Sassafras: Efficient Batch Single Leader Election

Jeffrey Burdges[1], Elizabeth Crites[1], Handan Kılınç Alper[2], Alistair Stewart[1], and Sergey Vasilyev[1]

[1] Web3 Foundation
[2] =nil; Foundation
firstname@web3.foundation, handankilinc1@gmail.com

Abstract. In a single secret leader election (SSLE), a set of participants elect exactly one leader, who remains anonymous until they announce themselves by providing a proof. SSLE protocols are used in proof-of-stake blockchains to elect the leader who publishes the next block. Anonymity of the leader is an important security property, as the leader makes for an attractive target and may be subject to denial-of-service (DOS) attacks.

In this work, we propose a novel single leader election protocol, called *Sassafras*. We depart from the common approach of shuffling for constructing SSLE and instead employ a ring verifiable random function, which hides the identity of the leader within a ring of participants. Moreover, Sassafras is designed for *batch* leader elections, in which a single leader is selected for several elections at once. This allows the rate of leader election to match the rate of block production, an often-sought property not met by most SSLE protocols in the literature. We characterize single leader election with batching in the form of an ideal functionality in the Universal Composability (UC) framework and prove that Sassafras realizes this functionality. Sassafras is secure against an adaptive adversary, while achieving a slightly relaxed notion of anonymity for leaders. Sassafras features exceptionally low communication and computational complexity, outperforming other SSLE protocols by an order of magnitude or more.

1 Introduction

Leader election is an essential component of distributed systems, in particular in blockchain protocols to assign block producers. In the Byzantine setting, secrecy of the leaders is crucial for preventing denial-of-service (DOS) attacks. Most blockchain protocols [20,28,21,25,24,9] deploy secret *probabilistic* leader election (PLE), which elects leaders in expectation with respect to their power. Despite its efficiency, PLE has drawbacks due to its probabilistic nature: the election might result in multiple or no leaders, and the additional runoff procedure for selecting a leader increases the time needed to extend the blockchain. Moreover, PLE has been shown to achieve weaker blockchain security than *single* secret leader election (SSLE) [3], in which a exactly one leader is elected (see Section 1.1).

In this work, we present Sassafras, a *batch* single leader election protocol, in which a single leader is selected for several elections at once, and analyze its application as a replacement for PLE in existing blockchain protocols following the longest chain rule [20,28,21,25,24,9]. The batch capability of Sassafras enables scalable deployment as a leader election protocol for blockchain, as the rate of leader election aligns with the rate of block production. Moreover, Sassafras features exceptionally low communication and computational complexity. This is achieved by relaxing the secrecy achieved by some leaders, while maintaining enough anonymity to ensure security when deployed on a blockchain.

Overview of Sassafras. At a high level, Sassafras operates as follows. Parties first obtain a random number from a randomness beacon, and use it to generate n_t pseudorandom numbers, where n_t is the number of tickets. The parties then determine which numbers qualify as winning, and anonymously publish, verify, and agree on these winners. The numbers are sorted to determine the order of the secret leaders, with the position of a number indicating its leader's rank.

The core challenge in Sassafras is generating verifiable random numbers while maintaining anonymity. While a standard verifiable random function (VRF) [29] could be used, the verification algorithm requires the public key of the generating party, which reveals their identity. To avoid this, Sassafras employs a *ring* verifiable random function [10], which ensures that the pseudorandom output is unique and verifiable, as in a VRF, but without revealing from which party in a set of public keys $\mathcal{PK}$ it was generated. In more detail, parties in Sassafras run the ring VRF evaluation algorithm n_t[3] times with inputs $r||i$ for $1 \leq i \leq n_t$ after obtaining r from the randomness beacon. They check if the evaluation outputs y_i fall below a fixed threshold – these are the winning outputs. To prove y_i has been generated correctly without revealing their identity, a party signs $r||i$, resulting in a signature σ_i that can be verified by anyone with $\mathcal{PK}$, but without identifying the signer. The winning tickets and their signatures are then published.

At this point, parties need a method of publishing the winning outputs and their signatures anonymously. If they publish them in clear, then their anonymity is immediately broken, even if σ_i can be verified without their keys, because an adversary could observe the network traffic and learn the source of any message. Thus, Sassafras requires an extra anonymity layer. The goal is to have a randomly selected party, which we call a repeater, receive a party's ticket y_i, σ_i. This is accomplished as follows. First, the party hides y_i, σ_i and the repeater's identity by encrypting y_i, σ_i using a shared symmetric key with the repeater. Then, they multicast the encryption to all parties.[4] In the end, the repeater capable of decrypting the ticket publishes y_i, σ_i on behalf of the sender party.[5] If the

[3] We show how to choose the parameter n_t in Proposition 1. Specifically, as we discuss in Section 5, n_t should be at least 6 for $\ell = 2^{13}$ elections, under the assumption that the fraction of corrupt parties α is less than ≈ 0.3 with 2^{14} parties.

[4] Formally, Sassafras is secure in the $\mathcal{F}_{\mathsf{com}}^{s}$-hybrid model (Theorem 1), where $\mathcal{F}_{\mathsf{com}}^{s}$ is a secure diffusion functionality, which hides the message and the receiver. Here, we describe Sassafras with a simple and efficient instantiation using symmetric encryption. By "multicast the encryption to all parties," we mean via a standard diffusion functionality $\mathcal{F}_{\mathsf{com}}$. Details are given in Section 3.1.

[5] All repeaters publish all tickets at once (see Section 4).

repeater is malicious, the anonymity of the sender is compromised. Otherwise, anonymity is preserved because the owner of y_i is known by the honest repeater only, and other parties are unable to decrypt the ciphertext. In the end, the output of the Sassafras protocol consists of the y_i-values with their signatures released by all repeaters. The sorted y_i-values determine the order of the leaders.

Let us now provide intuition for why the anonymity guarantees of Sassafras, despite the potential leakage of some leaders' identities, are sufficient to ensure the security of a blockchain deploying Sassafras. Suppose there are up to $\alpha = 1/3$ corrupt parties and $2/3$ honest parties. For each election, one of these parties is randomly selected as the repeater. With probability at least $(2/3)(2/3) = 4/9$, both the ticket holder and the repeater will be honest. Thus, the identity of the honest leader is protected. With probability at least $(2/3)(1/3) = 2/9$, the ticket holder will be honest, but the repeater will be adversarial. In this case, the repeater learns the identity of the ticket holder and can attack her when she is due to be the next leader. (We later refer to this to as a "weakly corrupted" slot.) The ticket holder may be attacked by an adversary and unable to produce blocks; these slots do not contribute to the total number of blocks for reasoning about chain safety. For each slot, the probability that the adversary produces a block is at most $1/3$, and the probability that an honest party produces a block is at least $4/9$ ($> 1/3$). In expectation, there will be more honest blocks appended to the current longest chain than adversarial blocks. From there, following an adaptation of the security proof for PLE-based proof-of-stake blockchains [21,4,9,30], which intuitively works by arguing that, by Chernoff bounds, for any long enough sequence of slots, there will be more honest blocks than adversarial ones, then we can find a k such that all public chains will include the same blocks that are more than k slots old. Thus, we see that even if an adversary can corrupt up to $1/3$ of parties, the longest chain rule is not violated. A formal analysis is given in Section 4.2.

Our Contributions. The contributions of this work are as follows:

- We propose a novel batch SLE protocol, Sassafras, in which a single leader is selected for several elections at once. Sassafras is secure against adaptive corruption, while achieving a slightly relaxed notion of anonymity for leaders.
- To capture formally the properties of batch SLE, we define an ideal functionality $\mathcal{F}_{\mathsf{bsle}}$, which offers modularity and flexibility for batch SLE in the Universal Somposability (UC) framework. In particular, $\mathcal{F}_{\mathsf{bsle}}$ may be instantiated so that it is tailored to different unpredictability levels for leaders, ranging from perfect unpredictability of the leader to partial or no unpredictability, depending on the requirements of the real-world protocol. We prove that Sassafras realizes $\mathcal{F}_{\mathsf{bsle}}$ in the presence of an adaptive adversary.
- We show how Sassafras can be deployed on a blockchain to elect block producers. In particular, we show how PLE can be replaced with Sassafras, maintaining chain safety and scalability and achieving faster block finality. To the best of our knowledge, Sassafras is the most efficient SLE protocol both in terms of computational and communicational overhead to date. We provide a concrete efficiency analysis in Table 2 and Section 5.

	Protocol	Comm.	Comp.	Security			Primitives
		On-chain	On-chain	Public	Anon. Set	Model	
Single Election	IO-based [7]	$O(1)$	$O(1)$	–	$(1-\alpha)n$	SM, static, ✓	IO
	TFHE-based [7]	$O(\alpha n)$	$O(\alpha n)$	–	$(1-\alpha)n$	SM, static, ✓	LWE, TFHE
	Shuffle-1 [7]	$O(\sqrt{n})$	$O(\sqrt{n})$	–	$(1-\alpha)\sqrt{n}$	static, ✗	DH-based Shuffle
	Shuffle-2 [17]	$O(n)$	$O(n)$	–	$(1-\alpha)n$	UC, static, ✓	DH-based Shuffle
	Shuffle-3 [17]	$O(n)$	$O(n)$	–	$(1-\alpha)n$	UC, adaptive, ✓	DH-based Shuffle
	PEKS-based [18]	$O(\alpha n)$	$O(\log^2 n)$	–	$(1-\alpha)n$	UC, static, ✓	Functional Encryption
Batch Election	WHISK [22]	$O(\sqrt{n})$	$O(\sqrt{n})$	$-1.25\frac{\ln(1-\alpha)}{\sqrt{n}}n$	$(1-\alpha)\sqrt{n}$	– , ✗	DH-based Shuffle
	Sassafras	$\mathbf{O(1)}$	$\mathbf{O(1)}$	$(1-\alpha)\alpha n$	$\frac{(1-\alpha)n}{2}$	**UC, adaptive**	**Ring VRF**

Table 1. Single secret leader election (SSLE) protocols. n is the total number of participants, α is the fraction of corrupt parties, and $(1-\alpha)$ is the fraction of honest parties. Batch Election refers to running ℓ elections at once, and values are given per election. Comm. and Comp. are communication and computational complexity, respectively. Public is the fraction of leaders in a batch election that are leaked, and the unpredictability level of a leader (Definition 5) is the reciprocal, e.g., $\frac{1}{(1-\alpha)\sqrt{n}}$. For Sassafras, this is calculated in Proposition 1. SM is the Standard Model and UC is the Universal Composability model. ✗ means the protocol is insecure under our threat model, as described below; ✓means it is secure in our model. The security model for WHISK [22] is marked '–' because no formal security analysis is provided. Other terms are defined below.

1.1 Related Work

Single Secret Leader Election (SSLE): Most SSLE protocols [7,17,22] rely on (repeated) shuffling a commitment list, which incurs high overhead due to complex shuffle proofs [5] and makes them impractical for on-chain use. Boneh et al. [7] suggest shuffling smaller sublists to reduce overhead, but this also reduces the unpredictability of the leader (discussed further below). Another issue in some SSLE protocols [7,17] is the lack of an efficient design for *sequential* leader election, which is crucial in blockchain protocols for sequential block producing. Shuffle-based SSLE requires first shuffling the list, then finalizing it on-chain, and only afterward obtaining randomness from a beacon. Since these steps must be executed in sequence, a leader involved in shuffling could influence the process if they were to gain knowledge of the randomness too early. Existing randomness beacons [21,28,16,31] are too slow for the required sequentiality. For example, Drand [23] produces fresh randomness only every 30 seconds, but some blockchains produce a block every couple of seconds [9], or even multiple blocks per second [2,6]. WHISK [22], a shuffle-based approach to SSLE following [7] and proposed for Ethereum[6], solves this issue by performing *batch* leader elections through a single randomness, as does Sassafras. Other approaches to SSLE, including functional encryption (e.g., public key encryption with keyword search (PEKS) [18]), indistinguishability obfuscation (IO) [7], and threshold fully homo-

[6] Ethereum currently deploys public leader election, where all leaders are known. They also assume $\alpha < 1/3$, as does every chain using traditional Byzantine Agreement that is asynchronously safe (e.g., Cosmos, Aptos, Sui, which also currently deploy public leader election). Thus, Sassafras makes an attractive option for deployment in most proof-of-stake blockchains.

morphic encryption (TFHE) [7], are impractical due to the heavy cryptographic operations required.

We provide a comparison of SSLE protocols in Table 1, categorizing single ($\ell = 1$) and batch ($1 \leq \ell < n$) single leader election protocols. Communication complexity refers to the message sizes added to blockchain blocks for ℓ elections, while computational complexity covers the verifications within each block. Larger message sizes and complex verification algorithms slow down blockchain scalability by increasing propagation time. Thus, we focus on the verification cost and message overhead from the election mechanism. Table 1 shows the practicality of batch elections, as single-election protocols must run ℓ times to achieve the same result.

Protocol	**Comm. On-chain**	**Comp. On-chain**
Shuffle-1 [7]	123.7 MB	40.9 ms
Shuffle-2 [17]	4831.84 MB	5241.9 ms
Shuffle-3 [17]	17718.31 MB	11793.6 ms
PEKS-based[18]	301.47 MB	16.9 ms
WHISK [22]	90.44 MB	20 ms
Sassafras	7.64 MB ✓	7.81 ms ✓

Table 2. The performance of existing SSLE protocols on a blockchain (See Section 5).

Unpredictability of Batch SLEs: We now give a detailed comparison of the unpredictability levels achieved by WHISK and Sassafras under the assumption that the adversary can corrupt up to 1/3 of parties. These can be a combination of corrupted and weakly corrupted parties, captured by $\alpha + \alpha_w \leq 1/3$, where α_w is the maximum fraction of weak corruptions (e.g., DOS-type attacks). (See Section 4.2 for more details.) WHISK, like Sassafras, permits a fraction of leaders to be exposed, as seen in Table 1 (under "Public"). Out of one million validators on Ethereum, the proposed number of validators to run the WHISK protocol is $n = 2^{14} = 16384$. Now consider when $\alpha = \alpha_w = 1/6$, so that $\alpha n = (1/6)(16384) = 2730$ parties can be corrupted, and 2730 parties can be weakly corrupted. Sassafras results in $(1-\alpha)\alpha n = (1-1/6)(1/6)(16384) = 2276$ leaked leaders, but the anonymity set is of size $(1-\alpha)n/2 = 6826$, so the leaders remaining achieve good privacy, and, as discussed in the introduction, the leaked leaders still allow consensus to succeed. In contrast, WHISK only leaks $\frac{-1.25\ln(1-\alpha)}{\sqrt{n}}n = 29$ leaders, but the anonymity set is only of size $(1-\alpha)\sqrt{n} = 107$ for the remaining honest leaders. That is, all honest leaders in the set of 16384 parties can be easily targeted for weak corruption. Indeed, it is in the adversary's interest to simply weakly corrupt all $1/3n = 5460$ parties in its "corruption budget." Thus, WHISK is insecure under our threat model, as is the single election protocol of [7]. The main purpose of deploying SSLE protocols in blockchains is to mitigate against DOS attacks, which our model accounts for strongly, and Sassafras achieves by design.

Moreover, WHISK targets efficiency over other SSLE protocols by performing $\ell < n$ batch elections with $O(\sqrt{n})$ communication and computational complexity. Sassafras's efficiency surpasses that of WHISK by performing an arbitrary number (not restricted to $\ell < n$) of batch elections with $O(1)$ communication and computational complexity. A detailed efficiency comparison with WHISK can be found in Table 2 and Section 5.

Proof-of-Stake Protocols: As mentioned in the introduction, existing proof-of-stake (PoS) protocols that elect secret leaders are based on probabilistic leader election (PLE) [30,20,28,21,19], in which multiple leaders may be elected for a slot, or no leaders at all. The additional runoff procedure for selecting a leader increases the time needed to extend the blockchain. Concretely, Azouvi and Cappelleti [3] show that the block finality time for a PoS protocol is decreased by 25% when SSLE is used versus PLE. Even more important, for an adversary who controls at most 1/3 of the parties, SSLE provides higher security than PLE against private attacks, a damaging attack where an adversary can create a private chain that overtakes the honest chain, breaking the security of a longest chain protocol. Specifically, they show that the common prefix parameter (how many blocks back all longest chains have the same prefix) against private attacks in a synchronous network with a perfect (i.e., unbiased) randomness beacon is decreased by roughly 25% against a malicious stake ratio α of 1/3 or 1/4. The same parameter decreases **70%** with an imperfect randomness beacon. While PLE-based PoS protocols provide security with more adversarial power (up to 1/2), they are significantly slower (25%) to finalize the blocks, while Sassafras provides security with less adversarial power (up to 1/3) but finalizes the blocks significantly faster.

2 Preliminaries

2.1 State Machine Replication (SMR)

Sassafras uses an SMR protocol to agree on a linearizable log[7]. In particular, this ensures that all parties reach consensus on the array of leaders. In SMR (Algorithm 1), each party maintains a replica of a deterministic state machine that updates an ordered set LOG using the algorithm TRANSITION, which processes messages from the network. Time is divided into sequential slots [28,21,9,20], and SMR accesses a clock $\mathcal{F}_{\mathsf{clock}}$ [4,1,13,26] that increments slots monotonically. Parties update LOG whenever the slot changes. A command $cmd \in \mathsf{LOG}$ is considered final once marked as such by the algorithm TRANSITION.

Algorithm 1 STATEMACHINE

$\mathsf{LOG} = \emptyset, \mathtt{input} = \emptyset, \mathsf{Bucket} = \emptyset$
$\mathsf{sl} \leftarrow \mathcal{F}_{\mathsf{clock}}$ // learns the current slot sl from $\mathcal{F}_{\mathsf{clock}}$
run PROCESS1 **and** PROCESS1 in parallel

PROCESS1
while true **do**
  $\mathsf{sl}_c \leftarrow \mathcal{F}_{\mathsf{clock}}$
  if $\mathsf{sl}_c \geq \mathsf{sl}$ **then**
    $cmds \leftarrow \mathsf{Bucket}$
    clean Bucket
    $\mathsf{sl} = \mathsf{sl}_c + 1$
    $\mathtt{input}.\text{APPEND}(cmds||\mathsf{sl}_c)$
    $\mathsf{LOG} = \text{TRANSITION}(\mathsf{LOG}, \mathtt{input})$

PROCESS2
while true **do**
  fetch messages from the network (Figure 2)
  receive $cmds$ from the network
  add $cmds$ to Bucket

[7] Linearizability means orderability of logs.

Definition 1 (Persistence). *Persistence guarantees that if cmd* $\in \mathsf{LOG}$ *is final in an honest state machine, then it is going to be final in other state machines in* Δ_{pers} *slots.*

Definition 2 (Liveness). *Liveness guarantees that any command cmd sent by an honest party will be included in* LOG *of all honest parties in* Δ_{live} *slots.*

Definition 3 (Security). *SMR is secure if it has the persistence and liveness properties i.e., it guarantees that cmd sent by an honest party becomes final in* $\Delta_{sec} = \Delta_{\mathsf{live}} + \Delta_{\mathsf{pers}}$ *slots.*

2.2 Universal Composability (UC) Framework

The security of Sassafras is proven in the Universal Composability (UC) framework [11,12]. In the UC framework, protocols π are modeled as ideal functionalities $\mathcal{F}$, with security defined by how well real-world protocols emulate these ideal functionalities, regardless of the environment $\mathcal{Z}$ or other composed protocols. Protocols π are constructed using distributed interactive Turing machines (ITMs), each with an inbox for messages from other ITMs, the adversary $\mathcal{A}$, or the environment $\mathcal{Z}$, which can activate ITMs anytime. An ITM instance (ITI) is identified by session and party identifiers $(\mathsf{sid}, \mathsf{pid})$. The environment $\mathcal{Z}$ oversees and controls communication between parties and $\mathcal{A}$, gathering insights for interactions with $\mathcal{A}$. In the *real world*, $\mathcal{Z}$ initiates ITMs and $\mathcal{A}$ to execute π with input $z \in \{0,1\}^*$ and security parameter λ, producing the output $\mathsf{EXEC}(\lambda, z)_{\pi,\mathcal{A},\mathcal{Z}}$. In the *ideal world*, a trusted entity $\mathcal{F}$ runs π, with the simulator Sim forwarding messages between $\mathcal{Z}$ and $\mathcal{F}$, and outputting $\mathsf{EXEC}(\lambda, z)_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}$.

Definition 4. *(UC-security of π[11,12]) Let π be a protocol executed in the real-world protocol and in the ideal world with the functionality $\mathcal{F}$. We say that π in the real world realizes $\mathcal{F}$ if for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$ in the real world, there exists a PPT simulator* Sim *such that for any environment $\mathcal{Z}$, the ensembles of the output of $\mathcal{Z}$ are indistinguishable, i.e.,* $\{\mathsf{EXEC}(\lambda, z)_{\pi,\mathcal{A},\mathcal{Z}}\}_{z\in\{0,1\}^*} \approx \{\mathsf{EXEC}(\lambda, z)_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}\}_{z\in\{0,1\}^*}$. *We call the real-world protocol UC secure if it realizes $\mathcal{F}$.*

A protocol π operates in *the hybrid world* when it runs in the real world but interacts with ideal functionalities.

3 Batch Single Leader Election

We formally define batch single leader election, which outputs single leaders for at least $\ell_{\min}$ elections. We define a functionality $\mathcal{F}_{\mathsf{bsle}}$ (Figure 1), which models batch SLE in the ideal world with a dynamic set of parties $\mathcal{P}$. Thus, it supports changes in the set of parties during its execution. The environment $\mathcal{Z}$ creates parties and controls the adversary Sim according to the UC framework

(Section 2.2). Each party $\mathsf{P}_i \in \mathcal{P}$ is assigned power pw_i, with the power distribution $\mathcal{D}[\mathsf{P}_i] = \frac{\mathsf{pw}_i}{\sum_{\mathsf{P}_j \in \mathcal{P}} \mathsf{pw}_j} = \rho_i$, representing the likelihood of P_i becoming a leader. Sim can adaptively corrupt up to an α-fraction of the total power, i.e., $\sum_{\mathsf{P}_i \in \mathcal{P}_\mathcal{A}} \mathcal{D}[\mathsf{P}_i] < \alpha$, where $\mathcal{P}_\mathcal{A}$ is the set of corrupted parties and $\beta = 1 - \alpha$.

An intuitive $\mathcal{F}_{\mathsf{bsle}}$ would randomly select one leader for each of the ℓ-elections and notify *only* the leader, ensuring fairness, unpredictability, robustness, and uniqueness (defined below). However, in real-world *batch* SLE, the leaders of ℓ-elections are not independent, unlike in single SLE, where elections are run separately ℓ times. This interdependence complicates efficient protocol design while preserving leader unpredictability. Therefore, we define $\mathcal{F}_{\mathsf{bsle}}$ to be adaptable to various leader unpredictability levels. It is modular, either resembling the intuitive $\mathcal{F}_{\mathsf{bsle}}$, with perfect unpredictability of leaders, or allowing more flexible designs for efficiency where anonymity of honest leaders is not perfect. We capture this by introducing an algorithm GetClue, whose specification depends on the desired unpredictability level of the batch SLE protocol.

[Batch Election in $\mathcal{F}_{\mathsf{bsle}}$]: $\mathcal{F}_{\mathsf{bsle}}$ receives a request of a batch election $\mathsf{eid} = (e, \mathcal{P}, \mathcal{D})$ from Sim. Here, eid is an identification of the election, where e is an identifier and $\mathcal{P}$ is the set of parties with power distribution $\mathcal{D}$. We remark that $\mathcal{P}$ can be specified by Sim arbitrarily. Thus, it can adapt to dynamically changing parties for each election. Recall that $\mathcal{P}$ and $\mathcal{D}$ must satisfy the assumption that $\sum_{\mathsf{P}_i \in \mathcal{P}_\mathcal{A}} \mathcal{D}[\mathsf{P}_i] < \alpha$ during the batch election.[8] After the request, $\mathcal{F}_{\mathsf{bsle}}$ runs Elect(eid), which is an algorithm that randomly selects a leader with respect to their power for each election and returns the leaders list $\mathtt{Leads}$. Elect is defined with fixed parameters $n_t \in \mathbb{N}$, the number of tickets, and $c \in (0, 1]$, used to determine the threshold for winning tickets. In Elect in Figure 1, each party P_i has a threshold value $\tau_i = c\frac{\mathcal{D}[\mathsf{P}_i]}{\rho_{\max}}$, where $\rho_{\max} = \max\{\mathcal{D}[\mathsf{P}]\}_{\mathsf{P} \in \mathcal{P}}$. We note that τ_i is proportional to the power of a party P_i, which is $\rho_i = \mathcal{D}[\mathsf{P}_i]$, and if $\mathcal{D}$ is an equal power distribution, then $\tau_i = c$ for all P_i.

Elect selects n_t random elements in $[0, 1)$ for each party P_i and checks if they are less than the threshold c. If it is the case, P_i is added to a list $\mathcal{L}_i$ (see Lines 4-6 of GetClue). We note that c can be interpreted as a leader cutoff parameter, as it directly controls the expected number of random elements passing their respective thresholds across all parties. A smaller c increases the difficulty of selection, thereby reducing the number of potential leaders, while a larger c increases the selection probability. However, c does not affect the distribution of leaders in $\mathtt{Leads}$, as the probability that $\mathtt{Leads}[i] = \mathsf{P}_j$ is:

$$\Pr[\mathtt{Leads}[i] = \mathsf{P}_j] = \frac{\tau_j}{\sum_{\mathsf{P}_k \in \mathcal{P}} \tau_k} = \frac{\mathcal{D}[\mathsf{P}_j]}{\sum_{\mathsf{P}_k \in \mathcal{P}} \mathcal{D}[\mathsf{P}_k]} = \mathcal{D}[\mathsf{P}_j],$$

where $\mathcal{D}[\mathsf{P}_k] = \frac{\mathsf{pw}_k}{\sum_{\mathsf{P}_\ell \in \mathcal{P}} \mathsf{pw}_\ell}$. This shows that the leader selection process respects the distribution $\mathcal{D}$, ensuring fairness across all parties.

[8] While $\mathcal{F}_{\mathsf{bsle}}$ could enforce this assumption directly, we choose to simplify the design by limiting Sim to respect this constraint.

$\mathcal{F}_{\mathsf{bsle}}$ is defined over parameters $\texttt{max-elect}, \ell_{\min}, n_t \in \mathbb{N}$, $c \in (0,1]$, $\gamma \in [0,1]$ and a PPT algorithm GetClue, which receives as an input an array and outputs a same-sized array including elements from clue domain $\mathcal{C}$. Each batch election is identified with $\mathsf{eid} = (e, \mathcal{P}, \mathcal{D})$. $\texttt{Leads}$ stores leaders of elections where the set of parties $\mathcal{P}$ joining the batch election specified by Sim. $\mathcal{F}_{\mathsf{bsle}}$ keeps the information about each batch election in DB.
[Register.] Upon receiving a message $(\textit{register}, \mathsf{sid}, \mathsf{P})$ from P, $\mathcal{F}_{\mathsf{bsle}}$ adds to the set of honest parties $\mathcal{P}_H$ and sends $(\textit{registered}, \mathsf{sid}, \mathsf{P})$ to Sim.
[Batch Election.] Upon receiving a message $(\textit{elect}, \mathsf{sid}, \mathsf{eid})$ from Sim, if $\mathsf{DB}[\mathsf{eid}]$ is defined, proceed to the the distribution phase below. Otherwise, set an empty list candidate election $\mathcal{C}$, let $\mathsf{counter} = 1$, and set $\mathsf{isDecided} = \mathsf{false}$. **Do the following until** $\mathsf{isDecided} = \mathsf{true}$:

- Run Elect$(\mathsf{eid}) \to \texttt{Leads}$ below:

  Elect(eid):
  for $\mathsf{P}_i \in \mathcal{P}$ **do**
  $\quad \tau_i \leftarrow \frac{c\rho_i}{\rho_{\max}}$
  $\quad \mathcal{L}_i \leftarrow \emptyset$
  $\quad$ **for** $k \in [1, n_t]$ **do**
  $\quad\quad \omega_k \leftarrow\!\$\ [0,1)$
  $\quad\quad$ **if** $\omega_k < \tau_i$: **add** P_i **to** $\mathcal{L}_i$ **then**
  return $\texttt{Leads} \leftarrow$ Shuffle$(\cup_{i=1}^{n} \mathcal{L}_i)$

- Obtain $\texttt{Clue} \leftarrow$ GetClue$(\texttt{Leads})^*$.
- Send $(\textit{candidate}, \mathsf{sid}, \mathsf{eid}, \texttt{Leads}_m, \mathcal{I}_m)$ to Sim. Here, $\mathcal{I}_m$ the list of elections that have a malicious leader and $\texttt{Leads}_m[i] = \texttt{Leads}[i]$ for all $i \in \mathcal{I}_m$.
- Add $(\texttt{Leak}, \texttt{Clue}, \texttt{Leads})$ to $\mathcal{C}$ and $(\mathsf{eid}, \mathsf{counter}, \{k : \texttt{Leads}[k] = \mathsf{P}_i\})$ to $\texttt{history}[\mathsf{P}_i]$ for all $\mathsf{P}_i \in \mathcal{P}_H$. Set $\mathsf{DB}[\mathsf{eid}] = (\texttt{Clue}, \texttt{Leads})$.
  - **If** $\mathcal{F}_{\mathsf{bsle}}$ receives $(\textit{re-elect}, \mathsf{sid}, \mathsf{eid})$ from Sim and $\mathsf{counter} < \texttt{max-elect}$, increment $\mathsf{counter}$.
  - **Else if** $\mathcal{F}_{\mathsf{bsle}}$ receives $(\textit{set}, \mathsf{sid}, \mathsf{eid}, \mathcal{P}, c)$ from Sim where $c \leq \mathsf{counter}$, $\mathsf{DB}[\mathsf{eid}] = \mathcal{C}[c]$ and set $\mathsf{isDecided} = \mathsf{true}$.
  - **Else:** Set $\mathsf{isDecided} = \mathsf{true}$
- When $\mathsf{isDecided} = \mathsf{true}$, start the distribution phase.

[Distribution.] Get $\texttt{Clue}, \texttt{Leads}$ from $\mathsf{DB}[\mathsf{eid}]$. Populate the array $\texttt{Leak}$ of size $|\texttt{Leads}|$ as follows: for each $\texttt{Leak}[i]$ where $\texttt{Leak}[i] \in \mathcal{P}_H$, get $b \leftarrow_\gamma \{0,1\}$ and if $b = 1$, let $\texttt{Leak}[i] = \texttt{Leads}[i]$. In the end, send $(\textit{result}, \mathsf{sid}, \mathsf{eid}, \texttt{Leak}, \texttt{Clue}, \texttt{Leads}_m, \mathcal{I}_m)$ to Sim. Then receive $(\textit{abort}, \mathsf{sid}, \mathsf{eid}, \mathcal{L}_{\mathsf{abort}})$ from Sim, where $\mathcal{L}_{\mathsf{abort}} \subseteq \mathcal{I}_m$. If $|\texttt{Leads}| - |\mathcal{L}_{\mathsf{abort}}| < \ell_{\min}$, abort. Otherwise, for all $k \in \mathcal{L}_{\mathsf{abort}}$, remove $\texttt{Leads}[k]$ from $\texttt{Leads}$. Send $(\textit{result}, \mathsf{sid}, \mathsf{eid}, \{k : \texttt{Leads}[k] = \mathsf{P}_i\})$ to each $\mathsf{P}_i \in \mathcal{P}_H$.
[Reveal.] Upon receiving a message $(\textit{reveal}, \mathsf{sid}, \mathsf{eid}, \mathsf{P}_i, k)$ from P_i, obtain $\texttt{Leads}$ from $\mathsf{DB}[\mathsf{eid}]$. If $\texttt{Leads}[k] = \mathsf{P}_i$ and the batch phase of eid is done, send $(\textit{reveal}, \mathsf{sid}, \mathsf{eid}, \mathcal{P}, \mathsf{P}_i, k)$ to all $\mathsf{P} \in \mathcal{P}$.
[Corrupt.] Upon receiving a message $(\textit{corrupt}, \mathsf{sid}, \mathsf{P}_i)$, move P_i from $\mathcal{P}_H$ to $\mathcal{P}_{\mathcal{A}}$. Send $(\textit{corrupted}, \mathsf{sid}, \texttt{history}[\mathsf{P}_i])$ to Sim.

Fig. 1. $\mathcal{F}_{\mathsf{bsle}}$ running Elect.

After running ELECT, $\mathcal{F}_{\mathsf{bsle}}$ runs the algorithm GETCLUE, which outputs a list $\texttt{Clue}$ in order of leadership where $|\texttt{Clue}| = |\texttt{Leads}|$. $\texttt{Clue}$ stores some clues about the honest leaders, i.e., if $\texttt{Leads}[k] \in \mathcal{P}_H$, $\texttt{Clue}[k]$ is some clue in the clue domain $\mathcal{C}$. Otherwise, $\texttt{Clue}[k] = \perp$. As an example, $\texttt{Clue}[k]$ can be the number of times that an honest leader of the election k is a leader within ℓ-elections or a subset of honest parties that includes $\texttt{Leads}[k]$.

After running ELECT, $\mathcal{F}_{\mathsf{bsle}}$ gives the list of elections $\mathcal{I}_m$ with malicious leaders and their malicious leaders $\texttt{Leads}_m$, where $\texttt{Leads}_m[i] = \texttt{Leads}[i]$ for all $i \in \mathcal{I}_m$ and for all malicious leaders in $\texttt{Leads}$. If Sim disagrees with the result, $\mathcal{F}_{\mathsf{bsle}}$ reruns ELECT, repeating the process until Sim agrees, or after at most $\texttt{max-elect}$ attempts. After the agreement, $\mathcal{F}_{\mathsf{bsle}}$ initiates distribution of the result.

We remark that even though the parameter c does not have any effect on the fair leader election, it is included because it plays plays a crucial role in controlling the anonymity of leaders in Sassafras (see Lemma 2). Therefore, it might give the impression that the functionality is tailored specifically to Sassafras. However, having c in $\mathcal{F}_{\mathsf{bsle}}$ does not affect the generality of $\mathcal{F}_{\mathsf{bsle}}$, as it can be always set to 1, effectively disabling its influence.

[Distribution:] In the distribution phase, $\mathcal{F}_{\mathsf{bsle}}$ generates the array $\texttt{Leak}$ by letting $\texttt{Leak}[i] = \texttt{Leads}[i]$ with probability γ, which is a fixed parameter of $\mathcal{F}_{\mathsf{bsle}}$. Note that if $\gamma = 0$, no honest leaders are revealed to Sim with the array $\texttt{Leak}$. $\mathcal{F}_{\mathsf{bsle}}$ then gives $\texttt{Leak}$, $\texttt{Clue}$, and the list $\texttt{Leads}_m, \mathcal{I}_m$ to Sim. Sim then sends a set $\mathcal{L}_{\mathsf{abort}} \subset \mathcal{I}_m$, indicating which malicious leaders are aborted. If $|\texttt{Leads}| - |\mathcal{L}_{\mathsf{abort}}| < \ell_{\min}$, $\mathcal{F}_{\mathsf{bsle}}$ aborts. Otherwise, it removes the leaders in $\mathcal{L}_{\mathsf{abort}}$ and updates the leadership array to size $\ell - |\mathcal{L}_{\mathsf{abort}}|$. Finally, it informs the honest parties of their leadership roles.

We remark that $\texttt{Clue}$ and $\texttt{Leak}$ serve different purposes here: $\texttt{Leak}$ reveals leaders of some elections, while $\texttt{Clue}$ gives extra information about who the leader could be, i.e., $\Pr[\texttt{Leads}[i] = \mathsf{P}_j | \texttt{Clue}[i]] < \Pr[\texttt{Leads}[i] = \mathsf{P}_j | \texttt{Leak}[i] = \mathsf{P}_j] = 1$.

[Reveal:] If P reveals its leadership for the ith election, it informs $\mathcal{F}_{\mathsf{bsle}}$. $\mathcal{F}_{\mathsf{bsle}}$ sends a message to other parties saying P is the leader of i if $\texttt{Leads}[i] = \mathsf{P}$.

[Corrupt:] When Sim corrupts P_i, $\mathcal{F}_{\mathsf{bsle}}$ adds it to a list of corrupted parties and gives information related to P_i in all previous batch elections that P_i joined.

We now describe the security properties that $\mathcal{F}_{\mathsf{bsle}}$ satisfies with respect to the algorithm GETCLUE.

Fairness: The leaders list $\texttt{Leads}$ output by ELECT ensures $\Pr[\texttt{Leads}[i] = \mathsf{P}_j] = \mathcal{D}[\mathsf{P}_j]$ for all $\mathsf{P}_j \in \mathcal{P}$, meaning the probability of being selected as a leader is proportional to the weight in $\mathcal{D}$. During the distribution phase, the Sim can abort malicious leaders, altering $\Pr[\texttt{Leads}[i] = \mathsf{P}_j]$ for adversarial parties. However, this does not reduce the likelihood of honest parties being selected, as only malicious leaders are removed, leaving honest leaders unaffected. Additionally, the relative proportion of honest leaders in $\texttt{Leads}$ increases, preserving fairness for honest parties. Thus, $\mathcal{F}_{\mathsf{bsle}}$ maintains fairness across elections, even under adversarial interference.

Robustness: $\ell_{\min}$-robustness means that $\mathcal{F}_{\mathsf{bsle}}$ does not abort, i.e., $|\texttt{Leads}| - |\mathcal{L}_{\mathsf{abort}}| < \ell_{\min}$. Intuitively, the parameters n_t and c should be selected in such a way that, even if all malicious leaders abort their leadership during the distribution phase, $\mathcal{F}_{\mathsf{bsle}}$ does not abort. Therefore, we have the following Lemma to ensure robustness with the right parameter selections for $\mathcal{F}_{\mathsf{bsle}}$:

Lemma 1. *$\mathcal{F}_{\mathsf{bsle}}$ is $\ell_{\min}$-robust except with the probability* $\exp(-\frac{(r-1)^2}{2r}\ell_{\min})$*, where* $r = \frac{cn_t\beta}{\rho_{\max}\ell_{\min}} > 1$ *and* $\beta = 1 - \alpha$*.*

The proof is given in Appendix B. We note that malicious leaders may abort their leadership after the distribution phase, as their behavior cannot be controlled. However, such aborts do not affect the leadership of honest leaders. In contrast, aborts during the distribution phase can impact honest leaders, as $\mathcal{F}_{\mathsf{bsle}}$ aborts if it cannot ensure at least $\ell_{\min}$ leaders. Consequently, the robustness of $\mathcal{F}_{\mathsf{bsle}}$ is directly tied to ensuring sufficient honest leaders during the distribution phase.

Uniqueness: Since Elect assigns a *single* leader for each election, $\mathcal{F}_{\mathsf{bsle}}$ satisfies uniqueness.

Unpredictability: This property quantifies the probability that an adversary can correctly guess the honest leader of an election [7], given its view. Defining unpredictability in batch elections is more complex than in single leader elections due to the interdependence between elections. For instance, in WHISK [22], when leaders reveal their leadership, the set of possible leaders for subsequent elections becomes smaller, increasing the simulator's chance of guessing the correct leader. A similar situation arises in Sassafras, where revealed or leaked leaderships influence the unpredictability of other elections.

In $\mathcal{F}_{\mathsf{bsle}}$, elections corresponding to honest parties in the `Leak` array do not provide unpredictability, as their leadership is explicitly exposed. The unpredictability of non-leaked honest leaders depends on the `Clue` and `Leak` arrays, as well as the set of honest leaders revealed during the reveal phase. To address these dependencies, we formally define the unpredictability of a leader with respect to the information provided by GetClue in $\mathcal{F}_{\mathsf{bsle}}$ below.

Definition 5 (θ-Unpredictability). *Let $\mathcal{F}_{\mathsf{bsle}}$ run a **specified** GetClue algorithm, and let* `Leads` *be an output of the* Elect *run in $\mathcal{F}_{\mathsf{bsle}}$. $\mathcal{F}_{\mathsf{bsle}}$ satisfies θ-unpredictability if for all elections s such that s has not been revealed in the reveal phase of $\mathcal{F}_{\mathsf{bsle}}$ and s is not leaked by* `Leak`*, the probability of correctly guessing the honest leader* $\texttt{Leads}[s]$ *for all simulators* Sim *is bounded by Θ. Specifically, for all honest parties* P*,* $\Pr\left[\texttt{Leads}[s] = \mathsf{P} \,\middle|\, \texttt{R}, \texttt{Clue}, \texttt{Leak}, \texttt{Leads}[s] \notin \mathcal{P}_{\mathcal{A}}\right] \leq \Theta$*, where* `R` *is an array that records which parties have revealed their leadership and* `Clue` *is the output of* GetClue(`Leads`)*.*

This definition ensures that the adversary's ability to guess the honest leader of an election is strictly bounded, even given the adversary's view consisting of `R`, `Clue`, and `Leak`.

Modularity of $\mathcal{F}_{\mathsf{bsle}}$: $\mathcal{F}_{\mathsf{bsle}}$ defines a batch SLE protocol with fairness, robustness, uniqueness and θ-unpredictability properties. The level of unpredictability θ achieved by $\mathcal{F}_{\mathsf{bsle}}$ strongly depends on the design of the abstract algorithm GetClue, which determines how `Clue` is generated. If the anonymity of an honest party is not a concern, GetClue can be designed such that $\theta = 1$, fully revealing leader information. In general, to meet the anonymity requirements of a batch SLE protocol, GetClue must be carefully defined to provide the maximum allowable information as quantified by the parameter θ.

Assume that $\mathcal{F}_{\mathsf{bsle}}$ is instantiated with a particular γ parameter and a **specific** GetClue algorithm, achieving θ-unpredictability with $\theta = \theta'$ under the given γ and GetClue. Its parameters $\ell_{\min}, \texttt{max-elect}, n_t, c$ are selected in accordance with Lemma 1. In this setting, any protocol π that securely realizes $\mathcal{F}_{\mathsf{bsle}}$ using the specified GetClue and parameters in the UC framework (Definition 4) inherits the fairness, robustness, uniqueness, and θ-unpredictability properties of $\mathcal{F}_{\mathsf{bsle}}$, where $\theta = \theta'$.

An instantiation of $\mathcal{F}_{\mathsf{bsle}}$ achieves perfect unpredictability, where $\theta = \rho_{\max}$, by setting $\gamma = 0$ (ensuring no leakage of honest leaders) and defining GetClue to output an empty array. In Section 4.1, we provide another instantiation of $\mathcal{F}_{\mathsf{bsle}}$ by specifying both $\gamma = \alpha$ and GetClue, and we demonstrate that the Sassafras protocol securely realizes this specific instance of $\mathcal{F}_{\mathsf{bsle}}$ with $\theta = \frac{\rho_{\max}}{\beta(1-\chi)}$ where $1 \geq \chi > c$.

3.1 Communication Model

We now describe the communication models $\mathcal{F}_{\mathsf{com}}$ and $\mathcal{F}^{s}_{\mathsf{com}}$ (Figure 2) for parties running the batch SLE protocol. $\mathcal{F}_{\mathsf{com}}$ captures the diffusion functionality [21,4,26] with a bounded delay Δ, ensuring messages reach all parties within this limit. We alternatively introduce $\mathcal{F}^{s}_{\mathsf{com}}$, which is an extension of $\mathcal{F}_{\mathsf{com}}$ with a 'secure diffusion' option. With this, a sender can hide the message and the receiver, and only the size of the message is revealed.

Realizing Secure Diffusion: Given $\mathcal{F}_{\mathsf{com}}$, we can instantiate $\mathcal{F}^{s}_{\mathsf{com}}$ with Diffie-Hellman (DH) key exchange and semantically secure encryption [15]. Each party runs DH key exchange with others and obtain shared keys, each of which is the hash of the DH key, i.e., hash of g^{ab} where a and b are DH private keys.[9] When sending a secret message to P, a party encrypts it with their shared key and sends it via $\mathcal{F}_{\mathsf{com}}$. All parties receive the encrypted message but only P can decrypt it, while others see random data. This setup [15] ensures message secrecy and receiver anonymity under the session-key security of the key exchange [14] and semantic security of the encryption scheme.

While this approach is secure, note that in the blockchain setting this means that every party that enters the system must ask every other party to engage in a key exchange with them. This has the disadvantages that each party needs to store a secret key for every other party and that parties need to know each

[9] For example, a key derivation hash function (KDF) is commonly used, which includes a salt to strengthen the key and make it usable for encryption.

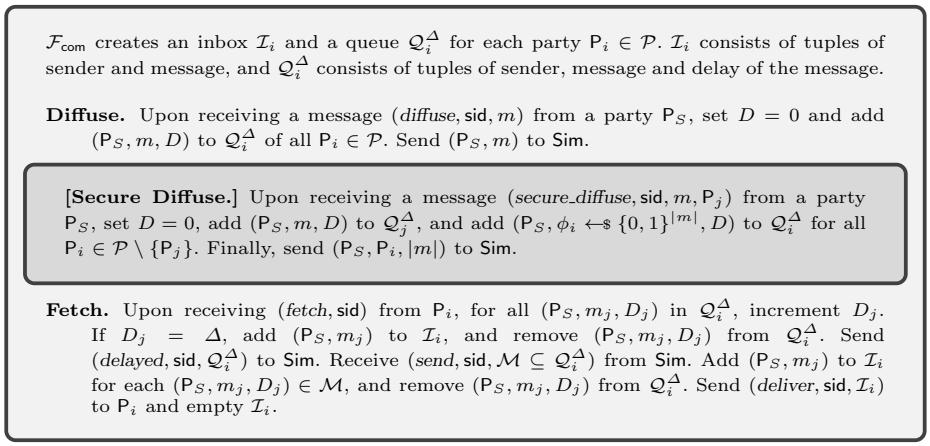

Fig. 2. $\mathcal{F}_{\mathsf{com}}$ without secure diffusion (the framed text) and $\mathcal{F}_{\mathsf{com}}^s$ with secure diffusion.

others' network identities for key setup. Now consider the following optimization. For each message, the sender generates an ephemeral key pair $(\mathsf{esk}, \mathsf{epk})$, where $\mathsf{epk} = g^{\mathsf{esk}}$, and the encrypted message is $(\mathsf{epk}, \mathsf{H}(\mathsf{pk}_\mathsf{P}^{\mathsf{esk}}) \oplus m)$, where pk_P is the receiver's public key, m is the ticket, and H is a random oracle whose output is the same fixed length as m. This has receiver anonymity, as under the computational Diffie-Hellman assumption in the random oracle model, the message is indistinguishable from a random group element and random string without knowledge of the receiver's secret key. The sender only needs to know the receiver's public key and not its network identity, and the receiver does not need to know anything. Using ephemeral keys has the advantage that arbitrarily many messages can be sent to the same receiver with the same key and that the one-time pad can be used as the symmetric encryption scheme.

4 Sassafras

We now introduce our single leader election protocol, Sassafras. Sassafras is described in the hybrid world where the following ideal functionalities exist: a randomness beacon functionality $\mathcal{F}_{\mathsf{rand}}$ (Figure 3), a ring VRF functionality $\mathcal{F}_{\mathsf{rvrf}}$ [10] (Figure 4) and a secure diffusion communication channel functionality $\mathcal{F}_{\mathsf{com}}^s$ (Figure 2). The Sassafras protocol does not use a functionality, but a State Machine Replication (SMR) protocol with properties defined in Section 2.1.

A ring VRF is a VRF that additionally allows the user to sign the VRF input and any extra message (e.g., auxiliary data) with a ring of public keys including their own, similar to ring signatures. The ring signature certifies that the VRF output has been generated with respect to one of the public keys in the ring.

To make the description of the protocol simpler and more compact, we avoid back and forth calls to the ring VRF functionality, and instead use the algorithms

of a scheme that realizes $\mathcal{F}_{\mathsf{rvrf}}$ [10]. We note that this choice has clearly no effect on the security and correctness of the final scheme, as we show in Appendix D.

Ring Verifiable Random Function [10]: A ring verifiable random function consists of the following polynomial-time algorithms:

- $\mathsf{rVRF.Gen}(pp) \to (\mathsf{sk}, \mathsf{pk})$ takes as input public parameters pp and outputs a secret key and public key.
- $\mathsf{rVRF.Eval}(\mathsf{sk}_i, m) \to y$ takes as input a secret key sk_i and a message $m \in \{0,1\}^*$ and outputs a ring VRF evaluation y.
- $\mathsf{rVRF.CommitRing}(\mathcal{PK}, \mathsf{pk}_i) \to (\mathsf{comring}, \mathsf{opring}_i)$ takes as input a ring of public keys $\mathcal{PK} = \{\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_n\}$ and a $\mathsf{pk}_i \in \mathcal{PK}$ and outputs a deterministic commitment $\mathsf{comring}$ to $\mathcal{PK}$ and opening opring_i.
- $\mathsf{rVRF.OpenRing}(\mathsf{comring}, \mathsf{opring}_i) \to \mathsf{pk}_i$ takes as input a commitment $\mathsf{comring}$ and opening opring and outputs a public key pk_i.
- $\mathsf{rVRF.Sign}(\mathsf{sk}_i, \mathsf{comring}, \mathsf{opring}_i, m, \mathsf{aux}) \to \sigma$ takes as input a secret key, commitment and opening, message, and auxiliary data $\mathsf{aux} \in \{0,1\}^*$ (e.g., the context in which m is used) and outputs a signature σ on (m, aux).
- $\mathsf{rVRF.Verify}(\mathsf{comring}, m, \mathsf{aux}, \sigma) \to (b, y)$ takes as input a commitment, message, auxiliary data, and a purported signature and outputs a bit $b \in \{0,1\}$ and a value y. $b = 1$ indicates acceptance and $b = 0$ indicates rejection.

$\mathsf{rVRF.CommitRing}$ and $\mathsf{rVRF.OpenRing}$ are optional algorithms, which are useful for succinct verification of large rings (else let $\mathsf{comring} = \mathcal{PK}$ and $\mathsf{opring}_i = \mathsf{pk}_i$). For example, $\mathsf{comring}$ could be a Merkle root of the keys in $\mathcal{PK}$, and opring the Merkle path of pk_i.

The security properties of a ring VRF, formally defined in $\mathcal{F}_{\mathsf{rvrf}}$ (Appendix C), are (1) correctness, (2) randomness: y is random and independent from the input and the key, (3) anonymity: y and σ do not leak any information about the public key of its signer except that the public key is in the ring, (4) unforgeability: an adversary should not be able to forge a ring VRF signature, and (5) uniqueness: there is only one valid y per key-input pair.

Finally, observe that the output of $\mathsf{rVRF.Eval}$ is independent of any specific ring. Consequently, the verification of two signatures for a given input using different rings results in the same evaluation value. This property allows a leader to disclose their identity as needed. For instance, suppose $y \leftarrow \mathsf{rVRF.Eval}(\mathsf{sk}_i, m)$ is verified via a ring VRF signature σ with a ring containing pk_i. Later, if the corresponding party wishes to affirm that y was generated using their key, they simply need to sign the same input with a ring containing only their key, i.e., $\mathcal{PK} = \{\mathsf{pk}_i\}$. This is precisely how leaders declare ownership of winning tickets in Sassafras.

Protocol Description: We now present the Sassafras protocol. Sassafras runs in the hybrid model $\mathcal{F}^s_{\mathsf{com}}, \mathcal{F}_{\mathsf{rand}}$ on top of State Machine Replication (SMR) (Section 2.1). Each party P_i generates a ring VRF key pair $(\mathsf{sk}, \mathsf{pk})$ by running $\mathsf{rVRF.Gen}(pp)$. All parties know the ring of public keys $\mathcal{PK} = \{\mathsf{pk}_i\}_{i=1}^n$, where n is the number of parties, and have equal power, i.e., $\mathcal{D}[\mathsf{P}] = \frac{1}{|\mathcal{P}|}$. Let $\mathsf{eid} = (e, \mathcal{P}, \mathcal{D})$,

where e is an agreed identifier that makes $\mathsf{eid} = (e, \mathcal{P}, \mathcal{D})$ unique. Before the protocol begins, each party runs $\mathsf{rVRF.CommitRing}(\mathcal{PK}, \mathsf{pk}_i) \rightarrow (\mathsf{comring}, \mathsf{opring}_i)$. Sassafras then runs in six phases: *Randomness Generation, Preparation, Distribution, Submission, Sortition,* and *Revelation*, as follows.

1) Randomness Generation: Each P_i sends $(\mathit{rand}, \mathsf{sid}, \mathsf{eid})$ to $\mathcal{F}_{\mathsf{rand}}$ (Figure 3) and receives randomness r_e. We employ this randomness beacon, wherein Sim can reset the randomness, because it is efficiently realizable on a blockchain [21,4,19].

Upon receiving $(\mathit{rand}, \mathsf{sid}, \mathsf{eid})$ from all honest parties in $\mathcal{P}$, if r_e is set, send $(\mathit{rand}, \mathsf{sid}, \mathsf{eid}, r_e)$ to all $\mathsf{P}_i \in \mathcal{P}$. Otherwise, sample r_e from $\{0,1\}^\lambda$ and send $(\mathit{rand_leak}, \mathsf{sid}, \mathsf{eid}, r_e)$ to Sim. Then set $\mathsf{counter} = 0$ and $\mathcal{C} = \emptyset$.

- Upon receiving the message $(\mathit{rand_reset}, \mathsf{sid}, \mathsf{eid})$ from Sim, sample another randomness $r_e \leftarrow\!\!\$\ \{0,1\}^\lambda$ if $\mathsf{counter} < \texttt{max-reset}$. Send the message $(\mathit{rand_leak}, \mathsf{sid}, \mathsf{eid}, r_e)$ to Sim. Set the $\mathsf{counter} = \mathsf{counter} + 1$ and the candidate set $\mathcal{C} = \mathcal{C} \cup \{r_e\}$.
- Upon receiving the message $(\mathit{rand_set}, \mathsf{sid}, \mathsf{eid}, r)$ from Sim, set $r_e = r$ if $r \in \mathcal{C}$.
- If no message is received from Sim or $\mathsf{counter} = \texttt{max-reset}$, move to the next step.

Send the message $(\mathit{rand}, \mathsf{sid}, \mathsf{eid}, r_e)$ to all $\mathsf{P}_i \in \mathcal{P}$.

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

2) Preparation: Parties prepare tickets using two parameters: $n_t \in \mathbb{N}$, the number of tickets, and a fixed parameter $c \in (0,1]$ used to determine the threshold for winning tickets. Each P_i generates n_t ring VRF outputs $y_{ij} \leftarrow \mathsf{rVRF.Eval}(\mathsf{sk}_i, r_e||j||e)$ for $1 \leq j \leq n_t$ (i.e., $m = r_e||j||e$ in the ring VRF description above). An index j is marked as a winner if $y_{ij} < c$. For each winning $j \in [1, n_t]$ that P_i holds, it generates a ring VRF signature $\sigma_{ij} \leftarrow \mathsf{rVRF.Sign}(\mathsf{sk}_i, \mathsf{comring}, \mathsf{opring}_i, r_e||j||e, \perp$ [10]), prepares a winning ticket (r_e, j, e, σ_{ij}), and adds the ticket to a list $\mathcal{L}_i$ of winning and non-winning tickets. For each $j \in [1, n_t]$ for which P_i is not the winner, P_i adds (r_e, Λ) to $\mathcal{L}_i$ anyway, in order to hide the number of winning tickets from the adversary. Here, Λ is a specific message of size of (j, e, σ_{ij}).

Note that each party obtains exactly n_t ring VRF outputs from $\mathsf{rVRF.Eval}$ for inputs $r_e||j||e$, with $j \in [1, n_t]$, due to the *uniqueness* property of ring VRF, which prevents parties from increasing their chances of having more winning tickets.

3) Distribution: P_i selects a random party P_ℓ, called a *repeater*, for each ticket t (whether a winner or not) in $\mathcal{L}_i$. It then sends t to P_ℓ using $(\mathit{secure_diffuse}, \mathsf{sid}, t, \mathsf{P}_\ell)$ via $\mathcal{F}^s_{\mathsf{com}}$, which hides the ticket's owner. (Recall that $\mathcal{F}^s_{\mathsf{com}}$ hides the message and the receiver with secure diffusion, which is necessary for unpredictability.)

When P_ℓ receives a ticket (r_e, j, e, σ_{ij}) from P_i, they verify it by checking that: (1) r_e is correct via $\mathcal{F}_{\mathsf{rand}}$, (2) $j \in [1, n_t]$, and (3) $\mathsf{rVRF.Verify}(\mathsf{comring}, r_e||j||e,$

[10] aux does not serve any purpose in Sassafras, so is set to $\perp$.

$\perp, \sigma_{ij})$ outputs $(1, y_{ij})$ with $y_{ij} < c$. If all checks pass, the ticket is considered valid (i.e., indeed a winning ticket).

4) Submission: This phase ends in Δ_{subm} slots. For all ℓ, repeater P_ℓ sends all of the valid tickets it has received as a command to SMR (Algorithm 1) via $\mathcal{F}_{\mathsf{com}}$. The algorithm TRANSITION($\mathsf{LOG}, (r_e, j, e, \sigma_{ij})$) of SMR verifies the ticket the same way the repeater does and, if it verifies, adds (r_e, j, e, σ_{ij}) to LOG. Note that all repeaters submit their tickets at the same time.

If a party sees that their repeater has not submitted their ticket(s) before the $\Delta_{\mathsf{subm}}{}^{th}$ slot (Section 2.1) after the randomness generation phase, they publish the tickets themselves, which reveals ownership. However, since a missing submission indicates that the repeater is malicious, the adversary gains no new information by this identity revelation. The submission phase ends Δ_{subm} slots after the randomness generation phase. We note that the length of Δ_{subm} should be chosen to account for the maximum time required to generate all tickets and for all repeaters to receive and submit them.

5) Sortition: After the submission phase, parties wait Δ_{sec} (Section 2.1) slots to ensure all tickets are finalized by SMR. They then sort the tickets by their y_{ij} values in descending order. The sorted list, `Tickets`, shows that `Tickets`$[u] = (r_e, j, e, \sigma_{ij})$ designates the signer of σ_{ij} as the leader of the uth election, with y_{ij} in u^{th} order. All parties can verify y_{ij} with $\mathsf{rVRF.Verify}(\mathsf{comring}, r_e||j||e, \perp, \sigma_{ij}) \rightarrow (1, y_{ij})$ without the ticket owner's key.

SMR ensures that all parties share the same, immutable leaders array. Due to the anonymity of ring VRF, parties do not know who generated the tickets, except the repeaters. If a repeater is malicious, the anonymity of that ticket is compromised, but unpredictability is maintained for others (see Section 4.1).

6) Revelation: When P_i wants to announce their leadership of the kth election where `Tickets`$[k] = (r_e, j, e, \sigma_{ij})$, it outputs a signature $\bar{\sigma}_{ij}$ generated by $\mathsf{rVRF.Sign}(\mathsf{sk}_i, \{\mathsf{pk}_i\}, \mathsf{opring}_{\mathsf{self}}, r_e||j||e, \perp)$ where $(\{\mathsf{pk}_i\}, \mathsf{opring}_{\mathsf{self}}) \leftarrow \mathsf{CommitRing}(\{\mathsf{pk}_i\}, \mathsf{pk}_i))$. Here, the ring consists of *only* the signer's key. When a party to whom the leader wishes to show their leadership receives $\bar{\sigma}_{ij}$, it runs $\mathsf{rVRF.Verify}(\{\mathsf{pk}_i\}, r_e||j||e, \perp, \bar{\sigma}_{ij})$. If it outputs $(1, y_{ij})$, the party verifies the leader of the kth election is P_i.

Sassafras operates with equal powers like other protocols [22,7,17,18], but can be extended to non-equal powers by dividing a party's power into equal parts, generating a new key for each.

Intuition for Network-Layer Anonymity. Consider Sassafras with $\mathcal{F}^s_{\mathsf{com}}$ instantiated as in Section 3.1. Even for a strong adversary that has global view and can observe all the traffic between parties, traffic correlation for tickets sent to honest repeaters cannot be used to link leaders and their tickets. This is because: 1) tickets are encrypted via a shared key between leader and repeater, which makes the tickets sent to different receivers indistinguishable, 2) tickets sent by all leaders are of the same size, and 3) the repeaters send out all of the winning tickets they received at the same time, eliminating the possibility to correlation via timing signatures.

4.1 Security Analysis of Sassafras

We prove that Sassafras realizes the functionality $\mathcal{F}_{\mathsf{bsle}}$ running the algorithm GetClue (Algorithm 2) with parameter $\gamma = \alpha$ and the clue domain $\mathcal{C} = \{1, \ldots, n_t\}$. We first analyze the unpredictability of $\mathcal{F}_{\mathsf{bsle}}$ according to Definition 5 with this instantiation. We then show the UC security of Sassafras.

GetClue *in our instantiation of* $\mathcal{F}_{\mathsf{bsle}}$*:* The algorithm GetClue (Algorithm 2) starts by initializing a clue domain list $\mathcal{C}_i = \mathcal{C}$ for each party P_i. This domain represents the set of possible clue values available for that party. The algorithm then iterates over each leader $\texttt{Leads}[i]$ to assign a unique clue value. For the ith leader P_j, a clue k is sampled uniformly at random from the current clue domain $\mathcal{C}_j$ (Line 5). After sampling, the chosen clue k is removed from $\mathcal{C}_j$ (Line 6). This ensures that each clue value in $\mathcal{C}$ is used *exactly once* for each party, preventing reuse of the same clue for the same leader.

Let us denote clue sets of each party P_i by $\mathcal{C} \setminus \texttt{ClueLists}[\mathsf{P}_i]$ where $\texttt{ClueLists}[\mathsf{P}_i]$ is the remaining set at the return line of Algorithm 2. The clue set of a party in $\mathcal{F}_{\mathsf{bsle}}$ running Algorithm 2 corresponds to the winner index set of a party in Sassafras. In Sassafras, an index j is considered a winner index of a winning ticket if the random value generated by $\mathsf{rVRF.Eval}$ is less than the threshold c. This winning ticket then determines the leader of one of the elections k in Sassafras. In $\mathcal{F}_{\mathsf{bsle}}$ running GetClue (Algorithm 2), the process of obtaining the clue set works in reverse: first, a leader for an election is selected with probability $\frac{c\rho_i}{\rho_{\max}} = c$ by running Elect with the equal power distribution, and then a unique, non-repeated clue index is assigned to the leader uniformly at random. Despite the difference in process order, the distribution of the clue set of a party in $\mathcal{F}_{\mathsf{bsle}}$ is identical to the distribution of the winner index set of a party in Sassafras. We formally establish the equivalence of these distributions in the proof of Theorem 1. This mapping is crucial for proving the realization of $\mathcal{F}_{\mathsf{bsle}}$ using Algorithm 2.

Algorithm 2 GetClue(Leads)

```
1: set ClueLists of size |P|
2: for P_i ∈ P do
3:     ClueLists[P_i] ← C
4: for i ∈ {1, 2, ..., |Leads|} do
5:     k ←$ ClueLists[Leads[i]]
6:     remove k from ClueLists[Leads[i]]
7:     Clue[i] ← k
   return Clue
```

Θ-Unpredictability: We show that unrevealed or non-leaked elections achieve a strong level of unpredictability, close to $\rho_{\max}$, even when $\texttt{Clue}$ provides information about who is *not* the leader in other elections. For instance, if an honest party is the leader of the ith election, where $\texttt{Clue}[i] = k$, this implies that the same party cannot be the leader of another election j where $\texttt{Clue}[j] = k$, as the same clue value is not reused for the leadership of the same party.

Our result in Lemma 2 shows that smaller c makes $\texttt{Clue}$ less useful for predicting leaders because the clue sets generated for each party vary significantly

across parties. This variation arises because fewer attempts result in smaller and more distinct sets of indices for each party, reducing overlap and making it harder for the adversary to correlate clues across elections.

Lemma 2. *$\mathcal{F}_{\mathsf{bsle}}$ running* GetClue *in Algorithm 2 is Θ-unpredictable where $\Theta = \frac{\rho_{\max}}{\beta(1-\chi)}$ and $1 \geq \chi > c$, except with probability $n_t \exp(-\beta\chi(\ln(\chi/c) - 1)/\rho_{\max})$.*

Lemma 2 establishes that $\mathcal{F}_{\mathsf{bsle}}$ with GetClue from Algorithm 2 is θ-unpredictable, where $\theta = \frac{\rho_{\max}}{\beta(1-\chi)}$. This formally shows that a smaller c results in a smaller θ, bringing it closer to $\rho_{\max}$. Specifically, as c decreases, the lower bound of χ ($\chi > c$) also decreases, allowing χ to take smaller values. This increases $1 - \chi$ for those values, leading to an overall decrease in θ across certain ranges of χ. The proof of Lemma 2 is given in Appendix B. We now state our main result.

Theorem 1. *Consider an instantiation of $\mathcal{F}_{\mathsf{bsle}}$ with Algorithm 2 where the parameter γ is set to α and parties have equal power. Sassafras realizes this instantiation $\mathcal{F}_{\mathsf{bsle}}$ in the $(\mathcal{F}_{\mathsf{rand}}, \mathcal{F}^s_{\mathsf{com}})$-hybrid model under the assumption that the underlying SMR is secure with parameter Δ_{sec} and the ring VRF scheme realizes $\mathcal{F}_{\mathsf{rvrf}}$ [10].*

We prove Theorem 1 in Appendix D. The intuition behind the proof is as follows: The output of Elect run by $\mathcal{F}_{\mathsf{bsle}}$ and the leaders selected by Sassafras are in the same distribution, so Sim uses provided leaders $\texttt{Leads}_m$ and `Clue` to output the same leaders in the real world. Similarly, since the distribution of clue sets of each party generated by GetClue in Algorithm 2 is the same as the winner index set of each party set in Sassafras, Sim uses `Clue` to simulate winner indices of Sassafras. Since we assume that the ring VRF scheme realizes $\mathcal{F}_{\mathsf{rvrf}}$, we replace the ring VRF scheme with $\mathcal{F}_{\mathsf{rvrf}}$ in Sassafras, which has a similar interface with the ring VRF definition given in Section 4 but via messages to $\mathcal{F}_{\mathsf{rvrf}}$. Then, we construct a simulator Sim, which simulates in the real world the honest parties of Sassafras, $\mathcal{F}_{\mathsf{rand}}, \mathcal{F}_{\mathsf{rvrf}}, \mathcal{F}^s_{\mathsf{com}}$, and simulates the adversary in $\mathcal{F}_{\mathsf{bsle}}$. When Sim receives a candidate election from $\mathcal{F}_{\mathsf{bsle}}$, Sim generates a randomness r_e as in $\mathcal{F}_{\mathsf{rand}}$ and leaks r_e to the adversary $\mathcal{A}$ in Sassafras. If $\mathcal{A}$ agrees on r_e, Sim sets the election in $\mathcal{F}_{\mathsf{bsle}}$. Otherwise, it asks $\mathcal{F}_{\mathsf{bsle}}$ to re-elect. After the election is set, Sim randomly picks ring VRF evaluation values less than c for $\mathcal{F}_{\mathsf{rvrf}}$ such that they correspond to the leaders in $\texttt{Leads}_m$ and `Leak`.

For the unknown honest leaders, Sim prepares tickets so that they can be verified via $\mathcal{F}_{\mathsf{rvrf}}$. Since the verification of a ticket does not require the ticket owner, Sim as $\mathcal{F}_{\mathsf{rvrf}}$ can simulate the verification even though it does not know the corresponding leaders in $\mathcal{F}_{\mathsf{bsle}}$, i.e., it just outputs verified when the signature of these tickets is given. For each leader in `Leak`, Sim randomly selects a malicious repeater to leak the ticket in the real world. Each leader in `Leads`, generated by Elect (run by $\mathcal{F}_{\mathsf{bsle}}$), has an attempt index $k \in [1, n_t]$ stored in `Clue`. This index corresponds to the winner index from Sassafras's preparation phase. Sim then maps the selected evaluation value for the ith leader as $(r||k||e)$ in $\mathcal{F}_{\mathsf{rvrf}}$,

where $\texttt{Clue}[i] = k$ and $\texttt{Leak}[i] = \mathsf{P}$. If some malicious tickets are not in SMR after Δ_{sec} slots, Sim deduces that $\mathcal{A}$ aborted the leaderships tied to those tickets and aborts them in $\mathcal{F}_{\mathsf{bsle}}$. Due to SMR's persistence, $\mathcal{A}$ cannot change the ticket order once finalized, ensuring the leader order remains the same in both worlds after Sim informs $\mathcal{F}_{\mathsf{bsle}}$ of the aborted leaderships.

During the simulation, Sim handles adaptive corruptions as follows. Once an honest party is corrupted, Sim treats it as malicious in subsequent steps. A challenge arises because the party was previously simulated without knowing its leaderships, verified via the $\mathcal{F}_{\mathsf{rvrf}}$ database. After corruption, Sim must provide the correct evaluation value when requested, but initially lacks knowledge of the ticket's association. Therefore, Sim corrupts the same party in $\mathcal{F}_{\mathsf{bsle}}$, retrieves all past leaderships from $\mathcal{F}_{\mathsf{bsle}}$, and updates its database to associate the tickets and their evaluation values with the corrupted party.

4.2 Sassafras with Blockchain and Blockchain with Sassafras

Sassafras and blockchain have a symbiotic relationship: Sassafras depends on the blockchain for liveness and persistence, while the blockchain relies on Sassafras to select leaders for block generation. This mutual dependence is essential for robustness in other SSLE protocols as well [22,7,17,18].

We build Sassafras on top of a secure blockchain protocol, such as [21,4,30,9], which runs in epochs $(e_1, e_2, \ldots)$. Each epoch has slots $(e_i = \{\mathsf{sl}_1, \ldots, \mathsf{sl}_{\texttt{eplen}}\})$, where each slot is assigned to a set of parties via a secret PLE before starting e_i. Parties in their assigned slots produce blocks according to the protocol.

We replace the election mechanism of the protocol with Sassafras, which we call $\mathfrak{B}$. The parameters n_t and c are set so that Sassafras realizes $\mathcal{F}_{\mathsf{bsle}}$ with $\ell_{\min} = \texttt{eplen}$ (see Proposition 1). Sassafras is first executed for e_1 using another SMR before $\mathfrak{B}$'s genesis block is released. During e_i, parties run $\mathfrak{B}$ based on the election results of e_{i-1} and execute Sassafras in parallel to elect leaders for e_{i+1}, using $\mathsf{eid} = (\mathcal{P}, \mathcal{D}, e_i)$. Parties run the preparation phase and give tickets to repeaters. Each repeater submits their ticket as a blockchain transaction during the submission phase, which ends $\Delta_{\mathsf{subm}} < \texttt{eplen}$ slots later. At the end of e_i, parties finalize tickets with $\mathfrak{B}$ and run the sortition phase to select the leaders for e_{i+1}. The first $\texttt{eplen}$ tickets in $\texttt{Tickets}$ determine the leaders in order.

Security analysis: We assume the same corruption model as in batch SLE and introduce a new type of corruption for blockchains, called *weak corruption*, which covers denial-of-service (DOS) attacks and is often overlooked in blockchain analyses, despite their relevance. The adversary $\mathcal{A}$ can weakly corrupt up to α_w-fraction of honest block producers. A weakly corrupted party cannot send or receive blocks during the attack. Afterward, the party receives all missed messages, but there is no guarantee that messages sent during the corruption were delivered.

Persistence in $\mathfrak{B}$ is ensured if the probability of having a slot with honest leader(s) is greater than 1/2 [27]. In our adversarial model, weakly corrupted

leaders cannot send blocks, acting outside the protocol. So we analyze the probability that a slot leader is weakly corrupted as well in $\mathfrak{B}$ with Sassafras, classifying all such slots as malicious. After this, we apply the same security analysis as the underlying blockchain protocol [21,4,9,30].

We first analyze the security of $\mathfrak{B}$ in an adversarial model we call the no-abort model, and from this, deduce that $\mathfrak{B}$ is a secure SMR. In this model, we assume $\mathcal{A}$ publishes all winning tickets of malicious parties during Sassafras's distribution phase. This restriction simplifies the analysis of slots, as without it, $\mathcal{A}$ could manipulate the probability of having an honest leader by withholding tickets. In the no-abort model, Sim is restricted to always send $(abort, \mathsf{sid}, \mathsf{DB}, \mathcal{L}_{\mathsf{abort}} = \emptyset)$ in $\mathcal{F}_{\mathsf{bsle}}$. We note that aborting slots does not help $\mathcal{A}$ compromise the security of $\mathfrak{B}$, as the election remains fair, as will be shown in the analysis.

Analysis of slots: We distinguish a slot sl as a public, malicious, or weak slot. Assuming that Sassafras realizes $\mathcal{F}_{\mathsf{bsle}}$ with Algorithm 2, so non-leaked election results are θ-unpredictable (Lemma 2) and fair, i.e., the probability that a slot is assigned to an honest party is $1-\alpha$, we obtain the following:

Public Slot: sl is a public slot if the repeater of the ticket assigned for sl is malicious. The probability of being a public slot is $p_{\mathsf{pSlot}} = (1-\alpha)\alpha = \alpha - \alpha^2$.

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

Weak Slot: sl is weak if its honest leader is weakly corrupted. If sl is public, $\mathcal{A}$ can weakly corrupt the leader with probability 1. If sl is neither public nor malicious, $\mathcal{A}$ can weakly corrupt up to α_w-fraction of parties, succeeding if one of them is the leader. Since the leader of sl is θ-unpredictable, the probability of a weak slot is $p_{\mathsf{wSlot}} = p_{\mathsf{pSlot}} + \theta\alpha_w n(1 - p_{\mathsf{mSlot}} - p_{\mathsf{pSlot}})$, where $n = |\mathcal{P}|$.

Next, we analyze the security of $\mathfrak{B}$ under the assumption that $p_{\mathsf{hSlot}} = 1 - p_{\mathsf{wSlot}} - p_{\mathsf{mSlot}} > \frac{1}{2}$. We conclude that $\mathfrak{B}$ provides at least the same security properties as its PLE-based version, which is secure when the probability of a slot having honest leaders exceeds $1/2$ [27]. While persistence analysis for SLE can be improved [3], this is beyond the scope of this paper.

Theorem 2. *Assuming $p_{\mathsf{hSlot}} > \frac{1}{2}$ and Sassafras executed before the genesis block realizes $\mathcal{F}_{\mathsf{bsle}}$ with Algorithm 2, $\mathfrak{B}$ has the liveness and persistence properties in **the no-abort model** with the epoch length $\mathtt{eplen} \geq \Delta_{\mathsf{subm}} + \Delta_{sec}$.*

Theorem 3. *$\mathfrak{B}$ is a secure SMR **iff** $\mathfrak{B}$ is a secure SMR in the no-abort model.*

The proofs are given in Appendix B. Theorems 1 and 2 show that Sassafras is secure with high probability, provided the parameters satisfy various assumptions defined in the theorems. We need to show what combinations of parameters are possible by instantiating $\mathfrak{B}$. For this we need to be specific about the negligible probabilities involved. The proof of the following proposition is given in Appendix E.

Proposition 1. *Given a failure probability $\epsilon > 0$ and an unpredictability $\theta < \frac{\rho_{\max}}{\alpha}$, we can find $\mathtt{eplen}, c, n_t$ such that with probability at least $1-\epsilon$, Sassafras is*

robust (Lemma 1) and θ-unpredictable (Lemma 2) and the security of $\mathfrak{B}$ holds. Furthermore, given $\rho_{\max} \leq \frac{1}{n'}$ for some n' and $\beta = \Theta(1)$, we can take $\theta = \frac{2\rho_{\max}}{\beta}$, $\epsilon = \exp(-\Theta(n'))$, $\Omega(\log n') \leq \texttt{eplen} \leq O(n')$, $c = \Theta(1)$ and $n_t = \Theta(1)$.

5 Conclusion

In this work, we propose a security model for batch single leader election in the UC framework and construct an efficient instantiation, Sassafras, which we prove secure in this model. In addition, we show how Sassafras can be deployed on a blockchain to elect block producers.

Sassafras has superior asymptotic performance over other single leader election protocols, as demonstrated in Table 1. To validate this, we conducted benchmark tests (see Appendix A for more details) using the same parameters as WHISK [22] with $n = 2^{14}$ and $\texttt{eplen} = 2^{13}$ slots, as shown in Table 2. Using Proposition 1, we set $n_t = 6$, $c = 0.18$, with $\alpha_w = 0.005$. This requires $\alpha < 0.29$ to maintain $p_{\mathsf{hSlot}} > 0.5$. In our analysis, we quantify the computational and communication overhead of the election protocol, focusing on the verification of election messages in a block and the additional data added to the blockchain when electing `eplen` leaders.

The communication overhead for electing `eplen` leaders in Sassafras is the size of all winning tickets. Assuming each block contains 2 winning tickets, the additional verification cost per block is for verifying two ring signatures[11]. Using the ring VRF construction from Section 6 of [10], each ring signature totals 432 bytes and takes about 3.9 ms to verify. The communication cost is $n \times n_t \times c$, representing the expected number of winning tickets, resulting in a 7.64 MB overhead for electing `eplen` leaders. With 2 tickets per block, the verification cost is 7.8 ms. Shuffle-based protocols, whether shuffling the entire party set [17] or a subset of size $\sqrt{n}$ [7,22], need inclusion of shuffle proofs [5] in a block for each election. We obtained the associated costs using the figures provided in Table 1 of [5], wherein verification of a shuffle proof requires $4n$ group operations, and the proof consists of $11\sqrt{n}$ group elements and $5n$ field elements. Notably, our communication cost results align with the analyses presented in [17,7]. PEKS-based SSLE [18], as shown in Table 2 of [18], requires 250 MB off-chain, and an estimated 300 MB on-chain. While more efficient than shuffle-based solutions, this still exceeds Sassafras's overhead. The computational cost of PEKS-based SSLE involves verifying two Bulletproofs [8] for encryption and decryption, as noted in [18], which remains reasonable for blockchain applications. For WHISK, we refer to the benchmark analysis in [22]. WHISK boasts computational cost comparable to that of PEKS-based solutions; however, its communication cost remains relatively high, and an order of magnitude higher than that of Sassafras.

In summary, our analysis shows that Sassafras outperforms other protocols in both computational and communication costs on-chain. Thus, Sassafras is an at-

[11] The number of winning tickets per block is a design decision of the blockchain protocol and can vary.

tractive option for real-world deployment, enabling scalable blockchain protocols with faster finalization [3] than probabilistic leader election.

References

1. Alper, H.K.: Network time with a consensus on clock. IACR Cryptol. ePrint Arch. (2019), https://eprint.iacr.org/2019/1348
2. Aptos: The aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure (2022), https://aptosfoundation.org/whitepaper/aptos-whitepaper_en.pdf
3. Azouvi, S., Cappelletti, D.: Private attacks in longest chain proof-of-stake protocols with single secret leader elections. In: Baldimtsi, F., Roughgarden, T. (eds.) AFT 2021, Arlington, Virginia, USA, September 26 - 28, 2021. pp. 170–182. ACM (2021). https://doi.org/10.1145/3479722.3480996
4. Badertscher, C., Gazi, P., Kiayias, A., Russell, A., Zikas, V.: Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 913–930. ACM (2018). https://doi.org/10.1145/3243734.3243848
5. Bayer, S., Groth, J.: Efficient zero-knowledge argument for correctness of a shuffle. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012, Cambridge, UK, April 15-19, 2012. LNCS, vol. 7237, pp. 263–280. Springer (2012). https://doi.org/10.1007/978-3-642-29011-4_17
6. Blackshear, S., Chursin, A., Danezis, G., Kichidis, A., Kokoris-Kogias, L., Li, X., Logan, M., Menon, A., Nowacki, T., Sonnino, A., Williams, B., Zhang, L.: Sui lutris: A blockchain combining broadcast and consensus. In: Luo, B., Liao, X., Xu, J., Kirda, E., Lie, D. (eds.) CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024. pp. 2606–2620. ACM (2024). https://doi.org/10.1145/3658644.3670286
7. Boneh, D., Eskandarian, S., Hanzlik, L., Greco, N.: Single secret leader election. In: AFT 2020, New York, NY, USA, October 21-23, 2020. pp. 12–24. ACM (2020). https://doi.org/10.1145/3419614.3423258
8. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short proofs for confidential transactions and more. In: SP 2018, May 21-23, 2018, San Francisco, California, USA. pp. 315–334. IEEE Computer Society (2018). https://doi.org/10.1109/SP.2018.00020
9. Burdges, J., Cevallos, A., Czaban, P., Habermeier, R., Hosseini, S., Lama, F., Alper, H.K., Luo, X., Shirazi, F., Stewart, A., Wood, G.: Overview of polkadot and its design considerations. CoRR **abs/2005.13456** (2020), https://arxiv.org/abs/2005.13456
10. Burdges, J., Ciobotaru, O., Crites, E., Alper, H.K., Stewart, A., Vasilyev, S.: Ring verifiable random functions. IACR Cryptol. ePrint Arch. (2023), https://eprint.iacr.org/2023/002
11. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. IACR Cryptol. ePrint Arch. (2000), http://eprint.iacr.org/2000/067
12. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS 2001, October 14-17, 2001, Las Vegas, Nevada, USA. pp. 136–145. IEEE Computer Society (2001). https://doi.org/10.1109/SFCS.2001.959888
13. Canetti, R., Hogan, K., Malhotra, A., Varia, M.: A universally composable treatment of network time. In: CSF 2017, Santa Barbara, CA, USA, August 21-25,

2017. pp. 360–375. IEEE Computer Society (2017). https://doi.org/10.1109/CSF.2017.38
14. Canetti, R., Krawczyk, H.: Analysis of key-exchange protocols and their use for building secure channels. In: Pfitzmann, B. (ed.) EUROCRYPT 2001, Innsbruck, Austria, May 6-10, 2001. LNCS, vol. 2045, pp. 453–474. Springer (2001). https://doi.org/10.1007/3-540-44987-6_28
15. Canetti, R., Krawczyk, H.: Universally composable notions of key exchange and secure channels. In: Knudsen, L.R. (ed.) EUROCRYPT 2002, Amsterdam, The Netherlands, April 28 - May 2, 2002. LNCS, vol. 2332, pp. 337–351. Springer (2002). https://doi.org/10.1007/3-540-46035-7_22
16. Cascudo, I., David, B.: SCRAPE: scalable randomness attested by public entities. In: Gollmann, D., Miyaji, A., Kikuchi, H. (eds.) ACNS 2017, Kanazawa, Japan, July 10-12, 2017. LNCS, vol. 10355, pp. 537–556. Springer (2017). https://doi.org/10.1007/978-3-319-61204-1_27
17. Catalano, D., Fiore, D., Giunta, E.: Adaptively secure single secret leader election from DDH. In: Milani, A., Woelfel, P. (eds.) PODC '22, Salerno, Italy, July 25 - 29, 2022. pp. 430–439. ACM (2022). https://doi.org/10.1145/3519270.3538424
18. Catalano, D., Fiore, D., Giunta, E.: Efficient and universally composable single secret leader election from pairings. In: Boldyreva, A., Kolesnikov, V. (eds.) PKC 2023, Atlanta, GA, USA, May 7-10, 2023. LNCS, vol. 13940, pp. 471–499. Springer (2023). https://doi.org/10.1007/978-3-031-31368-4_17
19. Chen, J., Micali, S.: Algorand: A secure and efficient distributed ledger. Theor. Comput. Sci. **777**, 155–183 (2019). https://doi.org/10.1016/J.TCS.2019.02.001
20. Daian, P., Pass, R., Shi, E.: Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In: Goldberg, I., Moore, T. (eds.) FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019. LNCS, vol. 11598, pp. 23–41. Springer (2019). https://doi.org/10.1007/978-3-030-32101-7_2
21. David, B., Gazi, P., Kiayias, A., Russell, A.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Tel Aviv, Israel, April 29 - May 3, 2018. LNCS, vol. 10821, pp. 66–98. Springer (2018). https://doi.org/10.1007/978-3-319-78375-8_3
22. Drake, J., Feist, D., Herold, G., Khovratovich, D., Maller, M., Simkin, M.: Whisk: A practical shuffle-based SSLE protocol for ethereum (2022), https://hackmd.io/@asn-d6/HyD3Yjp2Y
23. drand: Drand - a distributed randomness beacon daemon (2017), https://github.com/drand/drand
24. Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Sofia, Bulgaria, April 26-30, 2015. LNCS, vol. 9057, pp. 281–310. Springer (2015). https://doi.org/10.1007/978-3-662-46803-6_10
25. Hanke, T., Movahedi, M., Williams, D.: DFINITY technology overview series, consensus system. CoRR **abs/1805.04548** (2018), http://arxiv.org/abs/1805.04548
26. Katz, J., Maurer, U., Tackmann, B., Zikas, V.: Universally composable synchronous computation. In: Sahai, A. (ed.) TCC 2013, Tokyo, Japan, March 3-6, 2013. LNCS, vol. 7785, pp. 477–498. Springer (2013). https://doi.org/10.1007/978-3-642-36594-2_27

27. Kiayias, A., Quader, S., Russell, A.: Consistency of proof-of-stake blockchains with concurrent honest slot leaders. In: ICDCS 2020, Singapore, November 29 - December 1, 2020. pp. 776–786. IEEE Computer Society (2020). https://doi.org/10.1109/ICDCS47774.2020.00065
28. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Santa Barbara, CA, USA, August 20-24, 2017. LNCS, vol. 10401, pp. 357–388. Springer (2017). https://doi.org/10.1007/978-3-319-63688-7_12
29. Micali, S., Rabin, M.O., Vadhan, S.P.: Verifiable random functions. In: FOCS 1999, October 17-18, 1999, New York, NY, USA. pp. 120–130. IEEE Computer Society (1999). https://doi.org/10.1109/SFFCS.1999.814584
30. Pass, R., Shi, E.: The sleepy model of consensus. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017, Hong Kong, China, December 3-7, 2017. LNCS, vol. 10625, pp. 380–409. Springer (2017). https://doi.org/10.1007/978-3-319-70697-9_14
31. Syta, E., Jovanovic, P., Kokoris-Kogias, E., Gailly, N., Gasser, L., Khoffi, I., Fischer, M.J., Ford, B.: Scalable bias-resistant distributed randomness. In: SP 2017, San Jose, CA, USA, May 22-26, 2017. pp. 444–460. IEEE Computer Society (2017). https://doi.org/10.1109/SP.2017.45
32. Web3 Foundation: Ring vrf implementation (2022), https://github.com/davxy/ark-vrf

A Details of Our Benchmark Analysis

In our benchmark analysis for Sassafras, we use the ring VRF scheme rVRF from Section 6 of [10], instantiated with the Groth16-based proving system SpecialG, as described in Section 5.1 of the same work.[12] This construction operates over bilinear groups $\mathbb{G}_1$ and $\mathbb{G}_2$ from the BLS12-381 curve, and a separate group $\mathbb{G}'$[13] from Curve25519. In this setting, $\mathbb{G}_1$ elements are 48 bytes, $\mathbb{G}_2$ elements are 96 bytes, and $\mathbb{G}'$ elements are 32 bytes in size. The groups $\mathbb{G}_1$ and $\mathbb{G}_2$ are used in pairing operations.

We benchmarked this construction on our local machine with an AMD Ryzen 9 5950X 16-Core Processor, with the following average costs: Group multiplication in $\mathbb{G}_1$: 157.83×10^{-3} ms, group multiplication in $\mathbb{G}_2$: 507.62×10^{-3} ms, group multiplication in $\mathbb{G}'$: 79.9×10^{-3} ms, and pairing operation: 1.0386 ms.

In Sassafras, the main additional computational cost per block comes from verifying two ring VRF tickets. The cost is dominated by the rVRF.Verify algorithm, which includes (1) verification of a Chaum-Pedersen DLEQ proof (see PedVRF.Verify in Section 4 of [10]), requiring 3 multiplications in $\mathbb{G}_1$ and 2 in $\mathbb{G}'$ and (2) verification of a SpecialG (Groth16) proof (see Section 5.2 of [10]), which involves 1 multiplication in $\mathbb{G}_1$ and 3 pairing operations. The resulting ring signature consists of: a Pedersen commitment (32 bytes), a Groth16 proof (192 bytes), a Chaum-Pedersen proof (176 bytes) and one group element (32 bytes) for a total size of **432 bytes**.

Each rVRF.Verify call takes approximately 3.9 ms. Given that the expected number of winning tickets is $n \times n_t \times c$, the total communication overhead for

[12] Ring VRF has also been implemented in [32].

[13] $\mathbb{G}'$ is the group for the preout element, which is part of the ring VRF signature.

electing `eplen` leaders in a batch is around 7.64 MB on-chain. Assuming each block carries two ring VRF proofs, the ticket verification overhead per block is approximately 7.8 ms.

For the shuffle-based protocols [17,7], we use Curve25519 for our benchmark analysis. We rely on the implementation results presented in Section 6 of [5] to derive the computational and communication overhead shown in Table 2. Bayer and Groth [5] state that verifying a shuffle proof requires $4n$ group operations, and the proof consists of $11\sqrt{n}$ group elements and $5n$ field elements. Notably, our communication cost measurements align with the estimates reported in [17].

For WHISK [22], we use the analysis provided directly in the original work.

For the PEKS-based protocol [18], we use both the BLS12-381 curve and Curve25519. Communication overhead is derived from Table 2 in [18], while computational cost is estimated based on the verification of two Bulletproofs [8] used for encryption and decryption, as also discussed in [18].

B Proofs

Proof. (of Lemma 2.) Let E be the event we are conditioning on $\mathtt{R}$ as defined in Definition 5 and s be an election where $\mathtt{R}[s] = \perp$ and $\mathtt{Leads}[s] \in \mathcal{P}_H = \mathcal{P} \setminus \mathcal{P}_{\mathcal{A}}$ and $k = \mathtt{Clue}[s]$. Let $\mathcal{S}_k$ be the set of honest parties in $\mathtt{R}$ which *are not* a leader of an election $s' \neq s$ where $\mathtt{Clue}[s'] = k$. Given $\mathtt{R}[s] = \perp$, $\mathtt{Leads}[s] \in \mathcal{P}_H$ and $\mathtt{Clue}[s] = k$, $\Pr[\mathtt{Leads}[s] = \mathsf{P}_i | E] = 0$ if $\mathsf{P}_i \notin \mathcal{S}_k$ and $\Pr[\mathtt{Leads}[s] = \mathsf{P}_i | E] = \frac{\rho_i}{\sum_{j \in \mathcal{S}_k} \rho_j}$ 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 for honest parties that have $\omega_k \geq \tau_i$. These parties do not have any leadership with index k, so $\mathcal{N}_k \subset \mathcal{S}_k$. We show that except with probability $\exp(-\frac{\beta\chi(\ln(\chi/c)-1)}{\rho_{\max}})$ (not conditioned on E), for all indices k, we have $\sum_{\mathsf{P}_i \in \mathcal{N}_k} \rho_i \geq \beta(1-\chi)$ and deduce $\Pr[\mathtt{Leads}[s] = \mathsf{P}_i | E] \leq \frac{\rho_i}{\beta(1-\chi)}$.

We define a Bernoulli random variable $X_{i,k}$ which is 1 if $\omega_k < \tau_i$ for $\mathsf{P}_i \in \mathcal{P}_H$. So, we have $\sum_{\mathsf{P}_i \in \mathcal{N}_k} \rho_i = \sum_{\mathsf{P}_i \in \mathcal{P}_H} \rho_i (1 - X_{i,k}) = \beta - \sum_{\mathsf{P}_i \in \mathcal{P}_H} \rho_i X_{i,k} = \beta - X_k$. $\Pr[X_{i,k} = 1] = \frac{c\rho_i}{\rho_{\max}}$ so $E[X] = \mu = \sum_{\mathsf{P}_i \in \mathcal{P}_H} \frac{c\rho_i^2}{\rho_{\max}} \leq \sum_{\mathsf{P}_i \in \mathcal{P}_H} c\rho_i = \beta c$. Since $\chi > c$, this implies that $\mu < \beta\chi$. By applying the Chernoff bound, in the form of Lemma 3, we get $\Pr[X_k \geq \beta\chi] \leq \exp(\frac{-\mu - \beta\chi(\ln(\beta\chi/\mu)-1)}{\rho_{\max}}) \leq \exp(\frac{-\beta\chi(\ln(\beta\chi/\mu)-1)}{\rho_{\max}})$. Since $\mu \leq \beta c$, it is less than $\exp(\frac{-\beta\chi(\ln(\chi/c)-1)}{\rho_{\max}})$.

When $X_k \leq \beta\chi$, we have that $\sum_{\mathsf{P}_i \in \mathcal{S}_k} \rho_i \geq \sum_{\mathsf{P}_i \in \mathcal{N}_k} \rho_i \geq \beta(1-\chi)$. So, $\Pr[\mathtt{Leads}[s] = \mathsf{P}_i | E] = \frac{\rho_i}{\sum_{\mathsf{P}_j \in S_k} \rho_j} \leq \frac{\rho_i}{\sum_{\mathsf{P}_j \in \mathcal{N}_k} \rho_j} \leq \frac{\rho_i}{\beta(1-\chi)}$. For this property to hold for all parties in $\mathcal{P}_H$, we need $X_k \leq \beta\chi$ to hold for all indices k. For any one index, it holds except with probability $\exp(\frac{-\beta\chi(\ln(\chi/c)-1)}{\rho_{\max}})$. By a union bound, it holds for all indices k with probability $n_t \exp(\frac{-\beta\chi(\ln(\chi/c)-1)}{\rho_{\max}})$. □

Proof. (of Lemma 1.) We need to upper bound $\mathcal{L}_H = |\cup_{\mathsf{P}_i \in \mathcal{P}_H} \mathcal{L}_i| = |\{\omega_{i,k} : \mathsf{P}_i \in \mathcal{P}_H, k \in [1, n_t], \omega_{i,k} < \tau_i\}|$. Let's define a random variable $X_{i,k}$ to for $\mathsf{P}_i \in \mathcal{P}_h$ and $1 \leq k \leq n_t$. $X_{i,k}$ is 1 if $\omega_{i,k} < \tau_i$. Since $\Pr[X_{i,k} = 1] = \Pr[\omega_k < \tau_i] = \frac{c\rho_i}{\rho_{\max}}$, $E[\sum_{i,k} X_{i,k}] = \mu = \frac{\beta c n_t}{\rho_{\max}} = r\ell_{\min}$. Applying the Chernoff bound, for all $\delta \in$

$(0,1)$, $\Pr\Big[|\mathcal{L}_H| = \sum_{i,k} X_{i,k} \leq (1-\delta)\mu\Big] \leq \exp(-\frac{\delta^2\mu}{2})$. Therefore, $|\mathcal{L}_H| > (1-\delta)\mu$ except with the probability $\exp(-\frac{\delta^2\mu}{2})$. Since $r > 1$, we set $\delta = 1-1/r$ and obtain $(1-\delta)\mu = \ell_{\min}$. Thus, $|\mathcal{L}_H| > \ell_{\min}$ except with the probability $\exp(-\frac{(r-1)^2}{2r}\ell_{\min})$. □

Proof. (of Theorem 2.) If Sassafras satisfies $\mathcal{F}_{\mathsf{bsle}}$ with Algorithm 2 in epoch e_i, the same slot analysis applies to e_{i+1}. For e_1, this implies a probability of $p_{\mathsf{hSlot}} > \frac{1}{2}$ for having an honest leader, ensuring persistence and liveness as in $\mathfrak{B}$ with PLE. Consequently, all submitted tickets within Δ_{subm} are finalized by the end of e_1. We can then apply this slot analysis to all $e_i \geq 2$, demonstrating that $\mathfrak{B}$ ensures persistence and liveness in each epoch, just like the underlying blockchain protocol with PLE in the no-abort model. □

Proof. (of Theorem 3.) We first show that if $\mathcal{A}$ breaks the security of $\mathfrak{B}$, we can construct an adversary $\mathcal{B}$ to break the security of $\mathfrak{B}$ in the no-abort model. The reduction is straightforward after replacing Sassafras with $\mathcal{F}_{\mathsf{bsle}}$. $\mathcal{B}$ simulates $\mathcal{F}_{\mathsf{bsle}}$ against $\mathcal{A}$, relaying messages between them. If $\mathcal{A}$ aborts slots (i.e., sends $(abort, \mathsf{sid}, \mathcal{L}_{\mathsf{abort}})$), $\mathcal{B}$ sends $(abort, \mathsf{sid}, \emptyset)$ to $\mathcal{F}_{\mathsf{bsle}}$ and does not produce blocks for those slots in $\mathfrak{B}$. For the other slots, $\mathcal{B}$ mimics $\mathcal{A}$. Thus, if $\mathcal{A}$ breaks Sassafras, $\mathcal{B}$ does so in the no-abort model. Conversely, if $\mathcal{A}$ breaks the security of $\mathfrak{B}$ in the no-abort model, we can construct another adversary $\mathcal{B}$ to break $\mathfrak{B}$, simply aborting all slots where $\mathcal{A}$ does not produce blocks. □

B.1 A Version of the Chernoff Bound

We now prove the following version of the Chernoff bound.

Lemma 3. *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 $t = \ln(s/\mu)/w_{\max}$,

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

$$
\begin{aligned}
&= \exp(-st + \sum_i p_i(exp(w_i t) - 1)) \\
&\leq \exp(-st + \sum_i p_i w_i(exp(w_{\max} t) - 1)/w_{\max}) \\
&= \exp(-s(\ln(s/\mu)/w_{\max}) + \sum_i p_i w_i(s/\mu - 1)/w_{\max}) \\
&= \exp((-s\ln(s/\mu) - \mu + s)/w_{\max})
\end{aligned}
$$

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

C Ring Verifiable Random Function

Here, we give the ring VRF functionality $\mathcal{F}_{\mathsf{rvrf}}$ defined in [10]. We refer to [10] for a complete description of $\mathcal{F}_{\mathsf{rvrf}}$. Note that W below is implicitly part of the output σ of $\mathsf{rVRF.Sign}$ in Section 4. At a high level, the functionality works as follows:

[**Honest Ring VRF Signature and Evaluation.**] $\mathcal{F}_{\mathsf{rvrf}}$ replies to honest signature verification messages by generating an anonymous signature and a random evaluation value. It enforces that each message is mapped to one random evaluation value. Given the message $(\mathit{sign}, \mathsf{sid}, \mathcal{PK}, \mathsf{pk}, \mathsf{aux}, m)$, $\mathcal{F}_{\mathsf{rvrf}}$ returns a random anonymous key W, a signature σ, and a random output y for the message m, pk if $\mathsf{pk} \in \mathcal{PK}$. $\mathcal{F}_{\mathsf{rvrf}}$ makes sure that (pk, 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 $\mathsf{pk} \in \mathcal{PK}'$, $\mathcal{F}_{\mathsf{rvrf}}$ outputs the same W and y but the signature σ may change. Whenever $\mathcal{F}_{\mathsf{rvrf}}$ generates a signature, it creates a record $[m, \mathsf{aux}, W, \sigma, 1]$ to record that σ is a valid signature of m and aux signed by an anonymous key W. The parameter aux is used to bind the ring VRF signature to a context e.g., to prevent replay attacks.

[**Ring VRF Verification.**]**:** Given the message $(\mathit{verify}, \mathsf{sid}, \mathcal{PK}, W, m, \mathsf{aux}, \sigma)$, if σ is a valid signature, i.e., there exists a record $[m, \mathsf{aux}, W, \sigma, 1]$, $\mathcal{F}_{\mathsf{rvrf}}$ returns 1 (verified) and the random output y that corresponds to σ. Otherwise, it returns 0. The 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. Here, forgery means that $[m, \mathsf{aux}, W, \sigma, 1]$ exists in $\mathcal{F}_{\mathsf{rvrf}}$'s record where (pk, m) is mapped to W, but the honest party with pk never sends $(\mathit{sign}, \mathsf{sid}, \mathcal{PK}, \mathsf{pk}, m, \mathsf{aux})$. If a malicious party wants to register its signature to be verified with $\mathcal{PK}, W, m, \mathsf{aux}$, it just provides a signature σ' generated by $\mathcal{A}$ by sending $(\mathit{verify}, \mathsf{sid}, \mathcal{PK}, W, m, \mathsf{aux}, \sigma')$.

The verification process does not need the public key of the signer to verify the signature and outputs its random output. So, it does not reveal the signer's verification key. This is the most subtle difference of ring VRFs and VRFs. If a signer wants to reveal its identity, i.e., link a random output y to its verification key pk, the signer should re-sign the same message with a ring $\mathcal{PK} = \{\mathsf{pk}\}$ and

$\mathcal{F}_{\mathsf{rvrf}}$ runs a PPT algorithm Gen_{sign} during the execution and is parametrized with sets $\mathcal{S}_{eval}$ and $\mathcal{S}_W$ where $\mathcal{S}_{eval}$ and $\mathcal{S}_W$ are generated by a setup function $\mathsf{Setup}(\lambda)$.

[Key Generation.] Upon receiving a message $(\mathit{keygen}, \mathsf{sid})$ from P_i, send $(\mathit{keygen}, \mathsf{sid}, \mathsf{P}_i)$ to the simulator Sim. Upon receiving a message $(\mathit{verificationkey}, \mathsf{sid}, \mathsf{sk}, \mathsf{pk})$ from Sim, verify that sk or pk has not been recorded before for sid in $\mathtt{verification_keys}$. If it is the case, store the value sk, pk in the table $\mathtt{verification_keys}$ under P_i and return $(\mathit{verificationkey}, \mathsf{sid}, \mathsf{pk})$ to P_i.

[Honest Ring VRF Signature and Evaluation.] Upon receiving a message $(\mathit{sign}, \mathsf{sid}, \mathsf{ring}, \mathsf{pk}_i, \mathsf{aux}, m)$ from P_i, verify that $\mathsf{pk}_i \in \mathsf{ring}$ and that there exists a public key pk_i associated to P_i in $\mathtt{verification_keys}$. If it is not the case, ignore the request. If there exists no W' such that $\mathtt{anonymous_key_map}[m, W'] = \mathsf{pk}_i$, let $W \leftarrow\!\$\ \mathcal{S}_W$ and let $\mathsf{out} \leftarrow\!\$\ \mathcal{S}_{eval}$. Set $\mathtt{anonymous_key_map}[m, W] = \mathsf{pk}_i$ and set $\mathsf{evaluations}[m, W] = \mathsf{out}$. In any case (other than ignoring the request), obtain W, out where $\mathtt{anonymous_key_map}[m, W] = \mathsf{pk}_i$, $\mathsf{evaluations}[m, W] = \mathsf{out}$, and $(\mathsf{sk}, \mathsf{pk})$ is in $\mathtt{verification_keys}$. Then run $\mathsf{Gen}_{sign}(\mathsf{ring}, \mathsf{sk}, \mathsf{pk}, \mathsf{aux}, m) \rightarrow \sigma$. Let $\sigma = (\sigma, W)$ and record $[m, \mathsf{aux}, W, \mathsf{ring}, \sigma, 1]$. Return $(\mathit{signature}, \mathsf{sid}, \mathsf{ring}, W, \mathsf{aux}, m, \mathsf{out}, \sigma)$ to P_i.

[Malicious Ring VRF Evaluation.] Upon receiving a message $(\mathit{eval}, \mathsf{sid}, \mathsf{pk}_i, W, m)$ from Sim, if pk_i is recorded under an honest party's identity or if there exists $W' \neq W$ where $\mathtt{anonymous_key_map}[m, W'] = \mathsf{pk}_i$, ignore the request. Otherwise, record in the table $\mathtt{verification_keys}$ the value $(\perp, \mathsf{pk}_i)$ under Sim if $(\cdot, \mathsf{pk}_i)$ is not in $\mathtt{verification_keys}$. If $\mathtt{anonymous_key_map}[m, W]$ has not been defined before, set $\mathtt{anonymous_key_map}[m, W] = \mathsf{pk}_i$ and let $\mathsf{out} \leftarrow\!\$\ \mathcal{S}_{eval}$ and set $\mathsf{evaluations}[m, W] = \mathsf{out}$. In any case (other than ignoring the request), obtain $\mathsf{out} = \mathsf{evaluations}[m, W]$ and return $(\mathit{evaluated}, \mathsf{sid}, m, \mathsf{pk}_i, W, \mathsf{out})$ to P_i.

[Corruption:] Upon receiving $(\mathit{corrupt}, \mathsf{sid}, \mathsf{P}_i)$ from Sim, remove (x_i, pk_i) from $\mathtt{verification_keys}[\mathsf{P}_i]$ and store them in $\mathtt{verification_keys}$ under Sim. Return $(\mathit{corrupted}, \mathsf{sid}, \mathsf{P}_i)$.

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

[Ring VRF Verification.] Upon receiving a message $(\mathit{verify}, \mathsf{sid}, \mathsf{ring}, W, \mathsf{aux}, m, \sigma)$ from a party, do the following:

- If there exists a record $[m, \mathsf{aux}, W, \mathsf{ring}, \sigma, b']$, set $b = b'$.
- Else if $\mathtt{anonymous_key_map}[m, W]$ is an honest verification key and there exists a record $[m, \mathsf{aux}, W, \mathsf{ring}, \sigma', 1]$ for any σ', then let $b = 1$ and record $[m, \mathsf{aux}, W, \mathsf{ring}, \sigma, 1]$.
- Else relay the message $(\mathit{verify}, \mathsf{sid}, \mathsf{ring}, W, \mathsf{aux}, m, \sigma)$ to Sim and receive back the message $(\mathit{verified}, \mathsf{sid}, \mathsf{ring}, W, \mathsf{aux}, m, \sigma, b_{\mathsf{Sim}}, \mathsf{pk}_{\mathsf{Sim}})$. Then check the following:
  1. If $\mathsf{pk}_{\mathsf{Sim}}$ is an honest verification key, set $b = 0$.
  2. Else if $W \notin \mathcal{W}[m, \mathsf{ring}]$ and $|\mathcal{W}[m, \mathsf{ring}]| \geq |\mathsf{ring}_{mal}|$ where ring_{mal} is a set of malicious keys in ring, set $b = 0$.
  3. Else if there exists $W' \neq W$ where $\mathtt{anonymous_key_map}[m, W'] = \mathsf{pk}_{\mathsf{Sim}}$, set $b = 0$.
  4. Else set $b = b_{\mathsf{Sim}}$.

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

- If $W \notin \mathcal{W}[m, \mathsf{ring}]$, add W to $\mathcal{W}[m, \mathsf{ring}]$.
- If $\mathsf{evaluations}[m, W]$ is not defined, sample $y \leftarrow\!\$\ \mathcal{S}_{eval}$. Then, set $\mathtt{anonymous_key_map}[m, W] = \mathsf{pk}_{\mathsf{Sim}}$ and $\mathsf{evaluations}[m, W] = \mathsf{out}$.
- Otherwise, set $\mathsf{out} = \mathsf{evaluations}[m, W]$.

Finally, output $(\mathit{verified}, \mathsf{sid}, \mathsf{ring}, W, \mathsf{aux}, m, \sigma, \mathsf{out}, b)$ to the party.

[Corruption:] Upon receiving $(\mathit{corrupt}, \mathsf{sid}, \mathsf{P}_i)$ from Sim, remove $(\mathsf{sk}_i, \mathsf{pk}_i)$ from $\mathtt{verification_keys}[\mathsf{P}_i]$ and store them in $\mathtt{verification_keys}$ under Sim. Return $(\mathit{corrupted}, \mathsf{sid}, \mathsf{P}_i)$.

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

obtain σ'. In this way, when another party verifies σ' with $\mathcal{PK} = \{\mathsf{pk}\}$ and obtains y, they deduce that y is the random output of m and pk.

Sim can query the following message which is simulated differently but indistinguishable in the proof of Theorem 1:

[**Malicious Ring VRF Evaluation.**]**:** Sim sends $(\mathit{eval}, \mathsf{sid}, \mathsf{pk}, W, m)$ and receives a random evaluation value. $\mathcal{F}_{\mathsf{rvrf}}$ evaluates the input using pk only if no other anonymous key W' (where $W' \neq W$) has been previously associated with the same input and key $(m, \mathcal{PK})$. If this condition is met, it returns the stored evaluation value for (m, W); otherwise, a random value is returned.

The restriction to a unique anonymous key for each $(m, \mathcal{PK})$ pairing ensures that Sim cannot obtain multiple evaluation values for the same input and key, preserving the uniqueness property.

D Proof of Security for Sassafras

We now prove Theorem 1.

Proof. Since the ring VRF scheme realizes $\mathcal{F}_{\mathsf{rvrf}}$, we can replace the ring VRF scheme with $\mathcal{F}_{\mathsf{rvrf}}$; that is,

- rVRF.Eval: For honest parties, replaced by sending the message $(\mathit{sign}, \mathsf{sid}, \mathcal{PK}, \mathsf{pk}, \mathsf{aux}, m)$ and for malicious parties sending the message $(\mathit{eval}, \mathsf{sid}, \mathsf{pk}, W, m)$ to $\mathcal{F}_{\mathsf{rvrf}}$.
- rVRF.Sign: For honest parties, replaced by sending the message $(\mathit{sign}, \mathsf{sid}, \mathcal{PK}, \mathsf{pk}, \mathsf{aux}, m)$ and for malicious parties replaced by picking a signature arbitrarily and sending the message $(\mathit{verify}, \mathsf{sid}, \mathcal{PK}, W, m, \mathsf{aux}, \sigma)$ to $\mathcal{F}_{\mathsf{rvrf}}$.
- rVRF.Verify: For all parties, replaced by sending the message $(\mathit{verify}, \mathsf{sid}, \mathcal{PK}, W, m, \mathsf{aux}, \sigma)$ to $\mathcal{F}_{\mathsf{rvrf}}$.

So, we can construct a simulator Sim in the $(\mathcal{F}^s_{\mathsf{com}}, \mathcal{F}_{\mathsf{rand}}, \mathcal{F}_{\mathsf{rvrf}})$-hybrid model.

[**Simulation of leader election:**] Sim generates the ring VRF verification key pk_i as defined in $\mathcal{F}_{\mathsf{rvrf}}$ in the real world whenever $\mathcal{F}_{\mathsf{bsle}}$ sends a registration message for an honest party P_i. It sets $\mathsf{eid} = (e, \mathcal{P}, \mathcal{D})$ as agreed in the real protocol. Here $\mathcal{D}$ is set to the equal power distribution. Then it sends $(\mathit{elect}, \mathsf{sid}, \mathsf{eid})$ to $\mathcal{F}_{\mathsf{bsle}}$ and waits for the election result. If Sim receives $(\mathit{result}, \mathsf{sid}, \mathsf{eid}, \mathtt{Leak}, \mathtt{Clue}, \mathtt{Leads}_m, \mathcal{I}_m)$, it means that the election for e has already been executed. In this case, it moves directly to Simulation of the preparation phase below. Otherwise, if $\mathcal{F}_{\mathsf{bsle}}$ sends $(\mathit{candidate_election}, \mathsf{sid}, \mathsf{eid}, \mathtt{Leads}_m, \mathcal{I}_m)$ where $\mathsf{eid} = (e, \mathcal{P}, \mathcal{D})$, Sim registers each honest party P_i in $\mathcal{P}$ to $\mathcal{F}_{\mathsf{rvrf}}$ if P_i is not registered before.

[**Simulation of the randomness phase:**] Sim simulates $\mathcal{F}_{\mathsf{rand}}$ as described in Figure 3. Whenever Sim receives a message $(\mathit{rand_reset}, \mathsf{sid}, \mathsf{eid}')$, it sends $(\mathit{re\text{-}elect}, \mathsf{sid}, \mathsf{eid})$ to $\mathcal{F}_{\mathsf{bsle}}$ and receives new election results $\mathtt{Leads}_m$ from $\mathcal{F}_{\mathsf{bsle}}$. When the adversary sends the message $(\mathit{rand_set}, \mathsf{sid}, \mathsf{eid}', k_{set})$, Sim finalizes the randomness as r_e, which is the randomness generated when $\mathsf{counter} =$

k_{set} as $\mathcal{F}_{\mathsf{rand}}$ does. Then, Sim sends the message $(\mathit{set_election}, \mathsf{sid}, k_{set}, \mathsf{eid})$ to $\mathcal{F}_{\mathsf{bsle}}$ to finalize the election in the ideal world. Then, it receives the message $(\mathit{initial_election}, \mathsf{sid}, \mathsf{eid}, \mathtt{Leak}, \mathtt{Clue}, \mathtt{Leads}_m)$ from $\mathcal{F}_{\mathsf{bsle}}$.

Before Sim sends the message $(\mathit{rand_leak}, \mathsf{sid}, \mathsf{eid}', r_e)$ to $\mathcal{A}$ while simulating $\mathcal{F}_{\mathsf{rand}}$, it sets up ring VRF outputs that are supposed to be generated by $\mathcal{F}_{\mathsf{rvrf}}$ with a given randomness of eid' for each party. This setup is executed by Sim in a way that in the real world, it gives a consistent result with $\mathtt{Leads}_m, \mathtt{Leak}$ as described below.

[Setting up random outputs of $\mathcal{F}_{\mathsf{rvrf}}$ before *rand_leak*:] Sim selects winner ring VRF outputs randomly such that $y_1 < y_2 < \ldots < y_\ell$ where $\ell_{\max} = nn_t$ and $y_i < c$ after $\mathcal{F}_{\mathsf{bsle}}$ sends $(\mathit{mal_leaders}, \mathsf{sid}, \mathsf{eid}, \mathtt{Leads}_m)$. We note that Sim does not know $|\mathtt{Leads}|$, where $\mathtt{Leads}$ is the output ELECT in $\mathcal{F}_{\mathsf{bsle}}$. Therefore, it considers the maximum possible size, which is $\ell_{\max}$. It stores $y_1, y_2, \ldots, y_{\ell_{\max}}$ to a history list $\mathsf{Log}_r[\mathsf{eid}, \mathsf{counter}]$, 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 preparation with the input r_e that it selected while simulating $\mathcal{F}_{\mathsf{rand}}$:

[Malicious rVRF evaluation preparation:] Given r_e and the list $\mathcal{I}_{\mathsf{P}_i}$ of elections that a malicious party P_i is selected as a leader, Sim creates a list $\mathsf{evaluations}_{\mathcal{A}}$ to later populate $\mathsf{evaluations}$ of $\mathcal{F}_{\mathsf{rvrf}}$. Then, Sim behaves as follows: Sim maps each election in $\mathcal{I}_{\mathsf{P}_i}$ to a random, unique index between $[1, n_t]$. We note that it is possible to map each election to a *unique* index because $|\mathcal{I}_{\mathsf{P}_i}| \leq n_t$, as can be seen in ELECT. For each $k \in \mathcal{I}_{\mathsf{P}_i}$, Sim gets the mapped index j, and sets $\mathsf{evaluations}_{\mathcal{A}}[r_e||j||e, \mathsf{pk}_i] := y_k$. If there exists W such that $\mathtt{anonymous_key_map}[r_e||j||e, W] = \mathsf{pk}_i$, i.e., the input $r_e||j||e$ has been evaluated before generating r_e, then Sim outputs an abort message ABORT-COLLISION and ends the simulation.

[Simulating $\mathcal{F}_{\mathsf{rvrf}}$:] After receiving $(\mathit{more_info}, \mathsf{sid}, \mathsf{eid}, \mathtt{Leak}, \mathtt{Clue}, \mathtt{Leads}_m)$ from $\mathcal{F}_{\mathsf{bsle}}$, $\mathcal{F}_{\mathsf{rvrf}}$ does the following for the leaked honest leaders. For each $k \leq \ell = |\mathtt{Leak}| \leq \ell_{\max}$ 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 $\mathcal{F}_{\mathsf{rvrf}}$ does. Then, it sets $\mathtt{anonymous_key_map}[r_e||j||e, W] := \mathsf{pk}_i$ and $\mathsf{evaluations}[r_e||j||e, W] := y_k$, where y_k is obtained from $\mathsf{Log}_r[\mathsf{eid}, k_{set}]$. If $\mathsf{evaluations}[r_e||j||e, W]$ has been already defined with another value, Sim outputs a message ABORT-COLLISION and ends the simulation.

For each $k \leq \ell$ such that $\mathtt{Leak}[k] = \perp$, Sim gets $j \leftarrow \mathtt{Clue}[k]$, then samples $W \leftarrow\!\!\$\ \mathcal{S}_W$, and runs $\mathsf{Gen}_{sign}(\mathcal{PK}, W, r_e||j||e)$, which outputs a signature which are indistinguishable from random. Then, it sets $\mathsf{evaluations}[r_e||j||e, W] := y_k$. It also stores W in $\mathtt{W}[k]$ and σ in $\mathtt{signature}[k]$. If $\mathsf{evaluations}[r_e||j||e, W]$ has been already defined with another value, Sim outputs an abort message ABORT-COLLISION and ends the simulation. Remark that Sim cannot set $\mathtt{anonymous_key_map}[r_e||j||e, W]$ at this point because honest leader of kth election is not known yet. However, it is not an issue because $\mathcal{A}$ does have any interface in $\mathcal{F}_{\mathsf{rvrf}}$ to learn it for any honest party.

Sim simulates honest ring VRF signature as described in the functionality. Whenever $\mathcal{F}_{\mathsf{rvrf}}$ receives a message $(\mathit{eval}, \mathsf{sid}, \mathsf{pk}_i, W, m)$ from the adversary, it checks if $\mathsf{evaluations}_{\mathcal{A}}[m, \mathsf{pk}_i]$ is defined before. If it is defined, it sets $\mathsf{evaluations}[m, W] = \mathsf{evaluations}_{\mathcal{A}}[m, \mathsf{pk}_i]$. Otherwise, it behaves as defined $\mathcal{F}_{\mathsf{rvrf}}$. Thus $\mathcal{A}$ obtains the ring VRF values which leads malicious parties to be the leaders as in $\mathtt{Leads}_m$.

For the rest of the queries for $\mathcal{F}_{\mathsf{rvrf}}$, Sim behaves as described in $\mathcal{F}_{\mathsf{rvrf}}$.

[Simulation of the preparation phase:] Sim simulates the leaked elections as follows: We denote the corresponding y-values of leaked elections 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 $i_k \in [1, \ell]$ 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 $(\mathit{sign}, \mathsf{sid}, \mathcal{PK}, \mathsf{pk}_i, r_e||j||e)$ to $\mathcal{F}_{\mathsf{rvrf}}$ where $j \leftarrow \mathtt{Clue}[i_k]$. In the end, Sim constructs a list $\mathcal{L}_i$ for each honest party P_i such that $\mathcal{L}_i = \{(r_e, j, e, W_{i_k}, \sigma_{i_k})\}$ similar to the preparation phase of Sassafras. Remark that $\mathcal{L}_i$ includes only the leaked tickets differently than Sassafras 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 Sassafras which includes tickets that are not sent to $\mathcal{A}$. For each election $k \leq \ell$ such that $\mathtt{Leak}[k] = \perp$, it gets $W = \mathtt{W}[k], \sigma = \mathtt{signature}[k]$. In the end, it adds the ticket (r_e, j, e, W, σ) to $\mathcal{L}^*$ where $j = \mathtt{Clue}[k]$.

[Simulation of the distribution phase:] Here, Sim only knows the owner of an honest party's y value if its election is leaked. However, it is not a problem because the honest leaders of non-leaked slots cannot be known by the adversary. Sim distributes the leaked elections 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 $\mathcal{F}^s_{\mathsf{com}}$. Note that an honest party sends n_t messages in the distribution phase. In order to simulate this, Sim sends some random messages $n_t - |\mathcal{L}_i|$-times to each party as $\mathcal{F}_{\mathsf{com}}$.

[Simulation of the submission phase:] During the submission, Sim behaves exactly the same as an honest party in Sassafras. 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 $\mathtt{Leads}_m[k] = \mathsf{P}_i \in \mathcal{P}_m$, Sim adds k to the aborted slot list $\mathcal{S}_{\mathsf{abort}}$. It sends $(\mathit{aborted_slots}, \mathsf{sid}, \mathcal{S}_{\mathsf{abort}})$ to $\mathcal{F}_{\mathsf{bsle}}$. After Sim sends the aborted slots to $\mathcal{F}_{\mathsf{bsle}}$, if the sortition of tickets changes in LOG and is finalized, then Sim outputs ABORT-INCONSISTENCY and the simulation ends.

[Simulation of revealing:] When $\mathcal{F}_{\mathsf{bsle}}$ sends $(\mathit{reveal}, \mathsf{sid}, \mathsf{eid}, k, \mathsf{P}_i)$, Sim obtains r_e from $\mathcal{F}_{\mathsf{rand}}$ run for eid'. It sends $(\mathit{sign}, \mathsf{sid}, \{\mathsf{pk}_i\}, W, r_e||j||e)$ to $\mathcal{F}_{\mathsf{rvrf}}$ where $j \leftarrow \mathtt{Clue}[k]$ and $r_e||j||e$ is from the kth sorted ticket with random-

ness r_e. Then it obtains $\hat{\sigma}$ from $\mathcal{F}_{\mathsf{rvrf}}$. Then, it publishes it via $\mathcal{F}^s_{\mathsf{com}}$. If $\texttt{anonymous_key_map}[r_e||j||e, W]$ has not defined in $\mathcal{F}_{\mathsf{rvrf}}$'s simulation due to the unknown leader of election k, Sim sets $\texttt{anonymous_key_map}[r_e||j||e, W] = \mathsf{pk}_i$.

[Simulation of Corruption:] When Sim receives a message $(\mathit{corrupted}, \mathsf{sid}, \texttt{history}[\mathsf{P}_i])$ from $\mathcal{F}_{\mathsf{bsle}}$, Sim should not reveal $\mathcal{A}$ that P_i has been simulated. The only place that P_i's behaviour has been simulated is the elections that Sim did not know the leaderships of P_i. Now, after the corruption, Sim learns all the leaderships of P_i in the past even for the candidate elections thanks to the data $\texttt{history}[\mathsf{P}_i]$ provided by $\mathcal{F}_{\mathsf{bsle}}$. Therefore, Sim can set $\texttt{anonymous_key_map}$ of P_i for the inputs related to the past batch elections and the candidate batch elections: $\texttt{history}[\mathsf{P}_i]$ consists of the tuples $(\mathsf{eid}, \mathsf{counter}, \mathcal{I}_{\mathsf{P}_i} = \{j : \mathsf{Leads}[j] = \mathsf{P}_i\})$ i.e., the elections that P_i is selected as a leader in the election number $\mathsf{counter}$. For each batch election $\mathsf{eid} = (e, \mathcal{P}, \mathcal{D}) \in \texttt{history}[\mathsf{P}_i]$, Sim obtains $\mathsf{Log}_r[\mathsf{eid}, \mathsf{counter}]$ that it stored during the setting up random outputs and $r = \mathcal{C}[\mathsf{counter}]$ from $\mathcal{F}_{\mathsf{rand}}$ run for eid. In the end, it executes the same steps in the malicious rVRF evaluation preparation with the randomness r and $\mathcal{I}_{\mathsf{P}_i}$ (See [Malicious rVRF evaluation preparation] above).

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 analyze 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 $\mathcal{F}_{\mathsf{rvrf}}$ but the output of $\mathcal{F}_{\mathsf{rvrf}}$ stays the same. It differently outputs Abort-Collision when a given input $(\mathcal{PK}, r_e||j||e)$ is assigned to a value in $\mathsf{evaluations}$ 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 $\mathcal{F}_{\mathsf{rvrf}}$ before e starts is negligible. The simulation of honest parties are indistinguishable since Sim behaves as described in Sassafras. Other 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_{sec}-slots, all honest tickets are going to be final in Δ_{sec}-slots after the submission phase. 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_{sec}-slots, they are considered as final for all honest parties and cannot be changed.

Additionally, we need to show that the output of Elect and the leaders selected by Sassafras are in the same distribution since during simulation the output of simulated Sassafras comes from Elect. They are in the same distribution because both Elect and Sassafras select leaders using the same weight distribution $\mathcal{D}[\mathsf{P}_i]$ and threshold parameter c. In Elect, a leader is selected if a random value ω_k is less than the threshold $\tau_i = \frac{c \cdot \mathcal{D}[\mathsf{P}_i]}{\max\{\mathcal{D}[\mathsf{P}_j]\}} = c$, which mirrors Sassafras's condition where a leader's ticket is winner if $\mathsf{rVRF.Eval} < c$. Thus, the probability of P_i being selected as a leader is identical in both processes.

Lastly, we need to show that the distribution of clue sets of each party generated by GetClue in Algorithm 2 is the same as the winner index set of each

party set. In Sassafras, for each party P_i, a random value y is generated by $\mathsf{rVRF.Eval}$. If $y < c$, an index $j \in \mathcal{C}$ is selected as a winner index uniformly at random. This process repeats across elections, resulting in the winner index set for P_i. In $\mathcal{F}_{\mathsf{bsle}}$, leaders are first selected with probability c, and then a clue $j \in \mathcal{C}$ is assigned uniformly at random from the available clues for the leader. Each assigned clue is removed to ensure uniqueness. Both processes involve:

- Selecting a party P_i with probability c.
- Assigning a unique index $j \in \mathcal{C}$ uniformly at random for each selected leader.

Thus, the probability of any index j being included in the clue set of P_i in $\mathcal{F}_{\mathsf{bsle}}$ is identical to the probability of j being included in the winner index set of P_i in Sassafras. This completes the proof of Theorem 1. □

E Parameter Selection for Deploying Sassafras

Here we provide a proof of Proposition 1.

Proof. We begin with the first part. We need to set the parameter $\mathtt{eplen}$ so that the blockchain security holds with probability at least $1-\epsilon/3$. The probability that the blockchain security in an epoch holds given that the election succeeded and Sassafras 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 1, we need $\exp(-\frac{(r-1)^2}{2r}\mathtt{eplen}) \leq \epsilon/3$ where $r = \frac{cn_t\beta}{\rho_{\max}\mathtt{eplen}}$ 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 $n_t = \lceil \frac{r'\rho_{\max}\mathtt{eplen}}{c\beta} \rceil$ which ensures that $r > r'$, but for this we need c.

Lemma 4. *If* $n_t = \lceil \frac{r'\rho_{\max}\mathtt{eplen}}{c\beta} \rceil$ *and*

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

then Sassafras is $\frac{\rho_{\max}}{\beta(1-\chi)}$*-unpredictable.*

Proof. Putting the inequality for c to the power of $\frac{\beta\chi+\rho_{\max}}{\beta\chi}$, we have $c^{1+\frac{\rho_{\max}}{\beta\chi}} \leq \chi \exp(-1 - \frac{\rho_{\max}\ln(3r'\rho_{\max}\mathtt{eplen}/\beta\epsilon+3/\epsilon)}{\beta\chi})$. Then

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

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

Re-arranging and taking logs give that $\ln(\chi/c) - 1 \geq \frac{\rho_{\max}\ln((3/\epsilon)n_t)}{\beta\chi}$ and so

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

By Lemma 2, except with probability $\epsilon/3$, Sassafras is $\frac{\rho_{\max}}{\beta(1-\chi)}$-unpredictable.

We need to set $\chi = 1 - \frac{\rho_{\max}}{\beta\theta}$ to obtain θ-unpredictability. With parameters satisfying these, each property holds except with probability $\epsilon/3$. By a union bound, all three properties hold except with probability ϵ: that the election does not abort, that Sassafras is θ-unpredictable, and that the chain properties hold. This gives the first part of the proposition. Note that c and so n_t do not depend on ϵ in an ideal way. n_t has a term $(1/\epsilon)^{\rho_{\max}/\beta\chi}$. This means if $\rho_{\max}/\beta\chi$ is large then it takes poly$(1/\epsilon)$ VRF attempts by each party in each epoch to get a small error probability where poly is a polynomial. However, in the case of $\mathcal{D}$ 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 $\chi, \texttt{eplen}, c, n_t$ as above for the case $\rho_{\max} = 1/n'$, $\theta = 2\rho_{\max}/\beta$, $\beta = \Theta(1)$ 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 did not give an explicit constant for the lower bound on $\texttt{eplen}$ and there are practical reasons for making epochs long, because we need to perform Sassafras in every epoch of $\mathfrak{B}$. 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 $c \leq \chi^{\frac{\beta\chi}{\beta\chi+\rho_{\max}}} \exp(-1 - \frac{\rho_{\max}(\ln(3r'\rho_{\max}\texttt{eplen}/\beta\epsilon+3/\epsilon)-1)}{\beta\chi+\rho_{\max}})$. Here $\beta\chi = \Theta(1)$ and so $\beta\chi + \rho_{\max} = \Theta(1)$. The first term, $\chi^{\frac{\beta\chi}{\beta\chi+\rho_{\max}}}$, is $\Theta(1)$. We have $3r'\rho_{\max}\texttt{eplen}/\beta\epsilon + 3/\epsilon = O(1/\epsilon) = O(\exp(Cn'))$ and so $(\ln(3r'\rho_{\max}\texttt{eplen}/\beta\epsilon + 3/\epsilon) - 1) = O(Cn')$. We obtain that $c \leq \exp(-O(C))$ and so we can take $c = \Omega_C(1)$. Then $n_t = \lceil \frac{r'\rho_{\max}\texttt{eplen}}{c\beta} \rceil = O_C(1)$. Any choice of $C > 0$ gives the final part of the proposition. $\square$