BandChain Verifiable Randomness

Band Protocol Team

Abstract

This short whitepaper introduces the BandChain Verifiable Randomness, our solution for verifiable (pseudo-)random values based on the BandChain blockchain. Our protocol makes use of a *verifiable random function (VRF)* to cryptographically secure that the produced results have not been tampered with (e.g., have not been chosen in a non-uniform manner). Below, we present in detail how our protocol operates, as well as the guaranteed security it achieves.

1 Introduction

Modern decentralized applications (dApps) often rely, for security reasons, on "good" randomness, generated freshly and independently of the application's state. For example, consider an online lottery in which participants place their bids, a random result comes up, and winnings are awarded according to the bid placement. Similarly, consider a leadership election (often used in committee-based blockchain platforms) that proceeds in rounds to randomly elect a leader among a set of participants. For such applications, it is crucial to guarantee that the randomness is sampled uniformly and independently of the application's state, therefore making it hard to predict as well as ensuring that there are no malicious actors who can affect the outcome of such lottery winnings and leadership elections.

In this whitepaper, we present our protocol for the BandChain Verifiable Randomness. Our solution for verifiable pseudo-randomness is based on the distributed BandChain oracle network. BandChain is a high-performance public blockchain that provides APIs for data and services that are stored "off-chain," e.g., on the traditional web or from third-party providers. It supports generic data requests from other public blockchains and performs on-chain aggregation via the BandChain oracle scripts. Oracle results are stored on the BandChain and are returned to the calling dApp on the main blockchain (e.g., Ethereum), along with a proof of authenticity via customized one-way bridges (or, via an Inter-Blockchain Communication protocol). BandChain's versatility, low-latency, and trustless verification have already found plenty of use-cases from providing reliable and accurate price feeds.

To better illustrate how the BandChain Verifiable Randomness works, let us consider an example, the example of a randomized NFT minting platform. The overall flow of the process is shown in Figure 1, without looking into the internal workings of the BandChain yet. We describe it in detail below.

1. First, a user that is looking to mint an NFT through the dApp platform sends a mint transaction to its smart contract, along with a certain amount of tokens (e.g., ETH if the main-chain is Ethereum) as payment.

2. After the dApp contract receives and processes the transaction from the user, it calls the Band VRF contract on the main-chain, requesting a random value and using a *seed* specified in the user's transaction.

3. When the VRF contract function is called, it emits a RandomDataRequested event.

4. This event is detected by either the Band Foundation or any incentivized actor (e.g., bounty hunter) who sends the VRF data request transaction to the BandChain.

5. On the BandChain network, a series of interactions take place and, as a result of this, an already verified VRF output is produced, together with a proof of storage on the BandChain's ledger. The same party can then relay these back to the Band VRF contract on the main-chain.

6. The Band VRF contract verifies the proof of inclusion on the BandChain. If the proof passes verification, the contract forwards the VRF's random value output to the calling NFT dApp.

7. The NFT dApp uses the random value to generate the NFT attributes, and the final token is returned to the user.

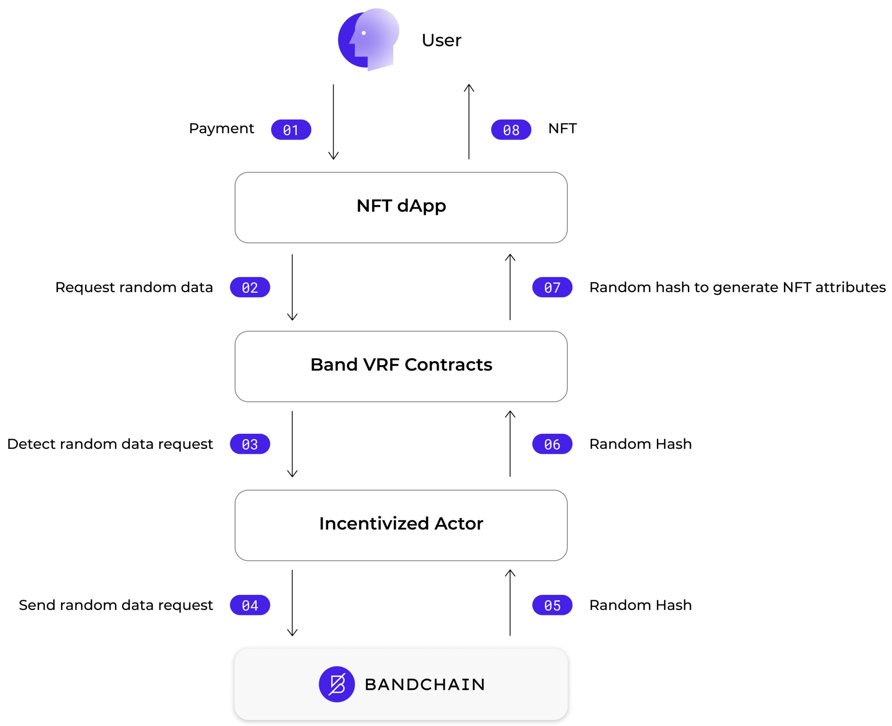

Figure 1: Sample flow of parties' interaction.

Subsequent sections of the whitepaper are structured as follows. In Section 2, we describe the BandChain network and establish the relevant terminologies. In Section 3, we present the definition and security properties of verifiable random functions, as well as the pseudo-code of the instantiation we have adopted. Section 4 includes a detailed explanation of our protocol for BandChain Verifiable Randomness. In addition to emphasizing how the different actors and components interact with each other, Section 5 highlights the subtle security challenges that we had to address and explain the rationale behind our design choices. Finally, in Section 6 we discuss the best practices for developing applications that will use the BandChain Verifiable Randomness, and some future plans for expanding the capabilities of our solution.

2 Background on BandChain

BandChain is built on top of the Cosmos SDK, and it relies on delegated Proof-of-Stake. For consensus, it uses Tendermint's Byzantine Fault Tolerance. The main BandChain participants are *validators* and *data providers*. The former are in charge of extending the chain by proposing and validating new blocks. Each of the validators are bound to the staked tokens, either from their own holdings or delegated to them by other parties (known as *delegators*, they hold tokens but instead of actively participating in the network, they delegate their stake to validators of their choice). On top of this functionality, BandChain validators are in charge of retrieving requested data from the data providers, which we will explain in more detail below. Data providers are parties that aim to collect revenue, by monetizing their services through making external, "off-chain" data available to the BandChain (e.g., acting as sources of randomness for our Verifiable Randomness application).

On the BandChain, *data sources* are APIs stored on the chain and essentially operate like templates specifying how off-chain data should be retrieved. Each data source, contains an owner identity, a fee, an address for receiving the fee and the executable code that is used to run to retrieve the data. For example, for retrieving a cryptocurrency price from an observatory (like CoinGecko), the data source would include the appropriate URL, query information about the specific coin, and a script to retrieve it (e.g., in cURL). BandChain also includes *oracle scripts* which operate like smart contracts. These oracle scripts are in charge of two main operations: (i) identify the appropriate data sources for the given requests and specify the set of validators that should access them, and (ii) aggregate the validators' responses and produce the final result. While (i) typically follows the BandChain randomized validator selection process, the aggregation process (ii) is fully customizable by the creator of the oracle script that meets the requirements of the application. We defer interested readers to [whi] for more details about the internal workings of the BandChain.

3 Verifiable Random Function (VRF)

A commonly used technique for applications that require ”good” random values to be produced rely on cryptography to produce values that are *pseudorandom*, i.e., impossible to distinguish from uniformly randomized values. The classic cryptographic tool used for this is called a Pseudo-Random Function (PRF) [Gol00]. Given a key k that is generated in private and kept secret, such function maps a *seed* to an output value $y = PRF(k, seed)$. The crucial property is that *someone that does not have access to* k, cannot distinguish in polynomial time the output y from a value that is sampled uniformly at random from the range of the PRF.

This is extremely convenient for applications that need to acquire many “random-looking” values y_i. Instead of sampling each one of them at random—a process that is known to be very error-prone and requires easy access to good-quality entropy (e.g., user-generated)—this random sampling takes place only once when choosing k. All subsequent y_i are indistinguishable from randomly chosen ones. The problem with PRFs is that there is no way for the party that generated k to “convince” others that y_i is computed honestly with the PRF for the predefined k which is crucial in certain scenarios to ensure y_i is not chosen *a posteriori*, i.e., with knowledge of the *seed*. This would only be possible by revealing k ahead of time, but this makes the value y_i predictable by everyone. For example, if in the context of a blockchain application the random value is used to determine the next block header (or the next “leader” that will choose it), and revealing k ahead of time allows users to choose *seed* in a way that will give them a favorable outcome.

Verifiable Random Functions (VRF) was proposed [MRV99] as a *public-key analog* of PRFs in order to address the above issue: allowing users to convince third parties that a given value was computed using a key chosen ahead of time and independently of the seed. At a high level, a VRF works with two keys: (i) a secret key *sk* that is used to compute a pseudo-random value y given a *seed* (similar to PRF), and (ii) a corresponding public key *pk* that can be used to verify y that y is computed using the VRF with the secret key that corresponds to this *pk* and the given *seed*. It is crucial that y is indistinguishable from a random value and “impossible” to compute, when given only *pk* (and without access to *sk*). Equally important, a user who attempts to cheat the system should not be able to convince others about an incorrect y for a given *seed*.

More formally, a VRF consists of the following algorithms. For illustration purposes, we explain how they are used in the context of an example with Alice, who wants to compute pseudo random values, and Bob who does not entirely trust Alice and wants to “check” that the values are computed honestly.

KeyGen takes a security parameter λ (which specifies the “length” of the keys). Alice runs this algorithm to get a key-pair (sk, pk). The first is her secret key and the second is her public key which she publishes.

Eval takes as input a secret key *sk* and a *seed*. Alice runs this to compute a pseudorandom value y corresponding to this seed, as well as a *proof* π for its validity.

Verify takes as input: a public key *pk*, a *seed*, a value y, and an accompanying proof-of-validity π. Based on this, the output will either be accept or reject. Bob runs it, and in the case that it accepts, he is convinced that y was indeed computed by running Eval for this *seed* and the secret key that corresponds to *pk*. Note that Bob has never seen this secret key (Alice has kept it private), but since he received Alice's *pk* before the *seed* was known, he is convinced that it was chosen independently of the *seed*.

3.1 VRF security properties

Above we described the API of a VRF and we abstractly described what security guarantees it offers. The three security properties of a VRF are **Unpredictability (UNP1)**, **Uniqueness (UNI2)**, and **Collision-Resistance (CR3)**. Here, we discuss these security properties in more detail; formal definitions can be found in [PWH+17, DGKR18].

- **Unpredictability (UNP1).** This ensures that the values y returned from Eval are distributed in a way that is, for all practical purposes, uniformly random. It is a fundamental property of a VRF as it basically says that the VRF behaves like a random oracle. In practice, this implies that Bob (not knowing the secret key) has no way to predict y that is better than "randomly guessing" even when knowing *seed*. So, if the input seeds are chosen with sufficiently high entropy, it is practically impossible to predict the output.

- **Uniqueness (UNI2).** This ensures that, after Alice publishes her pk, for every *seed* she can only produce a proof that will convince Bob for the correct VRF value $y = \mathsf{Eval}(sk, seed)$. In other words, for a given $(pk, seed)$ it is extremely hard to find two different VRF values that both pass the verification. This property is crucial to protect against a cheating actor, Alice, that tries to claim a specific output, different from the correct one, for her own purposes.

- **Collision-Resistance (CR3).** This ensures that it is computationally hard to find two different inputs $seed, seed'$, with the same sk to obtain the identical output value —much like the classic property of the cryptographic hash functions. The important difference is that for VRFs this property holds true even when against an adversary that knows sk! Note that this offers a different type of protection than UNI2. For example, it protects against a party that tries to claim a y that was computed from one input *seed* as if it was computed from a different $seed'$.

Cheating when generating keys? Another very important security aspect is whether a party can cheat by exploiting the methodology in which the keys are generated. For example, is it possible for Alice to publish a "bad" pk that allows her to produce convincing proofs for falsified values? There are two VRF variations with respect to this property: those that achieve the above properties only when a trusted party generates the keys (*trusted-security VRF*) and those that achieve it even when parties are entrusted to generate their own keys (*full-security VRF*). Clearly, the latter is a more desirable choice.

3.2 Our VRF instantiation

For our BandChain VRF application, we chose the VRF of [PWH+17] which has already been adopted in various other protocols, e.g., [GHM+17, BCC+21]. The construction is based on a well-studied cryptographic hardness assumption over prime-order elliptic curve groups. For our instantiation, we chose the widely-used Ed25519 curve [BDL+11] that achieves very good performance and has a transparent choice of parameters, as well as Elligator for our hash-to-curve instantiation. Our implementation is fully compliant with the latest VRF draft standard [GRPV21]. Moreover, we implemented all the necessary techniques to achieve *full security*, as explained above. For completeness, we include the pseudo-code description of our VRF in Figure 2.

4 Integrating the VRF with BandChain

Having defined how a VRF operates, we now describe how we use it to implement our Band Verifiable Randomness application. Recall that BandChain operates like a side-chain working in parallel with a main-chain that supports smart contracts (e.g., Ethereum).

4.1 Protocol Flow Description

At a high level, our protocol works as follows. First, the VRFProvider contract and the Bridge contract are deployed on the main-chain. The VRFProvider contract is in charge of receiving randomness requests from dApps and contains code that pre-processes the request in order to be ready for submission to the Band side-chain. It also works as the receiving end of the request's result.

Public parameters: G is a cyclic group of prime order q and generator g. Let H_1 be a hash function mapping arbitrary-length bitstrings onto $G - \{1\}_G$, H_2 a function that takes the bitstring representation of an element of G and shortens it to length 2λ (λ is the security parameter), and H_3 a hash function mapping arbitrary-length inputs to λ-bit integers.

```
KeyGen  Return secret key $sk \in \{1, \ldots, q-1\}$ chosen uniformly at random, and public key $pk = g^{sk}$.
Eval  On input seed $a$, compute output $y$ and proof $\pi$, as follows:
    1. $h \leftarrow H_1(a)$ and $\gamma \leftarrow h^{sk}$
    2. Choose random nonce $k \in \{0, \ldots, q-1\}$
    3. $c \leftarrow H_3(g, h, g^{sk}, h^{sk}, g^k, h^k)$.
    4. $s \leftarrow (k - c \cdot sk) \bmod q$
    5. Compute proof $\pi \leftarrow (\gamma, c, s)$, output value $y \leftarrow H_2(\gamma)$ and return $(y, \pi)$
Verify  Check proof $\pi = (\gamma, c, s)$ corresponds to input $s$ and output $y$, as follows:
    1. $u \leftarrow (p)^c \cdot g^s$ and $h \leftarrow H_1(a)$
    2. Check that $\gamma \in G$
    3. $v \leftarrow (\gamma)^c \cdot h^s$.
    4. Output accept if $c = H_3(g, h, pk, \gamma, u, v)$, else output reject
```

]

Figure 2: Pseudo-code for the VRF of [PWH+17].

The Bridge contract, as the name denotes, works as the connecting "bridge" between the two chains in order to validate the latest state of the side-chain and verify that the received results for VRF requests are indeed the ones computed and stored on the BandChain.

A third-party dApp that wishes to request a random value submits its request to the VRF-Provider contract, which then prepares the actual VRF input by expanding it into a VRF seed. This is picked up by incentivized actors and/or the Band foundation and is submitted as a VRF request to the BandChain. In particular, a *VRF Oracle Script* collects this request, maps it to the set of *VRF Data Sources* that is available to the chain, as well as a number of *BandChain Validators*. The *VRF Oracle Script* then randomly assigns it to a *VRF Data Source*, which in turn passes the request to a corresponding *VRF Provider API*. The assigned *VRF Provider API* evaluates the VRF based on the prescribed input using its VRF secret key, and broadcasts the result to the Band network. All chosen validators run the VRF verification algorithm using this provider's public key and, if the verification is successful, the result will be transmitted to the *VRF Oracle Script*. Finally, after collecting the necessary number of results from the validators, the oracle script accepts the majority as the final result which then becomes part of the BandChain state, which gets included in the next block's computation.

The final result is transmitted back to the main-chain's VRFProvider contract together with a Merkle tree proof for its inclusion on the BandChain's state. This proof is then checked with the Bridge contract. Upon successful checking, the result is returned to the original calling dApp.

The above protocol flow is depicted in Figure 3. We will now look into each part of this process in more detail.

4.1.1 Key Registration

As explained above, the BandChain's ecosystem includes multiple independent data providers. For our Verifiable Randomness application, each provider has its own VRF key-pair, generated via VRF.KeyGen. Recall that it is important that these keys are generated ahead of time and independently from any seed inputs—otherwise it may be possible to choose a key that will give a desired output for a specific input. After generating their keys, each provider registers its VRF public key with the BandChain. For the first deployment phase of our Band Verifiable Randomness Protocol, these public keys are hard-

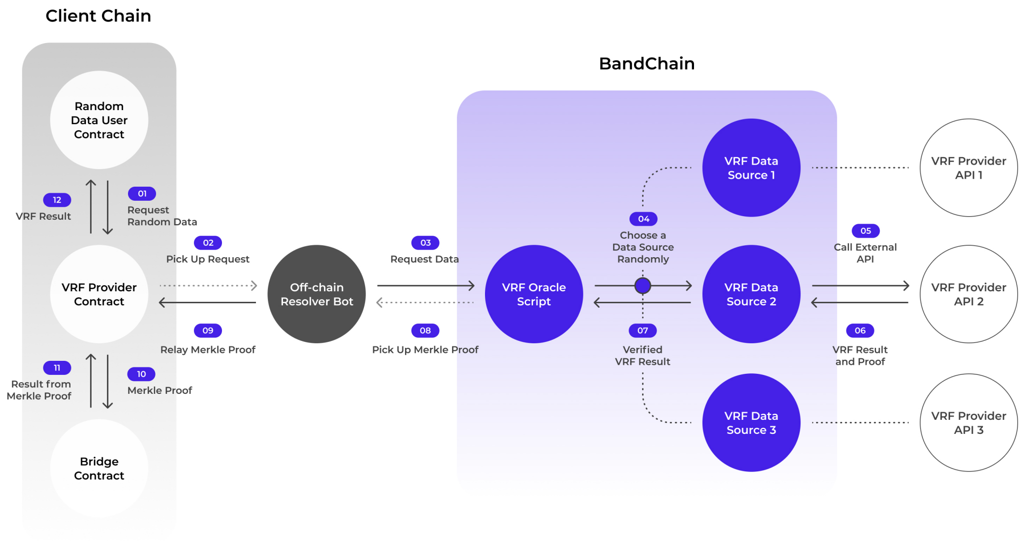

Figure 3: Band VRF Protocol Flow.

coded in the data sources. In Section 6, we discuss how this will be expanded to allow for efficient new provider registration, as well as key-rollovers of existing ones for the future versions of our protocol.

4.1.2 Seed and Request Pre-processing

Before forwarding the request to the BandChain, VRFProvider contract performs a pre-processing phase to translate the original seed provided from the consumer dApp to the expanded seed input that will be submitted to the VRF. This step takes place for security reasons in order to mitigate possible abuse of the system from malicious actors. In particular, during this phase, the submitted seed is appended with the latest main-chain block header, a timestamp indicating when this request is submitted, the chain ID, and a unique task nonce. These are then hashed using the cryptographic hash function SHA3-256 to create the final *expanded seed* that will be used as the VRF input.

4.1.3 Choosing the VRF Provider

After the request is relayed to the BandChain, the VRF oracle script loads the list of data sources that can be used to respond to it. Each VRF provider corresponds to one such data source that contains information to generate the HTTPS request to contact it, as well as its registered VRF key and when it became available. The next step is to choose the provider that will be assigned to execute this request. This choice is made by choosing one of the available providers/data-sources at random by hashing the request input. The result is then mapped to one of the data-sources; considering the hash function as a random oracle, each provider has the same probability to be chosen. From a utility perspective, this step is necessary in order to "map" the request to the VRF public key (and consequently to the VRF secret key, from the security properties described in Section 3) that will execute it. However, there are different ways to perform this, e.g., assign providers in a round-robin fashion. In Section 5 we provide more details about our choice to randomly assign the providers.

4.1.4 VRF Verification at the Validators

Next, the oracle script chooses a set of validators to execute the request. These parties are responsible for executing the data source corresponding to the selected *VRF Provider API*, collecting the response, and verifying the response using VRF Verify algorithm with its public key (as specified in the oracle/data source). If verification is successful, validators will forward the VRF value to the oracle script for the aggregation phase.

Validator sets are chosen using the pre-existing BandChain randomized validator selection process. This has similarities with the data source selection process outlined above. However, it takes into account the staked amount of each individual validators and prioritizes those that are more heavily vested (whose incentives are aligned to perform smooth and correct execution of the protocol). The final selection choice is randomized, weighted according to parties' stakes. We defer interested readers to [whi] for a more detailed description.

4.1.5 Returning the Result

After collecting the specified number of validators' responses (VRF values and proofs), the oracle script aggregates the results and uses the majority as the final VRF result – containing the value and the proof – to be the most frequent one. This result, together with request-specific information, is included in the next BandChain block via a Merkle path. The path and the result are then returned to the calling VRFProvider contract (again, assuming an incentivized actor, like a bounty hunter). It then uses the Bridge contract to verify the Merkle path, attesting the inclusion of this request's result in the BandChain, and returns the final result to the original calling dApp, closing the loop for this request.

5 Security Considerations

Having described the flow of our protocol, let us now discuss some of its security aspects. Starting with key registration, it is important that the VRF public key of the provider serving a given request has been registered *before* the request appears. Otherwise, providers might be able to choose a specific (sk, pk) pair to achieve a certain output y. Currently, public keys are stored on the corresponding data sources, and the data sources' ID are stored on the oracle script. As a result, only the data sources that are on the oracle script will be considered. Also, recall that our VRF implementation achieves full security, i.e., all public keys are checked for well-formedness.

5.1 Malicious dApp Owners

One important concern is to ensure that dApp owners cannot materially affect the returned output for a request (e.g., it should not be possible to submit a seed for which they already can predict the output). First, the provided seed is expanded with the latest main-chain block header, timestamp, chain ID, and unique nonce. This makes it hard for a dApp owner (assuming of course they cannot manipulate the main-chain itself) to predict the correct expanded seed that is produced by hashing these values together. Even if a dApp submits the same seed the expanded seed would be entirely different. Given the unpredictability property of the VRF explained above, this makes the returned output hard-to-guess. This also works as a protection against *replay attacks* where, after seeing the result for a given seed, a dApp manager may submit the same seed again to repeat the outcome. One unavoidable consequence is that it allows misbehaving dApps to keep submitting requests for the same seed hoping to get different results, if the returned one is not "favorable." Since the outcome is not fully determined from the seed alone, this becomes a possibility. Although this kind of "attack" is transparent (irrefutable evidence of it can be found on the main-chain) and thus may not be profitable. But as an additional preventive measure, the VRFProvider contract is made to be *stateful*. It maintains a list of submitted caller-seed pairs, and in case of repetition of the same seed from a caller, it does not go forward with the request.

5.2 Choice of Data Provider

Our seed-preprocessing also limits what can be achieved under the extreme case of a collusion between a dApp owner and a BandChain data provider. Even if the dApp owner has access to sk, by not being able to control the input limits the ability to manipulate the VRF output. Moreover, we take one more precautionary step by using a randomized assignment of providers. In practice, since the request input (extended seed) contains the corresponding timestamp and the latest main-chain block header, it is hard to guess ahead of time (prior to the time the request is processed by the VRFProvider contract) which provider will be assigned. So, even in the extreme cases that a dApp owner colludes with a data provider on the BandChain, not only do they need to be able to predict the expanded seed value but also be lucky enough so that this provider is assigned to serve the request. Note that this is better than other existing approaches that use a hard-coded VRF *pk* on the main-chain contract—in which case it is "easier" for such a collusion to succeed.

That said, one side-effect of our chosen method of assigning validators is that, since we map the request to a single provider, it is now possible that some requests are not served when the assigned provider is unavailable (e.g., due to connectivity issues). The alternative would be to map this to multiple providers and then deduce the final result using some chosen rules, e.g., by XORing all the providers' VRF outputs, choosing the first one to respond, picking one at random, and etc. While this helps mitigate the unavailability problem, it introduces another subtle issue. Assuming k providers are mapped to the request, this allows for multiple possible final results, depending on how these providers operate. As a concrete example, consider the case where the XOR of the different VRF outputs is taken as the final result. If k providers (not necessarily all of the mapped ones) collude with the intent of manipulating the result, they can choose from 2^k different results simply by deciding who opts-in or out. Observe that this strategy does not break the VRF security in any way–it just manipulates how the VRF is used. Motivated by these kind of issues, we decided to follow our explained approach that maps each request to a single data-source, i.e., a single VRF public key and in turn a unique valid final result. To avoid unavailability issues, we rely instead on incentive mechanisms that reward VRF providers for providing results (and this is fully aligned with the incentives of the delegated proof-of-stake model of the BandChain).

5.3 Result Verification

Due to our mechanism that assigns a single data provider (and consequently a unique VRF *pk*) to the request, the only way that the aggregated result of the BandChain oracle script is not the correct VRF output, given a properly computed expanded seed and a correct *sk* corresponding to this *pk*, is if the majority of the chosen validators to participate in the request behave dishonestly and coordinate to provide a fake result. Under the assumption that the majority of validators participate honestly, the oracle's aggregated output is the correct VRF evaluation.

Finally, the result is provided back to the main-chain together with a Merkle path that attests the inclusion of this request's full tuple in the BandChain ledger (tested by the Bridge contract). Barring an attack by a dishonest set of stakeholders with more than 1/3 of the total state, this suffices to guarantee the correctness and finality of the returned VRF output to the dApp.

5.4 Good Practices for dApp Developers

We provide here some general good practices regarding seed generation for dApps that plan to use the BandChain Verifiable Randomness, which complement our defense mechanisms. First, it is important to make the seed as unpredictable as possible. Even though we deploy several mechanisms to achieve this, it is still useful to try to ensure this at the dApp side as well. Second, in the context of the dApp, there should always be a "locking" point after which no new inputs or bids are accepted. For instance, after the seed has been generated and expanded, it becomes more plausible for parties to collaborate with the chosen VRF provider, learn the output before it is published, and place a "winning" bet. Finally, we plan to open-source our own library for dApp development tools for applications that wish to interact with the BandChain Verifiable Randomness, and we encourage developers to use this set of tools.

6 Future Steps

Finally, we discuss here some of the future expansions we plan for the upcoming versions of the BandChain Verifiable Randomness Protocol.

Currently, VRF public keys are stored in data sources. This mechanism for key registration allows us to easily handle key roll-overs or the addition of new providers, as each provider can easily update its own data source entry (instead of having to deploy a new oracle script each time). One feature we plan to add is to use standard key-rollover techniques (e.g., dual key pre-publication) to ensure smooth transition. One key point with this additional feature is that we need to ensure the oracle script only contains data sources with “live” public keys. Also, it is important to ensure that no public key that has been posted to a data source *after* the request’s timestamp is considered.

Another important feature that will be included in future versions is *on-chain verification* of the result by the BandChain oracle script. Recall that from the VRF uniqueness property, it is impossible for a “faked” response to pass verification. Hence, with this security feature in place, even a dishonest majority of validators cannot provide a fake result, making our protocol extremely robust to such misbehavior and entirely transparent.

References

[BCC$^+$21] Lorenz; Chacin Breidenbach, Christian; Chan, Benedict; Coventry, Alex; Ellis, Steven; Juels, Ari; Koushanfar, Farinaz; Miller, Andrew; Magauran, Brendan; Moroz, Daniel; Nazarov, Sergey; Topliceanu, Alexandru; Tramer, and Florian; Zhang. Chainlink 2. 0. The Next Steps in the Evolution of Decentralized Oracle Networks Economics and Cryptoeconomics Staking. *research. chain. link/whitepaper-v2.pdf.*, 0.(78.):2, July 2021.

[BDL$^+$11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-Speed High-Security Signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 2011.

[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 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 66–98. Springer, 2018.

[GHM$^+$17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68. ACM, 2017.

[Gol00] Oded Goldreich. *Foundations of Cryptography: Basic Tools.* Cambridge University Press, USA, 2000.

[GRPV21] Sharon Goldberg, Leonid Reyzin, Dimitrios Papadopoulos, and Jan Včelák. Verifiable Random Functions (VRFs). Internet-Draft draft-irtf-cfrg-vrf-09, Internet Engineering Task Force, May 2021. Work in Progress.

[MRV99] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable Random Functions. In *40th Annual Symposium on Foundations of Computer Science, FOCS ’99, 17-18 October, 1999, New York, NY, USA*, pages 120–130. IEEE Computer Society, 1999.

[PWH$^+$17] Dimitrios Papadopoulos, Duane Wessels, Shumon Huque, Moni Naor, Jan Včelák, Leonid Reyzin, and Sharon Goldberg. Making NSEC5 Practical for DNSSEC. Cryptology ePrint Archive, Report 2017/099, 2017. https://eprint.iacr.org/2017/099.

[whi] BandChain Whitepaper. https://docs.bandchain.org/whitepaper/. Accessed: 2021-08-31.