VRaaS: Verifiable Randomness as a Service on Blockchains

Jacob Gorman
Supra Research
jacobgorman613@gmail.com

Lucjan Hanzlik
CISPA Helmholtz Center for Information Security
hanzlik@cispa.de

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

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

Pratyay Mukherjee
Supra Research
pratyay85@gmail.com

Pratik Sarkar
Supra Research
iampratiksarkar@gmail.com

Sri AravindaKrishnan Thyagarajan
University of Sydney
t.srikrishnan@gmail.com

***Abstract*—Web3 applications, such as on-chain games, NFT minting, and leader elections necessitate access to unbiased, unpredictable, and publicly verifiable randomness. Despite its broad use cases and huge demand, there is a notable absence of comprehensive treatments of on-chain verifiable randomness services. To bridge this, we offer an extensive formal analysis of on-chain verifiable randomness services.**

We present the *first* formalization of on-chain verifiable randomness in the blockchain setting by introducing the notion of Verifiable Randomness as a Service (VRaaS). We formally define VRaaS using an ideal functionality $\mathcal{F}_{\mathsf{VRaaS}}$ in the Universal Composability model. Our definition not only captures the core features of randomness services, such as unbiasability, unpredictability, and public verifiability, but also accounts for many other crucial nuances pertaining to different entities involved, such as smart contracts.

Within our framework we study a generic design of *Verifiable Random Function (VRF)-based* randomness service – where the randomness requester provides an input on which the randomness is evaluated as VRF output. We show that it does satisfy our formal VRaaS definition. Furthermore, we show that the generic protocol captures many real-world randomness services like Chainlink VRF and Supra dVRF.

Moreover, we investigate the minimalism of the framework. Towards that, first we show that, the two transactions in-built in our framework are actually *necessary* for any randomness service to support the essential qualities. We also discover *practical vulnerabilities* in other designs such as Algorand beacon, Pyth VRF and Band VRF, captured within our framework.

I. Introduction

Access to verifiable, secure randomness is increasingly critical in the blockchain domain, providing the foundation for liveness and safety guarantees in distributed systems [fil]. This necessity extends to various Web3 applications [Alg, Supa, Chac], including online gaming, NFTs, finance, supply chain management, and many more. The rapidly expanding Web3 [a16a, a16b] ecosystems have exploded the need for randomness services. E.g., Chainlink VRF [Chae] which has served 17.7 million randomness requests [Chad] until August 2023, and Supra dVRF [Supb] has served 410K+ requests in July-September 2023 [Supd].

The randomness offered by these services is not only unpredictable but also bias-resistant, ensuring fairness among participants. Moreover, within the blockchain domain, achieving public verifiability of randomness is paramount. This ensures that participants, third parties, and auditors meticulously examine the legitimacy of the randomness generation process with conviction without requiring them to trust anyone, even in the future. Consequently, blockchain solutions often turn to on-chain verifiable random function services like Chainlink [Chae], Supra dVRF [Supb], or random-beacon services such as Drand [DRa]. These services are integrated into smart contract-enabled blockchains, allowing the contracts to access verifiable randomness for a fee.

However, we observe that current service architectures follow specific flows [Chae, Bana, Supb, Pyt] and interactions, which are not yet rigorously studied – this leaves significant scope for the inadvertent misuse of randomness, particularly while trying to optimize for low latency or gas cost (e.g., in blockchain Ethereum ecosystem). Such misuse may occur at the protocol-flow level without hurting the security of the underlying cryptographic primitives. This paper aims to address this significant gap between the theory and practice of on-chain verifiable randomness by formalizing the flows and analyzing the services.

To better understand the gap, let us look at existing randomness offerings. A typical randomness service (e.g. [Chaa, Supc]) constitutes a committee producing verifiable random function (VRF) outputs and uses a smart contract for bookkeeping. Upon receiving a randomness request from a requester, the smart contract computes a query input from the request and forwards

the input to the VRF committee. The VRF committee generates a *publicly verifiable proof of correctness* of the generated randomness. The smart contract verifies the proof and then returns the output to the requester via a callback[1] function. The smart contract also records the payment and ensures the atomicity of the service offering against the payable fee. While appearing simplistic from a high-level, the details are much more intricate – as illustrated with concrete examples next:

- Consider the vulnerabilities within systems where the service does not verify the *uniqueness* of the query input, especially when the output influences games like lotteries. This results in the same output for different randomness requests with the same query input value. An attacker, aiming to exploit the situation, might engage in front-running[2] the game by examining multiple requests and strategically using a favorable random value.
- Designs such as PythVRF [Pyt] or FlexiRand [KMMM23] preemptively provide the output to the requester before publishing it on-chain. The requester possesses a secret value that is required for the output computation. The requester has the option to either continue the protocol execution by providing the secret value to the smart contract or abort the protocol. This lets a malicious requester selectively abort by observing the output and discontinue the protocol if the output is unfavorable.
- Some services, e.g. DIA, Algorand [DIA, ST22] etc. crucially rely on "*properly implemented requester smart contracts*" to handle edge-cases. For example, the protocol from [ST22] relies on a periodic beacon to generate randomness, and the requester smart contract "*should*" handle situations when the beacon is unavailable or the beacon is discontinued. This incurs much higher responsibilities onto the smart contract designers plus requiring each contract to be audited carefully too.
- Additionally, efforts to optimize gas costs, as discussed in Kate et al. [KMMM23], may lead to a requester obtaining a private random seed (that is verifiable) from an output-private randomness service and then reuse the seed in a pseudorandom generator to generate multiple pseudorandom values across applications. However, given the seed, all these values are predictable and need to be verified together. Such nuances were not formally articulated in [KMMM23].

Given the wide plethora of attack vectors and venues and the intricacies of various design choices within randomness services, we delve into the formal analysis of such randomness services.

A. Our Contributions

We present our contributions in detail as follows:

A formal framework for VRaaS. We introduce a formal framework for "*Verifiable Randomness as a Service*" (VRaaS) that rigorously captures the required properties from verifiable randomness on a blockchain. We define VRaaS via an ideal functionality $\mathcal{F}_{\mathsf{VRaaS}}$ that formally captures the following properties – the output is (indistinguishable from) random, unique (even for the same requester inputs), unforgeable (i.e., unauthorized users cannot generate the output), and the randomness is generated and verified on-chain (i.e. the generated randomness is verified by a blockchain network). Our functionality is in the Universal Composability (UC) model [Can01], allowing seamless composition in Web3 applications.

VRaaS protocols based on VRF. We analyze the folklore VRF based generic protocol – the randomness requester provides an input to the service on which the randomness is evaluated as VRF output by a VRF committee using a keyed VRF computation. We show that it realizes the $\mathcal{F}_{\mathsf{VRaaS}}$ functionality formally. This protocol encompasses the most widely used randomness-service protocols, such as Chainlink [Chae] showcasing the wide applicability of our proposed ideal functionality.

Necessity of two transactions[3]. Our functionality has two on-chain transactions built into it. One may wonder whether it is possible to achieve similar properties using one or less transaction, as the on-chain transactions are expensive. We show that, it is, in fact, *necessary* to have two on-chain transactions to satisfy crucial properties of the randomness. A randomness requester posts a request on the blockchain via an on-chain transaction. This request is read; a verifiable randomness output is generated in response, and finally, the output is posted on the blockchain via another on-chain transaction. The first transaction is necessary for authenticating and recording a request on-chain, and removing this transaction would allow a malicious requester to launch multiple requests (that cannot be linked back) and then confirm the second transaction of whose output is more favorable, essentially biasing the output. Next, the second transaction is necessary to ensure that the output is available on-chain. Removing this transaction (and replacing it with a direct message to the requester) enables a malicious requester to deny the output if it is unfavorable and keep sampling until a favorable one is obtained.

[1]A callback function [AGR+20] allows a caller to delegate the execution back to the caller. The requester receives the output randomness via the callback function once the randomness request is fulfilled.

[2]In a front-running attack, an adversary creates a special transaction based on pending transactions. It manipulates the transactions' ordering to execute their transaction first.

[3]These are transactions/activities that are verified by blockchain miners or validators, and once confirmed, they are permanently recorded on the blockchain.

Vulnerabilities in existing services. In our pursuit of analyzing existing randomness services, we also encounter shortcomings in certain protocols that lead to natural vulnerabilities. These include crucial dependence (Algorand [ST22] and Band [Bana]) on the requester smart contract designs to guarantee output uniqueness for each request, reliance (PythVRF [Pyt]) on honest requester behaviors to complete the protocol execution, and incorrect usage (DIA xRandom [DIA]) of the VRaaS output in Web3 applications.

A new ledger functionality. Additionally, as part of our building blocks, we also propose a "*simplified*" ideal functionality $\mathcal{F}_{\mathsf{LED}}$ to model ledgers that support smart contracts. $\mathcal{F}_{\mathsf{LED}}$ extends the Gearbox ledger functionality [DMM+22] to enable running smart contracts since the original Gearbox functionality did not support smart contracts. In our VRaaS protocols, we formally capture blockchains and smart contracts using $\mathcal{F}_{\mathsf{LED}}$, demonstrating its application in blockchain-based protocols. This functionality may be useful in future analysis of other blockchain-based protocols.

Note. Our VRaaS framework is general enough to capture different types of randomness services and not just the VRF based approach. E.g. in a follow-up work, we will prove that randomness services based on beacons (like DRand) can be shown to implement our ideal functionality and is a VRaaS.

Responsible Disclosure. We communicated the vulnerabilities in Algorand beacon, DIA xRandom, and Band VRF to their respective teams in January 2024 and we had offered them at least four months to work on solutions/mitigations before making the vulnerabilities public. The DIA and Algorand teams had responded to resolve the issues. We will keep the Band team informed about our progress with this paper.

Roadmap. We present the necessary preliminaries in Section II. Then, we present the blockchain functionality $\mathcal{F}_{\mathsf{LED}}$ in Section III. Next, we propose our VRaaS functionality in the $\mathcal{F}_{\mathsf{LED}}$ model in Section IV. We present our VRF-based VRaaS protocol in Section V. We present the vulnerabilities in existing protocols in Section. VII. Finally, we conclude by proving in Section VIII that two transactions are necessary for a VRaaS protocol.

B. Related Works

Harmony [Har] offers an on-chain verifiable randomness service protocol using distributed BLS-based VRF. It uses a verifiable delay function to prevent a malicious party in the server committee from determining whether to abort or continue after determining the output. Boba Network [Bob] constructs a publicly verifiable distributed randomness service from a distributed beacon implemented from a distributed VRF. Recently, Kate et al. [KMMM23] introduced output-private VRF called FlexiRand, which produces secret yet verifiable pseudo-random values based on a blinded variant of the BLS signature [BLS01]. These protocols provide verifiable output and can be used to instantiate our VRF-based protocol.

Aptos Roll [Apt] is a novel on-chain randomness method introduced recently, providing instantaneous randomness within the Aptos blockchain. Aptos validators produce a beacon value for each block based on block content and a secret-shared key in this approach. When a client requests randomness in a transaction, validators/executors use the beacon value from the same block to generate randomness instantly by hashing it with client or smart contract data. This method is specific to Aptos' blockchain (or similar) design. In particular, their procedure crucially relies on the fact that block ordering takes place *before* the transaction execution. It does not apply to many prominent blockchains such as Ethereum, where block ordering and execution are not separated. In contrast, we focus on randomness services that utilize blockchain as an append-only ledger without depending on specific design features and only rely on the smart contract capabilities of the blockchain.

Another line of work considers verifiable randomness from verifiable delayed function (VDF). Chia [Chi] implemented a beacon using repeated-squaring VDFs in class groups. VeeDo [Vee] implemented a VDF-based beacon using SNARK-based VDFs. Cornucopia [CCB23] proposed accumulator-based VDFs. CRAFT [BDD+23] proposed a UC modeling of Time-Lock Puzzles and VDFs. [BBBF18, LW15] construct beacons from VDFs, proving them to be UC-secure by CRAFT. Given a VDF, they construct publicly verifiable beacons via time-lock puzzles. Perhaps the most popular design in this line of work is the Ethereum beacon chain protocol [Edg], which is a randomness generation protocol to elect the block proposer using a single secret leader election [BEHG20]. The high-level idea is that Ethereum validators publicly run a VDF for round i on the output of round $i-1$. The output of i is obtained by running the RANDAO protocol on the VDF output and the output of round $i-1$. Every validator node eventually generates the output due to the public VDF evaluation and the delayed property of the VDF ensures that adversarial validators cannot precompute the output before the honest validators. However, the VDF-based protocols are not a VRaaS since they act as beacons (like DRand beacon) that produce a single random value after a fixed time delay. To enable a client to use the randomness on-chain (as a VRaaS), it needs to request randomness via a request transaction, then the appropriate beacon value is used to generate the client's output and this output is appended on-chain via the fulfillment transaction to allow usage in Web3 applications. In a follow-up work (attached

as supplementary material) we show that randomness beacon-based protocols (like the ones based on DRand beacon) are VRaaS protocols. Extending the result to show that VDF beacon-based randomness services are also VRaaS is left as an interesting future work.

II. Preliminaries

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

Notation. We use $\mathbb{N}$ to denote the set of positive integers, $\mathbb{Z}$ to denote the set of all integers, and $[n]$ to denote the set $\{1, 2, \ldots, n\}$ (for $n \in \mathbb{N}$). We denote the 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. We use $x := \mathsf{val}$ 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 $\mathsf{R}(x) \to y$ 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 say a problem is computationally hard when given a problem instance, generated using the security parameter λ, for any probabilistic algorithm $\mathcal{A}$ that runs in $O(\mathsf{poly}(\lambda))$ time, the probability that $\mathcal{A}$ can solve the given problem instance is upper bounded by $\mathsf{negl}(\lambda)$, where $\mathsf{negl}()$ is a negligible function in security parameter λ. For an algorithm A, we denote $A(x; y)$ as running the algorithm on input x and public parameter y where y might be omitted for notational simplicity. We denote an empty string as ε. A tuple of the form $(a, \_)$ denotes the case where the first element is a and the second element could be any character/string/ε. An algorithm A having oracle access to an algorithm $\mathcal{O}$ is denoted as $\mathsf{A}^{\mathcal{O}}$.

$\mathsf{Rand}(u; w)$. The Rand function is defined over output distribution $\{0,1\}^w$ and input $u \in \{0,1\}^*$ as follows, where T is initially an empty list:

$$\begin{aligned} \mathsf{Rand}(u; w) &:= \text{If } \exists (u, r) \in T, \text{ return } r. \\ &:= \text{Else, sample } r \leftarrow \{0,1\}^w, \\ &\quad\; T := T \cup (u, r), \text{ and return } r. \end{aligned}$$

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 recall the UC model un Appendix. I. We assume the existence of a *default authenticated channel* in the real world between any two parties. This significantly simplifies our definitions and can be removed using an ideal authenticated channel functionality [Can04].

Simulatable Verifiable Random Function. We recall the notion of simulatable VRF from the work of [CL07] and modify it to allow simulation using a programmable random oracle H. It consists of a tuple of three algorithms and they have oracle access to the oracle H:

- $\mathsf{Gen}^{\mathsf{H}}(1^\lambda) \to (vk, sk)$: On input λ it outputs a public verification key vk and secret key sk.
- $\mathsf{Eval}^{\mathsf{H}}(sk, x) \to (y, \pi)$: On input sk and input $x \in \{0,1\}^\lambda$ it outputs a random string $y \in \{0,1\}^\lambda$ and a proof π.
- $\mathsf{Verify}^{\mathsf{H}}(vk, x, (y, \pi)) \to b \in \{0,1\}$: It outputs a bit denoting whether proof verified or not.

A simulatable VRF ensures 1) *uniqueness* - for every x there exists a single (y, π), and 2) *simulatability* - given an output $y \leftarrow \mathsf{Rand}(x; \lambda)$ on a previously unqueried input x the corresponding proof π can be simulated. The proof is obtained as $\pi \leftarrow \mathsf{SimProve}^{\mathsf{H}}(sk, y, x)$, where the simulator programs the random oracle H. The real world (vk, x, y, π) is indistinguishable from a simulated (vk, x, y, π).

We need simulation-based security instead of game-based pseudorandomness since we use VRF to achieve simulation-secure randomness service where the random output will be randomly sampled and the proof will be simulated. We also demonstrate that the most widely known VRF protocols - BLS/GLOW-VRF [GLOW21], RSA-based [PWH+17], and the Elliptic-Curve based [PWH+17] VRFs are simulatable by modeling the hash function used in the protocols as a programmable random oracle. We refer to Appendix A for details.

III. Smart Contract Compatible Ledger Functionality

Gearbox [DMM+22] provides a simple easy-to-use timed ledger functionality for submitting messages on a global ledger; however, this functionality does not support smart contracts. On the other hand, the functionality in [GRR+21] considers smart contracts; however, it is too generalized as it captures both public and private ledgers. There are other ledger functionalities [BMTZ17] that are Bitcoin-specific and do not support smart contracts. As we aim for a simpler variant that only captures the commonly employed public ledgers, we consider a simplification of the ideal functionality of [GRR+21] and merge that with the ledger functionality of [DMM+22]. This allows us to strengthen the simple timed ledger functionality of [DMM+22], allowing it to deploy and call smart contracts. We refer to Appendix F for a comparison among different ledger functionalities.

We present our ledger functionality $\mathcal{F}_{\mathsf{LED}}$ in Fig. 1. $\mathcal{F}_{\mathsf{LED}}$ is initialized for a session sid with smart contract functions SC and an empty global ledger $\mathsf{LOG}^{\mathsf{sid}}$. The

Parameters. Hash function H, a smart contract code-checker $\mathsf{Checker}$ algorithm, delay Δ that is unknown to all the parties but known to adversary $\mathcal{A}_{\mathsf{LED}}$ and a function $\mathsf{predict\text{-}time}(\cdot)$ to predict real-world time.

Party-Registration. Set of registered parties is denoted as $\mathcal{P}$. Upon activation by a new party P_i, register P_i as $\mathcal{P} = \mathcal{P} \cup \{P\}$ and initialize $\mathsf{LOG}_i^{\mathsf{sid}} = \emptyset$. If P_i is the first honest party registered on $\mathcal{F}_{\mathsf{LED}}$ then send $(\text{REGISTER}, \mathsf{sid}_C)$ to $\mathcal{F}_{\mathsf{clock}}$.

Corruption. When adversary $\mathcal{A}_{\mathsf{LED}}$ corrupts a registered party P_i mark it as corrupt. If there is no more honest registered parties then send $(\text{DE-REGISTER}, \mathsf{sid}_C)$ to $\mathcal{F}_{\mathsf{clock}}$.

Clock-read-update: $\mathcal{F}_{\mathsf{LED}}$ maintains a time variable t_L that is kept in-sync with clock-time.

- Upon any activation by a party or $\mathcal{A}_{\mathsf{LED}}$, $\mathcal{F}_{\mathsf{LED}}$ first sends $(\text{CLOCK-READ}, \mathsf{sid}_C)$ to $\mathcal{F}_{\mathsf{clock}}$ (Fig. 9) to receive the answer $(\text{CLOCK-READ}, \mathsf{sid}_C, t)$ and sets $t_L := t$ and then proceeds with the remaining actions.
- When an honest party P_i calls $\mathcal{F}_{\mathsf{LED}}$ at time t (via $(\text{CLOCK-READ}, \mathsf{sid}_C)$ to $\mathcal{F}_{\mathsf{clock}}$) with message x_i: update $\vec{\mathcal{I}^H} = \vec{\mathcal{I}^H} \| (x_i, P_i, t)$, and send $(\text{CLOCK-UPDATE}, \mathsf{sid}_C)$ to $\mathcal{F}_{\mathsf{clock}}$ if $\mathsf{predict\text{-}time}(\vec{\mathcal{I}^H}) > t$.

Interface for party P_i:

Session-Init(sid). If $\mathsf{LOG}^{\mathsf{sid}}$ exists then ignore this request. Else, initialize ledger $\mathsf{LOG}^{\mathsf{sid}} := \emptyset$, $\mathsf{HMap}[\mathsf{sid}, 0] := 0$, internal transaction queue $\mathsf{tque}_{\mathsf{sid}} := \emptyset$. Initialize internal memory.

Submit($\mathsf{sid}, \mathsf{tx}$). To submit a transaction tx at time t: Invoke $\mathcal{A}_{\mathsf{LED}}$ on (tx, P_i) and add it to $\mathsf{tque}_{\mathsf{sid}} = \mathsf{tque}_{\mathsf{sid}} \cup (\mathsf{tx}, P_i, t)$.

Read(sid). Send ("Read", P_i) to $\mathcal{A}_{\mathsf{LED}}$ to obtain $\mathsf{LOG}_i^{\mathsf{sid}}$ from $\mathcal{A}$ and perform the following:

1) If $\mathsf{LOG}_i^{\mathsf{sid}}$ is not a prefix of $\mathsf{LOG}^{\mathsf{sid}}$ and $\mathsf{LOG}_i^{\mathsf{sid}} \neq \mathsf{LOG}^{\mathsf{sid}}$ then return $\perp$ to P_i.
2) Else, if there are entries $(\mathsf{tno}; t'; \mathsf{tx}; \mathsf{txout}; h) \in \mathsf{LOG}^{\mathsf{sid}}$ that do not appear on $\mathsf{LOG}_i^{\mathsf{sid}}$ by time $t' + \Delta$ add it to $\mathsf{LOG}_i^{\mathsf{sid}}$.

Return $\mathsf{LOG}_i^{\mathsf{sid}}$ to P_i.

Interface for the Adversary $\mathcal{A}_{\mathsf{LED}}$:

Append($\mathsf{sid}, \mathsf{tno}, t', \mathsf{tx}$). When $\mathcal{A}_{\mathsf{LED}}$ submits tx at time t':

1) Verify that $\mathsf{tno} = \mathsf{tno}_{\mathsf{last}} + 1$, where $\mathsf{tno}_{\mathsf{last}}$ is the index of the latest entry on $\mathsf{LOG}^{\mathsf{sid}}$.
2) Verify that $(\mathsf{tx}, P_i, t) \in \mathsf{tque}_{\mathsf{sid}}$ for some party P_i.
   If any of the above checks fail then return INVALID to $\mathcal{A}_{\mathsf{LED}}$. Else, tx is valid, and $\mathcal{F}_{\mathsf{LED}}$ performs:
   a) If tx is of the form ("Deploy" SC_f): Parse $(f(), \mathsf{st}_f) := SC_f$ where $f()$ is the function description and st_f is its initialized state. If $\mathsf{Checker}(f(), \mathsf{st}_f) \neq 1$ or $f()$ exists in internal memory then set $\mathsf{txout} := \mathsf{fail}$, otherwise set $\mathsf{txout} := \mathsf{Success}$ and store $(f(), \mathsf{st}_f, P_i)$ in internal memory.
   b) If tx is of the form ("Call" f, s): If f is stored in internal memory, then execute $f()$ on input s to obtain output txout and updated state st_f:
   $$(\mathsf{txout}, \mathsf{st}_f) := f(\mathsf{tno}, t', s, \mathsf{st}_f, \mathsf{HMap}[\mathsf{sid}])$$
   If function $f()$ is not deployed then set $\mathsf{txout} := \mathsf{fail}$.
3) Set $h := \mathsf{H}(\mathsf{tno}, t', \mathsf{tx}, \mathsf{txout}, \mathsf{HMap}[\mathsf{sid}, \mathsf{tno} - 1])$. Assign $\mathsf{HMap}[\mathsf{sid}, \mathsf{tno}] := h$.
4) Append $(\mathsf{tno}; t'; \mathsf{tx}; \mathsf{txout}; h)$ to $\mathsf{LOG}^{\mathsf{sid}}$.
5) Remove (tx, P_i, t) from $\mathsf{tque}_{\mathsf{sid}}$.

Default Block. If there are transactions $(\mathsf{tx}, P_i, t) \in \mathsf{tque}_{\mathsf{sid}}$ that do not appear on $\mathsf{LOG}^{\mathsf{sid}}$ by time $t' = t + \Delta$ then run $\mathsf{Append}(\mathsf{sid}, \mathsf{tno}, t + \Delta, \mathsf{tx})$ where $\mathsf{tno} = \mathsf{tno}_{\mathsf{last}} + 1$ and $\mathsf{tno}_{\mathsf{last}}$ is the index of the latest entry on $\mathsf{LOG}^{\mathsf{sid}}$.

Fig. 1: Ideal Functionality $\mathcal{F}_{\mathsf{LED}}$ for Ledger

functionality allows parties to submit input transactions. Each input transaction can be of the following form:

- It can be a request to deploy a smart contract function $f()$. In this case, $\mathcal{F}_{\mathsf{LED}}$ parses the request as the function description and list of its initialized variables, denoted as st_f. Then, $\mathcal{F}_{\mathsf{LED}}$ runs a $\mathsf{Checker}$ algorithm on (f, st_f) to check the formatting. If the check passes and if a smart contract function with the same name was not previously defined then $f()$ gets deployed.
- It can contain a valid call to a smart contract function $f()$ on input s. In that case, $f()$ is run on the input transaction, smart contract state st_f, and a hash map containing the block hashes of all the blocks for the session. The output txout is generated and the smart contract state st_f is updated.

$\mathcal{F}_{\mathsf{LED}}$ and the participating parties crucially rely on a global clock $\mathcal{F}_{\mathsf{clock}}$ (Fig 9) from [BMTZ17] to keep track of time. We also need user authentication which is modeled using the ideal signature functionality of [Can04], similar to the Gearbox paper.

Security Guarantee. The adversary $\mathcal{A}_{\mathsf{LED}}$ can corrupt an arbitrary number of participating parties P_i and run on their behalf. We allow the adversary to order the transactions and submit them to $\mathcal{F}_{\mathsf{LED}}$ to get appended on the public ledger. When a party P_i wants to read the ledger by running the Read command, the adversary returns a prefixed view of the ledger, denoted as $\mathsf{LOG}_i^{\mathsf{sid}}$ to the party. The functionality ensures that the following two properties (same as Gearbox [DMM+22]) hold:

1) **Persistence.** Suppose parties P_i and P_j are honest, and $\mathsf{LOG}_i^{\mathsf{sid}}$ and $\mathsf{LOG}_j^{\mathsf{sid}}$ are outputs of $\mathsf{Read}(\mathsf{sid})$ obtained by P_i and P_j respectively at different times. Then either $\mathsf{LOG}_i^{\mathsf{sid}}$ is a prefix of $\mathsf{LOG}_j^{\mathsf{sid}}$ or vice versa. Also, $\mathsf{LOG}_i^{\mathsf{sid}}$ and $\mathsf{LOG}_j^{\mathsf{sid}}$ should be prefixes of the global ledger $\mathsf{LOG}^{\mathsf{sid}}$ maintained by $\mathcal{F}_{\mathsf{LED}}$. This is strictly enforced in Step 1 by $\mathcal{F}_{\mathsf{LED}}$ when a party runs the "Read" command.
2) **Bounded timestamps.** When a transaction is submitted to $\mathcal{F}_{\mathsf{LED}}$ at time t (obtained via the global time functionality $\mathcal{F}_{\mathsf{clock}}$), the functionality guarantees that it eventually gets appended to the ledger by the adversary by time $t' \leq t + \Delta$. If it is not appended by time $t' = t + \Delta$ then the functionality adds it in a default block, similar to the functionality in [BMTZ17]. Additionally, the functionality also ensures that every transaction on the global ledger appears on an honest party P_j's ledger $\mathsf{LOG}_j^{\mathsf{sid}}$ by time $t' + \Delta$ on issuing a Read command by P_j. This is enforced in Step in Step 2 of "Read" command. This property ensures that an adversary can delay a transaction by at most Δ time where Δ is a bounded delay only known to the adversary and $\mathcal{F}_{\mathsf{LED}}$. It also

guarantees the liveness of the ledger as all submitted transactions will eventually get appended.

We remark that a good simulator (corresponding to a protocol implementing $\mathcal{F}_{\mathsf{LED}}$) would avoid triggering the default block execution as it results in creating a block with timestamp $t' = t + \Delta$ (other transactions have timestamps $t' \leq t + \Delta$) and it gives a distinguishing advantage to a corrupt party viewing the ledger. Next, we discuss some of the design choices of our functionality and also how it captures existing smart contract services like on-chain payments and supports block hash computation.

Modeling Block Hash. $\mathcal{F}_{\mathsf{LED}}$ is parametrized by a hash function H. This is used to compute the block hash for every transaction/block being appended to the ledger. It captures block hash in real-world blockchains and we also use it in our VRaaS protocols later on. The details of the hash function can be modified based on the protocol implementing the ledger functionality. The functionality also stores a map, denoted as HMap, containing the block hash of each block and adds it to the block being added to the ledger.

Modeling Payments. Our functionality captures on-chain payments [KLM+24]. We assume that there is a smart-contract function $\mathtt{Pay}()$ and its state $\mathsf{st}_{\mathtt{Pay}}$ maintains the account balances of each registered user. Whenever a (sender) user wants to pay a (receiver) user, the sender calls the $\mathtt{Pay}()$ function with the amount and the receiver account details. The $\mathtt{Pay}()$ function updates the account balance of the receiver and sender in its state and reflects the transaction. This approach is similar to the existing approaches [DEF+19, KLM+24] for modeling payments.

Our functionality also implicitly supports callback where a smart contract function $f()$ calls a smart contract function $g()$ on some input and then $g()$ is run on the input. In such a case, the states of both smart contracts get updated and the output of running the two functions gets appended to the ledger as a single transaction txout.

IV. Verifiable Randomness Service

A VRaaS framework enables a requester to generate w bits of randomness by invoking VRaaS servers on requester input $x \in \{0,1\}^*$ and w. The output is unique for every request (even on the same inputs). The output is accompanied by a cryptographic proof. The proof attests to the integrity of the output corresponding to the registered server verification key and the requester's input.

Once a requester initiates a randomness request, the request gets authenticated on the blockchain (via on-chain transactions) and the VRaaS server should only compute the VRaaS output and not be burdened with authenticating requests. This is especially important for the multi-blockchain setting, where requesters request randomness via multiple blockchains. This is performed via a request transaction through which a requester can request randomness. Once a request is made, the VRaaS reads it and fulfills it by generating the output and the proof. This output and proof are validated and then appended to the ledger so that they can be used for on-chain applications. This is modeled via a fulfillment transaction. We formalize VRaaS via our ideal functionality $\mathcal{F}_{\mathsf{VRaaS}}$. The $\mathcal{F}_{\mathsf{VRaaS}}$ functionality needs to read the global ledger and submit transactions to it. We model this by providing $\mathcal{F}_{\mathsf{VRaaS}}$ access to the ledger functionality $\mathcal{F}_{\mathsf{LED}}$. This approach is similar to the on-chain payment services in [DEF+19, KLM+24] where the ideal functionality for payment has access to the ledger functionality.

$\mathcal{F}_{\mathsf{SERV}}$ functionality. We capture the VRaaS servers via a separate ideal functionality $\mathcal{F}_{\mathsf{SERV}}$ to allow modularity in the formalization of VRaaS. Abstracting out the servers as a separate functionality allows us to implement it using a centralized server protocol [Chae] or a distributed server protocol [Supb] without changing the protocol implementing $\mathcal{F}_{\mathsf{VRaaS}}$. Our $\mathcal{F}_{\mathsf{SERV}}$ functionality is adapted from the distributed server functionality of [KMMM23], except we had to customize it such that it works with the $\mathcal{F}_{\mathsf{VRaaS}}$ functionality. It can also be used to capture the centralized server setting capturing randomness services offered by a single server, e.g. Chainlink. Our $\mathcal{F}_{\mathsf{SERV}}$ is provided in Fig. 4. It is parametrized by a threshold t and $\mathcal{F}_{\mathsf{SERV}}$ is assumed to be corrupt if more than t participating servers are corrupt.

We formally define the ideal functionality $\mathcal{F}_{\mathsf{VRaaS}}$ for on-chain VRaaS in a smart contract-enabled blockchain environment in Fig. 2, 3. The blockchain is modeled as a ledger functionality $\mathcal{F}_{\mathsf{LED}}$ (described in Section III). We divide our functionality in two parts- setup phase and execution phase. The setup phase occurs once where the server verification key is registered, the ledger is initialized and the smart contracts for VRaaS are deployed. The execution phase models the randomness request process, fulfilling the request on-chain and finally verification of that fulfilled request.

Parties. $\mathcal{F}_{\mathsf{VRaaS}}$ is run between the VRaaS server modeled as $\mathcal{F}_{\mathsf{SERV}}$, multiple randomness requesters $(Q_1, \ldots, Q_n)$ and the ideal world adversary Sim. Other parties can assist in the fulfillment process of the protocol. We use the Sim algorithm to model the protocol steps used to implement $\mathcal{F}_{\mathsf{VRaaS}}$ (discussed later). $\mathcal{F}_{\mathsf{VRaaS}}$ and Sim participate in $\mathcal{F}_{\mathsf{LED}}$ as parties so that they can access the ledger. $\mathcal{F}_{\mathsf{VRaaS}}$ initializes the VRaaS public key as $\mathsf{VK} := \emptyset$, an internal request counter (for keeping track of randomness requests) $\mathsf{reqCtr} := 0$, and internal memory as $\mathsf{mem}_{\mathsf{F}} := \emptyset$.

Parties. Server functionality $\mathcal{F}_{\mathsf{SERV}}$, registered requesters $(Q_1, \ldots Q_n)$ and other parties. The adversary, denoted by Sim, can corrupt $\mathcal{F}_{\mathsf{SERV}}$, the requesters, and the other parties. We say $\mathcal{F}_{\mathsf{SERV}}$ is corrupt when the adversary corrupts more than the (respective) threshold number of parties participating in $\mathcal{F}_{\mathsf{SERV}}$. The above parties and $\mathcal{F}_{\mathsf{VRaaS}}$ participate in $\mathcal{F}_{\mathsf{LED}}$.

Notation. We denote each tuple $(\mathsf{tno}; t'; \mathsf{tx}; \mathsf{txout}; h)$ by $(\mathsf{tx}; \mathsf{txout})$ for notational simplicity. We denote a variable as _ when its value is irrelevant.

Initialize VRaaS verification key $\mathsf{VK} = \emptyset$, unique request counter $\mathsf{reqCtr} := 0$, and internal memory of $\mathcal{F}_{\mathsf{VRaaS}}$ as $\mathsf{mem_F} := \emptyset$.

Key-Registration.

1. Upon receiving ("`Key-Register`") from Sim forward message to $\mathcal{F}_{\mathsf{SERV}}$. Upon receiving $(vk, \mathsf{Verify}(\cdot))$ from $\mathcal{F}_{\mathsf{SERV}}$: If $\mathsf{VK} \neq \emptyset$, then ignore request.
2. Otherwise, set $\mathsf{VK} = vk$ and store $\mathsf{Verify}(\cdot)$ in $\mathsf{mem_F}$.

Ledger-Initialization.

3. Upon receiving ("`Init-Ledger`", sid) from Sim: Initialize the ledger for session sid by running $\mathcal{F}_{\mathsf{LED}}.\mathsf{Session\text{-}Init}(\mathsf{sid})$.

Smart Contract Deployment.

4. Upon receiving ("`Deploy`", $SC_{\texttt{Req-Rand}}, SC_{\texttt{Req-Fulf}}, \mathsf{sid}$) from Sim: Deploy the smart contracts `Req-Rand` and `Req-Fulf` on $\mathcal{F}_{\mathsf{LED}}$ by running:

$$\mathcal{F}_{\mathsf{LED}}.\mathsf{Submit}(\mathsf{sid}, \text{"}\texttt{Deploy}\text{"}\ SC_{\texttt{Req-Rand}}),$$

$$\mathcal{F}_{\mathsf{LED}}.\mathsf{Submit}(\mathsf{sid}, \text{"}\texttt{Deploy}\text{"}\ SC_{\texttt{Req-Fulf}}).$$

Smart Contract Functions deployed on $\mathcal{F}_{\mathsf{LED}}$:

- `Req-Rand`(x, w) :
  Compute unique identifier qID from (x, w) and state of the smart contract. Append to $\mathsf{LOG}^{\mathsf{sid}}$:

$$(\textsc{Req-Rand}, x, w; \mathsf{qID}).$$

- `Req-Fulf`(qINFO, y, π) :
  Generate qID and qINP from qINFO, and check $\mathsf{Verify}(\mathsf{VK}, \mathsf{qINP}, y, \pi) = 1$, where VK is hardcoded. If verification succeeds then append to $\mathsf{LOG}^{\mathsf{sid}}$:

$$(\textsc{Req-Fulf}, \mathsf{qINFO}, y, \pi; \mathsf{qID}, \mathsf{qINP}, \mathsf{VK}).$$

Fig. 2: Ideal Functionality $\mathcal{F}_{\mathsf{VRaaS}}$ (Setup Phase).

The following three functions can be run in parallel by multiple participating parties. For notational simplicity, we assume that $\mathsf{LOG}^{\mathsf{sid}}$ is used by the functions.

Request-Randomness.

5. *If Q_i is honest:* Upon receiving ("`Req-Rand`", x, w) from Q_i, where $x \in \{0,1\}^*$ and w denotes the number of random bits to be generated, run the request transaction on (x, w) as:

$$\mathcal{F}_{\mathsf{LED}}.\mathsf{Submit}(\mathsf{sid}, (\text{"}\texttt{Call Req-Rand}\text{"}, x, w)).$$

   Read qID once the request gets confirmed.
   If Q_i is corrupt: When Sim sends (qID, w, Q) ignore it if $\langle \textsc{Req-Rand}, \mathsf{qID}, w, \_, \_\rangle \in \mathsf{mem_F}$. Otherwise, proceed.
6. Store request in internal memory $\mathsf{mem_F} := \mathsf{mem_F} \cup \langle \textsc{Req-Rand}, \mathsf{qID}, w, \mathsf{reqCtr}, Q_i\rangle$.
7. Update counter $\mathsf{reqCtr} := \mathsf{reqCtr} + 1$.

Fulfillment.
Upon receiving ("`Eval-Req`", qID) from any party:

8. If the request identified by qID was marked `fulfilled`, i.e. $\langle \textsc{Req-Fulf}, \mathsf{qID}, \_, \_, \_, Q\rangle \in \mathsf{mem_F}$, then ignore it.
9. Invoke Sim("`Req-FulInp`", qID, w) $\rightarrow$ $(\mathsf{qINP}, \mathsf{qINFO})$ to obtain query input qINP and query information qINFO.
10. Fetch entry $\langle \textsc{Req-Rand}, \mathsf{qID}, w, \mathsf{rCtr}, Q_i\rangle \leftarrow \mathsf{mem_F}$ from memory corresponding to qID.
11. *If $\mathcal{F}_{\mathsf{SERV}}$ is honest*: Set $y \leftarrow \mathsf{Rand}(\mathsf{VK}, \mathsf{rCtr}; w)$ and obtain proof as $\pi := \mathcal{F}_{\mathsf{SERV}}$("`Sim-Proof`", $\mathsf{VK}, \mathsf{qINP}, w, y$).
    If $\mathcal{F}_{\mathsf{SERV}}$ is corrupt: Obtain output and proof as $(y, \pi) := \mathcal{F}_{\mathsf{SERV}}$("`Eval`", $\mathsf{VK}, \mathsf{qINP}, w$). If $\mathcal{F}_{\mathsf{SERV}}$ return $\perp$ then skip this fulfillment process.
12. Skip this fulfillment process if $\mathsf{Verify}(\mathsf{VK}, \mathsf{qINP}, y, \pi) \neq 1$.
13. If $\mathcal{F}_{\mathsf{SERV}}$ is honest or if $\mathcal{F}_{\mathsf{SERV}}$ is corrupt and Sim instructs $\mathcal{F}_{\mathsf{VRaaS}}$ (upon being queried by $\mathcal{F}_{\mathsf{VRaaS}}$ as Sim("`Run Fulfillment?`") $\rightarrow$ `Yes/No`) to run the fulfillment transaction then perform:

$$\mathcal{F}_{\mathsf{LED}}.\mathsf{Submit}(\mathsf{sid}, (\text{"}\texttt{Call Req-Fulf}\text{"}, \mathsf{qINFO}, y, \pi)).$$

    Once the transaction output is appended on $\mathcal{F}_{\mathsf{LED}}$ mark qID `fulfilled` by storing it in memory as $\mathsf{mem_F} := \mathsf{mem_F} \cup \langle \textsc{Req-Fulf}, \mathsf{qID}, \mathsf{qINP}, \mathsf{rCtr}, y, \pi, Q\rangle$.

Local Verification.
Upon receiving ("`Verify-Local`", qINP, y, π, Q), from any party M:

14. If $\langle \textsc{Req-Fulf}, \_, \mathsf{qINP}, \_, y, \pi, Q\rangle \in \mathsf{mem_F}$ then send VERIFIED to party M. Otherwise, send $\perp$ to M.

Fig. 3: Ideal Functionality $\mathcal{F}_{\mathsf{VRaaS}}$ (Execution Phase).

VRaaS-Key-Registration. To register a server P, $\mathcal{F}_{\mathsf{VRaaS}}$ receives a command ("`Key-Register`") from Sim and forwards it to $\mathcal{F}_{\mathsf{SERV}}$. Then $\mathcal{F}_{\mathsf{SERV}}$ generates a verification key vk and the $\mathsf{Verify}(\cdot)$ algorithm and returns it to $\mathcal{F}_{\mathsf{VRaaS}}$. The $\mathsf{Verify}(\cdot)$ algorithm will be used to verify the server output on-chain by smart-contracts without interaction with $\mathcal{F}_{\mathsf{SERV}}$. $\mathcal{F}_{\mathsf{VRaaS}}$ sets $\mathsf{VK} = vk$ as the registered verification key. Allowing $\mathcal{F}_{\mathsf{SERV}}$ to generate vk allows us to limit the problem of key-registration inside $\mathcal{F}_{\mathsf{SERV}}$. Looking ahead, this will also help us in simulation when $\mathcal{F}_{\mathsf{VRaaS}}$ samples a random output y for a request, and then Sim invokes the adversary $\mathsf{Sim_P}$ to generate the simulated proof π by using the knowledge of sk.

Ledger-Initialization. To initialize the ledger Sim triggers $\mathcal{F}_{\mathsf{VRaaS}}$ and then $\mathcal{F}_{\mathsf{VRaaS}}$ initiates a new ledger $\mathsf{LOG}^{\mathsf{sid}}$ corresponding to session sid by calling $\mathcal{F}_{\mathsf{LED}}$. If a ledger already exists for session sid then $\mathcal{F}_{\mathsf{LED}}$ ignores this request.

Smart Contract Deployment. The functionality initializes two smart contracts - `Req-Rand` for requesting randomness and `Req-Fulf` for verifying and storing the fulfilled requests. The smart contract functions are defined by the concrete protocol that implements $\mathcal{F}_{\mathsf{VRaaS}}$ and the $\mathsf{Verify}(\cdot)$ function. So our abstraction supports such customizations by allowing Sim to send the concrete smart contract function details to $\mathcal{F}_{\mathsf{VRaaS}}$. Details of the

Parties. Registered parties $\mathbf{P} := (P_1, \dots P_m)$. The adversary, denoted by $\mathsf{Sim_P}$, can corrupt the parties.
Parameter. $\mathcal{F}_{\mathsf{SERV}}$ is parametrized by corruption threshold t and a $\mathsf{Verify}(\cdot)$ function. Initialize $\mathsf{T}[\cdot,\cdot]$ and $\mathsf{T_{par}}[\cdot,\cdot]$.

Key-Registration.

1. Upon receiving ("`Key-Register`") from $\mathcal{F}_{\mathsf{VRaaS}}$ forward it to $\mathsf{Sim_P}$.
2. On $(\mathbf{P}, vk, \{vk_1, \dots, vk_m\})$ from $\mathsf{Sim_P}$: Parse $\mathbf{P} := \{P_1, \dots, P_m\}$ and when vk is unique:
   a) Define $\mathbf{P}_{\mathsf{CORR}} \subset \mathbf{P}$ is the set of corrupt servers and $\mathbf{P}_{\mathsf{HON}} := \mathbf{P} \setminus \mathbf{P}_{\mathsf{CORR}}$ is the set of honest servers.
   b) For each $P_i \in \mathbf{P}$ set $Keys[P_i] := vk_i$.
   c) If $|\mathbf{P}_{\mathsf{CORR}}| \geq t+1$, then mark server set $\mathbf{P}$ as "`Corrupt`".
   d) Send $(\mathbf{P}, vk, vk_i)$ to each $P_i \in \mathbf{P}_{\mathsf{HON}}$ and register $(\mathbf{P}, vk)$.
3. Send $(vk, \mathsf{Verify}(\cdot))$ to $\mathcal{F}_{\mathsf{VRaaS}}$.

Evaluation.
When $\mathcal{F}_{\mathsf{VRaaS}}$ sends ("`Sim-Proof`", $\mathsf{VK}, \mathsf{qINP}, w, y$) or ("`Eval`", vk, qINP, w):

4. Send ("`Eval`", qINP) to all servers in $\mathbf{P}$.
5. On ("`Partial-Eval`", qINP, vk_j) from server P_j: Forward request to $\mathsf{Sim_P}$. If $\mathsf{Sim_P}$ sends $(vk_j, \mathsf{qINP}, j, y_j, \pi_j)$ then forward it to P_j and set $\mathsf{T_{par}}[\mathsf{qINP}, vk, P_j] := (y_j, \pi_j)$. If the same query is repeated then fetch (y_j, π_j) from $\mathsf{T_{par}}$ and return it to P_j.
6. On ("`Aggregate`", $\mathsf{qINP}, vk, \{(y_i, \pi_i)\}_{i\in[\ell]}$) from a registered party P: If $\ell < t+1$, then return $\perp$. Otherwise perform the following based on the corruption of P:
   - If server set $\mathbf{P}$ is honest then $\mathcal{F}_{\mathsf{SERV}}$ was invoked on ("`Sim-Proof`", $\mathsf{VK}, \mathsf{qINP}, w, y$). Perform:
     a) Send $(\mathsf{qINP}, w, y, vk, \{(y_i, \pi_i)\}_{i\in[\ell]})$ to $\mathsf{Sim_P}$.
     b) If $\mathsf{Sim_P}$ sends $\perp$ then send it to $\mathcal{F}_{\mathsf{VRaaS}}$ and exit. Otherwise, receive π from $\mathsf{Sim_P}$ and register (y, π) as $\mathsf{T}[\mathsf{qINP}, vk] := (y, \pi)$.
     c) Send π to $\mathcal{F}_{\mathsf{VRaaS}}$.
   - If server set $\mathbf{P}$ is corrupt then $\mathcal{F}_{\mathsf{SERV}}$ was invoked on ("`Eval`", vk, qINP, w). Perform:
     a) Send $(\mathsf{qINP}, w, vk, \{(y_i, \pi_i)\}_{i\in[\ell]}, y)$ to $\mathsf{Sim_P}$.
     b) If $\mathsf{Sim_P}$ sends $\perp$ then send it to P and exit. Otherwise, receive (y, π) from $\mathsf{Sim_P}$ and register (y, π) as $\mathsf{T}[\mathsf{qINP}, vk] := (y, \pi)$.
     c) Send (y, π) to $\mathcal{F}_{\mathsf{VRaaS}}$.

Fig. 4: Helper Ideal Functionality $\mathcal{F}_{\mathsf{SERV}}$ for the Server in $\mathcal{F}_{\mathsf{VRaaS}}$.

smart contract functions are:

- `Req-Rand`: Given a request input x and the number random bits requested denoted as w, it generates a unique request identifier qID and appends them to the ledger. Under the hood, the protocol implementing $\mathcal{F}_{\mathsf{VRaaS}}$ should ensure that the simulator provides a smart contract function $SC_{\texttt{Req-Rand}}$ that fulfills the functionality of `Req-Rand`, i.e. it generates a unique qID for every request (even for different requests on the same requester input x) since qID uniquely identifies each request on the ledger. Looking ahead, in the fulfillment process a qINFO will be generated corresponding to each (x, qID) for a request. It basically contains the qID and some auxiliary information. This qINFO will be used to generate a qINP on which the $\mathcal{F}_{\mathsf{SERV}}$ will generate the output y and proof π. qINFO contains additional information like the requester's callback function, number of random bits requested, block hash etc, which are not included in qID and qINP.
  We note that if the qID is not unique for every request then the protocol implementing the $\mathcal{F}_{\mathsf{VRaaS}}$ cannot be proven as secure. It is because the $\mathcal{F}_{\mathsf{VRaaS}}$ functionality generates a random output for each request (by assigning a unique counter value reqId and running $\mathsf{Rand}()$ on that), whereas the protocol generates it for each qID. If a malformed $SC_{\texttt{Req-Rand}}$ smart contract assigns the same qID to two different requests then the output will be same for both requests in the real world, whereas in the ideal world it will be different leading to a concrete attack. This implicitly ensures that the simulator provided $SC_{\texttt{Req-Rand}}$ should be correct.
- `Req-Fulf`: Given qINFO, output y and proof π it generates qID and qINP, and verifies the generation of y on qINP by verifying the proof. Once verification succeeds the output is appended to the ledger. Under the hood, the protocol implementing $\mathcal{F}_{\mathsf{VRaaS}}$ should ensure that the simulator provides a smart contract function $SC_{\texttt{Req-Fulf}}$ that fulfills the functionality of `Req-Fulf`, i.e. it generates qINP and qID from qINFO, and the output y and proof π verifies w.r.t. input qINP. If the smart contract $SC_{\texttt{Req-Fulf}}$ does not correctly generate qINP and verify y then in the real-world execution a malformed output would be accepted, whereas in the ideal-world, the fulfillment step would be skipped since the $\mathcal{F}_{\mathsf{VRaaS}}$ functionality verifies the output in Step. 12. and continues with the fulfillment only if the output correctly verifies. As a result, a protocol using a malformed $SC_{\texttt{Req-Fulf}}$ cannot be proven to be secure and this implicitly ensures that the simulator provided $SC_{\texttt{Req-Rand}}$ should be correct. Furthermore, if this output y needs to be used in an application smart-contract `app` then the `Req-Fulf` should have a callback to $\texttt{app}(y)$. But for generality, we do not enforce $\mathcal{F}_{\mathsf{VRaaS}}$ to perform callbacks as the $\mathcal{F}_{\mathsf{VRaaS}}$ should capture VRaaS protocols where there is no callback in the fulfillment transaction.

Request-Randomness. To request w bits of randomness on input x the requester Q_i runs ("`Req-Rand`", x, w). In doing so, $\mathcal{F}_{\mathsf{VRaaS}}$ runs the VRaaS smart contract function `Req-Rand` on the requester input (x, w) to obtain a smart contract generated query ID qID. The request (x, w) and the qID gets appended to the ledger. $\mathcal{F}_{\mathsf{VRaaS}}$ assigns the current value of reqCtr to the request and stores $\langle \textsc{Req-Rand}, \mathsf{qID}, w, x, \mathsf{reqCtr}, Q_i \rangle$ in its memory. The

randomness request counter is incremented, i.e. $\mathsf{reqCtr} := \mathsf{reqCtr} + 1$ for each request. The reqCtr is internal to $\mathcal{F}_{\mathsf{VRaaS}}$ and ensures that each tuple is unique due to the uniqueness of reqCtr. This ensures that each request is stored in a unique format even though the requester input (x, Q_i) can be the same. Note that $\mathcal{F}_{\mathsf{VRaaS}}$ allows requesting randomness on the same input multiple times as it is a VRaaS. Later, the randomness will be generated corresponding to each reqCtr value, and assigning a unique reqCtr to each request ensures that the output randomness will be independently generated for each request. $\mathcal{F}_{\mathsf{VRaaS}}$ allows multiple requesters to request randomness concurrently via the "`Req-Rand`" command. Each request will be independently registered in $\mathsf{mem}_{\mathsf{F}}$ with a unique reqId value.

Fulfillment. Upon receiving ("`Eval-Req`", qID) as a randomness request from any party M: $\mathcal{F}_{\mathsf{VRaaS}}$ verifies that the request with ID qID has not been fulfilled (looking ahead $\mathcal{F}_{\mathsf{VRaaS}}$ registers each successful fulfillment request as fulfilled in its memory). This avoids double-fulfilling the same request, identified by qID. Once the checks pass, $\mathcal{F}_{\mathsf{VRaaS}}$ invokes Sim on (qID, w) to obtain query information qINFO and query input qINP. The output and the proof will be generated on qINP and the fulfillment transaction will verify it. qINP depends on the protocol and state of the $\mathsf{LOG}^{\mathsf{sid}}$ and so Sim is responsible for generating it. For example, qINP could include the block hash (i.e. hash of the block containing the request transaction). qINFO contains additional information that allows one to regenerate qID and qINP from it. qINFO acts as a placeholder to capture the details of the VRaaS protocol (implementing $\mathcal{F}_{\mathsf{VRaaS}}$) which do not appear in the input to the VRaaS servers. Once qINFO and qINP is generated by Sim, $\mathcal{F}_{\mathsf{VRaaS}}$ generates the output and proof (y, π) as follows based on $\mathcal{F}_{\mathsf{SERV}}$'s corruption.

- If VRaaS server P is honest, then $\mathcal{F}_{\mathsf{VRaaS}}$ samples a random w-bits string by running $\mathsf{Rand}(\mathsf{VK}, \mathsf{rCtr}; w)$ to generate the output y. The uniqueness of rCtr ensures that each request $(\mathsf{VK}, \mathsf{rCtr})$ uniquely corresponds to the session with verification VK. Once y is generated, $\mathcal{F}_{\mathsf{VRaaS}}$ invokes $\mathcal{F}_{\mathsf{SERV}}$ with input $(\mathsf{VK}, \mathsf{qINP}, w, y, Q)$ to obtain the simulated proof π that attests to the computation of y on qINP. In $\mathcal{F}_{\mathsf{SERV}}$, the simulator algorithm $\mathsf{Sim}_{\mathsf{P}}$ is run which returns the simulated proof to $\mathcal{F}_{\mathsf{SERV}}$, and that is forwarded to $\mathcal{F}_{\mathsf{VRaaS}}$. $\mathcal{F}_{\mathsf{VRaaS}}$ sets the output as (y, π) and runs the smart contract function `Req-Fulf` on (qINFO, y, π) to register the output as fulfilled on $\mathsf{LOG}^{\mathsf{sid}}$. Once the fulfillment transaction gets appended to the ledger $\mathcal{F}_{\mathsf{VRaaS}}$ registers the request as fulfilled by storing the entry $\langle \textsc{Req-Fulf}, \mathsf{qID}, \mathsf{qINP}, \mathsf{rCtr}, y, \pi, Q \rangle$[4] in its memory.
- Meanwhile, if VRaaS server P is corrupt then $\mathcal{F}_{\mathsf{VRaaS}}$ invokes $\mathcal{F}_{\mathsf{SERV}}$ with input $(\mathsf{VK}, \mathsf{qINP}, w)$ to obtain the output (y, π). In this case, Sim is responsible for running the fulfillment transaction on behalf of the corrupt $\mathcal{F}_{\mathsf{SERV}}$.

Local-Verification. This step is run by any party M that wants to verify an output (y, π) corresponding to a query input qINP. The verification process succeeds if the request is marked as fulfilled in $\mathcal{F}_{\mathsf{VRaaS}}$'s memory.

Given the above functionality we discuss how $\mathcal{F}_{\mathsf{VRaaS}}$ captures the following properties required for a VRaaS:

1) **Tackling Impersonation:** Each randomness request initiated by a requester Q_i is stored in the memory $\mathsf{mem}_{\mathsf{F}}$ along with the requester ID Q_i in Step 6. It prevents a different requester $Q' \neq Q_i$ from requesting randomness on behalf of Q_i. Similarly, upon fulfillment of a request the functionality stores the query input qINP, verifiable out (y, π), and the requester ID Q_i in $\mathsf{mem}_{\mathsf{F}}$ in Step 13. It binds the output to Q_i. If Q' wants to use this output then verification fails in Step.14. as the output is not registered with Q'.
2) **Input Uniqueness:** For each randomness request with requester input x, the smart contract adds a tuple containing qID. Then $\mathcal{F}_{\mathsf{VRaaS}}$ adds a unique nonce rCtr (maintained by the internal counter reqCtr) for each request in Step 6. The randomness request is stored in the memory as a tuple $\langle \textsc{Req-Rand}, \mathsf{qID}, w, \mathsf{reqCtr}, Q_i \rangle$. This tuple is unique due to the uniqueness of rCtr. Looking ahead, if a protocol implements $\mathcal{F}_{\mathsf{VRaaS}}$ then it should ensure that qID_i is unique for each request.
3) **Unbiasable Random Output:** When $\mathcal{F}_{\mathsf{SERV}}$ is honest and any party requests fulfillment of a query qID, $\mathcal{F}_{\mathsf{VRaaS}}$ generates a random w-bit output y on unique input $(\mathsf{VK}, \mathsf{rCtr}; w)$ by invoking Rand function in Step 11. It is ensured the output is random and unique due to the uniqueness of rCtr (as discussed above in "Input Uniqueness" paragraph). Looking ahead, if a protocol implements $\mathcal{F}_{\mathsf{VRaaS}}$ then it should ensure that qINP is unique for each request.
4) **Unforgeability:** Assuming $\mathcal{F}_{\mathsf{SERV}}$ is honest, an adversarial requester cannot forge a VRaaS output (y, π) on qINP without querying $\mathcal{F}_{\mathsf{VRaaS}}$ since the entry $\langle \textsc{Req-Fulf}, \mathsf{qID}, \mathsf{qINP}, \mathsf{rCtr}, y, \pi, Q \rangle$ will not be registered in $\mathsf{mem}_{\mathsf{F}}$ unless the entry was added (end of Step 13.) to $\mathsf{mem}_{\mathsf{F}}$ by $\mathcal{F}_{\mathsf{VRaaS}}$ after successful execution of the Fulfillment step. This would lead to the verification process outputting $\perp$.

[4]Storing qID and rCtr allows $\mathcal{F}_{\mathsf{VRaaS}}$ to check that the qID was fulfilled by running the consistency check in Step 8. Storing qINP is essential for the output verification in the next step.

5) **On-Chain Verifiability:** The request transaction and the fulfillment transaction can be verified on-chain as the output is appended on-chain. It can be further utilized for other on-chain applications.

We note that many existing randomness services [Chab] include the block hash h (of the block containing the request) inside qINP. This helps in ensuring unbiasability of the output even when $\mathcal{F}_{\mathsf{SERV}}$ is corrupt. Since block hash (of the current block) is not available during `Req-Rand`, it cannot be included in qID. Instead, Sim includes the block hash inside qINP and the block number (containing the `Req-Rand` transaction) inside qINFO. Later, in the fulfillment phase, `Req-Fulf` verifies the block hash of `Req-Rand` in qINP by matching it with HMap in $\mathcal{F}_{\mathsf{LED}}$ corresponding to the block number in qINFO. In Appendix. E, we show how to validate that qINFO corresponds to the correct request x and its qID.

Implementing $\mathcal{F}_{\mathsf{VRaaS}}$. A VRaaS protocol needs to satisfy the following conditions to implement $\mathcal{F}_{\mathsf{VRaaS}}$:

1) *Condition 1:* For every request, qID and qINP are unique to ensure that each request is treated independently and qID are recorded on-chain via `Req-Rand` transaction.
2) *Condition 2:* Given an unqueried qINP, the output y (obtained by performing computation on qINP and some secret information sk) is always unpredictable and pseudorandom. Combined with the previous condition, this ensures that the VRaaS output on an unqueried randomness request is always unpredictable and pseudorandom.
3) *Condition 3:* qINP and qID are generated from qINFO (by `Req-Fulf` transaction) and the output y verifies on-chain w.r.t. the proof π, qINP and server verification key vk. This ensures that the fulfillment is recorded on-chain corresponding to qID.

V. VRF-BASED PROTOCOL

In this section, we present a simple VRF-based randomness service $\pi_{\mathsf{R\text{-}VRF}}$ and prove that it implements $\mathcal{F}_{\mathsf{VRaaS}}$ functionality. Here, multiple requesters request randomness and there is a single VRaaS server that responds to those requests. We assume that the VRaaS server implements $\mathcal{F}_{\mathsf{SERV}}$ by running a VRF protocol. We assume that there is $\mathcal{F}_{\mathsf{RLY}}$ (Fig.7) that acts as a relay between $\mathcal{F}_{\mathsf{LED}}$ and the VRaaS server. $\mathcal{F}_{\mathsf{RLY}}$ reads messages from the blockchain and invokes the VRaaS server with it. Later, we show how to distribute the server by implementing $\mathcal{F}_{\mathsf{SERV}}$ using a server committee and $\mathcal{F}_{\mathsf{RLY}}$ using a relay committee.

Overview of $\pi_{\mathsf{R\text{-}VRF}}$. We present the generic VRF-based randomness service protocol $\pi_{\mathsf{R\text{-}VRF}}$ in Fig. 6 and the message flow can be visualized in Fig. 5. We assume there is a single VRaaS server that registers its public key vk on $\mathcal{F}_{\mathsf{LED}}$. At a high level, a requester requests λ bits of randomness on its input $x := \mathsf{param}$, where param are the parameters for its callback function (the requester receives the output randomness via the callback function once the randomness request is fulfilled).

The request gets uploaded on $\mathsf{LOG}^{\mathsf{sid}}$ as $\mathsf{qID} := (x, \mathsf{reqId})$ by running the `Req-Rand` smart contract, where reqId is a unique nonce for each request ensuring each qID is unique. $\mathcal{F}_{\mathsf{RLY}}$ reads the randomness requests from $\mathsf{LOG}^{\mathsf{sid}}$. $\mathcal{F}_{\mathsf{RLY}}$ verifies that the request was not processed by verifying that a corresponding output tuple is not present on $\mathsf{LOG}^{\mathsf{sid}}$.Next, to fulfill the request of the form $\mathsf{qID} = (x, \mathsf{reqId})$ it invokes the VRaaS server on input $\mathsf{qINP} := \mathsf{qID}$ to receive the verifiable output (y, π). This ensures that qINP is unique, and the output y is unpredictable and pseudorandom due to the VRF security; thus meeting Conditions 1 and 2. The $\mathcal{F}_{\mathsf{RLY}}$ then posts this output on $\mathsf{LOG}^{\mathsf{sid}}$ by running the `Req-Fulf` smart contract on the output. For on-chain verification, one needs to check that the output tuple $(\text{REQ-FULF}, \mathsf{qINFO}, y, \pi; \mathsf{qID}, \mathsf{qINP}, vk)$ is present on $\mathsf{LOG}^{\mathsf{sid}}$ (where $\mathsf{qINFO} = \mathsf{qINP} = \mathsf{qID}$). The fact that the output tuple exists on-chain proves that the VRF output (y, π) verifies w.r.t. (qINP, vk). Hence, condition 3 is trivially met. To formally prove that this protocol implements $\mathcal{F}_{\mathsf{VRaaS}}$ we need a secure simulatable VRF (Section II) since the $\mathcal{F}_{\mathsf{VRaaS}}$ functionality samples the random output in the ideal world and the simulator has to simulate the corresponding proof for it. Assuming such a VRF protocol, we prove that the protocol $\pi_{\mathsf{R\text{-}VRF}}$ implements the randomness service by proving Thm. 1 in Appendix. C.

Theorem 1. *Assume VRF $= (\mathsf{Gen}^{\mathsf{H}}, \mathsf{Eval}^{\mathsf{H}}, \mathsf{Verify}^{\mathsf{H}})$ is a simulatable verifiable random function (as described in Section II),* H *is a global random oracle and $\mathcal{F}_{\mathsf{LED}}$ is a global ledger functionality (Fig. 1). Then the protocol $\pi_{\mathsf{R\text{-}VRF}}$ (Fig. 6) UC-securely implements $\mathcal{F}_{\mathsf{VRaaS}}$ in the $(\mathcal{F}_{\mathsf{LED}}, \mathcal{F}_{\mathsf{RLY}})$-model against malicious and static corruption of the requesters and $\mathcal{F}_{\mathsf{SERV}}$ by a PPT adversary $\mathcal{A}$.*

We show how to relax these two assumptions and discuss how to extend the protocol against adaptive corruption. We also revisit the need for blockchain awareness (i.e. participating in $\mathcal{F}_{\mathsf{LED}}$) from $\mathcal{F}_{\mathsf{RLY}}$ and the server. These are presented in Appendix. D. We also argue that variants of $\pi_{\mathsf{R\text{-}VRF}}$ capture the Chainlink VRF [Chab] and Supra dVRF VRF [Supc].

VI. REAL-WORLD RANDOMNESS SERVICES

Finally, we conclude this section by showing that Chainlink VRF service and Supra dVRF service are captured via $\pi_{\mathsf{R\text{-}VRF}}$ protocol.

Analysis of Chainlink VRF Service. In Chainlink VRF [Chab], to request randomness the requester Q sets

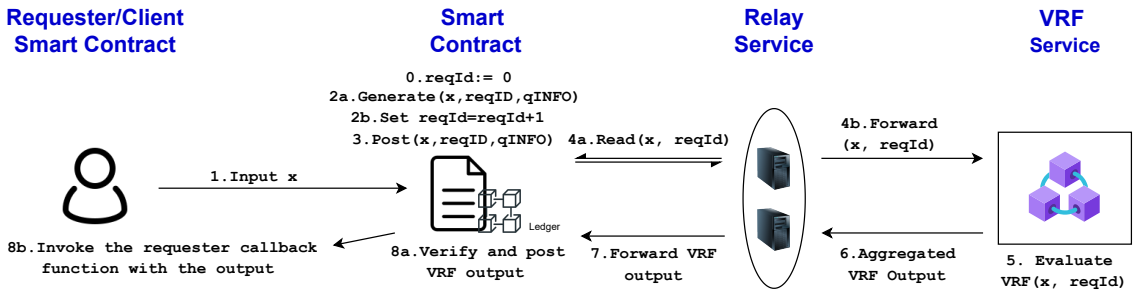

Fig. 5: Message flow in VRF based protocol $\pi_{\mathsf{R\text{-}VRF}}$. Input x includes the callback function parameters.

$x := (\mathsf{khash}, \mathsf{account}, \mathsf{config}, \mathsf{gas})$ and $w = \mathsf{len}$ where $\mathsf{khash} = \mathsf{H}(vk_i)$ specifies which trusted VRF server should fulfill the request, account is the client's account information, config is the number of confirmation blocks the VRF server should wait, gas is the maximum gas (on-chain instructions run by smart contract) cost to be executed in the callback function and len is the number of random bits the client wants. The client smart contract generates a randomness request. The Chainlink smart contract verifies the authenticity of the request by checking the account information. It maintains a counter reqId. It computes $\mathsf{preseed} = \mathsf{H}(\mathsf{khash}, \mathsf{cinfo}, \mathsf{account}, \mathsf{reqId})$ and $\mathsf{qID} = \mathsf{H}(\mathsf{khash}, \mathsf{preseed})$, and increments the counter reqId. It posts $(x, \mathsf{len}; \mathsf{qID})$ on the ledger and returns qID. It also maintains an on-chain commitment to store the hash of the above information for validation.

Once the transaction gets uploaded, the Chainlink VRF server reads $(x, \mathsf{len}; \mathsf{qID})$, and once config blocks have been confirmed on the ledger after the block containing the request, the Chainlink server computes preseed from $(x, \mathsf{len}; \mathsf{qID})$, verifies the hash of the request information against the on-chain commitment and computes the Elliptic-curve based VRF (Appendix A) on $\mathsf{qINP} := \mathsf{H}(\mathsf{preseed}, \mathsf{bhash})$ to obtain a verifiable output (y, π), where $y \in \{0,1\}^{\mathsf{len}}$, bhash is the block-hash (of the block containing $(x, \mathsf{len}; \mathsf{qID})$) preventing precomputation of the VRF output. The VRF server then calls the fulfillment smart contract on output (y, π) and qINFO consisting of – preseed, block number that contains the request transaction and user-specific information. The fulfillment transaction regenerates qINP, validates it against the on-chain commitments (to the hash of the request), and then verifies the output using π, and upon successful verification, it performs the callback with (y, π).

In the Chainlink VRF service, both $\mathcal{F}_{\mathsf{RLY}}$ and $\mathcal{F}_{\mathsf{SERV}}$ is instantiated using the centralized Chainlink VRF server, and as a result, the server has to be blockchain-aware to read the randomness requests from the blockchain. Next, we argue that the Chainlink VRF service is a VRaaS. The output y is unique for every randomness request. Each qID is guaranteed to be unique, even for the same client inputs, due to the unique reqId added by the Chainlink smart contract in preseed (assuming H is collision-resistant). The same argument holds for the uniqueness of qINP. This translates to unique pseudorandom output y due to the pseudorandomness of the underlying VRF protocol, proving Conditions 1 and 2. Condition 3 holds as qINP is recomputed from qINFO and y verifies due to the correctness of the VRF. We can further show that the Chainlink VRF service implements the $\mathcal{F}_{\mathsf{VRaaS}}$ functionality.

Analysis of Supra dVRF Service. In the Supra dVRF service [Supc], the $\mathcal{F}_{\mathsf{SERV}}$ is instantiated by a committee of VRF server nodes satisfying an honest majority. They run the GLOW-dVRF protocol [GLOW21], i.e. the distributed version of BLS signatures. $\mathcal{F}_{\mathsf{RLY}}$ is realized by a committee of relay nodes satisfying honest majority assumption. To request randomness, the requester Q sets its input $x := (\mathsf{param}, \mathsf{config}, \mathsf{cseed}, \mathsf{account})$ and $w = \mathsf{len}$ where param is the callback function, config is the number of confirmation blocks the VRF Committee should wait, cseed (can be empty) is client provided entropy, account is the account information (w.r.t. billing etc.) and len is the number of bits of randomness requested. The client smart contract invokes the Supra smart contract with input x and w to request randomness. The Supra smart contract checks that the client account has sufficient balance. It maintains an on-chain reqId. It computes $\mathsf{qID} := \mathsf{reqId}$ and increments $\mathsf{reqId} = \mathsf{reqId} + 1$. The final request transaction that is posted is of the form $(x, \mathsf{len}; \mathsf{qID}, \mathsf{aux})$ where aux contains some auxiliary information (the number of confirmation blocks, amount of randomness to be generated, etc).

Then, the committee of relay nodes reads the request $(x, \mathsf{len}; \mathsf{qID}, \mathsf{aux})$, and once config blocks have been confirmed on the ledger after qID, the relay nodes invoke the Supra dVRF committee on $\mathsf{qINP} = (x, \mathsf{qID}, \mathsf{bhash})$, where bhash is the block-hash of the $N + \mathsf{config}$ block and the request was confirmed on Nth block. This

Parties. VRaaS server P, Multiple requesters $(Q_1, \ldots, Q_n)$, ideal relay functionality $\mathcal{F}_{\mathsf{RLY}}$. Parties have access to the global random oracle $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$.

Key-Registration. VRaaS server P computes $(vk, sk) \leftarrow \mathsf{VRF.Gen}^{\mathsf{H}}(1^\lambda)$ and sends vk to the requesters.

Ledger-Initialization. Parties initialize $\mathsf{LOG}^{\mathsf{sid}}$ for session sid by running $\mathcal{F}_{\mathsf{LED}}.\mathsf{Session\text{-}Init}(\mathsf{sid})$.

Smart Contract Deployment. The server deploys the following two smart contract functions by running:

$$\mathcal{F}_{\mathsf{LED}}.\mathsf{Submit}(\mathsf{sid}, \text{“}\texttt{Deploy}\text{”}\ SC_{\texttt{Req-Rand}}),$$

$$\mathcal{F}_{\mathsf{LED}}.\mathsf{Submit}(\mathsf{sid}, \text{“}\texttt{Deploy}\text{”}\ SC_{\texttt{Req-Fulf}}).$$

Set $\mathsf{reqId} := 0$ and $\mathsf{reqFulf}[i] := \mathsf{false}$ for $\forall i$ and the smart contract functions are:

- `Req-Rand`(x, w) :
  Require$(w = \lambda)$. Set $\mathsf{qID} := (x, \mathsf{reqId})$ and update $\mathsf{reqId} := \mathsf{reqId} + 1$. Append to $\mathsf{LOG}^{\mathsf{sid}}$:
  $$(\text{REQ-RAND}, x, w; \mathsf{qID}).$$
- `Req-Fulf`(qINFO, y, π) :
  - Set $\mathsf{qINP} := \mathsf{qID} := \mathsf{qINFO}$
  - Require(reqFulf[reqId]=false)
  - Require(Verify$(vk, \mathsf{qINP}, (y, \pi))$.
  - Append $(\text{REQ-FULF}, \mathsf{qINFO}, y, \pi; \mathsf{qID}, \mathsf{qINP}, vk)$ to $\mathsf{LOG}^{\mathsf{sid}}$.
  - Obtain callback parameter $\mathsf{param} := x$.
  - Set reqFulf[reqId] = true
  - Perform callback as callback$((\mathsf{qINP}, y, \pi), \mathsf{param})$.

Following protocols are run in parallel by multiple requesters.

Request-Randomness. Requester Q sets $x := \mathsf{param}$, where param is the parameter for requester callback function. Requester smart contract runs $\mathcal{F}_{\mathsf{LED}}.\mathsf{Submit}(\mathsf{sid}, (\text{“}\texttt{Call Req-Rand}\text{”}, x, \lambda))$. Read qID once the request gets appended on $\mathsf{LOG}^{\mathsf{sid}}$.

Fulfillment. During fulfillment $\mathcal{F}_{\mathsf{RLY}}$ (Fig.7) is invoked by anyone with command $(\text{“}\texttt{Fulfill}\text{”}, (\mathsf{sid}, x, \mathsf{qID}))$. $\mathcal{F}_{\mathsf{RLY}}$ calls the server. The server P computes the verifiable output along with proof as $(y, \pi) \leftarrow \mathsf{VRF.Eval}^{\mathsf{H}}(sk, \mathsf{qINP})$ and returns (y, π) to the $\mathcal{F}_{\mathsf{RLY}}$, where $y \in \{0,1\}^\lambda$.

Local Verification. Check that $\mathsf{VRF.Verify}^{\mathsf{H}}(vk, \mathsf{qINP}, (y, \pi)) = 1$ and verify that there exists a tuple $(\text{REQ-FULF}, \mathsf{qINFO}, y, \pi; \mathsf{qID}, \mathsf{qINP}, vk)$ on $\mathsf{LOG}^{\mathsf{sid}}$.

Note: $\mathsf{qINFO} = \mathsf{qINP} = \mathsf{qID}$ always holds for an honest fulfillment of the request with query ID qID.

Fig. 6: VRF-Based Randomness Service $\pi_{\mathsf{R\text{-}VRF}}$ in $(\mathcal{F}_{\mathsf{LED}}, \mathcal{F}_{\mathsf{RLY}})$-model.

prevents precomputation of the VRF output by malicious server. Upon receiving the computation request, the dVRF committee computes partial VRF evaluations on qINP. The relay nodes collect partial evaluations from the Supra dVRF committee, verify the partial evaluations, and then combine them to obtain a verifiable output (y, π), where $y \in \{0,1\}^{\mathsf{len}}$. The relay nodes invoke the fulfillment transaction on $\mathsf{qINFO} = \mathsf{qINP}$ and the output (y, π) to post the output on-chain and send y to the respective requester Q via the callback function in param.

Next, we argue that the Supra dVRF service is a

Setting. The functionality interacts with VRaaS server P, and the ledger functionality $\mathcal{F}_{\mathsf{LED}}$, with $\mathsf{LOG}^{\mathsf{sid}}$.

- Upon $(\text{“}\texttt{Fulfill}\text{”}, (\mathsf{sid}, x, \mathsf{qID}))$ executes the following steps:
  1) Verifies whether $(\text{REQ-RAND}, x; \mathsf{qID})$ exists on $\mathsf{LOG}^{\mathsf{sid}}$ using $\mathcal{F}_{\mathsf{LED}}.\mathsf{Read}(\mathsf{sid})$ command, where $\mathsf{qID} = (x, \mathsf{rID})$. If succeeds, go to the next step.
  2) Set $\mathsf{qINP} := \mathsf{qID} = (x, \mathsf{rID})$.
  3) Verify that the tuple $(\text{REQ-FULF}, \mathsf{qINFO}, \_, \_; \mathsf{qID}, \mathsf{qINP}, vk)$ is not present on $\mathsf{LOG}^{\mathsf{sid}}$.
  4) If it succeeds invoke server P on qINP.
  5) Once P returns (y, π), run $\mathcal{F}_{\mathsf{LED}}.\mathsf{Submit}$ $(\mathsf{sid}, (\text{“}\texttt{Call Req-Fulf}\text{”}, \mathsf{qINFO}, y, \pi))$.

Fig. 7: Ideal Relay Functionality $\mathcal{F}_{\mathsf{RLY}}$.

VRaaS. The output y is unique for every randomness request. Each qID is guaranteed to be unique, even for the same client inputs, due to the unique reqId added by the Supra smart contract in qID. The same argument holds for the uniqueness of qINP. This translates to unique pseudorandom output y due to the pseudorandomness of the GLOW-dVRF protocol, proving Conditions 1 and 2. Condition 3 holds trivially as $\mathsf{qINP} = \mathsf{qINFO}$ and y verifies due to the correctness of the GLOW-dVRF. We can further show that the Chainlink VRF service implements the $\mathcal{F}_{\mathsf{VRaaS}}$ functionality.

Comparison. We briefly compare Chainlink VRF and Supra dVRF services:

1) The VRF protocol is run by a centralized server in Chainlink. In Supra dVRF, it is distributed among a committee of nodes that satisfy an honest majority.
2) In Chainlink, the VRF server acts as the relay node and hence it is blockchain-aware. This is not preferable in a multi-blockchain environment since it is not scalable as the number of blockchains increases. Whereas, in Supra dVRF service there is a separate relay committee that is blockchain aware and keeps track of requests on different blockchains. This allows the Supra dVRF server nodes to remain blockchain agnostic and focus only on dVRF computation. More details can be found in Appendix. D.
3) Supra also reuses the trust assumption of the dVRF committee to validate that the qINP is valid and compute the VRF output after it is deemed valid. Whereas, Chainlink maintains an on-chain commitment to the hash of the request and later matches with it to validate qINP. Supra's approach is more efficient in practice due to avoidance of on-chain commitments.

VII. Vulnerabilities in Existing Protocols

We present analyses of Algorand [ST22], Band-VRF [Bana] and Pyth-VRF [Pyt] and show how they fail to implement $\mathcal{F}_{\mathsf{VRaaS}}$. In Appendix G, we demonstrate vulnerabilities in the DIA xRandom smart contract code.

In the same section, we also formally analyse the vulnerabilities that can occur if Algorand's beacon is not correctly used to generate randomness.

Band VRF Protocol. The Band VRF [Bana] protocol is a VRF-based protocol where the requester (denoted as the client in their protocol) smart contract calls the VRF server smart contract to request randomness. The Band VRF protocol operates across two blockchains via a bridge service and we refer to their protocol [Bana] for more details on this. We observe a vulnerability in the Band VRF provider code [Banb]. One of the input parameters to Band VRF provider (line 111 in [Banb]) is a client seed clientseed. However, the same clientseed cannot be repeated for a given client smart contract. If the same client smart contract makes two randomness requests with the same clientseed, only the first one will be fulfilled (lines 117-119 in [Banb]). An attacker monitors the requests containing the different clientseed values. These requests are public since they are on-chain. The attacker launches a denial-of-service attack by running the same client smart contract with the same clientseed value. If the attacker's transaction gets confirmed first then the client's request will be denied. In our $\mathcal{F}_{\mathsf{VRaaS}}$ functionality, Step 5. cannot be simulated using this protocol as the Band VRF protocol rejects an honest client's request on clientseed if the adversary has already made the same request previously. In the ideal world the honest client's request will always be fulfilled allowing the adversary to distinguish.

Pyth-VRF. The Pyth-VRF protocol works in the commit-and-response paradigm where the server precomputes N random values $(x_1, \ldots, x_N)$ and commits to it on-chain. To request randomness, the requester samples a random number x_U and sends the hash $h_U = \mathsf{H}(x_U)$ to the smart contract. The smart assigns a unique sequence number i to the request and stores the hash value $h_U = \mathsf{H}(x_U)$ and sequence number on-chain. The smart contract increments the counter i so that every request gets assigned a unique sequence number. Upon receiving the sequence number i, the requester invokes the Pyth-VRF server with i to obtain x_i. Given x_i the requester computes the random value as $r := \mathsf{H}(x_i, x_U)$ and invokes the smart contract with x_i to complete the fulfillment process. We describe an attack in Pyth-VRF as follows. The requester preemptively computes the output value r once it receives x_i from the server. If the malicious requester doesn't like r then it does not complete the fulfillment phase. As a result, the output r is not generated on-chain. The smart contract or the server cannot generate the output since the requester's input x_U is secret. In our $\mathcal{F}_{\mathsf{VRaaS}}$ functionality, Steps 11., 13. cannot be simulated using this protocol as the malicious requester can reroll r (denoted as y in $\mathcal{F}_{\mathsf{VRaaS}}$) even when $\mathcal{F}_{\mathsf{SERV}}$ is honest and prevent the fulfillment process if the output is unfavorable.

VIII. Impossibility of Obtaining VRaaS using One Transaction

In our ideal functionality we have two transactions[5] built in. Therefore, any protocol that has one (or less) transaction would fail to satisfy our functionality. However, a protocol using one online transaction would reduce latency and gas costs for the requester. Nevertheless, we show that it is not possible to construct such a protocol satisfying our formalization. Below we provide some intuitions on how such protocols would be vulnerable to attacks in practice.

First, when the fulfillment transaction is not submitted, a malicious requester denies receiving an unfavorable output. It is not possible to distinguish whether the server aborted or the requester is lying.

Second, the request transaction acts as a registration of the client on-chain. So, when the request transaction is not submitted, a malicious requester may just run multiple request sessions with the randomness service in parallel, and then submit the fulfillment request for whichever is more favorable. This way, again a biased output can be obtained. For example, if the requester makes two simultaneous requests, with probability $3/4$ the first bit of the output is 0. Note that the randomness service may be able to link and subsequently stop the conflicting requests from the malicious requester assuming the clients are registered via a public-key infrastructure (PKI). However, the PKI setup on the blockchain requires an additional transaction and the randomness service is not practical if we expect the service to scale for a large number of requester clients since the servers have to keep track of all the registered clients' public keys.

Formally we prove the following theorem.

Theorem 2. *There does not exist an on-chain randomness service, in the $\mathcal{F}_{\mathsf{LED}}$ model, that provides publicly verifiable outputs satisfying unbiasability while using a single on-chain transaction.*

We provide the proof details in Appendix. H where we consider adversarial clients that can bias the output of a protocol if either the request or the fulfillment transaction is not deployed. Our impossibility result assumes that the underlying blockchain is modeled (using $\mathcal{F}_{\mathsf{LED}}$) as an append-only ledger supporting smart contracts. This rules out the Aptos Roll protocol that provides an on-chain randomness service using one transaction by assuming additional properties from the Aptos blockchain. It requires the Aptos blockchain to run a consensus on a block, then generate randomness on the block, and

[5]Online transaction refers to the request randomness transaction, denoted as $\mathsf{Submit}(\mathsf{sid}, (\texttt{"Call Req-Rand"}, x))$ and request fulfillment transaction, denoted $\mathsf{Submit}(\mathsf{sid}, (\texttt{"Call Req-Fulf"}, \mathsf{qINFO}, y, \pi))$ in $\mathcal{F}_{\mathsf{VRaaS}}$, and not the smart-contract deployment transactions.

finally append the block and the randomness using a single on-chain transaction.

IX. Conclusion

Our work performs a comprehensive analysis of on-chain verifiable randomness services. We first formalize the on-chain verifiable randomness in the blockchain setting by introducing the notion of Verifiable Randomness as a Service (VRaaS) – and define it via an ideal functionality. It features a smart contract on a ledger functionality and the VRF provider. We demonstrate that two blockchain transactions are necessary for a secure realization of VRaaS. Towards sufficiency, we show that the VRF based protocol is a VRaaS by proving that it implements our ideal functionality.

We rigorously analyzed existing Web3 randomness services and demonstrated that our framework captures services such as Chainlink VRF and Supra dVRF services. Our investigation also revealed susceptibility to attacks for other designs namely, Band VRF, Pyth VRF, DIA xRandom, and Algorand randomness beacon.

Beyond analyzing existing systems, our VRaaS framework also offers several challenges to consider in the future. These include securely employing VDFs, secure randomness usage patterns in multi-player Web3 applications, and generating private randomness.

References

[a16a] a16zcrypto. 2023 state of crypto report: Introducing the state of crypto index. https://a16zcrypto.com/posts/article/state-of-crypto-report-2023/.

[a16b] a16zcrypto. Introducing the 2022 state of crypto report. https://a16zcrypto.com/posts/article/state-of-crypto-report-a16z-2022/.

[AGR+20] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. Taming callbacks for smart contract modularity. *Proc. ACM Program. Lang.*, 4(OOPSLA):209:1–209:30, 2020.

[Alg] Partnerships of algorand. https://algorandtechnologies.com/about/our-partners/.

[AMMR18] Shashank Agrawal, Payman Mohassel, Pratyay Mukherjee, and Peter Rindal. DiSE: Distributed symmetric-key encryption. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1993–2010. ACM Press, October 2018.

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

[Bana] Band vrf guaranteed integrity on the blockchain. https://www.bandprotocol.com/vrf.

[Banb] Band vrf provider code. https://github.com/bandprotocol/vrf-and-bridge-contracts/blob/34df2ebb75355beffb9ad24efb12c4c3e2c328e5/contracts/vrf/provider_v2/VRFProviderBaseV2.sol#L111.

[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. Springer, Heidelberg, August 2018.

[BCH+20] Christian Badertscher, Ran Canetti, Julia Hesse, Björn Tackmann, and Vassilis Zikas. Universal composition with global subroutines: Capturing global setup within plain UC. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part III*, volume 12552 of *LNCS*, pages 1–30. Springer, Heidelberg, November 2020.

[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*, 2023.

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

[BGK+18] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 913–930. ACM Press, October 2018.

[BL22] Renas Bacho and Julian Loss. On the adaptive security of the threshold BLS signature scheme. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 193–207. ACM Press, November 2022.

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

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 324–356. Springer, Heidelberg, August 2017.

[Bob] Boba network: Distributed randomness beacon. https://boba.network.

[Bol03] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 31–46. Springer, Heidelberg, January 2003.

[Bro20] Richard Gendal Brown. The corda platform: An introduction, 2020. https://www.r3.com/wp-content/uploads/2019/06/corda-platform-whitepaper.pdf. (Accessedon28/05/2020).

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

[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.

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

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

[CGJ19] Arka Rai Choudhuri, Vipul Goyal, and Abhishek Jain. Founding secure computation on blockchains. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 351–380. Springer, Heidelberg, May 2019.

[Chaa] Chainlink random number generation. https://chain.link/vrf.

[Chab] Chainlink vrf: On-chain verifiable randomness. https://blog.chain.link/chainlink-vrf-on-chain-verifiable-randomness/.

[Chac] Use cases of chainlink. https://chain.link/use-cases.

[Chad] Chainlink. Chainlink product update: Q3 2023. https://blog.chain.link/chainlink-product-update-q3-2023/.

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

[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. ACM Press, November 2014.

[CKKR19] Jan Camenisch, Stephan Krenn, Ralf Küsters, and Daniel Rausch. iUC: Flexible universal composability made simple. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part III*, volume 11923 of *LNCS*, pages 191–221. Springer, Heidelberg, December 2019.

[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. Springer, Heidelberg, August 2007.

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

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

[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. Springer, Heidelberg, December 2020.

[CSW22] Ran Canetti, Pratik Sarkar, and Xiao Wang. Triply adaptive UC NIZK. In *ASIACRYPT 2022*, 2022.

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

[dd] DIA developer documentation. Dia dice game. https://docs.diadata.org/products/randomness-oracle/access-the-oracle#example-dice-game.

[DEF+19] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. Multi-party virtual state channels. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 625–656. Springer, Heidelberg, May 2019.

[DIA] Dia developer documents. https://docs.diadata.org/products/randomness-oracle/access-the-oracle.

[DMM+22] Bernardo David, Bernardo Magri, Christian Matt, Jesper Buus Nielsen, and Daniel Tschudi. GearBox: Optimal-size shard committees by leveraging the safety-liveness dichotomy. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 683–696. ACM Press, November 2022.

[DR23] Sourav Das and Ling Ren. Adaptively secure BLS threshold signatures from DDH and co-cdh. *IACR Cryptol. ePrint Arch.*, page 1553, 2023.

[DRa] Drand: Distributed randomness beacon. https://drand.love/.

[Edg] Ben Edgington. Upgrading ethereum. https://eth2book.info/capella/part2/building_blocks/randomness/.

[fil] Filecoin: A decentralized storage network.

[GJKR07] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20(1):51–83, January 2007.

[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 EuroS&P 2021*, 2021.

[GRR+21] Mike Graf, Daniel Rausch, Viktoria Ronge, Christoph Egger, Ralf Küsters, and Dominique Schröder. A security framework for distributed ledgers. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1043–1064. ACM Press, November 2021.

[GVPR18] Sharon Goldberg, Jan Vcelak, Dimitrios Papadopoulos, and Leonid Reyzin. Verifiable random functions (vrfs). *https://datatracker.ietf.org/doc/html/draft-goldbe-vrf-01*, 2018.

[Har] Harmony randomness service. https://docs.harmony.one/home/.

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

[KJG+18] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy*, pages 583–598. IEEE Computer Society Press, May 2018.

[KKK21] Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss. KACHINA - foundations of private smart contracts. In Ralf Küsters and Dave Naumann, editors, *CSF 2021 Computer Security Foundations Symposium*, pages 1–16. IEEE Computer Society Press, 2021.

[KKKZ19] Thomas Kerber, Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas. Ouroboros crypsinous: Privacy-preserving proof-of-stake. In *2019 IEEE Symposium on Security and Privacy*, pages 157–174. IEEE Computer Society Press, May 2019.

[KLM+24] Ranjit Kumaresan, Duc Viet Le, Mohsen Minaei, Srinivasan Raghuraman, Yibin Yang, and Mahdi Zamani. Programmable payment channels. In *ACNS 2024*, 2024.

[KMMM23] Aniket Kate, Easwar Vivek Mangipudi, Siva Maradana, and Pratyay Mukherjee. Flexirand: Output private (distributed) vrfs and application to blockchains. In *ACM CCS 2023*, 2023.

[KT23] Ilan Komargodski and Yoav Tamir. On distributed randomness generation in blockchains. In *Cyber Security, Cryptology, and Machine Learning: 7th International Symposium, CSCML 2023, Be'er Sheva, Israel, June 29–30, 2023, Proceedings*, page 49–64. Springer-Verlag, 2023.

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

[NPR99] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 327–346. Springer, Heidelberg, May 1999.

[PWH+17] Dimitrios Papadopoulos, Duane Wessels, Shumon Huque, Moni Naor, Jan Včelák, Leonid Reyzin, and Sharon Goldberg. Making nsec5 practical for dnssec. Cryptology ePrint Archive, Paper 2017/099, 2017.

[Pyt] Pythvrf: Random number generation on pyth network.

[SJSW19] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. ETHDKG: Distributed key generation with Ethereum smart contracts. Cryptology ePrint Archive, Report 2019/985, 2019. https://eprint.iacr.org/2019/985.

[ST22] Ori Shem-Tov. Usage and best practices for randomness beacon, Sep 2022.

[Supa] Partnerships of supra. https://supraoracles.com/partnerships/.

[Supb] Supra randomness service. https://supraoracles.com/.

[Supc] Supra whitepaper. https://supraoracles.com/docs/SupraOracles-VRF-Service-Whitepaper.pdf.

[Supd] Supra. Total randomness requests served by supra dvrf. https://supra.com/data/dvrf.
[Vee] Veedo — stark-based verifiable delay function.
[WCGF] David Isaac Wolinsky, Henry Corrigan-Gibbs, and Bryan Ford. Scalable anonymous group communication in the anytrust model. https://dedis.cs.yale.edu/dissent/papers/eurosec12-abs/.

APPENDIX A
ADDITIONAL PRELIMINARIES

We first recall the notion of a global random oracle and then show how to use them to upgrade existing VRF protocols to simulatable VRF protocols in the UC framework.

Global Random Oracle. A random oracle (RO) [CJS14, CSW20] H is parameterized by an arbitrary domain and a specified range of $\ell(\lambda)$ bits. 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. The work of [CJS14] introduced the notion of global random oracles (GRO) that allow sharing a random oracle instance among multiple sessions in the UC framework. It supported the idea of observability where the simulator can observe the queries made by a dummy adversary in its session. Then the work of [CDG+18] further extended this idea to incorporate programmability where the simulator can program the output of the global random oracle on a previously unqueried random message to be some specific value. We present the GRO functionality with observability and programmability from [CDG+18] in Fig. 8.

Parameters. Output size function $\ell(\lambda)$.
Variables. Initially empty lists $\mathsf{List_H}$, prog.

1) Upon ("HashQuery", m) from a machine (P, sid) or from the adversary:
   - Look up h such that $(m, h) \in \mathsf{List_H}$. If no such h exists:
     - Draw $h \leftarrow \{0,1\}^{\ell(\lambda)}$.
     - Set $\mathsf{List_H} := \mathsf{List_H} \cup \{(m, h)\}$.
   - Parse m as (s, m').
   - If this query is made by the adversary, or if $s \neq \mathsf{sid}$, then add (s, m, h) to the (initially empty) list of illegitimate queries $\mathcal{Q}_s$.
   - Output ("HashConfirm", h) to the caller.
2) On input ("Observe", sid) from the adversary:
   - If $\mathcal{Q}_{\mathsf{sid}}$ does not exist yet, then set $\mathcal{Q}_{sid} = .$
   - Output ("ListObserve", $\mathcal{Q}_{sid}$) to the adversary.
3) On input ("ProgramRO", m, h) with $h \in \{0,1\}^{\ell(\lambda)}$ from the adversary:
   - If $\exists h' \in \{0,1\}^{\ell(\lambda)}$ such that $(m, h') \in \mathsf{List_H}$ and $h \neq h'$, ignore this input.
   - Set $\mathsf{List_H} := \mathsf{List_H} \cup \{(m, h)\}$ and $\mathsf{prog} := \mathsf{prog} \cup \{m\}$.
   - Output ("ProgramConfirm") to the adversary.
4) On input ("IsProgrammed", m) from a machine (P, sid) or from the adversary:
   - If the input was given by (P, sid), parse m as (s, m'). If $s \neq \mathsf{sid}$, ignore this input.
   - Set $b := (m \in \mathsf{prog})$ and output ("IsProgrammed", b) to the caller.

Fig. 8: Global Random Oracle with Restricted Observability and Restricted Programmability [CDG+18].

A. Simulatable Verifiable Random Function

We demonstrate that the most widely known VRF protocols - BLS/GLOW-VRF [GLOW21], RSA-based [PWH+17], and the Elliptic-Curve based [PWH+17] VRFs are simulatable by modeling the hash function used in the VRF protocols as a programmable random oracle. Later, we show how to make them compatible with the global random oracle.

- **BLS/GLOW-VRF [GLOW21].** The verification key is $vk = g_2^{sk}$ and secret key is sk. The input is x. The proof is $\pi = \mathsf{H}_1(x)^{sk}$ and the output value is $y = \mathsf{H}_2(\pi)$ where $\mathsf{H}_1, \mathsf{H}_2$ are random oracles. The verifier checks $e(\pi, g_2) \stackrel{?}{=} e(\mathsf{H}_1(x), vk)$ and $y \stackrel{?}{=} \mathsf{H}_2(\pi)$.

  Simulatability. To simulate the VRF on unqueried input x to output y', the SimProve programs H_2 s.t. $\mathsf{H}_2(\pi) = y'$. The proof $\pi' = \pi$ remains unchanged. Simulatability follows from the unpredictability of $\pi = \mathsf{H}_1(x)^{sk}$ without querying the VRF on x.

- **RSA-based [PWH+17].** The verification key is (n, e) and the secret key is d. The input is x. The proof is $\pi = \mathsf{H}_1(x)^d \mod n$ and the output value is $y = \mathsf{H}_2(\pi)$ where H_1 is an IETF specified hash function and H_2 is a cryptographic hash function. The verification equation is $\pi^e \mod n \stackrel{?}{=} \mathsf{H}_1(x)$ and $y \stackrel{?}{=} \mathsf{H}_2(\pi)$.

  Simulatability. To simulate the VRF on unqueried input x to output y', SimProve programs H_2 s.t. $\mathsf{H}_2(\pi) = y'$. The proof $\pi' = \pi$ remains unchanged. Simulatability of the VRF follows from the unpredictability of $\pi = \mathsf{H}_1(x)^d \mod n$ without querying the VRF on x.

- **Elliptic-Curve-based [PWH+17].** This is used by Chainlink [Chab]. 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 discrete log proof as: sample $k \leftarrow \mathbb{Z}_q$, set $c = \mathsf{H}_3(g, h, vk, \gamma, g^k, h^k)$ and $s = k - cx \mod q$. Set y as the output and $\pi = (\gamma, c, s)$ 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 $c \stackrel{?}{=} \mathsf{H}_3(g, h, vk, \gamma, u, v)$.

  Simulatability. To simulate the VRF on unqueried input x to output y', SimProve programs H_2 s.t. $\mathsf{H}_2(\gamma^f) = y'$. The proof $\pi' = \pi$ remains unchanged. Simulatability follows from the unpredictability of γ without querying VRF on x.

- **Unforgeable VRF.** One can also build a simulatable VRF from a VRF protocol that satisfies unforge-

ability in the programmable random oracle model. Let $\mathsf{VRF}' = (\mathsf{Gen}', \mathsf{Eval}', \mathsf{Verify}')$ be a verifiable random function satisfying unforgeability. Then we construct a simulatable $\mathsf{VRF} = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$ assuming a programmable random oracle H as follows. Set $\mathsf{Gen} := \mathsf{Gen}'$. We set $\mathsf{VRF.Eval}(sk, x)$: Return output $(y, \pi) := (\mathsf{H}(y'), (y', \pi'))$ where $(y', \pi') := \mathsf{VRF}'.\mathsf{Eval}'(sk, x)$. To verify (x, y, π), parse (y', π'), check that 1) $y = \mathsf{H}(y')$, and 2) $\mathsf{Verify}'(vk, x, (y', \pi'))$. *Simulatability.* To argue simulatability, the simulator programs H on y' to return the specific random y. The adversary cannot distinguish since y' is unforgeable.

These protocols are widely used by Dfinity [HMW18], Chainlink VRF [Chab], and Supra dVRF [Supc]. We show that these protocols can be made simulatable and this is useful for proving simulation-based security of their underlying randomness protocol. However, simulatibility is a stronger notion than pseudorandomness [GLOW21] and it is known [CL07] that simulatibility is hard to achieve in the non-programmable random oracle model. So, we rely on a programmable random oracle to do the task of simulation. We note that the simulatability property is a game-based definition and so we argued it based on the security of a local random oracle. However, since we use this simulatable VRF in our $\mathcal{F}_{\mathsf{VRaaS}}$ functionality we need to assume a GRO as our VRaaS protocol works in the global UC model.

Extending to the GRO model. We need to extend the above simulatable VRFs to the GRO model. In the GRO model, to query a message m to the GRO H, each honest party queries $\mathsf{H}(\text{“}\texttt{IsProgrammed}\text{”}, m)$ to obtain output $(\text{“}\texttt{IsProgrammed}\text{”}, b)$. If b is 1, then the honest party aborts the protocol, else the honest outputs $\mathsf{H}(\text{“}\texttt{HashQuery}\text{”}, m)$. In the real world, if the adversary programs the GRO on message m then the honest party detects it via the “$\texttt{IsProgrammed}$” interface. Whereas, in the ideal world the simulator programs the GRO, and whenever the adversary tries to use the “$\texttt{IsProgrammed}$” interface the simulator returns $b = 0$. This prevents the adversary from programming the GRO without getting detected by the honest party. Whereas, the simulator has the power to program the GRO without getting detected by simulating the response of the “$\texttt{IsProgrammed}$” interface.

APPENDIX B
GLOBAL CLOCK FUNCTIONALITY $\mathcal{F}_{\mathsf{clock}}$

We present the global clock functionality $\mathcal{F}_{\mathsf{clock}}$ from [BMTZ17, BCH+20] in Fig. 9. The clock functionality enforces the parties and other ideal functionalities (sharing the clock) to individually register to the clock for a session sid_C. The parties and the functionalities also have the option to deregister themselves from the clock. The functionality keeps track of time for session sid_C via a variable t_C. Registered parties and functionalities keep track of the time t_C via clock-read command. The functionality increments the value t_C after receiving inputs from every party in the session sid_C.

For a session sid, when honest parties, with identifier id_i(for $i \in [m]$), receive inputs x_i from the environment store it in a sequence along with the timestamp t_i (obtained by sending CLOCK-READ to $\mathcal{F}$) as a timed honest-input sequence $\vec{\mathcal{I}}^H = ((x_1, id_1, t_1), \ldots, (x_m, id_m, t, m))$. Then according to [BMTZ17],

Definition 1. *A $\mathcal{F}_{\mathsf{clock}}$-hybrid protocol Π has a predictable synchronization pattern iff there exists an efficiently computable algorithm $\mathsf{predict\text{-}time}_\Pi(\cdot)$ such that for any possible execution of Π in a session sid (i.e., for any adversary and environment, and any choice of random coins) the following holds: If $\vec{\mathcal{I}}^H = ((x_1, id_1, t_1), \ldots, (x_m, id_m, t, m))$ is the corresponding timed honest-input sequence for this session, then for any $i \in [m-1]$: $\mathsf{predict\text{-}time}_\Pi((x_1, id_1, t_1), \ldots, (x_i, id_i, t_i)) = t_{i+1}$.*

Initialization and state:

- The functionality initializes the set $\mathcal{P} = \emptyset$. It maintains variables d_P for each registered party and a variable t_C for each session specified in a registered party.

Registrations

- Upon receiving $(\text{REGISTER}, \mathsf{sid}_C)$ from some party/functionality P (or from $\mathcal{A}$ on behalf of a corrupted P), set $\mathcal{P} = \mathcal{P} \cup \{P\}$, $d_P = 0$, and if $P = (\cdot, \mathsf{sid} \| \cdot)$ specifies a new sid, then initialize $t_C = 0$. Return $(\text{REGISTER}, \mathsf{sid}, P)$ to the caller.
- Upon receiving $(\text{DE-REGISTER}, \mathsf{sid}_C)$ from some party/functionality $P \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted $P \in \mathcal{P}$), set $\mathcal{P} := \mathcal{P} \setminus \{P\}$ and return $(\text{DE-REGISTER}, \mathsf{sid}, P)$ to the caller
- Upon receiving $(\text{GET-REGISTERED}, \mathsf{sid}_C)$ from $\mathcal{A}$, the functionality returns response $(\text{GET-REGISTERED}, \mathsf{sid}, \mathcal{P})$ to $\mathcal{A}$.

Synchronization

- Upon receiving $(\text{CLOCK-UPDATE}, \mathsf{sid}_C)$ from some party/functionality $P \in \mathcal{P}$ first verify that the dummy party providing the input encodes P as its pid; otherwise, ignore the request. Set $d_P = 1$, execute $Round - Update$, and forward $(\text{CLOCK-UPDATE}, \mathsf{sid}_C, P)$ to $\mathcal{A}$.
- Upon receiving $(\text{CLOCK-READ}, \mathsf{sid}_C)$ from any party/functionality $P = (\cdot, \mathsf{sid} \| \cdot)$, first check that the session identifier sid is a managed session; ignore the request otherwise. Execute $Round - Update$ and return $(\text{CLOCK-READ}, \mathsf{sid}_C, t_C)$ to the requester.

Corruptions

- Upon receiving $(\text{CORRUPT}, \mathsf{sid}_C, P_i)$ from $\mathcal{A}$ corrupting $P_i \in \mathcal{P}$, mark the party as corrupted and execute $Round - Update$. Return $(\text{CORRUPT}, \mathsf{sid}_C, P_i)$ to $\mathcal{A}$.

Procedure Round-Update:

- For each managed session sid do: if $d_P = 1$ for all uncorrupted $P = (\cdot, \mathsf{sid} \| \cdot) \in \mathcal{P}$, then update $t_C = t_C + 1$ and reset $d_P = 0$ for all parties $P = (\cdot, \mathsf{sid} \| \cdot) \in \mathcal{P}$.

Fig. 9: Global Clock functionality $\mathcal{F}_{\mathsf{clock}}$.

APPENDIX C
PROOF OF THEOREM 1

We prove that $\pi_{\text{R-VRF}}$ implements $\mathcal{F}_{\mathsf{VRaaS}}$ by proving Thm. 1. We assume that there exists a PPT environment $\mathcal{Z}$ that selects the inputs for the honest parties and it instructs a PPT adversarial algorithm $\mathcal{A}$ to corrupt a participating party 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, it forwards its view $\text{REAL}_{\pi_{\text{R-VRF}},\mathcal{A},\mathcal{Z}}(1^\lambda)$ to the environment $\mathcal{Z}$. In the ideal world execution of the protocol, we provide a PPT simulator Sim that given access to the adversarial algorithm $\mathcal{A}$ and the functionality $\mathcal{F}_{\mathsf{VRaaS}}$ produces the ideal world adversary view $\text{IDEAL}_{\mathcal{F}_{\mathsf{VRaaS}},\mathsf{Sim},\mathcal{Z}}(1^\lambda)$ and forwards it to the environment $\mathcal{Z}$. According to the UC definition, these two views should be indistinguishable.

Proof. Sim has access to the $\mathcal{F}_{\mathsf{LED}}$ functionality. The ideal world adversary $\mathcal{A}_{\mathsf{LED}}$ for $\mathcal{F}_{\mathsf{LED}}$ is invoked by Sim to setup $\mathcal{F}_{\mathsf{LED}}$. We denote the ideal world adversary for $\mathcal{F}_{\mathsf{SERV}}$ as $\mathsf{Sim}_{\mathsf{P}}$. We exhaustively consider all the corruption cases where $\mathcal{A}$ corrupts a combination of the VRaaS server and the client Q. We prove the security of our protocol in the $\mathcal{F}_{\mathsf{RLY}}$-model where Sim simulates $\mathcal{F}_{\mathsf{RLY}}$. We describe the simulation steps and argue indistinguishability between the real and ideal world execution for the two corruption cases as follows.

Server P is corrupt. The different steps are simulated by Sim and $\mathsf{Sim}_{\mathsf{P}}$ are as follows:

- We assume the parties securely generate the VRF setup string $\text{crs}_{\text{VRF}} \leftarrow \text{VRF.}\mathsf{Setup}(1^\lambda)$.
- **Setup Phase.** Initiate key-registration by invoking $\mathcal{F}_{\mathsf{VRaaS}}$ with ("`Key-Register`"). $\mathcal{F}_{\mathsf{VRaaS}}$ forwards this request to $\mathcal{F}_{\mathsf{SERV}}$ (controlled by Sim via $\mathsf{Sim}_{\mathsf{P}}$). Upon obtaining $(vk, \mathsf{Verify}(\cdot))$ from corrupt server P, verify $\mathsf{Verify}(\cdot)$ and then send it to $\mathcal{F}_{\mathsf{VRaaS}}$. Invoke $\mathcal{F}_{\mathsf{VRaaS}}$ with command ("`Init-Ledger`", sid) to initiate the ledger. Upon receiving `Req-Rand` and `Req-Fulf` from server P check it against $\mathsf{Verify}(\cdot)$ algorithm and forward it to $\mathcal{F}_{\mathsf{VRaaS}}$.
- **Request-Randomness.** If the requester Q is honest then Sim performs nothing, the honest requester directly interacts with $\mathcal{F}_{\mathsf{VRaaS}}$ without the involvement of Sim. If the requester Q is corrupt and the requester smart contract invokes $\mathcal{F}_{\mathsf{LED}}.\mathsf{Submit}(\mathsf{sid}, \text{"}\texttt{Call Req-Rand}\text{"}, x, w)$, then Sim reads qID once the request transaction gets appended. Sim returns (qID, w, Q) to $\mathcal{F}_{\mathsf{VRaaS}}$.
- **Fulfillment.** Sim simulates $\mathcal{F}_{\mathsf{RLY}}$ by verifying $(\text{REQ-RAND}, x, w; \mathsf{qID})$ exists on $\mathsf{LOG}^{\mathsf{sid}}$ and sets $\mathsf{qINFO} := \mathsf{qINP} := \mathsf{qID}$. Sim sends ("`Eval-Req`", qID, x) to $\mathcal{F}_{\mathsf{VRaaS}}$ and sends qINP to server P. When $\mathcal{F}_{\mathsf{VRaaS}}$ invokes Sim to obtain query input just return $(\mathsf{qINP}, \mathsf{qINFO})$. When $\mathcal{F}_{\mathsf{VRaaS}}$ invokes $\mathsf{Sim}_{\mathsf{P}}$ with input ("`Eval`", vk, qINP, w) forward it to server P. When P returns (y, π) then $\mathsf{Sim}_{\mathsf{P}}$ forwards it to $\mathcal{F}_{\mathsf{VRaaS}}$. When $\mathcal{F}_{\mathsf{VRaaS}}$ queries Sim("`Run Fulfillment?`") respond with `Yes`.
- **Local Verification.** To verify an output, Sim invokes $\mathcal{F}_{\mathsf{VRaaS}}$ with it and returns whatever $\mathcal{F}_{\mathsf{VRaaS}}$ outputs.

Indistinguishability Argument. We provide our hybrid argument as follows:

- Hyb_0: Real-world execution of the protocol.
- Hyb_1: Ideal world execution of the protocol. This is the same as the real-world execution of the protocol except if the corrupt server responds with an invalid (y, π) s.t. $\mathsf{Verify}(vk, \mathsf{qINP}, (y, \pi)) = 0$, then simulated $\mathcal{F}_{\mathsf{RLY}}$ forwards it to $\mathcal{F}_{\mathsf{VRaaS}}$ without running `Req-Fulf` by itself. In contrast, in Hyb_0, functionality $\mathcal{F}_{\mathsf{RLY}}$ invokes `Req-Fulf` with it and the verification is performed by the smart contract function `Req-Fulf`.
  The adversary successfully distinguishes between the two hybrids if the smart contract in `Req-Fulf` fails to detect that $\mathsf{Verify}(vk, \mathsf{qINP}, (y, \pi)) = 0$ and uploads an output transaction containing this invalid output where $\mathsf{qINP} = (x, \mathsf{reqId})$. Whereas, in the ideal world, $\mathcal{F}_{\mathsf{VRaaS}}$ detects this and rejects the fulfillment Step 12. A distinguisher distinguishing between the two hybrids breaks the security of $\mathcal{F}_{\mathsf{LED}}$ since the smart contract behaves incorrectly.

Client Q is corrupt and server P is honest. The different steps are simulated by Sim as follows:

- We assume the parties securely generate the VRF setup string $\text{crs}_{\text{VRF}} \leftarrow \text{VRF.}\mathsf{Setup}(1^\lambda)$ where Sim and $\mathsf{Sim}_{\mathsf{P}}$ knows td.
- **Setup Phase.** Initiate key-registration by invoking $\mathcal{F}_{\mathsf{VRaaS}}$ with ("`Key-Register`"). $\mathcal{F}_{\mathsf{VRaaS}}$ forwards this request to $\mathcal{F}_{\mathsf{SERV}}$, which is forwarded to $\mathsf{Sim}_{\mathsf{P}}$. $\mathsf{Sim}_{\mathsf{P}}$ generates $(vk,\ sk) \leftarrow \text{VRF.}\mathsf{SimGen}(1^\lambda, \text{td})$ by invoking the simulator of VRF and returns $(vk, \mathsf{Verify}(\cdot)$ to $\mathcal{F}_{\mathsf{VRaaS}}$. Invoke $\mathcal{F}_{\mathsf{VRaaS}}$ with command ("`Init-Ledger`", sid) to initiate the ledger. Generate `Req-Rand` and `Req-Fulf` and forward it to $\mathcal{F}_{\mathsf{VRaaS}}$.
- **Request-Randomness.** Requester Q is corrupt. If the requester smart contract invokes $\mathcal{F}_{\mathsf{LED}}.\mathsf{Submit}(\mathsf{sid}, \text{"}\texttt{Call Req-Rand}\text{"}, x, w)$, then Sim reads qID once the request transaction gets appended. Sim returns (qID, w, Q) to $\mathcal{F}_{\mathsf{VRaaS}}$.
- **Fulfillment.** The simulator algorithm Sim simulates $\mathcal{F}_{\mathsf{RLY}}$. Sim reads the request $(\text{REQ-RAND}, x; \mathsf{qID})$ on $\mathsf{LOG}^{\mathsf{sid}}$ and invokes $\mathcal{F}_{\mathsf{VRaaS}}$ with the command ("`Eval-Req`", qID, w). Sim sets $\mathsf{qINFO} := \mathsf{qINP} := \mathsf{qID}$. When $\mathcal{F}_{\mathsf{VRaaS}}$ invokes Sim to obtain query input just return $(\mathsf{qINP}, \mathsf{qINFO})$. $\mathcal{F}_{\mathsf{VRaaS}}$ samples $y' \leftarrow$

$\mathsf{Rand}(vk, \mathsf{rCtr}; w)$ where rCtr is the internal request counter of $\mathcal{F}_{\mathsf{VRaaS}}$ corresponding to qID. When $\mathcal{F}_{\mathsf{VRaaS}}$ invokes $\mathcal{F}_{\mathsf{SERV}}(\text{“}\texttt{Req-Proof}\text{”}, vk, \mathsf{qINP}, w, y)$, the request is forwarded to $\mathsf{Sim_P}$. $\mathsf{Sim_P}$ computes simulated proof $\pi' \leftarrow \mathsf{SimProve}(sk, \mathrm{td}, y', \mathsf{qINP}; \mathrm{crs}_{\mathsf{VRF}})$. Sim returns π' to $\mathcal{F}_{\mathsf{VRaaS}}$. In the simulated protocol, $\mathsf{Sim_P}$ runs the honest VRaaS server algorithm on qINP with sk to obtain the output (y', π') and sends it to the simulated $\mathcal{F}_{\mathsf{RLY}}$.

- **Local Verification.** To verify an output, Sim invokes $\mathcal{F}_{\mathsf{VRaaS}}$ with it and returns whatever $\mathcal{F}_{\mathsf{VRaaS}}$ outputs.

Indistinguishability Argument. We provide our hybrid argument as follows:

- Hyb_0: Real-world execution of the protocol.
- Hyb_1: Same as Hyb_0, except $\mathcal{F}_{\mathsf{VRaaS}}$ fulfills the request for party Q by setting $y' \leftarrow \mathsf{Rand}(vk, \mathsf{rCtr}; w)$ and $\mathsf{Sim_P}$ simulates the proof $\pi' \leftarrow \mathsf{SimProve}(sk, \mathrm{td}, y', \mathsf{qINP}; \mathrm{crs}_{\mathsf{VRF}})$.
  Indistinguishability follows due to the simulatability property of the VRF. It also guarantees that the output of $\pi_{\mathsf{R\text{-}VRF}}$ is pseudorandom.
- Hyb_2: Ideal world execution of the protocol. This is the same as Hyb_1, except local verification is performed by running the local verification steps of $\mathcal{F}_{\mathsf{VRaaS}}$ on it. If the output (y, π) on qINP is not registered in the memory of $\mathcal{F}_{\mathsf{VRaaS}}$ then $\mathcal{F}_{\mathsf{VRaaS}}$ sends $\perp$ during the verification process in the ideal world execution. An adversary distinguishing between the two worlds forges an output (y, π) on qINP and gets it registered on $\mathcal{F}_{\mathsf{LED}}$, corresponding to vk, by submitting a tuple of the form $(\textsc{Req-Fulf}, \mathsf{qINFO}, y, \pi; \mathsf{qID}, \mathsf{qINP}, vk)$ on $\mathsf{LOG}^{\mathsf{sid}}$, where $\mathsf{qINFO} := \mathsf{qID} := \mathsf{qINP}$. This means that the VRaaS server was never queried in the ideal world. Such an adversary forges the VRF output on qINP, given vk, and thus it breaks the unforgeability of the VRF primitive, leading to an attack on the simulatability of VRF.

Client is honest and server P is honest. The simulator performs nothing since there are no corruptions.

Client Q and server P are corrupt. Sim acts as a forwarder for messages between the corrupt server and the corrupt client by stimulating $\mathcal{F}_{\mathsf{RLY}}$.

Indistinguishability Argument. The adversary successfully distinguishes between the two hybrids if the smart contract in $\texttt{Req-Fulf}$ fails to detect that $\mathsf{Verify}(vk, \mathsf{qINP}, (y, \pi)) = 0$ and uploads an output transaction containing this invalid output where $\mathsf{qINP} = (x, \mathsf{reqId})$. Whereas in the ideal world, $\mathcal{F}_{\mathsf{VRaaS}}$ detects this and rejects the output in step 12. A distinguisher distinguishing between the two hybrids breaks the security of $\mathcal{F}_{\mathsf{LED}}$ since the smart contract behaves incorrectly. □

APPENDIX D
ADDITIONAL MATERIAL FOR VRF-BASED PROTOCOLS

In this section, we present the supplementary material that discuss different extensions of our VRF-based VRaaS protocol.

Distributing the VRaaS Server. The VRaaS server runs the VRF protocol, that consisting of three subprotocols - $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$. These protocols can be distributed by running a distributed VRF [GVPR18, Supc, GLOW21] protocol based on BLS [BLS01] or DDH-based VRFs. The server is replaced by a committee of n servers out of which at most t servers can be corrupt. The secret key is generated using a distributed key generation (DKG) algorithm [GJKR07]. Upon obtaining the secret key shares, each server outputs a partial evaluation of the input and proof that the partial evaluation is correct. An aggregator party/requester aggregates the partial evaluations from the servers, verifies them, finds $(t+1)$ correct partial evaluations, and computes the final output. Later, the Gen phase was securely implemented using distributed key generation (DKG) algorithms [GJKR07]. The most relevant work is by Galindo et al. [GLOW21] who formalized the security properties and analyzed three constructions. The first construction is a distributed pseudorandom function [AMMR18, NPR99], which is essentially a distributed counterpart of the Goldberg et al. [GVPR18] protocol with an appropriate zero-knowledge proofs and a specific DKG protocol (a variant of Gennaro et al. [GJKR07]) – this is termed as DDH-DVRF. While the computation is very efficient, the size of the final proof is proportional to the number of participants. The second construction they considered is the one that was proposed and also used by Dfinity [HMW18] – this is similar to DDH-DVRF, but uses bilinear pairing to enable a compact proof. However, the use of bilinear groups comes with a cost over discrete log groups (as mentioned later). The construction is very similar to BLS signatures [BLS01] and is used in many places [Clo, Cor, DAO, SJSW19]. Their final construction is called GLOW-DVRF – this was proposed in that paper. GLOW-DVRF uses bilinear pairing for final verification, but Schnorr's proof of exponent for partial verification.

Adaptive Security of $\pi_{\mathsf{R\text{-}VRF}}$. We briefly discuss the adaptive security of $\pi_{\mathsf{R\text{-}VRF}}$ where the adversary can adaptively corrupt servers in the VRaaS committee, nodes implementing $\mathcal{F}_{\mathsf{RLY}}$, and the requesters. We note that adaptive corruption of requesters and the relay nodes does not provide the adversary with any additional benefit since the requester and the relay nodes do not possess any private inputs or private randomness. Hence, adaptive corruption of the relay nodes and the requesters can be trivially simulated, where the simulator simulates the relay nodes and requesters by running the static

simulator, and upon adaptive corruption of the parties, the simulator provides the input and random tape as the simulated internal state. To obtain adaptive security for the server, the VRF.Gen has to be distributed using a DKG algorithm that is secure against adaptive corruption of parties. Similarly, the VRF.Eval algorithm also needs to be distributed using an adaptively secure protocol. The recent work of [DR23] provides an adaptively secure threshold BLS signature [Bol03] scheme that relies on the hardness of DDH and co-CDH in asymmetric pairing group assuming random oracles. Previous protocols [BL22] proved adaptive security for the same, assuming One More Discrete Logarithm in the Algebraic Group Model [DR23] would suffice for our adaptively secure instantiation of the VRaaS server in $\pi_{\mathsf{R\text{-}VRF}}$.

Realizing $\mathcal{F}_{\mathsf{RLY}}$. $\mathcal{F}_{\mathsf{RLY}}$ can be realized by a committee of m nodes satisfying an honest majority. The nodes will have access to the $\mathcal{F}_{\mathsf{LED}}$ functionality and the adversary $\mathcal{A}$ can corrupt at most $t' < \frac{m}{2}$ nodes. The VRaaS server/committee waits for $t'+1$ evaluation requests that match and then run the VRaaS evaluation protocol. A strict-honest majority is required among the relay nodes to ensure that the evaluation request is valid and exists on $\mathsf{LOG}^{\mathsf{sid}}$ since the VRaaS server/committee does not have access to $\mathcal{F}_{\mathsf{LED}}$. Upon evaluating the VRF, the VRaaS server (or the committee implementing $\mathcal{F}_{\mathsf{SERV}}$) sends the output to the relay nodes. We require that any of the relay nodes behave honestly and perform fulfillment to ensure liveness. This is the Anytrust [WCGF] assumption.

Access to ledger functionality $\mathcal{F}_{\mathsf{LED}}$. $\pi_{\mathsf{R\text{-}VRF}}$ assumes that the $\mathcal{F}_{\mathsf{RLY}}$ node, but not the VRaaS server, has access to $\mathcal{F}_{\mathsf{LED}}$ (i.e. blockchain-aware). In practical terms, this means that in a multi-blockchain environment, requesters can seek randomness from the VRaaS server through any blockchain. Each blockchain has an assigned $\mathcal{F}_{\mathsf{RLY}}$ that processes requester requests by forwarding them to the VRaaS server. The VRaaS server computes the output and sends it back to $\mathcal{F}_{\mathsf{RLY}}$, which then posts it on the originating blockchain. Supra dVRF [Supc] implements this. This approach simplifies scalability and allows efficient deployment of the VRaaS service on different blockchains by assigning a $\mathcal{F}_{\mathsf{RLY}}$ node to the blockchain, without modifying the server committee. However, it requires the $\mathcal{F}_{\mathsf{RLY}}$ to be implemented using a committee of nodes satisfying honest majority assumption. Another approach is to remove $\mathcal{F}_{\mathsf{RLY}}$ and allow the server to be blockchain-aware. The server reads the randomness requests from the blockchain, evaluates the output, and then posts it on the blockchain. Chainlink VRF [Chab] implements this. However, this is not preferable since the VRaaS server/committee has to keep track of different blockchains and it is not scalable as the number of blockchains increases. Onboarding a new VRaaS server/committee is also taxing as they have to be blockchain-aware for all the new blockchains.

Next, we argue that there cannot be a secure VRaaS protocol where 1) the VRaaS server does not participate in $\mathcal{F}_{\mathsf{LED}}$, 2) there does not exist any party other than the requester and VRaaS server who participates in $\mathcal{F}_{\mathsf{LED}}$, and 3) there are no other setup assumptions (like access to a broadcast channel) between the VRaaS server and requesters. We prove this by contradiction. Assume that such a protocol exists for on-chain verifiable randomness service between a VRaaS server and a requester where the requester is in charge of uploading the randomness requests and fulfillment/output transactions on $\mathcal{F}_{\mathsf{LED}}$. Since the VRaaS server does not have access to $\mathcal{F}_{\mathsf{LED}}$, it cannot read the randomness requests posted by the requester on $\mathcal{F}_{\mathsf{LED}}$. The only way the VRaaS server can read the requests is when the requester directly sends the request to the VRaaS server. The server evaluates the output and has to send it directly to the requester since the VRaaS server does not have access to $\mathcal{F}_{\mathsf{LED}}$. The requester is in charge of uploading this output as a transaction on $\mathcal{F}_{\mathsf{LED}}$. Such a protocol allows a malicious requester to query the VRaaS server on multiple inputs, and obtain multiple outputs. Then the requester chooses the input-output pair that favors it the most and registers that pair as a transaction on $\mathcal{F}_{\mathsf{LED}}$. The honest VRaaS server cannot detect this due to lack of access to $\mathcal{F}_{\mathsf{LED}}$. This breaks unbiasability of the randomness service.

APPENDIX E
VALIDATION OF qINFO BY REQ-FULF

The fulfillment transaction `Req-Fulf` in $\pi_{\mathsf{R\text{-}VRF}}$ is initiated by $\mathcal{F}_{\mathsf{RLY}}$ or the VRaaS server/committee in Chainlink [Chab]. When invoked with the input qINFO, it generates qID and qINP and verifies qINP against the output (y, π) using the verification key VK. The smart contract implementing $\mathcal{F}_{\mathsf{VRaaS}}$ must ensure the validity of qINFO supplied to `Req-Fulf` w.r.t x and qID. Failure to do so could lead to undesirable consequences, such as sending the verifiable output to the wrong requester, resulting in the incorrect account being charged for the output's generation cost as these are provided inside qINFO. To address this, we suggest the following approaches to validate that qINFO in `Req-Fulf` is correct w.r.t. the specific qID.

1) *On-Chain Storage of qINFO:* On-chain storage reqs[] can be used to store the mapping between qID and x during the randomness request phase as reqs[qID] $:=$ x. During the fulfillment phase, `Req-Fulf` generates qID from qINFO, obtains $x \leftarrow$ reqs[qID] and matches it with qINFO.
2) *On-Chain Commitment to qINFO :* On-chain storage is expensive so it is undesirable to store all of x on-chain. Instead, a hash of x could be stored as reqs[qID] $:=$ H(x) where H is a public hash function.

During fulfillment, `Req-Fulf` generates qID and x from qINFO and checks that $\mathsf{reqs}[\mathsf{qID}] = \mathsf{H}(x)$.

3) *Implicit Validation by VRaaS server+Relay Nodes:* The above solutions either require the smart contract to have access to the blockchain or maintain on-chain storage which can be expensive. We note that in real-world $\mathcal{F}_{\mathsf{SERV}}$ will be honest and we can reuse the trust assumption to validate qINFO. Recall in $\pi_{\mathsf{R\text{-}VRF}}$ (Fig. 6) the server generates qINP and it is either trusted or it is implemented using an honest majority server committee. The protocol can be modified to perform the validation. The relay node modifies qINP to $\mathsf{qINP}' = (\mathsf{qINP}, \mathsf{H}(\mathsf{qINFO}))$ where qINP is generated from qINFO. The relay node committee also has an honest majority assumption and so qINP' is correctly generated. The server computes (y, π) on qINP'. When `Req-Fulf` is invoked with qINFO, it generates qINP' and checks (y, π) w.r.t. to the key VK. If (y, π) verifies then it is guaranteed that qINP' was indeed generated by the server and hence qINFO was validated correctly. We call this technique *implicit validation* and it is more efficient in practice since the validation process reuses the implicit trust assumption of the server and the relay committee.

Appendix F
Additional Related Works

The work of [GRR+21] introduced a general framework for distributed ledgers that captures both private and public ledgers in the iUC model [CKKR19], along with support for smart contracts. They prove that the Bitcoin blockchain, the Ouroboros family [BGK+18, KKKZ19] of blockchains, non-blockchain protocols of Corda [Bro20] and Omniledger [KJG+18] implement their functionality. The work of [CGJ19] introduced the notion of blockchain-active adversaries, where the adversary can understand that it is being rewound in the proof by posting its state on the blockchain. In this model, they prove various impossibilities and show that achieving UC security is impossible for general circuits without setup assumption. Meanwhile, we consider the randomness service function and prove UC security in the programmable random oracle model. The recent work of [KT23] constructed a privacy-preserving smart contract based protocol in the model of [CGJ19] by relying on universally composable non-interactive zero knowledge [CSW22]. The work of [KKK21] considers modeling of privacy-preserving smart contracts in the Universal-Composability model of [Can01] and introduces the Kachina protocol for deploying private smart contracts. However, a simpler ledger functionality suffices for us since privacy is not required from smart contracts, in the context of on-chain randomness service. Gearbox [DMM+22] provides a simple timed ledger functionality. It is compatible with sharding but doesn't support smart contracts.

Appendix G
Vulnerabilities in Existing Services

DIA. The DIA xRandom randomness service operates on a beacon model, where requesters utilize the DRand [DRa] randomness beacon to generate random values. We found a bug in the code provided in their official documentation [DIA]. Given the DIA xRandom randomness service, two players (say P_1 and P_2) want to roll a dice and the player with the higher integer value on the roll wins the game. To do so each P_b ($b \in \{1, 2\}$) samples a seed_b and "commits" to it on-chain by sending it to the smart contract. Note that the commitment does not hide the value of seed_b. Once the two seeds are stored on-chain the DIA xRandom oracle is used to obtain randomness r. The final roll value of each player P_b is considered to be $\mathsf{roll}_b := (r + \mathsf{seed}_b)\%6$ ($\%$ denoting the remainder). If $(\mathsf{roll}_1 = \mathsf{roll}_2)$ then it is a draw. Otherwise, player P_1 is considered the winner if $\mathsf{roll}_1 > \mathsf{roll}_2$ and player P_2 is considered the winner if $\mathsf{roll}_2 > \mathsf{roll}_1$. In the above game, a malicious player P_2 chooses $\mathsf{seed}_2 := (\mathsf{seed}_1 + 1)\%6$ after seeing seed_1 in the commitment phase. P_2 always wins the game with probability $\frac{5}{6}$. This occurs since the roll_1 and roll_2 values are linearly correlated. We propose $\mathsf{roll}_b := \mathsf{H}(r, \mathsf{seed}_b, b)\%6$ to solve this issue as it breaks the correlation in the random oracle model. We also find another vulnerability in their smart contract [dd]. There is no commitment to the start of the Dice game or the end of player entry. Consider if two parties P_1 and P_2, both honest, wish to play the Dice Game. Some third-party Eve can perform a denial-of-service attack by continuously calling either the function `commitPlayer1` or `commitPlayer2`, which will continuously update the value of `latestRoundId`. If done frequently enough, no call to `RollDice` can ever be successfully executed as the game never reaches a state where the beacon for `_round=latestRoundId + 10` has been published, and this stalls the game. To fix this, they need a `startgame()` function which once called prevents `commitPlayer1` or `commitPlayer2` from being called again until `RollDice` has successfully executed.

Algorand. The Algorand randomness service operates on a beacon model, where users rely on the Algorand randomness beacon for generating random values. Best practices for requester protocols are outlined in the Algorand developer documentation, including the use of requester smart contracts to implement these practices. One key practice involves the beacon smart contract storing values for a limited number of rounds before updates occur, and requester protocols should read beacon values before updates. Requester protocols need to

handle scenarios such as beacon downtime, updates, or discontinuation of the beacon public key, which can be complex due to the need for proper auditing. An example scenario involves a client smart contract function, `fallbackrand()`, which uses the next beacon if the current one is unavailable. However, a malicious user could exploit this by delaying transactions, causing the current beacon to be unavailable (since its value is unfavorable) until a later beacon is available. This way `fallbackrand()` uses the later beacon, allowing the party to effectively reroll its randomness, thereby breaking the protocol's unbiasability. In our $\mathcal{F}_{\mathsf{VRaaS}}$ functionality, Step 11. cannot be simulated using this protocol as the requester can reroll y even when $\mathcal{F}_{\mathsf{SERV}}$ is honest. The only way to fix this is to ensure that every beacon will be eventually available and each randomness request is fulfilled with the one specific beacon that it was assigned to. This binds each qID in $\mathcal{F}_{\mathsf{VRaaS}}$ to one specific beacon and hence prevents the adversary from breaking unbiasability. We will discuss in our followup work how to build a VRaaS protocol (that implements $\mathcal{F}_{\mathsf{VRaaS}}$) from beacon-based services.

Appendix H
Proof of Theorem 2

We consider two cases. First, consider an arbitrary protocol that does not deploy the fulfillment transaction. For such protocol, we design an adversarial requester which works as follows

- Assume the protocol returns a bit output y. The requester makes a request and when it obtains the response (y, π) directly from the relay nodes it checks whether $y = 0$, if not then it denies receiving the output. Otherwise, it publishes the value.

Clearly, for such an adversary, in the ideal world, the fulfillment transaction would be executed in Step 13. by the ideal functionality $\mathcal{F}_{\mathsf{VRaaS}}$ – this implies that for any output (y, π), a verification query made in the ideal world by any honest party would always return 1. However, in the real world, when $y = 1$, the verification would fail as the protocol was never completed. So, the adversary would be able to distinguish between the real and ideal world with probability $1/2$ (i.e. whenever the random output y is 1).

In the second case, assume that there is no request transaction. A malicious requester's requests can not be linked. Thus, the malicious requester may provide two requests with signatures with different public keys and the attack works as follows:

- Send two unlinkable requests to the servers. The server returns two values (y_1, π_1) and (y_2, π_2). The servers have run the fulfillment transaction on both values.
- Now, the requester chooses the output whose value is 0 and links it. Essentially, the requester specifically chooses the output corresponding to its identity Q if the output is 0.

In the ideal world, both output pairs (y_1, π_1) and (y_2, π_2) would be linked to Q since a requester can only make queries from the identity that is registered with the functionality. So for any output, the probability of it being 0 would be $1/2$. Whereas, in the real world, the probability of the linked output to be 0 would be $3/4$, because, among four possible combinations, only one pair $(1, 1)$ would not result into a non-zero output. This will be distinguishable with probability $1/4$.

Appendix I
Universally Composable Security

We recall the standard Universal Composability framework of Canetti [Can01], with static and adaptive corruptions, for the two-party setting. And we conclude this section with the definition of $\mathcal{F}$-hybrid model, which is instrumental for security proofs in the UC model.

Static Security in the UC Model. In this model, the real world execution of protocol π is carried out between the honest parties P_1 and P_2 and an adversary $\mathcal{A}$, in the presence of an external entity called the environment $\mathcal{Z}$. All the parties are PPT Turing machines and $\mathcal{Z}$ has an auxiliary information z. At the outset of the protocol the environment initiates the parties with inputs and provides some initial information to $\mathcal{A}$. $\mathcal{Z}$ is allowed to interact with $\mathcal{A}$ throughout the protocol. At the outset of the protocol, $\mathcal{A}$ may or may not corrupt a party. Upon corruption of a party, $\mathcal{A}$ gets access to the internal state and input of that party. From now on the party will behave according to $\mathcal{A}$'s instructions (since we are in the malicious model). At the end of the protocol, the honest parties send their output to $\mathcal{Z}$ while $\mathcal{A}$ outputs $\perp$ on behalf of the corrupted parties and its internal state to $\mathcal{Z}$. We denote the view of $\mathcal{Z}$ as $\textsc{Real}_{\pi,\mathcal{A},\mathcal{Z}}(1^\lambda, z)$.

In the ideal world, we consider the honest parties P_1 and P_2, a PPT adversary Sim, $\mathcal{Z}$ and the functionality $\mathcal{F}$. Sim has a random tape r and security parameter λ. He simulates the role of $\mathcal{A}$ in the ideal world and whenever $\mathcal{A}$ corrupts a party in the real world Sim corrupts that party in the ideal world and gets access to its internal state. Sim invokes the algorithm of $\mathcal{A}$, in his head, in another internal protocol execution where Sim simulates the view of the honest parties to $\mathcal{A}$. We will denote this internal copy of $\mathcal{A}$ as $\mathcal{A}_{\mathsf{Int}}$. Based on the reply of $\mathcal{A}_{\mathsf{Int}}$ in the internal execution, Sim behaves accordingly in the ideal world execution. He extracts the inputs of the corrupted parties in the internal execution and invokes $\mathcal{F}$ in the ideal world with those inputs to obtain the output. In the internal execution he simulates the protocol in such a way that $\mathcal{A}_{\mathsf{Int}}$ obtains that output. At the end of the protocol, $\mathcal{A}_{\mathsf{Int}}$ forwards his view to Sim who forwards it to $\mathcal{Z}$. We denote the view of $\mathcal{Z}$ as $\textsc{Ideal}_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}(1^\lambda, z)$. We say

that a protocol π UC-securely implements a functionality $\mathcal{F}$ in the presence of static adversaries if the real world and ideal world views are indistinguishable.

Definition 2. *Let π be a protocol for computing a functionality $\mathcal{F}$. We say that π UC-securely computes the two party protocol functionality $\mathcal{F}$ in the presence of static adversaries if for every PPT adaptive real-world adversary $\mathcal{A}$ and every environment $\mathcal{Z}$, there exists a PPT ideal-world adversary Sim, such that:*

$$\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(1^\lambda, z) \overset{c}{\approx} \text{IDEAL}_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}(1^\lambda, z)$$

Adaptive Security in the UC Model. In the adaptive setting, $\mathcal{Z}$ can ask the real world adversary $\mathcal{A}$ to corrupt an honest party during the real world execution of the protocol or after the execution completes. During the execution, $\mathcal{A}$ can observe the public transcript of the protocol and based on that he can adaptively corrupt an honest party. Once a party gets corrupted, $\mathcal{A}$ gets access to the input and private randomness of the party, thus controlling the party from thereon. In case of post execution corruption, $\mathcal{A}$ observes the output and the transcript of the protocol, and then he corrupts the honest party to get access to the input and private randomness of the party. After post execution corruption occurs, $\mathcal{A}$ forwards its view to $\mathcal{Z}$. Based on that, $\mathcal{Z}$ constructs its real world view, which we denote as $\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(1^\lambda, z)$.

Similarly, in the ideal world $\mathcal{Z}$ can ask the ideal world adversary Sim to corrupt an honest party during the ideal world execution of the protocol or after the execution completes. When $\mathcal{Z}$ instructs Sim to corrupt an honest party in the ideal world, Sim obtains the input of the honest party, in the ideal world, and he instructs the internal world adversary $\mathcal{A}_{\mathsf{Int}}$ to corrupt the corresponding honest party in the internal world. Recall that Sim simulates the honest parties in the internal execution. When $\mathcal{A}_{\mathsf{Int}}$ corrupts an honest party in the internal world, Sim has to produce a private randomness for the simulated honest party such that it matches with the input of the honest party and the simulated transcript produced by Sim, in the internal world, on behalf of the honest party. Sim provides this matching randomness and the input of the simulated honest party to $\mathcal{A}_{\mathsf{Int}}$ in the internal world. In case of post execution corruption of an honest party, Sim obtains the honest party's input in the ideal world and produces the matching randomness (corresponding to the simulated transcript) in a similar fashion to $\mathcal{A}_{\mathsf{Int}}$ in the internal world. After post execution corruption occurs, $\mathcal{A}$ forwards its view to Sim, who forwards it to $\mathcal{Z}$. Based on that, $\mathcal{Z}$ constructs its ideal world view, which we denote as $\text{IDEAL}_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}(1^\lambda, z)$. We say that a protocol π UC-securely implements a functionality $\mathcal{F}$ in the presence of adaptive adversaries if the real world and ideal world views are indistinguishable.

Definition 3. *Let π be a protocol for computing a functionality $\mathcal{F}$. We say that π UC-securely computes the two party protocol functionality $\mathcal{F}$ in the presence of adaptive adversaries if for every PPT adaptive real-world adversary $\mathcal{A}$ and every environment $\mathcal{Z}$, there exists a PPT ideal-world adversary Sim, such that:*

$$\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(1^\lambda, z) \overset{c}{\approx} \text{IDEAL}_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}(1^\lambda, z)$$

The $\mathcal{F}$-hybrid model.. In order to construct our protocols, we utilize other secure two-party protocols as subprotocols. The standard way of doing this is to work in a "*hybrid model*" where both the parties interact with each other (as in the real model) in the outer protocol and use ideal functionality calls (as in the ideal world) for the subprotocols. The UC composition theorem states that if a protocol ρ UC-securely implements a functionality $\mathcal{F}$, then any execution of ρ in a bigger protocol can be replaced with ideal calls to the functionality $\mathcal{F}$. Specifically, while constructing a protocol π that uses ρ as subprotocol, for securely computing some functionality $\mathcal{F}$, the parties can run π and invoke $\mathcal{F}$. The execution of π that invokes $\mathcal{F}$, for each execution of ρ, is called the $\mathcal{F}$-*hybrid execution of* π and is denoted as $\pi^{\mathcal{F}}$. The hybrid ensemble $\mathsf{Hyb}_{\pi^{\mathcal{F}},\mathcal{A},\mathcal{Z}}(1^\lambda, z)$ describes $\mathcal{Z}$'s output after interacting with $\mathcal{A}$ and the parties running protocol $\pi^{\mathcal{F}}$. Whereas, the execution of π that considers execution of ρ is denoted as π^{ρ}. The hybrid ensemble $\mathsf{Hyb}_{\pi^{\rho},\mathcal{A},\mathcal{Z}}(1^\lambda, z)$ describes $\mathcal{Z}$'s output after interacting with $\mathcal{A}$ and the parties running protocol π^{ρ}. By UC security, the two hybrids $\mathsf{Hyb}_{\pi^{\mathcal{F}},\mathcal{A},\mathcal{Z}}(1^\lambda, z)$ and $\mathsf{Hyb}_{\pi^{\rho},\mathcal{A},\mathcal{Z}}(1^\lambda, z)$ are indistinguishable. This permits replacing executions of ρ, in π, with ideal calls to $\mathcal{F}$ functionality; thereby allowing π to execute in the $\mathcal{F}$-hybrid model. It simplifies the security proof of $\pi^{\mathcal{F}}$ as it can be performed in the $\mathcal{F}$-hybrid model, instead of proving security of ρ within the proof of π^{ρ}.