Sassafras and Semi-Anonymous Single Leader Election

Jeffrey Burdges, Handan Kılınç Alper, Alistair Stewart, and Sergey Vasilyev

Web3 Foundation

Abstract. A single-leader election (SLE) is a way to elect one leader randomly among the parties in a distributed system. If the leader is secret (i.e., unpredictable) then it is called a secret single leader election (SSLE). In this paper, we model the security of SLE in the universally composable (UC) model. Our model is adaptable to various unpredictability levels for leaders that an SLE aims to provide. We construct an SLE protocol that we call semi-anonymous single leader election (SASLE). We show that SASLE is secure against adaptive adversaries in the UC model. SASLE provides a good amount of unpredictability level to most of the honest leaders while it does not provide unpredictability to the rest of them. In this way, we obtain better communication overhead by comparing the existing SSLE protocols. In the end, we construct a PoS-protocol (Sassafras) which deploys SASLE to elect the block producers. Sassafras benefits from the efficiency of SASLE and gains significant security both to grinding attacks and the private attack as shown by Azouvi and Cappelletti (ACM AFT 2021) because it elects a single block producer.

1 Introduction

Leader election is required for many distributed systems for security and efficiency mainly because it distributes the tasks and minimizing the coordination. Having a single leader introduces the single point of failure. The leader may be malicious or honest leaders can be attacked e.g. by a denial-of-service (DoS) attack. One solution to this is to frequently change the leader and to have a secret leader election protocol which hides the leader until they announce themself.

Leader election mechanism is inevitable for blockchain protocols if we define the leader as the node who is eligible to publish a block with respect to the rules of the blockchain. One very known election mechanism on blockchain protocols is proof-of-work. Here, a node (called miner) is elected as a leader if it generates a block of which hash is less than a threshold. Some proof-of-stake (PoS) protocols [17,28,18,22] use the similar approach as well with respect to the stake instead of computational power. Even though this election mechanism provides secrecy of the leader(s) and requires no communication with the other nodes, it has some drawbacks. The main problem is that the outcome is probabilistic since the leaders are selected in expectation with respect to their computational power or stake. Therefore, there can be multiple leaders or no leaders at all. If the election outcome has more than one leaders then the efforts of the leaders except one will be wasted because they are not aware of other leaders. If the election outcome has no leader then the time to extend the blockchain will be wasted. Besides the wasting problem, it has been shown by Azouvi and Cappelletti [1] that PoS-protocols with single secret leader election (SSLE) provides higher security comparing to the probabilistic secret leader election. Therefore, it is critical to have an election mechanism that selects only one leader to generate a block.

SSLE is a distributed protocol that selects only one leader in a slot (round) and hides the leader till the leader announces itself by providing proof. SSLE is secure [5] if it provides uniqueness (i.e., electing only one leader), unpredictability (i.e., hiding the leader), and fairness (i.e.,

having the same chance to become a leader). One can formulate SSLE as a multi-party computation (MPC) protocol where parties run a function with private inputs and obtain private outputs. However, MPC does not scale well in blockchain protocols because it requires communication with all parties. The other solution is using shuffle protocols [5,14,14] which shuffles parties randomly and selects e.g., the first party after shuffling. They are better than MPC in terms of communication and computational complexity but blockchain protocols scale better with less complexity considering the existence of too many parties.

We introduce a batched semi-anonymous secret single leader election, SASLE, that uses a cryptographic primitive known as a RingVRF [7]. A RingVRF provides private verifiable randomness, their VRF output, associated with nodes' keys. The node can provide proof of this VRF output without having to reveal its identity, only that they are one of a ring of keys. Later they can provide proof that it was their randomness specifically. If nodes had access to an anonymous broadcast functionality, then they can broadcast their VRF output with an anonymous RingVRF proof, the protocol can sort these outputs, and this sorting determines a leader order. Each leader then proves that this is their output when it is their turn to be a leader. Unfortunately, broadcast mechanisms that provide good anonymity are either complex or are not live when nodes misbehave. We utilise a semi-anonymous broadcast mechanism, one which can reveal or conceal the identity of the broadcaster both with reasonable (typically $\Omega(1)$) probability, and then we get a semi-anonymous leader election with similar guarantees.

We give a blockchain consensus protocol Sassafras utilising our leader election protocol SASLE for which it will be sufficient that there are more honest and anonymous leaders than malicious ones. We do not need the leader election to provide good anonymity for every honest leader to provide this and so our semi-anonymous protocol suffices. In a nutshell, our contributions are as follows:

- We define a single leader election functionality in $\mathcal{F}_{\mathsf{sle}}$ to define the security of SLE in the universal composable (UC) model. Our functionality $\mathcal{F}_{\mathsf{sle}}$ is adaptable to different unpredictability levels from Boneh et al.'s unpredictability [5] to no-predictability. Instead of covering the protocols where all parties are equal, $\mathcal{F}_{\mathsf{sle}}$ can be deployed by protocols where parties have different power. This is a useful property in PoS blockchain protocols where the amount of stake determines the chance of being a leader. In short, we give a definition of generic single leader election definition to be deployed by any SLE protocol with different unpredictability and equality levels.
- We construct a single leader election protocol that we call semi-anonymous single leader election (SASLE). We show SASLE is secure against adaptive adversaries in the UC model. Compared the existing SSLE protocols [5,14,15,19], SASLE has drastically small message complexity in the strongest adversarial model. It is semi-anonymous because we do not provide the same unpredictability level for all honest parties. This is the cost that we pay while we enjoy a significant gain in message sizes comparing to the existing protocols [5,14,15,19].
- We construct a PoS-protocol that we call Sassafras (Semi-anonymous sortition (of) staked assignees (for) fixed-time rhythmic assignment (of) slots). Sassafras elects the leaders by $\mathcal{F}_{\mathsf{sle}}$. We show that Sassafras is a secure blockchain protocol (has liveness and persistence) as long as $\mathcal{F}_{\mathsf{sle}}$ provides certain level of unpredictability, fairness and robustness. Our analysis shows that Sassafras can replace $\mathcal{F}_{\mathsf{sle}}$ with SASLE in the real world. Therefore, Sassafras is the first PoS-blockchain protocol which elects single leader and is realizable without huge costs on-chain thanks to the lightweight communication overhead of SASLE.

1.1 Related Works

Single Secret Leader Election (SSLE): Boneh et al. [5] formalize SSLE in the standard model. They provide definitions of the fairness, uniqueness, robustness and unpredictability. They propose a construction that works in three types of classes: Indistinguishability obfuscation (IO), learning with error (LWE), threshold fully homomorphic encryption (TFHE), decisional Diffie-Hellman (DDH) and random shuffles. Because of the inefficiency of IO in the real-world executions, it is not practical even though its communication complexity is $O(1)$ and its unpredictability level is $\frac{1}{n-m}$ where m is the number of malicious parties. TFHE's communication complexity is $O(t)$ and its unpredictability level is $\frac{1}{n-m}$ as long as $m < t$ where t is the threshold for TFHE. TFHE-based construction is more realizable than IO-based one but it is still not good for the practical constructions. The most practical version among the constructions by Boneh et al. [5] is based on DDH and random shuffles. It requires $O(n)$ communication for each election. It can be reduced to $O(\sqrt{n})$ but the unpredictability level reduces to $\frac{1}{\sqrt{n}-m}$ in this case. Ethereum proposes a SSLE protocol based on the DDH and random shuffles [5] that is called Whisk [19]. Catalano et al. [14] formalize the SSLE in the UC model. They propose a construction based on public-key encryption with key-word search (PEKS). Their round complexity is in the worst case is $O(\log^2 n) + 3$ in the worst case with $O(t)$-communication complexity. Its communication complexity is as good as TFHE-based construction and it has more practical realizations than TFHE. Catalano et al. [15] convert the construction of Boneh et al. [5] based on DDH and random shuffle into a new protocol which is adaptively secure in the UC model. [15] is the only SLE protocol which is secure against adaptive adversaries in the UC model except our protocol SASLE. They succeed to keep the communication cost the same as Boneh et al.'s version while providing a stronger security.

SASLE elects the leaders simultaneously for $\texttt{eplen} \geq 1$ slots instead of executing the election sequentially for each slot and it is based on ring verifiable random function (rVRF) [7] which has efficient realizations (see Appendix A). In our protocol, each party communicates with other parties with a drastically smaller message size compared to the shuffle solutions [5,15,14,19] as we discuss in Section 6. The small message size reduces the on-chain cost significantly for our blockchain protocol Sassafras which deploys SASLE. Although our protocol is better in terms of communication overhead, we sacrifice unpredictability level. If $\alpha = \frac{n-m}{n}$ is our ratio of honest parties ala Byzantine agreement, then in expectation $1-\alpha$ of our slots will not be anonymous in SASLE, while the rest have unpredictability level of close to $\frac{1}{\alpha n}$. We achieve these levels using our own simplistic optimized mix network, but alternative anonymous networking schemes could perhaps improve upon these. We caution however that anonymous network designs typically ignore collusion between senders and routers, which worsens liveness for Sassafras. We show that our protocol SASLE is secure in the UC model with an adaptive adversarial model.

PoS-protocols with Secret Leader Election: Most of the PoS-protocols [30,17,28,18,16] elect the leaders secretly and probabilistically for slots with a leader election protocol which *do not* guarantee that there will be only one leader. Sleepy consensus [30] requires a random oracle to elect a leader(s) which can be replaced by a common reference string and pseudo random functions. Snow White [17], the family of Ouroboros protocols [28,18,2], Algorand [16], Dfinity [22] execute the leader election based on a randomness beacon which is updated periodically. In those, each party checks whether they are elected as a leader with a random function where the randomness is provided with the randomness beacon. Different than probabilistic leader election (PLE), we suggest using SASLE in our new PoS-blockchain protocol Sasafrass. Thanks to SASLE, Sassafras selects only one leader in each round and the significantly many rounds

of Sassafras provides almost perfect secrecy for the leader. In our security analysis, we consider the rounds which do not provide secrecy for their honest leaders as malicious rounds. The downside of this is that the malicious stake ratio in Sassafras should be less than around 30% which is 50% in most of the PLE-based PoS protocols. Azouvi and Cappelleti [1] show that single secret leader election protocols provide higher security than the PLE protocols for private attacks. Specifically, they show that the persistence parameter (or common prefix parameter) in SSLE against private attacks in a synchronous network with a perfect randomness beacon is decreased by roughly 25% against a 33% or 25% malicious stake ratio. The same parameter decreases **%70** with a (not perfect) randomness beacon which is used in Sassafras and most of the other PoS blockchains [28,18,2,30,17,16]. Therefore, Sassafras benefits the higher security level and efficiency that SASLE provides (faster finality) by reducing persistence parameter if we compare it with PLE-based protocols with the malicious stake ratio less than 0.3. The trade-off here is that PLE based PoS protocols provides security with more adversarial power but they are significantly slower (70 %) to finalize the blocks, while Sassafras assumes less adversarial power but finalizes the blocks significantly faster.

1.2 Overview of the paper

In Section 2, we give the cryptographic primitives and security definitions that we need to build SASLE. Then, we introduce our adversarial model and define the security of SLE in the UC model in Section 3. After giving all buildings block, we describe our protocol SASLE and analyse its security in Section 4. Finally, we introduce our PoS-protocol Sassafras in Section 5 in the UC model and show how to realize it in the real world in Section 5.2 and Section 5.3. We conclude our paper by making more concrete comparison of the message complexity with the existing SSLE protocols in Section 6.

2 Preliminaries

Universally Composable (UC) Model: The UC model [8,9] consists of an ideal functionality as a trusted entity in the ideal world. A functionality behaves as a trusted party and defines the execution of a protocol in the ideal world. We informally call that a real-world execution of a protocol (without a trusted entity) in the real world is UC-secure if it is indistinguishable from the version of this protocol with the ideal functionality $\mathcal{F}$ in the ideal world. In UC, a protocol π is defined with distributed interactive Turing machines (ITM). Each ITM has an inbox collecting messages from other ITMs, adversary $\mathcal{A}$ or environment $\mathcal{Z}$. Whenever an ITM is activated by $\mathcal{Z}$, the ITM instance (ITI) is created. We identify ITIs with an identifier consisting of a session identifier sid and the ITM identifier pid. A party P in the UC model is an ITI with the identifier $(\mathsf{sid}, \mathsf{pid})$.
π in the Real World: $\mathcal{Z}$ initiates some ITM's of π and the adversary $\mathcal{A}$ to execute an instance of π with the input $z \in \{0,1\}^*$ and the security parameter κ. The output of a protocol execution in the real world is denoted by $\mathsf{EXEC}(\kappa, z)_{\pi,\mathcal{A},\mathcal{Z}} \in \{0,1\}$. Let $\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ denote the ensemble $\{\mathsf{EXEC}(\kappa, z)_{\pi,\mathcal{A},\mathcal{Z}}\}_{z\in\{0,1\}^*}$.
π in the Ideal World: The ideal world consists of an incorruptible ITM $\mathcal{F}$ which executes π in an ideal way. The adversary Sim (called simulator) in the ideal world has ITMs which forward all messages provided by $\mathcal{Z}$ to $\mathcal{F}$. The output of π in the ideal world is denoted by $\mathsf{EXEC}(\kappa, z)_{\mathcal{F},\mathsf{Sim},\mathcal{Z}} \in \{0,1\}$. Let $\mathsf{EXEC}_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}$ denote the ensemble $\{\mathsf{EXEC}(\kappa, z)_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}\}_{z\in\{0,1\}^*}$. We refer to [8,9] for further details about the UC-model.

Definition 1. *(UC-security of π) Let π be the real-world protocol and $\mathcal{F}$ be the ideal-world functionality of π. We say that π UC-realizes $\mathcal{F}$ (π is UC-secure) if for all PPT adversaries $\mathcal{A}$ in the real world, there exists a PPT simulator* Sim *such that for any environment $\mathcal{Z}$,*

$$\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{Z}} \approx \mathsf{EXEC}_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}$$

where $\approx$ shows that two ensembles are indistinguishable [8,9].

π in the Hybrid World: In the hybrid world, the parties in the real world interact with some ideal functionalities. We say that a protocol π in hybrid world UC-realizes $\mathcal{F}$ when π consists of some ideal functionalities.

SASLE consists of two basic primitives: State machine replication to agree on a linearizable log [1] and ring VRF [7] to provide anonymity to leaders during the election process. We review these primitives shortly.

State Machine Replication (SMR): We define SMR over a set of parties $\mathcal{P} = \{\mathsf{P}_1, \mathsf{P}_2, \ldots, \mathsf{P}_n\}$ initiated with sid. Each party maintains a replica of the state machine which is a deterministic programme receiving a set of inputs and outputs an ordered set LOG with respect to an interactive transition algorithm Transition. Transition takes as an input LOG and a list of *cmd*s fetched from the network (See Figure 1) and outputs an updated LOG with respect to *cmd*s. One can imagine LOG as a set of *cmd*s. We assume a setting where the time is divided into sequential time units [28,18,6,17]. Each party has a clock $\mathcal{F}_{\mathsf{clock}}$ [2,29,11,25] which shows the current slot and increases monotonically.

In our SMR that we define in Algorithm 1, each party fetches messages from the network when the current slot, learnt from $\mathcal{F}_{\mathsf{clock}}$, is updated. Then, each party updates their local ordered set LOG according to the new messages.

Algorithm 1 StateMachine(sid)

$\mathsf{LOG} = \emptyset$
$\mathtt{input} = \emptyset$
$\mathsf{sl} \leftarrow \mathcal{F}_{\mathsf{clock}}$ // learns the current slot sl from $\mathcal{F}_{\mathsf{clock}}$
while true **do**
    $\mathsf{sl}_c \leftarrow \mathcal{F}_{\mathsf{clock}}$
    if $\mathsf{sl}_c \geq \mathsf{sl}$ **then**
        fetch messages from the network (Figure 1)
        receive cmds from the network
        $\mathsf{sl} = \mathsf{sl}_c + 1$
        $\mathtt{input}$.Append(*cmds*)
        LOG = Transition(LOG, $\mathtt{input}$)

We call that $cmd \in \mathsf{LOG}$ is *final* if it is marked as final by the algorithm Transition. SMR is secure if it satisfies the following properties.

Definition 2 (Persistence). *Given a SMR defined over parties in $\mathcal{P}$ running Algorithm 1, persistence guarantees that if $cmd \in \mathsf{LOG}$ of an honest party's state machine is marked as final, then state machines of other honest parties mark cmd as final.*

Definition 3 (Liveness). *Given a SMR defined over parties in $\mathcal{P}$ running Algorithm 1, liveness guarantees that a command $cmd \in \mathsf{LOG}$ sent by an honest party becomes final in Δ_{live}-slots.*

[1] Linearizability means orderability of logs

Ring Verifiable Random Function: A ring VRF [7] (rVRF) is a ring signature which also lets parties output a random number with respect to the set of public keys (called ring) and a public key. In this sense, it operates like a VRF but only proves the random output comes from a specific list without giving any information about which key. We give the ring VRF functionality $\mathcal{F}_{\mathsf{rvrf}}$ in Figure 4 in Appendix A. In a nutshell, the functionality works as follows:

[Key Generation]: Given the message $(\mathit{keygen}, \mathsf{sid})$ from an honest party, $\mathcal{F}_{\mathsf{rvrf}}$ returns a verification key $\mathsf{pk}^{\mathsf{rvrf}}$ to the party.

[Honest rVRF Signature]: Given the message $(\mathit{sign}, \mathsf{sid}, \mathcal{PK}, \mathsf{pk}^{\mathsf{rvrf}}, m, \mathsf{ass})$, $\mathcal{F}_{\mathsf{rvrf}}$ returns a random anonymous key W, a signature σ, and a random output y for the message $m, \mathsf{pk}^{\mathsf{rvrf}}$ if $\mathsf{pk}^{\mathsf{rvrf}} \in \mathcal{PK}$. The functionality makes sure that $(\mathsf{pk}^{\mathsf{rvrf}}, m)$ is associated with one anonymous key and one random output. So, when a party comes with the same message m, a different or the same ring $\mathcal{PK}'$ where $\mathsf{pk}^{\mathsf{rvrf}} \in \mathcal{PK}'$, $\mathcal{F}_{\mathsf{rvrf}}$ outputs the same W and y but the signature σ may change. Whenever $\mathcal{F}_{\mathsf{rvrf}}$ generates a signature, it creates a record $[m, \mathsf{ass}, W, \sigma, 1]$ to record σ is a valid signature of m and ass signed by an anonymous key W. The parameter ass is used to bind the ring VRF signature to a context e.g., to prevent replay attacks. Also, σ serves as a signature of ass which we use this property to remove the necessity of key evolving signatures in Sassafras which deploys SASLE (See Section 5.2).

[Verification]: Given the message $(\mathit{verify}, \mathsf{sid}, \mathcal{PK}, W, m, \mathsf{ass}, \sigma)$, if σ is a valid signature i.e., there exists a record $[m, \mathsf{ass}, W, \sigma, 1]$, $\mathcal{F}_{\mathsf{rvrf}}$ returns 1 (verified) and the random output y that corresponds σ. Otherwise it returns 0. Verification mechanism makes sure that σ is not a forgery and it does not break the uniqueness property i.e., $(m, \mathcal{PK})$ generates at most $|\mathcal{PK}|$ verified random outputs.

Remark that the verification process does not need the public key of the signer in order to verify the signature and output its random output. Therefore, the verification process does not reveal the signer's verification key. This is the most subtle difference of ring VRFs and VRFs. If a signer ever wants to reveal its identity i.e., link a random output y to its verification key $\mathsf{pk}^{\mathsf{rvrf}}$, the signer should sign the same message with a ring $\mathcal{PK} = \{\mathsf{pk}^{\mathsf{rvrf}}\}$ and obtain σ. In this way, when another party verifies σ with $\mathcal{PK} = \{\mathsf{pk}^{\mathsf{rvrf}}\}$ and obtains y, he deduces that y is the random output of m and $\mathsf{pk}^{\mathsf{rvrf}}$.

In a nutshell, $\mathcal{F}_{\mathsf{rvrf}}$, when given as input a message m and a set $\mathcal{PK}$ of participant, allows to create $|\mathcal{PK}|$ possible random outputs that appear independent from the inputs. The output can be verified to have been computed correctly by one of the participants in $\mathcal{PK}$ with the ring signature without revealing who they are. The author of the ring signature can show that the ring signature was generated by them and no other participant could have done so.

We use an extension of $\mathcal{F}_{\mathsf{rvrf}}$, which is called $\mathcal{F}^{s}_{\mathsf{rvrf}}$, for the case where the chance of being a leader is not the same for all parties. $\mathcal{F}^{s}_{\mathsf{rvrf}}$ computes another random value that is secret. Any party can verify that the secret value of the message satisfies a certain relation without learning any clue about the secret value. $\mathcal{F}^{s}_{\mathsf{rvrf}}$ has the following procedures and a parameter $\mathcal{R}$ which is a relation:

[Secret random element proof]: Given a message $(\mathit{secret\_rand}, \mathsf{sid}, \mathcal{PK}, \mathsf{pk}^{\mathsf{rvrf}}, W, y, m)$, it verifies that y is a random output generated for $(\mathcal{PK}, W, m)$. Then it selects a secret random element η if it is not defined before for $(\mathcal{PK}, W, m)$. Then, it checks whether $((m, y, \mathcal{PK}), (\eta, \mathsf{pk}^{\mathsf{rvrf}})) \in \mathcal{R}$. If it is the case it generates π for $(\mathcal{PK}, W, m)$ as a proof and outputs η and π. Otherwise, it only outputs η.

[Secret Verification]: Given a message $(\mathit{secret\_verify}, \mathsf{sid}, \mathcal{PK}, W, m, \pi)$, it verifies if π is generated for $(\mathcal{PK}, W, m)$. Otherwise, it does not verify.

$\mathcal{F}^s_{\mathsf{rvrf}}$ operates similar to the non-interactive zero-knowledge (NIZK) functionality [21] except that the witness is a random element selected by $\mathcal{F}^s_{\mathsf{rvrf}}$. We note that if all parties are equal in SASLE, deploying $\mathcal{F}_{\mathsf{rvrf}}$ is enough.

3 Single Leader Election

We first introduce the adversarial and the communication model and then define SLE.

3.1 Adversarial and Communication Model

Adversarial Model: Our model consists of n ITIs called parties. We denote by $\mathcal{P}$ the set of parties. Each party $\mathsf{P}_i \in \mathcal{P}$ is run with the input st_i called *stake* given by $\mathcal{Z}$. $\mathcal{D}_S$ is the stake distribution of the protocol where $\mathcal{D}_S[\mathsf{P}_i] = \frac{\mathsf{st}_i}{\sum_{\mathsf{P}_j \in \mathcal{P}} \mathsf{st}_j}$. Stake of a party P_i defines the probability that P_i is the leader of a slot i.e., $\rho_i = \mathcal{D}_S[\mathsf{P}_i]$. We assume that all parties and functionalities have the security parameter λ even if we do not explicitly write it. We assume that $\mathcal{Z}$ can corrupt at most $(1-\alpha)$ fraction of the stakes owned by the parties at any time. When a party is corrupted, it means that the current state of the party is shared with the adversary. We consider another type of corruption that we call *weak corruption* to cover the denial of service attacks (DDOS) which is a critical attack in a distributed system. $\mathcal{Z}$ can weakly corrupt at most α_w-fraction of the honest stakes within a limited period Δ_{weak}-slots. A weakly corrupted honest party cannot send or receive any message during this period. We give the details related to weakly corruption in our communication functionality $\mathcal{F}_{\mathsf{com}}$ in Figure 1. Remark that if the leader of a slot is known, $\mathcal{Z}$ can weakly corrupt all honest leaders during their slots. This shows the importance of the secrecy of the leaders in our model. $\mathcal{Z}$ can update the honest stakes during the execution under the constraint that $\mathcal{D}_S[\mathsf{P}_i] \leq \rho_{\max}$ for all $\mathcal{P}$. The reason for this constraint is to obtain a distributed system where honest parties are almost equal according to the $\mathcal{Z}$'s view. Imagine the scenario for an election protocol where there is only one honest party with the stake ratio α. In this case, even if the leader election algorithm does not leak the identity of the leader, $\mathcal{Z}$ always knows the leader.

Communication Model: We define the functionality $\mathcal{F}_{\mathsf{com}}$ that covers our communication model is in Figure 1. In $\mathcal{F}_{\mathsf{com}}$, $\mathcal{Z}$ can weakly corrupt within Δ_{weak}-slots. After the weak corruption period, the party receives all missed messages during the weakly corruption period, but it is not guaranteed that the messages that she sent during this period are delivered. If a party is not weakly corrupted, its message arrives to other parties within Δ-slots.

We alternatively give a 'secure diffuse' option in $\mathcal{F}_{\mathsf{com}}$. If $\mathcal{F}_{\mathsf{com}}$ has this option, we call it $\mathcal{F}^s_{\mathsf{com}}$. It is useful if a sender wants to hide the message and the receiver. A sender gives the message and the receiver's name to $\mathcal{F}^s_{\mathsf{com}}$ and then $\mathcal{F}^s_{\mathsf{com}}$ sends the message to the receiver and sends some junk messages to others. $\mathcal{F}^s_{\mathsf{com}}$ does not leak neither the message nor the receiver to $\mathcal{A}$.

We note that there exist models to capture diffuse functionality [18,2,26] with a bounded delay. However, we consider a different security model where $\mathcal{Z}$ delays messages within a bound and also corrupts weakly a limited number of parties which can cause delay some messages much longer than the network delay bound Δ. In our model, we let $\mathcal{Z}$ weakly corrupt at most α_w-fraction of stakes at the same time.

$\mathcal{F}_{\mathsf{com}}$ creates an inbox $\mathcal{I}_i$, two queue lists $\mathcal{Q}_i^{\Delta}$ and $\mathcal{Q}_i^{W}$ for each instance $\mathsf{P}_i = (\mathsf{pid}_i, \mathsf{sid}) \in \mathcal{P}$. $\mathcal{I}_i$ consists of tuples of sender and message, $\mathcal{Q}_i^{\Delta}$ consists of tuples of sender, message and delay of the message and $\mathcal{Q}_i^{W}$ consists of tuples of sender, message and weakly corruption period of the sender. $\mathcal{F}_{\mathsf{com}}$ holds the list of weakly corrupted parties in the list $\mathcal{W}$ of the session sid.

Weak Corruption. upon receiving $(\mathit{weakly\_corrupt}, \mathsf{P}_i)$ from Sim, add (P_i, W_i) to the list $\mathcal{W}$ and remove P_i from $\mathcal{P}$. Otherwise, ignore the request.

Diffuse by Honest Party. upon receiving a message $(\mathit{diffuse}, \mathsf{sid}, m)$ from an honest party P_S, set $D = 0$ and add (P_S, m, D) to $\mathcal{Q}_i^{\Delta}$ of all $\mathsf{P}_i \in \mathcal{P}$ and add (P_S, m) to $\mathcal{Q}_i^{W}$ for all $(\mathsf{P}_i, .) \in \mathcal{W}$. Send (P_S, m) to Sim.

Diffuse by Weakly Corrupted Party. upon receiving a message $(\mathit{diffuse}, \mathsf{sid}, m)$ from a weakly corrupted party $(\mathsf{P}_S, .) \in \mathcal{W}$, send (P_S, m) to Sim.

Diffuse by the Adversary. upon receiving a message $(\mathit{diffuse}, \mathsf{sid}, m, \mathsf{P}_S)$ from Sim where P_S is not an honest party, send $(\mathit{deliver}, \mathsf{sid}, m, \mathsf{P}_S)$ to all $\mathsf{P}_i \in \mathcal{P}$.

[**Secure Diffuse by Honest Party.**] upon receiving a message $(\mathit{secure\_diffuse}, \mathsf{sid}, m, \mathsf{P}_j)$ from an honest party P_S, set $D = 0$,

- add (P_S, m, D) to $\mathcal{Q}_j^{\Delta}$ if $\mathsf{P}_j \notin \mathcal{W}$,
- add (P_S, m) to $\mathcal{Q}_j^{W}$ if $\mathsf{P}_j \in \mathcal{W}$,
- add $(\mathsf{P}_S, \phi_i \leftarrow\$ \{0,1\}^{|m|}, D)$ to $\mathcal{Q}_i^{\Delta}$ for all $\mathsf{P}_i \in \mathcal{P} \setminus \{\mathsf{P}_j\}$
- add $(\mathsf{P}_S, \phi_i \leftarrow\$ \{0,1\}^{|m|})$ to $\mathcal{Q}_i^{W}$ for all $(\mathsf{P}_i, .) \in \mathcal{W} \setminus \{\mathsf{P}_j\}$.

Finally, send $(\mathsf{P}_S, \mathsf{P}_i, |m|)$ to Sim.

Fetch Message. upon receiving $(\mathit{fetch}, \mathsf{sid})$ from a party P_i, do the following:

- if $(\mathsf{P}_i, .) \in \mathcal{W}$, obtain (P_i, W_i) from $\mathcal{W}$ and increment W_i. If $W_i > \Delta_{\mathsf{weak}}$ or if Sim replies with the message $(\mathit{no}, \mathsf{sid}, \mathsf{P}_s)$ as an answer to the message $(\mathit{is\_still\_weak}, \mathsf{sid}, \mathsf{P}_s)$, add all $(\mathsf{P}_s, m) \in \mathcal{Q}_i^{W}$ to $\mathcal{I}_i$, empty $\mathcal{Q}_i^{W}$, remove (P_i, W_i) from $\mathcal{W}$, and P_i to $\mathcal{P}$. Then, send $(\mathit{deliver}, \mathsf{sid}, \mathcal{I}_i)$ to P_i and empty $\mathcal{I}_i$. Otherwise, ignore the request.
- for each element (P_S, m_j, D_j) in $\mathcal{Q}_i^{\Delta}$, increment D_j. If $D_j = \Delta$, add (P_S, m_j) to the inbox $\mathcal{I}_i$ and remove (P_S, m_j, D_j) from $\mathcal{Q}_i^{\Delta}$. Otherwise, send $(\mathit{delayed\_messaged}, \mathsf{sid}, \mathcal{Q}_i^{\Delta})$ to Sim. In response, receive $(\mathit{delay}, \mathsf{sid}, \mathcal{M})$ where $\mathcal{M} \subseteq \mathcal{Q}_i^{\Delta}$ is the list of delayed messages. Then, for every $(\mathsf{P}_S, m_j, D_j) \in \mathcal{Q}_i^{\Delta} \setminus \mathcal{M}$, add (P_S, m_j) to $\mathcal{I}_i$ and remove (P_S, m_j, D_j) from $\mathcal{Q}_i^{\Delta}$. In the end, send $(\mathit{deliver}, \mathsf{sid}, \mathcal{I}_i)$ to P_i and empty $\mathcal{I}_i$.

Release Weak Corruption. upon receiving $(\mathit{weak\_release}, \mathsf{sid}, \mathsf{P}_i)$ from Sim, remove $(\mathsf{P}_i, .)$ from $\mathcal{W}$ and add P_i to $\mathcal{P}$. Then, add all $(\mathsf{P}_S, m) \in \mathcal{Q}_i^{W}$ as $(\mathsf{P}_S, m, \Delta - 1)$ to $\mathcal{Q}_i^{\Delta}$.

Fig. 1. Functionality $\mathcal{F}_{\mathsf{com}}$ without secure diffusion (the framed text) and $\mathcal{F}_{\mathsf{com}}^{s}$ with secure diffusion.

3.2 Security of Single Leader Election

In this section, we define our security model for single leader election (SLE). SLE is a protocol which selects one party as a leader with respect to a defined distribution among the parties that joined the election. If the election is secret [5], no one knows who is elected until the leader reveals himself. Boneh et al. [5] defined a single secret leader election (SSLE) protocol formally with the security properties uniqueness, unpredictability, fairness and robustness in the standard model. In a nutshell, uniqueness guarantees that exactly one party is elected for each election, unpredictability guarantees that the adversary can find out who is elected with the probability negligibly more than the random guessing, fairness guarantees that each party has a equal chance to be elected and finally, robustness guarantees that even if some malicious parties abort the election, one party is still selected.

We define the single leader election (SLE) protocol in the UC model with the functionality $\mathcal{F}_{\mathsf{sle}}$ in Figure 2. $\mathcal{F}_{\mathsf{sle}}$ gives the uniqueness property. However, its unpredictability and fairness properties depend on the definition of the algorithm ELECT run by $\mathcal{F}_{\mathsf{sle}}$. In this way, we aim to cover different types of SLE protocols with different security properties. $\mathcal{F}_{\mathsf{sle}}$ defined over a fix set of parties with parameters $\ell_{\min} \in \mathbb{N}$ minimum number of elections, $\texttt{max-elect} \in \mathbb{N}$ maximum number of rejection by Sim and $\mathcal{D}_S$ probability distribution defining the probability that a party $\mathsf{P} \in \mathcal{P}$ is elected as leader in an election i.e., $\mathcal{D}_S[\mathsf{P}_i] = \rho_i$. It works as follows:

Leader Election: In this phase, $\mathcal{F}_{\mathsf{sle}}$ provides some candidate-elections to Sim up to $\texttt{max-elect}$-times and Sim decides one of them. This phase starts just after $\mathcal{F}_{\mathsf{sle}}$ receives a message for the election from all honest parties. $\mathcal{F}_{\mathsf{sle}}$ runs an algorithm ELECT which outputs three lists in order of leadership: $\texttt{Leads}, \texttt{Leak}$ and $\texttt{Clue}$ of size ℓ. The list $\texttt{Leads}$ stores the leaders of ℓ-elections i.e., $\texttt{Leads}[k] = \mathsf{P}_i$ where $1 \leq k \leq \ell$ is the k^{th} leader. The list $\texttt{Leak}$ stores an honest leader or $\perp$ meaning no information available i.e., if $\texttt{Leads}[k] = \mathsf{P}_i$ is an honest party, $\texttt{Leak}[k]$ is either P_i or $\perp$. Otherwise, $\texttt{Leak}[k] = \perp$. The list $\texttt{Clues}$ stores some clues about the honest leaders. i.e., if $\texttt{Leads}[k] = \mathsf{P}_i$ is an honest party, $\texttt{Clues}[k]$ is some clue. Otherwise, $\texttt{Clues}[k] = \perp$. The definition of clue depends on the definition of the algorithm ELECT. For example, the clue can be the number of times that an honest party is a leader within ℓ-elections. We define clue in Section 4.2 as positive integers. One can determine some slots that the party P_i is not a leader with them. After running ELECT, $\mathcal{F}_{\mathsf{sle}}$ gives the list $\texttt{Leads}_m$ which is generated by removing all honest parties in $\texttt{Leads}$ and an index set $\mathcal{I}_m$ such that $|\mathcal{I}_m| = |\texttt{Leads}_m|$ and $\texttt{Leads}_m[i] = \texttt{Leads}[i]$ for all $i \in \mathcal{I}_m$. If Sim does not agree, $\mathcal{F}_{\mathsf{sle}}$ reruns ELECT and repeats the same process till Sim agrees. This process repeats at most $\texttt{max-elect}$-times. If Sim agrees with one of the elections or $\mathcal{F}_{\mathsf{sle}}$ runs ELECT $\texttt{max-elect}$-times, $\mathcal{F}_{\mathsf{sle}}$ continues to the next phase.

Distribution: If Sim does not agree on any elections after $\texttt{max-elect}$-rerun, $\mathcal{F}_{\mathsf{sle}}$ considers the last run of ELECT is the election results. Otherwise, $\mathcal{F}_{\mathsf{sle}}$ considers the one selected by Sim as the election results. After setting up the election results, $\mathcal{F}_{\mathsf{sle}}$ gives the list $\texttt{Leak}$, $\texttt{Clue}$ and the list $\texttt{Leads}_m, \mathcal{I}_m$ to Sim. As a response, Sim sends a set $\mathcal{L}_{\mathsf{abort}} \subset \mathcal{I}_m$ meaning that Sim aborts the leadership in each index in $\mathcal{L}_{\mathsf{abort}}$. If $|\texttt{Leads}| - |\mathcal{L}_{\mathsf{abort}}| < \ell_{\min}$, $\mathcal{F}_{\mathsf{sle}}$ aborts because it does not succeed to obtain at least $\ell_{\min}$ leaders. Otherwise, it removes each party $\texttt{Leads}[i]$ from $\texttt{Leads}$ for all $i \in \mathcal{L}_{\mathsf{abort}}$. In the end, it sends the election results to the corresponding parties.

Reveal: When P wants to reveal its leadership for the i^{th} election, it informs to $\mathcal{F}_{\mathsf{sle}}$. $\mathcal{F}_{\mathsf{sle}}$ sends a message to other parties saying P is the leader in election i.

Corrupt: When Sim corrupts a party, $\mathcal{F}_{\mathsf{sle}}$ adds it to a list of corrupted parties.

We define next the security properties of $\mathcal{F}_{\mathsf{sle}}$.

Definition 4 (Fairness). *We call that $\mathcal{F}_{\mathsf{sle}}$ satisfies fairness if for all $\mathsf{P}_i \in \mathcal{P}$, $\Pr[\texttt{Leads}[k] = \mathsf{P}_i | k \in [1, |\texttt{Leads}|]] = \frac{\mathcal{D}_S[\mathsf{P}_i]}{\sum_{\mathsf{P}_j \in \mathcal{P}_h} \mathcal{D}_S[\mathsf{P}_j]}$ in the leader election phase.*

In short, fairness guarantees that the distribution of leadership follows the distribution of the stakes of the parties.

Definition 5 (Θ-Unpredictability). *We define a list $\texttt{Reveal}$ where $|\texttt{Reveal}| = |\texttt{Leads}|$ and $\texttt{Reveal}[k] = \texttt{Leads}[k]$ if $\texttt{Leads}[k]$ revealed its leadership at the election k.*

If for all $k \in [1, |\texttt{Leads}|]$ where $\texttt{Leak}[k] = \perp$ and $\texttt{Reveal}[k] = \perp$, $\Pr[\texttt{Leads}[k] = \mathsf{P}_i | \texttt{Leak}, \texttt{Clue}, \texttt{Reveal}] \leq \Theta$, we call that the k^{th} election is Θ-unpredictable in $\mathcal{F}_{\mathsf{sle}}$.

Θ-predictability indicates that the probability of successful leader-guess in one election cannot be greater than Θ if the election is not leaked or revealed. The perfect unpredictability level

$\mathcal{F}_{\mathsf{sle}}$ is defined with parameters $\ell_{\min}, \texttt{max-elect} \in \mathbb{N}$, the distribution $\mathcal{D}_S$, and the algorithm ELECT for the parties in set $\mathcal{P} = \{\mathsf{P}_i = (\mathsf{pid}_i, \mathsf{sid})\}$. $\mathcal{P}_{\mathsf{Sim}} \subseteq \mathcal{P}$ is the set of malicious parties.

Leader Election. upon receiving a message $(elect, \mathsf{sid})$ from all honest parties in $\mathcal{P}$, run the algorithm $\text{ELECT}(\mathcal{D}_S, \mathsf{param}, \lambda) \rightarrow \texttt{Leads}, \texttt{Clue}, \texttt{Leak}$. Then send $(candidate\text{-}election, \mathsf{sid}, \texttt{Leads}_m, \mathcal{I}_m)$ to Sim. Set $\mathsf{counter} = 1$, set the candidate-election list $\mathcal{C} = \emptyset$ and add $(\texttt{Leak}, \texttt{Clue}, \texttt{Leads})$ to $\mathcal{C}$. Then create a list $\texttt{history}[\mathsf{P}_i]$ for each P_i and add $(\mathsf{counter}, \{k : \texttt{Leads}[k] = \mathsf{P}_i, \mathsf{P}_i \in \mathcal{P}\})$ to $\texttt{history}[\mathsf{P}_i]$.
- Upon receiving the message $(reelect, \mathsf{sid})$ from Sim, if $\mathsf{counter} < \texttt{max-elect}$, run $\text{ELECT}(\mathcal{D}_S, \mathsf{param}, \lambda) \rightarrow \texttt{Leads}, \texttt{Clue}, \texttt{Leak}$ and set the election results $\texttt{Leak}, \texttt{Clue}, \texttt{Leads}$. Then send the message $(candidate\text{-}election, \mathsf{sid}, \texttt{Leads}_m, \mathcal{I}_m)$ to Sim, set $\mathsf{counter} = \mathsf{counter} + 1$ and add $(\texttt{Leak}, \texttt{Clue}, \texttt{Leads})$ to $\mathcal{C}$. Add $(\mathsf{counter}, \{\mathsf{sl} : \texttt{Leads}[\mathsf{sl}] = \mathsf{P}_i, \mathsf{P}_i \in \mathcal{P}\}$ to $\texttt{history}[\mathsf{P}_i])$. Otherwise, move to the next phase.
- Upon receiving the message $(set\_election, \mathsf{sid}, c)$ from Sim where $c \leq \mathsf{counter}$, set the election result $\texttt{Leak}, \texttt{Clue}, \texttt{Leads}$ generated for $\mathsf{counter} = c$. Move to the next phase.
- If Sim does not respond, move to the next phase.

Distribution. Send $(initial\text{-}election, \mathsf{sid}, \texttt{Leak}, \texttt{Clues}, \texttt{Leads}_m, \mathcal{I}_m)$ to Sim and receive $(aborted, \mathsf{sid}, \mathcal{L}_{\mathsf{abort}})$ from Sim where $\mathcal{L}_{\mathsf{abort}} \subseteq \mathcal{I}_m$. If $|\texttt{Leads}| - |\mathcal{L}_{\mathsf{abort}}| \leq \ell_{\min}$, abort. Otherwise, for all $k \in \mathcal{L}_{\mathsf{abort}}$, remove $\texttt{Leads}[k]$ from $\texttt{Leads}$. Send the message $(result, \mathsf{sid}, \{k : \texttt{Leads}[k] = \mathsf{P}_i, \mathsf{P}_i \in \mathcal{P}\})$ to each P_i.

Reveal. upon receiving a message $(reveal, \mathsf{sid}, \mathsf{P}_i, k)$ from an honest party P_i, if $\texttt{Leads}[k] = \mathsf{P}_i$ and distribution phase is completed, send $(reveal, \mathsf{sid}, \mathsf{P}_i, k)$ to all $\mathsf{P} \in \mathcal{P}$.

Corrupt. upon receiving a message $(corrupt, \mathsf{sid}, \mathsf{pid}_i)$, add P_i to $\mathcal{P}_{\mathsf{Sim}}$. Send $(corrupted, \mathsf{sid}, \mathsf{pid}_i, \{k : \texttt{Leads}[k] = \mathsf{P}_i, \mathsf{P}_i \in \mathcal{P}\}, \texttt{history}[\mathsf{P}_i])$ to Sim.

Fig. 2. Functionality $\mathcal{F}_{\mathsf{sle}}$.

is $\rho_{\max}$ meaning that the lists of `Leak` and `Clue` do not help for guessing so the best guess is the honest party that has the highest stake.

Definition 6 (Robustness). *The leader election mechanism of $\mathcal{F}_{\mathsf{sle}}$ is robust if $\mathcal{F}_{\mathsf{sle}}$ runs at least $\ell_{\min}$-elections with an overwhelming probability.*

As a result of this, $\mathcal{F}_{\mathsf{sle}}$ satisfies *uniqueness* if $\mathcal{F}_{\mathsf{sle}}$ is robust. Uniqueness means that $\mathcal{F}_{\mathsf{sle}}$ selects only one party for each election.

We remark that $\mathcal{F}_{\mathsf{sle}}$ provides uniqueness and robustness as it is defined in Boneh et al.'s security model. Unpredictability and fairness depend on the algorithm ELECT. One can define it with respect to the desired security level.

Definition 7 (Secure SLE Protocol). *A SLE protocol in the real world is secure if it realizes $\mathcal{F}_{\mathsf{sle}}$ defined in Figure 2.*

4 Semi-Anonymous Single Leader Election

We describe our semi-anonymous single leader election protocol SASLE. Then, we show that SASLE realizes $\mathcal{F}_{\mathsf{sle}}$ with Algorithm 2. We first give an overview about primitives on which SASLE is based.

Overview of SASLE: We assume that each party has individual probability of a being a slot leader ρ. When parties start to run SASLE, they first generate a random number together. They use this randomness to generate `max_attempts`-many tickets which include independent random numbers. Then they decide whether their ticket is winning according to their threshold defined with respect to ρ. After the decision, each party semi-anonymously outputs the winning tickets. In the end, all parties agree on all winning tickets and sort them in a descending order by looking at their randomness. The sorted winning tickets show that the party who generated

the ticket at the k^{th} order is the leader of the k^{th} election. This is the main idea behind SASLE but of course there are a few challenges in the protocol to keep the leaders secret until the leader reveals.

The first challenge is that tickets should be verifiable i.e., the ticket is generated with the agreed random number and the key of the party. Basically, this property can be satisfied by a verifiable random function (VRF) but it does not hide the ticket owner because verification requires the key of the ticket owner. Therefore, we need a ring VRF to generate tickets which shows that the ticket is generated with the random number and with one of the keys of the parties.

The second challenge is publishing the tickets. If a party publishes them itself, the ticket cannot be anonymous. So, it does not work. One solution could be using an anonymous broadcast channel so that the sender of the ticket cannot be followed but it is expensive to deploy. Therefore, we prefer to publish them semi-anonymously meaning that each party sends each ticket to a randomly selected another party that we call repeater by hiding the message and the repeater. Then, the repeater publishes it. Basically, if the repeater is malicious, the anonymity of the ticket is compromised. Otherwise, it is safe because the owner of the ticket is known by the honest repeater only. We note that the message and the repeater can be hidden if the sender encrypts the message with the key of the repeater and sends the ciphertext to all parties. The party who can decrypt it understands that he is the repeater.

The last challenge is the agreement on all winning tickets. For this, we build SASLE on top of a SMR which provides persistence and liveness. We note that existing SSLE protocols [5,14,19,15] require to agree on shuffle messages to obtain the same election result but they do not explain how to achieve it.

4.1 The Description of SASLE

SASLE works in the hybrid model $\mathcal{F}^s_{\mathsf{rvrf}}, \mathcal{F}^s_{\mathsf{com}}$ and $\mathcal{F}_{\mathsf{rand}}$ on top of a state machine replication. Each party registers to $\mathcal{F}^s_{\mathsf{rvrf}}$ and obtains $\mathsf{pk}^{\mathsf{rvrf}}$ before the election starts. We assume that each party learns the ring VRF public keys of other parties that join to the protocol. We let $\mathcal{PK} = \{\mathsf{pk}_i^{\mathsf{rvrf}}\}_{i=1}^n$ where n is the number of parties joining to the elections. SASLE consists of five phases.

Randomness Generation Phase: SASLE requires a randomness beacon to satisfy fairness (See Definition 4). The randomness beacon generates a verifiable random number r for the election which will be used in the next phases of the protocol. There are many such beacons in the literature based on publicly verifiable secret sharing scheme [28,13,31], verifiable delay functions [4] and verifiable random functions [18,2,16]. However, we describe this phase based on the randomness beacon of Ouroboros Praos [18] because of its efficient instantiation on a blockchain. It does not provide a perfectly uniform random number but it is still good enough to satisfy fairness. We describe the functionality $\mathcal{F}_{\mathsf{rand}}$ that provides such randomness beacon in Figure 3. $\mathcal{F}_{\mathsf{rand}}$ is simplified version of $\mathcal{F}_{RLB}$ given in [18]. It allows the adversary `max-reset`-times to reset the randomness.

During this phase, each party P_i sends the message $(\mathit{rand\_request}, \mathsf{sid}, e)$ and receives the randomness r_e generated for the epoch e. Then, they start the next phase. We note that e is a time label for which we select leaders with SASLE.

Preparation Phase: Each party P_i individually runs this phase to prepare their tickets to be used for the election. We have two parameters in this phase: The first one is `max_attempts` $\in \mathbb{N}$ which

The functionality $\mathcal{F}_{\mathsf{rand}}$ is parametrized with the number of allowed resets `max-reset`, the epoch e that is the period for which the randomness is generated.

[Epoch-Randomness Generation.] Sample r_e from $\{0,1\}^\lambda$ and send the message $(\mathit{rand\_leak}, \mathsf{sid}, e, r_e)$ to Sim. Set the $\mathsf{counter} = 0$ and the candidate set $\mathcal{C} = \emptyset$.

- Upon receiving the message $(\mathit{rand\_reset}, \mathsf{sid}, e)$, sample another randomness $r_e \leftarrow\$ \{0,1\}^\lambda$ if $\mathsf{counter} <$ `max-reset`. Send the message $(\mathit{rand\_leak}, \mathsf{sid}, e, r_e)$ to Sim. Set the $\mathsf{counter} = \mathsf{counter}+1$ and the candidate set $\mathcal{C} = \mathcal{C} \cup \{r_e\}$.
- Upon receiving the message $(\mathit{rand\_set}, \mathsf{sid}, e, r)$ from Sim, set $r_e = r$ if $r \in \mathcal{C}$.
- If no message is received from Sim, set the last element of $\mathcal{C}$ as the randomness r_e and continue to the next phase.

[Epoch-Randomness Request.] Upon receiving $(\mathit{rand\_request}, \mathsf{sid}, e)$ from a party P_i, send the message $(\mathit{epoch\_rand}, \mathsf{sid}, e, r_e)$ to P_i if the randomness is set.

Fig. 3. Functionality $\mathcal{F}_{\mathsf{rand}}$ for the randomness beacon.

is the total number of tickets that a party should generate and the second one is a threshold $c \in (0, 1]$.

Execution of this phase changes depending on the distribution $\mathcal{D}_S$. If each party has equal chance of winning i.e., $\rho_i = \rho_j = \frac{1}{|\mathcal{P}|}$ for all $\mathsf{P}_i, \mathsf{P}_j \in \mathcal{P}$, parties run this phase with $\mathcal{F}_{\mathsf{rvrf}}$. Otherwise, they run $\mathcal{F}^s_{\mathsf{rvrf}}$ which provides a way to obtain a secret and random output with respect to a relation $\mathcal{R}$ [7].

Each P_i first generates number of `max_attempts` ring VRF outputs with their signatures. For this, P_i sends the message $(\mathit{sign}, \mathsf{sid}, \mathcal{PK}, \mathsf{pk}^{\mathsf{rvrf}}, r_e||j||e, \mathsf{ass}_j)$ to $\mathcal{F}^s_{\mathsf{rvrf}}$ (resp. $\mathcal{F}_{\mathsf{rvrf}}$ if equal winning-case) for each $j \in [1, \texttt{max\_attempts}]$ and receives back $(\mathit{signature}, \mathsf{sid}, \mathcal{PK}, W_{ij}, r_e||j||e, y_{ij}, \sigma_{ij})$. Thus, P_i obtains an anonymous key W_{ij}, a random number $\{y_{ij}\}$ and their corresponding ring signature σ_{ij}. Here, the associated data ass_j can be $\perp$ but it can be defined meaningful according to the actual protocol that deploys SASLE (See Section 5.2). Therefore, it is not a parameter that is a necessary for security of SASLE.

- **[equal winning probabilities]:** If $y_{ij} \leq \tau_i = \frac{c\rho_i}{\max\{\rho_j\}_{j=1}^n} = c$, it means that the input $r_e||j||e$ passes the threshold so j is a winner. After comparing all random outputs with the threshold, P_i prepares a winning ticket $(r_e, j, e, W_{ij}, \sigma_{ij}, \mathsf{ass}_j)$ and adds it to a list $\mathcal{L}_i$ if j is the winner. Otherwise, it adds $(r_e, \perp, \perp, \perp, \perp, \perp)$ to $\mathcal{L}_i$ just for the sake of the anonymity i.e, to hide the number of winning tickets from the adversary.
- **[different winning probabilities]:** In this case, P_i needs to generate *another random value* to determine whether the input $r_e||j||e$ passes the threshold $\tau_i = \frac{c\rho_i}{\max\{\rho_j\}_{j=1}^n}$. So, it sends $(\mathit{secret\_rand}, \mathsf{sid}, \mathcal{PK}, \mathsf{pk}_i^{\mathsf{rvrf}}, W_{ij}, r_e||j||e)$ to $\mathcal{F}^s_{\mathsf{rvrf}}$ for the relation $\mathcal{R}$ defined in 1 for each $j \in [1, \texttt{max\_attempts}]$. If it obtains the secret output η_{ij} and its proof π_{ij} from $\mathcal{F}^s_{\mathsf{rvrf}}$ as a response, it means that the secret output of $r_e||j||e$ passes the threshold τ_i and the corresponding index j is a winner.

$$\mathcal{R} = \{((r_e||j||e, y_{ij}, \mathcal{PK}, \mathcal{D}), (\eta_{ij}, \mathsf{pk}_i^{\mathsf{rvrf}})) : \texttt{secrets}[W_{ij}, r_e||j||e] = \eta_{ij}, \\ \rho_i \leftarrow \mathcal{D}[\mathsf{pk}_i^{\mathsf{rvrf}}], \eta_{ij} < \frac{c\rho_i}{\max\{\{\rho_j\}_{j=1}^n\}}\} \tag{1}$$

In the end, P_i adds the winning ticket $(r_e, j, e, W_{ij}, \sigma_{ij}, \pi_{ij}, \mathsf{ass}_j)$ to a list $\mathcal{L}_i$ if y_{ij} is the winner. Otherwise, it adds $(r_e, \perp, \perp, \perp, \perp, \perp, \perp)$ just for anonymity.
We remind that the secret output is necessary in the case of having different winning probabilities in $\mathcal{D}$ because we need to show that a ticket is the winner without revealing the

threshold. If parties run this phase as in the equal-winning-probabilities case, we would not satisfy the anonymity of the ticket because the adversary could link the identity of a party with his threshold.

We remark that the number of random outputs that P_i obtain from $\mathcal{F}_{\mathsf{rvrf}}$ is exactly $\texttt{max\_attempts}$ for the messages $r_e||j||e$ where $j \in [1, \texttt{max\_attempts}]$ and for the ring $\mathcal{PK}$. This property comes from the uniqueness property of ring VRF.

Distribution Phase: In this phase, each party distributes the elements in their lists $\mathcal{L}_i$ to randomly selected other parties that we call repeater. In this way, we aim to hide the owner of each winning ticket by trusting the repeater who is going to publish it later. P_i selects a repeater P_ℓ randomly for each element m (either winner or not winner) in $\mathcal{L}_i$. Then, it sends each m to its repeater P_ℓ via $\mathcal{F}^s_{\mathsf{com}}$ by sending the message $(\textit{secure\_diffuse}, \mathsf{sid}, m, \mathsf{P}_\ell)$ to $\mathcal{F}^s_{\mathsf{com}}$. Remember that $\mathcal{F}^s_{\mathsf{com}}$ sends m to P_ℓ and also sends some junk messages to other parties. This is required to hide the repeater of P_i for the sake of unpredictability. In addition, the reason that P_i sends all elements in $\mathcal{L}_i$ either winner or not winner to a repeater is to hide the number of winning tickets.

Whenever a party P_ℓ receives a $(r_e, j, e, W_{ij}, \sigma_{ij}, \pi_{ij}, \mathsf{ass}_j)$ where $j, W_{ij}, \sigma_{ij}, \pi_{ij} \neq \perp$ (resp. $(r_e, j, e, W_{ij}, \sigma_{ij}, \mathsf{ass}_j)$ where $j, W_{ij}, \sigma_{ij} \neq \perp$ if equal stake) from a party P_i, it runs the algorithm VerifyPreparation which does the followings:

- checks whether r_e is the randomness generated for e via $\mathcal{F}_{\mathsf{rand}}$,
- checks whether $j < \texttt{max\_attempts}$,
- sends the message $(\textit{verify}, \mathsf{sid}, \mathcal{PK}, W_{ij}, r_e||j||e, \mathsf{ass}_j, \sigma_{ij})$ to $\mathcal{F}^s_{\mathsf{rvrf}}$ (resp. to $\mathcal{F}_{\mathsf{rvrf}}$ if equal-winning case) for the verification for message and verifies if it receives $(\textit{verified}, \mathsf{sid}, \mathcal{PK}, W_{ij}, r_e||j||e, \sigma_{ij}, y_{ij}, 1)$ from $\mathcal{F}^s_{\mathsf{rvrf}}$ (resp. $\mathcal{F}_{\mathsf{rvrf}}$),
- sends the message $(\textit{secret\_verify}, \mathsf{sid}, \mathcal{PK}, W_{ij}, r_e||j||e, \pi_{ij})$ to $\mathcal{F}^s_{\mathsf{rvrf}}$ to check y_{ij} is a winning ticket and verifies if it receives $(\textit{secret\_verified}, \mathsf{sid}, \mathcal{PK}, W_{ij}, r_e||j||e, \pi_{ij}, 1)$ (resp. checks if $y_{ij} < c$ if equal-winning case).

If all checks and verifications are successful, then it stores $(r_e, j, e, W_{ij}, \sigma_{ij}, \pi_{i,j}, \mathsf{ass}_j)$ (resp. $(r_e, j, e, W_{ij}, \sigma_{ij}, \mathsf{ass}_j)$ if equal stake) in its submission list for the epoch e.

Submission Phase: This phase ends in Δ_{sub}-slots. Whenever a repeater receives a valid ticket $(r_{e_{\mathsf{init}}}, j, e, W_{ij}, \sigma_{ij}, \pi_{ij}, \mathsf{ass}_j)$ (resp. $(r_{e_{\mathsf{init}}}, j, e, W_{ij}, \sigma_{ij}, \mathsf{ass}_j)$ if equal winning case), it sends it as a command to all state machines (See Algorithm 1) via $\mathcal{F}_{\mathsf{com}}$. The algorithm Transition of SMR works as follows given the input $(\mathsf{LOG}, (r_{e_{\mathsf{init}}}, j, e, W_{ij}, \sigma_{ij}, \pi_{ij}), \mathsf{ass}_j)$ where LOG is a set: It runs VerifyPreparation with the input $(r_{e_{\mathsf{init}}}, j, e, W_{ij}, \sigma_{ij}, \pi_{ij}, \mathsf{ass}_j)$ and if it verifies, it adds $(r_{e_{\mathsf{init}}}, j, e, W_{ij}, \sigma_{ij}, \pi_{ij}, \mathsf{ass}_j)$ to LOG. Otherwise it ignores.

If a party sees that a repeater does not distribute its tickets at latest $\Delta_{\mathsf{sub}} - \Delta_{\mathsf{weak}} - \Delta$ slots after the randomness phase, the party publishes them itself as it is the repeater even if it leaks the ownership. We remark that if a ticket is not published on time by its repeater, it means that the repeater is malicious. So, leaking the owner of the ticket is not a new information to the adversary.

Sortition Phase: After the submission phase, all parties wait Δ_{live}-slots to make sure that all election tickets of e are the finalized by the state machines. After tickets $(r_{e_{\mathsf{init}}}, j, e, W_{ij}, \sigma_{ij}, \pi_{ij}, \mathsf{ass}_j)$ (resp., $(r_{e_{\mathsf{init}}}, j, e, W_{ij}, \sigma_{ij}, \mathsf{ass}_j)$ if equal winning case) are marked as final by the state machines of parties, they sort the tickets with respect to sorting their y_{ij} values in a descending order. We denote the list of sorted tickets by $\texttt{Leader}$ i.e., $\texttt{Leader}[u] = (r_{e_{\mathsf{init}}}, j, e, W_{ij}, \sigma_{ij}, \pi_{ij}, \mathsf{ass}_j)$ (resp. $(r_{e_{\mathsf{init}}}, j, e, W_{ij}, \sigma_{ij}, \mathsf{ass}_j)$ if equal winning case) shows that the party with an anonymous key W_{ij} generated the ring signature σ_{ij} is the leader of the u^{th} election.

Underlying security of the state machine replication (liveness) guarantees that all parties have the same leaders array. Thanks to the anonymity provided by the ring VRF, they do not know who generated the tickets of slots except with the ones of which they were the repeater. Thanks to the zero-knowledge property given by $\mathcal{F}^s_{\mathsf{rvrf}}$, they do not learn any information about the threshold of the owner of the tickets as well.

If the repeater is malicious, the anonymity of a ticket is clearly compromised however as long as the malicious repeaters are bounded, we still achieve good unpredictibility level as we see in the security analysis.

Verification: Whenever a slot leader with the key $\mathsf{pk}_i^{\mathsf{rvrf}}$ wants to announce its leadership of k^{th} election, it contacts with $\mathcal{F}^s_{\mathsf{rvrf}}$ (resp. $\mathcal{F}_{\mathsf{rvrf}}$ if equal stake) by sending $(\mathit{sign}, \mathsf{sid}, \{\mathsf{pk}_i^{\mathsf{rvrf}}\}, W_{ij}, r_e||j||e, \mathsf{ass}_j)$ and obtains a signature $\hat{\sigma}_{ij}$. When a party receives $(\hat{\sigma}_{ij}, \mathsf{pk}^{\mathsf{rvrf}})$ from a leader of k^{th} election, it obtains $(r_{e_{\mathsf{init}}}, j, e, W_{ij}, \sigma_{ij}, \pi_{ij}, \mathsf{ass}_j)$ (resp. $(r_{e_{\mathsf{init}}}, j, e, W_{ij}, \sigma_{ij}, \mathsf{ass}_j)$) from $\texttt{Leader}[k]$ and sends the message $(\mathit{verify}, \mathsf{sid}, \{\mathsf{pk}_i^{\mathsf{rvrf}}\}, W_{ij}, r_e||i||e, \mathsf{ass}_j, \hat{\sigma}_{ij})$ and $(\mathit{verify}, \mathsf{sid}, \mathcal{PK}, W_{ij}, r_e||i||e, \mathsf{ass}_j, \sigma_{ij})$ to $\mathcal{F}^s_{\mathsf{rvrf}}$ ($\mathcal{F}_{\mathsf{rvrf}}$ if equal stake). If $\mathcal{F}_{\mathsf{rvrf}}$ outputs $(1, y)$ and $(1, y')$ where $y = y' = y_{ij}$, then the party verifies that $\sigma_{ij}, W_{ij}, y_{ij}$ are generated by $\mathsf{pk}^{\mathsf{rvrf}}$ so he is the leader.

4.2 Security Analysis of SASLE

We show that SASLE realizes the functionality $\mathcal{F}_{\mathsf{sle}}$ running the algorithm Elect (See Algorithm 2). We first analyse the fairness, unpredictability, robustness and uniqueness of $\mathcal{F}_{\mathsf{sle}}$ running the algorithm Elect (See Algorithm 2). Then we show that SASLE realizes $\mathcal{F}_{\mathsf{sle}}$ (See Definition 1).

$\mathcal{F}_{\mathsf{sle}}$ with Algorithm 2 The parameters of $\mathcal{F}_{\mathsf{sle}}$ is $\mathcal{D}_S, \texttt{max\_attempts} \in \mathbb{N}$, $c \in (0, 1]$, security parameter λ, the number of registered parties n, $\texttt{max-elect} = \texttt{max-reset}$ (defined in $\mathcal{F}_{\mathsf{rand}}$ in Figure 2), $\ell_{\min} = \texttt{eplen}$.

Given inputs $\mathcal{D}_S, \ell_{\max}, \mathsf{param} = (\texttt{max\_attempts}, c, \lambda)$, Elect first finds the ratio of malicious parties with respect to $\mathcal{D}_S$ (Line 1 in Algorithm 2). In Elect, each party P_i has a threshold value $\tau_i = c\frac{\mathcal{D}_S[\mathsf{P}_i]}{\max\{\mathcal{D}_S[\mathsf{P}_j]_{j=1}^n\}} = c\frac{\mathcal{D}_S[\mathsf{P}_i]}{\rho_{\max}}$. We note that τ_i proportional to the probability of being elected for a slot $\rho_i = \mathcal{D}_S[\mathsf{P}_i]$. Elect selects number of $\texttt{max\_attempts}$ random elements for each party and checks whether it is less than the threshold. If it is less than their threshold, the random value is added to a list $\mathcal{L}_i$ created for each party P_i. Beyond the random value, Elect also adds the corresponding attempt index of the random value to $\mathcal{L}_i$ (See Line 5-8 in Algorithm 2). After the creation of all lists, it shuffles randomly the union of lists which is $\mathcal{L}$ and obtains election results. If i^{th}-election ($\mathcal{L}[i]$) is assigned to an honest party, Elect decides whether leaking or not leaking the honest leader to the adversary by tossing a coin with probability $(1-\alpha)$ i.e., the coin outputs 1 with the probability $1-\alpha$. If the leader of i^{th}-election is decided to be leaked, it adds the leader to the list $\texttt{Leak}[i]$ (See Line 14-17 in Algorithm 2). In any case, it adds the attempt-index value in $\mathcal{L}$ to the list $\texttt{Clue}$. In the end, Elect outputs a list $\texttt{Leads}$ including the leaders of slots, a list $\texttt{Leak}$ including the some honest slot leaders to be leaked (i.e., deanonymized) and a list $\texttt{Clue}$ which includes indexes in $\{1, 2, \ldots, \texttt{max\_attempts}\}$.

Fairness: It is clear that Elect satisfies *fairness* because $\Pr[\texttt{Leads}[k] = \mathsf{P}_i | k \in [1, |\texttt{Leads}|]] = \rho$.

Θ-Unpredictability: Unpredictability is not as straightforward as fairness. The leaked slots are clearly predictable while other slots give maximum amount of unpredictability level (i.e., $\frac{\rho_i}{\alpha}$

Algorithm 2 ELECT($\mathcal{D}_S, \texttt{max\_attempts}, c, \lambda$)

$1 - \alpha = \sum_{\mathsf{P}_i \in \mathcal{P}_m}^{n} \mathcal{D}_S[\mathsf{P}_i]$
for $\mathsf{P}_i \in \mathcal{P}$ **do**
  $\tau_i \leftarrow \frac{c\mathcal{D}_S[\mathsf{P}_i]}{\max\{\mathcal{D}_S[\mathsf{P}_j]_{j=1}^{n}\}}$
  $\mathcal{L}_i \leftarrow \emptyset$
  for $k \in [1, \texttt{max\_attempts}]$ **do**
    $\omega_k \leftarrow\$ \{0,1\}^{\lambda}$
    if $\omega_k < \tau_i$ **then**
      add (P_i, k) **to** $\mathcal{L}_i$
$\mathcal{L} \leftarrow$ SHUFFLE$(\cup_{i=1}^{n} \mathcal{L}_i)$
for $k \in \{1, 2, \ldots, |\mathcal{L}|\}$ **do**
  retrieve (P_i, k) **from** $\mathcal{L}[k]$
  $\texttt{Leads}[k] \leftarrow \mathsf{P}_i$
  $\texttt{Clue}[k] \leftarrow k$
  if $\mathsf{P}_i \in \mathcal{P}_h$ **then**
    $b \leftarrow_{1-\alpha} \{0,1\}$
    if $b = 1$ **then**
      $\texttt{Leak}[k] \leftarrow \mathsf{P}_i$
    else: $\texttt{Leak}[k] \leftarrow \perp$
return Leads, Clue, Leak

for all $\mathsf{P}_i \in \mathcal{P}_H$) till some parties reveal their leadership. After reveals, the list Clue may give information to the adversary about who is *not* the leader in other elections. For example, if a party is selected for i^{th}-election and $k = \texttt{Clue}[\mathsf{sl}]$ and reveals its leadership, it implies that this party is not the leader of elections with the attempt-index k. The bigger max_attempts and smaller parameter c makes the values in Clue less useful for the adversary which wants to predict the leaders. We have a detailed analysis below.

Lemma 1. *For any $1 \geq \chi > c$, any non-leaked and non-revealed election s of an honest leader P_i, the probability that P_i is the leader of s^{th} election given* Leak, Clue, $\texttt{Leads}_m, \mathcal{I}_m$, *and revealed elections* Reveal *is at most $\frac{\rho_i}{\alpha(1-\chi)}$ except with the probability* max_attempts $\exp(-\alpha\chi(\ln(\chi/c) - 1)/\rho_{\max})$.

Proof. Let E be the event we are conditioning on, that the revealed leaderships are what they are. For an index k, let $\mathcal{S}_k$ be the set of honest parties who revealed and *do not* have a leadership with index k i.e., $\mathsf{P}_j \in \mathcal{S}_k$ if $\texttt{Leads}[s'] = \texttt{Reveal}[s'] = \mathsf{P}_j \in \mathcal{P}_H$, $\texttt{Clue}[s'] \neq k$ and $s' \neq s$ where $\mathcal{P}_H$ is the set of honest parties.. Given $\texttt{Leak}[s] = \perp$, $\texttt{Leads}[s] \in \mathcal{P}_H$ and $\texttt{Clue}[s] = k$, $\Pr[\texttt{Leads}[s] = \mathsf{P}_i | E] = 0$ if $\mathsf{P}_i \notin \mathcal{S}_k$ and $\Pr[\texttt{Leads}[s] = \mathsf{P}_i | E] = \frac{\rho_i}{\sum_{j \in \mathcal{S}_k} \rho_j}$ if $\mathsf{P}_i \in \mathcal{S}_k$. We thus need to lower bound $\sum_{j \in \mathcal{S}_k} \rho_j$.

Let $\mathcal{N}_k$ be the set of i for honest parties P_i that have $\omega_k > \tau_i$. These honest parties never reveals their leadership with index k and so $\mathcal{N}_k \subset \mathcal{S}_k$. We show that except with probability $\exp(-\frac{\alpha\chi(\ln(\chi/c)-1)}{\rho_{\max}})$ (not conditioned on E), for all indices k, we have $\sum_{\mathsf{P}_i \in \mathcal{N}_k} \rho_i \geq \alpha\chi$. From this, it follows that $\Pr[\texttt{Leads}[s] = \mathsf{P}_i | E] = \frac{\rho_i}{\sum_{\mathsf{P}_j \in S_k} \rho_j} \leq \frac{\rho_i}{\sum_{\mathsf{P}_j \in \mathcal{N}_k} \rho_j} \leq \frac{\rho_i}{\alpha(1-\chi)}$ as required.

We define a Bernoulli random variable $X_{i,k}$ which is 1 if $\omega_k < \tau_i$. So we have $\sum_{\mathsf{P}_i \in \mathcal{N}_k} \rho_i = \sum_{\mathsf{P}_i \in \mathcal{P}_H} \rho_i(1 - X_{i,k}) = \alpha - \sum_{\mathsf{P}_i \in \mathcal{P}_H} \rho_i X_{i,k}$. $X_{i,k}$ is 1 with probability $\frac{c\rho_i}{\rho_{\max}}$. Thus the expectation is this weighted sum, which we call μ, is $\mu = E[\sum_{\mathsf{P}_i \in P_H} \rho_i X_{i,k}] = \sum_{\mathsf{P}_i \in \mathcal{P}_H} \frac{c\rho_i^2}{\rho_{\max}}$. Note that $\mu = \sum_{\mathsf{P}_i \in \mathcal{P}_H} \frac{c\rho_i^2}{\rho_{\max}} \leq \sum_{\mathsf{P}_i \in \mathcal{P}_H} c\rho_i = \alpha c$. Note that also with our assumption $\chi > c$, this implies that $\mu < \alpha\chi$. Now, we apply the Chernoff bound, in the form of Lemma 8 to obtain $\Pr[\sum_{\mathsf{P}_i} \rho_i X_{i,k} \geq \alpha\chi] \leq \exp(\frac{-\mu - \alpha\chi(\ln(\alpha\chi/\mu)-1)}{\rho_{\max}})$. Using $\mu \leq \alpha c$ we get $\Pr[\sum_{\mathsf{P}_i \in \mathcal{P}_H} \rho_i X_{i,k} \geq \alpha\chi] \leq \exp(\frac{-\mu - \alpha\chi(\ln(\alpha\chi/\mu)-1)}{\rho_{\max}}) \leq \exp(\frac{-\alpha\chi(\ln(\alpha\chi/\mu)-1)}{\rho_{\max}}) \leq \exp(\frac{-\alpha\chi(\ln(\chi/c)-1)}{\rho_{\max}})$.

When $\sum_{\mathsf{P}_i \in \mathcal{P}_H} \rho_i X_{i,k} \leq \alpha\chi$, we have that $\sum_{\mathsf{P}_i \in \mathcal{S}_k} \rho_i \geq \sum_{\mathsf{P}_i \in \mathcal{N}_k} \rho_i \geq \alpha(1-\chi)$. For a non-malicious non-leaked slot sl of index k, the probability in the adversaries' view that an honest party P_i is the leader of sl is 0 if $\mathsf{P}_i \notin \mathcal{S}_k$ and $\frac{\rho_i}{\sum_{\mathsf{P}_j \in \mathcal{S}_k} \rho_j} \leq \frac{\rho_i}{\alpha(1-\chi)}$. For this property to hold for all non-leaked and non-revealed honest leaders, we need $\sum_{\mathsf{P}_i \in \mathcal{P}_H} \rho_i X_{i,k} \leq \alpha\chi$ to hold for all

indices k. For any one index, it holds except with probability $\exp(\frac{-\alpha\chi(\ln(\chi/c)-1)}{\rho_{\max}})$. By a union bound, it holds for all indices k with probability $\texttt{max\_attempts}\exp(\frac{-\alpha\chi(\ln(\chi/c)-1)}{\rho_{\max}})$.

Corollary 1. *For any $1 \geq \chi > c$, $\mathcal{F}_{\mathsf{sle}}$ with Algorithm 2 is $\frac{\rho_{\max}}{\alpha(1-\chi)}$-unpredictable except with probability* $\texttt{max\_attempts}\exp(-\frac{\alpha\chi(\ln(\chi/c)-1)}{\rho_{\max}})$.

Corollary 1 shows that if $1 \geq \chi \geq c$, the list `Clue` is not very helpful to the adversary to predict the slot leader.

Robustness: We show that $\mathcal{F}_{\mathsf{sle}}$ outputs at least `eplen` election results except with a negligible probability. Remark that it happens only if $|\texttt{Leads}| - |\mathcal{L}_{\mathsf{abort}}| \geq \texttt{eplen}$.

Lemma 2. *$\mathcal{F}_{\mathsf{sle}}$ with* Elect *described in Algorithm 2 is robust except with the probability* $\exp(-\frac{(r-1)^2}{2r}\texttt{eplen})$ *where* $r = \frac{c\texttt{max\_attempts}\alpha}{\rho_{\max}\texttt{eplen}}$ *as long as $r > 1$.*

Proof. We need to upper bound $|\cup_{\mathsf{P}_i\in\mathcal{P}_H}\mathcal{L}_i| = |\{\omega_{i,k} < \tau_i\} : \mathsf{P}_i \in \mathcal{P}_H|$. Let's define a random variable $X_{i,k}$ to for $\mathsf{P}_i \in \mathcal{P}_h$ and $1 \leq k \leq \texttt{max\_attempts}$. $X_{i,k}$ is 1 if $\omega_{i,k} < \tau_i$. So, $\Pr[X_{i,k} = 1] = \Pr[\omega_k < \tau_i] = \frac{c\rho_i}{\rho_{\max}}$. The expected value of $\sum_{i,k} X_{i,k} = \mu = \frac{\alpha c\texttt{max\_attempts}}{\rho_{\max}}$. From Chernoff bound, for all $\delta \in (0,1)$, $\Pr\left[|\cup_{\mathsf{P}_i\in\mathcal{P}_H}\mathcal{L}_i| = \sum_{i,k} X_{i,k} \leq (1-\delta)\mu\right] \leq \exp(-\frac{\delta^2\mu}{2})$ for any $\delta \in (0,1)$. Therefore, $|\cup_{\mathsf{P}_i\in\mathcal{P}_H}\mathcal{L}_i| > (1-\delta)\mu$ except with the probability $\exp(-\frac{\delta^2\mu}{2})$. Since $\mu = \frac{c\rho_i}{\rho_{\max}} = r\texttt{eplen}$ for $r > 1$, we set $\delta = 1 - 1/r$ and obtain $(1-\delta)\mu = \texttt{eplen}$. Thus we have $|\cup_{\mathsf{P}_i\in\mathcal{P}_H}\mathcal{L}_i| > \texttt{eplen}$ except with the probability $\exp(-\frac{\delta^2\mu}{2}) = \exp(-\frac{(1-1/r)^2}{2}\mu) = \exp(-\frac{(r-1)^2}{2r^2}\mu) = \exp(-\frac{(r-1)^2}{2r}\texttt{eplen})$.

Uniqueness: Lemma 2 shows that Elect elects only *one* party for each election. Therefore, $\mathcal{F}_{\mathsf{sle}}$ with Elect satisfies *uniqueness.*

SASLE realizes $\mathcal{F}_{\mathsf{sle}}$ with Algorithm 2

Theorem 1. *Assuming that the underlying state machine replication has persistence and liveness with the parameter Δ_{live}, SASLE realizes $\mathcal{F}_{\mathsf{sle}}$ with the Algorithm 2 in the $\mathcal{F}^s_{\mathsf{rvrf}}, \mathcal{F}_{\mathsf{rand}}$ and $\mathcal{F}^s_{\mathsf{com}}$-hybrid model.*

Proof. We construct a simulator Sim in the $\mathcal{F}^s_{\mathsf{com}}, \mathcal{F}_{\mathsf{rand}}, \mathcal{F}^s_{\mathsf{rvrf}}$-hybrid model. $\mathcal{F}_{\mathsf{sle}}$ in the ideal world runs the election for the parties $\mathcal{P} = \{\mathsf{P}_i = (\mathsf{sid}, \mathsf{pid}_i)\}_{i=1}^n$. In the real world, Sim registers the honest parties in $\mathcal{P}$ via $\mathcal{F}^s_{\mathsf{rvrf}}$. Sim starts to simulate the real protocol as soon as all parties register to $\mathcal{F}^s_{\mathsf{rvrf}}$ and agree on the ring VRF public keys $\mathcal{PK} = \{\mathsf{pk}^{\mathsf{rvrf}}\}_{i=1}^n$.
[**Simulation of leader election:**] When $\mathcal{F}_{\mathsf{sle}}$ sends $(\mathit{candidate\_election}, \mathsf{sid}, \texttt{Leads}_m, \mathcal{I}_m)$, Sim learns which malicious party is assigned to which slots for the first candidate election and simulation starts.
[**Simulation of the randomness phase:**] Sim simulates $\mathcal{F}_{\mathsf{rand}}$ as described in Figure 3. Whenever Sim sends the message $(\mathit{rand\_leak}, \mathsf{sid}, e, r_e)$ to the adversary while simulating $\mathcal{F}_{\mathsf{rand}}$, it sets up (secret) random outputs that are supposed to be generated by $\mathcal{F}^s_{\mathsf{rvrf}}$ with a given randomness of the epoch so that the election result in the real-world gives a consistent result with $\texttt{Leads}_m$ as described below. Whenever Sim receives a message $(\mathit{rand\_reset}, \mathsf{sid}, e)$, it sends $(\mathit{reelect}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{sle}}$ and receives new election results $\texttt{Leads}_m$ from $\mathcal{F}_{\mathsf{sle}}$.

When the adversary sends the message $(\textit{rand\_set}, \mathsf{sid}, e, k_{set})$, Sim finalizes the randomness of e as r_e which is the randomness generated when $\mathsf{counter} = k_{set}$ as $\mathcal{F}_{\mathsf{rand}}$ does. Then, Sim sends the message $(\textit{election\_set}, \mathsf{sid}, k_{set})$ to $\mathcal{F}_{\mathsf{sle}}$ to finalize the election in the ideal world as well. Then, it receives the message $(\textit{initial\_election}, \mathsf{sid}, \mathtt{Leak}, \mathtt{Clue}, \mathtt{Leads}_m)$ from $\mathcal{F}_{\mathsf{sle}}$.

[Setting up (secret) random outputs of $\mathcal{F}^s_{\mathsf{rvrf}}$ before *rand_leak*:] Sim selects randomly $y_1 < y_2 < \ldots < y_{\ell_{\max}}$ where $\ell_{\max} = n\mathtt{max\_attempts}$ after $\mathcal{F}_{\mathsf{sle}}$ sends $(\textit{candidate\_election}, \mathsf{sid}, \mathtt{Leads}_m, \mathtt{Clue}_m)$. It stores $y_1, y_2, \ldots, y_{\ell_{\max}}$ to a history list $\mathtt{log}_r[\mathsf{counter}]$ where $\mathsf{counter}$ is the current counter of $\mathcal{F}_{\mathsf{rand}}$. This information will be useful when $\mathcal{A}$ corrupts an honest party during the simulation. Next, Sim executes the step that we call the Malicious rVRF evaluation preperation with the input r_e that it selected while simulating $\mathcal{F}_{\mathsf{rand}}$ that it selected while simulating $\mathcal{F}_{\mathsf{rand}}$:

[Malicious rVRF evaluation preperation:] Given r_e, Sim obtains r_e that it selected while simulating $\mathcal{F}_{\mathsf{rand}}$ and creates a list $\mathtt{Out}_{\mathcal{A}}$ to later populate $\mathtt{Out}$ of $\mathcal{F}^s_{\mathsf{rvrf}}$. Given the list $\mathcal{I}_{\mathsf{P}_i}$ of elections that a malicious party P_i is selected as a leader, Sim behaves as follows: Sim assigns randomly a unique index between $[1, \mathtt{max\_attempts}]$ for each election in $\mathcal{I}_{\mathsf{P}_i}$. We note that it is possible to assign a unique index in $[1, \mathtt{max\_attempts}]$ to each value in $\mathcal{I}_{\mathsf{P}_i}$ because $|\mathcal{I}_{\mathsf{P}_i}| \leq \mathtt{max\_attempts}$ as it can be seen in Lines 5-8 in Algorithm 2. Let's denote the corresponding list by $\mathcal{J}_{\mathsf{P}_i}$. For each $i \leq \mathcal{I}_{\mathsf{P}_i}$, Sim gets $j \leftarrow \mathcal{J}_{\mathsf{P}_i}$, and sets $\mathtt{Out}_{\mathcal{A}}[r_e||j||e, \mathsf{pk}_i^{\mathsf{rvrf}}] := y_{\mathsf{sl}}$. If there exists W such that $\mathtt{anonymous\_key\_map}[r_e||j||e, W] = \mathsf{pk}_i^{\mathsf{rvrf}}$, then Sim outputs an abort message Abort-Collision and ends the simulation.

[Simulating $\mathcal{F}^s_{\mathsf{rvrf}}$:] As soon as receiving $(\textit{initial\_election}, \mathsf{sid}, \mathtt{Leak}, \mathtt{Clue}, \mathtt{Leads}_m)$ from $\mathcal{F}_{\mathsf{sle}}$, $\mathcal{F}^s_{\mathsf{rvrf}}$ does the following: For each $k \leq \ell_{\max}$ such that $\mathtt{Leak}[k] \neq \perp$, Sim gets $j \leftarrow \mathtt{Clue}[k]$, $\mathsf{P}_i \leftarrow \mathtt{Leak}[k]$, then selects randomly W from $\mathcal{S}_W$ as $\mathcal{F}^s_{\mathsf{rvrf}}$ does. Then, it sets $\mathtt{anonymous\_key\_map}[r_e||j||e, W] := \mathsf{pk}_i^{\mathsf{rvrf}}$ and $\mathtt{Out}[r_e||j||e, W] := y_k$ where y_k is obtained from $\mathtt{log}_r[k_{set}]$. It also samples randomly a secret random output $\eta < \tau_i$ and sets $\mathtt{secrets}[r_e||j||e, W] := \eta$. If $\mathtt{Out}[r_e||j||e, W]$ or $\mathtt{secrets}[r_e||j||e, W]$ has been already defined with another value, Sim outputs an abort message Abort-Collision and ends the simulation.

For each $k \leq \ell_{\max}$ such that $\mathtt{Leak}[k] = \perp$, Sim gets $j \leftarrow \mathtt{Clue}[k]$, then samples $W \leftarrow_{\$} \mathcal{S}_W$ and runs $\mathsf{Gen}_{sign}(\mathcal{PK}, W, r_e||j||e)$. It stores $[r_e||j||e, W, \mathcal{PK}, \sigma, 1]$ since it is a valid signature. It obtains a proof $\pi \leftarrow \mathsf{Gen}_{\pi}(\mathcal{PK}, W, m)$. Then, it sets $\mathtt{Out}[r_e||j||e, W, \mathcal{PK}] := y_k$, adds π to $\mathtt{zkproofs}[r_e||j||e, W]$ and records $[r_e||j||e, W, \mathcal{PK}, \sigma, 1]$ in $\mathcal{F}_{\mathsf{rvrf}}$'s database. Since $\mathcal{F}^s_{\mathsf{rvrf}}$ does not know the leader of the k^{th} election, it does not know the threshold of the leader. Therefore, it makes sure that there exists a valid proof even if it does not generate a secret random output for it. Since secret random output is never revealed, this part of the simulation is indistinguishable. It also stores W in $\mathtt{W}[k]$ and σ in $\mathtt{signature}[k]$. If $\mathtt{Out}[r_e||j||e, W]$ has been already defined with another value, Sim outputs an abort message Abort-Collision and ends the simulation. Remark that Sim cannot set $\mathtt{anonymous\_key\_map}[r_e||j||e, W]$ and $\mathtt{secrets}$ at this point because honest leader of k^{th} election is not known yet.

Sim simulates honest ring VRF signature, secret random element proof and secret verification as described in the functionality. Whenever $\mathcal{F}^s_{\mathsf{rvrf}}$ needs to randomly select an evaluation value for $(m, \mathsf{pk}_i^{\mathsf{rvrf}})$, it checks if $\mathtt{Out}_{\mathcal{A}}[m, \mathsf{pk}_i^{\mathsf{rvrf}}]$ is defined before. If it is defined, it sets $\mathtt{Out}[m, W] = \mathtt{Out}_{\mathcal{A}}[m, \mathsf{pk}_i^{\mathsf{rvrf}}]$. It picks $\eta_{ij} \leq \tau_i$ and sets $\mathtt{secrets}[m, W] = \eta_{ij}$. Otherwise, it behaves as defined $\mathcal{F}^s_{\mathsf{rvrf}}$. Thus $\mathcal{A}$ has random outputs which corresponds to the election result.

[Simulation of the preparation phase:] Sim simulates the leaked slots as follows: We denote the corresponding y-values of leaked slots of an honest party P_i by set $\mathcal{Y}_i = \{y_{i_1}, y_{i_2}, \ldots, y_{i_t}\}$ where $\mathtt{Leak}[i_k] = \mathsf{P}_i$ and $i_k \in \{1, \mathsf{sl}_2, \ldots, \mathsf{sl}_\ell\}$ for all $k \in [1, t]$. For each $\mathsf{P}_i \in \mathcal{P}_h$, Sim obtains σ_{i_k} and its anonymous key W_{i_k} by sending the message $(\textit{sign}, \mathsf{sid}, \mathcal{PK}, \mathsf{pk}_i^{\mathsf{rvrf}}, r_e||j||e)$ to $\mathcal{F}^s_{\mathsf{rvrf}}$ where

$j \leftarrow \texttt{Clue}[i_k]$. Similarly, it obtains π_{i_k} by sending the message $(\textit{secret\_rand}, \mathsf{sid}, \mathcal{PK}, W_{i_k}, r_e||j||e)$ to $\mathcal{F}^s_{\mathsf{rvrf}}$ where $j \leftarrow \texttt{Clue}[i_k]$. Remark that Sim already made sure that $\texttt{secrets}[r_e||j||e, W_{i_k}] < \tau_i$ in $\mathcal{F}^s_{\mathsf{rvrf}}$ for the leaked slots so π_{i_k} is not $\perp$. In the end, Sim constructs a list $\mathcal{L}_i$ for each honest party P_i such that $\mathcal{L}_i = \{(r_e, j, e, W_{i_k}, \sigma_{i_k}, \pi_{i_k})\}$ similar to the preparation phase of SASLE. Remark that $\mathcal{L}_i$ includes only the leaked tickets differently than SASLE but it is indistinguishable because $\mathcal{A}$ does not learn the non-leaked tickets at this phase.

Sim also prepares a list of tickets for the non-leaked slots. It creates a special list $\mathcal{L}^*$ which represents the union of subsets of $\mathcal{L}_i$ in SASLE which includes tickets that are not sent to $\mathcal{A}$. For each $\mathsf{sl} \leq \ell_{\max}$ such that $\texttt{Leak}[k] = \perp$, it gets $W = \texttt{W}[k], \sigma = \texttt{signature}[k]$ and $\pi = \texttt{zkproofs}[m, W]$. In the end, it adds the ticket $(r_e, j, e, W, \sigma, \pi)$ to $\mathcal{L}^*$ where $j = \texttt{Clue}[k]$.

[Simulation of the distribution phase:] Here, Sim only knows the owner of an honest party's y value if its slot is leaked. However, it is not a problem because the honest leaders of a non-leaked slots cannot be known by the adversary. Sim distributes the leaked slots among the malicious parties. In more detail, Sim does the following for each honest P_i: For each element in $\mathcal{L}_i$, it picks randomly a *malicious* party P_j and sends the element to P_j via $\mathcal{F}^s_{\mathsf{com}}$. Remark that an honest party sends a message using $\mathcal{F}^s_{\mathsf{com}}$ in the distribution phase $\texttt{max\_attempts}$-times. In order to simulate this, Sim sends some random messages from $\{0,1\}^{|m|}$ $\texttt{max\_attempts} - |\mathcal{L}_i|$-times to each party as if it comes from $\mathcal{F}^s_{\mathsf{com}}$.

[Simulation of the submission phase:] During the submission, Sim behaves exactly the same as an honest party in SASLE. Sim submits the tickets together with their signatures and proofs on behalf of the honest party who receives them from a malicious party. For each anonymous ticket in $\mathcal{L}^*$, Sim picks randomly an honest party and submits the ticket on behalf of this honest party.

In the end of the sortition phase, if there exits an honest ticket which is not finalized, Sim outputs the message ABORT-MISSING and the simulation ends. Otherwise, Sim checks which malicious tickets are not in LOG of SMR in order to find aborted slots. If y_{sl} is not obtained from the verification of one of the malicious ring signatures in LOG and $\texttt{Leads}_m[k] = \mathsf{P}_i \in \mathcal{P}_m$, Sim adds k to the aborted slot list $\mathcal{S}_{\mathsf{abort}}$. It sends $(\textit{aborted\_slots}, \mathsf{sid}, \mathcal{S}_{\mathsf{abort}})$ to $\mathcal{F}_{\mathsf{sle}}$. After Sim sends the aborted slots to $\mathcal{F}_{\mathsf{sle}}$, if the sortition of tickets changes in LOG and is finalized, then Sim outputs ABORT-INCONSISTENCY and the simulation ends.

[Simulation of revealing:] When $\mathcal{F}_{\mathsf{sle}}$ sends $(\textit{reveal}, \mathsf{sid}, k, \mathsf{P}_i)$, Sim sends $(\textit{sign}, \mathsf{sid}, \{\mathsf{pk}_i^{\mathsf{rvrf}}\}, W, r_e||j||e)$ to $\mathcal{F}^s_{\mathsf{rvrf}}$ where $j \leftarrow \texttt{Clue}[k]$ and $r_e||j||e$ is from the k^{th} sorted ticket. Then it obtains $\hat{\sigma}$. Then, it publishes it via $\mathcal{F}^s_{\mathsf{com}}$

[Simulation of Corruption:] When Sim receives a message $(\textit{corrupted}, \mathsf{sid}, \mathsf{pid}_i, \texttt{history}[\mathsf{P}_i])$, Sim adjusts the ring VRF outputs so that the election result and the candidate elections match with the real world election. Remember that $\texttt{history}[\mathsf{P}_i]$ consists of the tuples $(\mathsf{counter}, \{j : \texttt{Slot}[j] = \mathsf{P}_i\})$ i.e., the slots that P_i is selected as a leader in the election number $\mathsf{counter}$. For each election number $\mathsf{counter} \in \texttt{history}[\mathsf{P}_i]$, Sim obtains $\texttt{log}_r[\mathsf{counter}]$ that it stored during the setting up (secret) random outputs, $r = \mathcal{C}[\mathsf{counter}]$ and the list of slots $\mathcal{I}_{\mathsf{P}_i}$ (i.e., the second component of $(\mathsf{counter}, \{j : \texttt{Slot}[j] = \mathsf{P}_i\})$ in $\texttt{history}[\mathsf{P}_i]$) that P_i is selected in $\mathsf{counter}^{th}$ reelection in $\mathcal{F}_{\mathsf{sle}}$. In the end, it executes the same steps in the malicious rVRF evaluation preparation with the randomness r and $\mathcal{I}_{\mathsf{P}_i}$.

Claim. The view of $\mathcal{Z}$ in its interaction with Sim is indistinguishable from the view in its interaction with the real-honest parties.

Proof. We first analyse the indistinguishability of simulation of functionalities. The simulations of $\mathcal{F}_{\mathsf{rand}}, \mathcal{F}^s_{\mathsf{com}}$ are the same. Sim only simulates differently $\mathcal{F}^s_{\mathsf{rvrf}}$. It differently outputs

ABORT-COLLISION when a given input $(\mathcal{PK}, r_e||j||e)$ is assigned to a value in `Out` and `secrets` for a party P_i. Since r_e is learnt by the adversary after e is started and it is random, the probability that any input such as $(\mathcal{PK}, r_e||j||e)$ is queried to $\mathcal{F}^s_{\mathsf{rvrf}}$ before e starts is $\mathsf{negl}(\lambda)$.

Next we analyse the indistinguishability of the simulation of honest parties. First difference is that $\mathcal{L}_i$ generated during the simulation does not include all winning tickets of P_i as in the real protocol. It does not change the view of the $\mathcal{A}$ because $\mathcal{A}$ only learns the leaked tickets in $\mathcal{L}_i$ in the real protocol. Given that the number of leaked tickets are ξ, $\mathcal{A}$ also receives n_h`max_attempts`$-\xi$-junk messages from $\mathcal{F}^s_{\mathsf{com}}$ in the real protocol. Since Sim sends `max_attempts`$-|\mathcal{L}_i|$ junk messages for each honest party P_i beyond the leaked tickets, the simulation of honest parties during the distribution phase is indistinguishable. Second difference is in the sortition phase. If a y-value is missing in the in the end of the sortition phase, Sim outputs ABORT-MISSING. Since sortition period takes Δ_{live}-slots, all honest tickets are going to be final in Δ_{live}-slots. The other abort message that Sim outputs during the sortition phase is ABORT-INCONSISTENCY. Since all honest tickets are final in the end of Δ_{live}-slots, they are considered as final for all honest parties and cannot be changed.

Claim. The distributions of the outputs of Sim and $\mathcal{A}$ are indistinguishable.

Proof. The outputs of ELECT in Algorithm 2 and the election process in SASLE are in the same distribution. Therefore, the election results in both world matches as long as the Sim does not output any abort message which happens with negligible probability as shown in the above claim.

5 Sassafras

In this section, we describe our UC-secure proof-of-stake protocol Sassafras which uses SASLE as a leader election protocol. For the sake of generality, we describe Sassafras in the $\mathcal{F}_{\mathsf{sle}}$ which can be replaced by SASLE as shown in Theorem 1. We show in Section 5.2 how to instantiate Sassafras in the real-world.

Sassafras consists of sequential long time intervals epochs $(e_1, e_2, \ldots, e_{\mathfrak{L}})$ where $e_i \in \mathbb{N}$, each of which consists a number of sequential block production slots $(e_i = \{\mathsf{sl}_1, \mathsf{sl}_2, \ldots, \mathsf{sl}_{\mathtt{eplen}}\})$ up to `eplen`. In the beginning of an epoch, each slot of the epoch is assigned to *one* block producer with $\mathcal{F}_{\mathsf{sle}}$ executed before starting to the epoch. Each block producer produces a block when they are in a slot that they are the leader.

Sassafras is run in a key evolving signature functionality $\mathcal{F}_{\mathsf{sgke}}$ [18] to sign the blocks, $\mathcal{F}_{\mathsf{com}}$ (See Figure 1) and $\mathcal{F}_{\mathsf{sle}}$ (See Figure 2) hybrid model by some parties $\mathsf{P}_1, \mathsf{P}_2, \ldots, \mathsf{P}_n$. Each party (ITI) P_i is initiated with an input the session id sid and a stake value st_i given by the $\mathcal{Z}$ and starts running the protocol below. For the sake of simplification and not being wandered from the main contribution, we simply assume that there exists a mechanism (e.g. as in [2]) which helps the newly joining party to converge the current state of the protocol correctly.

Genesis Phase: Each P_i registers to $\mathcal{F}_{\mathsf{sgke}}$ to obtain the verification key $\mathsf{pk}_i^{\mathsf{kesign}}$ for the key evolving signature. Then, it gives to $\mathcal{F}_{\mathsf{init}}$ with the message $(\mathit{register}, \mathsf{sid}, \mathsf{st}_i, \mathsf{pk}_i^{\mathsf{kesign}})$. When $\mathcal{F}_{\mathsf{init}}$ is instructed by the environment, it sends all registered parties the message $(\mathit{genesis}, \mathsf{sid}, \mathbf{b}_0)$ where $\mathbf{b}_0$ is the genesis block. $\mathbf{b}_0$ includes the list of parties with their public keys and stake values who registered and protocol parameters: i.e., $\mathbf{b}_0 = (\{\mathsf{P}_i, \mathsf{pk}_i^{\mathsf{kesign}}, \mathsf{st}_i\}_{i=1}^n, \mathtt{max\text{-}elect}, \mathtt{eplen})$ where n is the number of parties registered to $\mathcal{F}_{\mathsf{init}}$ before the genesis block, `max-elect` is the parameter

of $\mathcal{F}_{\mathsf{sle}}$ and `eplen` is number of slots that one epoch has. We assume that genesis phase is epoch 0.

Election Phase: In this phase, parties run $\mathcal{F}_{\mathsf{sle}}$ to be assigned as a leader for the slots of an epoch $e \geq 1$. Before starting the epoch, each party obtains the stake distribution $\mathcal{D}_S$ of e from the previous epoch $e-1$. Then , they send a message $(\mathit{elect}, \mathsf{sid}||e)$ to $\mathcal{F}_{\mathsf{sle}}$ with the parameter $(\mathtt{eplen}, \mathtt{max\text{-}elect}, \mathcal{D}_S)$ and receive the slot numbers that they are the leaders.

Block Production: Each P_i produces a block when they are in a slot that they are selected as a slot leader. Each P_i keeps a local set of blockchains $\mathfrak{C}_i = \{\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_t\}$. All these chains have some common blocks, at least the genesis block, up until some height. We assume that each party has a local transaction bucket that contains the valid transactions to be added into blocks. All transactions in a block is validated with a transaction validation function ValidateTransaction which receives the transaction and a chain $\mathcal{C} \in \mathfrak{C}_i$ as an input and outputs valid/invalid.

When P_i is a slot leader of a slot sl of an epoch e, it produces the block of this slot as follows:

- it obtains the best chain $\mathcal{C}$ that was generated by Algorithm 3 [18] and retrieves the hash of the last block of $\mathcal{C}$ which is $H_{\mathcal{C}}$.
- retrieves a list of transactions $\mathcal{TX}$ from its transaction bucket.

In the end, P_i generates the block data $\mathsf{data} = (H_{\mathcal{C}}, \mathcal{TX}, \mathsf{sl}, e)$ and obtains the signature of the data σ_{sl} by sending $(\mathit{sign\_update}, \mathsf{sid}, \mathsf{data}, \mathsf{sl})$ to $\mathcal{F}_{\mathsf{sgke}}$. Then, P_i sends $\mathbf{b}_{\mathsf{sl}} = (\mathsf{data}, \sigma_{\mathsf{sl}})$ to the other parties via $\mathcal{F}_{\mathsf{com}}$ and sends message to $\mathcal{F}_{\mathsf{sle}}$ to reveal its leadership for sl.

In any case (being a slot leader or not being a slot leader), when P_i receives a block $\mathbf{b}_{\mathsf{sl}} = (\mathsf{data}, \sigma_t)$ produced by any block producer P_t for a slot sl, it starts validating the block with the algorithm ValidateBlock$(\mathbf{b}_{\mathsf{sl}}, \mathfrak{C}_i)$ which checks the validity of the followings:

- validity of the signature by sending the message $(\mathit{verify}, \mathsf{sid}, \mathsf{pk}_i^{\mathsf{kesign}}, \mathsf{data}, \mathsf{sl}, \sigma_t)$ to $\mathcal{F}_{\mathsf{sgke}}$,
- check whether $\mathcal{F}_{\mathsf{sle}}$ sends a message $(\mathit{reveal}, \mathsf{sid}, \mathsf{P}_t, \mathsf{sl})$,
- validity of the block i.e., if there exists a chain $\mathcal{C} \in \mathfrak{C}_i$ with the header $H(\mathcal{C})$ and ValidateTransaction$(\mathcal{C}, \mathcal{TX})$ is valid.

If a block $\mathbf{b}_{\mathsf{sl}}$ passes all the validation steps, P_j appends $\mathbf{b}_{\mathsf{sl}}$ to $\mathcal{C}$ and adds updated $\mathcal{C}$ to $\mathfrak{C}$. Otherwise, it ignores the block. In the end of the slot, P_j decides the best chain with the algorithm MaxValid$(\mathfrak{C}, \mathcal{C}_{\mathsf{best}})$ [18] (Algorithm 3) where C_{best} is the best chain in the previous run of the algorithm. MaxValid selects the longest chain which does not fork from the current best chain more than k blocks.

Algorithm 3 MaxValid$(\mathfrak{C}, \mathcal{C}_{best})$ [18]

```
C_max ← 𝔠[0] //obtain the first chain in 𝔠
for all C_cand ∈ 𝔠 do
    if |C_cand| > |C_max| ∧ C_best^⌈k ⪯ C_cand then
        C_max ← C_cand
return C_max
```

Update Phase: A party updates its stake by creating a transaction. The update will be valid ℓ_{update}-epochs later than the transaction is added to the blockchain.

5.1 Security Analysis of Sassafras

Our security analysis follows the analysis by Bernardo et al. [18] because Sassafras follows the same structure as Ouroboros Praos except for the leader election protocol and adversarial model. One can imagine Sassafras as Ouroboros Praos where only one party is assigned to a slot and a party can be weakly corrupted during its slot. We consider a party that is weakly corrupted during its slot as malicious. Therefore, we need the probability that the adversary guesses the leader of a slot during the slot.

We first analyse the security of Sassafras in an adversarial model that we call the no-abort model. In this model, the adversary never aborts some slots during the distribution phase of $\mathcal{F}_{\mathsf{sle}}$ i.e., $\mathcal{L}_{\mathsf{abort}} = \emptyset$ in the distribution phase. This is necessary to have the analysis of slots below. If we were not in this adversarial model, we could not easily compute the probability of a slot is not corrupted because the adversary influences it by aborting some of its slots. For example, assume that sl_i and sl_{i+2} are assigned to a malicious party and sl_{i+1} is assigned to an honest party in the end of election phase of $\mathcal{F}_{\mathsf{sle}}$. In this case, whether sl_{i+1} is going to be assigned to an honest party in the final election result depends on the adversary's decision. If it aborts sl_i, then sl_{i+1} is going to be assigned to a malicious party. We note that aborting the slots does not give any advantage to the adversary to break the security of chain properties of Sassafras as will be seen at the end of the analysis. It is complicated to define certain probabilities for slots with the adversary which can abort. One can imagine intuitively the aborted slots as empty slots where there is not any block production. We know that they do not affect the security of a blockchain as long as the probability of having a corrupted slot leader is less than half [18,27].

Analysis of slots: We distinguish slots as a public, malicious, weak and honest slots as defined below. We use the fact that $\mathcal{F}_{\mathsf{sle}}$ is θ-unpredictable and fair i.e., the probability that a slot is assigned to an honest party is α for each run of ELECT.

Public Slot: sl is a public slot if $\mathtt{Leak}[\mathsf{sl}] \neq \perp$. The probability of being a public slot is $p_{\mathsf{pSlot}} = \alpha(1-\alpha) = \alpha - \alpha^2$.

Malicious Slot: sl is malicious if its leader is malicious. The probability of being a malicious slot is $p_{\mathsf{mSlot}} = 1 - \alpha$.

Weak Slot: sl is a weak slot if the honest leader of the slot is weakly corrupted during this slot. If sl is a public slot then the adversary can weakly corrupt the honest leader during its slot. If a slot is neither public nor malicious, the adversary can weakly corrupt at most α_w-fraction of honest parties during the slot by guessing and succeeds if one of the weakly corrupted honest parties is the leader of the slot. Remember that each non-public and non-malicious slot is θ-unpredictable thanks to $\mathcal{F}_{\mathsf{sle}}$. Therefore, the probability of having a weak slot is $p_{\mathsf{wSlot}} = p_{\mathsf{pSlot}} + \theta\alpha_w n(1 - p_{\mathsf{mSlot}} - p_{\mathsf{pSlot}})$.

Honest slot: sl is an honest slot if it is neither weak nor malicious slot. The probability of having an honest slot is $p_{\mathsf{hSlot}} = 1 - p_{\mathsf{mSlot}} - p_{\mathsf{wSlot}}$.

Analysis of chain properties: We show the chain properties (defined below) of Sassafras in one epoch if certain relations satisfied between $p_{\mathsf{mSlot}}, p_{\mathsf{wSlot}}$ and p_{hSlot} assuming that $\mathcal{F}_{\mathsf{sle}}$ satisfies fairness with `max-elect`-reelection, Θ-unpredictability and robustness. The first property is the common prefix (CP) property [20] defined below. The CP property makes sure that the honest parties have a consensus on some blocks.

Definition 8 (Common Prefix (CP) Property [20]). *Common prefix with the parameter $k \in \mathbb{N}$ ensures that any chains $\mathcal{C}_i, \mathcal{C}_j$ possessed by two honest parties at the onset of the slots $\mathsf{sl}_i < \mathsf{sl}_j$ satisfies the following except with a negligible probability in k: the prefix of $\mathcal{C}_i$ obtained by removing the last k blocks is also the prefix of $\mathcal{C}_j$.*

Lemma 3 (CP). *Assuming $p_{\mathsf{hSlot}} > \frac{1}{2}$, Sassafras in the no-abort model satisfies the CP property with the parameter k except with the probability* $\texttt{max-elect}\exp(-\Theta(k))$.

Proof. If $\texttt{max-elect} = 1$, Kiayias et al. [27] shows if $p_{\mathsf{hSlot}} > \frac{1}{2}$, the CP property is guaranteed with except with the probability $\exp(-\Theta(k))$. So, Sassafras in the no-abort model satisfies the CP property except with the probability $\texttt{max-elect}\exp(-\Theta(k))$.

The next property is existential chain quality (ECQ) [20] which guarantees existence of at least one honest block in some length of a sub-chain.

Definition 9 (Existential Chain Quality (ECQ) Property [20]). *The ECQ property with parameter $k_{ecq} \in \mathbb{N}$ ensures that there exists at least one honest block in every k_{ecq} length portion of a blockchain owned by an honest party except with a negligible probability in k_{ecq}.*

Lemma 4 (ECQ). *Assuming $p_{\mathsf{mSlot}} < p_{\mathsf{hSlot}}$, $\mathcal{A}$ breaks the ECQ property in Sassafras in the no-abort model with the parameter $k_{ecq} \in \mathbb{N}$ except with probability* $1-\texttt{max-elect}\exp(-\Omega(k_{ecq}))$.

Proof. The analysis is similar to ECQ analysis in [18]. Let's consider consecutive k_{ecq} blocks $\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_{k_{ecq}}$ of a chain $\mathcal{C}$ owned by an honest party and assume that all of them generated by malicious parties. Imagine that the first honest block before $\mathbf{b}_1$ is $\mathbf{b}_{h_1} \in \mathcal{C}$ and the first honest block after $\mathbf{b}_{k_{ecq}}$ is $\mathbf{b}_{h_2} \in \mathcal{C}$. Remark that $\mathbf{b}_{h_1}$ exists because it is the genesis block in the worst case. Also, $\mathbf{b}_{h_2}$ exists because it is owned by an honest party so in the worst case it is the last block of $\mathcal{C}$. We know that the difference between the length of the chain whose last block is generated by the honest party and the length of the chain whose last block is generated by the next honest party is at least one because honest parties always build on top of the longest chain. Therefore, the length of the other chains that the honest parties build will be at least the number of honest slots (h) between the slot of $\mathbf{b}_{h_1}$ and $\mathbf{b}_{h_2}$. Since $\mathcal{C}$ is the honest party's best chain, it is longer than other chains meaning that $k_{ecq} \geq h$. Since $p_{\mathsf{mSlot}} < p_{\mathsf{hSlot}}$ the probability $h < k_{ecq}$ falls exponentially with k_{ecq} [18].

We next define honest chain growth (HCG) which is weaker version of the chain growth (CG) property we define later. The HCG and ECQ properties help us to prove the CG property.

Definition 10 (Honest Chain Growth (HCG) [2]). *The HCG property with parameters $s_{hcg} \in \mathbb{N}$ and $\kappa_{hcg} \in (0, 1]$ ensures that the sub-chain $\mathcal{C}[\mathsf{sl}_u + \mathsf{sl}_v]$ of a blockchain $\mathcal{C}$ owned by an honest party has the length at least $s_{hcg}\kappa_{hcg}$ where sl_u and $\mathsf{sl}_v \geq \mathsf{sl}_u + s_{hcg}$ are assigned to parties except with the negligible probability in s_{hcg}.*

Lemma 5 (HCG). *Assuming $\mathcal{F}_{\mathsf{sle}}$ satisfies fairness in number of* $\texttt{max-elect}$*-reelection, Θ-unpredictability, robustness, Sassafras in the no-abort model satisfies the honest chain growth property with $\kappa_{hcg} = p_{\mathsf{hSlot}}(1 - \delta_{\mathsf{hSlot}})$ in s_{hcg} slots except with the probability* $\texttt{max-elect}\exp(-\frac{s_{hcg}p_{\mathsf{hSlot}}\delta^2_{\mathsf{hSlot}}}{2})$ *where $0 < \delta_{\mathsf{hSlot}} < 1$.*

Proof. Consider a chain $\mathcal{C}$ owned by an honest party at slot sl. Let sl_u and sl_v be two slots where $\mathcal{C}[\mathsf{sl}_u]$ and $\mathcal{C}[\mathsf{sl}_v]$ are generated by honest parties and $\mathsf{sl}_u + s_{hcg} \leq \mathsf{sl}_v$. As discussed in Lemma 4, the chain grows between $\mathcal{C}[\mathsf{sl}_u]$ and $\mathcal{C}[\mathsf{sl}_v]$ at least the number of honest slots. Therefore, let's define a random variable S_i which is 1 if i is an honest slot and upper bound the probability of sum of honest slots between sl_u and sl_v with the Chernoff bound for all $0 < \delta_{\mathsf{hSlot}} < 1$.

$$\Pr\left[\texttt{honest\_slots} = \sum_{i=\mathsf{sl}_u}^{\mathsf{sl}_v} S_i \leq s_{hcg}p_{\mathsf{hSlot}}(1 - \delta_{\mathsf{hSlot}})\right] < \exp(-\frac{s_{hcg}p_{\mathsf{hSlot}}\delta^2_{\mathsf{hSlot}}}{2})$$

Chain growth (CG) property is stronger version of HCG as defined below:

Definition 11 (Chain Growth (CG) Property [20]). *The CG property with parameters $s_{cg} \in \mathbb{N}$ and $\kappa_{cg} \in (0,1]$ ensures that the sub-chain $\mathcal{C}[\mathsf{sl}_i + \mathsf{sl}_j]$ of a blockchain $\mathcal{C}$ owned by an honest validator has the length at least $s_{cg}\kappa_{cg}$ with sl_i and $\mathsf{sl}_j \geq \mathsf{sl}_i + s_{cg}$ except with the negligible probability in s_{cg}.*

Remark that HCG property is defined for two honest slots while the CG property is defined for any two slots in certain distance. Therefore, it is the more generic version of HCG.

Lemma 6 (CG). *Assuming that $\mathcal{F}_{\mathsf{sle}}$ satisfies fairness in number of* max-elect*-reelection, Θ-unpredictability ,robustness and Sassafras in the no-abort model satisfies ECQ property with the parameter k_{ecq}, Sassafras in the no-abort model satisfies CG property with the parameter $s_{cg} = 2s_{ecq} + s_{hcg}$ and $\kappa_{cg} = \kappa_{cg}\frac{s_{hcg}}{2s_{ecq}+s_{hcg}}$ where $s_{ecq}\kappa_{cg} \leq k_{ecq} \leq s_{ecq}$ except with the probability* $\texttt{max-elect}\exp(-\frac{s_{hcg}p_{\mathsf{hSlot}}\delta^2_{\mathsf{hSlot}}}{2})$.

Proof. We use the similar proof technique in [2]. Let slots $\mathsf{sl}_i, \mathsf{sl}_j$ be $\mathsf{sl}_j \geq \mathsf{sl}_i + s_{cg}$. Consider the sub-chain chain $\mathcal{C}[\mathsf{sl}_i, \mathsf{sl}_i + s_{ecq}]$ spanned between slots sl_i and $\mathsf{sl}_i + s_{ecq}$ and the sub-chain $\mathcal{C}[\mathsf{sl}_j - s_{ecq} : \mathsf{sl}_j]$ spanned between slots $\mathsf{sl}_j - s_{ecq}$ and sl_j. Thanks to the ECQ property $\mathcal{C}[\mathsf{sl}_i, \mathsf{sl}_i + s_{ecq}]$ contains at least one block $\mathbf{b}_1$ generated in an honest slot because $s_{ecq}\kappa_{cg} \leq k_{ecq}$. Similarly, $\mathcal{C}[\mathsf{sl}_j - s_{ecq} : \mathsf{sl}_j]$ contains at least one block $\mathbf{b}_2$ generated on an honest slot. Lastly, we consider another sub-chain between blocks $\mathbf{b}_1$ and $\mathbf{b}_2$. Since $s_{cg} = 2s_{ecq} + s_{hcg}$, the difference between slots of $\mathbf{b}_1$ and $\mathbf{b}_2$ is at least s_{hcg}. Thanks to the HCG property, the length of the the sub-chain between $\mathbf{b}_1$ and $\mathbf{b}_2$ is at least equal to $s_{hcg}p_{\mathsf{hSlot}}(1 - \delta_{\mathsf{hSlot}})$. So, the chain grows in between sl_i and sl_j at least $s_{hcg}\kappa_{hcg}$ which implies that $\kappa_{cg} = \kappa_{hcg}\frac{s_{hcg}}{2s_{ecq}+s_{hcg}}$ and $s = 2s_{ecq} + s_{hcg}$.

Definition 12 (Secure Blockchain [20]). *We call a blockchain protocol is secure if satisfies the following properties:*

- ***Persistence:*** *A transaction tx is called final if it is more than k blocks deeper in the blockchain. Persistence guarantees that if an honest party claims that tx is final, then other honest parties claim the finality of the transaction.*
- ***Liveness:*** *Liveness guarantees that a transaction tx generated by an honest party becomes final in Δ_{live}-slots.*

Theorem 2. *Assuming $\mathcal{F}_{\mathsf{sle}}$ satisfies fairness in number of* max-elect*-reelection, Θ-unpredictability, robustness, $p_{\mathsf{hSlot}} \geq \frac{1}{2}$ and given that k_{ecq} is the ECQ parameter, $k > 2k_{ecq}$ is the CP parameter, $s_{hcg} = \frac{k}{\kappa_{hcg}}$, $s_{ecq} = k_{ecq}/\kappa_{cg}$, the epoch length is* $\texttt{eplen} = 2s_{ecq} + s_{hcg}$ *and $\ell_{\mathsf{update}} \geq 1$, Sassafras in the no-abort model is a secure blockchain protocol with parameters k and $\Delta_{live} \leq s_{ecq} + \texttt{eplen}$.*

Proof Sketch: The overall result says that $\kappa_{cg} = \kappa_{hcg}\frac{s_{hcg}}{2s_{ecq}+s_{hcg}} = \frac{k}{s_{hcg}}\frac{s_{hcg}}{2s_{ecq}+s_{hcg}} = \frac{k}{\texttt{eplen}}$. The best chain at the end of an epoch grows at least k blocks in one epoch thanks to the chain growth.

Persistence: Thanks to the CP property, any transaction in a block becomes final after k-blocks later which happens at latest one epoch later.

Liveness: The only case that $\mathcal{F}_{\mathsf{sle}}$ aborts is the case when $|\texttt{Leads}| - |\mathcal{L}_{\mathsf{abort}}| < \texttt{eplen}$. Since Elect run by $\mathcal{F}_{\mathsf{sle}}$ is robust, $\mathcal{F}_{\mathsf{sle}}$ aborts with negligible probability. The second case is when the honest parties give different stake distribution to $\mathcal{F}_{\mathsf{sle}}$. All honest parties should run $\mathcal{F}_{\mathsf{sle}}$ in

an epoch e parametrised with a stake distribution $\mathcal{D}_S$ of e. Therefore, each honest party should agree on the stake distribution of e before contacting with $\mathcal{F}_{\mathsf{sle}}$. Since $\ell_{\mathsf{update}} \geq 1$ epoch, the honest parties obtain the same stake distribution from epochs less than $e - \ell_{\mathsf{update}}$ just before contacting $\mathcal{F}_{\mathsf{sle}}$ for the election of epoch e thanks to the common prefix property. Therefore, $\mathcal{F}_{\mathsf{sle}}$ never aborts except with negligible probability and all honest parties contacts with the same functionality $\mathcal{F}_{\mathsf{sle}}$ parametrized with the stake distribution of e. This shows that each slot is assigned to one party during the lifetime of Sassafras. Since $p_{\mathsf{hSlot}} \geq \frac{1}{2}$ and there will be at least one honest block in every s_{ecq} blocks which is finalized at most `eplen`-slots later. □

Theorem 3 (Security of Sassafras). *Assuming that Sassafras in the no-abort model is secure, then Sassafras is secure.*

Proof. We show that if an adversary $\mathcal{A}$ breaks the security of Sassafras then we can construct another adversary $\mathcal{B}$ breaking the security of Sassafras in the no-abort model. Reduction is trivial: Whenever $\mathcal{B}$ receives a message from $\mathcal{F}_{\mathsf{sle}}$, it relays it to $\mathcal{A}$ and responds $\mathcal{F}_{\mathsf{sle}}$ whatever $\mathcal{A}$ responds except with the case where $\mathcal{A}$ aborts some slots i.e., sends $(aborted\_slots, \mathsf{sid}, \mathcal{L}_{\mathsf{abort}})$. In this case, $\mathcal{B}$ sends $(aborted\_slots, \mathsf{sid}, \emptyset)$ to $\mathcal{F}_{\mathsf{sle}}$. For each slot that $\mathcal{A}$ aborts in the distribution phase of $\mathcal{F}_{\mathsf{sle}}$, $\mathcal{B}$ does not produce block in Sassafras in the no-abort model. For the rest of the slots, $\mathcal{B}$ behaves exactly as $\mathcal{A}$. Clearly, if $\mathcal{A}$ breaks the security of Sassafras, then $\mathcal{B}$ breaks the security of Sassafras in the no-abort model. □

5.2 Instantiation of Sassafras in the Real-World

We show that Sassafras in $\mathcal{F}_{\mathsf{sgke}}, \mathcal{F}^s_{\mathsf{com}}$ and $\mathcal{F}_{\mathsf{sle}}$ satisfies liveness and persistence. In this section, we focus on the real world instantiation of Sassafras, namely analysing briefly the instantiations of $\mathcal{F}_{\mathsf{sgke}}, \mathcal{F}^s_{\mathsf{com}}$ and $\mathcal{F}_{\mathsf{sle}}$ and questioning whether right parameters exist for the security of Sassafras.

Realizing secure diffusion: Given that we have $\mathcal{F}_{\mathsf{com}}$ (without secure diffusion), we can instantiate the secure diffusion with a Diffie-Hellman (DH) key exchange and semantically secure symmetric encryption scheme [12]. Each block producer runs DH-key exchange with every block producer once to obtain a shared secret key. Then, in the distribution phase of Sassafras, each party encrypts the ticket with the secret key of the repeater and sends the encryption with $\mathcal{F}_{\mathsf{com}}$. Whenever a receiver is able to decrypt a ciphertext (i.e., obtain a ticket after the decryption) with the secret generated together with the sender, the receiver learns the ticket and sees that it is the repeater of this ticket.

Realizing $\mathcal{F}_{\mathsf{sle}}$: In section 4, we show SASLE realizes $\mathcal{F}_{\mathsf{sle}}$ running Algorithm 2. So, we can replace $\mathcal{F}_{\mathsf{sle}}$ with SASLE in Sassafras. In a nutshell, Sassafras with SASLE works as follows: During the genesis phase, each P_i registers for $\mathcal{F}^s_{\mathsf{rvrf}}$ and obtains $\mathsf{pk}_i^{\mathsf{rvrf}}$ to be added in the genesis block as the signing key. We remind that SASLE needs a secure SMR. Therefore, we use Sassafras as our SMR used in SASLE. Also, we use SASLE to obtain secure SMR which is Sassafras. This looks like a chicken and egg problem but the solution is as follows. We assume that the first run of $\mathcal{F}_{\mathsf{sle}}$, just after the genesis phase, is not instantiated with SASLE. This can be instantiated with an inefficient or trusted third-party-based secret single-leader election which does not require SMR. The inefficiency is not a problem at the first run because it is before starting the actual block production protocol. After the election that realizes $\mathcal{F}_{\mathsf{sle}}$ is executed for the first epoch of Sassafras, the block production of the first epoch is executed as described in the Sassafras. We remark that now we obtain a secure SMR for the first `eplen`-slots of Sassafras according

to Theorem 3 (obtained from Theorem 2)). Therefore, SASLE can deploy this SMR. So, while running the first epoch of Sassafras, block producers run SASLE in parallel. Namely, they run the preparation phase of SASLE to generate the tickets and distribute the tickets to the repeaters as in the distribution phase. Since the SMR in SASLE is replaced with Sassafras, each repeater submits the tickets during the submission phase by generating a transaction including the ticket. Thus, the ticket will be added in a block during the block production of the first epoch since the first epoch of Sassafras satisfies liveness and persistence. Next, block producers run the sortition phase of SASLE in order to obtain the leaders of the second epoch. Now, Sassafras is a secure SMR for the second epoch as well because SASLE realizes $\mathcal{F}_{\mathsf{sle}}$. In a nutshell, once $\mathcal{F}_{\mathsf{sle}}$ is bootstrapped, block producers run SASLE during an epoch e to obtain the leaders of $e+1$ by using the Sassafras in e is the SMR. Whenever P_i produces a block after the first epoch, it also obtains a signature $(sign, \mathsf{sid}, \{\mathsf{pk}_i^{\mathsf{rvrf}}\}, W_{ij}, r_e||j||e, \mathsf{ass})$ from $\mathcal{F}^s_{\mathsf{rvrf}}$ (See the verification phase of SASLE in Section 4 for more details) to show its leadership of the slot sl where the ticket including $W_{ij}, r_e||j||e$ is in the order sl. Then, P_i obtains $\hat{\sigma_{ij}}$ from $\mathcal{F}^s_{\mathsf{rvrf}}$. In the end, along with the block $\mathbf{b}_{\mathsf{sl}}$, it also sends $\hat{\sigma}_{ij}$. Therefore, VALIDATEBLOCK validates $\hat{\sigma_{ij}}$ with $\mathcal{F}^s_{\mathsf{rvrf}}$ as well as the block to verify that the party who produces the block for sl is the leader of sl.

Realizing $\mathcal{F}_{sgke}$ and removing $\mathcal{F}_{sgke}$: $\mathcal{F}_{\mathsf{sgke}}$ in the real world realizable [18] by a forward secure key evolving signatures (KES) in the standard model [3,23]. A PoS-blockchain protocol needs a key evolving signature scheme because if a block producer is corrupted at some slot, the adversary should not able to produce blocks for the past slots. Otherwise, the security of the protocol can be trivially compromised in the adaptive adversarial model [18]. We realized that we can prevent an adversary to generate blocks for the past slots by a standard EUF-CMA signature scheme if Sassafras deploys SASLE. This is important optimization because KES is not as efficient as a classical signature scheme since it needs extra properties such as creating a new secret key from the old key and being able to sign with the secret key. Therefore, it matters especially in PoS-blockchain protocols to avoid using KES to sign a block. In this case, Sassafras works as follows: At the beginning of the election phase of SASLE, each party P_i obtains `max_attempts`-many ephemeral signing secret/public key pair $(\mathsf{esk}_j, \mathsf{epk}_j)$ from the key generation of the signature scheme. Then, it lets the associated data be $\mathsf{ass}_j = \mathsf{epk}_j$ and runs SASLE as described. Remember that ass_j is the part of the ring VRF signing progress. So, each ticket is in the form of $(r_e, j, e, W_{ij}, \sigma_{ij}, \pi_{ij}, \mathsf{epk}_j)$ now (See Section 4). The next change in Sassafras is in the block production phase. When the leader of a slot sl generates a block in the block production, it signs the block with esk which is the ephemeral secret key of the ephemeral public key that is in the ticket of the slot instead of signing with $\mathcal{F}_{\mathsf{sgke}}$. Then, the slot leader erases esk. When other block producers verify the block, they verify the signature of the block by running the verification algorithm of the signature scheme with epk. A verified $\hat{\sigma_{ij}}$ shows that epk belongs to the party who produces the ticket of the slot thanks to the given property of $\mathcal{F}_{\mathsf{rvrf}}$. Therefore, they make sure that epk is an ephemeral key of the block producer. This change gives forward security because each ephemeral key is independently generated and deleted just after they are used to produce the block. We remark that Sassafras without $\mathcal{F}_{\mathsf{sgke}}$ is still UC-secure because any EUF-CMA secure signature scheme is UC-secure [10]. So, we can replace the signature scheme with the signature functionality $\mathcal{F}_{\mathsf{sig}}$ [10].

5.3 Parameter selection of Sassafras deploying SASLE:

Theorems 1 and 2 show that Sassafras is secure with high probability, provided the parameters satisfy various assumptions we made along the chain properties. We need to show what com-

binations of parameters are possible by instantiating $\mathcal{F}_{\mathsf{sle}}$ with SASLE. For this we need to be specific about the negligible probabilities involved. It is not immediately obvious that consistent combinations of parameters exist.

Proposition 1. *Given $\rho_{\max}$, an upper bound on the highest probability in the distribution $\mathcal{D}_S$ and α, an upper bound on the sum of the probabilities in $\mathcal{D}_S$ for the malicious parties, a failure probability $\epsilon > 0$ and an unpredictability θ which must satisfy $\theta > \frac{\rho_{\max}}{\alpha}$, we can find* $\mathtt{eplen}, c, \mathtt{max\_attempts}$ *such that with probability at least $1-\epsilon$, SASLE is robust (Lemma 2) and θ-unpredictable (Corollary 1) and the chain properties (Section 5.1) hold. Furthermore, given $\rho_{\max} \leq \frac{1}{n'}$ for some n' and $\alpha = \Theta(1)$, we can take $\theta = \frac{2\rho_{\max}}{\alpha}$, $\epsilon = \exp(-\Omega(\log n'))$, $\Omega(\log n') \leq \mathtt{eplen} \leq O(n')$, $c = \Theta(1)$ and $\mathtt{max\_attempts} = \Theta(1)$.*

Proof. We first consider the first part. We need to set the parameter `eplen` so that the chain properties hold with probability at least $1-\epsilon/3$. The probability that the chain properties in an epoch hold given that the election succeeded and SASLE was θ-unpredictable is at least $1-\exp(O(\mathtt{eplen}))$. So, we can set $\mathtt{eplen} = \Omega(\log 1/\epsilon)$ and have this fail with probability at most $\epsilon/3$.

Next, we need the election to succeed except with probability at most $\epsilon/3$. By Lemma 2, we need $\exp(-\frac{(r-1)^2}{2r}\mathtt{eplen}) \leq \epsilon/3$ where $r = \frac{c\mathtt{max\_attempts}\alpha}{\rho_{\max}\mathtt{eplen}}$ has $r > 1$. Since $\frac{(r-1)^2}{2r}$ has derivative $\frac{(r-1)r-(r-1)^2}{2r^2} = \frac{(r-1)(r+1)}{2r^2} = \frac{1}{2} - \frac{1}{2r^2}$ which is positive when $r > 1$, if we can find an $r' > 1$ with $\exp(-\frac{(r'-1)^2}{2r'}\mathtt{eplen}) = \epsilon/3$, for $r \geq r'$, we have $\exp(-\frac{(r-1)^2}{2r}\mathtt{eplen}) \leq \epsilon/3$. Rearranging the equation for r' gives $r'^2 + 1 - 2r'(1 + \frac{\ln(3/\epsilon)}{\mathtt{eplen}}) = 0$. The only solution of this with $r' > 1$ is $r' = (1 + \frac{\ln(3/\epsilon)}{\mathtt{eplen}}) + \sqrt{(1 + \frac{\ln(3/\epsilon)}{\mathtt{eplen}})^2 - 1} = 1 + \frac{\ln(3/\epsilon)}{\mathtt{eplen}} + \sqrt{(\frac{\ln(3/\epsilon)}{\mathtt{eplen}})^2 + 2(\frac{\ln(3/\epsilon)}{\mathtt{eplen}})}$. Later, we will take $\mathtt{max\_attempts} = \lceil \frac{r'\rho_{\max}\mathtt{eplen}}{c\alpha} \rceil$ which ensures that $r > r'$, but for this we need c.

Lemma 7. *If* $\mathtt{max\_attempts} = \lceil \frac{r'\rho_{\max}\mathtt{eplen}}{c\alpha} \rceil$ *and*

$$c \leq \chi^{\frac{\alpha\chi}{\alpha\chi+\rho_{\max}}} \exp(-1 - \frac{\rho_{\max}(\ln(3r'\rho_{\max}\mathtt{eplen}/\alpha\epsilon + 3/\epsilon) - 1)}{\alpha\chi + \rho_{\max}})$$

then SASLE is $\frac{\rho_{\max}}{\alpha(1-\chi)}$-unpredictable.

Proof. Putting the inequality for c to the power of $\frac{\alpha\chi+\rho_{\max}}{\alpha\chi}$, we have

$$c^{1+\frac{\rho_{\max}}{\alpha\chi}} \leq \chi \exp(-1 - \frac{\rho_{\max}\ln(3r'\rho_{\max}\mathtt{eplen}/\alpha\epsilon + 3/\epsilon)}{\alpha\chi})$$

.

$$\begin{aligned}
c &\leq \chi \exp(-1 - \frac{\rho_{\max}\ln(3r'\rho_{\max}\mathtt{eplen}/\alpha\epsilon + 3/\epsilon)}{\alpha\chi}) c^{-\frac{\rho_{\max}}{\alpha\chi}} \\
&= \chi \exp(-1 - \frac{\rho_{\max}(\ln(3r'\rho_{\max}\mathtt{eplen}/\alpha\epsilon + 3/\epsilon) + \ln(1/c))}{\alpha\chi}) \\
&= \chi \exp(-1 - \frac{\rho_{\max}\ln(3r'\rho_{\max}\mathtt{eplen}/c\alpha\epsilon + 3/c\epsilon)}{\alpha\chi}) \\
&\leq \chi \exp(-1 - \frac{\rho_{\max}\ln(3r'\rho_{\max}\mathtt{eplen}/c\alpha\epsilon + 3/\epsilon)}{\alpha\chi}) \\
&= \chi \exp(-1 - \frac{\rho_{\max}\ln((3/\epsilon)(r'\rho_{\max}\mathtt{eplen}/c\alpha + 1))}{\alpha\chi}) \\
&\leq \chi \exp(-1 - \frac{\rho_{\max}\ln((3/\epsilon)\mathtt{max_attempts})}{\alpha\chi})
\end{aligned}$$

Re-arranging and taking logs give that

$$\ln(\chi/c) - 1 \geq \frac{\rho_{\max} \ln((3/\epsilon)\texttt{max\_attempts})}{\alpha\chi}$$

and so

$$\texttt{max\_attempts} \exp(-\frac{\alpha\chi(\ln(\chi/c) - 1)}{\rho_{\max}}) \leq \epsilon/3 \; .$$

By Corollary 1, except with probability $\epsilon/3$, SASLE is $\frac{\rho_{\max}}{\alpha(1-\chi)}$-unpredictable.

We need to set $\chi = 1 - \frac{\rho_{\max}}{\alpha\theta}$ to obtain θ-unpredictability. With parameters satisfying these, each property holds except with probability $\epsilon/3$. By a union bound, all three properties, that the election does not abort, SASLE is θ-unpredictable and the chain properties, hold except with probability ϵ. This gives the first part of the proposition. Note that c and so `max_attempts` do not depend on ϵ in an ideal way. `max_attempts` has a term that is $(1/\epsilon)^{\rho_{\max}/\alpha\chi}$. This means that if $\rho_{\max}/\alpha\chi$ is large then it takes poly$(1/\epsilon)$ VRF attempts by each party in each epoch to get a small error probability. However in the case of D being the uniform distribution over many parties, $\rho_{\max}$ is small. This is why we consider the case $\rho_{\max} \leq \frac{1}{n'}$ for a large n'. In the case where probabilities are not equal, we would suggest capping $\rho_{\max}$.

Consider choosing $\chi, \texttt{eplen}, c, \texttt{max\_attempts}$ as above for the case $\rho_{\max} = 1/n'$, $\theta = 2\rho_{\max}/\alpha$, $\alpha = \Theta(1)$ and $\epsilon = \exp(-Cn')$ for some constant C. Then $\chi = 1/2$ and $\texttt{eplen} > \Omega(\log(Cn'))$. We could take $\texttt{eplen} = \Theta(\log(n'))$, however we didn't give an explicit constant for the lower bound on `eplen` and there are practical reasons for making epochs long, because we need to perform SASLE in every epoch in Sassafras. Thus, we consider a looser upper bound of $O(n')$.

Next note that $r' = 1 + \ln(3/\epsilon)/\texttt{eplen} + \sqrt{(\ln(3/\epsilon)/\texttt{eplen})^2 + 2(\ln(3/\epsilon)/\texttt{eplen})} = O(1)$ since $\ln(3/\epsilon)/\texttt{eplen} = O(1)$. Now we need to take

$$c \leq \chi^{\frac{\alpha\chi}{\alpha\chi+\rho_{\max}}} \exp(-1 - \frac{\rho_{\max}(\ln(3r'\rho_{\max}\texttt{eplen}/\alpha\epsilon + 3/\epsilon) - 1)}{\alpha\chi + \rho_{\max}})$$

Here $\alpha\chi = \Theta(1)$ and so $\alpha\chi + \rho_{\max} = \Theta(1)$. The first term, $\chi^{\frac{\alpha\chi}{\alpha\chi+\rho_{\max}}}$ is $\Theta(1)$. We have $3r'\rho_{\max}\texttt{eplen}/\alpha\epsilon + 3/\epsilon = O(1/\epsilon) = O(\exp(Cn'))$ and so $(\ln(3r'\rho_{\max}\texttt{eplen}/\alpha\epsilon + 3/\epsilon) - 1) = O(Cn')$. We obtain that $c \leq \exp(-O(C))$ and so we can take $c = \Omega_C(1)$. Then $\texttt{max\_attempts} = \lceil \frac{r'\rho_{\max}\texttt{eplen}}{c\alpha} \rceil = O_C(1)$. Any choice of $C > 0$ gives the final part of the proposition. □

6 Conclusion

We constructed a batched SLE protocol (SASLE) that provides a significant gain compared to the existing SSLE protocols with respect to the message size. One of the implementations [32] of the ring VRF in Appendix A) based on KZG polynomial commitments [24] shows that the ring VRF signature size is 560 bytes. Therefore, the expected communication overhead is 224 KB in SASLE for 200-elections, 2^{14}-parties with equal stakes and the parameters $\texttt{max\_attempts} = 2$ and $c = \frac{200}{2^{14}}$. In the worst case, where all tickets pass the threshold, which happens with a negligible probability, the communication overhead is only 0.018 GB. According to [15], the communication overhead of [15,14,5] is around one gigabyte (GB) for 200 elections with 2^{14} parties with equal stakes. This is huge compared to SASLE's complexity. On the other hand, our protocol partially sacrifices secrecy, which is guaranteed only for some leaders.

Using SASLE, we constructed a PoS-blockchain protocol Sassafras. Sassafras requires a long epoch length to realize $\mathcal{F}_{\mathsf{rand}}$ in the real world [18] i.e., an epoch should be long enough to guarantee almost surely at least one honest block generated during that epoch. Sassafras utilizing SASLE with $n = 2^{10}$ parties with equal stakes is secure if $\texttt{eplen} = 2000$, $\texttt{max\_attempts} = 2^3$, $c = 0.5$, $\alpha > \frac{1}{\sqrt{2}}$. In this case, the number of ring VRF signatures which should be stored on chain in one epoch is 4096 in expectation which is roughly 2293 KB. This shows that the election mechanism that Sassafras deploys is lightweight in terms of data storage, which makes it feasible to store the data on chain. On the downside, Sassafras needs stricter honesty assumptions than the existing PoS-blockchain protocols deploying probabilistic leader election mechanisms [18,28,6,22,17], but it benefits from the security and efficiency gains that SLE provides [1].

References

1. Sarah Azouvi and Daniele Cappelletti. Private attacks in longest chain proof-of-stake protocols with single secret leader elections. In *AFT*, pages 170–182, 2021.
2. Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *ACM SIGSAC*, pages 913–930. ACM, 2018.
3. Mihir Bellare and Sara K Miner. A forward-secure digital signature scheme. In *Annual International Cryptology Conference*, pages 431–448. Springer, 1999.
4. Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *CRYPTO*, pages 757–788. Springer, 2018.
5. Dan Boneh, Saba Eskandarian, Lucjan Hanzlik, and Nicola Greco. Single secret leader election. *IACR Cryptol. ePrint Arch.*, 2020:25, 2020.
6. Jeffrey Burdges, Alfonso Cevallos, Peter Czaban, Rob Habermeier, Syed Hosseini, Fabio Lama, Handan Kılınç Alper, Ximin Luo, Fatemeh Shirazi, Alistair Stewart, et al. Overview of polkadot and its design considerations. *arXiv:2005.13456*, 2020.
7. Jeffrey Burdges, Handan Kılınç Alper, Alistair Stewart, and Sergey Vasilyev. Ethical identity, ring VRFs, and zero-knowledge continuations. Cryptology ePrint Archive, Paper 2023/002, 2023. `https://eprint.iacr.org/2023/002`.
8. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. ePrint Arch. 2000/067, 2000.
9. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE CLUSTER Conference*, pages 136–145. IEEE, 2001.
10. Ran Canetti. Universally composable signature, certification, and authentication. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, pages 219–233. IEEE, 2004.
11. Ran Canetti, Kyle Hogan, Aanchal Malhotra, and Mayank Varia. A universally composable treatment of network time. In *CSF*, pages 360–375. IEEE, 2017.
12. Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 337–351. Springer, 2002.
13. Ignacio Cascudo and Bernardo David. Scrape: Scalable randomness attested by public entities. In *ACNS*, pages 537–556. Springer, 2017.
14. Dario Catalano, Dario Fiore, and Emanuele Giunta. Efficient and universally composable single secret leader election from pairings. *ePrint Arch.*, 2021:344, 2021.
15. Dario Catalano, Dario Fiore, and Emanuele Giunta. Adaptively secure single secret leader election from DDH. 2022.
16. Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science*, 777:155–183, 2019.

17. Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. pages 23–41, 2019.
18. Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *EUROCRYPT*, pages 66–98, 2018.
19. Justin Drake, Dankrad Feist, Gottfried Herold, Dmitry Khovratovich, Mary Maller, and Mark Simkin. Whisk: A practical shuffle-based SSLE protocol for ethereum, 2022. `https://hackmd.io/@asn-d6/HyD3Yjp2Y`.
20. Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, pages 281–310. Springer, 2015.
21. Jens Groth, Rafail Ostrovsky, and Amit Sahai. New techniques for noninteractive zero-knowledge. *Journal of the ACM (JACM)*, 59(3):1–35, 2012.
22. Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.
23. Gene Itkis and Leonid Reyzin. Forward-secure signatures with optimal signing and verifying. In *Annual International Cryptology Conference*, pages 332–354. Springer, 2001.
24. Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, pages 177–194. Springer, 2010.
25. Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In *TCC*, pages 477–498. Springer, 2013.
26. Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In *TCC*, pages 477–498. Springer, 2013.
27. Aggelos Kiayias, Saad Quader, and Alexander Russell. Consistency of proof-of-stake blockchains with concurrent honest slot leaders. *arXiv:2001.06403*, 2020.
28. Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *CRYPTO*, pages 357–388, 2017.
29. Handan Kılınç Alper. Network time with a consensus on clock. ePrint Archive, Paper 2019/1348.
30. Rafael Pass and Elaine Shi. The sleepy model of consensus. In *Asiacrypt*, pages 380–409. Springer, 2017.
31. Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *IEEE SP*, pages 444–460, 2017.
32. Sergey Vasilyev. Ring-vrf ring proof v2.5. 2022. `https://github.com/w3f/ring-proof`.

A Ring Verifiable Random Function

Ring VRF functionality is defined as in Figure 4 in [7]. Burdges et al. [7] constructed the following ring VRF protocol which realizes $\mathcal{F}_{\mathsf{rvrf}}$. The public parameters are prime p-order group $\mathbb{G}$, generators $G_1, G_2 \in \mathbb{G}$. The protocol is built on top of NIZK protocols and random oracles: $\mathsf{H}, \mathsf{H}' : \{0,1\}^* \to \mathbb{F}_p$ and a hash-to-group function $\mathsf{H}_{\mathbb{G}} : \{0,1\}^* \to \mathbb{G}$. rVRF protocol works as follows:

- $\mathsf{rVRF.KeyGen}(1^\lambda)$: It outputs $\mathsf{sk}^{\mathsf{rvrf}} = x \leftarrow_\$ \mathbb{F}_p$ and $\mathsf{pk}^{\mathsf{rvrf}} = X = xG_1$.
- $\mathsf{rVRF.Sign}(\mathsf{sk}^{\mathsf{rvrf}}, \mathcal{PK}, m, \mathsf{ass})$: It lets $P = \mathsf{H}_{\mathbb{G}}(m)$ and lets $W = xP$. It generates $C = X + \beta G_2$ where $\beta \leftarrow_\$ \mathbb{F}_p$. Then, it generates a Schnorr proof π_{com} showing the following relation, i.e., $\mathsf{NIZK.Prove}(\mathcal{R}_{\mathsf{com}}, ((x,\beta),(G_1,G_2,\mathbb{G},C,W,P))) \to \pi_{com}$ where $\mathcal{R}_{\mathsf{com}} = \{((x,\beta),(G_1,G_2,\mathbb{G},C,W,P,\mathsf{ass})) : C = xG_1 + \beta G_2, W = xP\}$ Here Prove algorithm runs a non-interactive Chaum-Pedersen proof with the Fiat-Shamir transform: Sample random $r_1, r_2 \leftarrow \mathcal{F}_p$. Let $R = r_1G_1 + r_2G_2$, $R_m = r_1P$, and $c = \mathsf{H}'(\mathcal{PK}, m, W, C, R, R_m, \mathsf{ass})$. Set

$\pi_{com} = (c, s_1, s_2)$ where $s_1 = r_1 + cx$ and $s_2 = r_2 + c\beta$ It generates the second proof π_{pk} for the following relation $\mathcal{R}_{\mathsf{pk}} = \{(X, \beta), (G_1, G_2, \mathbb{G}, \mathcal{PK}, C)) : C - \beta G_2 = X \in \mathcal{PK}\}$ with the witness $(\mathcal{PK}, X, \beta)$ by running $\mathsf{NIZK.Prove}(\mathcal{R}_{\mathsf{pk}}, ((X, \beta), (G_1, G_2, \mathbb{G}, \mathcal{PK}, C)))$ In the end, $\mathsf{rVRF.Sign}$ outputs σ, W where $\sigma = (\pi_{com}, \pi_{pk}, C)$.

– $\mathsf{rVRF.Verify}(\mathcal{PK}, W, m, \mathsf{ass}, \sigma)$: Given $\sigma = (\pi_{com}, \pi_{pk}, C)$ and $\mathcal{PK}, W$, it runs $\mathsf{NIZK.Verify}(\mathcal{R}_{\mathsf{com}}, (G_1, G_2, \mathbb{G}, C, W, P, \mathsf{ass}), \pi_{com})$ and runs $\mathsf{NIZK.Verify}(\mathcal{R}_{\mathsf{pk}}, (G_1, G_2, \mathbb{G}, \mathcal{PK}, C), \pi_{pk})$. If all verify, it outputs 1 and the evaluation value $y = \mathsf{H}(m, W)$. Otherwise, it outputs $(0, \perp)$.

B Secret Ring VRF

Another version of $\mathcal{F}_{\mathsf{rvrf}}$ called $\mathcal{F}^s_{\mathsf{rvrf}}$ operates as $\mathcal{F}_{\mathsf{rvrf}}$. In addition, it also lets a party generate a secret element to check whether it satisfies a certain relation i.e., $((m, y, \mathcal{PK}), (\eta, \mathsf{pk}_i^{\mathsf{rvrf}})) \in \mathcal{R}$ where η is the secret random element. If it satisfies the relation, then $\mathcal{F}^s_{\mathsf{rvrf}}$ generates a proof. Proving works as $\mathcal{F}_{zk}$ [21] except that a part of the witness (η) is generated randomly by the functionality. $\mathcal{F}^s_{\mathsf{rvrf}}$ is useful in applications where a party wants to show that the random output y satisfies a certain relation without revealing his identity.

C A version of Chernoff bound

We will use the following Chernoff bound.

Lemma 8. *If $X = \sum_i w_i X_i$ is a weighted sum of independent (but not necessarily identical) Bernoulli random variables X_i with $0 \leq w_i \leq w_{\max}$ then, for any $s \geq E[X]$:*

$$Pr[X \geq s] \leq \exp((-E[X] - s(\ln(s/E[X]) - 1))/w_{\max})$$

Proof. Letting $p_i = E[X_i]$ and $\mu = E[X] = \sum_i p_i w_i,$, by Markov's inequality, for $t = \ln(s/\mu)/w_{\max}$,

$$\begin{aligned}
\Pr[X \geq s] &= Pr[e^{Xt} \geq e^{St}] \\
&\leq E[e^{Xt}]/e^{st} \\
&= \exp(-st) \prod_{i=1}^{n} (1 - p_i + p_i exp(w_i t)) \\
&= \exp(-st + \sum_i \ln(1 + p_i(exp(w_i t) - 1))) \\
&\leq \exp(-st + \sum_i \ln(\exp(p_i exp(w_i t) - 1))) \\
&= \exp(-st + \sum_i p_i(exp(w_i t) - 1)) \\
&\leq \exp(-st + \sum_i p_i w_i(exp(w_{\max} t) - 1)/w_{\max}) \\
&= \exp(-s(\ln(s/\mu)/w_{\max}) + \sum_i p_i w_i(s/\mu - 1)/w_{\max}) \\
&= \exp((-s\ln(s/\mu) - \mu + s)/w_{\max})
\end{aligned}$$

where the second inequality used that $1 + x \leq \exp(x)$ and the third used the convexity of exp.

$\mathcal{F}_{\mathsf{rvrf}}$ runs two PPT algorithms Gen_W and Gen_{sign} during the execution.

Key Generation. upon receiving a message $(keygen, \mathsf{sid})$ from a party P_i, send $(keygen, \mathsf{sid}, \mathsf{P}_i)$ to the simulator Sim. Upon receiving a message $(verificationkey, \mathsf{sid}, \mathsf{pk}^{\mathsf{rvrf}})$ from Sim, verify that $\mathsf{pk}^{\mathsf{rvrf}}$ has not been recorded before for sid; then, store in the table $\mathtt{verification\_keys}$, under P_i, the value $\mathsf{pk}^{\mathsf{rvrf}}$. Return $(verificationkey, \mathsf{sid}, \mathsf{pk}^{\mathsf{rvrf}})$ to P_i.

Malicious Key Generation. upon receiving a message $(keygen, \mathsf{sid}, \mathsf{pk}^{\mathsf{rvrf}})$ from Sim, verify that $\mathsf{pk}^{\mathsf{rvrf}}$ was not yet recorded, and if so record in the table $\mathtt{verification\_keys}$ the value $\mathsf{pk}^{\mathsf{rvrf}}$ under Sim. Else, ignore the message.

Corruption: upon receiving $(corrupt, \mathsf{sid}, \mathsf{P}_i)$ from Sim, remove $\mathsf{pk}_i^{\mathsf{rvrf}}$ from $\mathtt{verification\_keys}[\mathsf{P}_i]$ and store $\mathsf{pk}_i^{\mathsf{rvrf}}$ to $\mathtt{verification\_keys}$ under Sim. Return $(corrupted, \mathsf{sid}, \mathsf{P}_i)$.

Malicious Ring VRF Evaluation. upon receiving a message $(eval, \mathsf{sid}, \mathsf{pk}_i^{\mathsf{rvrf}}, W, m)$ from Sim, if $\mathsf{pk}_i^{\mathsf{rvrf}}$ is recorded under an honest party's identity or if there exists $W' \neq W$ where $\mathtt{anonymous\_key\_map}[m, W'] = \mathsf{pk}_i^{\mathsf{rvrf}}$, ignore the request. Otherwise, record in the table $\mathtt{verification\_keys}$ the value $\mathsf{pk}_i^{\mathsf{rvrf}}$ under Sim if $\mathsf{pk}_i^{\mathsf{rvrf}}$ is not in $\mathtt{verification\_keys}$. If $\mathtt{anonymous\_key\_map}[m, W]$ is not defined before, set $\mathtt{anonymous\_key\_map}[m, W] = \mathsf{pk}_i^{\mathsf{rvrf}}$ and let $y \leftarrow\$ \{0,1\}^{\ell_{\mathsf{rVRF}}}$ and set $\mathtt{Out}[m, W] = y$. Then, set $\mathtt{Out}[m, W] = y$, $\mathtt{anonymous\_key\_map}[m, W] = \mathsf{pk}_i^{\mathsf{rvrf}}$ obtain $y = \mathtt{Out}[m, W]$. Otherwise, obtain $y = \mathtt{Out}[m, W']$. Return $(evaluated, \mathsf{sid}, m, \mathsf{pk}_i^{\mathsf{rvrf}}, W, y)$ to P_i.

Honest Ring VRF Signature and Evaluation. upon receiving a message $(sign, \mathsf{sid}, \mathcal{PK}, \mathsf{pk}_i^{\mathsf{rvrf}}, m)$ from P_i, verify that $\mathsf{pk}_i^{\mathsf{rvrf}} \in \mathcal{PK}$ and that there exists a public key $\mathsf{pk}_i^{\mathsf{rvrf}}$ associated to P_i in the table $\mathtt{verification\_keys}$. If that wasn't the case, just ignore the request. If there exists no W' such that $\mathtt{anonymous\_key\_map}[m, W'] = \mathsf{pk}_i^{\mathsf{rvrf}}$, let $W \leftarrow\$ \{0,1\}^{w(\lambda)}$ and let $y \leftarrow\$ \{0,1\}^{\ell_{\mathsf{rVRF}}}$. If there exists W where $\mathtt{anonymous\_key\_map}[m, W]$ is defined, then abort. Otherwise, set $\mathtt{anonymous\_key\_map}[m, W] = \mathsf{pk}_i^{\mathsf{rvrf}}$ and set $\mathtt{Out}[m, W] = y$. Obtain W, y where $\mathtt{anonymous\_key\_map}[m, W] = \mathsf{pk}_i^{\mathsf{rvrf}}$ and $\mathtt{Out}[m, W] = y$ and run $\mathsf{Gen}_{sign}(\mathcal{PK}, W) \rightarrow \sigma$. Verify that $[m, W, \mathcal{PK}, \sigma, 0]$ is not recorded. If it is recorded, abort. Otherwise, record $[m, W, \mathcal{PK}, \sigma, 1]$. Return $(signature, \mathsf{sid}, \mathcal{PK}, W, m, y, \sigma)$ to P_i.

Malicious Requests of Signatures. upon receiving a message $(signature\_request, \mathsf{sid}, \mathcal{PK}, W, m)$ from Sim, obtain all existing valid signatures σ such that $[m, W, \mathcal{PK}, \sigma, 1]$ is recorded and add them in a list $\mathcal{L}_\sigma$. Return $(signatures, \mathsf{sid}, \mathcal{PK}, W, m, \mathcal{L}_\sigma)$ to Sim.

Ring VRF Verification. upon receiving a message $(verify, \mathsf{sid}, \mathcal{PK}, W, m, \sigma)$ from a party, do the following:

Cond.- 1 If there exits a record $[m, W, \mathcal{PK}, \sigma, b']$, set $b = b'$. (This condition guarantees the completeness and consistency.)

Cond.- 2 Else if $\mathtt{anonymous\_key\_map}[m, W]$ is an honest verification key and there exists a record $[m, W, \mathcal{PK}, \sigma', 1]$ for any σ', then let $b = 1$ and record $[m, W, \mathcal{PK}, \sigma, 1]$. (This condition guarantees that if m is signed by an honest party for the ring $\mathcal{PK}$ at some point, then the signature is $\sigma' \neq \sigma$ which is generated by the adversary is valid) Output $(verified, \mathsf{sid}, \mathcal{PK}, W, m, \sigma, \mathsf{Out}, b)$ to the party where $\mathsf{Out} = \mathtt{Out}[m, W]$ if $b = 1$. Otherwise, $\mathsf{Out} = \perp$.

Cond.- 3 Else relay the message $(verify, \mathsf{sid}, \mathcal{PK}, W, m, \sigma)$ to Sim and receive back the message $(verified, \mathsf{sid}, \mathcal{PK}, W, m, \sigma, b_{\mathsf{Sim}}, \mathsf{pk}_{\mathsf{Sim}}^{\mathsf{rvrf}})$. Then check the following:

(a) If $W \notin \mathcal{W}$ and $|\mathcal{W}[m, \mathcal{PK}]| > |\mathcal{PK}_{mal}|$ where $\mathcal{PK}_{mal}$ is a set of malicious keys in $\mathcal{PK}$, set $b = 0$. (This condition guarantees uniqueness meaning that the number of verifying outputs that Sim can generate for $m, \mathcal{PK}$ is at most the number of malicious keys in $\mathcal{PK}$.).

(b) Else if $\mathsf{pk}_{\mathsf{Sim}}^{\mathsf{rvrf}}$ is an honest verification key, set $b = 0$. (This condition guarantees unforgeability meaning that if an honest party never signs a message m for a ring $\mathcal{PK}$)

(c) Else if there exists $W' \neq W$ where $\mathtt{anonymous\_key\_map}[m, W'] = \mathsf{pk}_{\mathsf{Sim}}^{\mathsf{rvrf}}$, set $b = 0$.

(d) Else set $b = b_{\mathsf{Sim}}$.

In the end, record $[m, W, \mathcal{PK}, \sigma, b]$ if it is not stored. If $b = 0$, let $\mathsf{Out} = \perp$. Otherwise, do the following:

- if $W \notin \mathcal{W}[m, \mathcal{PK}]$, add W to $\mathcal{W}[m, \mathcal{PK}]$.
- if $\mathsf{pk}_{\mathsf{Sim}}^{\mathsf{rvrf}}$ is not recorded, record it in $\mathtt{verification\_keys}$ under Sim.
- if $\mathtt{Out}[m, W]$ is not defined, set $\mathtt{Out}[m, W] \leftarrow\$ \{0,1\}^{\ell_{\mathsf{rVRF}}}$, $\mathtt{anonymous\_key\_map}[m, W] = \mathsf{pk}_{\mathsf{Sim}}^{\mathsf{rvrf}}$. Set $\mathsf{Out} = \mathtt{Out}[m, W]$.
- otherwise, set $\mathsf{Out} = \mathtt{Out}[m, W]$.

Finally, output $(verified, \mathsf{sid}, \mathcal{PK}, W, m, \sigma, \mathsf{Out}, b)$ to the party and Sim.

Fig. 4. Functionality $\mathcal{F}_{\mathsf{rvrf}}$.

$\mathcal{F}^s_{\mathsf{rvrf}}$ for a relation $\mathcal{R}$ behaves exactly as $\mathcal{F}_{\mathsf{rvrf}}$. Differently, it has an algorithm Gen_π and it additionally does the following:

Secret Element Generation of Malicious Parties. upon receiving a message $(\mathit{secret\_rand}, \mathsf{sid}, \mathcal{PK}, \mathsf{pk}^{\mathsf{rvrf}}, W, m)$ from Sim, verify that $\mathtt{anonymous\_key\_map}[W] = (m, \mathcal{PK}, \mathsf{pk}_i^{\mathsf{rvrf}})$. If that was not the case, just ignore the request. If $\mathtt{secrets}[m, W, \mathcal{PK}]$ is not defined, obtain $y = \mathsf{Out}[m, W, \mathcal{PK}]$. Then, run $\mathsf{Gen}_\eta(m, \mathcal{PK}, \mathsf{pk}_i^{\mathsf{rvrf}}, y) \to \eta$ and store $\mathtt{secrets}[m, W, \mathcal{PK}] = \eta$. Obtain $\eta = \mathtt{secrets}[m, W, \mathcal{PK}]$ and return $(\mathit{secret\_rand}, \mathsf{sid}, \mathcal{PK}, W, \eta)$ to P_i.

Secret Random Element Proof. upon receiving a message $(\mathit{secret\_rand}, \mathsf{sid}, \mathcal{PK}, \mathsf{pk}^{\mathsf{rvrf}}, W, m)$ from P_i, verify that $\mathtt{anonymous\_key\_map}[W] = (m, \mathcal{PK}, \mathsf{pk}_i^{\mathsf{rvrf}})$. If that was not the case, just ignore the request. If $\mathtt{secrets}[m, W, \mathcal{PK}]$ is not defined, run $\mathsf{Gen}_\eta(m, \mathcal{PK}, \mathsf{pk}_i^{\mathsf{rvrf}}, y) \to \eta$ and store $\mathtt{secrets}[W, m] = \eta$. Obtain $\eta \leftarrow \mathtt{secrets}[m, W, \mathcal{PK}]$ and $y \leftarrow \mathtt{Out}[m, W, \mathcal{PK}]$. If $((m, y, \mathcal{PK}), (\eta, \mathsf{pk}_i^{\mathsf{vrf}})) \in \mathcal{R}$, run $\mathsf{Gen}_\pi(\mathcal{PK}, W, m) \to \pi$ and add π to a list $\mathtt{zkproofs}[m, W, \mathcal{PK}]$. Else, let π be $\perp$. Return $(\mathit{secret\_rand}, \mathsf{sid}, \mathcal{PK}, W, \eta, \pi)$ to P_i.

Secret Verification. upon receiving a message $(\mathit{secret\_verify}, \mathsf{sid}, \mathcal{PK}, W, m, \pi)$, relay the message to Sim and receive $(\mathit{secret\_verify}, \mathsf{sid}, \mathcal{PK}, W, m, \pi, \mathsf{pk}^{\mathsf{rvrf}}, \eta)$. Then,

- if $\pi \in \mathtt{zkproofs}[m, W, \mathcal{PK}]$, set $b = 1$.
- else if $\mathtt{secrets}[W, m] = \eta$ and $((m, y, \mathcal{PK}), (\eta, \mathsf{pk}_i^{\mathsf{vrf}})) \in \mathcal{R}$, set $b = 1$ and add to the list $\mathtt{zkproofs}[m, W, \mathcal{PK}]$.
- else set $b = 0$.

Send $(\mathit{verification}, \mathsf{sid}, \mathcal{PK}, W, m, \pi, b)$ to P_i.

Fig. 5. Functionality $\mathcal{F}^s_{\mathsf{rvrf}}$.