A Composable Security Treatment of ECVRF and Batch Verifications

Christian Badertscher[1(✉)], Peter Gaži[2], Iñigo Querejeta-Azurmendi[3], and Alexander Russell[4,5]

[1] Input Output, Zurich, Switzerland
christian.badertscher@iohk.io
[2] Input Output, Bratislava, Slovakia
peter.gazi@iohk.io
[3] Input Output, London, UK
querejeta.azurmendi@iohk.io
[4] Input Output, New Jersey, USA
alexander.russell@iohk.io
[5] University of Connecticut, Mansfield, USA

Abstract. Verifiable random functions (VRF, Micali *et al.*, FOCS'99) allow a key-pair holder to verifiably evaluate a pseudorandom function under that particular key pair. These primitives enable fair and verifiable pseudorandom lotteries, essential in proof-of-stake blockchains such as Algorand and Cardano, and are being used to secure billions of dollars of capital. As a result, there is an ongoing IRTF effort to standardize VRFs, with a proposed ECVRF based on elliptic-curve cryptography appearing as the most promising candidate.

In this paper, towards understanding the general security of VRFs and in particular the ECVRF construction, we provide an ideal functionality in the Universal Composability (UC) framework (Canetti, FOCS'01) that captures VRF security, and show that ECVRF UC-realizes it.

Additionally, we study batch verification in the context of VRFs. We provide a UC-functionality capturing a VRF with batch-verification capability, and propose modifications to ECVRF that allow for this feature. We again prove that our proposal UC-realizes the desired functionality. Finally, we provide a performance analysis showing that verification can yield a factor-two speedup for batches with 1024 proofs, at the cost of increasing the proof size from 80 to 128 bytes.

1 Introduction

A Verifiable Random Function (VRF, [19]) is a pseudo-random function whose correct evaluation can be verified. It can be seen as a hash function that is keyed by a public-private key pair: the private key is necessary to evaluate the function and produce a proof of a correct evaluation, while the public key can be used to verify such proofs. VRFs were originally considered as tools for mitigation of offline dictionary attacks on hash-based data structures; more recently they have found applications in the design of verifiable lotteries. In particular, VRFs

V. Atluri et al. (Eds.): ESORICS 2022, LNCS 13556, pp. 22–41, 2022.
https://doi.org/10.1007/978-3-031-17143-7_2

are fundamental primitives to several proof-of-stake ledger consensus protocols, such as those underlying the blockchains Algorand [14] and Cardano [12]. They allow for a pseudo-random selection of block leaders in the setting with adaptive corruption, an important security feature of these protocols.

There is an ongoing effort to standardize this primitive via an IRTF draft [15] that describes the desirable properties of VRFs and proposes (as of August '22) two concrete constructions. One of these constructions is based on RSA, while the other one relies on elliptic-curve cryptography (ECC); this latter construction is referred to as ECVRF. A clear advantage of ECVRF over the RSA-based alternative is the considerable improvement in key sizes it provides (for the same security level). Indeed, both Algorand and Cardano employ ECVRF, as do most of the existing implementations listed in the draft.

One of the VRF security properties articulated in the IRTF draft is that of *random-oracle-like unpredictability*. Roughly speaking, it requires that if the VRF input has sufficient entropy (i.e., cannot be predicted), then the output is indistinguishable from uniformly random. As the draft observes, this property is essential for the security of the leader-election mechanisms in PoS blockchains. The property is not formally defined in the draft, though a definition in the form of an ideal functionality in the Universal Composability (UC) framework [10] is given in [12]. The IRTF draft states that this strong notion is "believed" to be satisfied by the ECVRF construction; however, to the best of our knowledge, no formal proof of this claim exists to date. This state of affairs is clearly unsatisfactory: UC security is a desirable notion of security as it guarantees that the proven security provisions (in the sense of realizing an ideal functionality) are retained, by virtue of the composition theorem, when employing the scheme in higher-level applications. This is especially relevant for VRFs as a low-level primitive used in many protocols, including those mentioned above.

Returning to the ECVRF construction, another important benefit it provides is structural: it is essentially a Fiat-Shamir transformed [13] Σ-protocol [11] and therefore—at least in principle—suitable for batch verification. The idea for batch verification first appears in foundational work by Naccache et al. [20] and consists of verifying a batch of linear equations by verifying a random linear combination of these. Bernstein et al. [7] exploited this technique with the state-of-the-art algorithms in multi-scalar multiplication, achieving a factor-two improvement in signature verification using batches of 64 signatures. Such an improvement in verification times is of direct relevance for blockchains, as the routine task of joining the protocol—which requires synchronizing with the current ledger—involves verification of many blocks and their VRF proofs. Indeed, typical synchronization conventions demand verification of the entire existing blockchain. We note in passing that the possibility of batch verifications for Schnorr signatures [24] (derived from another type of Σ-protocol) is a significant competitive advantage over ECDSA, and was one of the reasons for Bitcoin [21] to switch to that type of signature [25]. The possibility of batch verification for ECVRF has already appeared in the IRTF draft mailing list [23]. However, a concrete proposal for the design, along with a formal security notion and a corresponding security proof, has not been given.

Our Contributions. In this work we close both of these gaps.

1. We propose a cleaner formalization of the VRF functionality in the UC framework, building on the original proposal from [12] (later revised in [3] to remove some issues in the original formulation).
2. We show that ECVRF UC-realizes this functionality in the random-oracle model (ROM). The proof of this claim is surprisingly involved, requiring a rather complex simulation. The proof appears in full detail in the full version of this paper [2]. We point out that this is the first comprehensive UC proof for this type of VRF construction and further shows that the simulation can be done in a *responsive* manner [8], a desirable property that simplifies the analysis of higher-level protocols using the VRF functionality (e.g., [3]). In particular, the simulation strategy described in [12] is not applicable (cf. related work below) and [12] does not provide a proof for the revised functionality.
3. We introduce a UC formalization for a VRF providing batch verification via a natural extension of the above VRF functionality.
4. We define a concrete instantiation of batch verification for the ECVRF construction and prove that it UC-realizes the above ideal functionality of a VRF with batch verification. Despite our focus on VRFs, we believe that our formalization would naturally carry over to other widely used Fiat-Shamir transformed Σ-protocols, such as Schnorr signatures or Ed25519.
5. To evaluate the efficiency improvements of the batch-compatible version, we compare the efficiency of the current draft version versus the batch-compatible primitive presented in this work. Roughly speaking, we observe that the batch compatible primitive can achieve a factor-two efficiency gain with batches of size 1024 in exchange for a trade-off with respect to its size, growing from 80 bytes to 128 bytes.
6. We provide an additional efficiency improvement, namely a simple range-extension that can be implemented "on the fly" in ECVRF, which can help higher-level protocols to reduce the number of VRF evaluations and proof verification at the cost of more evaluations of the hash function.

Related Work. The VRF notion was introduced by Micali *et al.* [19]. A stronger notion of VRF with security in the natural setting with malicious key generation was presented as a UC functionality by David *et al.* [12]. A particular instantiation, based on 2HashDH [16], was claimed to satisfy this stronger notion, but the provided simulation argument only holds for a revised version of the functionality which is first described in [3]. Jarecki *et al.* [16] provide a UC functionality of a slightly different notion, which is that of a Verifiable *Oblivious* Pseudo Random Function where two parties need to input some secret information in order to compute the random output.

The first systematic treatment of batch verification for modular exponentiation was presented by Bellare *et al.* [4], and adapted to digital signatures by Camenisch *et al.* [9]. The batch verification technique that we adopt was initially developed by Naccache et al. [20], and used by Bernstein *et al.* [7] and Wuille *et al.* [25]. Exploiting the batching technique in the context of VRFs was informally discussed in the IRTF group and mailing list [15,23].

2 Preliminaries

UC Security. We give a very brief overview of the UC security framework necessary to understand the rest of this work. For details we refer to [10]. In this framework a protocol execution (the so-called "real-world process") is represented by a group of interactive Turing machine instances (ITIs) running a protocol π, forming a protocol session. The environment $\mathcal{Z}$ orchestrates the inputs and receives the outputs of these machines. Additionally, an adversary is part of the execution and can corrupt parties and thereby take control of them (we assume throughout this work the standard UC adaptive corruption model defined in [10]). To capture security guarantees, UC defines a corresponding ideal process which is formulated w.r.t. an ideal functionality $\mathcal{F}$. In the ideal process, the environment $\mathcal{Z}$ interacts with the ideal-world adversary (called simulator) $\mathcal{S}$ and with functionality $\mathcal{F}$ (or more precisely, with protocol machines that simply relay all inputs and outputs to and from $\mathcal{F}$, respectively). A protocol π UC-realizes $\mathcal{F}$ if for any (efficient) adversary there exists an (efficient) simulator $\mathcal{S}$ such that for any (efficient) environment $\mathcal{Z}$ the real and ideal processes are indistinguishable. This means that the real protocol achieves the desired specification $\mathcal{F}$.

VRF Syntax. We denote by κ the security parameter. The domain of the VRF is denoted by $\mathcal{X}$ and its finite range is denoted by $\mathcal{Y}$ and typically represented by $\mathcal{Y} = \{0,1\}^{\ell_{\mathsf{VRF}}(\kappa)}$, where $\ell_{\mathsf{VRF}}(.)$ is a function of the security parameter. For notational simplicity we often drop the explicit dependence on κ.

Definition 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*. (If* $X \notin \mathcal{X}$ *or* $Y \notin \mathcal{Y}$*, we assume that* b *is* 0 *by default.)*

3 UC Security of Verifiable Random Functions

Modeling VRFs as a UC Protocol. Any verifiable random function VRF can be cast as a simple protocol π_{VRF} in the UC framework [10] as follows: Each party U_i in session sid acts as follows: on its first input of the form (KeyGen, sid), run $(sk, vk) \leftarrow \mathsf{VRF.Gen}(1^\kappa)$, output $(\mathsf{VerificationKey}, sid, vk)$ and internally store sk; any further key generation requests are ignored. On input $(\mathsf{EvalProve}, sid, m)$ for an input $m \in \mathcal{X}$ (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

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

The functionality interacts with parties denoted by $\mathcal{P} = \{U_1, \ldots, U_{|\mathcal{P}|}\}$ as well as 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 with $m \in \mathcal{X}$, verify that some $(U_i, v) \in S_{pk}$ is recorded. If such an entry is not stored or $m \notin \mathcal{X}$, 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 $y \xleftarrow{\$} \{0,1\}^{\ell_{\mathsf{VRF}}}$ and set $T[v, m] \hookleftarrow \{y, \{\pi\}\}$.
  3. If $T[v, m] = (y, S) \neq \perp$, set $T[v, m] \hookleftarrow \{y, S \cup \{\pi\}\}$.
  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 $(\mathsf{Eval}, sid, v, m)$, $m \in \mathcal{X}$, from $\mathcal{S}$ (if $m \notin \mathcal{X}$ the request is ignored), do the following:
  - Case 1: $\exists(U_i, v) \in S_{pk}$ where U_i is not corrupted: if $T[v, m] = (y, S)$ for $S \neq \emptyset$, 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 $y \xleftarrow{\$} \{0,1\}^{\ell_{\mathsf{VRF}}}$ 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 [12, 3]].** On input $(\mathsf{PastEvaluations}, sid)$ from $\mathcal{S}$, return S_{eval} to $\mathcal{S}$.

Fig. 1. The VRF functionality.

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 $(\mathsf{Verified}, sid, v', m, y, \pi, b)$.

Ideal Functionality $\mathcal{F}_{\mathsf{VRF}}^{\mathcal{X},\ell_{\mathsf{VRF}}}$. In Fig. 1 we present the functionality $\mathcal{F}_{\mathsf{VRF}}^{\mathcal{X},\ell_{\mathsf{VRF}}}$ that captures the desired properties of a VRF. The functionality provides interfaces for key generation, evaluation and verification, as well as separate adversarial interfaces for malicious key generation, evaluation, and leakage. 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 for adversarially generated keys. Furthermore, no incorrect association can be ever verified and every completed honest evaluation can be later verified correctly.

The functionality is based on [3,12], but contains several 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). Second, 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 [3].

Definition 2 (UC security of a VRF). *A verifiable random function* VRF *with input domain* $\mathcal{X}$ *and range* $\mathcal{Y} = \{0,1\}^{\ell_{\mathsf{VRF}}}$ *is called UC-secure if* π_{VRF} *UC-realizes* $\mathcal{F}_{\mathsf{VRF}}^{\mathcal{X},\ell_{\mathsf{VRF}}}$ *specified in Fig. 1.*

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 (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. 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.

$\texttt{Hash} : \{0,1\}^* \to \{0,1\}^{\ell(\kappa)}$	$\texttt{Expand\_key} : \{0,1\}^{2\kappa} \to \{0,1\}^{2\kappa} \times \{0,1\}^{2\kappa}$
$\texttt{Encode\_to\_curve} : \{0,1\}^* \to \mathbb{G}$	$\texttt{Nonce\_generation} : \{0,1\}^{2\kappa} \times \mathbb{E} \to \mathbb{Z}_q$
$\texttt{Compute\_scalar} : \{0,1\}^{2\kappa} \to S$	$\texttt{Hash\_pts} : \mathbb{E} \times \mathbb{E} \times \mathbb{E} \times \mathbb{E} \times \mathbb{E} \to \{0,1\}^{\kappa}$

Fig. 2. Domain of the helper functions for ECVRF (see [2] for more details). The functions `Hash` and `Encode_to_curve` are modeled as random oracles in the security argument and `Compute_scalar` is an encoding that preserves the min-entropy of its input. The remaining helper functions are implemented based on `Hash` (using domain-separation), where `Expand_key` is an adaptively secure pseudo-random generator, `Nonce_generation` is an adaptively secure pseudo-random function, and `Hash_pts` is a random oracle.

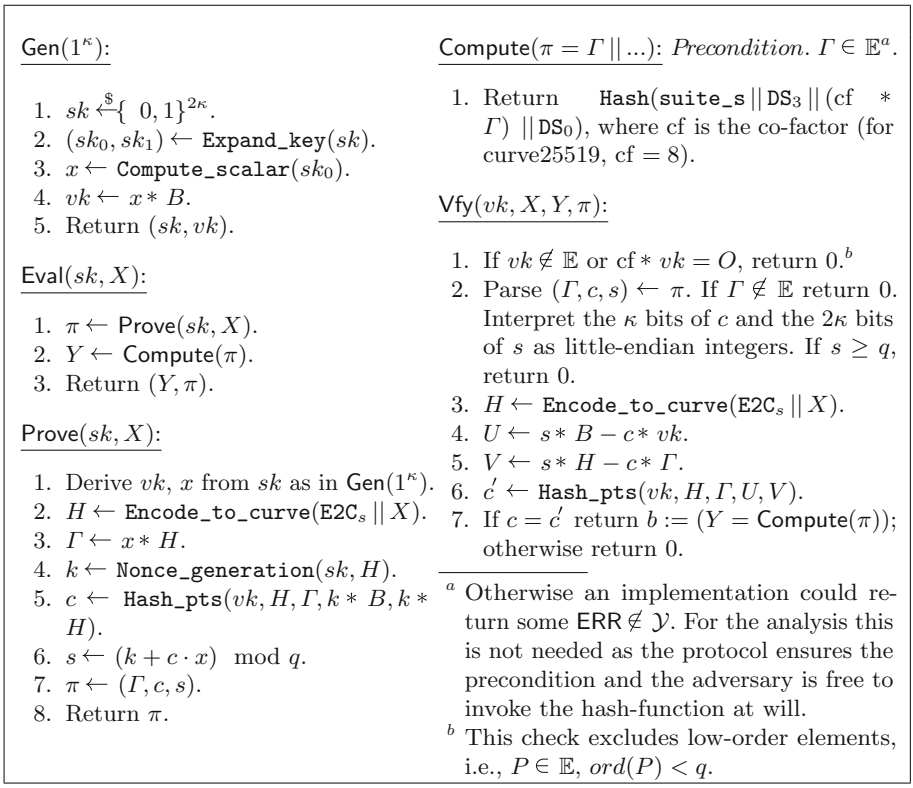

Fig. 3. Description of ECVRF, where B denotes the generator of the subgroup $\mathbb{G}$ of $\mathbb{E}$. Note that the salt value $\texttt{E2C}_s$ leaves room for more general use cases. We consider the case $\texttt{E2C}_s = vk$ in the analysis of the standard and its extensions.

4 The ECVRF Standard

This section recalls the elliptic-curve based schemes described in the IRTF draft [15] and focuses on the cipher suites $\texttt{suite\_s} \in \{0x03, 0x04\}$ for the sake of concreteness.

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 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 $a \xleftarrow{\$} S$ 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; thus, for $A = 0\,||\,1$ we have $A[..1] = 0$ and $A[1..] = 1$.

The standard makes use of helper functions, all of which are defined and introduced in [15]. For sake of simplicity we only state the specification of the security-relevant helper functions. The functions are briefly described in Fig. 2. As domain separators we use values between 0 and 5 in hexadecimal representation. In particular, we use $\texttt{DS}_i \leftarrow 0x0i$ for $i \in [0,5]$. The standard also uses `encode_to_curve_salt` to denote the salt used for the `Encode_to_curve` function, which we denote by $\texttt{E2C}_s$. Note that all EC-ciphersuites define the salt as the prover's public key which is the case we consider and analyze in this work. To give a concrete example, the deployed VRF construction in Cardano is instantiated with $\kappa = 128$ and elliptic curve edwards25519 which has cofactor 8. The prime order q is represented by 32 octets, or more precisely 253 bits, and the hash function is $\texttt{SHA512} : \{0,1\}^* \to \{0,1\}^{512}$. Conveniently, we choose $\ell(\kappa) = 4\kappa$. The function `Hash_pts` defines the associated *challenge space*, thus being the set $\mathcal{C} := \{0,1\}^\kappa$ interpreted as integers. For the function $\texttt{Compute\_scalar}(sk_0)$, 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.

The VRF Algorithms. The formal definition of a VRF in Sect. 3 denotes by Eval 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 we define the functions Prove and Compute to represent the proof generation and the output computation, respectively. The algorithms from the standard are given in Fig. 3.

5 $\mathsf{ECVRF}_{\mathsf{bc}}$: Batch Verification for ECVRF

In the interest of performance, we now study the possibility of batch-verifying the proofs generated by ECVRF. To this end, we introduce 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.

We divide the exposition of the changes in two steps. First, in Sect. 5.1 we present the changes on the protocol (involving the prover and the verifier) to make the scheme *batch-compatible*. Second, in Sect. 5.2 we describe the specific computation performed by the verifier to batch several proof verifications.

Intuition. The operations performed in steps 4 and 5 of Vfy appear as good candidates for batching across several proofs. Namely, instead of sequential scalar multiplications, one could perform a single multiscalar multiplication for all

proofs that are being verified. However, this trick can only be exploited if steps 4 and 5 are equality checks rather than computations. In ECVRF, the verifier has no knowledge of points U and V, and has to compute them first. We hence modify the scheme so that the prover includes points U and V in the transcript and the verifier can simply check for equality.

5.1 Making the Scheme Batch-Compatible

As discussed, in order to allow batch verification, steps 4 and 5 need to be equality checks. This requires a change in step 7 of Prove and changes in steps 2, 4, 5, and 7 of Vfy. Moreover, the challenge computation needs to be moved from step 6 to the position in between steps 3 and 4 (we call it step 3.5). The modifications result in scheme $\mathsf{ECVRF}_{\mathsf{bc}}$, summarized in Fig. 4.

Intuitively, this change has no implications on the security of the scheme, as it is common for (Fiat-Shamir-transformed) Σ-protocols to send the commitment of the randomness (sometimes called the announcement) instead of the challenge.[1] The choice of sending the challenge instead of the two announcements in ECVRF is simply to optimize communication complexity and efficiency.

5.2 Batch-Verification

To see how the changes described above allow for batch verification, first observe how steps 4 and 5 in $\mathsf{ECVRF}_{\mathsf{bc}}$ can be combined into a single check: if they validate, then so does the equation

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

where r, l are scalars chosen by the verifier. The reverse is also true with overwhelming probability if r and l are taken uniformly at random from a set of sufficient size (in particular, we choose the set $\mathcal{C}$ for convenience).

More generally, to verify n different $\mathsf{ECVRF}_{\mathsf{bc}}$ proofs, the verifier needs to check whether the equality relations $U_i = s_i * B - c_i * vk_i$ and $V_i = s_i * H_i - c_i * \Gamma_i$ 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 each $i \in [1, n]$ and, moreover, into a single verification

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

across all proofs, where r_i and l_i are random scalars. The full protocol to implement batch verification based on the above idea appears in Sect. 6.2. By using the state of the art multi-scalar multiplication algorithms, leveraging this trick provides significant running time improvements, as discussed in Sect. 7.

[1] As a matter of fact, ed25519 [7] is also a sigma protocol and encodes the announcement instead of the challenge in the non-interactive variant of this sigma-protocol.

$\underline{\mathsf{Prove}(sk, X)}$ remains unchanged except for step 7, which changes as follows:

7. Let $\pi \leftarrow (\Gamma, (k * B), (k * H), s)$.

$\underline{\mathsf{Compute}(\pi)}$ remains unchanged.

$\underline{\mathsf{Vfy}(vk, X, Y, \pi)}$ changes as follows:

1. Remains unchanged.
2. Parse π as tuple (Γ, U, V, s). If $\{\Gamma, U, V\} \not\subseteq \mathbb{E}$, return 0. Interpret the 2κ bits of s as a little-endian integer. If $s \geq q$, return 0.
3. Remains unchanged.

3.5. $c \leftarrow \mathtt{Hash\_pts}(vk, H, \Gamma, U, V)$.

4. If $U \neq s * B - c * vk$, return 0.
5. If $V \neq s * H - c * \Gamma$, return 0.
6. [Moved to step 3.5]
7. Return $b := (Y = \mathsf{Compute}(\pi))$.

Fig. 4. Description of modifications in $\mathsf{ECVRF}_{\mathsf{bc}}$ compared to ECVRF.

Invalid Batches. Note that if batch verification fails, one would need to break down the batch to determine which proof is invalid. However, in several practical cases (most notably, when validating the state of a blockchain), the verifier is primarily interested in whether the whole batch is valid (so that the respective part of the chain can be adopted); if the batch verification fails this has protocol-level consequences (e.g., disconnecting from the peer providing the invalid batch) that obviate the need for individual identification of the failed verification.

Pseudorandom Coefficients. We describe how the coefficients l_i, r_i can be securely computed in a deterministic manner, a feature that is favorable from a practical perspective. Similarly to the well-known Fiat-Shamir heuristic for Σ-protocols, it is essential that the values cannot be known to the prover when defining the proof string. To this end, we propose to compute the scalars by hashing the contents of the proof itself, the value of H for the corresponding public key, and an index.

Concretely, for a batch proof of proofs $\pi_1, \ldots, \pi_n$, one computes, for $i \in [1, n]$:

1. $\pi'_i \leftarrow H_i \,||\, \pi_i$,
2. $S_T \leftarrow \pi'_1 \,||\, \pi'_2 \,||\, \ldots \,||\, \pi'_n$,
3. $h_i \leftarrow \mathtt{Hash}(\mathtt{suite\_s} \,||\, \mathtt{DS}_4 \,||\, S_T \,||\, i \,||\, \mathtt{DS}_0)$,
4. $l_i \leftarrow h_i[..\kappa]$, and $r_i \leftarrow h_i[\kappa..2 \cdot \kappa]$.

The values 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 can treat the invocation as an evaluation of a random oracle obtained using domain separation on `Hash` (where we follow the usual format).

6 Security Analysis of $\mathsf{ECVRF}_{\mathsf{bc}}$ 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. We refer to the appendix of this work for background on zero-knowledge proofs and homomorphisms which turn out to be a conceptually elegant tool to argue about the security of the scheme.

6.1 Security Analysis of $\mathsf{ECVRF_{bc}}$

Recall from Sect. 3 how any VRF can be understood as a UC protocol. We now establish the security of the $\mathsf{ECVRF_{bc}}$ protocol without the batching step, but with the (minor) modifications introduced in Sect. 5.1. We work in the random-oracle model; that is, we introduce the two general functions H (abstracting the details of `Hash`) and H_{e2c} (abstracting the details of `Encode_to_curve`) which are in the model 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, so that invocations of H and H_{e2c} correspond to invocations of the respective functionalities as explained in Sect. 3. For simplicity and clarity in the UC protocols, we continue to write $\mathsf{H}(x)$ (resp. $\mathsf{H}_{e2c}(x)$) with the understanding that it stands for a call to an ideal object. Note that the remaining helper functions obtain their claimed security properties based on the assumption on H as is established in the proof.

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

Proof Overview. We refer to the full version of this work [2] for the full proof, which is rather involved, and provide here an overview. We must give a simulator such that the real VRF construction (where the above algorithms are executed) is indistinguishable from the ideal world consisting of the ideal VRF functionality plus the simulator (which has to produce an indistinguishable real-world view to the environment). The simulator of this construction can be thought of as performing the following four crucial tasks: it (1) simulates the honest parties' credentials, (2) simulates honest parties' VRF evaluations and proofs (without knowledge of the VRF output), (3) verifies VRF outputs, and (4) ensures that the answers to random-oracle queries are consistent with the outputs of the VRF functionality on the relevant random-oracle evaluations. Observing the definition of the VRF functionality, we see that it enforces several properties that make the simulation task challenging. In particular, unless a key is registered with the functionality, no VRF evaluation is possible. Furthermore, the simulator can only freshly evaluate the VRF on its registered keys or corrupted keys. Finally, the functionality performs an ideal verification in that it stores the mapping $(v, m) \mapsto y$ and answers verification requests specifying (v, m, y') with 1 only if $y = y'$. The difficulty is to argue that the simulator will always be "one step ahead" of the distinguishing environment. That is, if the random oracle produces an output that can correspond to a correct VRF output, then the simulator not only has to detect to which public key this output should be linked, but also that such a public key has in fact already been registered. Furthermore, if the simulator decides that no such public key can currently be associated to an output, this decision cannot be revised and corrected later (even if new public keys are generated). While performing a consistent simulation is tricky, ensuring the other properties requires a careful argumentation and we describe here a

selection of considerations that provide some intuition for the proof and why simulation is possible. On a high level, to correctly simulate verifications, the combination (v, m, y, π) must be mapped to the instance of the NIZK for the relation $R^{\text{cf}}_{B,H}$ (see Appendix A for the notation and definition), which is possible if the association of the (v, m) to the base point H is unique which can be based on the guarantees of the random oracle. Given the soundness of the NIZK the corresponding VRF output is derived based on the point $\phi_{\text{cf}}(\Gamma) := \text{cf} * \Gamma = x * H$, where x is the exponent fulfilling the equation $\phi_{\text{cf}}(v) = x * B$.

Finally, to determine whether the correct value y is specified, the simulator must be consistent with the functionality's output for (v, m). On an intuitive level, this requires the correct association between the protocol values Γ and H with the public key v and message m. First, we note that the probability of guessing a correct output without first computing the base point H can be shown to be negligible. If it has in fact been queried, then thanks to clever programming of the RO, the simulator can detect the relation. For a correct simulation, this assignment must be unique and one-to-one which can be established based on information-theoretic arguments and by the soundness of the NIZK. While the above reasoning is true if the simulator can actually obtain the value y from the functionality, for an honest party with public key v that has never evaluated the VRF on message m this is by definition not possible and we have to prove that only with negligible probability it is possible to find the correct point Γ for such an honest party. This follows by the hardness of the computational Diffie-Hellman problem in the group $\mathbb{G}$. We conclude by noting that an additional complication is to obtain a simulator which is responsive, i.e., which computes replies to queries without additional interaction with the ideal functionality. This aspect is mainly useful for protocol designers that rely on a responsive environment [3,8,12].

6.2 Security Analysis of $\mathsf{ECVRF_{bc}}$ with Batch Verifications

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

The Setting. We want to capture a general setting where the protocol is asked to verify a bunch of claimed VRF proofs originating from any source outside the system, including maliciously generated ones by the adversary. We model this setting using a global bulletin-board functionality $\mathcal{G}_{\text{BB}}$ and describe it in Fig. 5. This abstraction fits not only the public blockchain setting (which can be seen as a bulletin board), but any application that makes use of batch verifications where new proofs appear in the system over time, potentially visible and updatable by anyone including an adversary. Each instance of this functionality maintains a list of values. The list is append-only, but there is no other restriction on what is appended and thus the only guarantee it offers is that if we refer to an interval $[i \dots j]$ in the list associated to session *sid* then, once defined, the returned list of values is always the same. The functionality is a global setup [1] for full generality of the statement. In particular, once proven for this setting, simpler

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 $(\mathsf{Retrieved}, sid, i, j, L[i] \,||\, \ldots \,||\, L[j])$ to the caller.

Fig. 5. The global bulletin board.

settings (such as defining a protocol interface taking a batch of proofs directly from a caller) follow in a straightforward manner.

The Ideal World. In the ideal world, we introduce a new simple command to the VRF functionality described in Fig. 6. 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.

The UC Protocol. Recall from Sect. 3 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} but additionally implements the following procedure outlined in Sect. 5.2. To simplify notation, we continue to write H and H_{e2c} for general hash-function invocations and understand that this corresponds to evaluating the random oracles $\mathcal{F}_{\text{RO}}^{\mathcal{Y}}$ and $\mathcal{F}_{\text{RO}}^{\mathbb{G}}$, respectively.

- On input $(\mathsf{BatchVerify}, sid, i, j)$, send $(\text{RETRIEVE}, sid, i, j)$ to $\mathcal{G}_{\text{BB}}$ 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 $T_k = (m_k, y_k, \pi_k, v_k)$. If the tuple has wrong format, return $(\mathsf{BatchVerified}, sid, i, j, 0)$.
  2. For each T_k perform first the steps 1. to 3. and then step 3.5 of $\mathsf{ECVRF.Vfy}$, that is:

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

Same parameters and initialization as in Figure 1. Additionally, the functionality registers to the instance of $\mathcal{G}_{\mathrm{BB}}$ with the same session identifier *sid*.

- Key generation, malicious key generation, VRF evaluation and proof, malicious VRF evaluation, verification, and adversarial leakage are 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 $\mathcal{G}_{\mathrm{BB}}$ 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 $k = 1 \dots |L_{i:j}|$.
  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 $(\mathsf{BatchVerified}, sid, i, j, 0)$.

Fig. 6. The VRF functionality with Batch Verifications.

- 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 $H_k \leftarrow \mathsf{H}_{e2c}(v_k, m_k)$.
- Compute $c_k \leftarrow \mathsf{H}(\mathtt{suite\_s} \,||\, \mathtt{DS}_2 \,||\, H_k \,||\, \Gamma_k \,||\, U_k \,||\, V_k \,||\, \mathtt{DS}_0)[..\kappa]$.

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 $S_T \leftarrow \pi'_1 \,||\, \dots \,||\, \pi'_{|L_{i:j}|}$.
   - $\forall k \in [|L_{i:j}|] : h_k \leftarrow \mathsf{H}(\mathtt{suite\_s} \,||\, \mathtt{DS}_4 \,||\, S_T \,||\, k \,||\, \mathtt{DS}_0)$.
   - $\forall k \in [|L_{i:j}|] : l_k \leftarrow h_k[..\kappa]$.
   - $\forall k \in [|L_{i:j}|] : r_k \leftarrow h_k[\kappa..2 \cdot \kappa]$.
   - Evaluate

$$b_1 \leftarrow \Bigg(O = \sum_{k \in [|L_{i:j}|]} \Big(r_k * (s_k * B - c_k * v_k - U_k) + l_k * (s_k * H_k - c_k * \Gamma_k - V_k)\Big)\Bigg). \quad (1)$$

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.

Theorem 2. *Under the same assumptions as Theorem 1, the protocol* π^{+}_{ECVRF} *UC-realizes* $\mathcal{F}_{\mathsf{VRF}^+}^{\mathcal{X},\ell_{\mathsf{VRF}}}$ *(where* $\mathcal{G}_{\mathrm{BB}}$ *is a global setup), for* $\mathcal{X} = \{0,1\}^*$ *and* $\ell_{\mathsf{VRF}}(\kappa) = 4\kappa$.

Proof (Sketch). The proof needs to verify two things: first, similar to the reasoning in the Fiat-Shamir transform outlined in Appendix A, it must be the case that invocations of $\mathsf{H}(\ldots||S_T\,||\,k\,||\,\ldots)$ are in one-to-one correspondence with imaginary protocol runs, where a prover first presents S_T and an honest verifier picks the coefficients r_i and l_i uniformly at random. Second, we have to argue that no invalid statement can verify as part of the batch. Let $T_{\tilde{k}}$ be such an invalid tuple. Based on considerations discussed in Appendix A, 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). We see that the employed Σ-protocol is sound w.r.t. relation R^{cf}_{B,H_k} even for the relaxed verification $s_{\tilde{k}} * B - c_{\tilde{k}} * v_{\tilde{k}} - U_{\tilde{k}} \in \ker(\phi_{\mathrm{cf}}) \wedge s_{\tilde{k}} * H_{\tilde{k}} - c_{\tilde{k}} * \Gamma_{\tilde{k}} - V_{\tilde{k}} \in \ker(\phi_{\mathrm{cf}})$. Thus, the probability that the instance and proof run encoded in $T_{\tilde{k}}$ satisfies this check but $(v_{\tilde{k}}, \Gamma_{\tilde{k}}) \notin R^{\mathrm{cf}}_{B,H_k}$ is at most $1/|\mathcal{C}|$. Finally, if $s_{\tilde{k}} * B - c_{\tilde{k}} * v_{\tilde{k}} - U_{\tilde{k}} \in P + \ker(\phi_{\mathrm{cf}})$ for some $P \in \mathbb{G}$ (cf. Appendix A for a brief overview of the concepts here), it is straightforward to see that Eq. (1) holds only with probability at most $1/|\mathcal{C}|$ as we basically compute a random $r_{\tilde{k}}$-multiple of P (the other case for coefficient $l_{\tilde{k}}$ is symmetric). The theorem follows by taking the union bound over all batch verifications instructed by the environment. □

On-the-Fly Range Extension. We conclude this section by showcasing a simple range extension of the VRF which, in certain implementations, can significantly reduce the number of VRF evaluations at the cost of a hash function evaluation. All we have to do is to modify the algorithm $\mathsf{Compute}$ in π^{+}_{ECVRF} which 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 with the new output computation $\mathsf{Compute}'$ below by $\tilde{\pi}^{+}_{\mathsf{ECVRF}}$:

– $\mathsf{Compute}'(\pi)$, where string $\pi = \Gamma\,||\,...$ with $\Gamma \in \mathbb{E}$:
  1. Compute $Y \leftarrow \mathsf{H}(\mathtt{suite\_s}\,||\,\mathsf{DS}_3\,||\,(\mathrm{cf} * \Gamma)\,||\,\mathsf{DS}_0)$.
  2. Output
     $(\mathsf{H}(\mathtt{suite\_s}\,||\,\mathsf{DS}_5\,||\,1\,||\,Y\,||\,\mathsf{DS}_0), \ldots, \mathsf{H}(\mathtt{suite\_s}\,||\,\mathsf{DS}_5\,||\,c\,||\,Y\,||\,\mathsf{DS}_0))$.

Corollary 1. *Under the same assumptions as Theorem 2, protocol* $\tilde{\pi}^{+}_{\mathsf{ECVRF}}$ *UC-realizes* $\mathcal{F}^{\mathcal{X},c\cdot\ell_{\mathsf{VRF}}}_{\mathsf{VRF}^{+}}$*, for* $\mathcal{X} = \{0,1\}^{*}$ *and* $\ell_{\mathsf{VRF}}(\kappa) = 4\kappa$.

Proof (Sketch). The proof follows along the lines of the previous proofs. The only additional concern is the possibility of collisions among the values obtained for Y in the above construction, because we require that each fresh invocation of the output tuple computed in the second step corresponds to new evaluation points of the random oracle H. This bad event can be bounded by the standard collision probability of bitstrings drawn uniformly at random from $\{0,1\}^{\ell_{\mathsf{VRF}}}$. □

7 Performance Evaluation

In this section we evaluate the performance of the ECVRF-EDWARDS25519-SHA512-TAI ciphersuite as defined in the standard [15] against the batch-compatible variant proposed in this paper. Essentially, these are ECVRF and

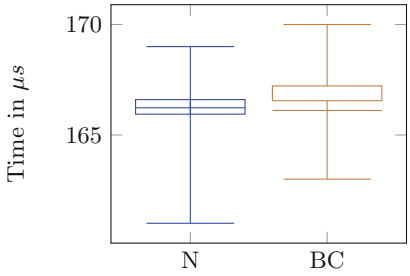

Fig. 7. ECVRF proof generation of the Batch Compatible (BC) version, and the Normal (N) one.

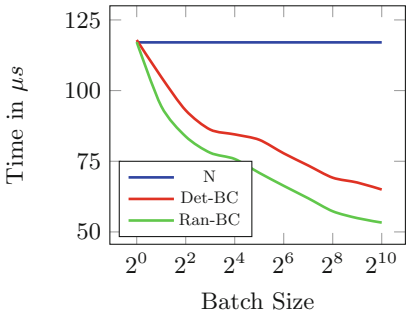

Fig. 8. ECVRF verification, comparing normal version (N), deterministic batch verification (Det-BC) and non-deterministic batch verification (Ran-BC).

$\mathsf{ECVRF}_{\mathsf{bc}}$, respectively, over the curve `edwards25519` with SHA512 as a hashing algorithm. We implement a Rust prototype of version 10 of the draft which we provide open source [22]. We use the curve25519-dalek [17] rust implementation for the curve arithmetic operations, which implements multiscalar multiplication with Strauss' [5] and Pippenger's [6] algorithms, and optimize the choice depending on the size of the batch. We ran all experiments in MacOS on a commodity laptop using a single core of an Intel i7 processor running at 2,7 GHz. For the batch-compatible version we implement both a deterministic verification (using the hashing techniques as described in Sect. 5) as well as a random verification where the scalars r_i, l_i are sampled uniformly at random from $\mathbb{Z}_{2^{128}}$. We benchmark the proving and verification times for each, using batches of size 2^l for $l \in \{1, \ldots, 10\}$. In the standard version, the size of a VRF proof consists of a (32-byte) elliptic curve point, a 16-byte scalar, and a 32-byte scalar. In the batch compatible version, rather than sending the challenge we send the two announcements, which results in three elliptic curve points and a 32-byte scalar. Therefore the modifications increase proof size from 80 to 128 bytes.

This results in a considerable improvement in verification time. Figure 7 shows that proving time is unaffected, and there is no difference between the normal ECVRF and $\mathsf{ECVRF}_{\mathsf{bc}}$ (as expected). In Fig. 8 we show the verification time per proof for different sized batches. We interpret the times of batch verification as a ratio with respect to ECVRF. Using deterministic batching, the verification time per proof is reduced to 0.71 with batches of 64 and to 0.56 with batches of 1024 signatures. With random coefficients, batching times get a bit better given that we no longer need to compute hashes for scalars l_i and r_i. The verification time per proof can be reduced to 0.6 with batches of 64 signatures, and up to 0.47 with batches of 1024.

A Brief Overview of Concepts Used in the Security Argument

We provide here a sketch of fundamental concepts used in the security argument. The extended version contains a detailed exposition [2].

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 [18]. 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 $L_{R_f} := \{z \mid \exists x : (z, x) \in R_f\}$. 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 $k \xleftarrow{\$} \mathbb{H}$ 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 1. ([18]). *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. $\forall z \in L_{R_f}, f(u) = z^\ell$.

The *Fiat-Shamir* Transform turns (in the random-oracle model) any Σ-Protocol into a secure non-interactive zero-knowledge protocol of knowledge. Intuitively, the assumed random oracle is like an honest verifier computing a challenge and thus preserves the above security properties. We refer to [2] for details.

Instantiation for $\mathsf{ECVRF}_{\mathsf{bc}}$. We recall that in $\mathsf{ECVRF}_{\mathsf{bc}}$ 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 $f_{B_1,B_2} : \mathbb{Z}_q \to \mathbb{T}; \quad x \mapsto (x * B_1, x * B_2)$. Since $\mathbb{G}$ is of prime order q, we can satisfy the conditions of Lemma 1 by letting $u = 0$ and $\ell = q$, and defining the challenge space to be a large subset $\mathcal{C} \subseteq [0, \ldots, q-1]$.

We therefore conclude that the embedded non-interactive zero-knowledge proof of knowledge in $\mathsf{ECVRF}_{\mathsf{bc}}$ has (in the random-oracle model) simulatable executions, and with only negligible probability can a valid proof for a wrong statement be generated.

On Domain Checks and the Canonical Epimorphism. Special care has to be taken in the analysis as $\mathsf{ECVRF}_{\mathsf{bc}}$ omits detailed domain checks which in general can impact security in that Lemma 1 cannot be applied directly (we have $\mathbb{G}$ a subgroup of $\mathbb{E}$ and the protocol could be run on values $z_i \in \mathbb{E} \setminus \mathbb{G}$ by a dishonest party as the verifier does not perform a domain check for $z_i \in \mathbb{G}$ but only for $\mathbb{E}$). We leave the general treatment of this to the full version of this work, and describe here a special case based on the *canonical epimorphism*: For $\mathsf{ECVRF}_{\mathsf{bc}}$, we can 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 $\mathbb{G}$ (recall that B and H are generators of $\mathbb{G}$). Stated differently, the above equality is satisfied when $(s*B-V-c*z_1) \in \ker(\phi_{\mathrm{cf}})$ and $(s*H-V-c*z_2) \in \ker(\phi_{\mathrm{cf}})$. As we show in the full version [2], the guarantees of Lemma 1 apply to this projected run of the protocol, in particular, we obtain the soundness guarantee for the relation

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

guaranteed by the above Σ-protocol (where technically speaking, 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 stricter equality checks $(s*B-V-c*z_1) = O$ and $(s * H - V - c * z_2) = O$).

References

1. Badertscher, C., Canetti, R., Hesse, J., Tackmann, B., Zikas, V.: Universal composition with global subroutines: capturing global setup within plain UC. In: Pass, R., Pietrzak, K. (eds.) TCC 2020. LNCS, vol. 12552, pp. 1–30. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64381-2_1
2. Badertscher, C., Gaži, P., Querejeta-Azurmendi, I., Russell, A.: On UC-secure range extension and batch verification for ecvrf. Cryptology ePrint Archive, Report 2022/1045 (2022). https://eprint.iacr.org/2022/1045

3. Badertscher, C., Gazi, P., Kiayias, A., Russell, A., Zikas, V.: Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018: 25th Conference on Computer and Communications Security, pp. 913–930, Toronto, ON, Canada, October 15–19, 2018. ACM Press
4. Bellare, M., Garay, J.A., Rabin, T.: Fast batch verification for modular exponentiation and digital signatures. Cryptology ePrint Archive, Report 1998/007 (1998). http://eprint.iacr.org/1998/007
5. Bellman, R., Straus, E.G.: 5125. The American Mathematical Monthly, 71(7), 806–808 (1964)
6. Bernstein, D.J., Doumen, J., Lange, T., Oosterwijk, J.-J.: Faster batch forgery identification. In: Galbraith, S., Nandi, M. (eds.) INDOCRYPT 2012. LNCS, vol. 7668, pp. 454–473. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34931-7_26
7. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.-Y.: High-speed high-security signatures. J. Cryptogr. Eng. **2**(2), 77–89 (2012)
8. Camenisch, J., Enderlein, R.R., Krenn, S., Küsters, R., Rausch, D.: Universal composition with responsive environments. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10032, pp. 807–840. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53890-6_27
9. Camenisch, J., Hohenberger, S., Østergaard Pedersen, M.: Batch verification of short signatures. J. Cryptol. **25**(4), 723–747 (2012)
10. Canetti, R.: Universally composable security. J. ACM **67**(5) (2020)
11. Cramer, R., Damgård, I., Schoenmakers, B.: Proofs of partial knowledge and simplified design of witness hiding protocols. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 174–187. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-48658-5_19
12. David, B., Gaži, P., Kiayias, A., Russell, A.: Ouroboros praos: an adaptively-secure, semi-synchronous proof-of-stake blockchain. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10821, pp. 66–98. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78375-8_3
13. Fiat, A., Shamir, A.: How to prove yourself: practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 186–194. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-47721-7_12
14. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. Cryptology ePrint Archive, Report 2017/454 (2017). http://eprint.iacr.org/2017/454
15. Goldberg, S., Reyzin, L., Papadopoulos, D., Vcelak, J.: Verifiable random functions (vrfs). Internet-Draft, IRTF (2022). https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vrf-14
16. Jarecki, S., Kiayias, A., Krawczyk, H.: Round-optimal password-protected secret sharing and T-PAKE in the password-only model. Cryptology ePrint Archive, Report 2014/650 (2014). http://eprint.iacr.org/2014/650
17. Lovecruft, I., de Valence, H.: curve25519-dalek (2022). https://github.com/dalek-cryptography/curve25519-dalek
18. Maurer, U.: Zero-knowledge proofs of knowledge for group homomorphisms. Des. Codes Cryptography **77**(2-3), 663–676 (2015)
19. Micali, S., Rabin, M.O., Vadhan, S.P.: Verifiable random functions. In: 40th Annual Symposium on Foundations of Computer Science, pages 120–130, New York, NY, USA, 17–19 October, 1999. IEEE Computer Society Press (1999)

20. Naccache, D., M'Rahi, D., Vaudenay, S., Raphaeli, D.: Can D.S.A. be improved? — Complexity trade-offs with the digital signature standard —. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 77–85. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0053426
21. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system, December 2008. https://bitcoin.org/bitcoin.pdf
22. Querejeta-Azurmendi, I.: Verifiable random function (2022). https://github.com/input-output-hk/vrf
23. Reyzin, L.: Vrf standardisation mailing archive (2021). https://mailarchive.ietf.org/arch/msg/cfrg/KJwe92nLEkmJGpBe-OST_ilr⟨_MQ
24. Schnorr, C.P.: Efficient signature generation by smart cards. J. Cryptol. **4**(3), 161–174 (1991). https://doi.org/10.1007/BF00196725
25. Wuille, P., Nick, J., Ruffing, T.: Schnorr signatures for secp256k1, January 2020. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki