InstaRand: Instantly Available and Instantly Verifiable On-chain Randomness

Jacob Gorman[1], Lucjan Hanzlik[2], Aniket Kate[3], Pratyay Mukherjee[1], Pratik Sarkar[1], and Sri AravindaKrishnan Thyagarajan[4]

[1] Supra Research
j.gorman@supra.com, p.mukherjee@supra.com, p.sarkar@supra.com
[2] CISPA Helmholtz Center for Information Security
hanzlik@cispa.de
[3] Supra Research/Purdue University
a.kate@supra.com
[4] University of Sydney
t.srikrishnan@gmail.com

Abstract. Web3 applications, such as on-chain gaming, require unbiased and publicly verifiable randomness that can be obtained quickly and cost-effectively whenever needed. Existing services, such as those based on Verifiable Random Functions (VRF), incur network delays and high fees due to their highly interactive nature. FlexiRand [CCS 2023] addressed these problems by hiding the output of the VRF and using that as a seed to derive many randomnesses locally. These randomnesses are instantly available for usage. However, these randomnesses can not be verified independently (or instantly) without disclosing the seed, leaving scope for malicious actors to cheat.

To solve this problem, we introduce a new notion, called instantly-verifiable VRF (iVRF), which enables the generation of many randomnesses from one VRF output seed, such that each of them is verifiable independently – this enables the *first* solution to *cost-effectively* generate randomnesses, such that they are *instantly available* and also *independently/instantly verifiable*.

To instantiate we propose a generic construction called InstaRand – it combines any (possibly distributed) VRF at the server's end with another VRF at the client's end to construct an iVRF. Our specific instantiation uses the BLS-based GLOW-DVRF [Euro S&P 2021] at the server's end and the DDH-based VRF of Goldberg et al. [RFC 2023] at the client's end. We use the universal composability framework to analyze the security. Moreover, due to its generality, InstaRand can be instantiated with any post-quantum secure VRF to yield a post-quantum secure iVRF.

Our experiments demonstrate that our instantiation of InstaRand is *highly practical*. The client incurs a *one-time* cost to generate the seed (server's VRF output) by querying the GLOW-DVRF servers once. Once the seed is set up, the client locally generates the pseudorandom value on demand in 0.18 *ms*, avoiding the client-server round-trip delay. Each value can be independently verified in 0.22 *ms*. This yields a 400× improvement in terms of output generation and 20× improvement in verification cost over existing solutions.

1 Introduction

Randomness plays a critical role in all forms of computing, serving various purposes such as generating cryptographic keys, enabling fair gameplay in online gaming, and minting NFTs on blockchains. With the increasing popularity of blockchains and Web3 applications like decentralized finance (DeFi) and gaming (GameFi [Wor]), the need for reliable sources of randomness has grown significantly[5]. Consider a multiplayer game where players take turns to roll a dice. If such a game is played over Web3, a cost-effective source of on-(block)chain randomness is essential. In fact, to ensure the confidence of the current and future participants, the randomness used must be (publicly) verifiable. As a result, there are several on-chain verifiable randomness services [GHK+24], such as Chainlink [Chaa], DRand [DRaa], Band [Ban], Supra dVRF [Sup], Dfinity [HMW18], etc., flourishing in the Web3 space. However, existing on-chain solutions are expensive and incur delays [Chac] during delivery. In particular, to obtain every randomness, each client must place a VRF request, which is then fulfilled by the VRF service interactively before it is available. So, not only does the price quickly escalate with a growing number of requirements[6], but each fulfillment also takes a while before it is available. For example, it would take 24 seconds [ych] on the Ethereum blockchain (to request and then fulfill via two transactions).

Recently FlexiRand [KMMM23] proposed a solution to mitigate these issues by introducing the notion of output-private VRFs. Their main idea was to use the output of a single VRF request multiple times, in that a requester (the entity running the gaming platform) obtains a single VRF output y and then locally generates several random values $z_1, z_2, \ldots$ where each $z_i = \mathsf{PRG}(y, i)$ for some appropriate pseudorandom generator PRG (can be a hash function). However, since in the blockchain setting, y becomes publicly available on the chain during fulfillment, that makes all z_i immediately predictable. So, y needs to be kept private with respect to everyone but the requester, who would compute the z_i values. For this purpose an output-private VRF was used: the requester, once a request is fulfilled, retrieves a blinded y from the VRF service via a blockchain, locally unblinds the same (as the blinding mask is exclusively known to the requester), and then uses the unblinded y to generate z_i locally. So, the amortized cost for the entire session decreases drastically – a single request suffices to generate all random values needed.

However, the necessity of keeping y private poses a new impediment to maintaining public verifiability. More specifically, the values $z_1, z_2 \ldots$ can not be verified independently/instantaneously now, because to verify each z_i one needs to first verify that y is indeed the correct VRF output, and each $z_i = \mathsf{PRG}(y, i)$. This requires the knowledge of y, unless one uses a zero-knowledge (ZK) proof [GWC19, YSWW21] which would make the procedure impractical due to

[5] Chainlink VRF [Chaa] has fulfilled 10.5 million [Chab] randomness requests in 2022.

[6] An average two-player Backgammon game takes 53.78 rolls [Esc24] costing $52,38,000$ gwei gas or $17.

non-blackbox use of the underlying PRG[7].On the other hand, the knowledge of y during verification of a single z_i lets everyone compute all other z_j $(i \neq j)$ values. In other words, there is no practical mechanism to allow verification of a particular z_i without breaking unpredictability for other z_j values. An optimistic approach would be that one waits until the end when all such z_i values are used up and then, one can verify y. This may leave scope for a malicious requester to supply malformed and biased z_i's *during* the game, which can not be caught until the very end when it could already be too late.

This conundrum begets the following question:

Can we design a practical cost-effective on-chain randomness service, whose output is immediately/instantly available and instantly verifiable?

In this work, we affirmatively resolve this question. We propose a new cryptographic primitive called *instantly-verifiable VRF (iVRF)* which enables anyone to verify each z_i individually/instantaneously without affecting the unpredictability of other z_j $(i \neq j)$ values while maintaining essentially the same amortized cost and instant availability of FlexiRand.

Instantly-Verifiable VRF (iVRF). In particular, iVRF is a special two-party protocol executed between a client (or a requester) and a server (or VRF node) where parties communicate over a bulletin board (a blockchain). It allows a client to make a single VRF request x to the server, and then reuse the server's response y to generate multiple pseudorandom outputs z_i (for session i) without further interaction with the server. The requirements for z_i values are that they are (i) pseudorandom even given all other z_j values, (ii) unbiasable, (iii) instantly verifiable with respect to y, server VRF verification key and client input x, and (iv) immediately available via local computation. We formalize via ideal functionality $\mathcal{F}_{i\mathsf{VRF}}$ (Fig. 3) in the Universal Composability (UC) framework [Can01].

InstaRand. Next, we propose a simple construction, named InstaRand, which generically takes any (possibly distributed) VRF [Chaa, Ban, Sup, KMMM23], combines with a VRF at the client side to construct a iVRF. The detailed workflow is as follows: In the setup phase, the client and the servers post their respective keys, vk_c and vk_s respectively and each of them holds the corresponding secret key sk_c and sk_s. In the pre-processing phase, the client posts a request, along with an input x that embeds the client's verification key vk_c. The input is then retrieved by the VRF server which produces an output y (plus a proof π) using sk_s. At this point the triple (x, y, π) can be verified publicly using the server's public key vk_s. In the online phase, once the VRF output y is retrieved by the client, who locally generates the z_i values using sk_c on y as the *seed* and a counter i. Each z_i can be independently verified by verifying with respect to vk_c and vk_s. Note that, the pre-processing phase is the only phase where server interaction is required, and that interaction is used with amortization in subsequent online

[7] We implement the PRG using Poseidon hash and generate a Plonk proof for each z_i. The proof generation is $400\times$ slower and verification is $20\times$ more expensive compared to InstaRand's proof generation and verification.

phases. We compare InstaRand, FlexiRand, and other existing VRF services qualitatively in Table. 1. Our contributions are summarized next.

1.1 Our Contributions

- We introduce instantly-verifiable VRF (iVRF), a new primitive that enables a client to interact with a VRF server *once* to produce multiple pseudorandom outputs z_i, that are independently (and instantly) verifiable[8] without affecting unpredictability of other z_js, for $j \neq i$. We formalize it via an ideal functionality $\mathcal{F}_{i\mathsf{VRF}}$ within UC framework [Can01].
- We propose InstaRand - a new generic construction for iVRF. We build it by using any VRF protocol in the random oracle model. It can be extended to the common reference string (crs) model as well. Prior designs, such as FlexiRand [KMMM23] do not support independent instantaneous verification, thereby falling short of iVRF requirement. Consequently, only InstaRand can be useful for many important applications, such as Web3 games that require many (pseudo-) random values, that are instantly verifiable. We formally show that our construction securely realizes $\mathcal{F}_{i\mathsf{VRF}}$.
- We extend iVRF to the distributed server setting to avoid a single point of failure on the server side. We formalize the distributed iVRF functionality via $\mathcal{F}_{i\text{-}\mathsf{DVRF}}$. Then we propose a distributed version of InstaRand that implements $\mathcal{F}_{i\text{-}\mathsf{DVRF}}$. This is done by simply distributing the server's secret key among multiple servers, which then act as the VRF committee. In particular, a t-out-of-n access structure would ensure resilience against up to t malicious corruptions. Also, guaranteed output delivery (aka robustness) can be ensured by setting $n \geq 2t+1$ among the committee of server nodes.
- Finally, we provide concrete instantiations and benchmark the performance. In particular, our client's VRF is instantiated with the DDH-based construction of Goldberg et al. [GRPV23], and the server's side VRF with BLS-based (distributed) construction by Galindo et al. [GLOW21] (abbreviated as GLOW-DVRF) – the choice was made considering the adaptability to the distributed setting, which is needed only at the server's end. Our benchmarking demonstrates that InstaRand protocol is highly practical – it takes 5.91 ms to generate the (one-time) server output (seed) y, the same as GLOW-DVRF in the pre-processing. In comparison, FlexiRand servers required 6.32 ms[9]. However, in an application setting requiring (i) cost amortization, (ii) instant availability and (iii) instant verifiability, to generate $N = 10$ pseudorandom z_i, FlexiRand's client takes about 33.2 ms[10] whereas

[8] We remark that we use the terms "independently" verifiable and "instantly" verifiable interchangeably. iVRF guarantees that the output is independently verifiable, whereas in the application, such as gaming, the requirement is instantly verifiable output.

[9] This increase in cost comes from the requirement of verifying proof of knowledge of the exponent with respect to the blinded input.

[10] This blow up in the cost of FlexiRand is due to the instant verifiability requirement, which requires one interaction with server per randomness – so no amortization is possible.

InstaRand's client takes only 4.6 ms – this marks more than 7× improvement, and scales with N (cf. Table 2).

In addition, we remark on some additional benefits.

Post-Quantum Security. InstaRand is a generic construction that uses *any* VRF as a black-box, and therefore can be post-quantum secure if the server-side and client-side VRFs are post-quantum secure. There are post-quantum secure VRFs from LWE [Mal24], isogenies [Lai24] and hash functions [EEK+23]. Using them on the server and client side suffices for us in the single-server setting. In the distributed setting, it is possible to thresholdize the server side VRF using a fully-homomorphic encryption (FHE) based universal thresholdizer [BGG+18, EY24], which would yield an iVRF protocol in the distributed setting.[11]

Single Transaction Randomness Service. To request randomness on the blockchain and use it in Web3 applications, the VRF client has to interact with the VRF servers via transactions on the blockchain. Each blockchain has its block latency to confirm a transaction via a block. For example, Ethereum blockchain takes 12 seconds to confirm a block, whereas Bitcoin blockchain takes 8 mins. So, the latency of any blockchain-assisted protocol is dominated by the transaction latency, which is determined by the number of required on-chain transactions. The recent work of [GHK+24] showed that a verifiable randomness service on the blockchain necessarily requires two transactions for each randomness request–this holds for existing randomness services like [Chaa, Ban, Sup, HMW18]. Aptos Roll [Lab] generates randomness using a single transaction by using specific properties of blockchains (discussed in Sec. 1.2), but it cannot be extended to general blockchains, like Ethereum. InstaRand circumvents this lower bound by having a preprocessing phase which requires two transactions, whereas each extended output requires only a single transaction. Therefore, we require $2 + N$ transactions for N outputs, reducing the amortized number of transactions.

Compatibility with Beacon Service. Due to its generality, the InstaRand protocol is also compatible with beacon services like (centralized) NIST beacon [PBB+], verifiable beacons like Algorand [Ken], DRand [DRaa] etc. instead of VRF services. It just requires replacing the server-side VRF output with an appropriate beacon output. The client-side protocol generates instantly verifiable outputs on the beacon output instead of the VRF output. However, deploying any beacon-based randomness services introduces additional challenges [ST22] as (i) the blockchain has reliable block time, (ii) beacon service is also timed, (iii) and that clocks for beacon nodes and the blockchain validators are perfectly synchronized. Our current iVRF formalization does not consider time. We leave this as an interesting future work.

[11] We note that the protocol using universal thresholdizer may not be practically efficient due to use of FHE. We leave constructing a practically efficient post-quantum secure distributed iVRF as an interesting future direction.

1.2 Related Works

In this section, we compare InstaRand with the relevant works from the literature and we present it in Table 1.

Protocols	Cost Amortization	Instant Availability	Instant Verifiability	Assumptions	Post-Quantum Security	# of Transactions for N outputs
VRF Services [Chaa, Ban, Sup, HMW18]	×	×	×	VRF	✓	$2N$
FlexiRand [KMMM23]	✓	✓	×	BOMDH	×	$4N$
FlexiRand [KMMM23] +ZK-SNARK [GWC19]	✓	✓	✓	BOMDH	×	$4 + N$
InstaRand (this work)	✓	✓	✓	VRF	✓	$2 + N$
Note: BOMDH: Bilinear Threshold One-More Diffie Hellman						

Table 1: Comparison of existing protocols.

(Distributed) VRFs. There are existing VRF-based randomness service protocols like Chainlink [Chaa], Band-VRF [Ban], Supra DVRF [Sup] and more general Verifiable Randomness as a service [GHK+24]. When a client makes a randomness request via a smart contract to the VRF service, the service evaluates a VRF on the input and returns the output and the proof via a smart contract to the client. However, this approach suffers from two limitations. Firstly, there is a significant latency overhead to complete a VRF request - the client has to make the request, wait for it to get confirmed on the blockchain, then the VRF provider evaluates it and sends it to the smart contract, and finally the smart contract verifies the proof and uploads the output – so, in essence, the randomnesses are not *instantly available*. Secondly, the output cannot be used as a seed to generate multiple randomnesses; since the output is publicly available, there is no unpredictability anymore. As a result, the client has to make a new randomness request which incurs the cost of blockchain plus server latency and at least *two* blockchain transactions – one for request, another for fulfillment – every time it needs a fresh random value. In comparison, InstaRand client avoids the server latency via pre-processing the server computation and reusing the server output non-interactively in the online phase to generate the randomness – if the randomness is used on blockchain, only one transaction is necessary then per randomness.[12]

FlexiRand. FlexiRand [KMMM23] introduced the idea of output-private VRF, in that only the client obtains the output y. A blinded version of the output is made public, which only the client can unblind. Anyone can verify that the blinded output was correctly generated, and once the final VRF output is unblinded it can be verified in the usual manner. The flow is similar to InstaRand: the client can

[12] We note that, the randomness generated can be used and verified off-chain, or even outside blockchain environment, such as Web2 gaming – this requires no transaction beyond the pre-processing.

request randomness in a preprocessing phase and by doing so the client shifts the latency overhead and the gas cost on the smart contract to the preprocessing phase such that the online phase is fast (non-interactive too) and inexpensive – this makes the VRF output *instantly available*. However, as stated in the beginning of the introduction section, the unblinded output y cannot be utilized to generate multiple pseudo-random values z_i's that are independently and publicly verifiable. This is because all randomnesses derives from y get leaked once the unblinded output is made public. This limits the application of FlexiRand to scenarios where verifiability can be postponed until all z_i's have been utilized. Hence, if instant verifiability is needed, a client, using FlexiRand, has to pay (in the preprocessing phase) the VRF service fee and incur the gas cost proportional to the total number of independent z_i values. This would totally obliterate the benefit of amortized cost and would restrict their potential usage in many Web3 gaming applications.

Adding ZK-SNARKs: One alternate approach is to prove in zero knowledge [GWC19] that z_i was correctly evaluated by running a PRG (implemented using a Poseidon hash) on the blinded y without exposing y. However, this requires 80ms to generate an SNARK proof of correct computation, and verification takes 4.5ms and at least 300k gas [Tha] on-chain. Meanwhile, InstaRand outputs are generated in 0.18ms and verified in 0.22ms costing 84k gas.

Moreover, it is unclear how to extend FlexiRand to the post-quantum setting since the *output-private* VRF requires a proof of correct computation of a committed VRF output that can be locally converted into a proof of correct computation of the revealed VRF output by the client. The current construction requires bilinear pairing and extending them to lattice or isogeny setting remains open. Meanwhile, InstaRand is a generic construction that uses any VRF as a blackbox, and therefore can be post-quantum secure if the server-side and client-side VRFs are post-quantum secure [Mal24, EEK+23].

Randomness Beacons. Another popular paradigm of obtaining randomness is to rely on randomness beacons [DRaa, Ken]. These beacons periodically post random values on a blockchain. To ensure that a randomness is only available after a certain time, often verifiable delay functions are used [Chi, CCB23, Vee, BDD+23, BBBF18, LW15]. Parties read those random values from the blockchain and use them in a Web3 game/application. However, the parties have to wait for the randomness beacon to output the randomness at each predefined epoch, rendering instant availability elusive. For example, the Drand beacon emits [Drab] a new random value every 30 seconds. This causes the Web3 game to stall until the beacon generates new randomness each time and it gets confirmed on the blockchain. Other protocols, such as SPURT [DKIR22], does generate randomness at sub-second levels. However, unlike InstaRand, the generated randomness cannot be verified efficiently against a *single* server public key, since the VSS-based beacon protocols do not run a DKG. Hence, to verify the SPURT (or Hydrand [SJSW20], Rondo [Men24]) output one has to verify the PVSS vectors which require $O(n)$ exponentiations or pairings. This would be expensive in terms of gas cost in many

Web3 applications where the generated randomness needs to be publicly verified via smart contracts.

Moreover, unless we deploy a technique similar to InstaRand (or FlexiRand), the beacon output cannot be used directly as a seed to generate multiple verifiable outputs. However, as mentioned earlier, InstaRand can be made compatible with a beacon service as well, instead of a VRF service.

Aptos Roll. The work of [Apt] is an interesting recent on-chain randomness approach that offers instant randomness to the Aptos' internal ecosystem. Here, the Aptos validators generate a beacon value corresponding to each transaction block. Whenever the Aptos validators include a client's randomness request to a block, the validators use the beacon, generated on the same block to fulfill the randomness instantly by hashing the beacon value with some client/smart contract data. Clearly, this approach crucially uses specific design aspects of Aptos' blockchain design. Consequently, it can work only on those chains where the block consensus and ordering need to occur first, followed by the beacon generation on the finalized block. Therefore, while this approach can work on blockchain with fixed nodes such as Aptos and Sui, it cannot be employed by many blockchains, such as Ethereum and its Layer-2 solutions. Conversely, InstaRand is in the same paradigm of existing designs, such as GLOW-DVRF, FlexiRand, etc., which uses blockchain in a blackbox manner, without relying on its design specifications. Therefore, unlike Aptos Roll, it can be deployed on *any* blockchain with smart-contract capability including Aptos.

Unbiasable VRFs. The recent work of [GS24] studied the scenario where the VRF evaluator can potentially sample a biased keypair (vk, sk), such that the VRF evaluation using secret key sk generates biased outputs. They formalize this with a new notion of unbiasable VRFs – a VRF is considered unbiasable if an adversarial evaluator cannot find a set of VRF keys whose evaluation on random inputs yields biased outputs. In InstaRand, a malicious client could potentially choose a pair of biased VRF keys in order to bias z_i's. However, our construction takes care of this by using random oracles without explicitly using unbiasable VRFs. Nevertheless, one may alternatively use unbiasable VRFs on the client side to achieve $i\mathsf{VRF}$ as well potentially yielding a construction without random oracles, albeit at the expense of efficiency.

Indexed VRF. The work of [EEK+23] introduced the notion of indexed VRF and demonstrated its usage in the algorand leader election. The VRF evaluator in [EEK+23] precomputes N random values together and commits to it using a Merkle tree and when required, the evaluator reveals each leaf and the path from the root as the proof. Their protocol requires securely storing the entire Merkle tree state and knowing an upper bound on N beforehand. Using this as a randomness service would require the VRF server to run the evaluation algorithm on each randomness request, and it would provide the same properties as a regular VRF service, with the added restriction that the VRF can be used to fulfill only N requests. Moreover, thresholdizing the server algorithm is non-trivial. Whereas, the InstaRand client generates randomness on demand without interacting with the VRF server and only needs to store the client-side VRF secret key.

Aggregatable VRF. There have been recent works on aggregatable VRFs [Mal24, DDKL24] where multiple VRF proofs for several VRF evaluations on different inputs under differents VRF keys are combined into a single constant sized proof. However, they do not support the properties of an iVRF since the client needs to query the VRF server for each randomness request.

2 Technical Overview

In this section, we provide an overview of the InstaRand protocol and discuss how to extend it to the distributed server setting. Finally, we conclude with a concrete application.

2.1 iVRF Primitive

The iVRF is a special two-party protocol, between a client (or a requester) and a server (or VRF node) where parties communicate over a bulletin board (that abstractly captures the blockchain setting). The server posts its verification key vk on the bulletin board. Then, the client queries the server with input x to obtain the server output y and proof π.[13] The triple (x, y, π) is (publicly) verifiable with respect to vk – we call this pre-verification. Next, the client performs local computation on (x, y), its secret state, and a session identifier i to generate randomness z_i for each session i. The output z_i values are (i) pseudorandom, (ii) unbiasable, (iii) independently verifiable with respect to y and (iv) remain unpredictable given all the other z_j values.

2.2 InstaRand Protocol

In this section, we build InstaRand and show that it is an iVRF.

First Attempt. Assume that the server has a pair of VRF keys - (vk_s, sk_s) that are posted on to the bulletin board. Let's consider a construction where the client queries the VRF server on input x. The server evaluates on x with sk_s to obtain output $(y, \pi) := \mathsf{VRF.Eval}(sk_s, x)$ where proof is π – pre-verification is immediate. This is sent to the client. To generate instantly verifiable outputs for session i, the client samples a VRF key pair - (vk_c, sk_c), and then evaluates VRF on (i, y) with secret key sk_c to obtain $(z_i, \rho_i) := \mathsf{VRF.Eval}(sk_c, (i, y))$ where ρ_i is the proof – the triple $(z_i, \rho_i, (i, y))$ can be verified with respect to vk_c.

Problem 1: The above solution breaks as a malicious client can bias the z_i values by sampling multiple vk_c values after obtaining y, computing the candidate z_i values and then choosing the particular vk_c which yields favorable z_i values, for example, choose a specific vk_c which makes the first bit of z_i to be 0.

[13] We refer to this interactive phase as pre-processing. We remark that this pre-processing is not a limitation, but a feature, which enables important properties such as instant availability and cost amortization. Among prior works, only FlexiRand supports such pre-processing due to output privacy.

Binding vk_c to x. To resolve we need to ensure that the input x is bound to use a *single* vk_c value. This is done by including vk_c in x; by setting $x := (u, vk_c)$ where u is the client's actual input and (vk_c, sk_c) is client's VRF key pair.

Problem 2: The above solution again falls short as a malicious client can choose a biased VRF [GS24] key pair (vk_c, sk_c), such that the evaluation of the client's VRF on (i, y) with secret key sk_c outputs non-pseudorandom values. For example, the client can choose a biased sk_c such that, regardless of input, all z_is would have the last bit fixed to 0 – such biasing issues are also considered in the recent work of [GS24]. They mitigate it using unbiasable VRFs, whereas we take a simpler and more efficient approach.

Tackling a Biased vk_c. We tackle the above issue by using a random oracle H jointly on the server's and client's VRF outputs.[14] In particular, if client's VRF computation on (i, y) using sk_c is expressed as $(w_i, \rho_i) := \mathsf{VRF.Eval}(sk_c, (i, y))$, then the output z_i for session-i is computed as $z_i := \mathsf{H}(i, w_i, y)$. This way, even if a malicious client biases its VRF output w_i, the final session output z_i remains pseudorandom due to the randomness of the server output y. Intuitively, this guarantees that as long as at least one among the client and server is honest, the corresponding VRF output is pseudorandom, which makes the final z_i pseudorandom. This yields an $i\mathsf{VRF}$ that is secure against a malicious client. However, this construction does not suffice for full simulation security.

Problem 3: To provide full simulation security, the simulator needs to extract the client's input (i, x) on which z_i is computed as the $i\mathsf{VRF}$ output by the client. However, the simulator cannot find out x from the output $z_i := \mathsf{H}(i, w_i, y)$ by observing the queries made to random oracle H since it does not contain x (and the proof of correct evaluation on x) as input to H.

Extraction using Random Oracle. Extraction of x is enabled simply by including input x [15] in the evaluation of $(w_i, \rho_i) := \mathsf{VRF.Eval}(sk_c, (x, y, i))$ plus the evaluation of $z_i := \mathsf{H}(i, w_i, x, y)$. We rely on an additional property of the client-side VRF that allows a simulator to check whether w_i is the correct evaluation of VRF on input (x, y, i) w.r.t. verification key vk_c. By using this property, the simulator checks that the queried input to H is indeed correct, and so it can program H on the correct (i, w_i, x, y, i) to return a random output (obtained from the ideal functionality). This is required to ensure that the final output is unbiasable even if the client is corrupt.[16] We also prove in Appendix. D that

[14] This achieves the same effect as an unbiasable VRF using the random oracle. Another approach could be to use unbiasable VRF generically, although that might incur more performance overhead. We note that in [GS24], the author's one of the main objectives was to design a construction without random oracles, which is orthogonal to ours.

[15] This is similar to the strategy of including the input within the oblivious PRF constructions of [JKX21] and similar primitives [AMMM18].

[16] We note that this problem could not been solved by including the proof ρ_i inside the oracle query since a malicious client can construct different ρ_i for the same w_i and choose the one for which it obtains a favorable z_i, hence breaking unbiasability.

the existing VRF protocols satisfy the above mentioned property in the random oracle model.

Final Protocol. We illustrate our final protocol in Fig. 1. Assume that the server's verification key vk_s is posted on the bulletin board. In the pre-processing phase, the client samples a key pair (vk_c, sk_c) for a client-side VRF protocol, making a VRF query to the server with input $x := (u, vk_c)$ where $u \in \{0,1\}^*$ is the client's input. The client receives the server-side VRF evaluation output y and proof π on input x. This is a one-time pre-processing cost. In the online phase, this y acts as an intermediate seed to generate multiple independently verifiable random values - $(z_1, z_2, \ldots, z_N)$, for N different sessions. For each session-i, the client generates an intermediate value $(w_i, \rho_i) = \mathsf{VRF}_c.\mathsf{Eval}(sk_c, (x, y, i))$ by locally evaluating VRF_c using its secret key sk_c. The final output for session i is set as $z_i = \mathsf{H}(i, w_i, x, y)$. Pre-verification is possible by verifying y using server verification key vk; final verification is done by verifying w_i using client verification key vk_c, and the hash computation. The output z_i is pseudorandom due to the pseudorandomness of y and w_i. And, $z_{j \neq i}$ values remain hidden from an external party unless the client reveals w_j.

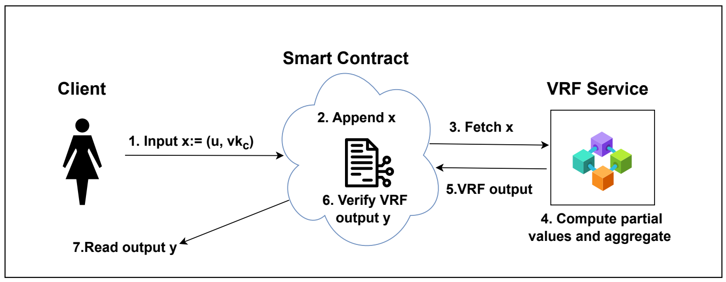
Extending to the crs model. The above protocol is proposed in the random oracle model to generate extended outputs on the order of microseconds. However, it can be extended to work in the crs model without relying on any random oracle. To do so, we would instantiate the client and server side algorithms with an unbiasable VRF [GS24] and the client gives a non-interactive proof (of knowledge) [CSW22] of correct computation of vk_c w.r.t sk_c in the preprocessing phase. The sk_c is extracted from the NIZK proof, and the client-side extended outputs z_i can be extracted by the simulator and the proof is simulated.

2.3 Instantiations

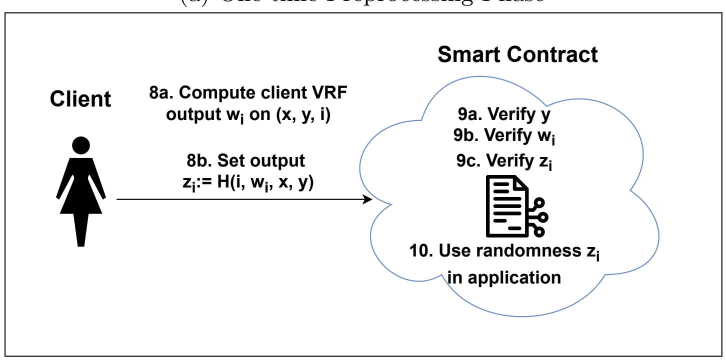
InstaRand requires two VRF protocols - one on the client side and the other on the server side. These can be instantiated from existing VRF constructions like GLOW-DVRF [GLOW21], RSA-based VRF [GRPV23] or DDH-based VRF [GRPV23]. Moreover, if the VRF is post-quantum secure, like the ones based on lattice [EKS+21] or isogeny [Lai23] assumptions, then the InstaRand protocol also guarantees post-quantum security. In our benchmarking, we implement the client VRF using the DDH-based VRF [GRPV23] and the server side VRF using GLOW-DVRF, the later chosen due to its adaptability to the distributed setting.

In the context of VRF, the server that possesses the secret key and calculates the VRF output y becomes a single point of failure for both secrecy/unpredictability and liveness. This means that the VRF outputs are entirely predictable to this node, and if this node crashes, the server-side VRF computation halts. To address this issue, instead of using a centralized VRF server, we use a distributed VRF (DVRF) [GLOW21] on the server side.

In a DVRF framework, unlike a centralized VRF, no single node has access to the entire secret key. Instead, the secret key is shared among multiple parties (referred to as VRF committee, denoted as $S_1, S_2, \ldots, S_n$), for example, using



(a) One-time Preprocessing Phase



(b) Randomness Generation for session i

Fig. 1: Overview of the InstaRand Protocol

Shamir's secret sharing scheme. This sharing is implemented using a Distributed Key-generation (DKG) protocol. On receiving an input x, each party S_i computes a partial evaluation-proof pair (y_i, π_i) using their share of the secret key sk_i. An aggregator, who may not hold any secret key and could be one of the servers in the VRF committee, can then publicly gather at least $t+1 \leq n$ such partial evaluations to aggregate them into the final output y and an accompanying proof π.[17] Due to its generality, the InstaRand server can be easily distributed by using the distributed VRF protocol from [GLOW21] based on BLS signatures [BLS01] using GLOW-DVRF [GLOW21].

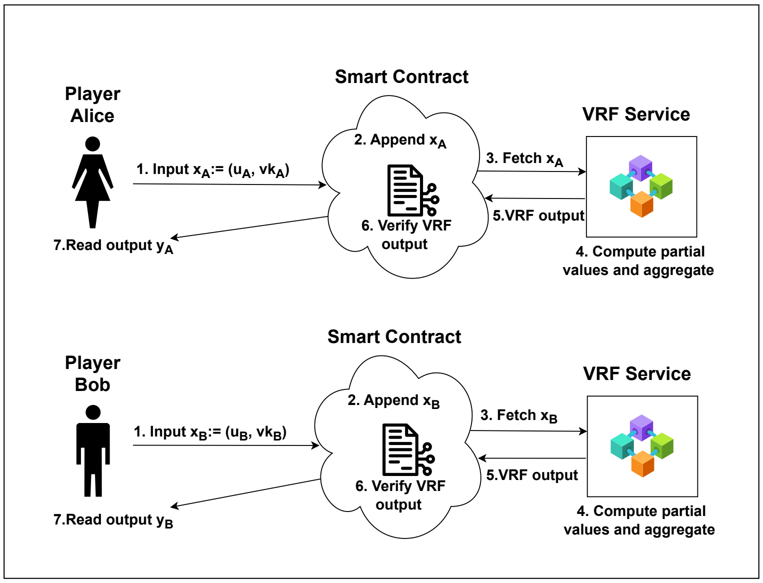
2.4 Application of InstaRand

Web3 Gaming (aka GameFi [Wor]). InstaRand is primarily useful in applications like Web3 gaming that demand a lot of instantly verifiable randomnesses. Consider a Web3 Backgammon game between two players, where the game needs to generate a random dice (values 1-6) for each round, and anyone should be able to publicly verify that the dice value was indeed generated using the VRF output of a randomness service [Chaa, Ban, Sup]. Moreover, it is not known in advance how many rounds a game will last so there is no fixed upper bound on the number of dice rolls that would be needed (such that they can be preprocessed using protocols like FlexiRand). Games like this necessitate on-demand solutions that are (i) cost-effective; (ii) instantly available (without much delay); (iii) instantly (and independently) verifiable – InstaRand meets all these requirements.

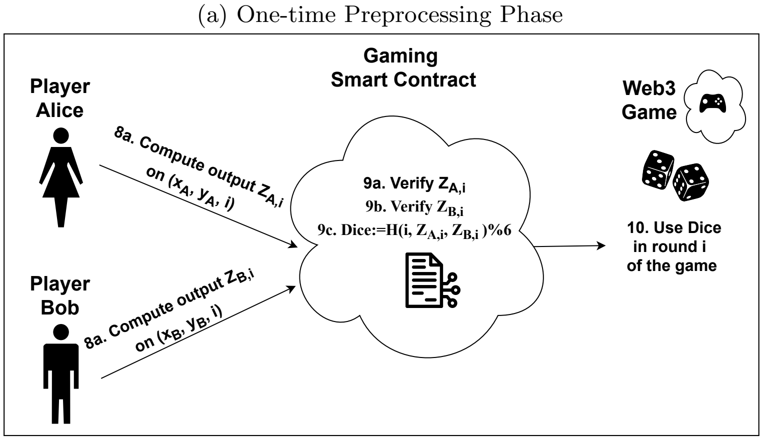
Two-Player Dice Game. We describe a subprotocol where two players – Alice and Bob, use InstaRand to roll a dice. Such a subprotocol can be used in bigger protocols for playing games like Yahtzee [carb], Snake-and-Ladder [pok], Backgammon [cara].

- *Input Generation:* Alice and Bob initially sample VRF keys (vk_A, sk_A) and (vk_B, sk_B). They consider their input as $x_A := (u_A, vk_A)$ and $x_B := (u_B, vk_B)$ respectively, where u_A and u_B are specific dates when vk_A and vk_B will be considered valid. The VRF public keys and u_A and u_B values are stored on-chain.
- *One-time Preprocessing:* Alice and Bob request VRF evaluation on x_A and x_B from the server VRF service to obtain y_A and y_B respectively.
- *Dice Generation:* During the game, to generate the dice for round i the players locally compute their outputs $z_{A,i}$ and $z_{B,i}$ from (x_A, y_A, i) and (x_B, y_B, i) using their secret keys sk_A and sk_B respectively using InstaRand. The parties reveal their outputs $z_{A,i}$ and $z_{B,i}$ on-chain. The gaming smart contract verifies

[17] We emphasize that, as every VRF output is publicly verifiable, an aggregator cannot successfully submit a wrong output to the blockchain. It is only required for the protocol's liveness, even under asynchrony. For enhanced liveness it is possible to also instantiate the aggregator using a separate group of nodes, and in that case the any-trust assumption (n>t) on the aggregator is sufficient: as long as there is a single honest aggregator, the liveness is maintained. In particular, we do *not* need any honest majority assumption.



(a) One-time Preprocessing Phase



(b) Dice Generation for Round i in Dice-Game

Fig. 2: Two-player Dice Game using InstaRand

$z_{A,i}$ and $z_{B,i}$ and then uses a hash function (modeled as a random oracle) to derive $\mathsf{dice}_i = 1 + \mathsf{H}(i, z_{A,i}, z_{B,i}) \bmod 6$, where $\mathsf{H} : \{0,1\}^* \rightarrow \mathbb{N}$. In case, one of the parties withholds their InstaRand outputs for a prespecified time and prevents the dice generation, then that party is considered to have forfeited the game.

We argue the following properties of the generated die:

- *Pseudorandomness:* The value dice_i is guaranteed to be pseudorandom as both $z_{A,i}$ and $z_{B,i}$ are instantly generated outputs of VRF, and the dice is obtained by applying the random oracle on both.
- *Public Pre-Verifiability:* The server's response can be verified by anyone since everything happens on-chain.
- *Instant (public) Verifiability:* The dice_i value can be verified by verifying $z_{A,i}$ and $z_{B,i}$ values, and this does not even leak $\mathsf{dice}_{j>i}$ values due to the $i\mathsf{VRF}$ property. This verification can happen either on-chain, or off-chain by the other player.[18]
- *Unpredicatability of $\mathsf{dice}_{j>i}$:* Even if Bob verifies the current $z_{A,i}$, future dice values $\mathsf{dice}_{j>i}$ remain unpredictable to Bob as $z_{A,j}$ remains private untill Alice reveals it, and vice versa.
- *Instant Availability:* Alice (resp. Bob) player may execute the pre-processing ahead of time, and when demanded, can just instant compute $z_{A,j}$ (resp. $z_{B,j}$) locally. This avoids any delay incurred due to interaction.

The dice remains unpredictable even if one of the parties collude with the gaming platform due to the unpredictability of the extended output of the other party. We also note that even if Alice and Bob are corrupt, still the dice is unbiasable due to the unbiasability of the InstaRand outputs. We provide a visualization of the dice game in Fig. 2. We also compare the InstaRand approach with other approaches – like VRF services-based, FlexiRand-based, coin-tossing-based, and other VRF-based solutions for completeness – this is provided in Appendix. A.

3 Preliminaries

We introduce the formal notations, recall the security models, and describe the necessary functionalities and primitives necessary for building our protocols.

3.1 Notations

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 $n \in \mathbb{N}$). We denote the

[18] We remark that it is important to have an on-chain pre-verification, as that is crucial for deploying a reward mechanism for the servers. However, the final verification, in this context, can just take place off-chain, as the other player would be interested to check whether the provided value is correctly computed. In other applications, however, it can also take place on-chain if required, as anyone can publicly verify this.

security parameter by λ. We assume that every algorithm takes λ as an implicit input. We use $y := \mathsf{D}(x)$ to denote the evaluation of a specifically deterministic algorithm D on input x to produce output y. Also, we write $x := \mathsf{val}$ to denote the assignment of a value val to the variable x. We use $x = y$ to check equality between x and y. We write $\mathsf{R}(x) \rightarrow y$ or $y \leftarrow \mathsf{R}(x)$ to denote the evaluation of a probabilistic algorithm R on input x to produce output y. We denote a randomized algorithm that runs in polynomial time as a probabilistic polynomial time (PPT) algorithm. We denote a negligible function in security parameter λ as $\mathsf{negl}(\lambda)$. We denote computational indistinguishability between two distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ as $\mathcal{D}_1 \stackrel{c}{\approx} \mathcal{D}_2$, i.e. no PPT adversary can distinguish between the two distributions, except with negligible probability. We denote a random oracle query on input message m as $\mathsf{H}(m) \in \mathcal{R}$ where the output is sampled from range $\mathcal{R}$. We prove security in the UC model. Formal details are in Appendix.B.

Random Function. The function $\mathsf{Rand}_{\mathcal{D}}(u)$ is defined over input $u \in \{0,1\}^*$ and output distribution $\mathcal{D}$ as:

$$\begin{aligned}\mathsf{Rand}_{\mathcal{D}}(u) &\coloneqq \text{ If } \exists (u,r) \in T, \text{ return } r.\\ &\coloneqq \text{ Else, sample } r \leftarrow \mathcal{D}, T \coloneqq T \cup (u,r)\\ &\quad\;\; \text{and return } r.\end{aligned}$$

where list T is initialized to the empty set, i.e. $T \coloneqq \emptyset$. Looking ahead, $\mathsf{Rand}_{\mathcal{D}}(\cdot)$ samples random values on fresh inputs from distribution $\mathcal{D}$ and it is maintained internally by $\mathcal{F}_{i\mathsf{VRF}}$.

3.2 Building Block: Verifiable Random Function

In this subsection, we recall the notion of *verifiable random functions* (VRF) from [CL07]. It ensures that the output on a previously unqueried input x is pseudorandom. A VRF over a distribution $\mathcal{D}$ is defined using a tuple of three PPT algorithms.

- $\mathsf{Gen}(1^\lambda) \rightarrow (vk, sk)$: It outputs the public verification key vk and secret key sk.
- $\mathsf{Eval}(sk, x) \rightarrow (y, \pi)$: It outputs a random string $y \in \mathcal{Y}$ and a proof π on input $x \in \mathcal{X}$.
- $\mathsf{Verify}(vk, x, (y, \pi)) \rightarrow b \in \{0,1\}$: It verifies the proof π of correct generation of y.

A VRF needs to satisfy the properties of correctness, uniqueness, and pseudorandomness. We refer to Def. 1 in Appendix B.3 for the formal definition. We also discuss VRF constructions from BLS-signatures [BLS01], RSA [GRPV23], and DDH [GRPV23] in Appendix D.

3.3 Building Block: Distributed VRF

In this subsection, we recall the notion of distributed VRF from [GLOW21]. A t-out-of-n DVRF over a distribution $\mathcal{D}$ is defined using a tuple of four PPT algorithms.

- $\mathsf{DKG}(1^\lambda, t, n) \rightarrow (vk, \mathbf{S}, \{vk_i, sk_i\}_{i\in[n]})$: It is the fully distributed key generation protocol that takes as input λ, number of parties n and the threshold t and outputs a set of qualified nodes $\mathbf{S}$, a global public verification key vk, a list $\{vk_1, vk_n\}$ of participating nodes' verification keys, and results in a list $\{sk_1, \ldots, sk_n\}$ of nodes' secret keys where each secret key is only known to the corresponding node.
- $\mathsf{PEval}(sk_i, x_i, vk_i) \rightarrow (i, y_i, \pi_i)$: Takes as input the secret key sk_i, input $x \in \mathcal{X}$ and verification key vk_i and outputs a triple (i, y_i, π_i) where $y_i \in \mathcal{Y}$ is the ith evaluation share and π_i is the corresponding proof.
- $\mathsf{PEvalVer}(vk_i, x, (i, y_i, \pi_i)) \rightarrow 0/1$: Takes as input the i verification key vk_i, input x and the partial evaluation triple (i, y_i, π_i) and outputs a decision bit denoting whether the verification succeeded or not.
- $\mathsf{Aggregate}(vk, \{vk_i\}_{i\in[n]}, x, \mathcal{E}) \rightarrow (y, \pi)$: Takes as input the global verification key vk, list of individual node verification keys $\{vk_i\}_{i\in[n]}$, input x, and a set $\mathcal{E} = \{(i_1, y_{i_1}, \pi_{i_1}), \ldots, (i_{|\mathcal{E}|}, y_{i_{|\mathcal{E}|}}, \pi_{i_{|\mathcal{E}|}})\}$ of verified partial evaluations originating from $|\mathcal{E}| > t + 1$ different nodes, and outputs the aggregated pseudorandom output y and correctness proof π.
- $\mathsf{Verify}(vk, x, (y, \pi)) \rightarrow b \in \{0, 1\}$: Takes as input the verification key vk, input x, output y, and proof π, and outputs a decision bit denoting whether the proof verified or not.

A DVRF needs to satisfy the properties of correctness, uniqueness, and strong pseudorandomness. We refer to Def. 2 in Appendix B.4 for the formal definition.

4 Instantly Verifiable VRF (iVRF)

We define the notion of instantly verifiable VRF via our ideal functionality $\mathcal{F}_{i\mathsf{VRF}}$ and show how to implement it using the InstaRand protocol.

4.1 Ideal Functionality $\mathcal{F}_{i\mathsf{VRF}}$

We present the ideal functionality $\mathcal{F}_{i\mathsf{VRF}}$ in Fig. 3. The client makes a single VRF query x to the server S. The functionality generates the server's output (y, π) and returns it to the client C. The server output (x, y, π) can be verified by anyone using the Pre-Verify command. Then the functionality generates N pairs of outputs - (i, z_i, δ_i) to capture the instant generation of client randomness from (x, y). These output pairs are returned to the client and registered in the memory corresponding to input (x, i). Each i-th output pair $(vk, x, i, z_i, \delta_i)$ is independently random and can be verified by any verifier V using the Inst-Verify command. The ideal functionality keeps track of the following variables:

Setting and parameters. The functionality interacts with servers S, clients C, and public verifiers V and an ideal world adversary Sim. It is parameterized with an integer N and an output distribution $\mathcal{D}$.

- **On** (S, vk) **from** Sim**:** Check if S was already registered, if not then register (S, vk).
- **On** (x, vk) **from a client** C**:** Skip unless (x, vk) is a fresh pair *and* (S, vk) was registered for some S. Otherwise:

  1. *Server Computation:* Forward the request to Sim, once Sim sends back (x, y, π) register (vk, x, y, π) and send it to C.
  2. *Instant Output Generation:*
     (a) If both S and C are corrupt: For every $i \in [N]$ receive (i, z_i, δ_i) from Sim and register $(vk, x, (i, z_i, \delta_i))$. Ignore future evaluation requests for the same (vk, x, i).
     (b) Otherwise, compute $z_i := \mathsf{Rand}_{\mathcal{D}}(x, y, i)$ for $i \in [1 \dots N]$ and send them to Sim, wait for the Sim to send back the proofs $(\delta_1, \dots, \delta_N)$. Then register $(vk, x, (i, z_i, \delta_i))$ and send it to C for $i \in [N]$. **(Random Output)**

- **On** $(\mathsf{Pre\text{-}Verify}, (vk, x, y, \pi))$ **from anyone**: If the entry (vk, x, y, π) is registered, then return 1, else return 0.
- **On** $(\mathsf{Inst\text{-}Verify}, (vk, x, i, z_i, \delta_i))$ **from anyone**: If the entry $(vk, x, (i, z_i, \delta_i))$ is registered with the functionality, then return 1, else return 0. **(Instantly Verifiable Output)**

Fig. 3: Ideal Functionality $\mathcal{F}_{i\mathsf{VRF}}$ for modeling VRF with instantly verifiable output

– Server's verification key vk,
– Client's input x,
– Server's one-time VRF output (y, π) evaluated on x - where y is the output and π is the proof,
– Client's instant verifiable output (i, z_i, δ_i) for session i on input x - where z_i is the output and δ_i is the proof.

Based on the corruption case, $\mathcal{F}_{i\mathsf{VRF}}$ ensures:

– *If either the server or the client is honest:* Then the outputs $(z_1, \ldots, z_N)$ is uniformly random.
– *If the client is honest:* Then an external adversary cannot predict the output z_j given the input x, server output (y, π) and the output of any (z_i, δ_i) for $i \neq j$.
– *If both server and the client are corrupt:* Then the adversary provides the outputs to the functionality. This is captured in Step. 2a.

We present our functionality in Fig. 3 and we discuss the properties that $\mathcal{F}_{i\mathsf{VRF}}$ provides:

1. **Random Output:** The outputs of each session i is uniformly random over the output distribution $\mathcal{D}$ if either the client or the server is honest.
2. **Instantly Available:** Each of the N outputs and proofs (z_i, δ_i) are instantly available to the client upon querying $\mathcal{F}_{i\mathsf{VRF}}$ with (x, vk).
3. **Instantly Verifiable Output:** Each of the outputs and proofs (z_i, δ_i) can be verified just given the server verification key vk, client input x and the session identifier i.
4. **Output Privacy:** If the client is honest then the output z_j is unpredictable even given the outputs $(z_1, \ldots, z_{j-1}, z_{j+1}, \ldots, z_N)$ and proofs $(\delta_1, \ldots, \delta_{j-1}, \delta_{j+1}, \ldots, \delta_N)$ of all other sessions.

$\mathcal{F}_{i\mathsf{VRF}}$ can also be modified to capture a single instance of an output-private VRF, defined in Flexirand [KMMM23]. To do so, set $N = 1$ and then run the functionality where z_1 is the output and δ_1 is the proof of the single instance of the output-private VRF. Instant-Verification of $\mathcal{F}_{i\mathsf{VRF}}$ is the verification algorithm of the single instance of the output-private. By making these modifications, our functionality captures the private VRF functionality of Flexirand. The work of [DGKR18] also proposed a VRF ideal functionality but their functionality is stronger as it requires the VRF output y to remain unpredictable when the server is corrupt. Moreover, they do not support the generation of instant verifiable outputs.

4.2 InstaRand Protocol π_{IRand}

The InstaRand protocol allows a VRF client to generate multiple random outputs, that are independently verifiable, using a single query to the VRF server. To do so, the client samples a key pair (pk_c, sk_c) for a client-side VRF protocol, denoted as VRF_c. The client makes a VRF query to the server with input $x = (u, vk_c)$ where u is the client's input. The server computes the VRF_s on x as y:

$$(y, \pi) \coloneqq \mathsf{VRF}_s.\mathsf{Eval}(sk_s, x),$$

where $x = (u, pk_c)$. The output y acts as an intermediate seed for the client to generate multiple independently verifiable randomness. For a session i, the client uses the secret key sk_c corresponding to pk_c to generate verifiable randomness on (x, y, i) as (w_i, ρ_i) by evaluating VRF_c.

$$(w_i, \rho_i) \coloneqq \mathsf{VRF}_c.\mathsf{Eval}(sk_c, (x, y, i))$$

The final randomness for session i is generated as z_i

$$z_i = \mathsf{H}(i, w_i, x, y),$$

where H is modeled as a random oracle. The proof for z_i is $\delta_i = (\pi, \rho_i, w_i, y)$. Note that the final output z_i is generated using a random oracle on both w_i and y. This is done to ensure that z_i is pseudorandom when either the client VRF_c output w_i, or the server VRF_s outputs y is pseudorandom. Similarly, if the client is honest then z_i is unpredictable to an external party since w_i is unpredictable. z_i also remains unpredictable given the verifiable output (z_j, δ_j) for session $j \neq i$ due to the unpredictability of w_i. This allows instant generation of verifiable randomness for different sessions, where each randomness can be independently verified. We present the InstaRand protocol, denoted as π_{IRand}, in Fig. 4. We prove that π_{IRand} implements $\mathcal{F}_{i\mathsf{VRF}}$ by proving Thm. 1.

Theorem 1. *Assuming* $(\mathsf{VRF}_s, \mathsf{VRF}_c)$ *are verifiable random functions with unique and pseudorandom outputs, then* π_{IRand} *(Fig. 4) implements* $\mathcal{F}_{i\mathsf{VRF}}$ *(Fig. 3) in the random oracle model against malicious corruption of the server, the client and the public verifier by a PPT adversary* $\mathcal{A}$*.*

Proof Sketch. The correctness of the VRF_s and VRF_c protocols ensure that the server output y and each output z_i will be publicly verifiable by verifying the proofs π and δ_i respectively. Next, we argue pseudorandomness of y and z_i. If the server is honest then the output y remains unpredictable and pseudorandom to everyone else due to the pseudorandomness of VRF_s. When the client is honest, each output z_i becomes unpredictable and pseudorandom due to the pseudorandomness of VRF_c. This holds even if the server is corrupt since z_i is obtained by performing a random oracle query on the server output y and the client's output w_i. Next, consider the case where the server is honest and the client is corrupt. In this scenario, each output z_i should satisfy unbiasability. When the client makes the VRF query x, the server computes y and this fixes each z_i value as the client side VRF computation on each input (x, y, i) is unique. Moreover, the z_i values will be uniformly distributed and unpredictable to the client when it queries the server. That is because the output y is unpredictable to the client and z_i is obtained by invoking the random oracle of y and unique

Primitives. $\mathsf{VRF}_s, \mathsf{VRF}_c$ are two verifiable random functions, $\mathsf{H} : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ is a random oracle.
Parties. VRF server S, client C, and public verifier V.

Server Key Gen. The server S generates $(vk_s, sk_s) \leftarrow \mathsf{VRF}_s.\mathsf{Gen}(1^\lambda)$ and posts vk_s to the bulletin board as the server verification key.

Server interaction (One-time Preprocessing)

Client Key Gen. The client samples $(vk_c, sk_c) \leftarrow \mathsf{VRF}_c.\mathsf{Gen}(1^\lambda)$ and posts vk_c to the bulletin board.

Client Input Gen. Assume client has input $u \in \{0,1\}^* \cup \varepsilon$. The client sends the input $x \coloneqq (u, vk_c)$ to the server.

Server VRF Evaluation. If x is fresh then the server computes $(y, \pi) \coloneqq \mathsf{VRF}_s.\mathsf{Eval}(sk_s, x)$, sends (y, π) to the client and stores x. Otherwise, ignore the evaluation request.

Client VRF Verification. The client verifies the generation of y by checking that $\mathsf{VRF}_s.\mathsf{Verify}(vk_s, x, (y, \pi)) \stackrel{?}{=} 1$, and sends $\perp$ to server if $\mathsf{VRF}_s.\mathsf{Verify}(vk_s, x, (y, \pi)) = 0$.

The following algorithms are run multiple times for different sessions $i \in [1 \ldots N]$.
Instant Output Generation. To generate verifiable random outputs for session i from y, the client performs the following:

$$(w_i, \rho_i) \coloneqq \mathsf{VRF}_c.\mathsf{Eval}(sk_c, (x, y, i)), \quad z_i = \mathsf{H}(i, w_i, x, y)$$

The client computes z_i as the output and the proof as $\delta_i = (\pi, \rho_i, w_i, y)$.

Pre-verification. To verify input (vk_s, x, y, π), return the output of $\mathsf{VRF}_s.\mathsf{Verify}(vk_s, x, (y, \pi))$.

Instant Verification. To verify input $(vk_s, x, i, z_i, \delta_i)$, parse $(\pi, \rho_i, w_i, y) \coloneqq \delta_i$ and $(u, vk_c) \coloneqq x$. Perform the following checks:

1. $\mathsf{VRF}_s.\mathsf{Verify}(vk_s, x, (y, \pi)) \stackrel{?}{=} 1$, *(one-time for each client input x)*
2. $\mathsf{VRF}_c.\mathsf{Verify}(vk_c, (x, y, i), (w_i, \rho_i)) \stackrel{?}{=} 1$, and
3. $z_i \stackrel{?}{=} \mathsf{H}(i, w_i, x, y)$.

If all checks pass then output z_i as the output for session i.

Fig. 4: InstaRand Protocol π_{IRand}

input parameters like x and i. The formal proof is more involved and we refer to Appendix. E for the detailed proof.

Instantiations. InstaRand protocol π_{IRand} can be constructed by instantiating VRF_c and VRF_s from the BLS-signatures based VRF [GLOW21], RSA-based VRF [GRPV23] or the DDH-based VRF [GRPV23]. We recall these VRF protocols in Appendix. D. For our empirical evaluation, we instantiate the client-side VRF_c with the DDH-based VRF protocol since the DDH-based protocol is the most practically efficient VRF protocol. However, it is not amenable to decentralization of the server. And so, the server side VRF_s in InstaRand is instantiated with the BLS-based VRF protocol. This is performed because we would like to distribute the server protocol by distributing the server-side VRF_s protocol in the next section. We will use the BLS-based GLOW-DVRF protocol for this purpose.

5 Instantly Verifiable DVRF (i-DVRF)

In a distributed setting, the VRF secret key sk (corresponding to verification key vk) is split among multiple servers, with a threshold $t+1$ out of n required to reconstruct the key. Even if up to t servers are compromised, the key remains secure. Clients interact with $t+1$ servers with input x. These servers compute their partial evaluations and send it to an aggregator A. The aggregator aggregates the partial evaluations to obtain the output $y = \mathsf{VRF}_{sk}(x)$ and an associated proof π. The aggregator returns (y, π) to the client. DVRF ensures that the output y is always publicly verifiable w.r.t to vk. In addition, y is pseudorandom if at most t servers are corrupt.

In addition to the above properties, we require that the server output y is consistent, i.e. y is independent of the participating server set, and liveness of the server computation, ensuring the protocol executes correctly regardless of malicious actions (guaranteed output delivery). Achieving these is straightforward: consistency is ensured using a t out of n secret sharing scheme like Shamir's, and availability/liveness is guaranteed by assuming $n \geq 2t+1$, which is ensured within our ideal functionality $\mathcal{F}_{i\text{-}\mathsf{DVRF}}$. Another requirement is robustness, ensuring that if aggregation is successful, so is pre-verification. This is enabled by allowing the Sim to verify the partial evaluations before aggregation and return $\perp$ if enough partial evaluations are not valid. Only, if $t+1$ evaluations are valid then the server VRF output y is registered.

5.1 Ideal functionality $\mathcal{F}_{i\text{-}\mathsf{DVRF}}$

We present the ideal functionality, denoted as $\mathcal{F}_{i\text{-}\mathsf{DVRF}}$, for capturing distributed VRFs that provide instantly verifiable outputs in Fig. 7 (Appendix. C). A set of servers - $(S_1, \ldots, S_n)$ denoted by $\mathbf{S}$, replaces the single VRF server. The public verification key vk is generated using a DKG. When the client makes a single VRF query x to the server set $\mathbf{S}$, the functionality collects partial evaluations from the servers. If there are enough valid evaluations then the functionality (via Sim) generates the server's output (y, π) and returns it to the client C. The server output (x, y, π) can be verified by anyone using the Pre-Verify command. The instant output generation of z_i values and their verification is the same as $\mathcal{F}_{i\mathsf{VRF}}$. Similar to $\mathcal{F}_{i\mathsf{VRF}}$, functionality $\mathcal{F}_{i\text{-}\mathsf{DVRF}}$ ensures that each ith output pair $(vk, x, i, z_i, \delta_i)$ is *independently verifiable*, and the z_i values are random and remain private unless the client discloses it.

5.2 Distributed InstaRand Protocol π_{DIRand}

We present the distributed version of InstaRand, denoted as π_{DIRand}, in Fig. 5. The construction is a natural extension of InstaRand (Fig.4) to the distributed setting where the server computation in the preprocessing phase is performed by a committee of server nodes. We assume that there is a DVRF protocol (refer to Def. 2 in Sec. 3.3). The servers jointly generate the public verfication key vk and server secret key shares sk_i using the DVRF.DKG protocol. When a client makes a query to the servers by sending input u to the servers, the servers perform partial evaluations using DVRF.PEval and send it to a public aggregator A. The public aggregator verifies the partial evaluations using DVRF.PEvalVer and then aggregates them to compute (y, π). The rest of the protocol is the same as the InstaRand protocol. The security of π_{DIRand} relies on the security of DVRF and VRF_c. Thm. 2 summarizes it.

Theorem 2. *Assuming DVRF is a distributed-VRF (Def. 2) and* VRF_c *is a verifiable random function with unique and pseudorandom outputs, then* π_{DIRand} *(Fig. 5) implements* $\mathcal{F}_{i\text{-}\mathsf{DVRF}}$ *(Fig. 7) in the random oracle model against malicious corruption of* t *among* n *servers, the client and the public verifier by a PPT adversary* $\mathcal{A}$*.*

Proof Sketch. The correctness of the DVRF and VRF_c protocols ensure that the server output y and each output z_i will be publicly verifiable by verifying the proofs π and δ_i respectively. Next, we argue pseudorandomness of y and z_i. If at most t servers are corrupt, then the output y remains unpredictable and pseudorandom to everyone else due to the strong pseudorandomness of DVRF. When the client is honest, each output z_i becomes unpredictable and pseudorandom due to the pseudorandomness of VRF_c. This holds irrespective of the pseudorandomness property of y since z_i is obtained by performing a random oracle query on the server output y and the client's output w_i. Next, we argue that each z_i value is unbiasable when the client is corrupt. When the client makes the VRF query x, the DVRF servers compute y, and this fixes each z_i value as the client-side VRF computation on each input (x, y, i) is unique. Moreover, the z_i values will be uniformly distributed and unpredictable to the client when it queries the server. That is because the output y is unpredictable to the client, and z_i is obtained by invoking the random oracle of y and unique input parameters like x and i. The formal proof of d-InstaRand is similar to the formal proof (in Appendix. E) of InstaRand except that the VRF server side algorithm is replaced by a DVRF.

Primitives. DVRF is a distributed verifiable random function, VRF_c is a verifiable random function, and $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ is a random oracle.
Parties. VRF servers $(S_1, \ldots, S_n)$, aggregator A, client C, and public verifier V.
Distributed Key Generation: Servers in set $\mathbf{S}$, jointly run the distributed key-generation $\mathsf{DVRF.DKG}(1^\lambda, t, n) \to (vk, \mathbf{S}, \{vk_j, sk_j\}_{j\in[n]})$ with security parameter λ, threshold t and total number of parties n. Each server S_j obtains a secret key sk_j. All the servers get the public verification keys $(vk, vk_1, \ldots, vk_n)$ and the list of qualified servers $\mathbf{S}$. The $(vk, vk_1, \ldots, vk_n)$ are posted on bulletin board.

Server interaction (One-time Preprocessing)

Client Key Gen. The client samples $(vk_c, sk_c) \leftarrow \mathsf{VRF}_c.\mathsf{Gen}(1^\lambda)$ and posts vk_c to the bulletin board.
Client Input Gen. Assume client has input $u \in \{0,1\}^* \cup \varepsilon$. The client sends the input $x \coloneqq (u, vk_c)$ to all the servers $S_1, \ldots, S_n$.
Server VRF Evaluation. If x is not fresh then the servers ignore x, otherwise the following steps are run:

- Each server $S_j \in S$ computes the partial evaluation $(j, y_j, \pi_j) \leftarrow \mathsf{DVRF.PEval}(sk_j, x_j, vk_j)$. Server S_j sends (y_j, π_j) to the aggregator A.
- The aggregator A initiates two sets $\mathbf{S}_\mathsf{A} := \emptyset$ and $\mathcal{E} := \emptyset$ and aggregates as:
  - For each $j \in [n]$, if $\mathsf{DVRF.PEvalVer}(vk_j, x, (j, y_j, \pi_j)) = 1$ then append j into S_A as $\mathbf{S}_\mathsf{A} := \mathbf{S}_\mathsf{A} \cup \{j\}$ and append (j, y_j, π_j) into $\mathcal{E}$ as $\mathcal{E} := \mathcal{E} \cup (j, y_j, \pi_j)$.
  - If $|\mathbf{S}_\mathsf{A}| < t + 1$ then output $\perp$, otherwise compute the output (y, π) by aggregating the partial evaluations in $\mathcal{E}$ as:

$$(y, \pi) := \mathsf{DVRF.Aggregate}(vk, \{vk_j\}_{j\in[n]}, x, \mathcal{E}).$$

  Aggregator A sends (y, π) as output to client.

Client VRF Verification. The client verifies y by checking that $\mathsf{DVRF.Verify}(vk, x, (y, \pi)) \stackrel{?}{=} 1$, and sends $\perp$ to the servers if it fails.
The following algorithms are run multiple times for different sessions $i \in [1 \ldots N]$.

Instant Output Generation. To generate verifiable random outputs for session i from y, the client sets $(w_i, \rho_i) \coloneqq \mathsf{VRF}_c.\mathsf{Eval}(sk_c, (x, y, i))$ and $z_i = \mathsf{H}(i, w_i, x, y)$. The client computes z_i as the output and the proof as $\delta_i = (\pi, \rho_i, w_i, y)$.
Pre-verification. To verify input (vk, x, y, π), return the output of $\mathsf{DVRF.Verify}(vk, x, (y, \pi)) \stackrel{?}{=} 1$.
Instant Verification. To verify input $(vk, x, i, z_i, \delta_i)$, parse $(\pi, \rho_i, w_i, y) \coloneqq \delta_i$ and $(u, vk_c) \coloneqq x$. Check:

1. $\mathsf{DVRF.Verify}(vk, x, (y, \pi)) \stackrel{?}{=} 1$, *(one-time for each client input x)*
2. $\mathsf{VRF}_c.\mathsf{Verify}(vk_c, (x, y, i), (w_i, \rho_i)) \stackrel{?}{=} 1$, and
3. $z_i \stackrel{?}{=} \mathsf{H}(i, w_i, x, y)$.

If all checks pass then output z_i as the output for session i.

Fig. 5: Distributed InstaRand Protocol π_{DIRand}

Instantiation. We instantiate the DVRF protocol using GLOW-DVRF [GLOW21] and it is based on BLS [BLS01] signatures. We recall it in Appendix D. Our client-side VRF is implemented using DDH-based VRF [GRPV23].

6 Experimental Results

Implementation and Setup. We implement the DDH-based VRF, GLOW-DVRF, FlexiRand, InstaRand and the distributed version of InstaRand in Rust. We also implement a Plonk [GWC19] proof for generating a proof proving the extended output of FlexiRand where the PRG is implemented using a Poseidon [GKS23] hash. In this case, the preprocessing phase consists of a single preprocessing phase of FlexiRand and a Plonk proof for committing the output on-chain using a hash. In the online phase, the client generates an output locally and a Plonk proof of correct computation. We run the server and client algorithms on an Apple MacBook Pro 18.3 with an 8-core M1 Pro processor and 16 GB RAM. We run our single-threaded implementation on MacOS Ventura 13.5.1.
Experiment Details. We benchmark the client and server algorithms of InstaRand and the DDH-based VRF [GRPV23] in the setting where the server is run by a single trusted node. The DDH-based VRF protocol and InstaRand client side VRF is implemented using the [GRPV23] VRF over Secp256k1 curve (used by Chainlink [Chad, line 56]). The server algorithm of InstaRand is run on the BN254 curve. Then we benchmark distributed InstaRand, GLOW-DVRF, and FlexiRand where the server node is replaced by a committee of nodes satisfying an honest-majority trust assumption. The server algorithm is all three protocols are run over the BN254 curve as it is the state-of-the-art curve for BLS-based protocols. The pairing-based protocols are implemented using substrate-bn [cra] from the crates.io library over BN254 curve. Finally, we compare the gas costs of all three protocols to demonstrate the practicality of deploying distributed InstaRand on-chain. We denote the distributed version of InstaRand and d-InstaRand. We benchmark the interactive preprocessing and non-interactive online costs separately. The preprocessing phase allows the client to make a query beforehand and use it later on in the application during the online phase. Only FlexiRand and InstaRand allow preprocessing of the output randomness. However, since we focus on the application where imstant verifiability is crucial, FlexiRand needs to interact to produce each randomn output.

6.1 Comparison of Client Computation Cost

We discuss the client computation cost for making a VRF query and then verifying the output. We show that InstaRand fares the same as the other protocols, like GLOW-DVRF and FlexiRand, in Table 2. In the DDH-based VRF and the BLS-based GLOW-DVRF, there is no preprocessing phase. In the online phase, the client makes the VRF query to the server, and then upon obtaining the server output and proof, the client verifies the proof. Verification cost is $0.21ms$ and $2.81ms$ respectively. In FlexiRand, the client cost in the preprocessing phase consists of the input blinding and pre-verification of the blinded output by verifying the proof. The online phase consists of unblinding the output. The total client cost is $3.19ms$. In FlexiRand+ZK-SNARK, the client additionally bears the burden of generating a hash of its private output, storing it on-chain and giving a Plonk proof of correct computation. This adds 80ms to the client cost. However, the preprocessing can be reused for multiple extended outputs. In InstaRand, the client cost in the preprocessing phase consists of sampling the keypair for the client side VRF in $0.17ms$. Then the client queries the server to obtain the output. The client then verifies this output in $2.81ms$. The client cost for preprocessing is $2.98ms$. It is a *one-time* cost that can be reused. The proof verification in DDH-VRF is cheaper since the client checks a proof of equal discrete logs. For others, the client performs a pairing check.

Protocols	$N = 1$		$N = 10$	
	Preprocessing Cost(ms)	Online Cost(ms)	Preprocessing Cost(ms)	Online Cost(ms)
DDH-VRF	-	0.21	-	2.1
GLOW-DVRF	-	2.81	-	28.1
FlexiRand	3.19	0.13	31.9	1.3
FlexiRand+ZK-SNARK	>83.19	>80	>83.19	>800
InstaRand, d-InstaRand	2.98	0.18	2.98	1.8

Table 2: Comparison of client's local computation costs for generating $N = 1, 10$ instantly verifiable outputs. Note that, DDH-VRF and GLOW-DVRF do not have pre-processing, and therefore do not support cost amortization or instant availability. Furthermore, since instant verifiability is mandated, FlexiRand needs to interact with VRF servers for each randomness.

Instant Output Generation. The DDH-based VRF, GLOW DVRF and FlexiRand does not allow instant output generation. As a result, we need to run these protocols N times to generate N outputs that are instantly verifiable. Meanwhile, we use the instant output generation property of InstaRand to generate N outputs in $0.18ms$ per output in the online phase. This requires the client to generate a proof of equal discrete logs. Each output can be verified in $0.22ms$ by verifying the proof. The preprocessing phase remains unchanged. This can be visualized in Table 2. It yields an instant improvement of $6\times$ over GLOW-DVRF and FlexiRand.

Estimates over Blockchains. In this setting, we assume there is a network delay of $120ms$ between client/server nodes and the blockchain, and it takes $\mathcal{T}ms$ for a transaction to get appended on the blockchain. In the DDH-based VRF and GLOW-DVRF, the client incurs a round trip delay of $480ms$ for making the request, servers reading the request, servers posting their output on blockchain and client reading the output. In addition, there is a $2\mathcal{T}ms$ for the request and fulfillment transaction, resulting in a total delay of $480 + 2\mathcal{T}ms$. In FlexiRand, there are four transactions - request transaction, verification of blinded input and fulfillment transaction with blinded output and unblinding the output, and one VRF evaluation by the servers, resulting in an additional $4\mathcal{T} + 960ms$ delay. Meanwhile, in InstaRand we incur two transactions in the preprocessing phase for the VRF evaluation by the server, resulting in an additional delay of $2\mathcal{T} + 480ms$. This cost gets amortized over multiple randomness requests. The online phase of InstaRand incurs $\mathcal{T}ms$ to use the output on-chain. Note that $\mathcal{T}$ is the main bottleneck in the delay as it varies from a few seconds (for Ethereum it takes 12 seconds [ych]) to even a few minutes (for Bitcoin).

6.2 Comparison of Server Computation Cost

Single Server Setting. In the DDH-based VRF, the VRF server evaluates the VRF of [GRPV23] on the client input over secp256k1 curve in $0.18ms$. In InstaRand, the server is implemented using the BLS protocol and takes $0.24ms$ over the BN254 curve. Operations over BN254 are more expensive compared to operations over the secp256k1 curve and that causes the additional $0.06ms$ delay. However, the server computation in InstaRand happens one-time in the preprocessing phase and can be reused in the online phase to generate multiple instantly verifiable outputs. We present the empirical results in Table 3.

Protocols	n	Preprocessing Cost(ms)	Online Cost(ms)
DDH-VRF	1	-	0.18
InstaRand	1	0.24	-
GLOW-DVRF	8	-	5.91
FlexiRand	8	6.32	-
FlexiRand+ZK-SNARK	8	6.32	-
d-InstaRand	8	5.91	-
GLOW-DVRF	16	-	11.35
FlexiRand	16	11.8	-
FlexiRand+ZK-SNARK	16	11.8	-
d-InstaRand	16	11.34	-
GLOW-DVRF	64	-	44.52
FlexiRand	64	45.19	-
FlexiRand+ZK-SNARK	64	45.19	-
d-InstaRand	64	44.52	-

Table 3: Comparison of server computation costs in the distributed n-server setting. For $n > 1$, the computation cost is the sum of a partial VRF evaluation, n partial verifications, and aggregation of n partial evaluations. The one-time preprocessing of InstaRand and d-InstaRand is reused for multiple requests.

Distributed Setting. In GLOW-DVRF, d-InstaRand and FlexiRand the server is implemented by an n-sized committee of server nodes who compute the partial

BLS-signatures and the proof of correct evaluation in parallel within $0.5ms$. These evaluations are sent to the aggregator who verifies n partial evaluations in $0.61ms$/evaluation and then aggregates them into the correct output. Aggregation and verification of partial evaluation grows with the value n. In Table 3, we present this entire computation cost assuming there is no network delay. Note that FlexiRand is slightly more expensive since the server nodes have to verify the blinded client input as well. The d-InstaRand server is also slightly faster than GLOW-DVRF server since we use an optimized (Appendix. D) BLS-based VRF where the servers need to compute one less hash. Considering network delay of $120ms$ between client/server and the blockchain, the servers in FlexiRand and InstaRand perform the VRF evaluations in the preprocessing phase, hence effectively restricting the client-server delay of $480ms$ to the preprocessing phase. Whereas GLOW-DVRF performs the VRF evaluation in the online phase and as a result, the GLOW-DVRF client incurs this delay in the online phase.

6.3 Comparison of Gas Cost Estimates

We perform the gas cost analysis in this section. All gas costs were taken as an average of 100 samples executed on Remix VM (London) over the Ethereum blockchain, which has a base fee [Git] of $21k$ for each transaction. Additionally, it takes $22k$ gas to store a commitment on-chain, $9k$ gas to hash to a group element on Secp256k1 curve, $62k$ gas to hash to a group element on BN254 curve, $80k$ gas to perform a pairing check. A BLS verification involves hashing to BN254 curve and a pairing check, costing $142k$ gas. The DDH-based VRF verification costs $33k$ gas - hashing to Secp256k1 curve and verifying a Proof of two equal discrete logs. It takes $300k$ gas [Tha] to store a hash on-chain and verify a Plonk proof of correct hash computation on-chain. Using these numbers, we analyze the gas costs for DDH-VRF, GLOW-DVRF, d-InstaRand, and FlexiRand (discussed in Appendix. F) in Table 4.

Protocols	$N = 1$		$N = 10$	
	Preprocessing Cost (gas)	Online Cost (gas)	Preprocessing Cost (gas)	Online Cost (gas)
DDH-based VRF	-	130k	-	1300k
GLOW-DVRF	-	196k	-	1960k
FlexiRand	306k	201k	3060k	2010k
FlexiRand+ZK-SNARK	>606k	>300k	>606k	$3000k$
d-InstaRand	302k	84k	**302k**	**840k**

Table 4: Comparison of gas costs for verifying N instantly verifiable outputs.

DDH-VRF and GLOW-DVRF. Both protocols have a request and a fulfillment transaction. The request transactions for both cost $54k$ gas – $22k$ gas for storing the request on-chain, $21k$ transaction, and the rest for bookkeeping purposes. The verification transaction costs for DDH-VRF and GLOW DVRF are $76k$ gas and $192k$ gas respectively. DDH-VRF incurs $33k$ gas for verifying

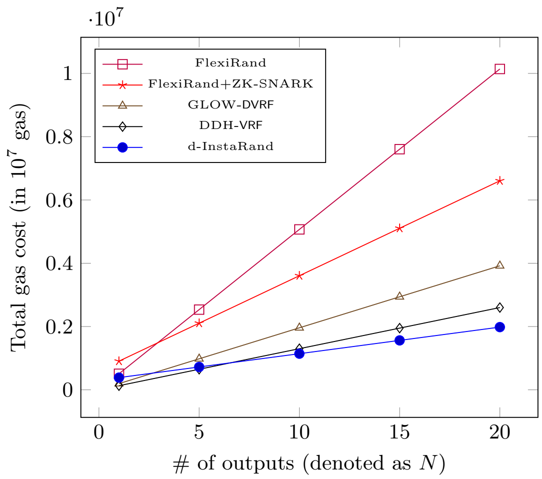


Fig. 6: Comparison of total gas cost for verifying N instantly verifiable outputs.

the proof and GLOW-DVRF incurs $192k$ gas for hashing to BN254 and performing a pairing. Besides that, $21k$ gas is spent for the transaction, and the rest for updating the fulfillment status of the request. Hence, the total gas cost of DDH-VRF and GLOW-DVRF are $130k$ and $192k$.

d-InstaRand. In the preprocessing phase, the client queries the GLOW-DVRF servers to obtain an evaluation. The request transaction costs $58k$, similar to the request transaction in GLOW-DVRF, except an extra $4k$ gas is spent since the query input consists of the additional client VRF verification key. The fulfillment transaction (namely Pre-verification) costs $244k$ gas – GLOW-DVRF verification plus storing the output y on-chain for later use. Hence, the preprocessing phase has a *one-time* cost of $302k$ gas or $0.97 USD. The online gas cost for d-InstaRand is $84k$ gas – $33k$ gas for verifying the client side VRF proof, $21k$ transaction cost, and the rest for obtaining the output by hashing and updating the fulfillment status of the request. Note that, it is slightly higher than the DDH-VRF verification due to performing an additional hash to obtain the final output. It costs $0.27 USD to generate each output.

In Fig.6, we compare the total gas costs of all the protocols for generating N outputs that can be independently verified. For generating $N = 1$ output, the DDH-based VRF is the least expensive since the three other protocols require performing a pairing check on the smart contract. For $N = 10$ outputs, GLOW-DVRF, FlexiRand and FlexiRand+ZK-SNARK are $1.7\times$, $4.4\times$ and $3\times$ more expensive than InstaRand. Also, d-InstaRand becomes less expensive than DDH-based VRF. This is mainly because DDH-VRF and GLOW-DVRF require $2N$ transactions to generate N outputs whereas InstaRand only requires $N + 2$ transactions. Each transaction costs $21k$ gas. FlexiRand requires $4N$ transactions and performs two pairing checks. Meanwhile, FlexiRand+ZK-SNARK approach runs the preprocessing of FlexiRand once and verifies a plonk proof in the preprocessing and online phases. However, InstaRand is consistently outperforms others, and the improvement further increases with higher values of N.

7 Conclusion

Web3 applications, such as on-chain gaming, require unbiased and publicly verifiable randomness that can be generated on demand and verified instantaneously within the application. Existing services, such as those using Verifiable Random Functions (VRF), are significantly slow, or other solutions, like FlexiRand [CCS 2023], lack instant verification capability. To solve this, we introduce an instantly verifiable VRF (iVRF) that generates multiple randomnesses from one VRF output seed, such that each value can be verified independently. This is the *first* cost-effective solution for generating randomness that is immediately available and instantly verifiable.

We build an iVRF using the InstaRand construction – that combines any (possibly distributed) VRF at the server's end with another VRF at the client's end to build an iVRF. Our specific instantiation uses the BLS-based GLOW-DVRF [Euro S&P 2021] at the server's end and the DDH-based VRF of Goldberg et al. [RFC 2023] at the client's end. An InstaRand client incurs a *one-time* pre-processing cost to generate the seed (or server's VRF output) by querying the GLOW-DVRF servers once. Once the seed is set, the client locally generates the random value on demand in 0.18 ms. This avoids the client-server round trip delay (of 240 ms), hence generating outputs instantly.

References

AMMM18. Shashank Agrawal, Peihan Miao, Payman Mohassel, and Pratyay Mukherjee. PASTA: PASsword-based threshold authentication. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 2042–2059, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

Apt. Aptos. Aip-41 - move apis for randomness generation. https://github.com/aptos-foundation/AIPs/blob/main/aips/aip-41.md.

Ban. Band vrf guaranteed integrity on the blockchain. https://www.bandprotocol.com/vrf.

BBBF18. Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 757–788, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

BDD$^+$23. Carsten Baum, Bernardo David, Rafael Dowsley, Ravi Kishore, Jesper Buus Nielsen, and Sabine Oechsner. CRAFT: composable randomness beacons and output-independent abort MPC from time. In *PKC'23*, volume 13940 of *Lecture Notes in Computer Science*, pages 439–470. Springer, 2023.

BGG$^+$18. Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 565–596, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

BLS01. Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 514–532, Gold Coast, Australia, December 9–13, 2001. Springer, Heidelberg, Germany.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

Can04. Ran Canetti. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, page 219. IEEE Computer Society, 2004.

cara. cardgames.io. Backgammon. https://cardgames.io/backgammon/.

carb. cardgames.io. Yahtzee. https://cardgames.io/yahtzee/.

CCB23. Miranda Christ, Kevin Choi, and Joseph Bonneau. Cornucopia: Distributed randomness beacons at scale. *IACR Cryptol. ePrint Arch.*, page 1554, 2023.

Chaa. Chainlink vrf: On-chain verifiable randomness. https://blog.chain.link/chainlink-vrf-on-chain-verifiable-randomness/.

Chab. Chainlink. The chainlink network in 2023. https://blog.chain.link/the-chainlink-network-in-2023/.

Chac. Chainlink. Solving Web3's Latency Problem to Unleash High-Speed dApps . https://blog.chain.link/solving-low-latency-problem/.

Chad. Chainlink. Usage of Seckp256 curve by Chainlink. https://github.com/smartcontractkit/chainlink/blob/develop/contracts/src/v0.8/vrf/VRF.sol.

Chi. Chia. https://www.chia.net/.

CJS14. Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical UC security with a global random oracle. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 597–608, Scottsdale, AZ, USA, November 3–7, 2014. ACM Press.

CL07. Melissa Chase and Anna Lysyanskaya. Simulatable VRFs with applications to multi-theorem NIZK. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 303–322, Santa Barbara, CA, USA, August 19–23, 2007. Springer, Heidelberg, Germany.

cra. crates.io. substrate-bn: Pairing cryptography with the Barreto-Naehrig curve . https://crates.io/crates/substrate-bn.

CSW20. Ran Canetti, Pratik Sarkar, and Xiao Wang. Efficient and round-optimal oblivious transfer and commitment with adaptive security. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 277–308, Daejeon, South Korea, December 7–11, 2020. Springer, Heidelberg, Germany.

CSW22. Ran Canetti, Pratik Sarkar, and Xiao Wang. Triply adaptive UC NIZK. In *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part II*, volume 13792 of *Lecture Notes in Computer Science*, pages 466–495. Springer, 2022.

DDKL24. Bernardo David, Rafael Dowsley, Anders Konring, and Mario Larangeira. MUSEN: aggregatable key-evolving verifiable random functions and applications. In *Security and Cryptography for Networks - 14th International Conference, SCN 2024, Amalfi, Italy, September 11-13, 2024, Proceedings, Part II*, volume 14974 of *Lecture Notes in Computer Science*, pages 317–337. Springer, 2024.

DGKR18. Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 66–98, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

DKIR22. Sourav Das, Vinith Krishnan, Irene Miriam Isaac, and Ling Ren. Spurt: Scalable distributed randomness beacon with transparent setup. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 2502–2517. IEEE, 2022.

DRaa. Drand: Distributed randomness beacon. https://drand.love.

Drab. Drand. Multi Frequency Support and Timelock Encryption Capabilities are coming in drand! https://drand.love/blog/2022/02/21/multi-frequency-support-and-timelock-encryption-capabilities/.

EEK+23. Muhammed F. Esgin, Oguzhan Ersoy, Veronika Kuchta, Julian Loss, Amin Sakzad, Ron Steinfeld, Xiangwen Yang, and Raymond K. Zhao. A new look at blockchain leader election: Simple, efficient, sustainable and post-quantum. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS 2023, Melbourne, VIC, Australia, July 10-14, 2023*, pages 623–637. ACM, 2023.

EKS+21. Muhammed F. Esgin, Veronika Kuchta, Amin Sakzad, Ron Steinfeld, Zhenfei Zhang, Shifeng Sun, and Shumo Chu. Practical post-quantum few-time verifiable random function with applications to algorand. In *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part II*, volume 12675 of *Lecture Notes in Computer Science*, pages 560–578. Springer, 2021.

Esc24. Zoo Escape. BACKGAMMON DICE STATISTICS. https://zooescape.com/backgammon-stats.pl, 2024.

EY24. Ehsan Ebrahimi and Anshu Yadav. Strongly secure universal thresholdizer. In *Advances in Cryptology - ASIACRYPT 2024 - 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, December 9-13, 2024, Proceedings, Part III*, volume 15486 of *Lecture Notes in Computer Science*, pages 207–239. Springer, 2024.

GHK+24. Jacob Gorman, Lucjan Hanzlik, Aniket Kate, Easwar Vivek Mangipudi, Pratyay Mukherjee, Pratik Sarkar, and Sri Aravinda Krishnan Thyagarajan. Vraas: Verifiable randomness as a service on blockchains. *IACR Cryptol. ePrint Arch.*, page 957, 2024. Accepted in 38th IEEE Computer Security Foundations Symposium.

Git. Gitbook. Base Fee for an Ethereum Transaction. https://ethereumbuilders.gitbooks.io/guide/content/en/design_rationale.html.

GKS23. Lorenzo Grassi, Dmitry Khovratovich, and Markus Schofnegger. Poseidon2: A faster version of the poseidon hash function. In *Progress in Cryptology - AFRICACRYPT 2023 - 14th International Conference on Cryptology in Africa, Sousse, Tunisia, July 19-21, 2023, Proceedings*, volume 14064 of *Lecture Notes in Computer Science*, pages 177–203. Springer, 2023.

GLOW21. David Galindo, Jia Liu, Mihai Ordean, and Jin-Mann Wong. Fully distributed verifiable random functions and their application to decentralised random beacons. In *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*, pages 88–102. IEEE, 2021.

GRPV23. Sharon Goldberg, Leonid Reyzin, Dimitrios Papadopoulos, and Jan Vcelák. Verifiable random functions (vrfs). *RFC*, 9381:1–47, 2023.

GS24. Emanuele Giunta and Alistair Stewart. Unbiasable verifiable random functions. In *Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part IV*, volume 14654 of *Lecture Notes in Computer Science*, pages 142–167. Springer, 2024.

GWC19. Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, page 953, 2019.

HMW18. Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018.

JKX21. Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. On the (in)security of the diffie-hellman oblivious PRF with multiplicative blinding. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 380–409, Virtual Event, May 10–13, 2021. Springer, Heidelberg, Germany.

Ken. Anne Kenyon. Randomness on Algorand. https://developer.algorand.org/articles/randomness-on-algorand/.

KMMM23. Aniket Kate, Easwar Vivek Mangipudi, Siva Maradana, and Pratyay Mukherjee. Flexirand: Output private (distributed) vrfs and application to blockchains. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 1776–1790. ACM, 2023.

Lab. Aptos Labs. Roll with Move: Secure, instant randomness on Aptos. https://aptoslabs.medium.com/roll-with-move-secure-instant-randomness-on-aptos-c0e219df3fb1.

Lai23. Yi-Fu Lai. CAPYBARA and TSUBAKI: verifiable random functions from group actions and isogenies. *IACR Cryptol. ePrint Arch.*, page 182, 2023.

Lai24. Yi-Fu Lai. Capybara and tsubaki: Verifiable random functions from group actions and isogenies. *IACR Commun. Cryptol.*, 1(3):1, 2024.

LW15. Arjen K. Lenstra and Benjamin Wesolowski. A random zoo: sloth, unicorn, and trx. *IACR Cryptol. ePrint Arch.*, page 366, 2015.

Mal24. Giulio Malavolta. Key-homomorphic and aggregate verifiable random functions. In *Theory of Cryptography - 22nd International Conference, TCC 2024, Milan, Italy, December 2-6, 2024, Proceedings, Part IV*, volume 15367 of *Lecture Notes in Computer Science*, pages 98–129. Springer, 2024.

Men24. Meng, Xuanji and Sui, Xiao and Yang, Zhaoxin and Rong, Kang and Xu, Wenbo and Chen, Shenglong and Yan, Ying and Duan, Sisi. Rondo: Scalable and Reconfiguration-Friendly Randomness Beacon. *Cryptology ePrint Archive*, 2024.

PBB$^+$. René Peralta, Harold Booth, Luís T. A. N. Brandão, John Kelsey, and Carl Miller. Interoperable randomness beacons. https://csrc.nist.gov/projects/interoperable-randomness-beacons/beacon-20.

pok. poki. Snake-and-Ladder. https://poki.com/en/g/snakes-and-ladders.

SJSW20. Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. Hydrand: Efficient continuous distributed randomness. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 73–89, 2020.

ST22. Ori Shem-Tov. Usage and best practices for randomness beacon, Sep 2022. https://developer.algorand.org/articles/usage-and-best-practices-for-randomness-beacon/.

Sup. Supra whitepaper. https://supraoracles.com/docs/SupraOracles-VRF-Service-Whitepaper.pdf.

Tha. Justin Thaler. SNARK Security and Performance . https://a16zcrypto.com/posts/article/snark-security-and-performance/.

Vee. Veedo — stark-based verifiable delay function. https://github.com/starkware-libs/veedo.

Wor. World Economic Forum. What is GameFi - and how could crypto regulations shape it? https://www.weforum.org/agenda/2022/11/gamefi-finance-shaped-by-crypto-regulations/.

ych. ycharts. Ethereum Average Block Time. https://ycharts.com/indicators/ethereum_average_block_time.

YSWW21. Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2986–3001, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

A Comparison of Different Dice Game Solutions

We compare our InstaRand solution (Fig 2) for the die roll with other solutions as follows:

- *Comparison with VRF services:* If we use a VRF service to play the same dice game, then the generation of each dice would require a fresh randomness query to the VRF server by the parties, incurring $240m$secs latency (assuming a standard $120m$secs server-client network latency). InstaRand clients avoid this delay.
- *Comparison with FlexiRand:* If we use FlexiRand for this game, then all the dice (an average two-player Backgammon game takes 53.78 rolls [Esc24]) have to be preprocessed/precomputed since its output is not independently verifiable. This avoids the online delay of the previous approach, albeit at the cost of blowing up the preprocessing phase. In contrast, the preprocessing phase of InstaRand does not grow with the number of random values generated in the online phase.
- *Comparison with Coin-Toss:* An alternate solution is where the players commit to random values and then open them to generate the randomness. But this would require two on-chain transactions for one party and one transaction for the other party in the online phase. In contrast, the online phase of InstaRand requires a single on-chain transaction for both parties. Saving one transaction could save us a few seconds (on the Ethereum blockchain) to a few minutes (on the Bitcoin blockchain) for rolling a single die since it is the major bottleneck in deploying Web3 dApps [Chac].
- *Comparison with client-run VRFs:* There is another approach to rolling the dice, similar to our protocol. The clients post their own VRF keys in the preprocessing phase and skip the query to the VRF service. In the online phase, the clients evaluate their VRFs on the round number i to generate $Z_{A,i}$ and $Z_{B,i}$ respectively. The gaming smart contract verifies those values and then generates the dice using the hash function. This approach works when at least one of the clients is honest. It has the same online cost as us and it skips the query to the VRF service. However, if the two clients collude then they can effectively bias the random dice value, as there is no contribution from an external VRF service in the dice randomness. The previous protocol based on commit-reveal coin-tossing also suffers from the same limitation. This can be a problem in a Snake-and-Ladder tournament where a bunch of players collude together such that one of them always wins their pairwise match and gets more points than the other honest players in the tournament.

B Additional Preliminaries

We present the additional preliminaries in this section.

B.1 Random Oracle

A random oracle (RO) [CJS14,CSW20] H is parameterized by an arbitrary domain and a specified range $\mathcal{R}$. An RO query on message m is denoted by $\mathsf{H}(m)$. The plain random oracle assumption guarantees that $\mathsf{H}(m)$ is indistinguishable from an element uniformly sampled from $\mathcal{R}$ if m was not queried before. An observable

RO additionally grants the simulator/reduction to observe (but not influence) the queries made, to H, by the adversary. A programmable RO [CSW20] allows the simulator/reduction to program the RO to output a value y on $\mathsf{H}(m) \coloneqq y$ on a previously unqueried input message.

B.2 Universal-Composability (UC) Model

We follow the Universal Composability Framework [Can01], in that a real-world multi-party protocol realizes an ideal functionality in the presence of an adversary. We assume the existence of a *default authenticated channel* in the real world between any two parties. This can be modeled using an ideal authenticated channel functionality [Can04]. We refer to the original work of Canetti [Can01] for a detailed description of the security model.

B.3 Building Block: Verifiable Random Function

In this subsection, we recall the notion of *verifiable random functions* (VRF) from [CL07]. It ensures that the output on a previously unqueried input x is pseudorandom. A VRF over a distribution $\mathcal{D}$ is defined using a tuple of three PPT algorithms.

- $\mathsf{Gen}(1^\lambda) \rightarrow (vk, sk)$: is the key generation algorithm that takes as input the security parameter λ and outputs a key pair (vk, sk), where vk is the public verification key and sk is the secret key.
- $\mathsf{Eval}(sk, x) \rightarrow (y, \pi)$: is the evaluation algorithm that takes as input a secret key sk and input $x \in \mathcal{X}$ and outputs a random string $y \in \mathcal{Y}$ and a proof π.
- $\mathsf{Verify}(vk, x, (y, \pi)) \rightarrow b \in \{0, 1\}$: is the verification algorithm that takes as input the verification key vk, input x, output y, and proof π, and outputs a decision bit denoting whether the proof verified or not.

Definition 1 (Verifiable Random Function). *A tuple of algorithms denoted as $\mathsf{VRF} = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$ is a verifiable random function (VRF) over input space $\mathcal{X}$ and output space $\mathcal{Y}$ if it fulfills:*

- ***Correctness****. For all $(vk, sk) \leftarrow \mathsf{Gen}(1^\lambda)$, $x \in \mathcal{X}$ and, $(y, \pi) \leftarrow \mathsf{Eval}(sk, x)$, it holds that $\mathsf{Verify}(vk,\ x,\ (y,\ \pi)) = 1$.*
- ***Uniqueness****. For all $vk \in \{0, 1\}^*$ that lie in the verification key space corresponding to $\mathsf{Gen}(1^\lambda)$, and for all $x \in \mathcal{X}$, there does not exist any pair of $(y_0, \pi_0), (y_1, \pi_1) \in \{0, 1\}^*$ s.t. $y_0 \neq y_1$, and $\mathsf{Verify}(vk,\ x,\ (y_0,\ \pi_0)) = \mathsf{Verify}(vk,\ x, (y_1,\ \pi_1)) = 1$.*
- ***Pseudorandomness****. On input vk, even with oracle access to $\mathsf{Eval}(sk, \cdot)$ no PPT adversary can distinguish y_0 (where $(y_0, \pi) \leftarrow \mathsf{Eval}(sk, x)$) from a uniform random element y_1 (where $y_1 \leftarrow \mathsf{Rand}_{\mathcal{Y}}(x)$) without explicitly querying for it. More formally, $\forall$ PPT adversaries $\mathcal{A}$ the following two distributions are computationally indistinguishable:*

$$(vk, x, y_0) \overset{c}{\approx} (vk, x, y_1),$$

where $(vk, sk) \leftarrow \mathsf{Gen}(1^\lambda)$ *is generated following the protocol. The adversary* $\mathcal{A}$ *outputs* $x \leftarrow \mathcal{A}^{\mathsf{Eval}(sk,\cdot)}(vk)$. *Then the output values generated as* $(y_0, \pi) \leftarrow \mathsf{Eval}(sk, x)$, *and* $y_1 \leftarrow \mathsf{Rand}_{\mathcal{Y}}(x)$. *And input* x *was not queried to* $\mathsf{Eval}(sk, \cdot)$, *i.e.* $x \notin Q_\gamma$ *for* $\gamma \in \{0, 1\}$, *where* Q_γ *is the list of queries made by* $\mathcal{A}$.

B.4 Building Block: Distributed VRF

In this subsection, we recall the notion of distributed VRF from [GLOW21]. A t-out-of-n DVRF over a distribution $\mathcal{D}$ is defined using a tuple of four PPT algorithms.

- $\mathsf{DKG}(1^\lambda, t, n) \rightarrow (vk, \mathbf{S}, \{vk_i, sk_i\}_{i\in[n]})$: is a fully distributed key generation protocol that takes as input the security parameter λ, number of parties n and the threshold t and outputs a set of qualified nodes $\mathbf{S}$, a global public verification key vk, a list $\{vk_1, vk_n\}$ of participating nodes' verification keys, and results in a list $\{sk_1, \ldots, sk_n\}$ of nodes' secret keys where each secret key is only known to the corresponding node.
- $\mathsf{PEval}(sk_i, x_i, vk_i) \rightarrow (i, y_i, \pi_i)$: is a partial evaluation algorithm that takes as input the secret key sk_i, input $x \in \mathcal{X}$ and verification key vk_i and outputs a triple (i, y_i, π_i) where $y_i \in \mathcal{Y}$ is the ith evaluation share and π_i is the corresponding proof.
- $\mathsf{PEvalVer}(vk_i, x, (i, y_i, \pi_i)) \rightarrow 0/1$: is an algorithm to verify the i partial evaluation and it takes as input the i verification key vk_i, input x and the partial evaluation triple (i, y_i, π_i) and outputs a decision bit denoting whether the verification succeeded or not.
- $\mathsf{Aggregate}(vk, \{vk_i\}_{i\in[n]}, x, \mathcal{E}) \rightarrow (y, \pi)$: is the aggregation algorithm that takes as input the global verification key vk, list of individual node verification keys $\{vk_i\}_{i\in[n]}$, input x, and a set $\mathcal{E} = \{(i_1, y_{i_1}, \pi_{i_1}), \ldots, (i_{|\mathcal{E}|}, y_{i_{|\mathcal{E}|}}, \pi_{i_{|\mathcal{E}|}})\}$ of verified partial evaluations originating from $|\mathcal{E}| > t + 1$ different nodes, and outputs the aggregated pseudorandom output y and correctness proof π.
- $\mathsf{Verify}(vk, x, (y, \pi)) \rightarrow b \in \{0, 1\}$: Takes as input the verification key vk, input x, output y, and proof π, and outputs a decision bit denoting whether the proof verified or not.

Definition 2 (Distributed Verifiable Random Function). *A tuple of algorithms denoted as* $\mathsf{DVRF} = (\mathsf{DKG}, \mathsf{PEval}, \mathsf{Aggregate}, \mathsf{Verify})$ *is a strongly pseudorandom distributed verifiable random function (DVRF) over input space* $\mathcal{X}$ *and output space* $\mathcal{Y}$ *if it fulfills:*

- ***Correctness****. For all* $(vk, \mathbf{S}, \{vk_i, sk_i\}_{i\in[n]}) \leftarrow \mathsf{DKG}(1^\lambda, t, n)$, $x \in \mathcal{X}$, *and all possible sets* Ω *of size* $t \leq |\Omega| \leq n$, $(y, \pi) \leftarrow \mathsf{Aggregate}(vk, \{vk_i\}_{i\in[n]}, x, \mathcal{E})$, *where* $\mathcal{E} = \{(i, y_i, \pi_i)\}_{i\in\Omega}$ *is the set of valid partial evaluations of nodes in set* Ω *i.e.* $\leftarrow \mathsf{PEval}(sk_i, x_i, vk_i) \wedge 1 \leftarrow \mathsf{PEvalVer}(\ vk_i,\ x, (i, y_i, \pi_i))$, *then it holds that* $\mathsf{Verify}(vk,\ x,\ (y,\ \pi)) = 1$.

- ***Uniqueness***. *For all* $vk \in \{0,1\}^*$ *that lie in the verification key space corresponding to* $\mathsf{Gen}(1^\lambda)$*, and for all* $x \in \mathcal{X}$*, there does not exist any pair of* $(y_0, \pi_0), (y_1, \pi_1) \in \{0,1\}^*$ *s.t.* $y_0 \neq y_1$*, and* $\mathsf{Verify}(vk,\ x,\ (y_0,\ \pi_0)) = \mathsf{Verify}(vk,\ x, (y_1,\ \pi_1)) = 1$.
- ***Strong Pseudorandomness***. *This says that even if the adversary corrupts* t *of the* n *signers, and makes partial evaluation queries on the challenge input, and yet the* VRF *evaluation on the challenge input is pseudorandom. It is stronger than the standard pseudorandomness of the* VRF *as the adversary is not allowed to make partial evaluation queries on the challenge input in the standard pseudorandomness game. We refer to [GLOW21] for the formal definition.*

C Ideal Functionality $\mathcal{F}_{i\text{-DVRF}}$ for Distributing $\mathcal{F}_{i\text{VRF}}$

We present the ideal functionality $\mathcal{F}_{i\text{-DVRF}}$ in Fig. 7 and we refer to Sec. 5.1 for more details on the functionality.

D VRF and DVRF Instantiations

We recall the BLS-based, RSA-based, and DDH-based VRF protocols. We also optimize the BLS-based construction such that the verification algorithm costs one less hash. This optimization reduces the "*Pre-verification*" in InstaRand. We also show that for each protocol given a VRF output y on input x and verification key vk, the simulator can check the correctness of the output y (without the VRF proof) under the random oracle assumption. This is required for our security proofs.

- **Optimized BLS-signature-based VRF [BLS01].** This VRF is defined over groups $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_T$ with a bilinear mapping $e : \mathcal{G}_1 \times \mathcal{G}_2 \to \mathcal{G}_T$. Let g_1 and g_2 be generators over $\mathcal{G}_1$ and $\mathcal{G}_2$ respectively. The verification key is $vk = g_2^{sk}$ and the secret key is sk. The input is $x \in \{0,1\}^\lambda$. We optimize this VRF s.t. the output is $y = \mathsf{H}(x)^{sk}$ where $\mathsf{H} : \{0,1\}^\lambda \to \mathcal{G}_1$ is a random oracle. The verifier checks $e(y, g_2) \stackrel{?}{=} e(\mathsf{H}(x), vk)$. The simulator also performs the same check to verify output correctness. This saves us one hash compared to the original BLS-based VRF of [GLOW21] where the final output is obtained by further hashing y. It is not a problem in InstaRand if y is a group element instead of a 256-bit string since it is used as an input in the client-side VRF to produce the z_i values. Such optimizations were also explored in FlexiRand.
- **RSA-based [GRPV23].** The verification key is (n, e) and the secret key is d. The input is x. The proof is $\rho = \mathsf{H}_1(x)^d \mod n$ and the output value is $y = \mathsf{H}_2(\rho)$ where H_1 is an IETF specified hash function and H_2 is a cryptographic hash function. The verification equation is $\rho^e \mod n \stackrel{?}{=} \mathsf{H}_1(x)$ and $y \stackrel{?}{=} \mathsf{H}_2(\rho)$. To check output correctness, the simulator observes the random oracle queries made to H_2 to find out ρ, and then the simulator runs the verification check to verify the output.

Setting and parameters. The functionality interacts with servers S, public aggregator A, clients C, and public verifiers V and an ideal world adversary Sim. It is parameterized with an integer N, number of servers n, server corruption threshold t and an output distribution $\mathcal{D}$.

– **On** $(\mathbf{S}, vk, \{vk_1, \ldots, vk_n\})$ **from Sim:** Parse $\mathbf{S} := \{S_1, \ldots, S_n\}$ and when vk is unique: **(Distributed Key Generation)**

1. Define $\mathsf{C_S} \subset \mathbf{S}$ is the set of corrupt servers and $\mathsf{H_S} := \mathbf{S} \setminus \mathsf{C_S}$ is the set of honest servers.
2. If $n < 2t + 1$, then exit the procedure. **(Liveness)**
3. For each $S_i \in \mathbf{S}$ set $Keys[S_i] := vk_i$.
4. If $|\mathsf{C_S}| \geq t + 1$, then mark server set $\mathbf{S}$ as `Corrupt`.
5. Send $(\mathbf{S}, vk, vk_i)$ to each $S_i \in \mathsf{H_S}$ and register $(\mathbf{S}, vk)$.

– **On** (x, vk) **from a client C:** Skip unless (x, vk) is a fresh pair *and* $(\mathbf{S}, vk)$ was registered for some $\mathbf{S}$. Otherwise:

1. *On* $(\mathit{Partial\text{-}Eval}, x, vk)$ *from server* S_j*:* Fetch $vk_j \leftarrow Keys[S_j]$ and forward $(\mathsf{Partial\text{-}Eval}, x, vk, vk_j)$ to Sim. If Sim sends $\perp$ then send it to S_j. If Sim sends (vk_j, x, j, y_j, π_j) then forward it to S_j and set $T_{\mathsf{part}}[x, vk, S_j] := (y_j, \pi_j)$. If the same query is repeated then fetch (y_j, π_j) from T_{part} and return it to S_j.
2. *On* $(\mathit{Aggregate}, x, vk, \{(y_1, \pi_1), \ldots, (y_\ell, \pi_\ell)\})$ *from* A*:* If $\ell < t + 1$, then return $\perp$. Otherwise, forward this message to Sim. If Sim sends $\perp$ then send it to A and exit. Otherwise, receive (vk, x, y, π) from Sim and register (y, π) as $T[x, vk] := (y, \pi)$ and send (vk, x, y, π) to C.
3. *Instant Output Generation:*
   (a) If both $\mathbf{S}$ and C are corrupt: For every $i \in [N]$ receive (i, z_i, δ_i) from Sim and register $(vk, x, (i, z_i, \delta_i))$. Ignore future evaluation requests for the same (vk, x, i).
   (b) Otherwise, compute $z_i := \mathsf{Rand}_{\mathcal{D}}(x, y, i)$ for $i \in [1 \ldots N]$ and send them to Sim, wait for the Sim to send back the proofs $(\delta_1, \ldots, \delta_N)$. Then register $(vk, x, (i, z_i, \delta_i))$ and send it to C for $i \in [N]$. **(Random Output)**

– **On** $(\textsf{\textbf{Pre-Verify}}, (vk, x, y, \pi))$ **from anyone**: If the entry (vk, x, y, π) is registered, then return 1, else return 0.

– **On** $(\textsf{\textbf{Inst-Verify}}, (vk, x, i, z_i, \delta_i))$ **from anyone**: If the entry $(vk, x, (i, z_i, \delta_i))$ is registered with the functionality, then return 1, else return 0. **(Independently Verifiable Output)**

Fig. 7: Ideal Functionality $\mathcal{F}_{i\text{-}\mathsf{DVRF}}$ for modeling distributed VRF with instantly verifiable output

– **DDH-based on Elliptic Curves [GRPV23].** The secret key is $sk \in \mathbb{Z}_q$ and the public verification key is $vk = g^{sk}$. The input is x. Compute $h = \mathsf{H}_1(x)$ and $\gamma = h^{sk}$. The output is $y = \mathsf{H}_2(\gamma^f)$ for a public parameter f. To compute the VRF proof, compute a proof of equal discrete logs for (g, vk) and (h, γ) as: sample $k \leftarrow \mathbb{Z}_q$, set $c = \mathsf{H}_3(g, h, vk, \gamma, g^k, h^k)$ and $s = k - c \cdot sk \bmod q$. Set y as the output and $\rho = (\gamma, c, s)$ as the proof. To verify the output check that 1) $y \stackrel{?}{=} \mathsf{H}_2(\gamma^f)$, and 2) compute $u = (vk)^c \cdot g^s$, $h = \mathsf{H}_1(x)$, $v = \gamma^c \cdot h^s$ and check $c \stackrel{?}{=} \mathsf{H}_3(g, h, vk, \gamma, u, v)$. To perform the output correctness check, the simulator programs H_1 s.t. it returns g^{r_i} for different queries $\mathsf{H}_1(q_i)$ made by the client. When the client submits the output y for input x, the simulator finds out γ by observing H_2 and finds r s.t. $\mathsf{H}_1(x) = g^r$. Then the simulator returns 1 if $(g, \mathsf{H}_1(x), vk, \gamma)$ forms a DDH tuple by checking $vk^r \stackrel{?}{=} \gamma$.

E Security Proof of Instarand

We prove that π_{IRand} implements $\mathcal{F}_{i\mathsf{VRF}}$ in the real-ideal world paradigm by proving Thm. 1.

Denote $\mathtt{Adv}_s^p$ and $\mathtt{Adv}_c^p$ as the advantages of an adversary in the pseudorandomness games of VRF_s and VRF_c respectively. Denote $\mathtt{Adv}_s^u$ and $\mathtt{Adv}_c^u$ as the advantages of an adversary in the uniqueness games of VRF_s and VRF_c respectively. Assuming adversary $\mathcal{A}$ makes at most q to the random oracle H in π_{IRand}, we concretely show that the advantage of $\mathcal{A}$, denoted as $\mathtt{Adv}$, is upper bounded as:

$$\mathtt{Adv} \leq q \cdot (\mathtt{Adv}_s^p + \mathtt{Adv}_c^p) + \mathtt{Adv}_s^u + \mathtt{Adv}_c^u$$

Proof. We prove that π_{IRand} implements $\mathcal{F}_{i\mathsf{VRF}}$ by proving Thm. 1 in the UC model [Can01]. We assume that there exists a PPT adversarial algorithm $\mathcal{A}$ that corrupts participating parties in the protocol execution. In the real-world execution of the protocol, $\mathcal{A}$ corrupts a party and interacts with the rest of the honest parties. At the end of the protocol execution, we denote its view as $\textsc{Real}_{\pi_{\mathsf{IRand}},\mathcal{A},\mathcal{E}}(1^\lambda)$. In the ideal world execution of the protocol, the honest parties provide their input to $\mathcal{F}_{i\mathsf{VRF}}$ and we provide a PPT simulator Sim that given access to the adversarial algorithm $\mathcal{A}$ and the functionality $\mathcal{F}_{i\mathsf{VRF}}$ produces the ideal world adversary view $\textsc{Ideal}_{\mathcal{F}_{\mathsf{iVRF}},\mathsf{Sim},\mathcal{E}}(1^\lambda)$. According to the real-ideal world paradigm, these two views should be indistinguishable.

We consider the following four exhaustive corruption cases for the corruption of the client and the server. For each of them, we construct our simulator algorithms and argue indistinguishability of real and ideal world views of the environment. We assume that the public verifier is always controlled by the adversary in all four cases.

1. Both server S and client C are honest. In this case, an adversary corrupts a public verifier who views the final output (z_i, δ_i) values. We argue that the z_i values will be random to an external verifier (who is not the server or the client). We provide the simulation algorithm in Fig. 8. The simulator generates a correct (vk_s, sk_s) on behalf of the server and registers it. For evaluation on input (x, vk_s) the simulator evaluates VRF_s on the input using sk_s. For Instant Output Generation the functionality samples random $(z_1, \ldots, z_N)$ values and sends it to the simulator for the simulated proofs. To construct proof ρ_i, the simulator evaluates (w_i, ρ_i) running the usual protocol steps and then programs the random H on input (i, w_i, x, y) s.t. it outputs z_i. The simulator aborts if the programming fails. To perform the pre-verification and instant verification requests by an external verifier the simulator invokes $\mathcal{F}_{i\mathsf{VRF}}$ on the request and returns the output of $\mathcal{F}_{i\mathsf{VRF}}$. We provide the simulator algorithm in Fig. 8.

Primitives. $\mathsf{VRF}_s, \mathsf{VRF}_c : (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$ are two verifiable random functions, $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ is a random oracle.

Parties. Simulator Sim, functionality $\mathcal{F}_{i\mathsf{VRF}}$.

Server Key Gen. The simulated server generates $(vk_s, sk_s) \leftarrow \mathsf{VRF}_s.\mathsf{Gen}(1^\lambda)$. Sim invokes $\mathcal{F}_{i\mathsf{VRF}}$ with input (S, vk_s) and posts vk_s to the bulletin board in the simulated protocol as the server verification key.

Client Input Gen. The honest client invokes $\mathcal{F}_{i\mathsf{VRF}}$ with input (x, vk_s) and this is forwarded to the simulator by $\mathcal{F}_{i\mathsf{VRF}}$.

Server VRF Evaluation. The simulated server computes $(y, \pi) := \mathsf{VRF}_s.\mathsf{Eval}(sk_s, x)$. It sends (x, y, π) to $\mathcal{F}_{i\mathsf{VRF}}$ and (y, π) to the simulated client.

Client VRF Verification. To verify input (vk_s, x, y, π), return the output of $\mathcal{F}_{i\mathsf{VRF}}$ on input $(\mathsf{Pre\text{-}Verify}, (vk_s, x, y, \pi))$.

The following algorithms are run multiple times for different sessions $i \in [1 \ldots N]$.
Instant Output Generation. The simulator obtains $(z_1, \ldots, z_N)$ from $\mathcal{F}_{i\mathsf{VRF}}$. The simulated client performs the following for $i \in [N]$:

$$(w_i, \rho_i) := \mathsf{VRF}_c.\mathsf{Eval}(sk_c, (x, y, i)).$$

The simulated client programs H s.t. $\mathsf{H}(i, w_i, x, y) := z_i$ and sets the proof as $\delta_i = (\pi, \rho_i, w_i, y)$. If the programming of H fails then the simulator aborts. Otherwise, Sim returns $(\delta_1, \ldots, \delta_N)$ to $\mathcal{F}_{i\mathsf{VRF}}$.

Pre-verification. To verify input (vk_s, x, y, π), return the output of $\mathcal{F}_{i\mathsf{VRF}}$ on input $(\mathsf{Pre\text{-}Verify}, (vk_s, x, y, \pi))$.

Instant Verification. To verify input $(vk_s, x, i, z_i, \delta_i)$, return the output of $\mathcal{F}_{i\mathsf{VRF}}$ on input $(\mathsf{Inst\text{-}Verify}, (vk_s, x, i, z_i, \delta_i))$.

Fig. 8: Simulator when both server S and client C are honest

An adversary $\mathcal{A}$ corrupting a public verifier can distinguish between the real and ideal world if the programming of H fails. In this case, the adversary has to guess the values of both y and w_i without querying x to $\mathcal{F}_{i\mathsf{VRF}}$ since $\mathcal{F}_{i\mathsf{VRF}}$

only allows unique queries. Such an adversary breaks pseudorandomness of both VRF_s and VRF_c in the random oracle model.
Indistinguishability Argument: We provide the formal hybrids and argue indistinguishability as follows:

- Hyb_0: Real-world execution of the protocol in Fig.4.
- Hyb_1: This is the same as Hyb_0, except the simulator aborts if the adversary makes any query of the form $\mathsf{H}(\_,\_,x,y)$, (here $\_$ denotes any value) where x is the simulated client input and $\mathsf{VRF}_s.\mathsf{Verify}(vk_s, x, (y, \pi))$ for π computed using the VRF_s evaluation.
  A distinguisher distinguishes between the two hybrids if it predicts y. The adversary for the pseudorandomness game observes the set of $(\_,\_,x,y)$ values and returns one of them randomly to the challenger of pseudorandomness game as the response on challenge x. If the distinguisher distinguishes between the two hybrids with advantage $\mathtt{Adv}^1_{0,1}$ and makes q queries to H, then the pseudorandomness adversary of VRF_s wins with probability $\mathtt{Adv}^p_s$ computed as:
  $$\frac{\mathtt{Adv}^1_{0,1}}{q} \leq \mathtt{Adv}^p_s$$
- Hyb_2: This is the same as Hyb_1, except the simulator aborts if the adversary makes any query of the form $\mathsf{H}(i, w_i, x, y)$ for $i \in [N]$ and prohibits the programming of the random oracle.
  A distinguisher between the two hybrids distinguishes if it makes a valid RO query containing (i, w_i) as the input for $i \in [N]$. In such a case, we construct an adversary for the pseudorandomness game of VRF_c. The adversary returns (i, w_i) as the output on input (x, y, i) and wins the game. If the distinguisher distinguishes between the two hybrids with advantage $\mathtt{Adv}^1_{1,2}$ and makes q queries to H, then the pseudorandomness adversary of VRF_c wins with probability $\mathtt{Adv}^p_c$ where:
  $$\frac{\mathtt{Adv}^1_{1,2}}{q} \leq \mathtt{Adv}^p_c$$
  We note that since both server and client are honest, replacing the pre-verification and instant verification steps in the protocol by invocations to $\mathcal{F}_{i\mathsf{VRF}}$ does not provide any additional advantage to the adversary and these changes are equivalent.
- Hyb_3: This is the same as Hyb_2, except the simulator performs the pre-verification step by invoking $\mathcal{F}_{i\mathsf{VRF}}$ on the pre-verification request instead of running the protocols steps of π_{IRand}.
  An adversary distinguishes between the two hybrids if it generates a pre-verification request on a different $(y', \pi') \neq (y, \pi)$ s.t. (y', π') verifies w.r.t. (vk_s, x). The request successfully verifies in Hyb_2 but fails to verify in Hyb_3 since one of them will not be registered with $\mathcal{F}_{i\mathsf{VRF}}$. In this case, we construct an adversary for breaking uniqueness of vk_s who returns $(vk_s, x, (y, \pi), (y', \pi'))$ as the answer to the challenger of the uniqueness game. If the distinguisher distinguishes between the two hybrids with an advantage $\mathtt{Adv}^1_{2,3}$, then the uniqueness adversary of VRF_s wins with probability $\mathtt{Adv}^u_s$ computed as:
  $$\mathtt{Adv}^1_{2,3} \leq \mathtt{Adv}^u_s.$$

- Hyb_4: This is the same as Hyb_3, except the simulator performs the instant verification step by invoking $\mathcal{F}_{i\mathsf{VRF}}$ on the instant verification request instead of running the protocols steps of π_{IRand}. This is the ideal world execution of the protocol.

  An adversary distinguishes between the two hybrids if it generates an instant verification request on two different requests containing $(w, \rho) \neq (w', \rho')$ s.t. both verify w.r.t. (vk_c, x, y, i). The request successfully verifies in Hyb_3 but fails to verify in Hyb_4 since one of them will not be registered with $\mathcal{F}_{i\mathsf{VRF}}$. In this case, we construct an adversary for breaking uniqueness of vk_c who returns $(vk_c, (x, y, i), (w, \rho), (w', \rho'))$ as the response to the challenger of the uniqueness game. If the distinguisher distinguishes between the two hybrids with advantage $\mathtt{Adv}^1_{3,4}$, then the uniqueness adversary of VRF_c wins with probability $\mathtt{Adv}^u_c$ where:

$$\mathtt{Adv}^1_{3,4} \leq \mathtt{Adv}^u_c.$$

An adversary distinguishes the real and ideal world with an advantage $\mathtt{Adv}^1$ upper bounded as:

$$\mathtt{Adv}^1 \leq q \cdot (\mathtt{Adv}^p_s + \mathtt{Adv}^p_c) + \mathtt{Adv}^u_s + \mathtt{Adv}^u_c.$$

2. Server S is corrupt and client C is honest. In this case, an adversary corrupts the server and a public verifier who views the final output (z_i, δ_i) values. We argue that the z_i values will be random to an external verifier even though y is generated by the corrupt server. We provide the simulation algorithm in Fig. 9. The simulator forwards the vk_s, generated by the corrupt server, to $\mathcal{F}_{i\mathsf{VRF}}$. For evaluation on input (x, vk_s) the simulator receives an evaluation request on input (x, vk_s) and forwards it to the server to obtain the output (y, π). The simulated client aborts the protocol if the verification of (y, π) fails else the simulator forwards this output to $\mathcal{F}_{i\mathsf{VRF}}$ and it gets registered. For Instant Output Generation the functionality samples random $(z_1, \ldots, z_N)$ values and sends it to the simulator for the simulated proofs. To construct proof ρ_i, the simulator evaluates (w_i, ρ_i) running the usual protocol steps and then programs the random H on input (i, w_i, x, y) s.t. it outputs z_i. The simulator aborts if the programming fails. To perform the pre-verification and instant verification requests by an external verifier the simulator invokes $\mathcal{F}_{i\mathsf{VRF}}$ on the request and returns the output of $\mathcal{F}_{i\mathsf{VRF}}$. We provide the simulator algorithm in Fig. 9.

An adversary $\mathcal{A}$ corrupting a public verifier can distinguish between the real and ideal world if the programming of H fails or if the pre-verification fails in the ideal world but the pre-verificaiton succeeds in the real world. In this case, the adversary either has to guess the values of w_i without querying or it breaks the uniqueness of VRF_s in the real world execution by producing two different (y, π) and (y', π') values that verify against (x, vk_s).

Indistinguishability Argument: We provide the formal hybrids and argue indistinguishability as follows:

- Hyb_0: Real-world execution of the protocol in Fig.4.

Primitives. $\mathsf{VRF}_s, \mathsf{VRF}_c : (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$ are two verifiable random functions, $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ is a random oracle.

Parties. Simulator Sim, functionality $\mathcal{F}_{i\mathsf{VRF}}$.

Server Key Gen. The corrupt server sends vk_s to everyone. Sim receives it and invokes $\mathcal{F}_{i\mathsf{VRF}}$ with input (S, vk_s).

Client Input Gen. The honest client invokes $\mathcal{F}_{i\mathsf{VRF}}$ with input (x, vk_s) and this is forwarded to the simulator by $\mathcal{F}_{i\mathsf{VRF}}$. The simulated client sends x to the corrupt server.

Server VRF Evaluation. The corrupt server sends (y, π) to the simulated client.

Client VRF Verification. The simulated client sends (x, y, π) to $\mathcal{F}_{i\mathsf{VRF}}$ if $\mathsf{VRF}_s.\mathsf{Verify}(vk_s, x, (y, \pi)) = 1$. Otherwise, the simulator sends $\perp$ to $\mathcal{F}_{i\mathsf{VRF}}$ and $\perp$ to the corrupt server.

The following algorithms are run multiple times for different sessions $i \in [1 \dots N]$.
Instant Output Generation. The simulator obtains $(z_1, \dots, z_N)$ from $\mathcal{F}_{i\mathsf{VRF}}$. The simulated client performs the following for $i \in [N]$:

$$(w_i, \rho_i) := \mathsf{VRF}_c.\mathsf{Eval}(sk_c, (x, y, i)).$$

The simulated client programs H s.t. $\mathsf{H}(i, w_i, x, y) := z_i$ and sets the proof as $\delta_i = (\pi, \rho_i, w_i, y)$. If the programming of H fails then the simulator aborts. Otherwise, Sim returns $(\delta_1, \dots, \delta_N)$ to $\mathcal{F}_{i\mathsf{VRF}}$.

Pre-verification. To verify input (vk_s, x, y, π), return the output of $\mathcal{F}_{i\mathsf{VRF}}$ on input $(\mathsf{Pre\text{-}Verify}, (vk_s, x, y, \pi))$.

Instant Verification. To verify input $(vk_s, x, i, z_i, \delta_i)$, return the output of $\mathcal{F}_{i\mathsf{VRF}}$ on input $(\mathsf{Inst\text{-}Verify}, (vk_s, x, i, z_i, \delta_i))$.

Fig. 9: Simulator when server S is corrupt and client C is honest

- Hyb_1: This is the same as Hyb_0, except the simulator aborts if the adversary makes any query of the form $\mathsf{H}(i, w_i, x, y)$ for $i \in [N]$ and prohibits the programming of the random oracle.
  A distinguisher between distinguishes the two hybrids if it makes a valid RO query containing (i, w_i) as the input for $i \in [N]$. In such a case, we construct an adversary for the pseudorandomness game of VRF_c. The adversary returns (i, w_i) as the output on input (x, y, i) and wins the game. If the distinguisher distinguishes between the two hybrids with an advantage $\mathtt{Adv}^2_{0,1}$ and makes q queries to H, then the pseudorandomness adversary of VRF_c wins with probability $\mathtt{Adv}^p_c$ where:
  $$\frac{\mathtt{Adv}^2_{0,1}}{q} \leq \mathtt{Adv}^p_c$$
- Hyb_2 : This is the same as Hyb_1, except the checks of pre-verification are performed by invoking $\mathcal{F}_{i\mathsf{VRF}}$ on the input request instead of running the protocols steps of π_{IRand}.
  An adversary distinguishes between the two hybrids if it generates a pre-verification request on a different $(y', \pi') \neq (y, \pi)$ s.t. (y', π') verifies w.r.t. (vk_s, x). The request successfully verifies in Hyb_1 but fails to verify in Hyb_2 since one of them will not be registered with $\mathcal{F}_{i\mathsf{VRF}}$. In this case, we construct an adversary for breaking uniqueness of vk_s who returns $(vk_s, x, (y, \pi), (y', \pi'))$ as the answer to the challenger of the uniqueness game. If the distinguisher distinguishes between the two hybrids with an advantage $\mathtt{Adv}^2_{1,2}$, then the uniqueness adversary of VRF_s wins with probability $\mathtt{Adv}^u_s$ computed as:
  $$\mathtt{Adv}^2_{1,2} \leq \mathtt{Adv}^u_s.$$
- Hyb_3: This is the same as Hyb_2, except the simulator performs the instant verification step by invoking $\mathcal{F}_{i\mathsf{VRF}}$ on the instant verification request instead of running the protocols steps of π_{IRand}. This is the ideal world execution of the protocol.
  An adversary distinguishes between the two hybrids if it generates an instant verification request on two different requests containing $(w, \rho) \neq (w', \rho')$ s.t. both verify w.r.t. (vk_c, x, y, i). The request successfully verifies in Hyb_2 but fails to verify in Hyb_3 since one of them will not be registered with $\mathcal{F}_{i\mathsf{VRF}}$. In this case, we construct an adversary for breaking uniqueness of vk_c who returns $(vk_c, (x, y, i), (w, \rho), (w', \rho'))$ as the response to the challenger of the uniqueness game. If the distinguisher distinguishes between the two hybrids with advantage $\mathtt{Adv}^2_{2,3}$, then the uniqueness adversary of VRF_c wins with probability $\mathtt{Adv}^u_c$ where:
  $$\mathtt{Adv}^2_{2,3} \leq \mathtt{Adv}^u_c.$$

An adversary distinguishes the real and ideal world with an advantage $\mathtt{Adv}^2$:

$$\mathtt{Adv}^2 \leq q \cdot \mathtt{Adv}^p_c + \mathtt{Adv}^u_s + \mathtt{Adv}^u_c.$$

3. Server S is honest and client C is corrupt. In this case, an adversary corrupts the client C and a public verifier who views the final output (z_i, δ_i)

values. We argue that even though the client is corrupt still the $z_i := \mathsf{H}(i, w_i, x, y)$ values will be random since y is randomly distributed. The z_i values are also unique since w_i will be unique as they are the output of VRF_c, which guarantees output uniqueness. We provide the simulation algorithm in Fig. 10. The simulator generates a correct (vk_s, sk_s) on behalf of the server and registers it. For evaluation on input (x, vk_s) the simulator evaluates VRF_s on the input using sk_s. Sim also observes the random oracle queries by client and Sim aborts if the client has queried H on input $(\_, \_, x', y')$ where $y' = \mathsf{VRF}_s.\mathsf{Eval}(sk_s, x')$. For Instant Output Generation, when the corrupt client queries $\mathsf{H}(i, w_i, x, y)$, the simulator observes it, verifies that w_i is the correct output and then invokes $\mathcal{F}_{i\mathsf{VRF}}$ to obtain the random output z_i. The simulator programs $\mathsf{H}(i, w_i, x, y) = z_i$ as the output. This guarantees that the output is unbiasable against a malicious client. Then the client generates the proof δ_i and returns it, which is forwarded to $\mathcal{F}_{i\mathsf{VRF}}$. To perform the pre-verification requests by an external verifier the simulator invokes $\mathcal{F}_{i\mathsf{VRF}}$ on the request and returns the output of $\mathcal{F}_{i\mathsf{VRF}}$. In the instant verification step, when one of the simulated parties obtains this output and proof for verification, the simulator checks y using the pre-verification stage. Then Sim checks that w_i is indeed the correct VRF_c output on (x, y, i) and $z_i := \mathsf{H}(i, w_i, x, y)$. Once these verification checks pass, Sim sets $\mathsf{Rand}_{\mathcal{Y}}(x, y, i) := z_i$ so that $\mathcal{F}_{i\mathsf{VRF}}$ obtains z_i when it queries $\mathsf{Rand}_{\mathcal{Y}}(x, y, i)$ in the Output Generation step of $\mathcal{F}_{i\mathsf{VRF}}$. Then, $\mathcal{F}_{i\mathsf{VRF}}$ invokes Sim with z_i and Sim returns δ_i as the proof to $\mathcal{F}_{i\mathsf{VRF}}$. This enables our Sim to correctly simulate the z_i s.t. it matches with the output of $\mathsf{Rand}_{\mathcal{Y}}(x, y, i)$ while ensuring z_i is random since it is the output of the random oracle. We provide the simulator algorithm in Fig. 10.

An adversary $\mathcal{A}$ corrupting the client and the public verifier can distinguish between the real and ideal world if it breaks the pseudorandomness of VRF_s by guessing the output y' of VRF_s on x' without querying the server, or if the corrupt client breaks the uniqueness of VRF_c producing two different w_i values for the same (x, y, i). The adversarial client can choose the output that favors it the most and then produce it as the output of the *Instant Output Generation* step.

Indistinguishability Argument: We provide the formal hybrids and argue indistinguishability as follows:

- Hyb_0: Real-world execution of the protocol in Fig.4.
- Hyb_1: This is the same as Hyb_0, except if the client has queried H on input $(\_, \_, x', y')$ where y' is computed as $y' = \mathsf{VRF}_s.\mathsf{Eval}(sk_s, x')$, and x' was not queried to the server then Sim aborts.
  A distinguisher between distinguishes the two hybrids if it makes a valid RO query containing $(\_, \_, x', y')$ where $y' = \mathsf{VRF}_s.\mathsf{Eval}(sk_s, x')$. The protocol continues in Hyb_0, whereas the simulator aborts in Hyb_1. In such a case, we construct an adversary for the pseudorandomness game of VRF_s. When the adversary makes such a query among the list of RO queries the adversary returns one of the queries randomly as the output to the challenger of the pseudorandomness game in VRF_s. If the distinguisher distinguishes between the two hybrids with an advantage $\mathtt{Adv}^3_{0,1}$ and makes q RO queries then the

Primitives. $\mathsf{VRF}_s, \mathsf{VRF}_c : (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$ are two verifiable random functions, $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ is a random oracle.

Parties. VRF server S, client C.

Server Key Gen. The simulated server S generates $(vk_s, sk_s) \leftarrow \mathsf{VRF}_s.\mathsf{Gen}(1^\lambda)$. Sim invokes $\mathcal{F}_{i\mathsf{VRF}}$ with input (S, vk_s) and sends vk_s to the corrupt parties in the simulated protocol as the server verification key.

Client Input Gen. The corrupt client sends x to the simulated server. If the client has queried H on input $(\_, \_, x', y')$ where $y' = \mathsf{VRF}_s.\mathsf{Eval}(sk_s, x')$ and x' was not queried to the server then Sim aborts. Otherwise, Sim invokes $\mathcal{F}_{i\mathsf{VRF}}$ with input (x, vk_s) on behalf of the corrupt client.

Server VRF Evaluation. When $\mathcal{F}_{i\mathsf{VRF}}$ forwards this request to Sim for server computation, the simulated server computes $(y, \pi) := \mathsf{VRF}_s.\mathsf{Eval}(sk_s, x)$. The simulated server sends (y, π) to the corrupt client. Sim returns (x, y, π) to $\mathcal{F}_{i\mathsf{VRF}}$.

Client VRF Verification. The client verifies the generation of y by checking that $\mathsf{VRF}_s.\mathsf{Verify}(vk_s, x, (y, \pi)) \stackrel{?}{=} 1$.

The following algorithms are run multiple times for different sessions $i \in [1 \dots N]$.
Instant Output Generation. When the corrupt client queries (i, w_i, x, y) to H for $i \in [N]$, the simulator checks that w_i is the correct evaluation of VRF_c on input (x, y, i) by using the output correctness property of VRF_c. Once it is verified, the simulator invokes $\mathcal{F}_{i\mathsf{VRF}}$ to obtain random outputs $(z_1, z_2, \dots, z_N)$. The simulator stores them locally and then programs $\mathsf{H}(i, w_i, x, y) = z_i$. The corrupt client obtains z_i as the output and computes the corresponding proof as $\delta_i = (\pi, \rho_i, w_i, y)$ for $i \in [N]$. Once the corrupt client returns δ_i as the proof, the simulator forwards it to $\mathcal{F}_{i\mathsf{VRF}}$.

Pre-verification. To verify input (vk_s, x, y, π), return the output of $\mathcal{F}_{i\mathsf{VRF}}$ on input $(\mathsf{Pre\text{-}Verify}, (vk_s, x, y, \pi))$.

Instant Verification. To verify input $(vk_s, x, i, z_i, \delta_i)$, if this request was previously made then return the output of $\mathcal{F}_{i\mathsf{VRF}}$ on input $(\mathsf{Inst\text{-}Verify}, (vk_s, x, i, z_i, \delta_i))$. Otherwise, Sim performs the following:

1. Sim performs the following checks:
   (a) The output of $\mathcal{F}_{i\mathsf{VRF}}$ on input $(\mathsf{Pre\text{-}Verify}, (vk_s, x, y, \pi))$ is 1,
   (b) $\mathsf{VRF}_c.\mathsf{Verify}(vk_c, (x, y, i), (w_i, \rho_i)) \stackrel{?}{=} 1$, and
   (c) $z_i \stackrel{?}{=} \mathsf{H}(i, w_i, x, y)$.
2. If any of the checks fail then return 0 to the party who invoked the *Instant Verification* command.
3. If all the above checks pass then Sim sets $\mathsf{Rand}_{\mathcal{Y}}(x, y, i) := z_i$ and $\mathcal{F}_{i\mathsf{VRF}}$ obtains z_i when it queries $\mathsf{Rand}_{\mathcal{Y}}(x, y, i)$. Then, $\mathcal{F}_{i\mathsf{VRF}}$ invokes Sim with z_i and Sim returns δ_i as the proof to $\mathcal{F}_{i\mathsf{VRF}}$. Return 1 to the party who invoked the *Instant Verification* command.

Fig. 10: Simulator when server S is honest and client C is corrupt

pseudorandomness adversary of VRF_s wins with probability $\mathtt{Adv}_s^p$ computed as:

$$\frac{\mathtt{Adv}_{0,1}^3}{q} \leq \mathtt{Adv}_s^p$$

– Hyb_2 : This is the same as Hyb_1, except the checks of pre-verification are performed by invoking $\mathcal{F}_{i\mathsf{VRF}}$ on the input request instead of running the protocols steps of π_{IRand}.
  An adversary distinguishes between the two hybrids if it generates a pre-verification request on a different $(y', \pi') \neq (y, \pi)$ s.t. (y', π') verifies w.r.t. (vk_s, x). The request successfully verifies in Hyb_1 but fails to verify in Hyb_2 since one of them will not be registered with $\mathcal{F}_{i\mathsf{VRF}}$. In this case, we construct an adversary for breaking uniqueness of vk_s who returns $(vk_s, x, (y, \pi), (y', \pi'))$ as the answer to the challenger of the uniqueness game. If the distinguisher distinguishes between the two hybrids with an advantage $\mathtt{Adv}_{1,2}^3$, then the uniqueness adversary of VRF_s wins with probability $\mathtt{Adv}_s^u$ computed as:

$$\mathtt{Adv}_{1,2}^3 \leq \mathtt{Adv}_s^u.$$

– Hyb_3: This is the same as Hyb_2, except the simulator performs the instant verification step by following the simulation steps in Fig.10 instead of running the protocols steps of π_{IRand}. This is the ideal world execution of the protocol. An adversary distinguishes between the two hybrids if it generates an instant verification request on two different requests containing $(w, \rho) \neq (w', \rho')$ s.t. both verify w.r.t. (vk_c, x, y, i). The request successfully verifies in Hyb_2 but fails to verify in Hyb_3 since one of them will not be registered with $\mathcal{F}_{i\mathsf{VRF}}$. In this case, we construct an adversary for breaking uniqueness of vk_c who returns $(vk_c, (x, y, i), (w, \rho), (w', \rho'))$ as the response to the challenger of the uniqueness game. If the distinguisher distinguishes between the two hybrids with advantage $\mathtt{Adv}_{2,3}^3$, then the uniqueness adversary of VRF_c wins with probability $\mathtt{Adv}_c^u$ where:

$$\mathtt{Adv}_{2,3}^3 \leq \mathtt{Adv}_c^u.$$

The uniqueness of VRF_c ensures that the output z_i is still uniformly distributed since it is the output of a random oracle queried on y and w_i. The value y is pseudorandom and remains hidden from a client until it queries the server with input x. And once x is queried the value w_i gets fixed due to the uniqueness of VRF_s and VRF_c. Hence, an adversary distinguishes the real and ideal world with advantage $\mathtt{Adv}^3$ where:

$$\mathtt{Adv}^3 \leq q \cdot \mathtt{Adv}_s^p + \mathtt{Adv}_s^u + \mathtt{Adv}_c^u.$$

4. Both Server S and client C are corrupt. In this case the adversary corrupts the server and the client. Pseudorandomness of the output is not guaranteed but we guarantee the uniqueness of the output on input (x, i) and server key vk_s. The first time when these steps are performed, the simulator checks the output by running the protocol steps and then registers them by calling $\mathcal{F}_{i\mathsf{VRF}}$. The next time, when the same pre-verification or instant verification request is made on the same input and output pair, the simulator invokes the pre-verification and instant verification steps of $\mathcal{F}_{i\mathsf{VRF}}$ to ensure only the registered output verifies on the input, otherwise it is rejected. We present the formal simulator algorithm in Fig. 11.

Primitives. $\mathsf{VRF}_s, \mathsf{VRF}_c : (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$ are two verifiable random functions, $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ is a random oracle.

Parties. VRF server S, client C.

Server Key Gen. The corrupt server sends vk_s to everyone. Sim receives it and invokes $\mathcal{F}_{i\mathsf{VRF}}$ with input (S, vk_s).

Client Input Gen. Performs its own adversarial algorithm.

Server VRF Evaluation. Performs its own adversarial algorithm.

Client VRF Verification. Performs its own adversarial algorithm.

The following algorithms are run multiple times for different sessions $i \in [1 \ldots N]$.
Instant Output Generation. Performs its own adversarial algorithm.

Pre-verification. To verify input $(vk_s, x, i, z_i, \delta_i)$, if this request was previously made then return the output of $\mathcal{F}_{i\mathsf{VRF}}$ on input $(\mathsf{Pre\text{-}Verify}, (vk_s, x, y, \pi))$. Otherwise, Sim performs the following:

1. If $\mathsf{VRF}_s.\mathsf{Verify}(vk_s, x, y, \pi) = 0$: Return 0 to the party who invoked the *Instant Verification* command.
2. If $\mathsf{VRF}_s.\mathsf{Verify}(vk_s, x, y, \pi) = 1$: Then Sim invokes $\mathcal{F}_{i\mathsf{VRF}}$ with input (x, vk_s) on behalf of corrupt client. When $\mathcal{F}_{i\mathsf{VRF}}$ forwards the same request to Sim, then Sim return (x, y, π) to $\mathcal{F}_{i\mathsf{VRF}}$. Return 1 to the party who invoked the *Pre-verification* command.

Instant Verification. To verify input $(vk_s, x, i, z_i, \delta_i)$, if this request was previously made then return the output of $\mathcal{F}_{i\mathsf{VRF}}$ on input $(\mathsf{Inst\text{-}Verify}, (vk_s, x, i, z_i, \delta_i))$. Otherwise, Sim performs the following:

1. Sim performs the following checks:
   (a) The output of *Pre-verification* on input $(\mathsf{Pre\text{-}Verify}, (vk_s, x, y, \pi))$ is 1,
   (b) $\mathsf{VRF}_c.\mathsf{Verify}(vk_c, (x, y, i), (w_i, \rho_i)) \stackrel{?}{=} 1$, and
   (c) $z_i \stackrel{?}{=} \mathsf{H}(i, w_i, x, y)$.
2. If any of the checks fail then return 0 to the party who invoked the *Instant Verification* command.
3. If all the above checks pass then Sim sets $\mathsf{Rand}_{\mathcal{Y}}(x, y, i) := z_i$ and $\mathcal{F}_{i\mathsf{VRF}}$ obtains z_i when it queries $\mathsf{Rand}_{\mathcal{Y}}(x, y, i)$. Then, $\mathcal{F}_{i\mathsf{VRF}}$ invokes Sim with z_i and Sim returns δ_i as the proof to $\mathcal{F}_{i\mathsf{VRF}}$. Return 1 to the party who invoked the *Instant Verification* command.

Fig. 11: Simulator when both server S and client C are corrupt

An adversary $\mathcal{A}$ distinguishes between the real and ideal world if it breaks the uniqueness of VRF_s or VRF_c computing two different outputs for the same input that verifies in the real world but fails to verify in the ideal world since the input-output pair is already registered by $\mathcal{F}_{i\mathsf{VRF}}$ and for the same input no other output will verify. If such an attack is possible the adversarial client will choose the output that favors it the most (for the same input) and then produce it as the output of the *Instant Output Generation* step.

Indistinguishability Argument: We provide the formal hybrids and argue indistinguishability as follows:

- Hyb_0: Real-world execution of the protocol in Fig.4.
- Hyb_1 : This is the same as Hyb_0, except the simulator performs the pre-verification step by following the simulation steps in Fig.11 instead of running the protocols steps of π_{IRand}.

  An adversary distinguishes between the two hybrids if it generates a pre-verification request on a different $(y', \pi') \neq (y, \pi)$ s.t. (y', π') verifies w.r.t. (vk_s, x). The request successfully verifies in Hyb_0 but fails to verify in Hyb_1 since one of them will not be registered with $\mathcal{F}_{i\mathsf{VRF}}$. In this case, we construct an adversary for breaking uniqueness of vk_s who returns $(vk_s, x, (y, \pi), (y', \pi'))$ as the answer to the challenger of the uniqueness game. If the distinguisher distinguishes between the two hybrids with an advantage $\mathtt{Adv}^3_{1,2}$, then the uniqueness adversary of VRF_s wins with probability $\mathtt{Adv}^u_s$ where:

  $$\mathtt{Adv}^3_{1,2} \leq \mathtt{Adv}^u_s.$$

- Hyb_3: This is the same as Hyb_2, except the simulator performs the instant verification step by following the simulation steps in Fig.11 instead of running the protocols steps of π_{IRand}. This is the ideal world execution of the protocol.

  An adversary distinguishes between the two hybrids if it generates an instant verification request on two different requests containing $(w, \rho) \neq (w', \rho')$ s.t. both verify w.r.t. (vk_c, x, y, i). The request successfully verifies in Hyb_2 but fails to verify in Hyb_3 since one of them will not be registered with $\mathcal{F}_{i\mathsf{VRF}}$. In this case, we construct an adversary for breaking uniqueness of vk_c who returns $(vk_c, (x, y, i), (w, \rho), (w', \rho'))$ as the response to the challenger of the uniqueness game. If the distinguisher distinguishes between the two hybrids with advantage $\mathtt{Adv}^4_{2,3}$, then the uniqueness adversary of VRF_c wins with probability $\mathtt{Adv}^u_c$ where:

  $$\mathtt{Adv}^4_{2,3} \leq \mathtt{Adv}^u_c.$$

  Hence, an adversary distinguishes the real and ideal world with an advantage $\mathtt{Adv}^4$:

  $$\mathtt{Adv}^4 \leq \mathtt{Adv}^u_s + \mathtt{Adv}^u_c.$$

An adversary $\mathcal{A}$ corrupting server and/or client and/or verifier has an advantage $\mathtt{Adv}$ upper bounded as follows:

$$\begin{aligned}\mathtt{Adv} &\coloneqq \max(\mathtt{Adv}^1, \mathtt{Adv}^2, \mathtt{Adv}^3, \mathtt{Adv}^4)\\ &\leq \max(q \cdot (\mathtt{Adv}_s^p + \mathtt{Adv}_c^p) + \mathtt{Adv}_s^u + \mathtt{Adv}_c^u,\\ &\quad q \cdot \mathtt{Adv}_c^p + \mathtt{Adv}_s^u + \mathtt{Adv}_c^u,\\ &\quad q \cdot \mathtt{Adv}_s^p + \mathtt{Adv}_s^u + \mathtt{Adv}_c^u, \mathtt{Adv}_s^u + \mathtt{Adv}_c^u)\\ &\coloneqq q \cdot (\mathtt{Adv}_s^p + \mathtt{Adv}_c^p) + \mathtt{Adv}_s^u + \mathtt{Adv}_c^u\end{aligned}$$

where the $\mathcal{A}$ makes at most q queries to the random oracle.

F Gas Cost Estimation of FlexiRand

Here, we present our gas cost estimations for FlexiRand. We use the same cost estimates as in Section 6.3.

In FlexiRand, the client initially submits an input to obtain a formatted input that consists of a unique request identifier. This initial request transaction cost the same $54k$ gas as the request for DDH-VRF and GLOW-DVRF. Then, the client blinds the formatted input and submits a proof of correct blinding using a Schnorr proof. This cost turns out to be $78k$ gas. This transaction consists of validating the proof, ensuring that no blinding has yet been submitted for x, and storing the blinded input on-chain. The cost is dominated by the $45k$ gas required to store the blinded input (and input) on-chain, $21k$ for the transaction, and additionally, the smart contract has to verify the Schnorr proof. Next, the servers evaluate the BLS-based VRF on the blinded input to generate the blinded output. The smart contract is run on this blinded output to fulfill the request. This step costs $174k$ gas. This transaction consists of ensuring that the request has not yet been fulfilled, and validating the blinded output w.r.t. the blinded input via a pairing check. The cost is dominated by $80k$ gas for the pairing check, $21k$ for the transaction, and $45k$ gas required to store the blinded output (and blinded input) on-chain. Note that, the blinded input is already in the BN254 curve and so FlexiRand avoids spending $62k$ to hash the input to the curve. Hence, the preprocessing phase of FlexiRand takes $306k$ gas.

In the online phase, when the client unblinds the output, the smart contract hashes the input and then performs a pairing check on the unblinded output and the input. This step costs $201k$ gas. It is mainly dominated by $21k$ gas to register the transaction, $62k$ gas to hash to the BN254 curve, and $80k$ gas to perform a pairing check. However, since FlexiRand does not support instant output generation, the preprocessing and online gas costs would scale with the number of outputs being generated.

FlexiRand+ZK-SNARK Approach. In this approach the FlexiRand preprocessing step is run only once, even for multiple extended outputs. Additionally, the preprocessing step consists of computing a hash of private output y as $h := \mathsf{H}(y)$ and giving a proof of correct computation of it. This incurs an additional gas cost of $300k$ of storing the hash value on-chain and verifying it. Later in the online phase, the Plonk proof proves that the extended output $z_i := \mathsf{H}(y, i)$, where $h := \mathsf{H}(y)$, is obtained by hashing the private output y committed inside h. This requires proving the correct computation of two poseidon hashes inside the plonk proof. This takes $80ms$ to compute the proof. And verifying this proof takes around $4.5ms$ and $300k$ gas on-chain.