Verifiable Random Peer Sampling

Mauro Staver
mauro.staver@epfl.ch
EPFL

Supervised by: Pierre-Louis Roman

January 1, 2023

Contents

1 Introduction

Distributed ledgers, often called blockchains, have recently emerged as an alternative to classic centralized ledgers, offering a more balanced distribution of power between users. As such, they lead to increased security, transparency, and decentralization, which reduces the overall risk of fraud, errors, and censorship. In particular, public (permissionless) blockchains tackle networks in completely open settings and are the key technology enabling the innovative field of cryptocurrencies.

Public blockchains generally use consensus mechanisms to determine who is allowed to write on the ledger. We consider blockchain-based systems to be Byzantine Fault Tolerant (BFT) if the participants can reach consensus despite the presence of Byzantine faults (i.e., malicious or faulty nodes that may try to disrupt the consensus process). Well-known consensus mechanisms that meet the BFT property include Proof-of-Work (PoW) [1, 2, 3] and Proof-of-Stake (PoS) [4, 5], which are used by Bitcoin and Ethereum, respectively.

More recently, various BFT epidemic algorithms [6, 7, 8] have been proposed to address issues related to traditional consensus mechanisms. These algorithms enable extremely fast dissemination of information in very large networks at low cost. They typically operate by repeatedly sampling small sets of random peers in the network and communicating with them to estimate the overall system state and ensure coordination among correct (non-Byzantine) nodes.

As such, it is critical that nodes have access to fresh samples of random peers such that the proportion of Byzantine nodes in the sample is kept as low as possible. Providing such samples is the role of a so-called Secure *Random Peer Sampling* (RPS) service [9].

1.1 Network Attacks

More generally, an RPS service is a key determinant that defines the topology of the network underlying blockchain-based systems. While network topologies are often abstracted for convenience, higher-level protocols tend to make strict assumptions on their key properties. For example, most public blockchains assume strong network connectivity, i.e., that a path exists between any two nodes in the network. Such assumptions create requirements that are difficult to enforce in large open systems and can be exploited via network-based attacks to generate profit for attackers at the expense of victims.

For example, Sybil attacks can easily undermine these assumptions. In a Sybil attack, a malicious entity creates an arbitrary number of fake nodes in order to manipulate the network to their advantage. Eclipse attacks, a type of Sybil attack, can break the assumed strong connectivity by isolating targeted victims from the rest of the network by creating a virtual wall of Sybil entities [10] around them. Since an Eclipse attacker controls the inbound and outbound traffic of targeted nodes, they can send selected information to the victims and trick them into accepting double-spend transactions [11]. More subtly, Eclipse attacks can also be used by miners to gain an advantage over their competitors [12], preventing them from mining useful blocks or broadcasting them to the rest of the network.

Optimistic distributed ledgers, typically implemented through the aforementioned epidemic algorithms, are particularly vulnerable to these types of attacks. In these ledgers, nodes decide to add a record to their version of the ledger simply if a majority of their neighbors in the network graph do the same. As a result, attackers do not even need to completely eclipse a victim node to manipulate it. Merely placing a majority of Sybil nodes in the victim's neighborhood (i.e., a partial Eclipse attack) is sufficient to control the victim's decision. One example of such a ledger is Avalanche, valued at more than 3 billion USD as of January 2023.

A carefully-designed RPS service is therefore critical for the proper functioning of epidemic algorithms, as well as public blockchains in general. Diverse peer samples with a minimal proportion of Byzantine nodes should result in a topology with strong network connectivity among correct nodes, allowing them to efficiently disseminate information. However, designing such a service is challenging in the adversarial context of Byzantine nodes and Sybil attackers.

1.2 Secure Random Peer Sampling

Due to high churn in public blockchains, nodes typically don't have complete knowledge of the membership, i.e., every node doesn't know every other node. Therefore, Random Peer Samplers (RPSs) have traditionally been implemented via gossip algorithms, where nodes periodically exchange neighbors with their neighbors. Such approaches achieve both strong network connectivity and low network knowledge (logarithmic in the size of the network), even under high node turnover [13, 14]. Unfortunately, most

gossip-based RPSs [15, 16, 17] assume honest behaviors from nodes, hence are unable to defend well-behaving nodes from malicious ones, and those that do only protect against a small scope of adversaries.

For instance, secure peer sampling [18] focuses on parrying hub attacks (i.e., few nodes with many connections) and does not address attacks using many Sybils with few connections, such as Eclipse attacks, while Brahms [19] does not assume the existence of "massive Sybil attacks" altogether. More recently, HAPS [20] and BASALT [21] use the network's physical topology to limit the presence of Sybil nodes probabilistically; however, they still do not guarantee complete protection against Sybil attacks and do not provide a mechanism for eclipsed nodes to recover and continue to make progress.

1.3 Verifiable Random Peer Sampling

Prominently, none of the approaches we know of offer *accountability*, the ability to hold nodes responsible for their actions within the system. Indeed, if such a mechanism existed in a peer sampling setting, we could punish malicious nodes that try to bias the randomness of the samples. In cryptocurrency systems, accountability goes hand-in-hand with financial incentives, allowing honest nodes to take staked tokens from malicious nodes and redistribute them to honest ones. More generally, malicious nodes could be expelled from the system, eventually reducing the frequency of attacks and improving the overall system performance.

Accountability can be achieved by an even stronger property named *verifiability*. Verifiability allows nodes to cryptographically verify that another node has followed the designated protocol correctly. For example, in the context of peer sampling, a node would be able to verify that the received sample is not biased in any way. In case a node detects a biased sample, it can reject it and immediately accuse the node responsible for it, yielding accountability.

Our contribution.
Recognizing the utility of these properties, we propose *Verifiable Random Peer Samplers* (VRPSs), a new class of peer sampling protocols aimed at providing defense for large-scale public blockchains against network attacks. A VRPS allows nodes to sample peers in a way that allows them to verify that the samples are unbiased. As such, it can be used to build a random network overlay while preventing attackers from placing Sybil nodes at desired positions in the overlay, deterring Eclipse attacks. In the subsequent sections, we will formally introduce VRPSs, provide a general framework for implementing a VRPS, explore various distributed systems models in which this framework can be applied, and discuss potential implementation techniques.

Our main contribution is *Ora*, a first in the line of, hopefully many, VRPSs. Ora operates within the Oracle model, which defines two roles: sampling oracles, responsible for providing peer samples, and clients that request samples from oracles. We consider a strong adversary, achieving safety in the *Anytrust* model, which merely assumes that at least one sampling oracle in the entire network is honest.

The high-level idea behind Ora is quite straightforward: clients register themselves at oracles and get a signed ticket as confirmation that they have a chance of being included in future samples produced by that oracle. Upon querying, the oracle must return a sample from the set of all clients to whom it issued a ticket. Accompanying the sample is a cryptographic proof that testifies that the sample was computed correctly using a verifiable source of randomness.

Ora makes extensive use of Merkle trees, a data structure that allows for efficient and secure verification of the contents of a large database without having to transmit the entire database. In particular, oracles maintain a Merkle tree of registered clients, allowing them to produce short proofs testifying the inclusion of any registered client. Furthermore, to ensure that the set of registered clients is consistent across queries, oracles use append-only Merkle trees that also allow for verifying that a previously committed subtree is included in the current tree.

Such a design still comes with a plethora of issues. Malicious oracles can, for example, include certain clients multiple times in the Merkle tree, giving them a higher chance of being sampled. They can also present inconsistent trees to different clients in a way that each tree is consistent with the previous version of itself (i.e., a split-view attack), not allowing the targeted clients to detect any inconsistencies. We show how to resolve these shortcomings and others in Sections 4 and 5.

We achieve a system in which an oracle's misbehavior might not be detected immediately, but it will eventually be detected. The time until detection naturally trades off with system performance, allowing immediate detection at the expense of network traffic and high strain on specific nodes. By using randomization, we can essentially circumvent this trade-off, allowing misbehaviors to be detected both quickly and at a low cost. This fundamental insight enabled us to develop Ora, a practical VRPS.

2 Definitions

We will now formally define a Random Peer Sampling service and its variants. Let $A(S, r, k)$ be a deterministic sampling algorithm that outputs a sample of k elements from the set S, using r as a source of randomness. In the peer sampling context, the set S corresponds to the set of all active nodes in the network. A Random Peer Sampling service RPS^A is then defined as a service that, upon invocation, outputs the sample produced by A. As noted in the discussion above, a Secure Peer Sampling service $SRPS^{A_s}$ is an RPS service parameterized by a secure sampling algorithm A_s that ensures a large diversity of nodes in the sample, while minimizing the appearance of malicious nodes.

VRPS definition. We define a Verifiable Random Peer Sampling service $VRPS^A$ as an RPS service that also allows clients to cryptographically verify that its output is indeed the sample produced by A.

A VRPS can be used to implement an SRPS simply by parameterizing it with a secure sampling algorithm A_s suitable for an SRPS. In fact, a VRPS can, by definition, be used to implement an RPS parameterized by any sampling function A while adding the benefits of verifiability and accountability. This makes VRPSs highly practical in the adversarial context of public blockchains. Nevertheless, designing and implementing VRPSs for practical use is quite challenging and requires the use of advanced tools from the fields of cryptography and probabilistic proof systems.

Requirements. In order for the client to accept a sample p provided by the VRPS, he will have to be convinced of the following:

- parameters given to the sampling algorithm A, namely the set S from which we sample and the source of randomness r, are correct
- sample p is the result of executing the sampling algorithm A on given parameters S and r

We consider the source of randomness r to be correct if it is a random number not biased by either the VRPS or the client. As such, this parameter can be used as an unbiased source of randomness by the sampling algorithm A to generate unbiased samples. It is important to note that it is necessary for the client to be unable to bias the randomness, as the client will become a neighbor with the sampled peers. If clients are permitted to bias the randomness, they could maliciously choose their neighbors, allowing them to negatively impact honest nodes through inbound connections. We propose the use of *Verifiable Random Functions* (VRFs) as a cryptographic primitive used to generate r.

We consider the sampling set S to be correct if it contains all nodes that the VRPS knows about. Since clients don't have access to the memory of the VRPS, verifying the correctness of the sampling set is almost impossible to achieve in practice. Our peer sampler, Ora, requires nodes offering the VRPS service to store all registered clients in an append-only Merkle tree, enabling clients to ensure that no node is removed from the sampling set. As we'll show, such a setup is still susceptible to so-called split-view attacks, which we try to mitigate by clever coordination between honest nodes.

We require the sampling algorithm A to be deterministic so that its execution can be verified by *verifiable computing* techniques, such as SNARKs and STARKs. We aim to use such tools to produce proofs testifying the statement $p = A(S, r, k)$.

3 Cryptographic Tools

We will now provide a brief overview of each of the tools that were just mentioned and introduce cryptographic signatures as a well-known primitive that can be used to achieve accountability. Not only do we employ these tools in our solution, but we also consider them as a valuable toolset that can be utilized in designing any future VRPSs.

3.1 VRFs

Verifiable Random Functions (VRFs) are cryptographic functions that allow a prover to convince the verifier that some value was generated randomly. More specifically, they are verifiable pseudorandom number generators that, given an input seed value, produce an output that can be verified. They can be used to generate a source of randomness in a verifiable and unbiased way, and as such, they play a key role in the design of VRPSs.

Given the input value x, the owner of the secret key can produce the output of the VRF, as well as a short proof attesting to the output. On the other hand, using the proof and the public key, a verifier can verify that output. In the context of VRPSs, we will have clients choose the input value x, and VRPSs provably generate the randomness r. Since clients do not know the secret key, they cannot bias the output of the VRF by choosing its input, and since the VRPS does not know the input of the VRF beforehand, he is also unable to bias it.

3.2 Merkle Trees

A Merkle tree is a data structure that allows for efficient and secure verification of its contents. It has the following structure: each leaf node contains a data element, and each non-leaf node contains the hash of its children. Consequently, the root node of the tree, known as the Merkle root, contains the hash of all the data elements in the tree. The Merkle root can be used to commit the entire dataset contained in the tree. Merkle trees allow for generating proofs that attest to the inclusion of any leaf node in the tree. The size of these proofs is logarithmic in the number of leaf nodes contained in the tree.

We will make extensive use of append-only Merkle trees, where leaves can only be added and can't be removed or modified. Append-only Merkle trees allow for generating so-called consistency proofs that allow a prover to convince the verifier that two versions of the Merkle tree are consistent with each other. This means that the verifier can check that the later version of the tree includes all elements in the earlier version, in the same order, and that all new elements come after the elements in the older version. Similarly to inclusion proofs, the size of these proofs is logarithmic in the number of leaf nodes contained in the tree.

3.3 Verifiable Computing

Verifiable computing techniques allow verifiers to check the correctness of computations performed by provers. Specifically, the prover can perform the computation $f(x)$ and produce a short proof, testifying the correctness of the statement $y = f(x)$. These techniques have recently found widespread use in public blockchains due to their ability to efficiently implement zero-knowledge proof systems, which allow transactions to be verified without revealing sensitive information.

Various tools exist, such as SNARKs, which require a trusted setup, and STARKs, which do not require a trusted setup but suffer a larger proof size. Due to their complexity, we will use these tools as abstractions, skipping their implementation details.

3.4 Signatures

Cryptographic signatures are a well-known tool in the field of cryptography, used to provide authenticity and integrity for a variety of applications. They use public key algorithms to allow provers to sign arbitrary data such that anyone can verify their signature.

We will use cryptographic signatures to achieve accountability. Specifically, VRPS provers will be required to sign all their proofs attesting to the validity of the samples they produce. If a client discovers an invalid proof or two conflicting proofs, he will be able to hold the VRPS prover accountable, as the signed proofs will identify the VRPS as the source.

4 Framework for Verifiable Random Peer Sampling

We propose a general framework that enables honest clients to obtain verifiably unbiased samples from a possibly malicious VRPS as follows:

1. Client and the VRPS agree on the sampling algorithm A, as well as the set S from which the sample will be drawn.
2. Client selects the sample size k and an input to the VRF x, and sends them to the VRPS.
3. VRPS computes $r, \pi_r = VRF(x)$, where r is the generated random integer, and π_r is a proof attesting to r.
4. VRPS computes $p, \pi_p = A(S, r, k)$, where p is the sample of k elements from S produced by A, and π_p is a proof attesting to p.
5. VRPS sends $(r, \pi_r), (p, \pi_p)$ to the client.
6. Client checks the validity of π_r and π_p and decides to accept the sample p if both checks pass.

In practice, the VRPS will store the set S as a Merkle tree, allowing the client and the VRPS to agree on a set without the client needing to have access to the entire set, but only the Merkle root. Therefore, the client and the VRPS will simply agree on the Merkle root, and from now on, we will use S to refer to this Merkle root. As we'll see, agreeing on S in a way that does not allow a malicious VRPS to bias the samples is a central challenge in the design of VRPSs.

Since Merkle trees do not correspond directly to sets but rather to lists; we require VRPSs to store each node only once in the Merkle tree, not allowing for duplicate nodes as this gives them a higher chance of being sampled, thus biasing the randomness of the samples. To address this issue, it will prove helpful to associate each element with its index in the underlying list, allowing one to verify that an element is located at its specified index in the Merkle tree. If the same element is detected at different indexes, it can be inferred that the VRPS is cheating.

In step 4, we resort to using verifiable computing techniques (SNARKs, STARKs, or similar) to produce the succinct proof π_p, attesting to the correctness of the computation of A. We will now present the generic form of the sampling algorithm A, which can be used as a template to implement specific sampling algorithms.

Algorithm 1 Generic Sampling Algorithm

Input: Merkle root S, set size n, randomness r, sample size k
Output: sample p

```
rng = RNG(r)
i = 0
L = empty list

while(i < k)
    idx = rng.next() mod n
    item = GetItemAtIndex(idx)
    proof = GetMerkleProof(idx)
    assert(item, proof)
    b = IncludeOrReject(L, item)
    if(b)
        L.append(item)
        i++

return L
```

The proposed algorithm repeatedly samples numbers in the range $[0, n-1]$ using a deterministic pseudorandom number generator (RNG) initialized with the randomness r. These numbers correspond to indexes associated with elements in the Merkle tree S. For each such index, the algorithm fetches the element at the specified index using the $GetItemAtIndex$ method and the Merkle proof using $GetMerkleProof$, which attests that the element is indeed located at the specified index in the

Merkle tree S. These two methods can be considered as *Oracle methods* implemented by the VRPS. Since we don't trust the VRPS, we need to verify that the chosen item is indeed at the specified index in the tree S, which we do by verifying the proof using the *assert* method. If the proof is invalid, the algorithm terminates and returns an empty sample.

At this point, we are confident that the chosen element was randomly sampled from the tree S (assuming r is correct), and we can decide whether to include this element in the sample using the $IncludeOrReject$ method, which returns a boolean decision. The logic of the specific sampling algorithm can be embedded in this function. For example, a uniform sampling algorithm would simply check if the element is already included in the sample.

The algorithm repeats this process until the sample size does not reach k, and then returns the sample as a list. The client can verify the correctness of this sample by simply re-executing the sampling algorithm and relying on the VRPS to return answers to the Oracle methods. However, this can pose a strain on the client, as he may have to check many Merkle proofs. To alleviate this, we use verifiable computing techniques, which allow the VRPS to provide a concise proof attesting to the correctness of this computation. The verifier can then simply check this single proof.

Secure sampling algorithm. To design a sampling algorithm that prevent Sybil attacks, we can take inspiration from the BASALT peer sampler. BASALT assumes two types of Sybil attacks:

1. *Institutional attacks*, in which the attacker may control many IP addresses located in a limited number of continuous address blocks.

2. *Botnet attacks*, in which the attacker may control a smaller number of addresses in the whole IP address space.

We can use this assumption to implement the $IncludeOrReject$ method, which decides to include a peer in the sample based on its IP address and the IP addresses of the peers already included in the sample. The resulting sample will then include a minimal proportion of Sybil nodes. Other, application-specific sampling algorithms may be used to achieve the desired network topology. For example, some applications can benefit from, and assume, a tree-like topology.

We will now discuss how this framework can be applied in various distributed systems models.

4.1 p2p Model

In the p2p model, there are no central authorities offering the peer sampling service. Instead, each node offers the peer sampling service directly to its peers. Since most traditional peer samplers operate in this model, it is a natural starting point. Therefore, we consider a system in which each node offers the VRPS service to its current peers.

Assume two peers, where one plays the role of the client, and the other plays the role of the VRPS. Following our framework, the client and the VRPS first need to agree on the sampling set. The VRPS will commit to a Merkle root S, and the client will have to ensure that S doesn't correspond to a malicious set chosen by the VRPS.

The sampling set should contain the current peers of the VRPS, which the VRPS obtained by querying his previous peer's VRPSs, playing the role of the client. This insight allows us to understand the recursive nature of a node's peer set, namely that its current peer set is a function of its past peer set.

Assume a base case where each node is given a set of random peers resulting in a starting topology, i.e., assume that each node's starting peer set S_0 is correct. Furthermore, assume that the client queries the VRPS at a time when the VRPS's peer set is S_t, meaning that the VRPS updated his peer set t times by querying other VRPSs.

In order to convince the client that his current peer set S_t is correct, the VRPS would have to inductively prove that each transition $S_i \to S_{i+1}$ is correct, starting from S_0.

We envision two possible methods that allow the VRPS to do so:

1. *Recursive proof compositions*, that allow a prover to construct a proof for an arbitrarily complex statement by combining a series of sub-statements that together imply the larger statement. The VRPS would have to provide a recursive proof attesting to the validity of each set transition, resulting in the current set S_t.

2. Using a *distributed ledger* to record the proofs of all set transitions from each node in the system.

Since nodes typically sample new peers every few hours or even minutes, the second method involving the use of a distributed ledger is not practical due to the high overhead that the ledger imposes.

4.2 Oracle Model

The oracle model is similar to the traditional client-server model. Typical client-server architectures are not suitable for use in public blockchains because they require clients to trust centralized servers. However, by introducing verifiability, we can eliminate the need for clients to trust servers, enabling the use of similar architectures in public blockchains.

In the oracle model, we define two roles: *sampling oracles* and *clients*. The oracles provide the VRPS service to clients, and allow the onboarding of new clients by adding them to their node set, enabling them to be included in future samples, a process we refer to as *client registration* or *bootstrapping*. We believe that the Oracle model is superior to the p2p model because the samples are drawn from a much larger pool of peers, resulting in higher-quality samples.

The oracle's node set, in a public blockchain setting, may become overcrowded with inactive nodes due to the high churn rate, where nodes frequently join and leave the network. This can lead to samples being populated with inactive nodes. To prevent this issue, each client is given a time period during which they are considered alive. After this time has passed, an oracle will declare the client as dead unless the oracle hears back from them.

In other words, each client registration will have an expiration time T, after which an oracle will consider the client to be dead unless he re-registers in the meantime. We consider a client registration to be *active* if its time to expiration hasn't yet expired.

Oracles must satisfy the following requirements:

1. maintain only a single set of nodes
2. accept all valid register requests
3. include only active registrations in the set
4. include all active registrations in the set
5. respond to client requests by providing verifiable samples from the set

Following our framework, we are again tasked with designing a mechanism that allows clients and oracles to agree on a sampling set, or rather its Merkle root. In general, an oracle will propose a Merkle root S and the client has the option to either accept or reject it. A client should accept the proposed set if he is convinced that the above requirements are met. However, proving that an oracle's set contains only all active registered clients is difficult since a malicious oracle could always leave out certain clients and purposefully include malicious ones.

We can attempt to use append-only Merkle trees by having the oracles use them to store all registered clients. A client could then keep the oracle's last known Merkle root, allowing him to use Merkle consistency proofs to at least ensure that the oracle's set is growing consistently across queries. For example, upon registering for the first time, a client will receive a Merkle root S_t of the (supposedly) current oracle's set. The next time a client queries this oracle, the oracle will propose the root $S_{t'}$, and the client can verify that $S_t \to S_{t'}$, convincing him that the newly proposed set contains all of the elements from the previous set.

Even though this setup has many flaws, it is a step in the right direction. We can identify two significant problems: First, the oracle is unable to remove dead nodes from its tree due to it being append-only. Second, such a setup is susceptible to split-view attacks, in which the oracle can present a successive sequence of mutually consistent trees to a single client; however, he can also provide different sequences to different clients. Therefore, the client has no way of knowing whether his sequence contains all of the active nodes.

We will now describe our VRPS, Ora, explaining in detail how it deals with the aforementioned issues.

5 Ora VRPS

Ora is a VRPS that operates within the Oracle model and assumes a highly adversarial environment. Despite that, it relies on verifiability to achieve safety in the sense that a malicious oracle will quickly be detected for cheating, allowing clients to hold it accountable.

As described in the previous section, Ora's oracles store all registered clients in an append-only Merkle tree. The key insight that enabled us to solve the aforementioned issues related to this setup is the following. If we can enforce an oracle to present only a single global sequence, then newly joined honest clients could force the sequence to progress by demanding a proof that their registration is included in the set. As such, a client querying the oracle for a fresh sample can be sure that a proposed set is correct if the Merkle consistency proof testifying $S_t \rightarrow S_{t'}$ is valid, where S_t is the last known Merkle root that the client knows about and $S_{t'}$ is a newly proposed root.

Ora manages to accomplish this to a large extent by introducing a new role called *validators*. Validators act as intermediaries between clients and oracles, through which all traffic must pass. This allows them to have a broad overview of the proposed Merkle roots and ensure that only a single sequence progression is presented globally. Ora manages to achieve safety in the Anytrust model, by merely assuming that at least one validator is honest.

Setup. We consider a static set of oracle nodes, a static set of validator nodes, and a dynamic set of client nodes. Assume that the number of oracles and validators is logarithmic in the number of clients. Furthermore, assume that all clients know about all oracles and validators. Note that we consider the sets of oracles and validators to be static purely to simplify our discussion. We can allow for dynamic sets by using, for instance, asynchronous reconfiguration [22].

Next, we only assume that at least one validator is honest, allowing for all oracles and all but one validator to be malicious. We also make no assumptions on the honesty of clients, but we are only concerned about providing unbiased samples to honest clients. We allow malicious clients, oracles, and validators to collude in an attempt to bias the samples of honest clients, creating a highly adversarial environment.

Obtaining samples. Let S_o be the Merkle root corresponding to the current set maintained by some oracle and S_v, S_c be the last Merkle roots (previously committed by that oracle) that some validator and client know about, respectively. A client wanting to obtain a sample will first choose a validator and an oracle. He will then contact the validator, sending him S_c and x, the input for the VRF.

Upon receiving the request, the validator will in turn query the oracle, sending it S_v, S_c, x. The oracle should respond to the validator by returning him the commitment to a set S_o, Merkle consistency proofs $S_c \rightarrow S_o$ and $S_v \rightarrow S_o$, and $(r, \pi_r), (p, \pi_p)$, where r is the randomness outputted by the VRF, p is the sample computed on a set corresponding to S_o, and πs are the proofs attesting to each. The validator will then check the consistency proof $S_v \rightarrow S_o$, update its S_v by setting it to S_o, and forward the oracle's message back to the client. The client will check the consistency proof $S_c \rightarrow S_o$, update its S_c by setting it to S_o, check the proofs π_r, π_p and choose to accept the sample p if all checks pass.

To gain intuition, assume that only a single validator exists in the set of validators, making him the honest one. We claim that, under this assumption, this protocol results in the client only accepting unbiased samples from the set of all correct clients. Since all correct clients will follow this protocol, every request will pass through the single validator, creating a multiplexing effect that ensures only a single sequence of tree roots, preventing split-view attacks. Indeed, if a malicious oracle tries to respond with inconsistent roots to different requests, the validator's consistency check will fail.

Of course, assuming a single honest validator is unrealistic; not only does it imply trusting a centralized server, going against the decentralized principles of public blockchains, but it also makes the system unscalable. Nevertheless, the underlying idea of the protocol remains the same when we assume a set of multiple validators, where at least one is honest.

Randomization. To adhere to this assumption, we define a way for clients to choose validators and oracles in a way that guarantees that a client will query a given oracle through the honest validator within a fixed number of sampling rounds. More specifically, a client will, for a fixed oracle, choose a different validator in every subsequent sampling round until all validators have been exhausted. This way, he is guaranteed to stumble upon the honest validator within a fixed number of sampling rounds.

When this happens, the honest validator will detect whether the client suffered a split-view (i.e., obtained samples from an incorrect set) in previous rounds. The honest validator is able to do so because the client will send him an inconsistent S_c, obtained from the oracle in previous rounds. Note

that after an oracle provides an inconsistent root to a client, all future roots that the client receives must be consistent with this root, because the client checks the consistency of the roots within his own local sequence. Simply put, once an oracle commits an inconsistent root, it cannot take it back. Therefore, an inconsistent root commitment is guaranteed to reach the honest validator where it will be detected.

In practice, a client will want to periodically obtain fresh samples. We will now provide the pseudocode for the peer sampling loop that periodically chooses a validator and an oracle, and invokes the protocol to obtain samples.

Algorithm 2 Peer Sampling Loop

Input: time interval t, validator set Val, oracle set Ora

```
Val = Permute(Val)
Ora = Permute(Ora)

while(true)
    for(i = 0; i < Val.size; i++)
        for(j = 0; j < Ora.size; j++)
            p = ObtainSample(Val[i], Ora[j])
            sleep(t)
```

The peer sampling loop first computes the random permutations of the validators and the oracles via the *Permute* method. Then for each validator, it iterates over the oracles, obtaining a sample for each pair. It should be clear that we achieve the desired guarantee in this way. Namely, that the client will, for any given oracle, query it through the honest validator within a fixed number of sampling rounds. An extra benefit of using this algorithm is that it naturally achieves load-balancing, a complementing side effect. It might be unclear why we have to first query all oracles for a given validator, and not the other way around, but this is due to the registrations' expiry times discussed in Section 4.2, which we will now revisit.

Registrations/Bootstrapping. When a client queries an oracle, the request also counts as re-registration. Therefore, the oracle should also return, except for the sample-related messages, a signed ticket with a timestamp indicating the time until which it will consider the client to be alive. To simplify our discussion, we assume that all oracles use the same registration expiry time T, meaning that the client will be declared dead after time T from the point of the query.

In this case, the client should query oracles in a circular round-robin fashion, querying the next one every time period T/o, where o is the total number of oracles. This way, he will remain registered with all oracles at all times, giving him an unbiased chance of being included in the samples of other clients. In other words, the Peer sampling loop defined above should be given $t = T/o$ as input.

Dead nodes. We still haven't described a mechanism that enables us to exclude dead nodes from the samples. Due to the Merkle tree being append-only, the oracles cannot directly remove dead nodes from its tree; rather dead nodes will implicitly be skipped when executing the sampling algorithm $A(S, r, k)$.

To do this, we will introduce another *Oracle method*, namely $IsDead(item)$, which returns true if the oracle deems the given node as dead. As with the other Oracle methods, clients need to have a way to verify their output. Referring back to Section 4, the outputs of the Oracle methods $GetItemAtIndex$ and $GetMerkleProof$ were verified by including the execution of the Merkle proof inside the sampling algorithm A, which the VRPS then executed and provided a proof attesting to the correctness of its execution. Since an oracle cannot construct a proof attesting to the statement that a node is dead, we don't have anything to include in the sampling algorithm A. Still, we can catch a malicious oracle trying to declare a live node as dead by the following mechanism.

We modify the sampling algorithm A to also return a list of all the nodes that the oracle declares as dead, or if we expect a large number of dead nodes, a small sample of the declared dead nodes. The client will then try to contact the (supposedly dead) nodes from this sample. If a certain node is alive, it will respond by returning a valid ticket signed by the oracle, attesting that its registration should still be considered active. The client now has two conflicting statements signed by the oracle and can accuse the oracle of cheating. Note that we require the oracle to sign all of its messages.

6 Discussion

Guilty until proven innocent. There are still some caveats and potential ways that adversaries can try to hurt honest clients that we haven't yet discussed. For example, an oracle can selectively choose to answer only specific queries, not allowing certain clients to enter its sampling set. This way, a malicious oracle can try to bias the randomness of its samples by manipulating who gets included in its set.

As a countermeasure, we propose that clients or validators inform other members of the system by gossiping the register request signed by the harmed client. The honest members of the system would then try to query the specified oracle with that register request, refusing to query it with their own queries until they get a proof that the harmed client is included in their set. This way, the malicious oracle would quickly lose all its honest clients or be forced to include the harmed client in its set, allowing for the system to continue operating normally.

In the design of VRPSs, we generally recommend taking the approach that assumes nodes as being "Guilty until proven innocent". Since a node not wanting to provide a proof generally means that it knows that the proof would allow honest nodes to keep it accountable for malicious behavior.

Duplication attacks. As indicated in Section 4, an oracle can try to include certain nodes multiple times in its tree, giving them a higher chance of being sampled and thus biasing the produced samples. Therefore, we required that each element be associated with its index in the list underlying the Merkle tree. This can be done by simply hashing the node with its associated index, using the hashed value as the leaf of the tree. Thus if an oracle provides the same element, sampled from different positions in the tree, we can accuse him of cheating.

To detect this, the validators can store a hashmap mapping between nodes and their stated indexes, updating this hashmap upon receiving every sample. Since oracles must provide a consistent sequence of tree roots, or otherwise they will be detected, once they commit a tree with a duplicate element, they can no longer remove the duplicate from the tree. Therefore, eventually, they will produce a sample containing the duplicate element, and an honest validator will be able to detect that the element is a duplicate. This means that the more duplicate elements the oracle adds, the quicker it will be detected.

We can speed up the detection of duplicates even more by requiring oracles to transmit to validators all nodes they added after S_v instead of simply transmitting the Merkle consistency proof $S_v \rightarrow S_o$, as we do in Ora. This would allow validators to maintain a copy of the oracle's Merkle tree locally, allowing for easy detection of duplicates.

Inbound connections. We should also consider how a client handles its inbound connections. He should not blindly trust another client contacting him and stating that he is his peer. The contacting client should send proof attesting that the client does indeed appear in an unbiased sample obtained by some oracle. To do so, the contacting client can send him $\pi_{p'}$, the proof of computation of the sampling algorithm A, and the committed root $S_{c'}$ corresponding to the set from which the sample was drawn. The client would then query the oracle with his own S_c to check whether $S_{c'}$ is consistent with his S_c and check the proof $\pi_{p'}$, accepting the connection if both checks pass.

Efficiency. In terms of efficiency, we expect Ora to be slower than traditional peer samplers due to the overheads that proof systems introduce. However, the overheads of Ora largely depend on the verifiable computing technique used to verify the computation of the sampling algorithm A, making it difficult to speculate on the performance of Ora without an implementation. That being said, a drop in performance is natural because VRPSs offer much stronger properties than classical RPSs, namely verifiability and accountability.

By default, Ora doesn't offer immediate detection of misbehavior, allowing clients to suffer biased samples for a fixed number of rounds. However, detection of malicious behavior can be made immediate if the client queries the oracle through all validators simultaneously, to ensure that the query passes through the honest validator. This would, however, lead to extremely high network traffic and poses a large strain on the validators, deeming the architecture to be unscalable. Our solution offers natural load-balancing by having the clients choose validators in a circular round-robin fashion in the order of its own random permutation. However, this implies a slower average time until the detection of malicious behavior. Simultaneously querying through a subset of validators can then be considered a middle-ground, allowing for quicker average detection time based on the increased probability of including an honest oracle while keeping the network traffic and the strain on the oracles low, depending on the size of the subset.

7 Conclusion

Public blockchains continue to pave the way as a promising technology with practical use cases. Due to the adversarial context in which they operate, it is necessary to build systems able to prevent attacks with minimal assumptions on the honesty of nodes. Attesting to this, we have introduced Verifiable Random Peer Samplers (VRPSs), a new class of peer samplers offering both accountability and verifiability, properties not exhibited by any previous peer samplers to our knowledge. VRPSs are specifically designed to thwart Sybill attacks, which attackers use to gain profits at the expense of their victims.

In this paper, we provide a general framework allowing the design of VRPSs that we hope will help future researchers to build the next iteration of VRPSs. We also defined two models in which this framework can be applied. Namely, the p2p model and the Oracle model.

We designed Ora, a first in the line of hopefully many VRPSs. Ora operates in the Oracle model and achieves safety in highly adversarial conditions. In future work, we aim to implement Ora, benchmark it with other peer samplers, and test its resiliency to Sybil attacks, measuring it's performance in the process.

References

[1] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In 2013 IEEE Symposium on Security and Privacy, SP '13, 2013.

[2] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[3] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper, 2014. http://bitcoinaffiliatelist.com/wp-content/uploads/ethereum. pdf.

[4] 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, SOSP '17, 2017.

[5] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Advances in Cryptology – CRYPTO 2017, 2017.

[6] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos- ˘ Adrian Seredinschi. 2019. The consensus number of a cryptocurrency. In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. 307–316.

[7] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and DragosAdrian Seredinschi. 2019. Scalable Byzantine reliable broadcast. In 33rd International Symposium on Distributed Computing (DISC 2019). Schloss DagstuhlLeibniz-Zentrum fuer Informatik.

[8] Team Rocket. 2018. Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies.

[9] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. 2007. Gossip-based peer sampling. ACM Trans. Comput. Syst. 25, 3 (August 2007), 8–es. https://doi.org/10.1145/1275517.1275520

[10] John R Douceur. The sybil attack. In International Workshop on Peer-to-Peer Systems, IPTPS '02, 2002.

[11] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In 24th USENIX Security Symposium, USENIX Security '15, 2015.

[12] Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In 2016 IEEE European Symposium on Security and Privacy, EuroSP '16, 2016.

[13] J. Leitao, J. Pereira, and L. Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), pages 419–429, 2007.

[14] Andr´e Allavena, Alan Demers, and John E. Hopcroft. Correctness of a gossip based membership protocol. In Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, PODC '05, 2005.

[15] M´ark Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. ACM Transactions on Computer Systems, 25(3), 2007.

[16] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. CYCLON: inexpensive membership management for unstructured P2P overlays. Journal of Network and Systems Management, 13(2):197–217, 2005.

[17] J. Leitao, J. Pereira, and L. Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), pages 419–429, 2007.

[18] Gian Paolo Jesi, Alberto Montresor, and Maarten van Steen. Secure peer sampling. Computer Networks, 54(12):2086–2098, 2010.

[19] Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. Brahms: Byzantine resilient random membership sampling. Computer Networks, 53(13):2340–2359, 2009.

[20] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure Routing for Structured Peer-to-Peer Overlay Networks. In Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation, OSDI '02, 2002.

[21] Alex Auvolat, Yérom-David Bromberg, Davide Frey, François Taïani. BASALT: A Rock-Solid Foundation for Epidemic Consensus Algorithms in Very Large, Very Open Networks. 2021. ffhal-03131734f

[22] Kuznetsov, P., Tonkikh, A. Asynchronous reconfiguration with Byzantine failures. Distrib. Comput. 35, 477–502 (2022). https://doi.org/10.1007/s00446-022-00421-1