Supra VRF Service

Supra Research

Web3 applications based on blockchains regularly need access to randomness that is *unbiased*, *unpredictable*, and *publicly verifiable*. A verifiable random function (VRF) protocol satisfies these requirements naturally, and there is a tremendous rise in the use of VRF services. As most blockchains cannot maintain the secret keys required for VRFs, Web3 applications interact with external VRF services via a smart contract where a VRF output is exchanged for a fee. However, a single VRF service node may become a single point of failure. Therefore, the VRF service is deployed in a decentralized fashion as a committee consisting of a number of nodes, each holding only a share of secret – the notion is called distributed VRF (DVRF) and offers additional security properties such as *consistency*, *robustness*, *liveness/availability* and *strong pseudorandomness*.

While this smart contract-based service offers the much-needed public verifiability immediately, it severely limits the way a client can employ the VRF service: the VRF requests cannot be made in advance, *and* the output cannot be reused. This introduces significant latency and monetary overhead. To resolve this we extend the traditional notion of VRF by adding a novel *output-privacy* requirement, in that the VRF output is only revealed to the client; everyone else, including the smart-contract and the service nodes observes blinded/masked values. We call this notion Output-private VRF (PVRF) and incorporate it to Supra's VRF framework as an additional service (like its non-private counterpart this notion too supports decentralization). In our design, we observe a moderate computational overhead of around 2x for VRF service nodes to add output-privacy. The client may decide whether to go for a standard non-private VRF or a output-private VRF based on the requirements.

This document elaborates on the entire VRF service provided by Supra. In particular, it includes the definitions, technical details, security guarantees of Supra's distributed VRF protocol and its output-private counterpart. This document also provides the details of the VRF framework.

Contents

1 Introduction

Verifiable Random Functions. Randomness is a precious resource in computing. With the gigantic rise of blockchain technology and Web3 based applications, the demand for a reliable source of randomness has increased enormously. However, given that the on-chain randomness-generation within a smart contract is an expensive procedure, a natural approach is to delegate this to off-chain computation. Off-chain computations, nevertheless, must be verified on-chain to ensure the integrity of computation. Verifiable random functions enable such functionality. A Verifiable Random Function, V is a keyed deterministic function which, on an input x, outputs a string $y = V_{\mathsf{sk}}(x)$. The secret-key sk is selected uniformly at random. Intuitively, the VRF provides two main security guarantees: (i) *pseudorandomness*, which implies that, as long as the secret-key is hidden, the output is *indistinguishable* from a uniform random string – this ensures both *unpredictability* and *unbiasability*; (ii) *verifiability*, which implies that given x, y and a proof π, anyone can *publicly verify* that y is indeed computed correctly as $V_{\mathsf{sk}}(x)$ – such a proof is produced using the secret-key sk.

Distributed VRFs. If a single node holds the entire VRF secret key, then it becomes a single point of failure – (i) if it is compromised, the adversary knows the key sk, and the VRF output is not unpredictable anymore; (ii) if there's a system/network issue with this node, the computation may not terminate. To mitigate these, in the Supra VRF framework we consider a variant of VRF, called *Distributed* VRF [GLOW21], in that the secret-key is shared among many parties (let us denote them by $P_1, P_2, \ldots, P_n$ and call them VRF clan) using a Shamir's secret sharing scheme [Sha79],[1] done using an appropriate Distributed Key-generation (DKG) protocol. On an input x, each party P_i computes a *partial* evaluation-proof pair (y_i, π_i) using their shares of secret-key sk_i. An aggregator, who may not hold any secret-key, can publicly gather $t < n$ such partial evaluations to aggregate them into the final output y and an accompanying proof π – this procedure is public. Typically, a t out of n system is used, where any $t+1$ parties together hold the secret key (via Shamir's secret-sharing), but no t or less parties can find out the key.[2] In addition to the above guarantees a DVRF provides more guarantees. Namely, it provides (i) *consistency*, which guarantees that any $t+1$ participants would yield the same output y for the same x – this is achieved immediately by Shamir's secret sharing. The distributed procedure is clearly more resilient than a centralized VRF. In particular, a DVRF is resilient to $\leq t$ many Byzantine (a.k.a. malicious) faults, who might attempt to collude. The pseudorandomness naturally extends to a stronger version, in that the adversary corrupts up to t (possibly colluding) nodes and may take part into the execution through them – this is termed (ii) *strong pseudorandomness* by Galindo et al. [GLOW21]. Furthermore, a major advantage of DVRF is to have a (iii) *liveness/availability* guarantee, which ensures guaranteed output delivery as long as the number of faulty/crashed nodes stay within a limit. To guarantee liveness we assume an honest majority $n \geq 2t+1$. Additionally we would require a (iv) *robustness* guarantee, which requires that any incorrect computation by a VRF clan node is detected immediately (as opposed to only at the final verification time). Robustness, and the honest majority assumption together gives liveness as long as there are at most t faulty nodes.

VRF Framework. In the blockchain industry many enterprises offer VRF as a service for a fee, in that the service (consists multiple nodes in case of DVRF) and a user, such as a gaming platform, communicate via a *smart contract*. Here, as detailed in Figure 3, the user makes a randomness request to the VRF service via a smart contract. The smart contract then forms an input tag (INP) of a specific format (for more details on the input formation, see Section 6.2) which is then sent to the VRF service via relay/aggregator nodes. Upon receiving the partial responses from the VRF nodes, the relay/aggregator nodes aggregate and forward the response to the smart contract. The smart contract verifies the response, records the VRF output, invokes the callback function provided by the requester, and pays the VRF service.

Output-private DVRF. However, we identify a limitation for the standard smart-contract based framework as described above. The issue is that the output becomes public, as soon as it appears in the smart contract. This incurs a couple of limitations in its utility: (i) the requester cannot make its request in advance

[1] The definition of DVRF does not, however, require any specific secret sharing scheme. But since all our constructions are based on that we use it as an integral part.

[2] We remark that Shamir's secret sharing provides information theoretic security – this is the strongest possible security, which guarantees that even if the corrupt parties have unbounded computational power, they can not recover the secret-key.

towards having the randomness ready when the play begins. The request has to be *synchronized* with the application. As a result, the use of publicly verifiable external randomness introduces a significant *latency* overhead for the requester – it has to put the play on hold as it initiates and completes the VRF request. (ii) as the output is public, it *cannot be re-used* by the requester in the future (for example using a counter-mode hash[3] to generate multiple random values to be used at different times when needed), which also results in significant overhead w.r.t. gas cost and VRF service fee as the requester has to make individual requests each time a new randomness is required. In a nutshell, this compels the requesting platforms (or user) to carefully design their games/services such that their players/clients cannot exploit the public VRF outputs, and furthermore, the latency and monetary overheads stay affordable.

We overcome these limitations of the VRF service by introducing a novel privacy primitive called *Output Private VRF (PVRF)* and thereby adds significantly more flexibility to the Web3-based VRF services. While maintaining all aforementioned properties of VRF, PVRF additionally ensures that the user alone can observe the VRF output. The smart contract and anybody else can only observe a blinded-yet-verifiable version of the output. In this setting, there is a designated user who alone is allowed to obtain the output $y = V_{\mathsf{sk}}(x)$. In particular, no party within the VRF clan gets to see the output, but only a blinded value, which can only be unblinded by the user. Nevertheless, the threshold computation is similar to that of the DVRF, but now based on the blinded value. The blinded outputs are aggregated publicly (possible only if there are $\geq t+1$ legitimate responses), and then the final blinded y is sent to the user, who then unblind it to obtain y. In this paper we elaborate on this concept further, and put forward a distributed PVRF design by extending our DVRF design. We provide basic intuitive arguments in favor of its security in this document. The full formalization with rigorous provable security analysis is provided in an academic research paper [KMMM23].

Summary of content. In summary, this paper contains the following:

- The core DVRF algorithm we use in our VRF service.
- A new primitive PVRF, and an accompanying design.
- The end-to-end overall VRF service framework.

Our core DVRF algorithm is largely based on the DVRF construction called GLOW, presented by Galindo et al. [GLOW21]. We present the construction using our notations. Our Output-private DVRF construction is achieved extending GLOW using ideas from a threshold oblivious PRF construction by Jarecki et al. [JKKX17]. However, in that paper the main goal was to achieve "input privacy." Instead, we aim to achieve output privacy.

2 Related Work

VRF. The concept of VRF was introduced by Micali, Rabin and Vadhan [MRV99]. They first noticed the similarities between VRFs and *unique* signatures (produces a unique signature for each message) Their construction is based on RSA signatures. Later, this was improved by the work of Dodis and Yampolskiy [DY05] – this construction is based on bilinear pairing and collision resistant hash functions, and is more efficient than Micali et al.'s construction. Feasibility of "theoretically optimal"[4] VRFs was settled by Hofheinz and Jager [HJ16] – as expected, the design is not practical. This was later improved by Kohl [Koh19] and very recently by Niehues [Nie21]. Nir Bitansky [Bit20] explores the relations between VRFs and other cryptographic concepts such as non-interactive zero-knowledge proofs. Post-quantum secure VRFs were explored by Esgin et al. [EKS+21]. In the practical regime, the most important construction was proposed by Goldberg et al. [GVPR18], which is being used by many enterprises such as Algorand and is now in the process

[3] One may use any standard hash function H, such as SHA3, and generates several random numbers as $z_1 = \mathsf{H}(y,1), z_2 = \mathsf{H}(y,2), z_3 = \mathsf{H}(y,3), \ldots$. Of course, once y is public, all z_is are predictable. Looking ahead, using our output-private VRF one could ensure that y stays private to the user. Therefore, the outputs $z_1, z_2, \ldots$ can be used when required. However, without y one can not verify z_i. Therefore, the verification must be deferred to at a later time, say once z_{10} is used, and after that a new VRF request must be made – the overall cost reduces by a factor of 10.

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

of IETF standardization. The VRF design combines a pseudorandom function and a simple zero-knowledge proof of exponent (namely Schnorr's [Sch90]). The designs elaborated in this paper are conceptually related to this approach. In the rest of the paper we refer to this construction as GVPR.

Distributed VRF. Distributed VRF was first considered by the work of Dodis [Dod03], which requires a trusted dealer. Kuchta and Manulis [KM13] proposed a generic construction based on aggregate signatures. However, the most important work was done by Galindo et al. [GLOW21] who formalized the security properties and analyzed three constructions. Crucially, they define a stronger variant of pseudorandomness, called strong pseudorandomness which guarantees pseudorandomness of the output, even when a limited number of partial evaluations are produced by the adversary. The first construction is a variant of distributed PRF [AMMR18, NPR99], which is essentially a distributed counterpart of the GVPR construction with appropriately adjusted zero-knowledge proofs and a specific distributed key-generation 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 construction does satisfy strong pseudorandomness. The second construction they considered is the one which was proposed and also used by Dfinity [HMW18] – this is similar to DDH-DVRF, but uses bilinear pairing to enable a compact proof. However, use of bilinear groups come with a cost over discrete log groups. The construction is very similar to BLS signatures [BLS01] and is used in many places [Clo, Cor, DAO, SJSW19]. Galindo et al. showed that Dfinity-DVRF, while providing compact proofs, does *not* satisfy strong pseudorandomness. Their final construction is called GLOW – this was proposed in that paper. GLOW uses bilinear pairing for final verification, but Schnorr's proof of exponent for partial verification – as a result not only is security improved, but the computation time is also improved by about 2.5x. The only cost is in terms of the size of partial proofs, which increases a little, but still stays well within the allowed bandwidth. In our VRF service we deploy this construction and an output-private variant of the same.

VRF Applications. Many blockchain services use VRFs internally as a crucial source of randomness. For example, Cardano [Car] and Polkadot [Pol] implement VRFs for block production. Dfinity [HMW18] uses a DVRF (namely Dfinity-DVRF, as mentioned above) for producing a decentralized random beacon. Chainlink offers the most popular VRF service. Their VRF architecture essentially deploys the GVPR algorithm along with some optimizations. However, from their description [Cha] it seems that their VRF secret-key is not decentralized (in other words, they do not use a DVRF), and therefore, their scheme is susceptible to a *single point of failure*. This means that if a specific node is compromised, the entire secret-key is known to the attacker. Subsequently, given the secret-key, the output of the VRF is completely predictable. This paper describes a VRF service which is inherently decentralized and thereby *not* vulnerable to a single point of failure. Furthermore, even in the distributed setting, the underlying algorithms (based on GLOW) provides stronger provable security assurances than Dfinity-DVRF.

3 Preliminaries

Notation. We use $\mathbb{N}$ to denote the set of positive integers, $\mathbb{Z}$ to denote the set of all integers and $[n]$ to denote the set $\{1, 2, \ldots, n\}$ (for $n \in \mathbb{N}$). A tuple of values is denoted by the vector notation $\vec{v} = (v_1, v_2 \ldots)$. We denote the security parameter by κ. A security parameter is essentially the quantification of security required from a scheme. For example, a security parameter can be set to 128 or 256 to get 128/256-bit security. We assume that every algorithm takes κ as an implicit input and all definitions work for any sufficiently large choice of $\kappa \in \mathbb{N}$. We will omit mentioning the security parameter explicitly except in a few places. We use $\mathsf{negl}(\kappa)$ to denote a negligible function in the security parameter; a function $f : \mathbb{N} \to \mathbb{N}$ is considered negligible if for every polynomial p, it holds that $f(n) < 1/p(n)$ for all large enough values of n.

We work on groups $\mathbb{G}$ of prime order p. For κ-bit security we require that p be a κ-bit prime. When we say that a group $\mathbb{G}$ is a public parameter, we mean to say that its order is known and some generator is agreed upon.

Throughout the paper we use the symbol $\perp$ to denote invalidity; in particular, if any algorithm returns $\perp$ that means the algorithm failed or detected an error in the process. We use $\mathsf{D}(x) =: y$ or $y := \mathsf{D}(x)$ to denote the evaluation of a deterministic algorithm D on input x to produce output y. Often we use $x := \mathsf{val}$ to

denote the assignment of a value val to the variable x. We write $\mathsf{R}(x) \to y$ or $y \leftarrow \mathsf{R}(x)$ to denote evaluation of a randomized algorithm R on input x to produce output y. R can be determinized as $\mathsf{R}(x; r) =: y$, where r is the explicit randomness used by R. For a boolean condition $b := (x = y)$ we denote that if $x = y$ satisfies then b gets the value 1, otherwise if $x \neq y$ and the check fails it gets the value 0.

Adversarial Model. We are in a distributed setting, where the total number of nodes/parties is denoted by n. Parties can be corrupt in an arbitrary malicious manner and also may collude with each other – this is also known as Byzantine corruption. For any threshold computation, we assume that at least $t+1$ participants are needed – looking ahead, this is guaranteed by t out of n Shamir's secret sharing, in that any $t+1$ parties can reconstruct the secret. We assume that at most $f \leq t$ parties can be maliciously corrupt. Looking ahead, we always assume $f = t < \lceil n/2 \rceil$ – this restriction is required for the liveness/availability.[5] Henceforth we use f and t interchangeably, both denoting the "corruption threshold" as well as "reconstruction threshold." We assume a static model of corruption, where the set of corrupt parties are non-adaptive, which implies the set of corrupt parties remain the same throughout the protocol execution.

Network Model. Each protocol presented here uses a sub-protocol for distributed key generation (DKG). We do not specify the details of the DKG protocol. Once the DKG is done, only point-to-point channels are required.

Families, Clans, Tribes. We use specific Supra terminologies for different groups of participants, classified according to their fault tolerances. Any group of parties has two parameters, n denoting the number of all parties and f denoting the maximum number of corrupt parties. Families refer to a group with *dishonest majority or any-trust*, which has at least $f + 1$ parties; clans refer to one with *honest simple majority* that is with at least $2f + 1$ parties; and tribes to one with *honest super majority* that has at least $3f + 1$ parties. In this paper, the parties executing the distributed VRF protocols are in the honest simple majority setting and hence are referred to as the VRF clan (as mentioned above). In Supra VRF Framework (Section 6), we will be using relay/aggregator network, which requires an any-trust setting and thus is a family. The optimized version uses Supra SMR Service which requires a honest super majority setting.

Polynomial Interpolation. A polynomial $P(x)$ over a finite field $\mathbb{F}$ of degree t can be expressed as $P(x) = c_0 + c_1 x + c_2 x^2 \dots c_t x^t$, where each coefficient is in $\mathbb{F}$. Given any $\ell \geq t + 1$ evaluation points $P(j_1), \dots, P(j_\ell)$, where $S = \{j_1, \dots, j_\ell\}$ there are scalars $\lambda_{i,j,S}$ such that for any $i \in \mathbb{N}$:

$$P(i) = \sum_j \lambda_{i,j,S} P(j)$$

Importantly, the Lagrange coefficient $\lambda_{i,j,S}$ corresponding to j depends on the set S and the evaluation point at i and nothing else.

3.1 Building Blocks

Discrete Log groups. All constructions are based on cyclic groups of prime order, where the discrete logarithm is hard. For a group $\mathbb{G}$ with any elements g and $h = g^x$, we denote $x = \mathsf{DLOG}_g(h)$ to denote the discrete logarithm of h to the base g.

Bilinear Pairing. Our construction relies on bilinear pairing. We consider three groups $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$, among which the source groups $\mathbb{G}_0$, $\mathbb{G}_1$ and $\mathbb{G}_T$ all are multiplicative groups[6] of prime order p. The corresponding generators are denoted by g_0, g_1 and g_T. There is an efficiently computable map $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$ which is

- **bilinear:** for any $a, b \in \mathbb{Z}_p$:

$$e(g_0^a, g_1^b) = e(g_0, g_1^b)^a = e(g_0^a, g_1)^b = e(g_0, g_1)^{ab}$$

[5] This implies that even if t parties are corrupt, there will be at least $t + 1$ honest nodes available to finish the computation.

[6] In many places $\mathbb{G}_0, \mathbb{G}_1$ are considered as additive groups, in that case one has to replace the multiplication with addition and subsequently exponentiation with scalar multiplication in our presentation

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

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

Shamir's Secret Sharing [Sha79]. We use Shamir's secret sharing scheme. Let p be a prime, and n, t be positive integers such that $t < n$. An (n, t, p)-Shamir's Secret Sharing ((n, t, p)-SSS for short) scheme is a pair of algorithms $(\mathsf{Share}, \mathsf{Recon})$ that work as follows.

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

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

NIZK for Knowledge of Exponent (Schnorr's proof [Sch90]). Our construction uses non-interactive zero-knowledge (NIZK) proof for knowledge of exponents. In particular, given a cyclic group $\mathbb{G}$ of prime order p, an instance $\mathsf{inst} = (g, h)$ and witness $\mathsf{wit} = k$ such that $k = \mathsf{DLOG}_g(h)$. Also consider a hash function $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_p$. So the set of public parameters is defined as $\mathsf{pp} := (\mathsf{H}, \mathbb{G})$, which is provided as an input to all algorithms implicitly. Then the proof system consists of the following two algorithms:

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

NIZK for Equality of Discrete Log [CP93]. Our construction uses non-interactive zero-knowledge (NIZK) proof for equality of discrete logarithms, which is quite similar to the above proof. We highlight the crucial differences in blue. In particular, given a cyclic group $\mathbb{G}$ of prime order p, an instance of the language consists of the statement $\mathsf{inst} = (g, h, x, y)$ and witness $\mathsf{wit} = k$ such that $k = \mathsf{DLOG}_g(x) = \mathsf{DLOG}_h(y)$. Also consider a hash function $\mathsf{H} : \{0,1\}^* \to \mathbb{Z}_p$. So the set of public parameters is defined as $\mathsf{pp} := (\mathsf{H}, \mathbb{G})$, which is provided as an input to all algorithms implicitly. Then the proof system consists of the following two algorithms:

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

3.2 (Distributed) Verifiable Random Functions

A verifiable random function scheme implements a *deterministic* keyed function $V : \{0,1\}^\kappa \times \{0,1\}^* \to \{0,1\}^\gamma$. It yields pseudorandom strings. More precisely, the key $\mathsf{k} \leftarrow_\$ \{0,1\}^\kappa$ is chosen uniformly at random; then for any input $x \in \{0,1\}^*$, the output $y := V_\mathsf{k}(x)$ is pseudorandom. Additionally, the scheme enables a proof system for anyone to *publicly* verify whether y was computed correctly, given x and a proof. We assume that all algorithms have access to public parameters.

Definition 1 (VRF)**.** A Verifiable Random Function (VRF) consists of three algorithms $(\mathsf{Keygen}, \mathsf{Eval}, \mathsf{Verify})$ with the following description:

- $\mathsf{Keygen}(1^\kappa) \to (\mathsf{sk}, \mathsf{vk})$. The randomized key-generation algorithm takes the security parameter as input and outputs a secret-key, verification-key pair.
- $\mathsf{Eval}(\mathsf{sk}, \mathsf{vk}, x) \to (y, \pi)$. The evaluation algorithm takes the key-pair and any input x to produce an output y and a proof of correct evaluation π. For a VRF function V, we denote $y = V_\mathsf{sk}(x)$.
- $\mathsf{Verify}(\mathsf{vk}, (x, y, \pi)) =: 1/0$. The verification algorithm takes the verification key and a triple of input, output and proof to check whether the output was correctly produced from the input executing the evaluation algorithm. If the check succeeds it outputs 1, otherwise it outputs 0.

They should satisfy the following requirements:

> **Correctness:** It simply requires that a correctly generated value $y = \mathsf{Eval}(\mathsf{sk}, \mathsf{vk}, x)$ always verifies successfully.
>
> **Uniqueness:** If (x, y, π) verifies successfully, then y is uniquely determined by $y = V_\mathsf{sk}(x)$ – no other $y' \neq y$ would verify successfully.
>
> **Pseudorandomness:** For any given input x, if $(y, \pi) \leftarrow \mathsf{Eval}(\mathsf{sk}, \mathsf{vk}, x)$, then y is pseudorandom to anyone who does not know sk. Furthermore, this is true even when many tuples $(x_1, y_1, \pi_1), (x_2, y_2, \pi_2), \ldots$ are provided such that $(y_i, \pi_i) \leftarrow \mathsf{Eval}(\mathsf{sk}, \mathsf{vk}, x_i)$ and $x_i \neq x$ for any i.

A distributed VRF protocol is executed between a number of mutually distrusting parties. To compute a VRF function $V_\mathsf{sk}(x)$ all participants need to know the input x. Then they interact, after which everyone gets the output $y := V(x)$, if the protocol succeeds. Otherwise, if the execution fails no one receives any output. Note that, participants can behave in an arbitrarily malicious (a.k.a. Byzantine) manner.

Definition 2 (DVRF)**.** A distributed VRF (DVRF) is executed between n parties $P_1, \ldots, P_n$ with a threshold[7] $t < n$. In contrast with the standard VRF defined above, it has the key-generation and evaluation

[7] Recall that, since we assume $f = t$, this denotes both the reconstruction threshold and the corruption threshold.

procedures distributed. We do not provide specific details of the distributed key-generation (DKG) procedure here, instead we just mention the input and output. For the evaluation protocol, we describe a partial evaluation algorithm which is run by each participant using its own secret share, and a public aggregation procedure which aggregates the partial evaluations.

- The distributed key-generation protocol DKG: In this protocol no party has any input barring the parameters $(1^\kappa, n, t)$ and the public parameters. Each party uses its own internal random coins to generate messages which are then sent to the SMR channel. In the end, if the protocol execution succeeds, a set of qualified participants, $Q \subseteq \{P_1, \ldots, P_n\}$ of size $t < |Q| = q \leq n$ gets the verification key tuple $\vec{\mathsf{vk}} := (\mathsf{vk}_1, \mathsf{vk}_2, \ldots, \mathsf{vk}_q)$ and a public key pk and each party P_i gets a corresponding secret key-share sk_i.
- Partial Evaluation $\mathsf{Part.Eval}(\mathsf{sk}_i, \mathsf{vk}_i, x) \to (y_i, \pi_i)$: This randomized algorithm is run by each party P_i, who uses their key-pair $(\mathsf{sk}_i, \mathsf{vk}_i)$ to produce a partial output y_i and a zero-knowledge proof of correct partial computation π_i.
- Aggregation algorithm $\mathsf{Aggregate}(\mathsf{pk}, \vec{\mathsf{vk}}, x, (y_1, \pi_1), (y_2, \pi_2), \ldots, (y_\ell, \pi_\ell)) =: (y, \pi)$: This algorithm gathers ℓ many partial evaluations from the participating set S and proofs of correctness. If $\ell \leq t$ (that is $|S| \leq t$), the algorithm just returns $\perp$ to denote failure. Otherwise, if $\ell > t$ this algorithm proceeds to verify each triple $(\mathsf{vk}_i, y_i, \pi_i)$ with respect to the public key pk. If less than $t+1$ verification succeeds, then this algorithm returns $\perp$; otherwise this algorithm returns a pair (y, π).
- The algorithm Verify is non-interactive and stays the same as the standard VRF.

Apart from satisfying the standard correctness guarantees described in Definition 1, they should satisfy the following requirements:

Consistency. A successful execution of the protocol should produce the same value. $y = V_{\mathsf{sk}}(x)$, irrespective of the participating set S, as long as there are at least $t+1$ participants (that is $|S| \geq t+1$) who provide correct partial evaluations.

Availability/Liveness. Even at most t parties are malicious corrupt and collude, they can not prevent the honest parties from obtaining the output. This is also known as *guaranteed output delivery* in the MPC literature (see, for example, Cohen and Lindell [CL14] for a formal definition).

Robustness: Assume that there is a malicious adversary which corrupts at most t participants within the VRF clan throughout the protocol. Even in this case, if the aggregation does not fail (that is returns a value which is not $\perp$) despite running on adversarial partial evaluations, then the output must verify successfully.

Strong pseudorandomness. Even if at most t participants are maliciously corrupt and collude throughout the protocol, they are not able to distinguish between an output generated by applying the VRF correctly $y = V_{\mathsf{sk}}(x)$ on an input and a uniformly random value from the output range – this holds even if the colluding participants take part in several executions on inputs $x_1, x_2, \ldots$ such that $x \neq x_i$. Note that, through these executions, the colluding parties (possibly) learn not only $y_1 = V_{\mathsf{sk}}(x_1), y_2 = V_{\mathsf{sk}}(x_2), \ldots$, but also all the partial evaluations too. This additional information makes the security requirement stronger than the non-interactive case (and hence the qualification "strong").

4 Supra core VRF Construction

In this section we describe the core algorithm used for our VRF, which we call $\mathsf{Core\text{-}VRF}$. As mentioned earlier, our core VRF algorithm is based on a construction put forward by Galindo et al. [GLOW21] called GLOW. However, in contrast to their construction, we do not use a concrete DKG protocol. The rest of the protocol is basically the same as GLOW. Before detailing the protocol we describe the ingredients/parameters:

Ingredients

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

The protocol is detailed in Figure 1.

Core VRF Security. The *consistency* property is immediate from Shamir's secret sharing. The *availability/liveness* also follows directly from the bound $f = t < \lceil n/2 \rceil$. To see *robustness* we notice that during the aggregation procedure, each partial evaluation is verified individually. Since we use a sound NIZK system for partial evaluation, a successful verification at this stage implies that each partial evaluation computes correctly as $\mathsf{H}_1(x)^{s_i}$. Therefore, the aggregated value too is indeed equal to $\mathsf{H}_1(x)^s$. Once we are convinced about this, the next step just follows from the correctness of the BLS signature.

Finally, *strong pseudorandomness* intuitively follows from the fact that, even if an attacker obtains up to t partial evaluation on an input x, due to the security of secret sharing, the remaining partial evaluations remain completely hidden/unpredictable – this leads to the unpredictability of the final proof π (which is equal to z). Subsequently, the VRF output which is a hash value of the proof $\mathsf{H}_2(\pi)$ remains pseudorandom. However, a formal proof requires rigorous reduction to the underlying hardness assumption over the Type-3 bilinear group. In particular, as shown by Galindo et al. [GLOW21], the specific assumption needed here is co-CDH [BDN18]. For formal proofs we refer to the Galindo et al. paper [GLOW21].

5 Output-private DVRF (PVRF)

In this section we introduce the concept of Output-private DVRF (PVRF). Intuitively, this is a distributed protocol, in that a user U who holds the input x, sends a request to the VRF clan, who then runs the protocol, at the end of which a blinded output is given to U, who then unblinds the output. Looking ahead, in our construction when the user sends the request to the VRF clan, along with the input x, a blinded input is also given, and the same blinding value is used later to unblind the output.

Definition 3 (PVRF)**.** An Output-private DVRF is executed between n parties $P_1, \ldots, P_n$ with a threshold $t < n$ and a user U. In contrast to DVRF, the syntax of PVRF has slightly different partial evaluation and aggregation. Additionally, there are two non-interactive algorithms Blind and $\mathsf{Unblind}$, both executed by the user U. Major changes are highlighted by blue.

- The distributed key-generation protocol DKG: This is exactly the same as in the DVRF definition. In this protocol no party has any input barring the parameters $(1^\kappa, n, t)$ and the public parameters. Each party uses its own internal random coins to generate messages which are then sent to the SMR channel. In the end, if the protocol execution succeeds, a set of qualified participants, $Q \subseteq \{P_1, \ldots, P_n\}$ of size $t < |Q| = q \leq n$ gets the verification key tuple $\vec{\mathsf{vk}} := (\mathsf{vk}_1, \mathsf{vk}_2, \ldots, \mathsf{vk}_q)$ and a public key pk and each party P_i gets a corresponding secret key-share sk_i.
- Blinding $\mathsf{Blind}(x) \to (x, v, \pi^*, \mathsf{st})$. This algorithm is a randomized procedure which outputs a blinded input v an accompanying proof of correctness π^* and a state st (to be used later during the unblinding procedure).
- Partial Evaluation $\mathsf{Part.Eval}(\mathsf{sk}_i, \mathsf{vk}_i, (x, v, \pi^*)) \to (z_i, \pi_i)$. This randomized algorithm is run by each party P_i with a common triple (x, v, π^*). Each party first verifies the pair (x, v, π^*), and if it fails, then it outputs $\perp$. Otherwise, each party uses its key-pair $(\mathsf{sk}_i, \mathsf{vk}_i)$ to produce a blinded partial output z_i and a proof of correct partial computation π_i.

- $\mathsf{DKG}(1^\kappa, n, t)$. Our distributed key-generation protocol takes the system parameters as input.

  Once it concludes, parties in Q get the tuple of verification key $\vec{\mathsf{vk}} = (\mathsf{vk}_1, \mathsf{vk}_2 \ldots, \mathsf{vk}_q)$ and a public key pk and each party receives a secret key sk_i where:

  - For all $i \in [q]$: $\mathsf{sk}_i = s_i \in \mathbb{Z}_p$ such that $(s_1 \ldots, s_q) = \mathsf{Sharing}(s, q, t)$ for a uniformly random $s \in \mathbb{Z}_p$.
  - For all $i \in [n]$: $\mathsf{vk}_i = g_1^{s_i} \in \mathbb{G}_1$.
  - $\mathsf{pk} = g_2^s \in \mathbb{G}_2$.

- $\mathsf{Part.Eval}(\mathsf{sk}_i, \mathsf{vk}_i, x) \to (y_i, \pi_i)$. The randomized partial evaluation algorithm is executed by each party P_i on a given input $x \in \{0,1\}^*$. It uses the key-share $s_i = \mathsf{sk}_i$ and the public verification key $\mathsf{vk}_i = g_1^{s_i}$ and executes the following steps:

  - Compute $y_i := \mathsf{H}_1(x)^{s_i} \in \mathbb{G}_1$.
  - Compute the proof $\pi_i \leftarrow \mathsf{EqProve}(\mathsf{inst}, \mathsf{wit})$ where $\mathsf{inst} = (g_1, \mathsf{H}_1(x), \mathsf{vk}_i, y_i)$ and $\mathsf{wit} = s_i$.
  - Output (y_i, π_i).

- $\mathsf{Aggregate}(\vec{\mathsf{vk}}, x, \{(y_i, \pi_i)\}_{i \in S}) =: (y, \pi)$. The aggregation algorithm gathers ℓ (y_i, π_i) pairs (say $|S| = \ell$) and if $\ell \leq t$, then output $\perp$. Otherwise, it executes the following steps for each $i \in [\ell]$:

  - Check whether for each $i \in S$ if $\mathsf{EqVer}((g_1, \mathsf{H}_1(x), \mathsf{vk}_i, y_i), \pi_i)$ succeeds (that is, outputs 1). Record the successful (y_i, π_i) pair in a set T. If the number of successful partial verification is *insufficient*, that is $|T| \leq t$, then output $\perp$. Otherwise go to the next step:
  - Compute the aggregated value as $z := \prod_{j \in T} y_j^{\lambda_{0,j,T}} \in \mathbb{G}_1$ using Lagrange Interpolation in the exponent. In other words, run the *linear* Recon algorithm in the exponent to reconstruct the secret as $z = \mathsf{H}_1(x)^s \in \mathbb{G}_1$.
  - Compute the accompanying proof as $\pi := z \in \mathbb{G}_1$ and the aggregated value as $y := \mathsf{H}_2(z) \in \{0,1\}^\gamma$.
  - Output (y, π).

- $\mathsf{Verify}(\mathsf{pk}, (x, y, \pi)) =: 1/0$: The verification algorithm takes the public key pk and a pair (y, π) as input is executed by any party as follows:

  - Output result of the check $(e(\mathsf{H}_1(x), \mathsf{pk}) = e(\pi, g_2)) \wedge (\mathsf{H}_2(\pi) = y))$.

Figure 1: The Core-VRF Construction based on GLOW

- Aggregation algorithm $\mathsf{Aggregate}(\mathsf{pk}, \vec{\mathsf{vk}}, v, (z_1, \pi_1), (z_2, \pi_2), \ldots, (z_\ell, \pi_\ell)) =: z$. This algorithm gathers ℓ many *blinded* partial evaluations from the participating set S and proofs of correctness with respect to the value v. If $\ell \leq t$ (that is $|S| \leq t$), the algorithm just returns $\perp$ to denote failure. Otherwise, if $\ell > t$ this algorithm proceeds to verify each triple $(\mathsf{vk}_i, z_i, \pi_i)$ with respect to the public key pk. If any verification fails, then this algorithm returns $\perp$; otherwise this algorithm returns a blinded output z.
- The algorithm $\mathsf{PreVer}(\mathsf{vk}, (v, z)) \to 1/0$ is non-interactive is similar to the standard DVRF, but now with respect to the blinded values.
- Unblinding $\mathsf{Unblind}(\mathsf{st}, z) \to (y, \pi)$. The unblinding algorithm uses a state information (ideally generated during the blinding procedure) to unblind the value z to obtain the output y and also an accompanying proof π.
- The algorithm Verify is non-interactive and stays the same as the standard VRF.

The requirements such as **correctness**, **availability**, **consistency** are the same as DVRF. The **pseudorandomness** and **robustness** are now changed slightly,[8] in that the user U might also be corrupt (and collude) along with up to t parties within the VRF clan. Finally, there's an additional requirement, namely **output-privacy**. Intuitively output-privacy ensures that, even if the adversary corrupts up to t servers, it can not learn anything about the VRF output y from the communication transcript and the public values. In the paper [KMMM23] we provide a fully formal definition following the strong universal composability framework [Can01].

5.1 Supra PVRF Construction.

We describe our PVRF construction in Fig. 2. The construction combined ideas from GLOW [GLOW21] and the threshold oblivious VRF construction of Jarecki et al. [JKKX17]. We show that this construction satisfies our new PVRF notion (Definition 3), which guarantees output privacy. The changes from the DVRF construction (Fig. 1) are highlighted in blue. First we describe the ingredients/parameters for the construction below.

Ingredients

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

PVRF Construction Security. Consistency, availability and correctness can be argued along the line of the DVRF construction. For robustness, the setting slightly changes here, whence a new scenario arises: in particular, what if the user is corrupt, colluding with t other parties and the triple (x, v, π^*) is not a legitimate one, which means the verification within $\mathsf{Part.Eval}$ fails. In this case, to argue that robustness still holds, we observe that within the qualifying set there exists at least one honest party, who can detect that the triple is indeed illegitimate and output $\perp$. Therefore, the adversarial parties would not have sufficient partial evaluations to construct the final output. To prove pseudorandomness one needs to rely on a different,

[8] Notably, we do not consider a strong pseudorandomness, because of a technical reason – all partial evaluations are hidden by the blinding factor, and hence do not give any additional information to the adversary.

- The DKG protocol is exactly the same as the DVRF (Figure 1), and hence is skipped.
- $\mathsf{Blind}(x) \to (x, v, \pi^*, \mathsf{st})$. On input x, the blinding algorithm works as follows:
  - Choose a random value $\rho \leftarrow_{\$} \mathbb{Z}_p$ and define $\mathsf{st} := \rho$.
  - Compute $v := h^{\rho}$ where $h := \mathsf{H}_1(x)$.
  - Compute a NIZK proof $\pi^* \leftarrow \mathsf{KExpProve}((h, v), \rho)$ with the statement (h, v) and witness ρ.
  - Output $(x, v, \pi^*, \mathsf{st})$
- $\mathsf{Part.Eval}(\mathsf{sk}_i, \mathsf{vk}_i, (x, v, \pi^*)) \to (z_i, \pi_i)$. The randomized partial evaluation algorithm is executed by each party P_i on a given input $x \in \{0,1\}^*$. It uses the key-share $s_i = \mathsf{sk}_i$ and the public verification key $\mathsf{vk}_i = g_1^{s_i}$ and executes the following steps:
  - Compute $h := \mathsf{H}_1(x)$ and then verify the proof $\mathsf{KExpVer}((h, v), \pi^*)$ – if this fails then output $\perp$, otherwise go the the next step.
  - Compute $z_i := v^{s_i} \in \mathbb{G}_1$.
  - Compute the proof $\pi_i \leftarrow \mathsf{EqProve}(\mathsf{inst}, \mathsf{wit})$ where $\mathsf{inst} = (g_1, v, \mathsf{vk}_i, z_i)$ and $\mathsf{wit} = s_i$.
  - Output (z_i, π_i).
- $\mathsf{Aggregate}(\vec{\mathsf{vk}}, v, \{(z_i, \pi_i)\}_{i \in S}) =: z$. The aggregation algorithm gathers ℓ (z_i, π_i) pairs (say $|S| = \ell$) and if $\ell \leq t$, then output $\perp$. Otherwise, it executes the following steps for each $i \in [\ell]$:
  - Check whether for each $i \in S$ if $\mathsf{EqVer}((g_1, v, \mathsf{vk}_i, z_i), \pi_i)$ succeeds (that is, outputs 1). Record the successful (z_i, π_i) pair in a set T. If the number of successful partial verifications is *insufficient*, that is $|T| \leq t$, then output $\perp$. Otherwise go to the next step:
  - Compute the aggregated value as $z := \prod_{j \in T} z_j^{\lambda_{0,j,T}} \in \mathbb{G}_1$ using Lagrange Interpolation in the exponent. In other words, run the *linear* Recon algorithm in the exponent to reconstruct the secret as $z = \mathsf{H}_1(x)^s \in \mathbb{G}_1$ and output z.
- $\mathsf{PreVer}(\mathsf{pk}, (v, z)) =: 1/0$: The pre-verification algorithm just uses a pairing verification between blinded input and output.
  - Output result of the check $(e(v, \mathsf{pk}) = e(z, g_2))$.
- $\mathsf{Unblind}(z, \mathsf{st}) =: (y, \pi)$. The unblinding algorithm takes a blinded output z and a state st as inputs and then works as follows:
  - Parse $\rho := \mathsf{st}$.
  - Use ρ to unblind as $\pi := z^{\rho^{-1}}$.
  - Compute $y := \mathsf{H}_2(\pi)$.
  - Output (y, π).
- $\mathsf{Verify}(\mathsf{pk}, (x, y, \pi)) =: 1/0$: The verification algorithm takes the public key pk and a pair (y, π) as input is executed by any party as follows:
  - Output result of the check $(e(\mathsf{H}_1(x), \mathsf{pk}) = e(\pi, g_2)) \wedge (\mathsf{H}_2(x, \pi) = y))$.

Figure 2: Supra PVRF Construction

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

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

more complex assumption, called the Threshold Bilinear OMDH assumption, which is a variant of a similar assumption introduced in the paper [JKKX17]. This can be seen by observing the structural similarities in their threshold oblivious PRF and our PVRF constructions. Finally, output privacy follows partly from the structural similarities with BLS signatures, and relies on a variant of CDH assumption (namely co-CDH) over bilinear pairing. For full details we refer to our research paper [KMMM23].

Performance Analysis. We evaluate the performance of our PVRF construction and compare it with the GLOW [GLOW21]. We implement [pvr] our PVRF by extending the GLOW-DVRF framework [Gal, GLOW21] written in C++. The framework supports mcl [mcl] and RELIC [AGM+] cryptographic libraries. See Section 6 for details.

In our PVRF construction, for a given input, the requester generates a random blinding value and a NIZK proof of the correctness of the blinded input. The proof is a Schnorr signature-based proof of knowledge of the DLog exponent. The proof consists of two elements, one scalar and one group $\mathbb{G}_1$ element. After receiving the blinded input, each VRF node verifies the zero-knowledge proof before computing the partial evaluation of the VRF. The requester receives the aggregated evaluation and unblinds the output private VRF using the pre-computed blinding value to obtain the final VRF output.

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

The computation times for the steps of input-blinding at the requester and the partial evaluation at the VRF node are presented in Table 1; the table provides the timings for the operations using both the `mcl` and the RELIC libraries. The reported values are taken as a mean over 100 iterations over each operation. The VRF output would be verified by a smart contract; though we do not deploy the smart contract, our estimates indicate that the gas cost for the VRF verification on the BN256 curve would be $\sim 250k$ gwei. Roughly, each partial evaluation becomes about 2x more computationally expensive to accommodate for output-privacy.

6 Supra VRF Service Framework

Supra offers a VRF service, in that a randomness request from an user application is sent to VRF clan via an external smart contract (e.g. Ethereum). Then the VRF clan sends their response back, which is verified at the smart contract. We depict the message flow in the VRF service framework in Figure 3. To avail the service any user first forwards their own input to the smart contract along with the callback function to be called with the VRF output; this is indicated by step 1 in Figure 3. The smart contract may be running on any blockchain service like Ethereum. When an input from the user is sent to the smart contract, after verifying the input format and checking that the same value has not been requested previously, the smart contract combines that with additional information (detailed below) forming the VRF input INP. The system

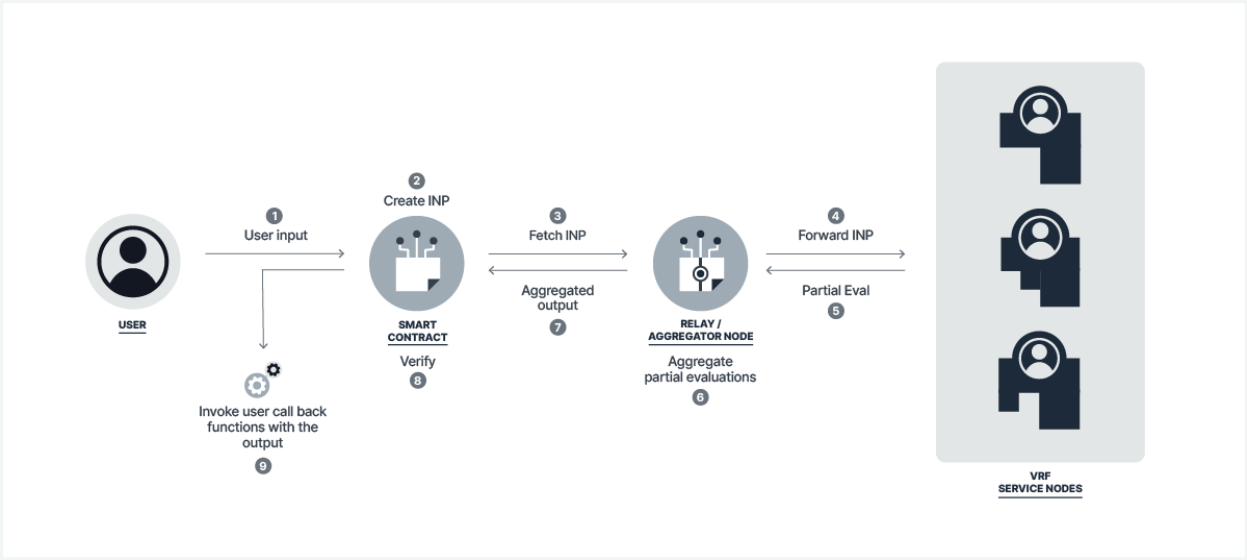

Figure 3: Message flow during the evaluation of the VRF value by the VRF service nodes upon request from a user. Steps 1-4 involve the request path where the user's input is provided to the smart contract, which then produces the VRF input (INP) that is sent to VRF clan via the relay family. Steps 5-9 involve the response path, in that the VRF output, partially produced by VRF clan, aggregated at the relay family and is then forwarded to the smart contract.

consists of relay nodes which monitor the smart contract; these relay nodes forward INP to the VRF service nodes after verifying the legitimacy of the request (possibly by verifying a signature). Each of these nodes generate a partial evaluation on the user input by running the $\mathsf{Part.Eval}_{\mathsf{sk}_i}(\mathsf{INP})$ and forwards the evaluation along with the (zero-knowledge) proof of correctness to the aggregator nodes. The relay nodes now acting as aggregators, collect each of the partial evaluations from the different VRF clan nodes for each user input forwarded. Whenever at least $t+1$ partial evaluations are obtained, they run the $\mathsf{Aggregate}(\cdot)$ algorithm after verifying the zero-knowledge proofs. The final VRF output and an accompanying proof are sent to the smart contract, which then verifies the correctness of the VRF output and if that succeeds, invokes the user specified callback function with the VRF value as the input. Below we summarize the steps of Figure 3:

1. The user forwards its own input to the smart contract.
2. The smart contract combines user input to other values and produces the VRF input INP.
3. The relay nodes fetch the input, verify legitimacy of INP (for example, by verifying the signature provided by the contract), whether it was previously used, and if all checks are satisfactory, forwards INP to all nodes within the VRF clan.
4. Each node in the VRF clan computes the partial evaluation on INP with the zero-knowledge proof of correct evaluation and sends them to the relay family nodes – now they act as aggregators.
5. When more than t partial evaluations are obtained at an aggregator node, they are aggregated to compute the VRF output and an accompanying proof of correctness. The pair is then sent to the smart contract as a response. The smart contract verifies the VRF output, and if that succeeds it invokes the user-specified callback function.

Optimizing the VRF verification using batching. We use a slightly different architecture using the Supra SMR service for optimization. In this architecture the smart contract processes a number of user requests together in batches, producing individual INP for each of them. The relay nodes also process them in batches, fetching a number of VRF inputs (say, $\mathsf{INP}_1, \mathsf{INP}_2, \ldots$) and forwards them to the VRF clan.

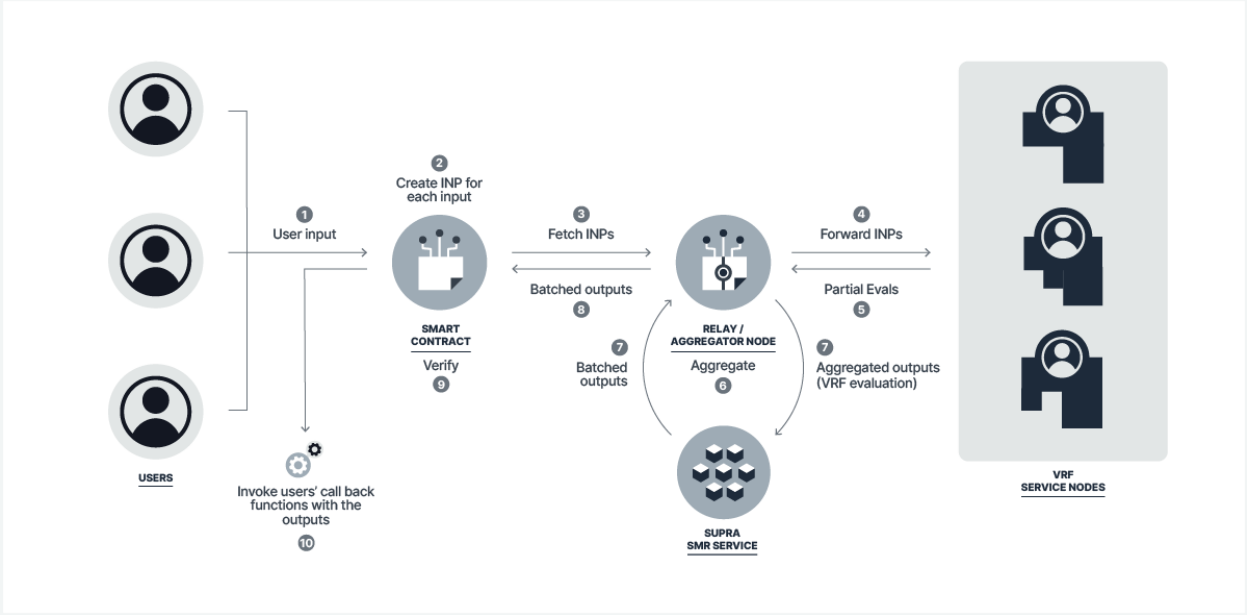

Figure 4: Message flow for optimized VRF service framework. The difference is that now many users are making VRF requests. Each of them is processed within the smart contract individually, and then they are forwarded to VRF clan in a batch – this is depicted in steps 1-4. Batched message flow is marked with a thicker arrow. The main difference comes in step 7, when multiple aggregated values are forwarded to the Supra SMR service, which verifies them individually, and for a batch of successfully verified VRF outputs produces a single signature. The signature and the batch VRF outputs are sent to the smart contract, which simply verifies the signature. The rest of the process is the same as for the unoptimized case.

A node (which holds the secret sk_i, say) in the VRF clan, verifies the legitimacy of each of the INP_j and responds with $\mathsf{Part.Eval}_{\mathsf{sk}_i}(\mathsf{INP}_j)$ plus an accompanying zero knowledge proof of correctness. The relay nodes then aggregate them to compute a batch of VRF outputs and accompanying proofs. The batch of VRF outputs and proofs are then sent to the Supra SMR service as a transaction, which verifies each output and then produces one single signature for the entire batch, if all of them are legitimate. The relay nodes fetch the batch of VRF outputs (excludes individual proofs of correctness) and the signature from the Supra SMR service and forwards them to the contract. The smart contract only verifies the signature and if that succeeds, invokes each of the users' call back functions with the respective VRF output values. Below we summarize the steps of Figure 4, where main differences are highlighted in blue:

1. Multiple users forward their inputs to the smart contract.
2. The smart contract combines each user's input with other values and produces multiple VRF inputs $\mathsf{INP}_1, \mathsf{INP}_2, \ldots$.
3. The relay nodes fetch the inputs, verify the legitimacy of each INP_j (for example, by verifying the signature provided by the contract), whether it was previously used, and if all checks are satisfactory forwards $\mathsf{INP}_1, \mathsf{INP}_2 \ldots$ to all nodes within the VRF clan.
4. Each node in the VRF clan computes the partial evaluation on each INP_j with the zero-knowledge proof of correct evaluation and sends them to the relay nodes – now they act as aggregators.
5. The relay nodes aggregate the evaluations for each INP_j once $t+1$ partial evaluations are available, and then send a batch of VRF outputs along with the accompanying proof to the Supra SMR service as a transaction.
6. The SMR service verifies all output, proof pairs and then if all of them satisfy, generates a signature for a batch of VRF outputs.

7. The relay node fetches a batch of outputs and the signature from the Supra SMR service transaction and forwards it to the contract.

8. The contract simply verifies the signature, and if that succeeds it invokes the user-specified callback function the respective VRF output values.

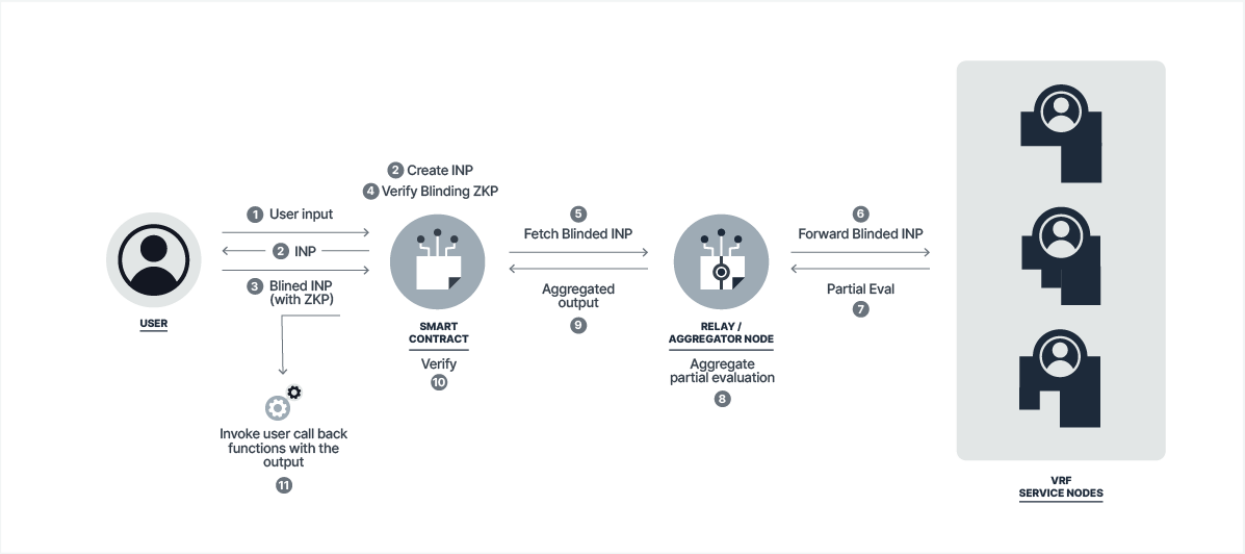

Figure 5: Message flow during the evaluation of the private VRF. In this, after forming the input INP at the smart contract, the user obtains it and blinds it. The blinded INP along with zero knowledge proof of the blinding factor and the correctness of blinding is forwarded to the smart contract. The smart contract verifies the zero knowledge proof, after which the relay nodes fetch the blinded INP.

6.1 Private VRF (without batching)

For the PVRF computation, the above framework stays the same in each case. However, the work-flow changes slightly. In particular, initially when the user forwards its input to the smart contract, the smart contract creates the VRF input INP and sends it back to the user, who then blinds INP and sends a pair consisting of blinded INP and an accompanying zero-knowledge proof of correct blinding. The contract then checks the proof, and if that succeeds, marks it "ok" – the relay nodes fetch the blinded input only if it is marked "ok." The rest of the work flow is the same as before (the smart contract will run PreVer instead of Verify now). This is depicted in Figure 5. The callback function should run the Unblind algorithm on the blinded output inside it to obtain the VRF output.

6.2 Constructing the VRF input, INP.

The VRF input is produced by the smart contract. Each input INP is a concatenation of the following values:

- User input – this is user's chosen input and maybe empty.
- Block-hash – this is included to ensure that no one can request the input before the block-hash is computed. This prevents one from pre-computing a VRF output to be used at a later time.
- Unique nonce –a unique nonce generated at the specific smart contract each time a VRF is called. This ensures that each VRF input is different. For this the smart contract must keep a state (for example, a counter).
- Chain id – this distinguishes inputs generated at two different blockchains (for example, Ethereum and Solana).

- User address – this is user-specific information to distinguish between requests from different users.
- Callback function name – this is included to distinguish between two different functions coming from the same user at about the same time.
- DVRF or PVRF – this is a flag distinguishing between a PVRF and DVRF. This is a crucial input, because without this a PVRF request maybe maliciously processed as a DVRF, leading to exposure of the output.

7 Conclusion

In this paper we describe the Supra VRF service in detail. Specifically, we provide the definitions of VRF, distributed VRF (DVRF) and output-private DVRF (PVRF) – among which PVRF is a novel concept introduced in this paper. Then we provide Supra's core-DVRF construction (GLOW [GLOW21]) and also based on that a PVRF construction; we provide security intuitions for the constructions. Finally we elaborate on the VRF service frameworks for DVRF, and an optimized batched VRF. We also provide the additional changes one needs to make for the output-private versions. All our VRF protocols ensure decentralization of the secrets at all times and achieve stronger security properties than the existing deployments. In our research paper [KMMM23] we provide rigorous formalization and security proofs for our PVRF constructions.

References

[AGM+] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic. 13

[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: 25th Conference on Computer and Communications Security*, pages 1993–2010, Toronto, ON, Canada, October 15–19, 2018. ACM Press. 4

[BDN18] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018, Part II*, volume 11273 of *Lecture Notes in Computer Science*, pages 435–464, Brisbane, Queensland, Australia, December 2–6, 2018. Springer, Heidelberg, Germany. 9

[Bit20] Nir Bitansky. Verifiable random functions from non-interactive witness-indistinguishable proofs. *Journal of Cryptology*, 33(2):459–493, April 2020. 3

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532, Gold Coast, Australia, December 9–13, 2001. Springer, Heidelberg, Germany. 4

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press. 11

[Car] Cardano. Ouroboros Protocol. https://cardano-foundation.gitbook.io/stake-pool-course/lessons/introduction/ouroboros. 4

[Cha] Chainlink. Chainlink VRF: On-Chain Verifiable Randomness. https://blog.chain.link/chainlink-vrf-on-chain-verifiable-randomness/. 4

[CL14] Ran Cohen and Yehuda Lindell. Fairness versus guaranteed output delivery in secure multiparty computation. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 466–485, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Heidelberg, Germany. 8

[Clo] Cloudflare. Decentralized Verifiable Randomness Beacon. https://developers.cloudflare.com/randomness-beacon/. 4

[Cor] Corestar. Corestar Arcade: Tendermint-based Byzantine Fault Tolerant (BFT) middleware with an embedded BLS-based random beacon. https://github.com/corestario/tendermint. 4

[CP93] David Chaum and Torben P. Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *Advances in Cryptology – CRYPTO'92*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105, Santa Barbara, CA, USA, August 16–20, 1993. Springer, Heidelberg, Germany. 6

[DAO] DAOBet (ex — DAO.Casino). To Deliver On-Chain Ran- dom Beacon Based on BLS Cryptography. https://daobet.org/blog/on-chain-random-generator/. 4

[Dod03] Yevgeniy Dodis. Efficient construction of (distributed) verifiable random functions. In Yvo Desmedt, editor, *PKC 2003: 6th International Workshop on Theory and Practice in Public Key Cryptography*, volume 2567 of *Lecture Notes in Computer Science*, pages 1–17, Miami, FL, USA, January 6–8, 2003. Springer, Heidelberg, Germany. 4

[DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *PKC 2005: 8th International Workshop on Theory and Practice in Public Key Cryptography*, volume 3386 of *Lecture Notes in Computer Science*, pages 416–431, Les Diablerets, Switzerland, January 23–26, 2005. Springer, Heidelberg, Germany. 3

[EKS+21] Muhammed F. Esgin, Veronika Kuchta, Amin Sakzad, Ron Steinfeld, Zhenfei Zhang, Shifeng Sun, and Shumo Chu. Practical post-quantum few-time verifiable random function with applications to algorand. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part II*, volume 12675 of *Lecture Notes in Computer Science*, pages 560–578. Springer, 2021. 3

[Gal] David Galindo. Distributed Verifiable Random Functions: an Enabler of Decentralized Random Beacons. https://github.com/fetchai/research-dvrf. 13

[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. 4

[GLOW21] David Galindo, Jia Liu, Mihai Ordean, and Jin-Mann Wong. Fully distributed verifiable random functions and their application to decentralised random beacons. In *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*, pages 88–102. IEEE, 2021. 2, 3, 4, 8, 9, 11, 13, 17

[GVPR18] Sharon Goldberg, Jan Vcelak, Dimitrios Papadopoulos, and Leonid Reyzin. Verifiable random functions (vrfs). 2018. 3

[HJ16] Dennis Hofheinz and Tibor Jager. Verifiable random functions from standard assumptions. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A: 13th Theory of Cryptography Conference, Part I*, volume 9562 of *Lecture Notes in Computer Science*, pages 336–362, Tel Aviv, Israel, January 10–13, 2016. Springer, Heidelberg, Germany. 3

[HMW18] Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018. 4

[JKKX17] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17: 15th International Conference on Applied Cryptography and Network Security*, volume 10355 of *Lecture Notes in Computer Science*, pages 39–58, Kanazawa, Japan, July 10–12, 2017. Springer, Heidelberg, Germany. 3, 11, 13

[KM13] Veronika Kuchta and Mark Manulis. Unique aggregate signatures with applications to distributed verifiable random functions. In Michel Abdalla, Cristina Nita-Rotaru, and Ricardo Dahab, editors, *CANS 13: 12th International Conference on Cryptology and Network Security*, volume 8257 of *Lecture Notes in Computer Science*, pages 251–270, Paraty, Brazil, November 20–22, 2013. Springer, Heidelberg, Germany. 4

[KMMM23] Aniket Kate, Siva Maradana, Easwar Mangipudi, and Pratyay Mukherjee. Output Private VRFs. Personal Communication, 2023. 3, 11, 13, 17

[Koh19] Lisa Kohl. Hunting and gathering - verifiable random functions from standard assumptions with short proofs. In Dongdai Lin and Kazue Sako, editors, *PKC 2019: 22nd International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 11443 of *Lecture Notes in Computer Science*, pages 408–437, Beijing, China, April 14–17, 2019. Springer, Heidelberg, Germany. 3

[mcl] mcl - A portable and fast pairing-based cryptography library. . https://github.com/herumi/mcl. 13

[MRV99] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science*, pages 120–130, New York, NY, USA, October 17–19, 1999. IEEE Computer Society Press. 3

[Nie21] David Niehues. Verifiable random functions with optimal tightness. In Juan A. Garay, editor, *Public-Key Cryptography - PKC 2021 - 24th IACR International Conference on Practice and Theory of Public Key Cryptography, Virtual Event, May 10-13, 2021, Proceedings, Part II*, volume 12711 of *Lecture Notes in Computer Science*, pages 61–91. Springer, 2021. 3

[NPR99] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 327–346, Prague, Czech Republic, May 2–6, 1999. Springer, Heidelberg, Germany. 4

[Pol] Polkadot. Polkadot Wiki – Randomness. https://wiki.polkadot.network/docs/learn-randomness. 4

[pvr] Pri-DVRF – Anonymous link. https://anonymous.4open.science/r/PVRF_IMPL-1F04/README.md. 13

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252, Santa Barbara, CA, USA, August 20–24, 1990. Springer, Heidelberg, Germany. 4, 6

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979. 2, 6

[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. 4