IOHK Technical Report:

On UC-Secure Range Extension and Batch Verification for ECVRF

Christian Badertscher[1], Peter Gaži[1], Iñigo Querejeta-Azurmendi[1], and Alexander Russell[1]

[1] *IOHK,* firstname.lastname@iohk.io

December 10, 2021

Abstract

This technical report contains three important results. First, it describes a simple construction in the random-oracle model (ROM) that generically extends the range of a verifiable random function (VRF) specified as a UC functionality. We prove our construction UC secure and show that it can be used in Ouroboros to reduce the number of VRF evaluations (per slot) and VRF verifications (per block) from two to one at the price of additional hash-function evaluations.

As a second result, we show that the *Elliptic Curve VRF (ECVRF)* construction, whose standardization by the IETF is progressing, achieves the strong notion of UC security in the ROM.

Finally, we show how ECVRF can be tweaked and equipped with a batch-verification capability for increased efficiency. We formalize the security goal of batch verification in UC and formally prove the security of this construction in the ROM.

1 Verifiable Random Functions

We first define the syntax of a verifiable random function (VRF). We denote by κ the security parameter. The domain $\mathcal{X}$ and the range $\mathcal{Y}$ of the VRF are finite sets represented by $\mathcal{X} = \{0,1\}^{\ell(k)}$ and $Y = \{0,1\}^{\ell_{\mathsf{VRF}}(k)}$ respectively, where $\ell(.)$ and $\ell_{\mathsf{VRF}}(.)$ are functions of the security parameter. For notational simplicity we often drop the explicit dependence on κ.

Definition 1.1 (VRF Syntax)**.** A verifiable random function (VRF) consists of a triple of PPT algorithms $\mathsf{VRF} := (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Vfy})$:

- The probabilistic algorithm $(sk, vk) \leftarrow \mathsf{Gen}(1^\kappa)$ takes as input the security parameter κ in unary encoding and outputs a key pair, where sk is the secret key and vk is the (public) verification key.

- The probabilistic algorithm $(Y, \pi) \leftarrow \mathsf{Eval}(sk, X)$ takes as input a secret key sk and $X \in \mathcal{X}$ and outputs a function value $Y \in \mathcal{Y}$ and a proof π.

- The (possibly probabilistic but usually deterministic) algorithm $b \leftarrow \mathsf{Vfy}(vk, X, Y, \pi)$ takes as input a verification key vk, input value $X \in \mathcal{X}$, output value $Y \in \mathcal{Y}$, as well as a proof π, and returns a bit b.

In the context of Ouroboros [DGKR18, BGK$^+$18]], we need that the VRF algorithms implement an ideal object that we call the VRF functionality. For security this means intuitively that all outputs generated by the VRF algorithms are indistinguishable from outputs of a truly random function—even to an attacker who could potentially craft its own private VRF key. We assume in the following some familiarity with the UC framework [Can20].

Ideal Functionality $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$

$\mathcal{F}_{\mathsf{VRF}}$ interacts with its set of registered parties $\mathcal{P}$ denoted by $U_1, \ldots, U_{|\mathcal{P}|}$ and the adversary/simulator $\mathcal{S}$. It maintains tables $T[\cdot,\cdot]$ that are initially empty (denoted by symbol $\perp$). The tables are initialized on-the-fly. The functionality maintains a set S_{pk} to keep track of registered keys, and S_{eval} to keep track of all known VRF evaluations.

- **Key Generation.** Upon receiving a message (KeyGen, sid) from U_i s.t. $(U_i, \cdot) \notin S_{pk}$, hand $(\mathsf{KeyGen}, sid, U_i)$ to $\mathcal{S}$ (ignore the request if $(U_i, \cdot) \in S_{pk}$). Upon receiving $(\mathsf{VerificationKey}, sid, U_i, v)$ from $\mathcal{S}$:
  1. If U_i is corrupted, ignore the request.
  2. If $(U_i, \cdot) \notin S_{pk}$ and $\forall(\cdot, v') \in S_{pk} : v \neq v'$, set $S_{pk} \leftarrow S_{pk} \cup \{(U_i, v)\}$ and return $(\mathsf{VerificationKey}, sid, v)$ to U_i.
  3. Else, ignore the request.
- **Malicious Key Generation.** Upon receiving a message $(\mathsf{KeyGen}, sid, v)$ from $\mathcal{S}$, do the following: if $\forall(\cdot, v') \in S_{pk} : v \neq v'$, set $S_{pk} \leftarrow S_{pk} \cup \{(\mathcal{S}, v)\}$. Return the activation to $\mathcal{S}$.
- **VRF Evaluation and Proof.** Upon receiving a message $(\mathsf{EvalProve}, sid, m)$ from U_i, verify that some $(U_i, v) \in S_{pk}$ is recorded. If not, then ignore the request. Else, send $(\mathsf{EvalProve}, sid, U_i, m)$ to $\mathcal{S}$ and upon receiving $(\mathsf{EvalProve}, sid, U_i, m, \pi)$ from $\mathcal{S}$, do the following:
  1. Ignore the request if the proof is not unique, i.e., if $\exists T[v', m'] = (y', S')$ such that $\pi \in S' \wedge ((v' \neq v) \vee (m' \neq m))$.
  2. If $T[v, m] = \perp$, assign $y \leftarrow_{\$} \{0,1\}^{\ell_{\mathsf{VRF}}}$ and set $T[v, m] \leftarrow \{y, \{\pi\}\}$.
  3. If $T[v, m] = (y, S) \neq \perp$, set $T[v, m] \leftarrow \{y, S \cup \{\pi\}\}$.
  4. Set $S_{\text{eval}} \leftarrow S_{\text{eval}} \cup \{(v, m, y)\}$ and output $(\mathsf{Evaluated}, sid, m, y, \pi)$ to U_i.
- **Malicious VRF Evaluation.** Upon receiving a message $(\mathsf{Eval}, sid, v, m)$ from $\mathcal{S}$, do the following:

  Case 1: $\exists(U_i, v) \in S_{pk}$ where U_i is not corrupted: let $T[v, m] = (y, S)$. If $S \neq \emptyset$, return $(\mathsf{Evaluated}, sid, y)$ to $\mathcal{S}$. Otherwise, ignore the request.

  Case 2: $(\mathcal{S}, v) \in S_{pk}$ or $\exists(U_i, v) \in S_{pk}$, U_i corrupted: if $T[v, m] = \perp$, first choose $y \leftarrow_{\$} \{0,1\}^{\ell_{\mathsf{VRF}}}$ and set $T[v, m] \leftarrow (y, \emptyset)$. Return $(\mathsf{Evaluated}, sid, y)$ to $\mathcal{S}$

  Else: Ignore the request.
- **Verification.** Upon receiving a message $(\mathsf{Verify}, sid, m, y, \pi, v')$ from any ITI M, send $(\mathsf{Verify}, sid, m, y, \pi, v', S_{\text{eval}})$ to $\mathcal{S}$. Upon receiving $(\mathsf{Verified}, sid, m, y, \pi, v', \phi)$ from $\mathcal{S}$ do:

  Case 1: $v' = v$ for some $(\cdot, v) \in S_{pk}$ s.t. $T(v, m) = (y, S)$ for some set S.
  1. If $\pi \in S$, then set $f \leftarrow 1$.
  2. Else, if $\phi = 1$ and $\forall T[\tilde{v}, \tilde{m}] = (y', S') : \pi \notin S'$, then set $T[v, m] = (y, S \cup \{\pi\})$ and $f \leftarrow 1$.
  3. Else, set $f \leftarrow 0$.

  Else: Set $f \leftarrow 0$.

  Provide the output $(\mathsf{Verified}, sid, v', m, y, \pi, f)$ to the caller M.
- **Adversarial Leakage [New compared to [DGKR18, BGK+18]].** On input $(\mathsf{PastEvaluations}, sid)$ from $\mathcal{S}$, return S_{eval} to $\mathcal{S}$.

Figure 1: The VRF functionality.

VRF as a UC protocol. Any verifiable random function VRF can be cast as a simple protocol π_{VRF} in the UC framework [Can20] as follows: Each party U_i in session sid acts as follows: on its first input of the form (KeyGen, sid), run $(sk, vk) \leftarrow \mathsf{VRF.Gen}(1^\kappa)$ and output $(\mathsf{VerificationKey}, sid, vk)$ and internally store sk (and ignore key generation requests from now on). On input $(\mathsf{EvalProve}, sid, m)$ (and if a key has been generated before) evaluate $(Y, \pi) \leftarrow \mathsf{VRF.Eval}(sk, m)$ and output $(\mathsf{Evaluated}, sid, Y, \pi)$. (If no key has been generated yet, evaluation queries are ignored.) On input $(\mathsf{Verify}, sid, m, y, \pi, v')$, the party evaluates $b \leftarrow \mathsf{VRF.Vfy}(v', m, y, \pi)$ and finally returns $(\mathsf{Verified}, sid, v', m, y, \pi, b)$.

Definition 1.2 (UC security of a VRF)**.** A verifiable random function VRF (with input domain $\mathcal{X} = \{0,1\}^\ell$ and range $\mathcal{Y} = \{0,1\}^{\ell_{\mathsf{VRF}}}$) is called UC-secure, if π_{VRF} UC-realizes $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$ specified in Figure 1.

The functionality captures all we want from a VRF, from correctness to security: the function table corresponding to each public key is a truly random function (and thus also guarantees a unique association of the key-value pair to output Y), even if the adversary can pick its own crafted public key. Furthermore, no wrong association can be ever verified and every completed honest evaluation can be later verified correctly. Note that the functionality is based on [DGKR18, BGK+18], but contains some modifications: first, verification is now more in-line with typical UC formulations for (signature) verification, where the adversary is given some limited influence (in prior versions, the adversary had to inject proofs in between verification request and response to accomplish the same thing). Furthermore, the uniqueness notion for proofs has been correctly adjusted to catch the corner case that schemes might choose to de-randomize the prover (akin signatures) which is a crucial point later when we look at ECVRF. The remaining changes are merely syntactical compared to [BGK+18]. If π_{VRF} UC realizes this functionality, then this means that the triple of algorithms VRF is essentially computationally indistinguishable from this functionality and therefore can be considered correct and secure.

Random oracles in UC. When working in the random-oracle model, the UC protocol above is changed as follows: whenever VRF prescribes a call to a particular hash function to hash some value x, this is replaced by a call of the form $(\textsc{eval}, sid, x)$ to an instance of a so-called random oracle functionality, which internally implements an ideal random function $\{0,1\}^* \to \mathcal{Y}'$ and returns the corresponding function value back to the caller. This functionality is specified in Figure 2. We will often use the notation $\mathsf{H}(x)$ in the specifications to refer to a general hash function with the understanding that this call will be treated as a random oracle call in the security proof.

2 Generic VRF Range Extension in the ROM

2.1 Specification

Let $\mathsf{H} : \{0,1\}^* \to \mathcal{Y}$ denote a general hash function. Let VRF be a verifiable random function with input-value domain $\mathcal{X}$ and output domain $\mathcal{Y}$.

We construct a VRF $\widetilde{\mathsf{VRF}}$ with input-value domain $\mathcal{X}$ and output domain $\mathcal{Y}^c$ for a fixed constant $c > 0$. In the following, we let $\mathtt{CONST}_i, i = 1, \ldots, c$ be c fixed and pairwise different constants (of fixed length) and $||$ denotes concatenation of bitstrings. The algorithms are defined as follows:

Key Generation: Key generation remains unchanged: $\widetilde{\mathsf{VRF}}.\mathsf{Gen}(1^\kappa) := \mathsf{VRF.Gen}(1^\kappa)$.

$\mathcal{F}_{\mathsf{RO}}^{\mathcal{Y}}$

The functionality is parameterized by the output domain $\mathcal{Y}$. It maintains a (dynamically updatable) function table $\mathcal{T}$ (initially $\mathcal{T} = \emptyset$). For simplicity we write $T[x]$ to denote the function value assigned to x in the table $\mathcal{T}$ (if defined) and use the expression $T[x] = \perp$ to denote that no pair of the form $(x, \cdot)$ is in $\mathcal{T}$.

- Upon receiving $(\textsc{eval}, sid, x)$ from some party U_p (or from the adversary), do the following:
  1. If $T[x] = \perp$ sample a value y uniformly at random from $\mathcal{Y}$, set $T[x] \leftarrow y$ and add $(x, T[x])$ to $\mathcal{T}$.
  2. Return $(\textsc{eval}, sid, x, T[x])$ to the caller.

Figure 2: The random-oracle functionality idealizing a hash function $\{0,1\}^* \rightarrow \mathcal{Y}$.

Evaluation: The algorithm $\widetilde{\mathsf{VRF}}.\mathsf{Eval}(sk, X)$ for $X \in \mathcal{X}$ is defined as follows:

1. Run $(Y, \pi) \leftarrow \mathsf{VRF}.\mathsf{Eval}(sk, X)$.
2. Compute $Y_i \leftarrow \mathsf{H}(\mathtt{CONST}_i \,||\, Y)$.
3. Return the pair $((Y_1, \ldots, Y_c), (\pi, Y))$.

Verification: The algorithm $\widetilde{\mathsf{VRF}}.\mathsf{Vfy}(vk, X, Y, \pi)$ is defined as follows, where $X \in \mathcal{X}$ and $Y \in \mathcal{Y}^c$:

1. Parse $\pi = (\pi', Y')$ where $Y' \in \mathcal{Y}$ (return 0 in case of parsing error).
2. Return $b := \mathsf{VRF}.\mathsf{Vfy}(vk, X, Y', \pi') \wedge \left(\bigwedge_{i=1}^{c} Y_i = \mathsf{H}(\mathtt{CONST}_i \,||\, Y') \right)$.

Rationale of the construction. Before we cast the above construction in the provable security parlance of Ouroboros [DGKR18, BGK+18], we provide here a non-technical justification of the above construction. Assume that the underlying VRF provides all guarantees we informally demanded above, then our construction enjoys basically the same properties: the correctness properties follows from the correctness properties of the underlying VRF and the fact that H is a public function.

For security, we observe three properties for Y_i: (1) it is unpredictable to anyone not knowing the secret key, (2) it cannot be manipulated even by the owner of the secret key, and (3) it is unpredictable to the owner of the secret key without evaluating the VRF. In particular note that Y_i can only be determined by someone who knows the value Y' (since in the ROM, H is a random function), and Y' can only be computed by someone having the secret key and otherwise is unpredictable thanks to the security of the underlying VRF. Furthermore, since H is a public function, Y_i is determined fully by Y' (and the constant $\mathtt{CONST}_i$).

3 Security Analysis of the Range-Extension Construction

The required level of security of a VRF in the setting of Ouroboros is UC security. UC security is a strong notion and this strength is the main reason why the above construction needs a more formal security argument. In the following, we assume some familiarity with the security arguments in [DGKR18, BGK+18].

3.1 Range Extension as a Modular UC Protocol

The construction $\widetilde{\mathsf{VRF}}$ can be cast as a modular UC protocol $\pi_{\widetilde{\mathsf{VRF}}}$, where we assume that the protocol has access to the hybrid functionality $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$ idealizing the underlying scheme VRF with range $\{0,1\}^{\ell_{\mathsf{VRF}}}$) (and also access to the random oracle $\mathcal{F}_{\mathrm{RO}}^{\mathcal{Y}}$ to idealize H):

Each party U_i in session sid acts as follows: on input (KeyGen, sid), relay this input to $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$ and when receiving the answer $(\mathsf{VerificationKey}, sid, vk)$ return this answer as output. On input $(\mathsf{EvalProve}, sid, m)$ relay this input to $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$ and when receiving the answer $(\mathsf{Evaluated}, sid, Y, \pi)$, query, for $i/in[1,c]$, the random oracle $\mathcal{F}_{\mathrm{RO}}^{\mathcal{Y}}$ with input $(\textsc{eval}, sid, (\mathtt{CONST}_i \,||\, Y))$. Let Y_i be the obtained answers. Then output the return value $(\mathsf{Evaluated}, sid, (Y_1, \ldots, Y_c), (\pi, Y))$. Finally, on input $(\mathsf{Verify}, sid, m, y, \pi, v')$, parse $\pi = (\pi', Y')$ and $y = (Y_1, \ldots, Y_c) \in \{0,1\}^{c \cdot \ell_{\mathsf{VRF}}}$. If the format is wrong, return $(\mathsf{Verified}, sid, v', m, y, \pi, 0)$. Otherwise, query $(\mathsf{Verify}, sid, m, Y', \pi', v')$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$ and let the returned decision bit be b. Then query the $\mathcal{F}_{\mathrm{RO}}^{\mathcal{Y}}$, for $i/in[1,c]$, via $(\textsc{eval}, sid, (\mathtt{CONST}_i \,||\, Y'))$ and denote the RO outputs by y_i. Then compute $b' \leftarrow b \wedge \left(\bigwedge_{i=1}^{c} Y_i = y_i \right)$ and return $(\mathsf{Verified}, sid, v', m, y, \pi, b')$.

3.2 The UC Realization Statement

The formal theorem of our range extension can be stated in very simple terms:

Theorem 3.1. *Protocol $\pi_{\widetilde{\mathsf{VRF}}}$ UC-realizes $\mathcal{F}_{\mathsf{VRF}}^{\ell,c\cdot\ell_{\mathsf{VRF}}}$.*

Proof. We first describe the simulator $\mathcal{S}$ for the so-called dummy real-world adversary that is under the control of the environment $\mathcal{Z}$.[1] The simulator interacts with functionality $\mathcal{F}_{\mathsf{VRF}}^{\ell,c\cdot\ell_{\mathsf{VRF}}}$ and simulates towards the environment a transcript that is indistinguishable from a protocol run of $\pi_{\widetilde{\mathsf{VRF}}}$, where the environment interacts with parties running algorithms as specified in $\pi_{\widetilde{\mathsf{VRF}}}$ and additionally has access to the adversarial interface of the assumed (hybrid) functionality $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$ and the random-oracle functionality $\mathcal{F}_{\mathrm{RO}}^{\mathcal{Y}}$. The simulator internally emulates an execution of $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$ and emulates the random oracle by maintaining a function table $H[x]$ (initially empty).

Reaction on requests from $\mathcal{F}_{\mathsf{VRF}}^{\ell,c\cdot\ell_{\mathsf{VRF}}}$. We first define the simulation upon the different outputs of $\mathcal{F}_{\mathsf{VRF}}^{\ell,c\cdot\ell_{\mathsf{VRF}}}$ (provoked as reactions of inputs by honest parties).

On $(\mathsf{KeyGen}, sid, U_i)$**:** Then obtain a new verification key from the emulated instance $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$; that is, ask the environment to provide a new key vk_i and return $(\mathsf{VerificationKey}, sid, U_i, vk_i)$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell,c\cdot\ell_{\mathsf{VRF}}}$.

On $(\mathsf{EvalProve}, sid, U_i, m)$**:** The simulator obtains the output (y, π) on input m from its simulated instance $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$; this means it first obtains a proof π from the environment and then sampling a new value $y \in \{0,1\}^{\ell}_{\mathsf{VRF}}$ at random provided m has not been asked before. Then, the simulator defines $\pi' := (\pi, y)$ and returns $(\mathsf{EvalProve}, sid, m, \pi')$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell,c\cdot\ell_{\mathsf{VRF}}}$. The simulator stores internally $(\mathtt{PROG}, i, m, y)$ to prepare for programming the RO.

On $(\mathsf{Verify}, sid, m, y, \pi, v', S_{\text{eval}})$**:** The simulator first checks for new entries $(\mathtt{PROG}, i, m, y)$ added in previous activations. For each of these entries, it parses the set S_{eval} of all previously evaluated VRF values to obtain $(v_i, m, (y_1, \ldots, y_c))$ where $(y_1, \ldots, y_c) \in (\{0,1\}^{\ell_{\mathsf{VRF}}})^c$ and assigns for each of these new entries the random-oracle value $H[(\mathtt{CONST}_j \,||\, y)] \leftarrow y_j$, $j = 1, \ldots, n$ if the locations $x_j = (\mathtt{CONST}_j \,||\, y)$ have not been programmed already. If such an assignment is not

[1] We point out that a UC proof w.r.t. this adversary implies security against any adversary.

possible because the location ($\texttt{CONST}_i \,||\, y$) have already been programmed with different values y_i respectively, then abort the simulation. We call this event $\mathsf{SIMFAIL}$.

Next, the simulator parses y as $(y_1, \ldots, y_c)$ and π as pair (π', y') and verifies the combination (m, y', π', v') using the internally emulated functionality $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$. Part of this is sending $(\mathsf{Verify}, sid, m, y', \pi, v', S'_{\text{eval}})$ to the environment (for the set S'_{eval} maintained by the internally emulated functionality), and when the environment returns the verification result $(\mathsf{Verified}, sid, m, y', \pi', v', b')$ to this query, $\mathcal{S}$ provides this input to its internally emulated instance. It then checks that $y_i = H[(\texttt{CONST}_j \,||\, y')]$ for all $i = 1 \ldots c$. If all checks are fulfilled $\mathcal{S}$ sends the reply $(\mathsf{Verified}, sid, m, y, \pi, v', b)$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell,c\cdot\ell_{\mathsf{VRF}}}$. If any check fails, it sends the reply $(\mathsf{Verified}, sid, m, y, \pi, v', 0)$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell,c\cdot\ell_{\mathsf{VRF}}}$.

Interaction with environment (adversarial interface). Whenever invoked with an input from the environment, the simulator first checks for new entries $(\texttt{PROG}, i, m, y)$ added in previous activations. It thus first obtains the set via query S_{eval} $(\mathsf{PastEvaluations}, sid)$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell,c\cdot\ell_{\mathsf{VRF}}}$. For each of the entries $(\texttt{PROG}, i, m, y)$, it parses the set S_{eval} of all previously evaluated VRF values to obtain $(v, m, (y_1, \ldots, y_c))$ for $(y_1, \ldots, y_c) \in (\{0,1\}^{\ell_{\mathsf{VRF}}})^c$ and assigns $H[(\texttt{CONST}_j \,||\, y)] \leftarrow y_j$, $j = 1, \ldots, n$, if the locations $x_j = (\texttt{CONST}_j \,||\, y)$ have not been programmed already. If such an assignment is not possible because the location ($\texttt{CONST}_i \,||\, y$) have already been programmed with different values y_i respectively, then abort the simulation. We call this event $\mathsf{SIMFAIL}$.

Whenever the environment asks for an RO-evaluation for a new value x, then $\mathcal{S}$ samples a value $y \in \{0,1\}^{\ell}_{\mathsf{VRF}}$ at random and assigns $H[x] \leftarrow y$. If a function value for x is already defined, then return $H[x]$.

Whenever activated by $(\mathsf{KeyGen}, sid, v)$ from the environment, $\mathcal{S}$ provides this as input to the internally emulated instance and invokes $\mathcal{F}_{\mathsf{VRF}}^{\ell,c\cdot\ell_{\mathsf{VRF}}}$ on input $(\mathsf{KeyGen}, sid, v)$ and returns whatever is returned by the functionality.

Whenever activated with $(\mathsf{Eval}, sid, v, m)$ from $\mathcal{Z}$ (malicious evaluation of the underlying VRF functionality), $\mathcal{S}$ emulates this input on the internally emulated functionality $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$. When a simulated value y is obtained, then $\mathcal{S}$ invokes $\mathcal{F}_{\mathsf{VRF}}^{\ell,c\cdot\ell_{\mathsf{VRF}}}$ with $(\mathsf{Eval}, sid, v, m)$ to receive the function values $(y_1, \ldots, y_c)$ it sampled for m (and w.r.t. v) and $\mathcal{S}$ programs the RO by setting $H[\texttt{CONST}_i \,||\, y] \leftarrow y_i$ for $i = 1, \ldots, c$ unless the locations have already been written to with different values. As above, if such an assignment cannot be made because the location ($\texttt{CONST}_i \,||\, y$) have already been programmed with different values y_i respectively, then abort the simulation (even $\mathsf{SIMFAIL}$). Finally, return to $\mathcal{Z}$ with output $(\mathsf{Evaluated}, sid, y)$.

When activated with input $(\mathsf{PastEvaluations}, sid)$ or with verification requests or verification results towards the internally emulated functionality $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$, then provide the received input to the emulated instance of $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$ and return to the environment whatever the emulated instance outputs.

Finally, whenever a party is corrupted, $\mathcal{S}$ corrupts the corresponding party in $\mathcal{F}_{\mathsf{VRF}}^{\ell,c\cdot\ell_{\mathsf{VRF}}}$ and marks it as corrupted in its internally emulated instance of $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$.

Analysis of the simulation. We observe that the simulation only fails in case it has to abort. The probability of event $\mathsf{SIMFAIL}$ corresponds to the probability that a location $x = (\texttt{CONST}_i \,||\, y)$ of the random oracle has been evaluated before the simulator could program it correctly with the value y_i chosen by the ideal functionality. This probability is, however, negligible since upon each new evaluation of an honest party, the value y simulated by $\mathcal{S}$ is chosen uniformly at random. The probability of a collision with any previously queried value $x' = (\texttt{CONST}_i \,||\, y')$ is negligible. As long as the simulator does not abort, it exactly mimics $\pi_{\widetilde{\mathsf{VRF}}}$: it internally simulates the underlying hybrid

StakingProcedure(...)

The following staking procedure is executed by party p. We highlight the usage of the VRF functionality and how the block is created.

Send $(\mathsf{EvalProve}, sid, \eta_j \,\|\, \mathtt{sl} \,\|\, \mathtt{NONCE})$ to $\mathcal{F}_{\mathsf{VRF}}$, denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by $(\mathsf{Evaluated}, sid, y_\rho, \pi_\rho)$.

Send $(\mathsf{EvalProve}, sid, \eta_j \,\|\, \mathtt{sl} \,\|\, \mathtt{TEST})$ to $\mathcal{F}_{\mathsf{VRF}}$, denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by $(\mathsf{Evaluated}, sid, y_T, \pi_T)$.

Send $(\mathsf{EvalProve}, sid, \eta_j \,\|\, \mathtt{sl})$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell, 2\cdot\ell_{\mathsf{VRF}}}$; obtain response $(\mathsf{Evaluated}, sid, (y_\rho, y_T), \pi)$. $(S1)$

if $y_T < T_p^{\mathsf{ep}}$ **then**

⋮

Create new valid content for the block $\mathtt{st}$ (for details see [BGK+18]). ▷ Local ops, party does not lose activation.

⋮

Set $crt = (U_p, y_T, \pi)$, $\rho = (y_\rho, \pi)$ and $h \leftarrow H(\mathtt{head}(\mathcal{C}_{\mathsf{loc}}))$. $(S2)$

Send $(\mathsf{USign}, sid, U_p, (h, \mathtt{st}, \mathtt{sl}, crt, \rho), \mathtt{sl})$ to $\mathcal{F}_{\mathsf{KES}}$; obtain $(\mathsf{Signature}, sid, (h, \mathtt{st}, \mathtt{sl}, crt, \rho), \mathtt{sl}, \sigma)$. ▷ This call returns immediately and the party does not lose activation.

Set $B \leftarrow (h, \mathtt{st}, \mathtt{sl}, crt, \rho, \sigma)$ and update $\mathcal{C}_{\mathsf{loc}} \leftarrow \mathcal{C}_{\mathsf{loc}} \,\|\, B$.

Send $(\textsc{multicast}, sid, \mathcal{C}_{\mathsf{loc}})$ to $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$ and proceed from here upon next activation.

else

...

end if

Figure 3: Staking procedure (excerpt).

VRF functionality and ensures that whenever a proof π is defined to be a valid proof (w.r.t. $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$) for output value y on input m (for some party resp. verification key), then (π, y) is a valid proof for m for the vector $(y_1, \ldots, y_c)$ that $\mathcal{F}_{\mathsf{VRF}}^{\ell,c\cdot\ell_{\mathsf{VRF}}}$ samples for that same party resp. verification key. This establishes the claim. □

Remark. Note that the simulator is responsive. This shows that the VRF functionality can be used in responsive environments, i.e., where the queries to the (dummy) adversary are expected to be answered immediately[2] This is a useful modeling property and we refer to [CEK+16, BGK+18] for the relevant details, as they are outside the scope of this paper.

4 Usage of the Range-Extension Construction in Ouroboros

The purpose of this section is twofold: first, we show how to define formally the staking procedure of Ouroboros using the extended VRF functionlaity and we have to argue about the security. Next, we apply the composition theorem and show how the construction offers room for optimizations. The two most important places where VRF evaluation and verification happens are the staking procedure, cf. Figure 3 (for full details, we refer to the original papers), and the procedure to verify chains, cf. Figure 4, respectively. In each case, we show how the introduction of $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$ affects the code. We depict in gray boxes the original code which is no longer needed and is deleted. The dashed boxes show the effective changes and additions to the code.

Security. The reader might have noticed that we have proven the statement with a slightly different (weaker) VRF functionlaity than what is used in [DGKR18, BGK+18]. The reason is that the range extension does not work for the stronger functionality presented there. However,

[2] That is, without activating any other machine for any other purpose than providing the answer back to $\mathcal{F}_{\mathsf{VRF}}$.

IsValidChain(...)

⋮
Instructions to parse a chain and a first bunch of syntactical validity checks (see [BGK+18] for details).
⋮
for each block B in $\mathcal{C}$ from epoch $\mathtt{ep}$ with epoch randomness η_{ep} **do**

  Parse B as $(h, \mathtt{st}, \mathtt{sl}, crt, \rho, \sigma)$.

  ⋮

  Parse crt as $(U_{p'}, y_T, \pi_T)$ for some p'.

  Parse crt as $(U_{p'}, y_T, \pi)$ for some p'. $(V1)$

  Parse ρ as (y_ρ, π_ρ).

  Parse ρ as VRF output y_ρ. $(V2)$

  Send $(\mathsf{Verify}, sid, \eta_{\mathsf{ep}} \| \mathtt{sl} \| \mathtt{TEST}, y_T, \pi_T, v^{\mathrm{vrf}}_{p'})$ to $\mathcal{F}_{\mathsf{VRF}}$; obtain response $(\mathsf{Verified}, sid, \eta_{\mathsf{ep}} \| \mathtt{sl} \| \mathtt{TEST}, y_T, \pi_T, b_1)$.

  Send $(\mathsf{Verify}, sid, \eta_{\mathsf{ep}} \| \mathtt{sl} \| \mathtt{NONCE}, y_\rho, \pi_\rho, v^{\mathrm{vrf}}_{p'})$ to $\mathcal{F}_{\mathsf{VRF}}$; obtain response $(\mathsf{Verified}, sid, \eta_{\mathsf{ep}} \| \mathtt{sl} \| \mathtt{NONCE}, y_\rho, \pi_\rho, b_2)$.

  Send $(\mathsf{Verify}, sid, \eta_{\mathsf{ep}} \| \mathtt{sl}, (y_T, y_\rho), \pi, v^{\mathrm{vrf}}_{p'})$ to $\mathcal{F}^{\ell, 2\cdot\ell_{\mathsf{VRF}}}_{\mathsf{VRF}}$; obtain $(\mathsf{Verified}, sid, v^{\mathrm{vrf}}_{p'}, \eta_{\mathsf{ep}} \| \mathtt{sl}, (y_T, y_\rho), \pi, b)$. $(V3)$

  Set $\mathsf{badvrf} \leftarrow \left(b_1 = 0 \vee b_2 = 0 \vee y_T \geq T^{\mathsf{ep},\mathcal{C}}_{U_p'}\right)$.

  Set $\mathsf{badvrf} \leftarrow \left(b = 0 \vee y_T \geq T^{\mathsf{ep},\mathcal{C}}_{U_p'}\right)$. $(V4)$

  ⋮

  Further instructions to verify a block (see [BGK+18] for details).

  ⋮

end for
...

Figure 4: Chain validation (excerpt).

the functionality $\mathcal{F}_{\mathsf{VRF}}$ that we put forth here is sufficient to prove the security of Ouroboros by a straightforward inspection of the staking procedure.[3]

Consider Figure 3. First, we observe that thanks to the range extension, we can simply deal with one VRF invocation. The protocol needs two verifiable random values: first the value y_T to determine slot leadership, second the value y_ρ which contributes to the epoch randomness of the future epoch. We obtain both these values in one go from $\mathcal{F}^{\ell, 2\cdot\ell_{\mathsf{VRF}}}_{\mathsf{VRF}}$. The functionality, however, has a weakness: it allows the adversary to learn the output values (y_T, y_ρ), but only *after* the call returned to the party with value $(\mathsf{Evaluated}, sid, (y_\rho, y_T), \pi)$. In other words, the adversary is only able to learn the output values (y_T, y_ρ) from functionality $\mathcal{F}^{\ell, 2\cdot\ell_{\mathsf{VRF}}}_{\mathsf{VRF}}$ (via input $(\mathsf{PastEvaluations}, sid)$ or via a subsequent verification query) only once the party loses or gives up its activation token. The original formulation of $\mathcal{F}_{\mathsf{VRF}}$ in [DGKR18, BGK+18] guaranteed that $\mathcal{F}_{\mathsf{VRF}}$ never by itself would leak this. But now we see that this change is immaterial to the security of Ouroboros: the party, once the values (y_T, y_ρ) are obtained, it never loses the activation until it multicasts the block on the last depicted instruction in Figure 3. At this point, however, the function values are revealed to the adversary "for free", as we multicast the values over the Internet. Since there is no additional security concern regarding verification, we conclude that the introduction of $\mathcal{F}^{\ell, \ell_{\mathsf{VRF}}}_{\mathsf{VRF}}$ is sound.

[3]Note that any VRF that realizes the stronger functionality also realizes the weaker one presented here. Therefore, any previously deployed VRF can be used as the basis of our range-extension construction.

Implementation and Optimization. After showing security, we now can invoke the UC composition theorem by which we can securely replace the modular invocation of $\mathcal{F}_{\mathsf{VRF}}^{\ell,2\cdot\ell_{\mathsf{VRF}}}$ by the construction based on VRF and H. We now showcase what this means for the protocol and how one can apply optimizations at several places. Consider again Figure 3 (and the lines $S1$ and $S2$) as well as Figure 4 (and lines $V1$ to $V4$).

$S1$: This line is implemented by evaluating $(y, \pi) \leftarrow \mathsf{VRF.Eval}(sk_p, \eta_j \,\|\, \mathtt{sl})$ and then defines $y_T \leftarrow \mathsf{H}(\mathtt{TEST} \,||\, y)$ and $y_\rho \leftarrow \mathsf{H}(\mathtt{NONCE} \,||\, y)$.

$S2$: In this line, we can apply an optimization: we can set $crt = (U_p, (y, \pi))$ and set $\rho = \epsilon$ (empty string). The reason is that whenever the protocol needs the verifiable values y_T and y_ρ, they can be computed on-the-fly based on the knowledge of (y, π), i.e., the output $\mathsf{VRF.Eval}(.)$. Thus, storing (y, π) in a block is sufficient. This also means that computing y_ρ above is actually not needed in the staking procedure.

$V1, V2$: Here, we can apply an optimization and in view of the above parse $crt = (U_p, (y, \pi))$ and recompute the values $y_T \leftarrow \mathsf{H}(\mathtt{TEST} \,||\, y)$ and $y_\rho \leftarrow \mathsf{H}(\mathtt{NONCE} \,||\, y)$.

$V3$: This line can be implemented by just computing $b := \mathsf{VRF.Vfy}(v_{p'}^{\mathrm{vrf}}, \eta_{\mathtt{ep}} \,\|\, \mathtt{sl}, y, \pi)$. Since we recomputed the values y_T and y_ρ above in $V1, V2$, $b = 1$ directly implies the validity of y_T and y_ρ for input $\eta_{\mathtt{ep}} \,\|\, \mathtt{sl}$ and w.r.t. verification key $v_{p'}^{\mathrm{vrf}}$.

$V4$: This line is implemented using the recomputed value of y_T.

As a final remark note that when computing the epoch randomness at an epoch boundary based on a sequence of valid blocks, then the contribution of a block $B \leftarrow (h, \mathtt{st}, \mathtt{sl}, crt, \rho, \sigma)$ to the epoch randomness must be recomputed based on $crt = (U_p, (y, \pi))$ analogously to above, i.e., by computing $y_\rho \leftarrow \mathsf{H}(\mathtt{NONCE} \,||\, y)$.

In summary, this shows that we have reduced the number of VRF evaluations (per slot) and VRF verifications (per block) from two to one, at the price of an additional hash function evaluation in each case.

5 The ECVRF Standard

This section recalls the elliptic-curve based schemes described in the `draft-irtf-cfrg-vrf-10` IRTF draft [GRPV] and focuses on the cipher suites `suite_s` $\in \{0x03, 0x04\}$ for the sake of concreteness. We begin by introducing notation and general functions.

5.1 Notation

We denote by $\mathbb{E}(\mathbb{F}_p)$ the finite abelian group based on an elliptic curve over a finite prime-order field $\mathbb{F}_p$ (note that we simplify the notation and drop the explicit dependency on $\mathbb{F}_p$ and security parameter κ). Most importantly, we assume the order of the group $\mathbb{E}$ to be of the form $\mathrm{cf} \cdot q$ for some small *cofactor* cf and large prime number q, and that the (hence) unique subgroup $\mathbb{G}$ of order q is generated by a known base point B, i.e., $\mathbb{G} = \langle B \rangle$ (q is represented by $\approx 2\kappa$ bits) in which the computational Diffie-Hellman (CDH) problem is believed to be hard. Group operations are written in additive notation, scalar multiplication for points $P \in \mathbb{E}$ is denoted by $m * P = \underbrace{P + \cdots + P}_{m}$, and the neutral element by $O = 0 * P$. We use $a \leftarrow_\$ S$ to denote that a is selected uniformly at random from a set S. When working with binary arrays, $a \in \{0,1\}^*$, we denote by $a[X..Y]$ the slice of a from position X till position $Y-1$. Moreover, we denote by $a[..X]$ and $a[X..]$ the slice from position 0 till $X-1$ and from X till the end respectively. As usual, the operator $||$ denotes concatenation of strings, such that, given $A = 0 \,||\, 1$, we have that $A[..1] = 0$ and $A[1..] = 1$.

The standard makes use of helper functions, all of which are defined and introduced in [GRPV]. For sake of simplicity we state the specification of the security-relevant helper functions and show how they are modeled later on in the security proof.

`Hash`: This is a concrete hash function which will be modeled as a general hash function, respectively a random oracle, $\mathsf{H} : \{0,1\}^* \to \{0,1\}^{\ell(\kappa)}$, in the analysis. Conveniently, we choose $\ell(\kappa) = 4\kappa$.

`ECVRF_hash_to_curve`: This is a particular hash function (specified by the cipher suite) that takes an arbitrary string $S \in \{0,1\}^*$ as input, and hashes it to a point in the prime order group $\mathbb{G}$. Specific details of this function can be found in [GRPV]. This function will be modeled as a separate random oracle $\mathsf{H}_{s2c} : \{0,1\}^* \to \mathbb{G}$ in the security proof.

`Expand_key`: This function takes as input a secret seed $sk \in \{0,1\}^{2\kappa}$, and returns a pair $(sk_0, sk_1) \in \{0,1\}^{2\kappa} \times \{0,1\}^{2\kappa}$. The specification prescribes that the seed is hashed $h_{sk} \leftarrow \texttt{Hash}(sk)$, and that the pair $(h_{sk}[..2\kappa], h_{sk}[2\kappa..])$ is returned. The function can thus be modeled as a very simple, random key-derivation function $\mathsf{KDF} : \{0,1\}^{2\kappa} \to \{0,1\}^{4\kappa}$ based on the random oracle directly as $\mathsf{KDF}(sk) := \mathsf{H}(sk)$.

`ComputeScalar`: A helper function used to derive the secret exponent from a (random) bitstring $s \in \{0,1\}^{2\kappa}$. The output domain of this function is a set $S \subseteq [|\mathbb{G}|]$ of size $2^{2\kappa - c}$, for some small constant c, and $\texttt{ComputeScalar}(X)$ is the uniform distribution on S, where X is the random variable with the uniform distribution over 2κ bistrings.

`ECVRF_nonce_generation`: A function that derives a nonce $k \in \mathbb{Z}_q$ from a pair $(sk, H) \in \{0,1\}^{2\kappa} \times \mathbb{E}$. Internally, the algorithm first extends the secret key into a pair of random strings $(sk_0, sk_1) = \texttt{Expand\_key}(sk)$. It then appends to sk_1 the given input, H, in binary form and computes $k \leftarrow \texttt{Hash}(sk_1 \,||\, H)$ (that is, interpreting the bitstring as an integer) and returns $k \mod q$. More details can be found in section 5.4.2.2 of the standard. As we elaborate later, the distribution of the function $\mathsf{RF}^{\text{nonce}}_{sk_1}(H) := \mathsf{H}(sk_1 \,||\, H) \mod q$ derived from a random oracle (again interpreting the output as an integer) has negligible statistical distance to the distribution obtained from choosing a function uniformly at random from the set of all functions $F : \mathbb{E} \to \mathbb{Z}_q$.

`ECVRF_hash_points`: A function that takes as input four EC points, $A_i \in \mathbb{E}$ for $i \in \{1, \ldots, 4\}$, and hashes them (together with some padding), into an integer of κ bits. In more detail, the points are interpreted in binary form and hashes them into a binary array $r \leftarrow \texttt{Hash}(\texttt{suite\_s} \,||\, 0x02 \,||\, A_1 \,||\, A_2 \,||\, A_3 \,||\, A_4 \,||\, 0x00)$ (where the "wrapping" constants are domain separators). Finally, the string $r[..\kappa]$ is returned. This is the helper function to instantiate the Fiat-Shamir heuristic, which computes a challenge in a sigma protocol by hashing the transcript. In the security proof, this will thus be treated as the random-oracle evaluation $\mathsf{H}(\texttt{suite\_s} \,||\, 0x02 \,||\, A_1 \,||\, A_2 \,||\, A_3 \,||\, A_4 \,||\, 0x00)[..\kappa]$. The associated *challenge space* is thus the set $\mathcal{C} := \{0,1\}^{\kappa}$ interpreted as integers.

To give a concrete example, the deployed VRF construction in Cardano is instantiated with $\kappa = 128$ and elliptic curve curve25519 which has cofactor 8. The prime order q is represented by 32 octets, or more precisely 253 bits, and the hash function is $\texttt{SHA512} : \{0,1\}^* \to \{0,1\}^{512}$. For the function $\texttt{ComputeScalar}(sk_0)$, the string is first pruned: the lowest three bits of the first octet are cleared, the highest bit of the last octet is cleared, and the second highest bit of the last octet is set.

This buffer is interpreted as a little-endian integer, forming the secret scalar x, which results in an output domain containing 2^{251} different elements.

5.2 The VRF Algorithms

The formal definition of a VRF in Section 1 describes the Eval function, as the function that computes the output of the VRF evaluation, together with its proof. In this section the two actions are treated separately to follow the approach taken by the standard, and define the functions Prove and $\mathsf{Compute}$ to represent the proof generation, and the output computation respectively. The algorithms from the standard are given as follows:

$\mathsf{Gen}(1^\kappa)$**:** Let $sk \leftarrow_\$ \{0,1\}^{2\kappa}$. Derive a scalar from sk as follows:

1. Let $(sk_0, sk_1) \leftarrow \texttt{Expand\_key}(sk)$.
2. $x \leftarrow \texttt{ComputeScalar}(sk_0)$.

Compute $vk \leftarrow x * B$. Finally, return sk, vk.

$\mathsf{Eval}(sk, X)$**:** The evaluation is a two-stage process:

1. $\pi \leftarrow \mathsf{Prove}(sk, X)$
2. $Y \leftarrow \mathsf{Compute}(\pi)$
3. Return (Y, π)

where:

$\mathsf{Prove}(sk, X)$**:** The proof generation consists of the following steps:

1. Using sk, derive the public key vk, and the secret scalar x, as described in $\mathsf{Gen}(1^\kappa)$.
2. Let $H \leftarrow \texttt{ECVRF\_hash\_to\_curve}(vk, X)$.
3. Let $\Gamma \leftarrow x * H$.
4. Compute a nonce $k \leftarrow \texttt{ECVRF\_nonce\_generation}(sk, H)$.
5. Compute the challenge by hashing the transcript. To this end call the helper function $\texttt{ECVRF\_hash\_points}(H, \Gamma, k * B, k * H)$ and interpret the result as an integer in little-endian representation, c.
6. Compute the response $s \leftarrow (k + c * x) \mod q$
7. Let $\pi \leftarrow \Gamma \,||\, c \,||\, s$.
8. Return π.

$\mathsf{Compute}(\pi)$**:** The VRF output computation goes as follows for a (proof) string $\pi = \Gamma \,||\, ...$:

Precondition: $\Gamma \in \mathbb{E}$.[4]

1. Output $\texttt{Hash}(\texttt{suite\_s} \,||\, 0x03 \,||\, (\mathrm{cf} * \Gamma) \,||\, 0x00)$, where cf is the co-factor (in case of curve25519, this cofactor is 8) and again domain separation is applied.

$\mathsf{Vfy}(vk, X, Y, \pi)$**:** Verification proceeds as follows:

1. Check that $vk \in \mathbb{E}$, and then that $\mathrm{cf} * vk \neq O$, otherwise output 0 and halt.[5]

[4] If not fulfilled an implementation could signal an error by returning an error symbol $\mathsf{ERR} \notin \mathcal{Y}$. For the analysis, this is not needed as the protocol ensures the precondition and the adversary is free to invoke the hash-function at will.

[5] This check excludes low-order elements, i.e., $P \in \mathbb{E}$, $ord(P) < q$.

2. Parse π as tuple (Γ, c, s). Check that $\Gamma \in \mathbb{E}$. If this check fails, output 0 and halt. Interpret the κ bits of c and the 2κ bits of s as little-endian integers. If $s \geq q$, output 0 and halt.
3. Let $H \leftarrow \texttt{ECVRF\_hash\_to\_curve}(vk, X)$.
4. Let $U \leftarrow s * B - c * vk$.
5. Let $V \leftarrow s * H - c * \Gamma$.
6. Let $c' \leftarrow \texttt{ECVRF\_hash\_points}(H, \Gamma, U, V)$
7. If $c = c'$ output $b := (Y = \mathsf{Compute}(\pi))$; otherwise output 0.

6 Batch Verification for ECVRF

In the interest of performance, we study the possibility of batch-verifying the proofs generated by ECVRF. To this end, we describe slight modifications that allow for an efficient batch-verification algorithm. Next, we prove that batch-verification does not affect the security properties of individual proofs, and therefore conclude that the presented modifications in this section does not affect the security properties of Ouroboros. We divide the exposition of the changes in two steps. First, we present the changes on the protocol (involving the prover and the verifier) to make the scheme batch-compatible. Secondly, we describe the specific computation performed by the verifier to batch several proof verifications. We note that a mention of this technique first appeared in the mailing group of the IRTF draft [Rey]. However, as far as we know, there has not been a formal description of this technique, nor the required analysis of such changes. This section covers these gaps.

6.1 Overview

To achieve an efficient batch of the verifications, the single operations which can be improved by computing them for several proofs are steps 4 and 5 of the verification algorithm. We can achieve an important improvement if, instead of computing sequential scalar multiplications, we perform a single multiscalar multiplication for all proofs that are being verified. This batching technique was already introduced by Naccache [NMVR95], and later used by Bernstein [BDL$^+$12] for signature verification batching. However, this trick can only be exploited if steps 4 and 5 are equality checks rather than computations. As it is currently defined, the verifier has no knowledge of points U and V, and computes them with steps 4 and 5. If, contrarily, the prover included points U and V in the transcript and the verifier simply checked for equality, then the multiscalar optimisation could be exploited.

We first describe the changes that make the batching possible and then proceed with the actual description of the batch-verification.

6.2 Making the scheme batch-compatible

As introduced above, in order to allow batch verification, steps 4 and 5 need to be equality checks. This requires a change in step 7 of Prove, changes in steps 2, 4, 5, and 7 of Vfy. Also, we need to move the challenge computation from step 6, to somewhere in between step 3 and 4 (we call it step 3.5). In summary:

$\mathsf{Prove}(sk, X)$: The proof generation is the same except for step 7, which now has to be:

7. Let $\pi \leftarrow \Gamma \,||\, U \,||\, V \,||\, s$

$\mathsf{Compute}(\pi)$**:** The procedure $\mathsf{Compute}$ remains unchanged, as we leave the first element of the proof string unchanged.

$\mathsf{Vfy}(vk, X, Y, \pi)$**:** Verification proceeds as follows:

1. As before.
2. Parse π as tuple (Γ, U, V, s). Check that $\Gamma, U, V \in \mathbb{E}$. If this is not the case, output 0 and halt. Interpret the 2κ bits of s as a little-endian integer. If $s \geq q$, output 0 and halt.
3. As before.

3.5. Let $c \leftarrow \texttt{ECVRF\_hash\_points}(H, \Gamma, U, V)$.

4. Check if $U = s * B - c * vk$.
5. Check if $V = s * H - c * \Gamma$.
6. [Moved to step 3.5]
7. If both equality checks in steps 4. and 5. succeeded, output $b := (Y = \mathsf{Compute}(\pi))$; otherwise output 0.

This change has no implications on the security of the scheme. Note that it is common for (Fiat-Shamir-transformed) Σ-protocols to send the commitment of the randomness (sometimes called the announcement) instead of the challenge.[6] Sending the challenge instead of the two announcement is simply a communication complexity and efficiency decision. In the sequel, we refer to the scheme that includes the above modification by ECVRF.

6.3 Batch-Verification

The changes described above allow for batch verification. To see how this is possible, we first note how steps 4 and 5 can be combined into a single check. In particular if steps 4 and 5 validate, then so does the following equation:

$$O = r * (s * B - c * vk - U) + l * (s * H - c * \Gamma - V)$$

where r, l are random scalars chosen by the verifier. The reverse is also true with overwhelming probability, given that r, l are taken uniformly at random from a set of sufficient size (in particular, we choose the set $\mathcal{C}$ for convenience). Using the state of the art multi scalar multiplication algorithms, using this trick for batch verification considerably improves the running times. In particular, assume that there are n different ECVRF proofs to verify. The verifier needs to check if the following equality relations

$$U_i = s_i * B - c_i * vk_i,$$
$$V_i = s_i * H_i - c_i * \Gamma_i$$

hold for each of the proofs. This can be merged into a single equality check

$$O = r_i * (s_i * B - c_i * vk_i - U_i) + l_i * (s_i * H_i - c_i * \Gamma_i - V_i)$$

for $i \in [1, n]$. The performance boost comes when we combine all these individual checks into a single verification. In particular, the verifier could compute the following single check

$$O = \sum_{i \in [1,n]} (r_i * (s_i * B - c_i * vk_i - U_i) + l_i * (s_i * H_i - c_i * \Gamma_i - V_i))$$

[6] As a matter of fact, ed25519 [BDL+12] is also a sigma protocol and encodes the announcement instead of the challenge in the non-interactive variant of this sigma-protocol.

where r_i and l_i are random scalars.

Note that when a batch is invalid then we need to break down the batches to determine which is the invalid proof. However, in several practical cases (such as validating the state of a blockchain), when multiple VRFs need to be validated in a batch we expect most of the time all of them to be valid, making this risk reasonable in practice.

To avoid including the additional requirement for the verifier to have a secure source of randomness, we proceed with a description of how the (pseudo) random scalars can be computed using a hash function in the random-oracle model.

Computing Pseudorandom Coefficients. It is of interest to maintain the deterministic nature of the ECVRF verification intact. If for batch verification we require a source of randomness for verifying the equations, this determinism would be lost. Hence, we explore what would be the best way to compute this randomness in a deterministic manner. The important property of this deterministic (pseudo) randomness generation is, similar to the known Fiat-Shamir heuristic for Sigma-protocols, that the value (or seed) *cannot* be known to the prover when defining the proof string. To this end, we compute the pseudo-random scalars by hashing the contents of the proof itself, the value of H for the corresponding public key and an index.

In particular, we compute l_i and r_i as follows. For a batch proof of proofs $\pi_1, \ldots, \pi_n$, we compute, for $i \in [1, n]$:

- $\pi'_i \leftarrow H_i \,||\, \pi_i$,
- $S_T \leftarrow \pi'_1 \,||\, \pi'_2 \,||\, \ldots \,||\, \pi'_n$,
- $l_i \leftarrow \texttt{Hash}(\texttt{suite\_s} \,||\, 0x4c \,||\, i \,||\, S_T \,||\, 0x00)[..\kappa]$, and
- $r_i \leftarrow \texttt{Hash}(\texttt{suite\_s} \,||\, 0x52 \,||\, i \,||\, S_T \,||\, 0x00)[..\kappa]$,

where $0x4c$ and $0x52$ stand for "l" and "r", respectively, in the domain separators. l_i and r_i are treated as little-endian integers and are thus picked from the domain $\mathcal{C}$ as the challenge defined earlier. As before, the security analysis will treat both invocations as evaluations of the random oracle $\mathsf{H}(\cdot)$, i.e, as $\mathsf{H}(\texttt{suite\_s} \,||\, 0x4c \,||\, i \,||\, S_T \,||\, 0x00)$ and $\mathsf{H}(\texttt{suite\_s} \,||\, 0x52 \,||\, i \,||\, S_T \,||\, 0x00)$, respectively.

Summary and specification. In summary, batch verification of a sequence of tuples $T_i = (vk_i, X_i, Y_i, \pi_i)$, $i = 1, \ldots, n$, encompasses the following steps:

1. Perform the basic consistency check for each T_i, $i = 1, \ldots, n$:
   - Verify that $vk_i \in \mathbb{E}$ and then that $\mathrm{cf} * vk_i \neq O$.
   - Parse and verify π_i as tuple $(\Gamma_i, U_i, V_i, s_i) \in \mathbb{E}^3 \times \mathbb{Z}_q$ (cf. Section 6.2, Step 2. of $\mathsf{Vfy}(.)$).
   - Compute $H_i \leftarrow \texttt{ECVRF\_hash\_to\_curve}(vk_i, X_i)$.
   - Compute $c_i \leftarrow \texttt{ECVRF\_hash\_points}(H_i, \Gamma_i, U_i, V_i)$.
2. If any of the above check fails then return 0.
3. Perform the batch verification:
   - For all $i \in [n]$ evaluate:
     - Set $\pi'_i \leftarrow H_i \,||\, \pi_i$ for all $i \in [n]$,
     - Let $S_T \leftarrow \pi'_1 \,||\, \ldots \,||\, \pi'_n$,

  - $l_i \leftarrow \mathtt{Hash}(\mathtt{suite\_s} \,||\, 0x4c \,||\, i \,||\, S_T \,||\, 0x00)[..\kappa]$,
  - $r_i \leftarrow \mathtt{Hash}(\mathtt{suite\_s} \,||\, 0x52 \,||\, i \,||\, S_T \,||\, 0x00)[..\kappa]$,

  and interpret l_i, r_i as little-endian integers.

- Evaluate

$$b_1 \leftarrow \left(O = \sum_{i \in [n]} (r_i * (s_i * B - c_i * vk_i - U_i) + l_i * (s_i * H_i - c_i * \Gamma_i - V_i)) \right)$$

4. Evaluate $b_2 \leftarrow (\forall i \in [n] : Y_i = \mathsf{Compute}(\pi_i))$.

5. Output $b_1 \wedge b_2$.

7 Security Analysis of ECVRF and Batch Verifications

We first analyze the security of the standard without batch verifications in the next section and prove the security including batch verifications afterwards.

7.1 Security Analysis of ECVRF

We first recall some preliminaries about zero-knowledge proofs of knowledge for a generic class of protocols.

7.1.1 On Σ-Protocols for Group Homomorphisms

We recall here a general class of zero-knowledge proofs of knowledge, namely the three-round protocols that prove the knowledge of a preimage of a (presumably one-way) group homomorphism [Mau15]. Consider two groups $(\mathbb{H}, \circ)$ and $(\mathbb{T}, \star)$ together with a homomorphism $f : \mathbb{H} \to \mathbb{T}$, i.e.,

$$f(x \circ y) = f(x) \star f(y).$$

Let R_f be the relation defined by $(z, x) \in R_f :\leftrightarrow f(x) = z$. Consider the following three-round protocol between prover P and verifier V for the language $L_{R_f} := \{z \,|\, \exists x : (z, x) \in R_f\}$. That is, the common input is the *proof instance* $z \in \mathbb{T}$ (and the relation R_f), where the prover is supposed to know a value $x \in \mathbb{H}$ s.t. $f(x) = z$.

1. **$P \to V$:** P samples $k \leftarrow_\$ \mathbb{H}$ and sends $t := f(k)$ to V.

2. **$V \to P$:** V picks at random an integer $c \in \mathcal{C} \subset \mathbb{N}$ and sends it to P.

3. **$P \to V$:** P computes $s := k \circ x^c$ and sends s to V. V accepts the protocol run if and only if the equality

$$f(s) = t \star z^c$$

   holds.

The security of this protocol follows from the following lemma:

Lemma 7.1 ([Mau15])**.** *Let R_f a relation as described above relative to a group homomorphism $f : \mathbb{H} \to \mathbb{T}$. The above protocol is a Σ-Protocol for the language L_{R_f} if there are two publicly known values $\ell \in \mathbb{Z}$ and $u \in \mathbb{H}$ s.t.*

1. $\forall c, c' \in \mathcal{C}$, $c \neq c'$: $\gcd(c - c', \ell) = 1$, *and*
2. $f(u) = z^\ell$.

Proof Sketch. We give an outline of the proof of [Mau15]. We need to prove three properties:

- Completeness: The property that on input z and private input x with $(z, x) \in R_f$, then an honest execution always accepts. This is clearly satisfied.
- Special soundness: From any z and any pair of accepting conversations for z denoted $(t, c, s), (t, c', s')$ with $c' \neq c$, one can efficiently compute x such that $(z, x) \in R_f$. The protocol satisfies this. The solution is

$$x := u^a \circ (s'^{-1} \circ s)^b,$$

  and a and b are computed using the Extended Euclidean algorithm (EEA) as solutions to the equation $\ell a + (c' - c)b = 1$ over the integers. Note that $f(s'^{-1} \circ s) = z^{c'-c}$ and

$$f(x) = f(u^a \circ (s'^{-1} \circ s)^b) = f(u)^a \star f(s'^{-1} \circ s)^b = z^{\ell a + (c-c')b} = z.$$

- Special honest-verifier zero-knowledge: the property that there is an efficient simulator S such that on input $z \in L_{R_f}$ and a random challenge $c \in \mathcal{C}$, it generates an accepting conversation (t, c, s) with the same probability distribution as generated by a conversation between honest prover P and honest verifier V on common input z and private input x (s.t. $f(x) = z$) for P. This is achieved by the above protocol: given a challenge c and the statement z, the simulator selects $s \in H$ at random, computes $t := f(s) \star z^{-c}$ and outputs (t, c, s).

This concludes the proof sketch. □

The lemma implies that the protocol is a proof-of-knowledge with knowledge error $1/|\mathcal{C}|$. For our analysis, we only need the implication that if we have a statement $z \notin L_{R_f}$, then the probability that a malicious prover convinces the verifier is at most $1/|\mathcal{C}|$, as in this case, no extractor can exist. We implicitly assume that any fun is rejected if the values do not belong to the expected domain.

On domain checks of the proof instance. The above protocol assumes that the values are indeed in the domain of interest as in particular the existence of values $u \in \mathbb{H}$ and $\ell \in \mathbb{Z}$ Lemma 7.1 could depend on the group order of $\mathbb{T}$ (such as the one discussed below). We need to relax the relation a bit if domain checks on the instance $z \in \mathbb{T}$ are omitted.[7] This is especially relevant if $\mathbb{T}$ is a subgroup of some larger group $\mathbb{T}'$ s.t. the protocol could be run on input $z \in \mathbb{T}' \setminus \mathbb{T}$ by a dishonest party while the verifier does not perform a domain check for $z \in \mathbb{T}$ (but only for $z \in \mathbb{T}'$).

Corollary 7.2. *Consider the Σ-Protocol as in Lemma 7.1 in the above setting, where an honest prover aborts on instances $z \in \mathbb{T}' \setminus \mathbb{T}$ and otherwise executes the protocol. The protocol is a zero-knowledge proof of knowledge for relation R_f as above on instances $z \in \mathbb{T}$, and additionally, it provides special soundness on instances $z \in \mathbb{T}' \setminus \mathbb{T}$ for the relation $(z, x) \in R_{f,e} :\leftrightarrow f(x) = z^e$ if we can fix $u \in \mathbb{H}$ and $\ell \in \mathbb{Z}$ as above such that*

[7] Note that the expected security guarantees indeed become weaker: consider a cyclic group $\langle g \rangle$ of order $2q$ with $q > 2$ and let $\mathbb{T} = \langle h := g^2 \rangle$ be a subgroup together with the homomorphism $f(x) = h^x$ (which is the instantiation to obtain the typical Schnorr DL-proof). A malicious prover might choose the instance $z = h^x \star g^q$ and with probability $1/2$ the challenge c is even in which case the correct answer is $s := k + cx$ as $f(s)$ equals $f(k) \star z^c$. Still z is not a power of h (z has order $2q$) and thus no x can exist such that $(z, x) \in R_f$.

1. $\forall c, c' \in \mathcal{C}$, $c \neq c'$: $\gcd(c - c', \ell) \,|\, e$, *and*

2. $f(u) = z^\ell$.

Proof. We find the greatest common divisor of ℓ and $c' - c$ and let it equal g. We further obtain values a, b s.t. $\ell a + (c' - c)b = g$ by the EEA. By the same reasoning as above, $\tilde{x} := u^a \circ (s'^{-1} \circ s)^b$ satisfies $f(x) = z^g$. Now, we assume that $e = d \cdot g$ for some d, thus $x := \tilde{x}^d$ and $f(x) = f(\tilde{x})^d = z^e$. □

If for each instance $z \in \mathbb{T}'$ we can identify such an exponent e, the protocol can be assumed to be sound for any z in the sense that the probability of passing a protocol run on an instance z such that z^e has no preimage under the homomorphism, is at most $1/|\mathcal{C}|$.

Non-interactive Σ-Protocols. A standard result about Σ-protocols is that they can be made non-interactive (via the Fiat-Shamir transform) in the random-oracle model while preserving soundness and zero-knowledge. Consider the proof w.r.t. a given instance z. A prover P can, instead of sending the first message to the verifier, evaluate $\mathsf{H}(t)$ to obtain a random challenge c and conclude the proof by generating the string s as above. The proof string can be represented by (z, t, s). A verifier can thus verify the proof by calling the oracle on input t to obtain the challenge c and verify as in the protocol above.

Soundness is preserved since talking to the verifier is equivalent to talking to the random oracle. As long as the number of random-oracle queries is limited and the challenge space is larger, soundness is broken with only negligible probability.

Zero-knowledge is preserved since the interaction with the verifier is completely removed and replaced by the random oracle that has the behavior of an honest verifier in Step 2. Note that in the random-oracle model, the simulator is allowed to *pogram* the RO outputs as long as the outputs have the same uniform distribution. Simulation thus works by choosing a challenge c at random, simulate the protocol conversation as above on input z to obtain (t, c, s) and define the oracle's output on input t to be c. The proof string is the tuple (z, t, s). Note that this strategy works as long as the position on a random input t is programmable, which only fails with negligible probability if $|H|$ is large.

The above arguments can be generalized to settings where the instance is not fixed (but for example derived by some context protocol). The above mentioned mapping between (interactive) protocol runs (with an honest verifier) and evaluations of the random oracle is retained when the random oracle is invoked as $\mathsf{H}(aux \,||\, t)$, where aux contains sufficient information to identify the "protocol run" in the above reasoning (which binds the oracle output to a the context such as the instance, the relation etc.). This is of particular importance when proving the security in a composable framework.

7.1.2 Instantiation for ECVRF

We recall that in ECVRF we deal with a prime-order subgroup $\mathbb{G}$ of order q of an elliptic curve of order $\mathrm{cf} \cdot q$. Let B_1 and B_2 be two generators of this subgroup. Essentially, the Σ-protocol of interest is an equality proof of discrete logarithm, i.e., given two values z_1 and z_2 prove knowledge of x such that $x * B_1 = z_1 \wedge x * B_2 = z_2$.

To instantiate the above generic scheme, we let $\mathbb{H} := (\mathbb{Z}_q, +)$ and define $(\mathbb{T}, \oplus) := (\mathbb{G}, +) \times (\mathbb{G}, +)$ as the direct product of $\mathbb{G}$, where the binary operation $\oplus$ on $\mathbb{T}$ is defined component-wise. The homomorphism is given by

$$f_{B_1,B_2} : \mathbb{Z}_q \to \mathbb{T}; \quad x \mapsto (x * B_1, x * B_2),$$

as obviously, $((x+y)*B_1,(x+y)*B_2) = (x*B_1+y*B_1, x*B_2+y*B_2) = (x*B_1, x*B_2)\oplus(y*B_1, y*B_2)$, and the relation $R_{B_1,B_2} \subseteq \mathbb{T} \times \mathbb{Z}_q$ is formally defined by

$$((z_1, z_2), x) \in R_{B_1,B_2} :\leftrightarrow x * B_1 = z_1 \wedge x * B_2 = z_2. \tag{1}$$

Since $\mathbb{G}$ is of prime order q, we can satisfy the conditions of Lemma 7.1 by letting $u = 0$ and $\ell = q$, and defining the challenge space to be a large subset $\mathcal{C} \subseteq [0, \dots, q-1]$. We therefore conclude that the embedded non-interactive zero-knowledge proof of knowledge in ECVRF has (in the random-oracle model) simulatable executions, and with only negligible probability can a valid proof for a wrong statement be generated.

As for the above mentioned domain checks, we conclude that the embedded protocol, without having the verifier check that $z \in \mathbb{T}$, we fall into the realm of Corollary 7.2 (where instances (z_1, z_2) are checked to merely belong to $\mathbb{E}\times\mathbb{E}$). Therefore, since the elliptic curve group $\mathbb{E}$ satisfies $|\mathbb{E}| = \mathrm{cf}\cdot q$ (with $\mathrm{cf} = 8$ in the concrete case of curve25519) we can pick $\ell = \mathrm{cf}\cdot q$ and thus obtain the guarantees from Corollary 7.2 for the choice $e = \mathrm{cf}$, that is for the relation $R^{\mathrm{cf}}_{B,H} \subseteq \mathbb{E}\times\mathbb{E}$ (and B, H generators of subgroup $\mathbb{G}$), defined by

$$(z_1, z_2) \in R^{\mathrm{cf}}_{B,H} :\leftrightarrow x * B = \mathrm{cf} * z_1 \wedge x * H = \mathrm{cf} * z_2. \tag{2}$$

The canonical epimorphism. Viewed from a different angle, Corollary 7.2 is a general statement that says that the verification equations of a particular run of the protocol can be interpreted in a different but related way (that might depend on the order of the particular instance) for which it constitutes a proof of knowledge. For finite elliptic curve groups as above, we can see that any run of the protocol can be interpreted in group $\mathbb{G}$: Consider the map $P \mapsto \mathrm{cf} * P$ which is the canonical epimorphism $\phi_{\mathrm{cf}} : \mathbb{E} \to \mathbb{G}$ and the corresponding map $P + \ker(\phi_{\mathrm{cf}}) \mapsto \phi_{\mathrm{cf}}(P)$ which identifies the isomorphism establishing $\mathbb{E}/\ker(\phi_{\mathrm{cf}}) \cong \mathbb{G}$ by the fundamental theorem on homomorphisms. From this we can deduce by Lagrange's Theorem that $|\mathbb{E}| = |\mathbb{G}| \cdot |\ker(\phi_{\mathrm{cf}})|$. Since the choice of the representatives is immaterial one can think of each coset $P + \ker(\phi_{\mathrm{cf}})$ to be represented by a point $P \in \mathbb{G}$ (and the kernel consists of the low-order points, i.e., elements of order strictly less than q).

Denoting the first round message of the prover by (U, V), the projected verification equation in step 3 of the Σ-Protocol becomes $(O, O) = (\phi_{\mathrm{cf}}(s * B - U - c * z_1), \phi_{\mathrm{cf}}(s * H - V - c * z_2))$ which is an equation in the prime-order group $\mathbb{T}$. More generally speaking, the above equality is satisfied when, in a run of the given Σ-protocol, it holds that $(s * B - V - c * z_1) \in \ker(\phi_{\mathrm{cf}})$ and $(s * H - V - c * z_2) \in \ker(\phi_{\mathrm{cf}})$. By the reasoning in the proof of Lemma 7.1, from any two runs (with the same first round message) that are accepting under the mapping ϕ_{cf}, we can extract a solution x for which $(x * \phi_{\mathrm{cf}}(B), x * \phi_{\mathrm{cf}}(H)) = (\phi_{\mathrm{cf}}(z_1), \phi_{\mathrm{cf}}(z_2))$. Since B and H are known generators of group $\mathbb{G}$, the above identification of the associated isomorphism implies $\phi_{\mathrm{cf}}^{-1}(\phi_{\mathrm{cf}}(B)) = B$ and $\phi_{\mathrm{cf}}^{-1}(\phi_{\mathrm{cf}}(H)) = H$ and in the other case, we have $\phi_{\mathrm{cf}}^{-1}(\phi_{\mathrm{cf}}(z_i)) \in P_i + \ker(\phi_{\mathrm{cf}})$ for representatives $P_i \in \mathbb{G}$. In summary, this establishes special soundness with respect to the relation

$$(z_1, z_2) \in R^{\mathrm{cf}}_{B,H} :\leftrightarrow x * B = \phi_{\mathrm{cf}}(z_1) \wedge x * H = \phi_{\mathrm{cf}}(z_2) \tag{3}$$

for the Σ-protocol above, where we could relax the checks performed by the verifier to $(s * B - V - c * z_1) \in \ker(\phi_{\mathrm{cf}})$ and $(s * H - V - c * z_2) \in \ker(\phi_{\mathrm{cf}})$ instead of equality checks $(s * B - V - c * z_1) = O$ and $(s * H - V - c * z_2) = O$.

7.1.3 The UC Construction Statement

Recall from Section 1 how any VRF can be understood as a UC protocol. We now show the security of the ECVRF protocol without the batching step, but already with the (minor) modifications introduced in Section 6.2. We work in the random-oracle model, that is, the two general functions H (abstracting the details of `Hash`) and H_{s2c} (abstracting the details of `ECVRF_hash_to_curve`) are represented by two instances of the random oracle functionality, which are $\mathcal{F}_{\mathrm{RO}}^{\mathcal{Y}}$, for $\mathcal{Y} = \{0,1\}^{\ell_{\mathsf{VRF}}}$, and $\mathcal{F}_{\mathrm{RO}}^{\mathbb{G}}$, respectively, and invocations of H and H_{s2c} correspond to invocations of the respective functionalities as explained in Section 1.

Theorem 7.3. *Let $\mathbb{E}$ and its prime-order subgroup $\mathbb{G}$ be defined as in Section 5.1. The protocol π_{ECVRF} UC-realizes $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$, for $\ell = \{0,1\}^*$ and $\ell_{\mathsf{VRF}}(\kappa) = 4\kappa$, in the random-oracle model and under the assumption that the CDH problem is hard in $\mathbb{G}$.*

We note that the theorem translates to the unmodified algorithms by converting proof strings of the form $\pi = (\Gamma, c, s)$ for a VRF evaluation (vk, m, y) to proof strings of the form $\pi' = (\Gamma, U, V, s)$ which is straightforward to do as explained before.

Proof. We first describe the simulator and include in its description a variety of consistency checks. We later argue that the simulation is identical to the real-world execution, until the point where a consistency check fails. We then bound the associated probabilities of these *bad events*.

Description of the simulator. We now describe the simulator $\mathcal{S}$ for the construction. While formally the simulator simulates two instances of the random-oracle functionality towards $\mathcal{Z}$, we keep the notation H_{s2c} and H for simplicity. $\mathcal{S}$ maintains two tables T_{s2c} and T_h to store the mapping corresponding to the ideal function implemented by the RO. We use T_z to store all instances of completed VRF evaluations and their associated proofs (mirroring what the functionality stores) and T_{exp} to store the random base points H assigned to pairs (v, m) together with its exponent w.r.t. base B of the group $\mathbb{G}$. We further keep a table $T_{\mathrm{Disallowed}}$ to store information on which outputs of the RO yield inconsistent simulations. Finally, we have T_{pk} to store the mapping of honest users to public keys and we store private parameters of honest parties in T_{priv}.[8]

- On receiving $(\mathsf{KeyGen}, sid, U_i)$ from $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$: Pick three random strings, $s, s_0, s_1 \in \{0,1\}^\kappa$. Compute the scalar x from s_0 as in the real world and define the public key $v \leftarrow x * B$. Evaluate $\mathsf{KGENFAIL} \leftarrow \exists i : T_{\mathrm{pk}}[i] = (\cdot, v)$ and abort if true. Otherwise, store the tuple $(\mathsf{sk}, U_i, s, s_0, s_1, x)$ in T_{priv} and (U_i, v) in T_{pk} and provide the input $(\mathsf{VerificationKey}, sid, U_i, v)$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$.
- On receiving $(\mathsf{EvalProve}, sid, U_i, m)$ from $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$ the following steps are preformed:
  1. Obtain the entry (U_i, v) from T_{pk}.
  2. If for this honest party U_i we have $(v, m, \cdot, \pi) \in T_z$, then return $(\mathsf{EvalProve}, sid, U_i, m, \pi, 1)$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$. Otherwise, proceed to the next step.
  3. Invoke $\mathsf{H}_{s2c}(v, m)$ (i.e., make a simulated RO call) to obtain the instance base point H and retrieve the tuple (v, m, H, B, t) from T_{exp}, where $H := t * B$ (which is guaranteed to exist after the RO call). Define $\Gamma := t * v$.
  4. At this point, the statement and the relation of the NIZK proof are defined: $z = (v, \Gamma)$ and the relation is defined by $R_{B,H}$ as defined in equation (1).

[8]Looking ahead, this distinction is crucial when arguing security. The simulation is design such that except for corruption queries, the set T_{priv} is not used in the simulation. In particular, if party U_i is never corrupted, knowledge of its secret key is not required for a correct simulation.

5. The proof string π is now simulated as explained in Section 7.1.1: For the above relation, this means we pick random $s \in \mathbb{Z}_q$ and $c \in \mathcal{C}$, compute $t = (U, V) \leftarrow (s * B - c * v, s * H - c * \Gamma)$, and define $\pi := \Gamma \,||\, U \,||\, V \,||\, s$.
6. Program the RO: evaluate $\mathsf{EVALFAIL}_1 \leftarrow (T_h[\texttt{suite\_s} \,||\, 0x02 \,||\, H \,||\, \Gamma \,||\, U \,||\, V \,||\, 0x00] \neq \perp)$. If $\mathsf{EVALFAIL}_1$ holds, then abort the simulation, otherwise pick $r \leftarrow_\$ \{0,1\}^{3\kappa}$ and assign $T_h[\texttt{suite\_s} \,||\, 0x02 \,||\, H \,||\, \Gamma \,||\, U \,||\, V \,||\, 0x00] \leftarrow c \,||\, r$ (where c is represented as a bitstring).
7. Evaluate $\mathsf{EVALFAIL}_2 \leftarrow \exists (v', m', \cdot, \mathcal{P}') \in T_z$ such that $\pi \in \mathcal{P}' \wedge ((v' \neq v) \vee (m' \neq m))$. Abort if $\mathsf{EVALFAIL}_2$ holds (proof is not unique).
8. If $(v, m, \cdot, \cdot) \notin T_z$ then insert $(v, m, ?, \{\pi\})$ into T_z. Otherwise retrieve the entry of the form $(v, m, y, \mathcal{P})$ and update it to $(v, m, y, \mathcal{P} \cup \{\pi\})$.
9. Store $(\mathsf{proof}, U_i, H, s, c)$ in T_{priv}.
10. Return $(\mathsf{EvalProve}, sid, U_i, m, \pi)$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell, \ell_{\mathsf{VRF}}}$.

- On receiving $(\mathsf{Verify}, sid, m, y', \pi, v', S_{\text{eval}})$ from $\mathcal{F}_{\mathsf{VRF}}^{\ell, \ell_{\mathsf{VRF}}}$, do the following:

1. Parse π as four values $(\Gamma, U, V, s) \in \mathbb{E}^3 \times \mathbb{Z}_q$ and verify that the order of v' is at least q. If these conditions are not satisfied but $(v', m, y', \mathcal{P}) \in T_z$ with $\pi \in \mathcal{P}$, then $\mathsf{VERFAIL}_1 \leftarrow 1$ and the simulation is aborted. Otherwise return $(\mathsf{Verified}, sid, m, y, \pi, v', 0)$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell, \ell_{\mathsf{VRF}}}$.
2. Make a call to $\mathsf{H}_{s2c}(v', m)$ to obtain the base point H. Retrieve the associated exponent t from T_{exp}. Invoke $\mathsf{H}(\texttt{suite\_s} \,||\, 0x02 \,||\, H \,||\, \Gamma \,||\, U \,||\, V \,||\, 0x00)$ to derive challenge c.
3. Evaluate the truth value of the proof string: $\texttt{f}_\pi \leftarrow (s * B = U + c * v') \wedge (s * H = V + c * \Gamma)$. Evaluate $\mathsf{VERFAIL}_2 \leftarrow (\texttt{f}_\pi = 0) \wedge (v', m, y', \mathcal{P}) \in T_z$ with $\pi \in \mathcal{P}$. Abort the simulation if $\mathsf{VERFAIL}_2$ holds.
4. If $\texttt{f}_\pi = 0$ then return $(\mathsf{Verified}, sid, m, y, \pi, v', 0)$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell, \ell_{\mathsf{VRF}}}$.
5. At this point we have a claimed instance (v', Γ), and a valid proof π for the claim $(v', \Gamma) \in L_{R^{\text{cf}}_{B,H}}$ where the relation is defined by equation (2). Define $\mathsf{VERFAIL}_3 \leftarrow t * (\text{cf} * v') \neq (\text{cf} * \Gamma)$. Abort if $\mathsf{VERFAIL}_3$ holds.
6. If $(v', m, \cdot, \cdot) \in T_z$, then make an internal call to $\mathsf{H}(\texttt{suite\_s} \,||\, 0x03 \,||\, (\text{cf} * \Gamma) \,||\, 0x00)$ to obtain the hash y and go to the next step. Otherwise, let $\mathsf{VERFAIL}_4 \leftarrow (v', m, \cdot, \cdot) \notin T_z \wedge T_h[\texttt{suite\_s} \,||\, 0x03 \,||\, (\text{cf} * \Gamma) \,||\, 0x00] \neq \perp$, abort if the condition holds and else set $y \leftarrow \perp$ and set $T_{\text{Disallowed}} \leftarrow T_{\text{Disallowed}} \cup \{(\text{cf} * \Gamma, y')\}$.
7. Evaluate $\mathsf{VERFAIL}_5 \leftarrow (y = y') \wedge \exists (v'', m'', \cdot, \mathcal{P}'') \in T_z$ such that $\pi \in \mathcal{P}'' \wedge ((v'' \neq v) \vee (m'' \neq m))$. Abort if $\mathsf{VERFAIL}_5$ holds (proof is not unique).
8. If $y = y'$ then retrieve the record $(v', m, y', \mathcal{P}) \in T_z$ (for some $\mathcal{P}$), update the entry to $(v', m, y', \mathcal{P} \cup \{\pi\})$ and return $(\mathsf{Verified}, sid, m, y, \pi, v', 1)$. Otherwise the simulator returns $(\mathsf{Verified}, sid, m, y, \pi, v', 0)$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell, \ell_{\mathsf{VRF}}}$.

The simulation for the random oracle is done as follows:

- **Invocation of H_{s2c} on input $s \in \{0,1\}^*$:**

If $s = (v, m) \in \{P \in \mathbb{E} : ord(P) \geq q\} \times \{0.1\}^\ell$: If $T_{s2c}[(v, m)] \neq \perp$, return $T_{s2c}[(v, m)]$. Otherwise, pick a random $t \in \mathbb{Z}_q$, define $H := t * B$, and store (v, m, H, B, t) in T_{exp}. Define $\mathsf{ROCOL} \leftarrow (\exists i, j, i \neq j, T_{s2c}[i] = (\cdot, \cdot, H_i, \cdot, \cdot), T_{s2c}[j] = (\cdot, \cdot, H_j, \cdot, \cdot) : H_i = H_j)$, define $\mathsf{ROIDENT} \leftarrow \exists i : T_{s2c}[i] = (\cdot, \cdot, H_i, \cdot, \cdot) \wedge ord(H_i) = 1$.

Else: If $T_{s2c}[s] = \bot$, pick $H \leftarrow_\$ \mathbb{G}$ and set $T_{s2c}[s] \leftarrow H$. Return $T_{s2c}[s]$.

- **Invocation of** H **on input** $s \in \{0,1\}^*$**:**

If $s = (\texttt{suite\_s} \,||\, 0x03 \,||\, P \,||\, 0x00)$**,** $P \in \mathbb{G}$**:** Perform the following steps:

1. Ensure consistency with the functionality:
   (a) If this is an internal call, the set S_{eval} is provided by the functionality as part of the most recent input.[9] Otherwise, the set S_{eval} is obtained via querying $(\mathsf{PastEvaluations}, sid)$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$.
   (b) Let the entries of T_{exp} be denoted by (v_i, m_i, H_i, B, t_i).
   (c) Define $S := \{(v_i, m_i, H_i, B, t_i) \in T_{exp} \,|\, t_i * (\text{cf} * v_i) = P\}$.
   (d) Evaluate $\mathsf{ROFAIL}_1 \leftarrow |S| > 1$ and abort if ROFAIL_1 holds.
   (e) **If** $S = \emptyset$:
      i. If $T_h[s] = \bot$, assign y to a random value in $\{0,1\}^{\ell_{\mathsf{VRF}}}$.
      ii. Otherwise, let $y \leftarrow T_h[s]$.
   (f) **If** $S = \{(v, m, H, t)\} \wedge (v, m, \cdot) \in T_z$:
      i. If there is an entry $(v, m, y', \mathcal{P}) \in T_z$ for $y' \in \{0,1\}^{\ell_{\mathsf{VRF}}}$, then set $y \leftarrow y'$.
      ii. Otherwise, find $(v, m, y') \in S_{\text{eval}}$ and update the entry $(v, m, ?, \mathcal{P})$ in T_z to $(v, m, y', \mathcal{P})$. Set $y \leftarrow y'$.
   (g) **If** $S = \{(v, m, H, t)\} \wedge (v, m, \cdot) \notin T_z$; do the following:
      i. If $(\cdot, v) \notin T_{\text{pk}}$, then send $(\mathsf{KeyGen}, sid, v)$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$ and add $(\mathcal{S}, v)$ to T_{pk}.
      ii. Set $\mathsf{ROFAIL}_2 \leftarrow 1$ if $(U_i, v) \in T_{pk}$ for U_i that is not corrupted. Abort if ROFAIL_2 holds.
      iii. At this point, send $(\mathsf{Eval}, sid, v, m)$ to $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$ and obtain the result $(\mathsf{Evaluated}, sid, y')$, $y \leftarrow y'$ and add $(v, m, y, \emptyset)$ to T_z.
2. Evaluate $\mathsf{ROFAIL}_3 \leftarrow T[s] \neq \bot \wedge T[s] \neq y$. Abort if ROFAIL_3 holds.
3. If $T_h[s] \neq \bot$, return $T_h[s]$. Otherwise, set $T_h[s] \leftarrow y$
4. Evaluate $\mathsf{ROFAIL}_4 \leftarrow (P, y) \in T_{\text{Disallowed}}$. Abort if ROFAIL_4 holds.
5. Return y.

If $s = (\texttt{suite\_s} \,||\, 0x02 \,||\, H \,||\, \Gamma \,||\, U \,||\, V \,||\, 0x00), (H, \Gamma, U, V) \in \mathbb{E}^4$**:** If $T_h[s] \neq \bot$, return $T_h[s]$. Otherwise, pick a random challenge c and an additional random string $r \leftarrow_\$ \{0,1\}^{3\kappa}$ and assign $T_h[s] \leftarrow c \,||\, r$ (where c is represented as a bitstring).

Else: If $T_h[s] = \bot$, pick y at random from the set $\{0,1\}^\kappa$ and set $T_h[s] \leftarrow y$. Return $T_h[s]$.

- **Upon corruption of party** U_i**:** Retrieve the record $(\mathsf{sk}, U_i, s, s_0, s_1, x)$ and all records of the form $(\mathsf{proof}, U_i, H, s, c)$ from T_{priv} and ensure a correct programming of the RO as follows:

  1. If $T_h[s] \neq \bot$ then set $\mathsf{CORRFAIL}_1 \leftarrow 1$ and abort. Otherwise, set $T_h[s] \leftarrow s_0 \,||\, s_1$.
  2. If $T_h[x] \neq \bot$ for some $x = s_1 \,||\, x'$ then set set $\mathsf{CORRFAIL}_2 \leftarrow 1$ and abort. Otherwise, for each record $(\mathsf{proof}, U_i, H, s, c)$ program the RO as follows:
     (a) Compute the nonce as $k \leftarrow s - cx$.

[9] Recall that an internal call is a call from within another part of the simulator, in this case from within a verification simulation. Note that this distinction is crucial to achieve a responsive simulator, because such a simulator must not activate any other machine before returning the result to a verification request.

(b) Set $T_h[s_1 \,||\, H] \leftarrow_\$ \{n \in [2^{4\kappa} - 1] \,|\, n \mod q = k\}$ (where integers are encoded as bitstrings).

3. Mark U_i as corrupted and return s to the adversary.

This concludes the description of the simulator.

Analysis of the simulation. The failure conditions of the simulator play a crucial role in our argument. Recall that the simulator performs consistency checks, and if they fail to hold, it aborts. We first note that the checks performed by the simulator can be phrased as bad events for both the real and the ideal executions. Recall that the real execution refers to the random experiment where the environment $\mathcal{Z}$ interacts with protocol π_{ECVRF} and the dummy adversary, and the ideal execution refers to the random experiment, where the environment interacts with the ideal protocol for $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$ and the ideal-world adversary (aka simulator) $\mathcal{S}$ as defined above. We define the events in Figure 5 that imply a consistent simulation. We now argue by a that $\mathcal{Z}$'s views in the real and ideal executions are indistinguishable as long as none of the bad events F_x of Figure 5 occur (we denote by $\overline{F_x}$ the complement of F_x). We analyze the different inputs that $\mathcal{Z}$ can provide:

Key Generations: New keys are sampled identically in the real and ideal world and all public keys are unique until the point when bad event F_{KG} occurs. In particular, $\overline{F_{KG}}$ implies $\mathsf{KGENFAIL} = 0$ and the simulation is perfect.

Evaluations: During the proof generation performed by an honest party with public key v on message m, in both worlds, the base point H is derived by an invocation of $\mathsf{H}_{s2c}(v, m)$ which is distributed identically. As long as bad events F_{col} and F_{id} do not occur, both worlds proceed to generating a proof string. If the party has already performed a proof on input m, then in both worlds, the exact same proof string is returned and otherwise, a new base point H is derived in the same way. The proof string consists of four values Γ, U, V, and s which are simulated as derived in Lemma 7.1 (based on a random exponent $k \leftarrow_\$ \mathbb{Z}_q$) unless the random oracle turns out not to be programmable at location (H, Γ, U, V), which can only be if the location has been queried before which is exactly captured by F_{Evl1}. The output distribution in the real world on the other hand is generated using function $\mathsf{RF}^{\text{nonce}}_{sk_1}(H)$, which implies an output distribution on a fresh input H that has a statistical distance of at most $2^{-2\kappa}$ from the uniform distribution on $\mathbb{Z}_q$.[10] Both worlds output this proof string unless it is not unique, which can only happen if bad event F_{Evl2} occurs. Therefore, the simulation is indistinguishable from the real world and does not abort.

Verifications: Consider the tuple $I = (v, m, y, \pi)$ submitted for verification, where $\pi = \Gamma \,||\, \ldots$ is a proof string which is either valid or invalid with respect to (v, m) (recall that Γ and the fixed based point B together with v, m precisely define the instance and the relation of the NIZK). We observe that in both worlds the proof is rejected if it does not have the correct format or the public key v has low order, as long as F_{VF1} does not occur. Furthermore as long as bad event F_{VF2} does not occur, all verification results are consistent, in particular no invalid proof string π has ever be contained in a tuple that has been deemed valid.

We observe that in both worlds as long as F_{VF3} does not occur (i.e., the environment provides a convincing proof of a wrong statement and hence breaks soundness), the tuple I can only successfully verify, if it encodes a valid statement, i.e., by Corollary 7.2 we get that

[10]The skew simply comes from the fact that the cardinalities $|\{n \in [2^{4\kappa} - 1] \,|\, n \mod q = k\}|$, for a given $k \in \mathbb{Z}_q$ where q is a 2κ-bit integer, are not identical as they might differ by at most one.

in this case π correctly asserts the fact that $(v, m, .)$ is such that there is an x such that $x * B = \text{cf} * v \wedge x * H = \text{cf} * \Gamma$, where H is the unique base point associated to (v, m) (unless F_{col} or F_{id} would occur). This in particular implies that as long as F_{VF3} does not occur, the function value y for $(v, m, ., \pi)$ can only be $\mathsf{H}(... \,||\, P \,||\, ...)$ with $P = \text{cf} * \Gamma = x * H$ because there is exactly one $x \in \mathbb{Z}_q$ such that $x * B = \text{cf} * v \in \mathbb{G}$ is fulfilled, where B is the reference base point of $\mathbb{G}$ of order q. We further see that unless F_{VF4} occurs, the function value $y = \mathsf{H}(... \,||\, \text{cf} * \Gamma \,||\, ...)$ has been queried after $\mathsf{H}_{s2c}(v, m)$ was invoked the first time and in this case both worlds do define $\mathsf{H}(... \,||\, \text{cf} * \Gamma \,||\, ...)$ to be the output unless any of the bad events F_{ROFi} occur during the evaluation of the random oracle. And if $\mathsf{H}(... \,||\, \text{cf} * \Gamma \,||\, ...)$ has never been invoked so far, both worlds let the tuple I be deemed invalid unless F_{pred} happens (in which case, the environment predicted a RO output correctly in the real world). Finally, the proof string is unique in both worlds unless F_{VF5} occurs. In conclusion, as long as none of the above bad events occur, we see that both worlds (in particular the ideal world) can deem the tuple valid if the function output y specified in I is the correct value, and there is only one correct value for the function output for (v, m) for this tuple, which is $\mathsf{H}(... \,||\, \text{cf} * \Gamma \,||\, ...)$. In any other case, the tuple will be rejected.

RO queries to H_{s2c}: Both worlds sample random elements with identical distributions, and both worlds return the sampled values as long as F_{col} or F_{id} do not occur.

RO queries to H: For any input other than $x = 0x04 \,||\, 0x03 \,||\, P \,||\, 0x00$, both worlds return consistent function values, which have been sampled uniformly at random. Also, for any fresh input $x = 0x04 \,||\, 0x03 \,||\, P \,||\, 0x00$, both worlds compute uniformly random values to be result of the query (where the simulator either samples on its own or obtains a uniformly random value from $\mathcal{F}_{\mathsf{VRF}}^{\ell, \ell_{\mathsf{VRF}}}$), but the simulator might fail to achieve consistency in which case it aborts. As long as it does not abort, the outputs are thus identically distributed and consistent with the entire $\mathcal{Z}$. To see consistency, we argue as follows:

First, observe that if a point P (from the set of distinct points queried to the random oracle) is associated with a key-message pair (v, m), then this is a valid association, in the sense that valid proof strings $\pi = \Gamma \,||\, \ldots$ can only exist that assert $(v, \Gamma) \in L_{R^{\text{cf}}_{B,H}}$, where $\text{cf} * \Gamma = P$ and H is derived from (v, m). The assignment is unique assuming $\overline{F_{ROF1}}$. Also the converse is true, i.e., at most one of the distinct points P queried to the random oracle can be associated with (v, m) as long as none of the bad events occur. Based on $\overline{F_{col}}$ and $\overline{F_{id}}$ we can assume that H is a generator uniquely associated to (v, m) and we have $b * B = \phi_{\text{cf}}(v)$ for some $b \neq 0$ (since we exclude low-order public keys by conditioning on $\overline{F_{VF1}}$). Excluding soundness failure, in view of equation (3) from any two valid proofs $\pi = \Gamma \,||\, \ldots$ and $\pi' = \Gamma' \,||\, \ldots$ asserting $(v, \Gamma), (v, \Gamma') \in L_{R^{\text{cf}}_{B,H}}$, we conclude using $\phi_{\text{cf}}(\Gamma) = p * B$ and $\phi_{\text{cf}}(\Gamma') = p' * B$ (for some exponents p, p'), that $H = p/b * B = p'/b * B$. Since the computations p/b and p'/b are over $\mathbb{Z}_q$, the uniqueness follows. Therefore, in order to get a consistent simulation, this assignment must be computed by the simulator upon the first invocation of the random oracle that specifies P. In which case, the random oracle is programmed with the output y that a correctly proven VRF evaluation would result in.

This is possible except when (1) (v, m) has never been queried before and v belongs to an honest party (as in this case, the simulator cannot obtain the random value y from the functionality), (2) the point x has been programmed already with a value y' that is inconsistent with what the $\mathcal{F}_{\mathsf{VRF}}^{\ell, \ell_{\mathsf{VRF}}}$ outputs (which happens when the simulator could not associate P to a pair (v, m) upon the first invocation of the form $\mathsf{H}(... \,||\, P \,||\, ...)$.), and (3) if the value y has already been rejected as the function value associated with P during a verification request. In any other case, the output is made consistent with (v, m) , i.e., any valid proof (assuming F_{VF3} does not occur) will assert the function value y as the output associated to (v, m). The conditions (1)-(3) are precisely captured by F_{ROF2}, F_{ROF3}, and F_{ROF4}.

Corruptions of honest parties: When a party is corrupted, its secret key material is leaked, which here is the basic seed s from which all other values are derived. We observe that all values derived from s are explainable as long as we can program the random oracle on the respective domains, which is precisely possible unless any of F_{Corr1} or F_{Corr2} occur.

Bounding the probabilities of bad events. It now remains to bound the probability of a failure due to a bad event being triggered, where, in view of [BR06], a failure can formally be modeled as a "failure flag" which is set when the first bad event specified in Figure 5 is triggered in the execution. As argued above both worlds are indistinguishable until the point of a failure (note that by definition, in any execution, at most one of the defined events can occur as the first bad event triggering the failure). Therefore, we now bound this probability by bounding for each event F_x the probability that F_x occurs as a consequence of an input by the environment issued at some point in the execution where the flag has not been set yet (that is, none of the conditions of any bad event have been fulfilled yet, which we denote by $\overline{F_{x'}}$). Note that by the above analysis, the probability of this is identical in the real and the ideal experiments.

Event F_{KG}: If n denotes the upper bound on the number of public keys in the system, the probability of a collision is upper bounded by $n/2^{2\kappa-c}$, where c is the loss induced by function `ComputeScalar`(.). The number of public keys can be upper bounded by the sum of key generation requests made by $\mathcal{Z}$ and the number of random-oracle queries made by $\mathcal{Z}$ to H_{s2c}.

Event F_{Evl1}: A fresh proof string contains at least the entropy of the nonce, where for example U is a random point in $\mathbb{G}$. If n_h denotes the upper bound on the RO queries, the probability of a collision is at most n_h/q per honest VRF evaluation.

Event F_{Evl2}: A proof string $\pi = \Gamma \,||\, U \,||\, V \,||\, s$ for (v, m) is valid if $(s*B, s*H) = (U + c*v, V + c*\Gamma)$, where H is by assumption the unique point associated with (v, m). Since we deal with an honestly generated proof, the string s is uniformly distributed, and since the RO has not been programmed before, the challenge c is a random challenge.

Assume that there was any other tuple $I = (v', m', y', .)$ with $(v', m') \neq (v, m)$, for which π would satisfy the verification equations. We can assume the base associated to (v', m') to be $H' \neq H$. To pass the associated verification equation, and assuming for simplicity that c' is fixed, we would at least need that $V = s * H' - c' * \Gamma$ which equals $s * H - c * \Gamma$. Now, let $H = h * B$ and $H' = h' * B$ for $h \neq h'$ by assumption. Therefore, $(s \cdot h) * B - (c \cdot h \cdot x) * B = (s \cdot h') * B - (c' \cdot h \cdot x) * B = V$. Since V is a point in $\mathbb{G}$, we thus see that the relation $s \cdot (h' - h) + (c - c') \cdot (h \cdot x) = 0$ must hold over $\mathbb{Z}_q$, which, based on the above, is an equation $S \cdot a_1 + C \cdot a_2 = 0$ for two independent random variables S and C (where the support of C is a subset of the support of S) chosen by the honest verifier conditioned on the other bad events not happening, and fixed $a_1, a_2 \neq 0$. The probability to obtain, in an honest evaluation, a valid proof string for a particular other instance is thus at most $1/q$. The number of instances is upper bounded by the upper bound n_{s2c} on the number of calls to H_{s2c}. In an execution, the probability of event F_{Evl2} can thus be upper bounded by $m \cdot (n_{s2c}/q)$ where m is an upper bound on the number of honest VRF evaluations.

Sim. Check	Corresp. Bad Event	Event occurs when...
KGENFAIL	F_{KG}	$\mathcal{Z}$ provides input (KeyGen, sid) to honest party U_i and the resulting (real or simulated) public key v collides with any previously queried $(v', \cdot), v \in \mathbb{G}$, to H_{s2c}.
EVALFAIL$_1$	F_{Evl1}	$\mathcal{Z}$ provides input $(\mathsf{EvalProve}, sid, m)$ to honest party U_i and the resulting (real or simulated) EC points (H, Γ, U, V) collide with a previous tuple $A_1, \ldots, A_4$ for which $\mathsf{H}(\texttt{suite\_s} \,\|\, 0x02 \,\|\, A_1 \,\|\, \ldots \,\|\, A_4 \,\|\, 0x00)$ has been evaluated.
EVALFAIL$_2$	F_{Evl2}	$\mathcal{Z}$ provides input $(\mathsf{EvalProve}, sid, m)$ to honest party U_i and the resulting (real or simulated) proof string π collides with some proof string π' for which $(\mathsf{Verified}, sid, v', m', y', \pi', 1)$ has been output previously.
VERFAIL$_1$	F_{VF1}	$\mathcal{Z}$ issues $(\mathsf{Verify}, sid, m, y, \pi, v')$ where π is a valid proof w.r.t. (v', m) but has the wrong format or $ord(v') < q$.
VERFAIL$_2$	F_{VF2}	$\mathcal{Z}$ issues $(\mathsf{Verify}, sid, m, y, \pi, v')$ where π is an invalid proof string (w.r.t. (v', m)) for which previously either $(\mathsf{Evaluated}, sid, m, y, \pi)$ has been output to honest party associated with public key v', or $(\mathsf{Verified}, sid, v', m, y, \pi, 1)$ has been output by some honest party.
VERFAIL$_3$	F_{VF3}	$\mathcal{Z}$ issues $(\mathsf{Verify}, sid, m, y, \pi, v')$ where $\pi = \Gamma \,\|\, \ldots$ is a valid proof (w.r.t. (v', m)) but it holds that $(v', \Gamma) \notin L_{R^e_{B,H}}$ for any $e \mid \mathrm{cf}$.
VERFAIL$_4$	F_{VF4}	$\mathcal{Z}$ issues $(\mathsf{Verify}, sid, m, y, \pi, v')$ for a valid proof $\pi = \Gamma \,\|\, \ldots$ (w.r.t. (v', m)) s.t. $(v', \Gamma) \in L_{R^{\mathrm{cf}}_{B,H}}$ and $ord(v') \geq q$ and $H = \mathsf{H}_{s2c}(v', m)$, but there has been a previous call $\mathsf{H}(\texttt{suite\_s} \,\|\, 0x03 \,\|\, \mathrm{cf} * \Gamma \,\|\, 0x00)$ that happened before (v, m) was queried the first time to $\mathsf{H}_{s2c}(.)$.
VERFAIL$_5$	F_{VF5}	$\mathcal{Z}$ issues $(\mathsf{Verify}, sid, m, y, \pi, v')$ for a valid proof $\pi = \Gamma \,\|\, \ldots$ (w.r.t. (v', m)) for which also $\mathsf{H}(\ldots \,\|\, \mathrm{cf} * \Gamma \,\|\, \ldots) = y$ holds, but which collides with some proof string π' for which $(\mathsf{Verified}, sid, v'', m'', y'', \pi'', 1)$ has been output previously for either $v'' \neq v'$ or $m'' \neq m$. *(Here and below we abbreviate the domain separators for the RO by "$\ldots$".)*
-	F_{pred}	$\mathcal{Z}$ issues $(\mathsf{Verify}, sid, m, y', \pi, v')$ for a valid proof $\pi = \Gamma \,\|\, \ldots$ (w.r.t. (v', m)) and $\mathsf{H}(\ldots \,\|\, \mathrm{cf} * \Gamma \,\|\, \ldots)$ has never been called, but $\mathsf{H}(\ldots \,\|\, \mathrm{cf} * \Gamma \,\|\, \ldots) = y'$.
ROCOL	F_{col}	$\mathcal{Z}$ provides an input (v, m) to H_{s2c} that returns a base point H that equals to a previously generated one on input (v', m') for either $v \neq v'$ or $m \neq m'$.
ROIDENT	F_{id}	$\mathcal{Z}$ provides an input (v, m) to H_{s2c} that returns 0, the identity element.
ROFAIL$_1$	F_{ROF1}	$\mathcal{Z}$ makes a call $\mathsf{H}(... \,\|\, P \,\|\, ...)$, $P \in \mathbb{G}$, such that there exist distinct values $H_1 \neq H_2$ and possibly distinct values v_1, v_2, Γ_1, Γ_2 such that $(v_1, \Gamma_1) \in L_{R^{\mathrm{cf}}_{B,H_1}}$ and $(v_2, \Gamma_2) \in L_{R^{\mathrm{cf}}_{B,H_2}}$ with $\mathrm{cf} * \Gamma_1 = P = \mathrm{cf} * \Gamma_2$ and each H_i has been obtained previously by a query to $\mathsf{H}_{s2c}(v_i, m_i)$ for some m_i.
ROFAIL$_2$	F_{ROF2}	$\mathcal{Z}$ makes a call $\mathsf{H}(... \,\|\, P \,\|\, ...)$, $P \in \mathbb{G}$, for which there is a public key $v \in \mathbb{G}$ associated to an honest party U_i and a message m s.t. $\mathsf{H}_{s2c}(v, m) = H$, such that $(v, \mathrm{cf}^{-1} * P) \in L_{R_{B,H}}$ (i.e., $v = x * B \wedge \mathrm{cf} * x * H = P$) but there has never been any output $(\mathsf{Evaluated}, sid, m, \cdot, \cdot)$ toward U_i. (Here, cf^{-1} refers to the multiplicative inverse of cf modulo prime q.)
ROFAIL$_3$	F_{ROF3}	$\mathcal{Z}$ makes a call $\mathsf{H}(... \,\|\, P \,\|\, ...)$, $P \in \mathbb{G}$, such that there is an EC point v' that satisfies for some Γ', $\mathrm{cf} * \Gamma = P$, $(v', \Gamma') \in L_{R^{\mathrm{cf}}_{B,H}}$, and $ord(v') \geq q$ and $(v', \cdot)$ has been queried to H_{s2c} to obtain H, but there has been a previous call $\mathsf{H}(... \,\|\, P \,\|\, ...)$ with the same EC point P, but no such value v' existed at the time of the previous call.
ROFAIL$_4$	F_{ROF4}	$\mathcal{Z}$ makes a call $\mathsf{H}(... \,\|\, P \,\|\, ...)$, $P \in \mathbb{G}$, for a new input point P which hashes to a value y' for which $(\mathsf{Verified}, sid, v, m, y', \pi, 0)$ has been output previously, where $\pi = \Gamma \,\|\, ...$ is a valid proof string and $\mathrm{cf} * \Gamma = P$.
CORRFAIL$_1$	F_{Corr1}	$\mathcal{Z}$ makes a call $\mathsf{H}(s)$ and s equals the secret key (real or simulated) of an honest party.
CORRFAIL$_2$	F_{Corr2}	$\mathcal{Z}$ makes a call $\mathsf{H}(s \,\|\, x)$ for some x, and where s equals the (real or simulated) seed for the nonce generation function.

Figure 5: Definition of events that imply a consistent simulation.

Event F_{VF1}**:** In the real world, the verification algorithm rejects a verification request if the order of the public key is not at least q. Furthermore, the proof string is parsed as a 4 tuple and rejected if not correct. The simulator on the other hand will never evaluate $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$ on any pair (v, m), since those are never added to the set T_{exp} and consequently never added to T_z. Hence, such requests are rejected in both worlds and the probability of this event is 0.

Event F_{VF2}**:** In both worlds, (v, m) maps to a unique base point H. In the ideal world, tuples $(v, m, ., \pi)$ are ever accepted where π fulfills the conditions as stated above for event F_{Evl2}. Second, all proof strings generated on honest evaluations are correct. In summary, if $(v, m, ., \pi)$ does not fulfill the verification equations, then this tuple will never be successfully verified since verification is deterministic. This holds for both the real and ideal worlds. The probability of failure conditioned on the other bad events not occurring is therefore 0.

Event F_{VF3}**:** By definition of the event, we have a pair (v, m), the two bases B and H, and an accepting proof string $\pi = \Gamma \,||\, U \,||\, V \,||\, s$ but (v, Γ) is not in the language of the NIZK. This is bounded by the soundness of the proof scheme: By Section 7.1.1, we can consider every verification request as a proof run between a potentially malicious prover and an honest verifier. Each such run is uniquely identified by the auxiliary information (v, Γ, H, B) and the first message is the pair (U, V). The mapping to the non-interactive version, where the honest verifier is implemented by a random oracle, is generically achieved by evaluating it on the tuple (v, Γ, H, B, U, V) and since by by Corollary 7.2, an invalid instance passes the run with probability at most $1/|\mathcal{C}|$, the same holds for the non-interactive version. Assuming that each pair (v, m) is uniquely mapped to its base point H and since the base point B is fixed throughout, the random oracle can be evaluated on tuple (H, Γ, U, V) to preserve the reasoning from above. By domain separation of this invocation, we observe that obtaining challenges does not interfere with evaluating the VRF. In summary, the probability of this bad event (conditioned on the other bad events not occurring) is upper bounded by $m/|\mathcal{C}|$, where m denotes an upper bound on the number of verification requests.

Event F_{VF4}**:** For this event to happen before any other bad event happens, we assume a fixed point $\mathrm{cf} * \Gamma$ for which the random oracle has been evaluated but there was no pair (v, m) and associated point H, such that v is was detected to satisfy $(v, \Gamma) \in L_{R^{\mathrm{cf}}_{B,H}}$. We are now given a tuple $(v', m', ., \pi)$ and can assume for this case that since $\pi = \Gamma \,||\, ...$ is a valid proof, it holds that $(v', \Gamma) \in L_{R^{\mathrm{cf}}_{B,H}}$.

To bound the probability of this event, we bound the probability that for a fixed $P = \mathrm{cf} * \Gamma = p * B$ for some $p \in \mathbb{Z}_q$, a random oracle call $\mathsf{H}_{s2c}(v', m')$ for a pair that has not been queried before, yields a valid instance for (v', Γ) and relation $R^{\mathrm{cf}}_{B,H}$, where all values are fixed and $H = h * B$ is sampled at random during the RO evaluation. Furthermore, since no other bad event has happened, the random oracle call did not produce the identity element or a collision. Since P is fixed before calling the random oracle, and similarly, $\mathrm{cf} * v = \phi_{\mathrm{cf}}(v') = x * B$ for some x is fixed before evaluating the random oracle, we would need that h satisfies the equation $(x \cdot h) * B = x * H = p * B$ in group $\mathbb{G}$, i.e., $h = p/x$ computed over $\mathbb{Z}_q$ where we need $x \neq 0$ or, equivalently, $v' \notin \ker(\phi_{\mathrm{cf}})$, which holds since we condition on $\overline{F_{VF1}}$ that excludes low-order points. Therefore, the event $h = p/x$ happens with probability at most $1/q$. If n_h denotes the upper bound on random-oracle queries to H and n_{s2c} denotes an upper bound on the number of random-oracle queries to H_{s2c}, we obtain that event F_{VF4} (conditioned on the other bad events not occurring) occurs with probability at most $n_{s2c} \cdot (n_h/q)$.

Event F_{VF5}**:** Here we bound the probability that a proof string $\pi = (\Gamma \,||\, U \,||\, V \,||\, s)$ is valid for (v, m, y) and but we have already previously successfully evaluated tuple (v', m', y, π) where $\mathsf{H}(... \,||\, \mathrm{cf} * \Gamma|...) = y$. Since the proof is valid and none of the other bad events have occurred, we have $(v, \Gamma), (v', \Gamma) \in L_{R^{\mathrm{cf}}_{B,H}}$. Let $\mathrm{cf} * \Gamma = p * B$ for some p.

Since we can assume that F_{VF4} did not occur, we can assume that $\mathsf{H}(... \,||\, \mathrm{cf} * \Gamma \,||\, ...)$ was queried for the first time at some time in the execution at which $H' = \mathsf{H}_{s2c}(v', m')$ as well as $H = \mathsf{H}_{s2c}(v, m)$ are already evaluated. Therefore, we directly reach a contradiction to $\overleftarrow{F_{ROF1}}$. Therefore, the probability of this event conditioned on none of the previous events happening, is 0.

Event F_{pred}**:** The chances that a given y' equals $\mathsf{H}(... \,||\, P \,||\, ...)$ for some P that has never been queried to the random oracle, is $2^{-4\kappa}$. Let m denote the number of verification queries, where each query can be seen as identifying the query P_i and the corresponding guess y'_i. m can be partitioned as the sum $m = m_1 + \cdots + m_j$ where $j = |\{P_1, \ldots, P_m\}$, where m_k is the number of verification requests identifying point P_k. The probability of predicting at least one value correctly is thus upper bounded by $\sum_{k=1}^{j} m_k \cdot 2^{-4\kappa} = m \cdot 2^{-4\kappa}$.

Event F_{col}**:** This is a standard collision probability on outputs of the random oracle H_{s2c} on inputs (v, m) where v is an EC point of order at least q. Conditioned on the event that none of the results are the identity element, if n_{s2c} denotes an upper bound on these queries, the probability of this event can be bounded by $n_{s2c}^2/(q-1)$.

Event F_{id}**:** The probability that any of n_{s2c} queries as above result in the sampling of the identity element of $\mathbb{G}$ is bounded by n_{s2c}/q.

Event F_{ROF1}**:** Recall that any two distinct queries $\mathsf{H}_{s2c}(v_i, m_i)$ and $\mathsf{H}_{s2c}(v_j, m_j)$ result in random base points H_i resp. H_j. In this case, we condition in particular on $\overline{F_{col}}$ and $\overline{F_{id}}$, which means that if we have n_{s2c} distinct queries to the random oracle, this induces a sequence $(h_1, \ldots, h_{n_{s2c}})$ drawn from the set $\mathbb{Z}_q \setminus \{0\}$ without repetition. We now bound the probability that any two positions in this sequence fulfill the relation to provoke the event.

We know that $\mathrm{cf} * v_i = x_i * B$, $\mathrm{cf} * v_j = x_j * B$, for some exponents x_i and x_j. The critical relation is whether the sampled points H_i, H_j, written as $h_i * B$ and $h_j * B$, respectively, satisfy, for certain Γ_i and Γ_j, the equations $(x_i \cdot h_i) * B = \phi_{\mathrm{cf}}(\Gamma_i) = P = \phi_{\mathrm{cf}}(\Gamma_j) = (x_j \cdot h_j) * B$. This implies that $x_i \cdot h_i = x_j \cdot h_j$ over $\mathbb{Z}_q$, or equivalently $h_i/h_j = x_j/x_i$, where x_j, x_i are fixed before sampling h_i and h_j.

Given a fixed coefficient $a_{ij} \in \mathbb{Z}_q$, the probability that the two values h_i, h_j satisfy $h_i = a_{ij} \cdot h_j$ is at most $1/(q-2)$. By the union bound, the probability of provoking F_{ROF1} conditioned on none of the bad events happening is at most $n_{s2c}^2/(q-2)$.

Event F_{ROF2}**:** Assume we have an environment $\mathcal{Z}$ that provokes event F_{ROF2} and no other bad event and denote the probability of this event by ϵ. This means that there is an honest party $\tilde{U}$ with public key $v = x * B$, and a message m s.t. $H = \mathsf{H}_{s2c}(v, m)$, but the party has never evaluated the VRF on input (v, m). In particular, it has never computed the point $\Gamma = x * H$.

Assume $\mathcal{Z}$ provides a point P in such an execution such that $\mathrm{cf} * x * H = P$ holds w.r.t. a key of an honest party $\tilde{U}$. Then, we can construct an algorithm $\mathcal{A}$ that solves the computational Diffie-Hellman problem in group $\mathbb{G}$ with probability at least $\epsilon'(n_h, n_{s2c}, |\mathcal{P}|, c)$, where n_h is an upper bound on the number of random-oracle queries to H, n_{s2c} is an upper bound on the number of random-oracle queries to H_{s2c}, $\mathcal{P}$ is the set of registered parties, and c is the loss induced by function ComputeScalar(.), i.e., the constant such that the size of the support of honestly generated public keys is $2^{2\kappa-c}$.

$\mathcal{A}(P_1, P_2)$ works as follows: it maintains a $|\mathcal{P}| \times n_{s2c}$ matrix M, where the ith row stores all returned queries $\mathsf{H}_{s2c}(v_i, \cdot)$ for the public key associated with party U_i. Furthermore, it stores for all points $P \in \mathbb{G}$ provided in an invocation $\mathsf{H}(... \,||\, P \,||\, ...)$, the point $P' \in \mathbb{G}$ s.t. $\mathrm{cf} * P' = P$ in an array N of size $1 \times n_h$. $\mathcal{A}$ now first picks a random location (i, j) in M, defines $v_i = P_1$, and $M(i, j) = P_2$. It then emulates the ideal world execution towards $\mathcal{Z}$, injecting P_1 as public key and P_2 as random base point $P_2 = \mathsf{H}_{s2c}(P_1, m_j)$, where the tuple $(P_1, m_j, P_2, B, ?)$ is added to T_{exp} since the exponent is not known. Any consistency check done by the simulator that would involve the exponent of P_2 w.r.t. base point B, is set to be satisfied. $\mathcal{A}$ stops the execution when either one of the following stopping conditions occur: (1) $\mathcal{Z}$ corrupts U_i; (2) $\mathcal{Z}$ requests U_i to evaluate the VRF on m; (3) $\mathcal{Z}$ terminates. In any case, the output is determined by picking a random position k in array N and returning $N[k]$.

We observe that conditioned on none of the other bad events occurring during the emulation, the emulation provides, until the point when it stops, an identical view to $\mathcal{Z}$ as the ideal execution as long as no EC point P is provided as input to the random oracle for which $(P_1, P_2, \mathrm{cf}^{-1} * P)$ is a Diffie-Hellman triple: Conditioned on none of the other bad events happening, the computation of the set S defined in step 1(c) of the simulation of the random-oracle query $\mathsf{H}(... \,||\, P \,||\, ...)$ is correct except until the point when the emulation fails to detect the relation $t_H * \mathrm{cf} * v = P$, where t_H is the exponent of $\mathsf{H}_{s2c}(v, m')$ to the base B. Clearly, the emulation only fails to detect the relation w.r.t. P_2 if for some x we have $x \cdot \mathrm{cf} \cdot P_1 = P$ and $P_2 = x * B$. That is the associated point $P' = (x \cdot y) * B$ for $P_2 = x * B$ and $P_1 = y * B$ that we are looking for.

Since by definition of the event, there must be at least one entry (i, j) in matrix M such that (v_i, m) was not evaluated and party U_i is not corrupted, we obtain that the success probability of $\mathcal{A}$ is at least $\epsilon' = \epsilon/(n_h \cdot n_{s2c} \cdot |\mathcal{P}| \cdot 2^c)$, where ϵ is the probability of event F_{ROF2} happening conditioned on none of the other bad events occurring, and where the (small and constant) factor 2^{-c} is due to the probability that a random point P_1 is a valid public key in the correct domain of $\mathsf{Gen}(1^\kappa)$.

Event F_{ROF3}**:** The condition of this event is that a given RO evaluation $\mathsf{H}(... \,||\, P \,||\, ...)$ a subsequent call to $\mathsf{H}_{s2c}(v', m')$ for some v' results in a base point H' from which a valid proof instance (v', Γ') with $\mathrm{cf} * \Gamma' = P$ can be deduced. By definition of the event, P is fixed before any such instance (v', Γ') is known. Therefore, there must have been a fresh call $\mathsf{H}_{s2c}(v', m')$ for some m', which resulted in random base point H'. Since v' is fixed before, the exponent x, such that $\mathrm{cf} * v' = x * B$, is fixed before the point H' is sampled. In order to deduce a valid instance Γ', the relation $x * H' = P$ must hold. Since H and P are elements of $\mathbb{G}$, we write $H' = h' * B$ and $P = p * B$ and see that the relation $(x \cdot h') * B = p * B$ implies that the relation $h' = p/x$ must hold over $\mathbb{Z}_q$. Given an upper bound n_h on the RO queries to H and an upper bound n_{s2c} on the number of RO queries to H_{s2c}, there can be at most n_{s2c} queries to H_{s2c} that could result in any of the relations to hold with any of the at most P points queried before. An upper bound on the probability of the event F_{ROF3} conditioned on no other bad event happening can be obtained by a union bound which yields $n_h \cdot n_{s2c}/(q-1)$.

Event F_{ROF4}**:** Conditioned on $\overline{F_{pred}}$, the probability of F_{ROF4} is 0. The reason is that if $(\mathsf{Verified}, sid, v, m, y', \pi, 0)$ (where $\pi = \Gamma \,||\, ...$) has been output to a party, then the input $(\mathsf{Verify}, sid, m, y', \pi, v)$ must have been given as input which correctly predicted $\mathsf{H}(... \,||\, \mathrm{cf} * \Gamma \,||\, ...)$ before it was called.

Functionality $\mathcal{G}_{\mathsf{BB}}$

The function maintains a (dynamically updatable) list L_s (initially empty). The functionality manages the set $\mathcal{P}$ of registered machines (identified by extended identities), i.e., a machine is added to $\mathcal{P}$ when receiving input REGISTER (and removes a machine from P when receiving DE-REGISTER. The requests give activation back to the calling machine).

- Upon receiving (ADD, sid, x) from $P \in \mathcal{P}$ or from the adversary, set $L \leftarrow L || x$ output $(\mathsf{Updated}, sid, L)$ to the adversary.
- Upon receiving $(\text{RETRIEVE}, sid, i, j)$ from $P \in \mathcal{P}$ or from the adversary, do the following: if $L[j]$ is undefined, return $(i, j, \emptyset)$ to the caller. Otherwise, return the result $(\mathsf{Retrieved}, sid, i, j, L[i] || \dots || L[j])$ to the caller.

Figure 6: The global bulletin board.

Event F_{Corr1}**:** This event only occurs if the environment correctly guesses the secret seed of a party. There are at most $|\mathcal{P}|$ honest parties, and if n_h is an upper bound on the number of RO evaluations to H, the probability of this event is no more than $n_h \cdot |\mathcal{P}| \cdot 2^{-2\kappa}$.

Event F_{Corr2}**:** This event only occurs if the environment correctly guesses the bitstring s_1 of an honest party. Conditioned on $\overline{F_{Corr1}}$, the probability of this event is no more than $n_h \cdot |\mathcal{P}| \cdot 2^{-2\kappa}$.

It is easy to see that all these failure probabilities are negligible in the security parameter. The theorem follows. □

Remark. Note that the simulator is responsive. This shows that the VRF functionality can be used in responsive environments, i.e., where the queries to the (dummy) adversary are expected to be answered immediately.[11] This is a useful modeling property and we refer to [CEK+16, BGK+18] for the relevant details, as they are outside the scope of this paper.

7.2 Security Analysis of ECVRF with Batch Verifications

We first describe the setting and the ideal world that idealizes the security requirements for batch verifications.

7.2.1 The Setting

We model the setting where there is a global database as a reference of stored VRF evaluations and proofs. We call this global bulletin-board functionality $\mathcal{G}_{\text{BB}}$ and describe it in Figure 6. Each instance of the functionality maintains a list of tuples (v, m, y, π). The list is append-only, but there is no other restriction on what to append and thus the only guarantee it offers is that if we refer to an interval $[i..j]$ in the list associated to session *sid*, then, once defined, the result is always the same. The functionality is a global setup [BCH+20], i.e., treated as a global subroutine. It is the reference for claimed VRF proofs connected to a particular session visible and updatable by anyone.

[11] That is, without activating any other machine for any other purpose than providing the answer back to $\mathcal{F}_{\mathsf{VRF}}$.

Ideal Functionality $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}^+}$

$\mathcal{F}_{\mathsf{VRF}}$ interacts with its set of registered parties $\mathcal{P}$ denoted by $U_1, \ldots, U_{|\mathcal{P}|}$ and the adversary/simulator $\mathcal{S}$. It maintains tables $T[\cdot,\cdot]$ that are initially empty (denoted by symbol $\perp$). The tables are initialized on-the-fly. The functionality maintains a set S_{pk} to keep track of registered keys, and S_{eval} to keep track of all known VRF evaluations. The functionality registers to the instance of $\mathcal{G}_{\text{BB}}$ with the same session identifier *sid*.

- **Key Generation.** *As in in Figure 1.*
- **Malicious Key Generation.** *As in Figure 1.*
- **VRF Evaluation and Proof.** *As in Figure 1.*
- **Malicious VRF Evaluation.** *As in Figure 1.*
- **Verification.** *As in Figure 1.*
- **Batch Verification.** Upon receiving a message $(\mathsf{BatchVerify}, sid, i, j)$ from any party, send $(\text{RETRIEVE}, sid, i, j)$ to $\mathcal{G}_{\text{BB}}$ to receive the list $(i, j, L_{i:j})$. Then output $(\mathsf{BatchVerify}, sid, i, j)$ to the adversary. Upon receiving $(\mathsf{BatchVerified}, sid, i, j, b)$ do the following:
  1. If $L_{i:j} = \emptyset$ then return $(\mathsf{BatchVerified}, sid, i, j, 0)$ to the caller.
  2. Parse each entry of $L_{i:j}$ as tuple (m_k, y_k, π_k, v_k) for $k = 1 \ldots |L_{i:j}|$.
  3. Evaluate the condition $f \leftarrow \forall k \in [|L_{i:j}|] : (\cdot, v_k) \in S_{pk} \wedge T(v_k, m_k) = (y_k, S) \wedge \pi_k \in S$. If $f = 1$, return $(\mathsf{BatchVerified}, sid, i, j, 1)$ to the caller.
  4. Evaluate the condition $f' \leftarrow \forall k \in [|L_{i:j}|] : (\cdot, v_k) \in S_{pk} \wedge T(v_k, m_k) = (y_k, \cdot)$. If $f' = 1$ return $(\mathsf{BatchVerified}, sid, i, j, b)$.
  5. Return $(\mathsf{BatchVerified}, sid, i, j, 0)$.
- **Adversarial Leakage.** *As in Figure 1.*

Figure 7: The VRF functionality with Batch Verifications.

7.2.2 The Ideal World

In the ideal world, we introduce a new simple command to the VRF functionality described in Figure 7. Upon input $(\mathsf{BatchVerify}, sid, i, j)$, the functionality retrieves the corresponding list from $\mathcal{G}_{\text{BB}}$ and if the list is non-empty, it verifies whether all claimed combinations are known are stored as valid combinations. In this case the functionality returns 1. If this is not the case, but all pairs (v_i, m_i, y_i) specify the correct input-output-pairs as stored by the functionality, i.e., $T(v_i, m_i) = y_i$, then the functionality lets the adversary decide on the output value. This case captures the fact that although the proofs strings might not be stored in the functionality (or will never be), batch verification will never assert a wrong input-output mapping. In any other case, the output is defined to be 0.

7.2.3 The UC Protocol

Recall from Section 1 that any VRF can be formulated as a UC protocol. We now show how to formulate batch verification as an extended protocol π^+_{ECVRF} that is identical to π_{ECVRF} and additionally implements the following procedure outlined in Section 6.3. To simplify notation, we keep writing H and H_{s2c} for general hash-function invocations and understand that this corresponds to evaluating the random oracles $\mathcal{F}^{\mathcal{Y}}_{\text{RO}}$ and $\mathcal{F}^{\mathbb{G}}_{\text{RO}}$, respectively.

- On input $(\mathsf{BatchVerify}, sid, i, j)$, send $(\textsc{retrieve}, sid, i, j)$ to $\mathcal{G}_{\mathrm{BB}}$ and receive the answer $(\mathsf{Retrieved}, sid, i, j, L_{i:j})$. If $L_{i:j} = \emptyset$ then return $(\mathsf{BatchVerified}, sid, i, j, 0)$. Otherwise, do the following

  1. Parse every item in the list as tuple, i.e., for each $k \in [|L_{i:j}|]$ obtain $T_k = (m_k, y_k, \pi_k, v_k)$. A tuple has the wrong format, return $(\mathsf{BatchVerified}, sid, i, j, 0)$.
  2. For each T_k perform the first steps 1. to 3. and step 3.5 of $\mathsf{ECVRF.Vfy}$, that is:
     – Verify that $v_k \in \mathbb{E}$ and then that $\mathrm{cf} * vk \neq O$.
     – Parse and verify π_k as tuple $(\Gamma_k, U_k, V_k, s_k) \in \mathbb{E}^3 \times \mathbb{Z}_q$.
     – Compute $H_k \leftarrow \mathsf{H}_{s2c}(v_k, m_k)$.
     – Compute $c_k \leftarrow \mathsf{H}(\texttt{suite\_s} \,||\, 0x02 \,||\, H_k \,||\, \Gamma_k \,||\, U_k \,||\, V_k \,||\, 0x00)[..\kappa]$.
  3. If any check fails then return $(\mathsf{BatchVerified}, sid, i, j, 0)$.
  4. Perform the batch verification:
     – Set $\pi'_k \leftarrow H_k \,||\, \pi_k$ for all $k \in [|L_{i:j}|]$.
     – Let $S_T \leftarrow \pi'_1 \,||\, \ldots \,||\, \pi'_{|L_{i:j}|}$.
     – $\forall k \in [|L_{i:j}|] : l_k \leftarrow \mathsf{H}(\texttt{suite\_s} \,||\, 0x4c \,||\, k \,||\, S_T \,||\, 0x00)[..\kappa]$.
     – $\forall k \in [|L_{i:j}|] : r_k \leftarrow \mathsf{H}(\texttt{suite\_s} \,||\, 0x52 \,||\, k \,||\, S_T \,||\, 0x00)[..\kappa]$.
     – Evaluate

$$b_1 \leftarrow \left(O = \sum_{k \in [|L_{i:j}|]} \left(r_k * (s_k * B - c_k * v_k - U_k) + l_k * (s_k * H_k - c_k * \Gamma_k - V_k) \right) \right). \tag{4}$$

  5. Evaluate $b_2 \leftarrow (\forall k \in [|L_{i:j}|] : y_k = \mathsf{Compute}(\pi_k))$.
  6. Define $b \leftarrow b_1 \wedge b_2$ and return $(\mathsf{BatchVerified}, sid, i, j, b)$ to the caller.

7.2.4 The UC Construction Statement

Theorem 7.4. *Under the same assumptions as Theorem 7.3, the protocol π^{+}_{ECVRF} UC-realizes $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}^+}$ (where $\mathcal{G}_{\mathrm{BB}}$ is a global subroutine), for $\ell = \{0,1\}^*$ and $\ell_{\mathsf{VRF}}(\kappa) = 4\kappa$.*

Proof. Consider the simulator in the proof of Theorem 7.3 and denote it $\mathcal{S}_{\mathsf{ECVRF}}$. We build our new simulator $\mathcal{S}^+$ on top of $\mathcal{S}_{\mathsf{ECVRF}}$ as follows: we simulate identically to $\mathcal{S}_{\mathsf{ECVRF}}$ and ensure that at any point in time all combinations stored in $\mathcal{G}_{\mathrm{BB}}$ are verified with the functionality to prepare for batch verifications. We thus get the following simulator $\mathcal{S}^+$:

- On receiving $(\mathsf{KeyGen}, sid, U_i)$ from $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}^+}$ invoke $\mathcal{S}_{\mathsf{ECVRF}}$ on the same input and return to $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}^+}$ whatever $\mathcal{S}_{\mathsf{ECVRF}}$ outputs (and abort if $\mathcal{S}_{\mathsf{ECVRF}}$ aborts).

- On receiving $(\mathsf{EvalProve}, sid, U_i, m)$ from $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}^+}$, invoke $\mathcal{S}_{\mathsf{ECVRF}}$ on the same input and return to $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}^+}$ whatever $\mathcal{S}_{\mathsf{ECVRF}}$ outputs (and abort if $\mathcal{S}_{\mathsf{ECVRF}}$ aborts).

- On receiving $(\mathsf{Verify}, sid, m, y', \pi, v', S_{\mathrm{eval}})$ from $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}^+}$ invoke $\mathcal{S}_{\mathsf{ECVRF}}$ on the same input and return to $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}^+}$ whatever $\mathcal{S}_{\mathsf{ECVRF}}$ outputs (and abort if $\mathcal{S}_{\mathsf{ECVRF}}$ aborts).

- On receiving $(\mathsf{BatchVerify}, sid, i, j)$ from $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}^+}$, retrieve the list $L_{i:j}$ from $\mathcal{G}_{\mathrm{BB}}$ and perform the batch verification steps like the protocol (i.e., emulate the steps from Item 1 to Item 6 of the batch verification) to derive the return value b and return $(\mathsf{BatchVerified}, sid, i, j, b)$ to $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}^+}$. Define $\mathsf{SIMFAIL}^+$ if $b = 1$ but there exists a tuple $(m', y', \pi' = \Gamma' \,||\, \ldots, v'))$ but $\mathsf{Compute}(\pi') \neq y'$. Abort if $\mathsf{SIMFAIL}^+$ occurs.

- On receiving $(\mathsf{Updated}, sid, L)$ from $\mathcal{G}_{\mathrm{BB}}$, $\mathcal{S}^+$ determines all new added tuples T_i of the correct form (m_i, y_i, π_i, v_k) and calls $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}^+}$ with input $(\mathsf{Verify}, sid, m_i, y, \pi, v')$, (which in turn triggers the simulation $\mathcal{S}_{\mathsf{ECVRF}}$ on input $(\mathsf{Verify}, sid, m_i, y_i, \pi_i, v_i, S_{\mathrm{eval}})$ as above). Finally, $\mathcal{S}^+$ outputs $(\mathsf{Updated}, sid, L)$ to the environment.

- **Invocation of H_{s2c} on input $s \in \{0,1\}^*$:** Perform the same actions as $\mathcal{S}_{\mathsf{ECVRF}}$ (abort if $\mathcal{S}_{\mathsf{ECVRF}}$ aborts).

- **Invocation of H on input $s \in \{0,1\}^*$:** First, perform a case distinction on the separated domains $s = (\texttt{suite\_s} \,||\, 0x4c \,||\, S \,||\, 0x00)$ resp. $s = (\texttt{suite\_s} \,||\, 0x52 \,||\, S \,||\, 0x00)$ which are simulated as follows: If $T_h[s] \neq \bot$, return $T_h[s]$. Otherwise, pick a random challenge c and an additional random string $r \leftarrow_\$ \{0,1\}^{3\kappa}$ and assign $T_h[s] \leftarrow c \,||\, r$ (where c is represented as a bitstring). For any other domain, perform the respective actions of $\mathcal{S}_{\mathsf{ECVRF}}$ (abort if $\mathcal{S}_{\mathsf{ECVRF}}$ aborts).

- **Upon corruption of party U_i:** Perform the same actions as $\mathcal{S}_{\mathsf{ECVRF}}$ (abort if $\mathcal{S}_{\mathsf{ECVRF}}$ aborts).

Analysis of the simulation. We first consider the same set of bad events defined in Figure 5, but we formally extend the events F_{VFi} to not only includes queries $(\mathsf{Verify}, sid, m, y, \pi, v')$, made by $\mathcal{Z}$, but also that a tuple of the form $T = (m, y, \pi, v')$ is added as part of a query $(\textsc{add}, sid, T)$ to $\mathcal{G}_{\mathrm{BB}}$.

We first observe that any environment $\mathcal{Z}$ which does not make any invocation of the form $(\mathsf{BatchVerify}, sid, i, j)$ to any honest party and which has non-negligible advantage in distinguishing the real and ideal executions, contradicts Theorem 7.3. Since the only difference between the two executions is the availability of the bulletin board $\mathcal{G}_{\mathrm{BB}}$, we can design an environment $\mathcal{Z}'$ which internally runs $\mathcal{Z}$ and emulates $\mathcal{G}_{\mathrm{BB}}$ towards it, and whenever new updates are pushed on $\mathcal{G}_{\mathrm{BB}}$, $\mathcal{Z}'$ lets the challenge protocol verify these updates. For all other queries, it invokes the main parties of its challenge session. If at any point, the execution is aborted (in which case $\mathcal{Z}'$ must be connected to an ideal execution), the distinguisher outputs 0, and in any other case outputs whatever $\mathcal{Z}$ outputs. Since no other entity ever writes or reads from $\mathcal{G}_{\mathrm{BB}}$ except $\mathcal{Z}$ in the real world, if $\mathcal{Z}'$ interacts with π_{ECVRF} then the view emulated towards $\mathcal{Z}$ is exactly the view it has when interacting with π^+_{ECVRF}. And if $\mathcal{Z}'$ interacts with $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}}$ (and simulator $\mathcal{S}_{\mathsf{ECVRF}}$) then the view emulated towards $\mathcal{Z}$ is exactly the view it has when interacting with $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}^+}$ (and simulator $\mathcal{S}^+$) until the point where a failure event is provoked as defined in Figure 5. Therefore, the distinguishing advantage of $\mathcal{Z}'$ is at least the advantage of $\mathcal{Z}$.

It thus suffices to analyze the real and ideal executions' behavior on inputs $(\mathsf{BatchVerify}, sid, i, j)$. By definition of $\mathcal{S}^+$, whenever a tuple $T_k = (m_k, y_k, \pi_k, v_k)$ is added to $\mathcal{G}_{\mathrm{BB}}$, this is equivalent to have $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}^+}$ verify the tuple (m, y, π, v') (and the verification is identical to the verification of $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}}$). Therefore, in the ideal execution with $\mathcal{F}^{\ell,\ell_{\mathsf{VRF}}}_{\mathsf{VRF}^+}$ and $\mathcal{S}^+$, the output of any query $(\mathsf{BatchVerify}, sid, i, j)$ is 1 if all tuples $T_i, \ldots, T_j$ have been successfully verified.

For the remaining cases, we see that the simulator can decide on the value, with the restriction that the output can only be decided to be 1, if all input-output pairs $((v_i, m_i), y_i)$ are consistent with function table of the $\mathcal{F}_{\mathsf{VRF}^+}^{\ell,\ell_{\mathsf{VRF}}}$.

That is consider the case that we have $b = 1$ upon batch verification and none of the bad events defined in Figure 5 occur. The simulator has made, for each tuple $T_k = (m_k, y_k, \pi_k = \Gamma_k \,||\, \cdot, v_k)$ a call $\mathsf{H}_{s2c}(v_k, m_k)$ (to obtain H_k) and a call $\mathsf{H}(...\,||\,\mathrm{cf} * \Gamma_k \,||\, ...) =$ (to obtain y_k). The latter call associates the point $\mathrm{cf} * \Gamma_k$ with at most one pair (v', m') that satisfies the relation $t' * (\mathrm{cf} * v') = \mathrm{cf} * \Gamma_k$ (where t' is such that $t' * B = H'$), i.e., for which $(v', \Gamma_k) \in R^{\mathrm{cf}}_{B,H'}$.[12]. If such a match is found (and no bad event occurs), the simulation has consistently programmed the random oracle $\mathsf{H}(...\,||\,\mathrm{cf} * \Gamma_k \,||\, ...)$ to match the output of the functionality on a query for (v', m').

Therefore, the computed batch verification value $b = 1$ by the simulator must be accepted by the functionality if for each Γ_k specified in tuple T_k the pair (v', m') that is associated to each Γ_k specified in tuple T_k is exactly the pair (v_k, m_k) listed in tuple T_k. Stated differently, and in view of equation (3), assuming that no tuple breaks the NIZK soundness individually (condition on $\overline{VF_3}$ and since the simulator verifies every proof string added to $\mathcal{G}_{\mathrm{BB}}$), the simulator could only fail to simulate if the entire batch verifies, but for a tuple T_k, with $\phi_{\mathrm{cf}}(v_k) = x_k * B$ for some $x_k \neq 0$[13], we have that $\phi_{\mathrm{cf}}(\Gamma_k) \neq x_k * H_k$, where H_k (conditioned on $\overline{F_{col}}$ and $\overline{F_{id}}$) is the unique generator associated to (v_k, m_k). This motivates the following new bad event F_{Batch} that rules out this case and which implies that the simulator never aborts. Based on the above considerations as long as none of the bad events (including F_{Batch}) occur, $\mathcal{Z}$'s views in the real and ideal executions must be indistinguishable.

New event F_{Batch}. This is the event that $\mathcal{Z}$ provides input $(\mathsf{BatchVerify}, sid, i, j)$, which refers to tuples $T_i, \ldots, T_j$, upon which the computed result is $(\mathsf{BatchVerified}, sid, i, j, 1)$, but at least one of the tuples, say $T_k, i \leq k \leq j$, encodes correctly formatted values (m_k, y_k, π_k, v_k), $\pi_k = \Gamma_k \,||\, \ldots$, such that $y_k = \mathsf{Compute}(\pi_k)$, but $(v_k, \Gamma_k) \notin R^{\mathrm{cf}}_{B,H_k}$ for $H_k = \mathsf{H}_{s2c}(v_k, m_k)$.

Bounding the probability of the new bad event. We now bound the probability of event F_{Batch} to happen conditioned on none of the other bad events occurring. Due to the condition in particular on $\overline{F_{VFi}}$, event F_{Batch} can only be triggered on input $(\mathsf{BatchVerify}, sid, i, j)$, where all tuples $L_{i:j} = T_i, \ldots, T_j$ in $\mathcal{G}_{\mathrm{BB}}$ are defined and well-formed. Furthermore, we can assume that for each $T_k = (m_k, y_k, \pi_k, v_k)$, $\pi_k = \Gamma_k \,||\, U_k \,||\, V_k \,||\, s_k$, it holds that $\mathsf{Compute}(\pi_k) = \mathsf{H}(...\,||\,\mathrm{cf} * \Gamma_k \,||\, ...) = y_k$, as otherwise, the batch verification output is fixed to 0. Likewise, all relations under Item 2 of the batch verification step must hold. Furthermore, since all proof strings to $\mathcal{G}_{\mathrm{BB}}$ are assumed to be implicitly verified, by $\overline{VF4}$, no tuple added to $\mathcal{G}_{\mathrm{BB}}$ constitutes a soundness breach of the NIZK.

Thus, we investigate the probability that equation (4) is satisfied despite of the existence of a tuple $T_{\tilde{k}} = (m_{\tilde{k}}, y_{\tilde{k}}, \pi_{\tilde{k}}, v_{\tilde{k}})$ with $\pi_{\tilde{k}} = (\Gamma_{\tilde{k}}, U_{\tilde{k}}, V_{\tilde{k}}, s_{\tilde{k}})$, where $(v_{\tilde{k}}, \Gamma_{\tilde{k}}) \notin R^{\mathrm{cf}}_{B,H_{\tilde{k}}}$, for which by assumption the equations

$$U_{\tilde{k}} = s_{\tilde{k}} * B - c_{\tilde{k}} * v_{\tilde{k}},$$
$$V_{\tilde{k}} = s_{\tilde{k}} * H_{\tilde{k}} - c_{\tilde{k}} * \Gamma_{\tilde{k}}$$

are not simultaneously satisfied, where $H_{\tilde{k}}$ is the unique base associated to $(v_{\tilde{k}}, m_{\tilde{k}})$ and $c_{\tilde{k}}$ is the

[12] And note that at most one point $P \in G$ can be associated to (v', m') as argued in the proof of Theorem 7.3 based on no bad events being triggered so far.

[13] This follows by $\overline{F_{VF1}}$.

challenge associated to this proof string for this proof instance. We therefore have

$$r_{\tilde{k}} * (s_{\tilde{k}} * B - c_{\tilde{k}} * v_{\tilde{k}} - U_{\tilde{k}}) + l_{\tilde{k}} * (s_{\tilde{k}} * H_{\tilde{k}} - c_{\tilde{k}} * \Gamma_{\tilde{k}} - V_{\tilde{k}}) \\ = \sum_{k \in [|L_{i:j}|] \setminus \{\tilde{k}\}} -(r_k * (s_k * B - c_k * v_k - U_k) + l_k * (s_k * H_k - c_k * \Gamma_k - V_k))$$

as an equation over the elliptic curve group $\mathbb{E}$. Towards the argument, define

$$Q := \sum_{k \in [|L_{i:j}|] \setminus \{\tilde{k}\}} -(r_k * (s_k * B - c_k * v_k - U_k) + l_k * (s_k * H_k - c_k * \Gamma_k - V_k))$$

$$Q_1^{(r)} := s_{\tilde{k}} * B; \quad Q_2^{(r)} := c_{\tilde{k}} * v_{\tilde{k}}; \quad Q_3^{(r)} := U_{\tilde{k}};$$

$$Q_1^{(l)} := s_{\tilde{k}} * H_{\tilde{k}}; \quad Q_2^{(l)} := c_{\tilde{k}} * \Gamma_{\tilde{k}}; \quad Q_3^{(l)} := V_{\tilde{k}}$$

which allows us to rewrite the equation as

$$l_{\tilde{k}} * (Q_1^{(l)} - Q_2^{(l)} - Q_3^{(l)}) + r_{\tilde{k}} * (Q_1^{(r)} - Q_2^{(r)} - Q_3^{(r)}) = Q. \tag{5}$$

Similar to the Fiat-Shamir transform, we can consider the verification as the non-interactive version of an interactive proof, where the prover presents a list $L_{i:j}$ of tuples and the verifier samples the coefficients r_k and l_k at random from a large space $\mathcal{C}$, and the probability of a soundness failure is bounded by the probability that equation (5) happens to be satisfied as described above. In the random-oracle model, the honest verifier can be replaced by the random oracle as described in Section 7.1.1, if there is a one-to-one mapping between protocol runs and invocations to the random oracle. We observe that given our assumptions of none other bad event happening, for each list of tuples presented by a potentially malicious prover, the sampling $l_k \leftarrow \mathsf{H}(\texttt{suite\_s} \,||\, 0x4c \,||\, k \,||\, S_T \,||\, 0x00)[..\kappa]$ and $r_k \leftarrow \mathsf{H}(\texttt{suite\_s} \,||\, 0x52 \,||\, k \,||\, S_T \,||\, 0x00)[..\kappa]$ is performed using different inputs to the random oracle, which establishes the mapping. In particular, S_T is the ordered list specifying for each k, $H_k \,||\, \Gamma_k \,||\, U_k \,||\, V_k \,||\, s_k$ which, assuming no collision among the random base points H_k assigned to (v_k, m_k), is the representation for the tuple (m_k, y_k, π_k, v_k) and $y_k = \mathsf{Compute}(\pi_k)$ must hold. Therefore, different lists obtained from $\mathcal{G}_{\mathrm{BB}}$ result in different values for S_T, and by domain separation, independent random coefficients are chosen.

Returning to equation (5) it is easy to see that if either $Q_1^{(l)} - Q_2^{(2)} - Q_3^{(l)} \in \mathbb{G}$ or $Q_1^{(r)} - Q_2^{(r)} - Q_3^{(r)} \in \mathbb{G}$, and recall that by assumption at least one sum does not equal the identity, the equation is fulfilled with probability at most $1/|\mathcal{C}|$ over the random choice of the coefficients.

For the general case, where $Q_1^{(z)} - Q_2^{(z)} - Q_3^{(z)} \neq O$ for at least one $z \in \{l, r\}$, denote $Q_1 := Q_1^{(z)}$ and $P := -(Q_2^{(z)} + Q_3^{(z)})$. We thus have $Q_1 + P \neq O$, where $Q_1 \in \mathbb{G}$ and $P \in \mathbb{E}$ and $P \notin \mathbb{G}$ by assumption. We observe that any $l_{\tilde{k}}$ for which $l_{\tilde{k}} * (Q_1 + P) = Q$, we obtain a solution for $l_{\tilde{k}} * \phi_{\mathrm{cf}}(Q_1 + P) = \phi_{\mathrm{cf}}(Q)$, where the right-hand sides are independent of the random coefficient and the points $Q_1 + P$ and Q are defined before sampling the random coefficient. Thus, as long as $Q_1 + P \notin \ker(\phi_{\mathrm{cf}})$, the probability to satisfy the condition is at most $1/|\mathcal{C}|$. Therefore, the probability of passing the check is at most $1/|\mathcal{C}|$ provided that at least one of $Q_1^{(r)} - Q_2^{(r)} - Q_3^{(r)}$ and $Q_1^{(l)} - Q_2^{(l)} - Q_3^{(l)}$ is not in the kernel of ϕ_{cf}.

The remaining case is simple based on our considerations in Section 7.1.2: a tuple $T_{\tilde{k}}$ fixes the entire instance of a particular proof, i.e., $B, H_{\tilde{k}}, v_{\tilde{k}}, \Gamma_{\tilde{k}}$, and encodes a particular run of the associated Σ-protocol where the challenge is computed correctly based on the random oracle using the Fiat-Shamir transform (otherwise, the entire sequence of tuples is rejected). In view of equation (3), we see that the employed Σ-protocol is sound w.r.t. relation R^{cf}_{B,H_k} even for the relaxed verification

$Q_1^{(r)} - Q_2^{(r)} - Q_3^{(r)} \in \ker(\phi_{\text{cf}}) \wedge Q_1^{(l)} - Q_2^{(l)} - Q_3^{(l)} \in \ker(\phi_{\text{cf}})$. Thus, the probability that the instance and proof run encoded in $T_{\tilde{k}}$ satisfies this check but $(v_{\tilde{k}}, \Gamma_{\tilde{k}}) \notin R^{\text{cf}}_{B,H_k}$ is at most $1/|\mathcal{C}|$. The theorem follows by taking the union bound over all batch verifications instructed by the environment. □

8 Putting Everything Together

We analyzed the range-extension construction in Section 3 without batch verification in a modular way based on any VRF that UC-realizes $\mathcal{F}_{\mathsf{VRF}}^{\ell,\ell_{\mathsf{VRF}}}$. Nevertheless, it is easy to see that batch verification and range extension can be done in a single step in the protocol above. All we have to do is to modify the algorithm $\mathsf{Compute}$ in π^+_{ECVRF} (and in particular, this changes the format of the tuples $T = (m, y, \pi, v)$ only in one place, i.e., $y \in \{0,1\}^{c \cdot \ell_{\mathsf{VRF}}}$, where c is the fixed constant in the range-extension construction. We denote the new protocol by $\tilde{\pi}^+_{\mathsf{ECVRF}}$:

- $\mathsf{Compute}'(\pi)$: The output computation goes as follows for a (proof) string $\pi = \Gamma \,||\, \ldots$.

  Precondition: $\Gamma \in \mathbb{E}$.

  1. Compute $Y \leftarrow \mathsf{H}(\texttt{suite\_s} \,||\, 0x03 \,||\, (\text{cf} * \Gamma) \,||\, 0x00)$.
  2. Output $(\mathsf{H}(\texttt{suite\_s} \,||\, 0x04 \,||\, 1 \,||\, Y \,||\, 0x00), \ldots, \mathsf{H}(\texttt{suite\_s} \,||\, 0x04 \,||\, c \,||\, Y \,||\, 0x00))$.

where we follow the format for domain separation for ECVRF. We obtain the following corollary.

Corollary 8.1. *Under the same assumptions as Theorem 7.4, protocol $\tilde{\pi}^+_{\mathsf{ECVRF}}$ UC-realizes $\mathcal{F}_{\mathsf{VRF}^+}^{\ell,c\cdot\ell_{\mathsf{VRF}}}$, for $\ell = \{0,1\}^*$ and $\ell_{\mathsf{VRF}}(\kappa) = 4\kappa$.*

Proof Sketch. The only difference in the simulation compared to the proof of Theorem 7.4 is that the output of the VRF functionality $y = (y_1, \ldots, y_c)$ w.r.t. (v, m) must additionally be made consistent with the value of the random oracle in the domain-separated positions $(\texttt{suite\_s} \,||\, 0x04 \,||\, i \,||\, Y \,||\, 0x00)$ for $i = 1, \ldots, c$, where Y is obtained by evaluating $\mathsf{H}(\texttt{suite\_s} \,||\, 0x03 \,||\, P \,||\, 0x00)$ and P is derived from a valid proof string $\pi = \Gamma \,||\, \ldots$ as $P = \text{cf} * \Gamma$.

We recall from the proofs of Theorem 7.3 and Theorem 7.4 that as long as the bad events defined in Figure 5 do not occur, that if a point P (from the set of points queried ot the random oracle as above) is associated with a key-message pair (v, m) , then this is a valid association[14] and that the assignment is unique. Also the converse is proven, i.e., at most one of the points P queried to the random oracle can be associated with (v, m) as long as none of the bad events occur. Since the simulation is consistent, the assignment of points P to pairs (v, m) can be done upon the first invocation of the form $\mathsf{H}(\ldots \,||\, P \,||\, \ldots)$.

Finally, correctly predicting the random-oracle output Y derived from point P (that is associated to (v, m)) is a negligible probability event. Therefore, all the pairs (i, Y), $i = 1, \ldots, c$, queried to the RO are to be programmed just at the moment when $Y \leftarrow_{\$} \{0,1\}^{\ell_{\mathsf{VRF}}}$ is defined for the first time in the simulation and associated to the pair (v, m) via point P. Similar to the proof of Theorem 3.1, a consistent simulation is only possible if none of these positions (i, Y) for $i = 1, \ldots, c$ has been programmed before, which is an event that can be bounded by the (negligible) collision probability of bitstrings drawn uniformly at random from $\{0,1\}^{\ell_{\mathsf{VRF}}}$. Therefore, if neither such collisions nor any of the above defined bad events occur we obtain a simulator for which the real and ideal executions are indistinguishable. The claim follows. □

[14] In the sense that valid proof strings can exist that prove the statement $(v, \Gamma) \in L_{R^{\text{cf}}_{B,H}}$, where $\text{cf} * \Gamma = P$ and H is derived from (v, m).

References

[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, *Theory of Cryptography*, pages 1–30, Cham, 2020. Springer International Publishing.

[BDL+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, September 2012.

[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: 25th Conference on Computer and Communications Security*, pages 913–930, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany.

[Can20] Ran Canetti. Universally composable security. Journal of the ACM, Vol. 67, No. 5, 2020.

[CEK+16] Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. Universal composition with responsive environments. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016, Part II*, volume 10032 of *Lecture Notes in Computer Science*, pages 807–840, Hanoi, Vietnam, December 4–8, 2016. Springer, Heidelberg, Germany.

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

[GRPV] Sharon Goldberg, Leonid Reyzin, Dimitrios Papadopoulos, and Jan Vcelak. Verifiable random functions. Internet-Draft, IRTF. `https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vrf-10`.

[Mau15] Ueli Maurer. Zero-knowledge proofs of knowledge for group homomorphisms. *Des. Codes Cryptography*, 77(2-3):663–676, December 2015.

[NMVR95] David Naccache, David M'RaÏhi, Serge Vaudenay, and Dan Raphaeli. Can d.s.a. be improved? — complexity trade-offs with the digital signature standard —. In Alfredo De Santis, editor, *Advances in Cryptology — EUROCRYPT'94*, pages 77–85, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[Rey] Leonid Reyzin. Vrf standardisation mailing archive. `https://mailarchive.ietf.org/arch/msg/cfrg/KJwe92nLEkmJGpBe-OST_ilr_MQ`.