j = (k_1^j, \ldots, k_M^j)^T, k^j \otimes b_i = K^j b_i \) and \( V = \{ k^j \otimes b_i \} \) if \( i=0,1,\ldots, Q-1 \). Then the conditions \( AB = I \) and \( ABK = AK^2B = \cdots = AK^k B = O \) can be rewritten as

\[
\begin{pmatrix}
1 & b = b_1 \\
0 & b \in V \setminus \{ b_1 \}
\end{pmatrix}
\]

for all \( 1 \leq i \leq Q \). Therefore \( b_1 \) doesn't belong to linear span of \( V \setminus \{ b_1 \} \). Suppose that \( W = V \setminus \{ b_1, b_2, \ldots, b_Q \} \) is a linearly independent set. Then \( W \cup \{ b_1 \} \) is also linearly independent set, similarly \( W \cup \{ b_2, b_3 \} \) is a linearly independent set as \( b_3 \) is independent of \( W \cup \{ b_1, b_2 \} \). And finally \( V \) is a linearly independent set. Contradicts lemma 2. So \( W = \{ k^j \otimes b_i \} \) is a linearly dependent set. Then there exists \( x_{ij} \) for all \( 1 \leq j < t, 1 \leq i \leq Q \) at least one of them is non zero s.t

\[
\sum_{j=1}^{Q} x_{ij} k^j \otimes b_i = 0.
\]

Since non of the \( k \)'s entries are zeroes,

\[
\sum_{j=1}^{Q} x_{ij} k^j \otimes b_i = 0
\]

and let \( a_1^T, \ldots, a_Q^T \) be the rows of \( A, b_1, \ldots, b_Q \) be the columns of \( B \), and \( k = (k_1, \ldots, k_M) \). Then all these \( a_i \)'s , \( b_j \)'s and \( k \) are \( w \)-dimensional column vectors.

Let \( k^j = (k_1^j, \ldots, k_M^j)^T, k^j \otimes b_i = K^j b_i \) and \( V = \{ k^j \otimes b_i \} \) if \( i=0,1,\ldots, Q-1 \). Then the conditions \( AB = I \) and \( ABK = AK^2B = \cdots = AK^k B = O \) can be rewritten as

\[
\begin{pmatrix}
1 & b = b_1 \\
0 & b \in V \setminus \{ b_1 \}
\end{pmatrix}
\]

for all \( 1 \leq i \leq Q \). Therefore \( b_1 \) doesn't belong to linear span of \( V \setminus \{ b_1 \} \). Suppose that \( W = V \setminus \{ b_1, b_2, \ldots, b_Q \} \) is a linearly independent set. Then \( W \cup \{ b_1 \} \) is also linearly independent set, similarly \( W \cup \{ b_2, b_3 \} \) is a linearly independent set as \( b_3 \) is independent of \( W \cup \{ b_1, b_2 \} \). And finally \( V \) is a linearly independent set. Contradicts lemma 2. So \( W = \{ k^j \otimes b_i \} \) is a linearly dependent set.

\[
\sum_{j=1}^{Q} x_{ij} k^j \otimes b_i = 0.
\]

Since \( b_i \)'s are linearly independent of \( W \), so \( x_{ij} = 0 \) for all \( 1 \leq i \leq Q \). So

\[
\sum_{j=1}^{Q} x_{ij} b_i + \sum_{j=1}^{Q} x_{ij+1} k^j \otimes b_i = 0.
\]

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

**THEOREM 6 (GENERIC GROUP HARDNESS OF \( (0, T, N, L, 1)^{T-\mathrm{BOMDH}} \)).** Let \( G_1 \times G_2 \to G_T \) be generic bilinear pairing map, where each group of primes 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 \( G_1, G_2 \) and \( G_T \) respectively, where \( \xi^{(1)}(\cdot), \xi^{(2)}(\cdot) \) and \( \xi^{(T)}(\cdot) \) 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 \( (\xi^{(1)}(a), \xi^{(2)}(b)) \), outputs \( \xi^{(1)}(a+b) \) for all \( i \in \{1,2,T\} \).
- Bilinear pairing oracle, which on input \( (\xi^{(1)}(a), \xi^{(2)}(b)) \), outputs \( \xi^{(T)}(ab) \).
- \( T-\mathrm{BOMDH} \) oracle, which on input \( (i, \xi^{(1)}(a)) \), outputs \( O_D(i, \xi^{(1)}(a)) = \xi^{(1)}(D(i)a) \).
If $\text{Adv}^{\mathcal{O}_d(\ldots)}_{\mathcal{A}}(t, n, l, r, s, p)$ is the probability that $\mathcal{A}(\xi^{(1)}(1), \xi^{(1)}(D(0)), \xi^{(2)}(1), \xi^{(2)}(D(0)), \delta^{(1)}(u_1), \ldots, \delta^{(1)}(u_l))$ outputs $$(\xi^{(1)}(D(0)u_1), \ldots, \xi^{(1)}(D(u_l)))$$ after making $r$ group operation queries to all three groups combined, e bilinear pairing queries and $q_1$ queries to $O_D(i, \ldots, w) = 1$, s.t. $\max_{\mathcal{X}_T,q}(q) < l$, then

$$\text{Adv}^{\mathcal{O}_d(\ldots)}_{\mathcal{A}}(t, n, l, r, s, p) < \left( \frac{\epsilon \omega + 2}{\epsilon} + \left( \frac{\epsilon \omega + 2}{\epsilon} \right)^2 \right) \epsilon \omega$$

Proof. Let's construct a $\mathcal{B}$ which simulates the real challenger while interacting with $\mathcal{A}$. It maintains a list $\mathfrak{L} := \{F_x, x^{(1)}(\ldots)\}_{i=1,\ldots,\sigma}$, where $F_x(U_1, \ldots, U_n, A_0, A_1, \ldots, A_I)$ is a polynomial of degree at most $e$, and $x^{(1)}(\ldots)$'s are random elements in $\mathbb{G}_1$. Initially $\mathcal{B}$ sets $\sigma = l + 2$ and initializes list $\mathfrak{L}$ by setting $F_1 = F_2 = D(0) = a_0, F_3 = u_1, \ldots, F_{t_n + 2} = u_l$, and picks

$$x^{(1)}(1, \ldots, x^{(1)}(D(0)), x^{(2)}(1, \ldots, x^{(2)}(D(0)), \delta^{(1)}(u_1), \ldots, \delta^{(1)}(u_l))$$

as random elements in respective groups corresponding to upper indices and $a_0, \ldots, a_l \in \mathbb{G}_1$. $\mathcal{B}$ sends

$$x^{(1)}(1, \ldots, x^{(1)}(D(0)), x^{(2)}(1, \ldots, x^{(2)}(D(0)), \delta^{(1)}(u_1), \ldots, \delta^{(1)}(u_l))$$

to $\mathcal{A}$ as

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 $F_{s_1+s_2} = F_{s_1} + F_{s_2}$, if there exists $t \leq \sigma$ such that $(F_t, x^{(1)}(\ldots))$ is in $\mathfrak{L}$, then $\mathcal{B}$ outputs $x^{(1)}(\ldots)$ to $\mathcal{A}$. Otherwise $\mathcal{B}$ picks random group element $x^{(1)}(\ldots)$ from $\mathcal{G}_1$ which is different from the previously chosen ones and sends it to $\mathcal{A}$ and sets $\sigma + t, e + 2$.

- Bilinear pairing operation query: $\mathcal{A}$ inputs $(1, s_1)$ and $(2, s_2)$. Then $\mathcal{B}$ computes $F_{s_1+s_2} = F_{s_1}F_{s_2}$. If there exists $t \leq \sigma$ such that $(F_t, x^{(1)}(\ldots))$ is in $\mathfrak{L}$, then $\mathcal{B}$ outputs $x^{(1)}(\ldots)$ to $\mathcal{A}$. Otherwise $\mathcal{B}$ picks random group element $x^{(1)}(\ldots)$ from $\mathcal{G}_1$ which is different from the previously chosen ones and sends it to $\mathcal{A}$ and sets $\sigma + t, e + 2$.

- $O_D(\ldots)$ oracle query: $\mathcal{A}$ inputs $k \in [n]$ and $s \in [\sigma]$. Then $\mathcal{B}$ computes $F_{s+1} = (L_{k}^{T}a)F_{s}$. If there exists $t \leq \sigma$ such that $(F_t, x^{(1)}(\ldots))$ is in $\mathfrak{L}$, then $\mathcal{B}$ outputs $x^{(1)}(\ldots)$ to $\mathcal{A}$. Otherwise $\mathcal{B}$ picks random group element $x^{(1)}(\ldots)$ from $\mathfrak{G}_1$ which is different from the previously chosen ones and sends it to $\mathcal{A}$ and sets $\sigma + t, e + 2$.

Where $L_{k}^{T} = [1, k, \ldots, k^\ell]^{T}$ and $a = [a_0, \ldots, a_l]^{T}$.

$\mathcal{A}$ finally outputs $(F_{t_n}, \ldots, F_{t_n})$, and it wins if $F_{t_n} = u_1D(0)$ for all $i \in [l]$.

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

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

$$F_s = \sum_{i=1}^{w} a_i^{(x)} x^{(s)} + \sum_{i=0}^{r} y^{(s)}(i) u_i,$$

where

$$v_i = \sum_{Z \subseteq \{1, \ldots, l\} \in Z} \left[ \sum_{j=0}^{r_{\mathcal{A}}} \beta_j^{(i)}(u_j) \prod_{i \in Z} (a_{k_{ij}}^{(T)}) \right]$$

where $\alpha_i$, $y_i$, and $\beta_j^{(i)}$’s and $\beta_j^{(i)}$’s are all field elements specified by $\mathcal{A}$. (And suppose that in the $i$th $O_D(\ldots)$ oracle query, $\mathcal{A}$’s second is $k_1$; then $L_{k_1}^{T}$ must appear, so $i \in Z$ holds in the expression of $v_i$.)

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

Now consider the case where $\deg(F_{t_n}) \leq 1$ 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.,

$$v_i = \sum_{j=0}^{r_{\mathcal{A}}} \beta_j^{(i)}(u_j)(L_{k_1}^{T}a).$$

Then

$$F_s = \sum_{i=1}^{w} a_i^{(x)} x^{(s)} + \sum_{i=0}^{r} y^{(s)}(i) u_i.$$

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

$$\begin{align*}
F_{s_1} &= \begin{pmatrix} 1 \\ \vdots \end{pmatrix} + Cu + \begin{pmatrix} 1 \\ \vdots \end{pmatrix} (11) \\
F_{s_2} &= \begin{pmatrix} 1 \\ \vdots \end{pmatrix} \ (12)
\end{align*}$$

where

$$A = \begin{bmatrix} a_1^{(s_1)} & \cdots & a_w^{(s_1)} \\
\vdots & \ddots & \vdots \\
a_1^{(s_r)} & \cdots & a_w^{(s_r)} \end{bmatrix}, B = \begin{bmatrix} \beta_1^{(1)} & \cdots & \beta_1^{(1)} \\
\vdots & \ddots & \vdots \\
\beta_w^{(1)} & \cdots & \beta_w^{(1)} \end{bmatrix}$$

Let $b = Bu + b_0$. Now substitute eq. (12) into eq. (11), then we get

$$\begin{align*}
F_{s_1} &= A \begin{pmatrix} b_1^{(1)} \\ \vdots \end{pmatrix} + Cu + \begin{pmatrix} 1 \\ \vdots \end{pmatrix} (13)
\end{align*}$$
Note that
\[
\begin{pmatrix}
L_{k_1}^T a \\
\vdots \\
L_{k_w}^T a
\end{pmatrix} = \left[ \sum_{i=0}^t a_i k_i \right] = a_0 \begin{pmatrix} 1 \\ \vdots \end{pmatrix} + a_1 \begin{pmatrix} k_1 \\ \vdots \end{pmatrix} + \cdots + a_t \begin{pmatrix} k_t \\ \vdots \end{pmatrix}
\]
So
\[
b \odot \begin{pmatrix}
L_{k_1}^T a \\
\vdots \\
L_{k_w}^T a
\end{pmatrix} = a_0 K^0 b + \cdots + a_t K^t b,
\]
where
\[
K = \begin{pmatrix}
k_1 \\
\vdots \\
k_w
\end{pmatrix}
\]
so now substituting eq. (14) in eq. (13) we get
\[
\begin{pmatrix}
F_{s_1} \\
\vdots \\
F_{s_{1-w}}
\end{pmatrix} = A (a_0 K^0 b + \cdots + a_t K^t b) + C u + \left[ \begin{pmatrix} (s_1) \\ \vdots \end{pmatrix} \right] \begin{pmatrix} Y_{0} \\ \vdots \end{pmatrix}
\]
\[
= a_0 A K^0 b + \cdots + a_t A K^t b + C u + \left[ \begin{pmatrix} (s_1) \\ \vdots \end{pmatrix} \right] \begin{pmatrix} Y_{0} \\ \vdots \end{pmatrix}
\]
substituting back \( b = B u + b_0 \) in eq. (15) we get
\[
\begin{pmatrix}
F_{s_1} \\
\vdots \\
F_{s_{1-w}}
\end{pmatrix} = a_0 A K^0 B u + a_0 A K^0 b_0 + \cdots + a_0 A K^t B u + a_t A K^t b_0 + C u + \left[ \begin{pmatrix} (s_1) \\ \vdots \end{pmatrix} \right] \begin{pmatrix} Y_{0} \\ \vdots \end{pmatrix}
\]
For a fixed random \( u \), the right side of the above equation is a linear function of \( a \). If \( A \) wins, then
\[
\begin{pmatrix}
F_{s_1} \\
\vdots \\
F_{s_{1-w}}
\end{pmatrix} = a_0 u,
\]
then upon comparing the last two equations, we get
\[
(AB - I) u + A b_0 = A K B u + A K b_0 = \cdots = A K^t B u + A K^t b_0 = 0 \tag{16}
\]
For fixed random \( u \) the above equations evaluate to 0 with probability \( \frac{1}{p} \). The other case where the above equations become zero is when \( AB - I = A K B = \cdots = A K^t B = 0 \). But according to lemma 3, at least one of \( AB - I, A K B, \ldots, A K^t B \) isn’t zero. Then that particular matrix equation evaluates to zero with probability for a fixed \( u \) is again \( \frac{1}{p} \).

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

So far the above computations are for a fixed random \( u \), now if we fix both \( u \) and \( a \), then we have to account for two distinct polynomials evaluating to the same value for random \( u \) and \( a \). This event happens with probability \( \frac{(\sigma)}{p} \), where \( \sigma \leq l + e + r + w \). Thus
\[
Adv_{A}^{O_2}(\cdot) (t, n, l, r, e, q, p) \leq \frac{(ewl)^2}{p} + \frac{(l + e + r + w)^2 ew}{2p}
\]
\( \square \)

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