Accountable Secret Leader Election

Miranda Christ
Columbia University, New York, NY, USA

Kevin Choi
New York University, NY, USA

Walter McKelvie
Columbia University, New York, NY, USA

Joseph Bonneau
New York University, NY, USA
a16z crypto research, New York, NY, USA

Tal Malkin
Columbia University, New York, NY, USA

Abstract

We consider the problem of secret leader election with *accountability*. Secret leader election protocols counter adaptive adversaries by keeping the identities of elected leaders secret until they choose to reveal themselves, but in existing protocols this means it is impossible to determine who was elected leader if they fail to act. This opens the door to undetectable withholding attacks, where leaders fail to act in order to slow the protocol or bias future elections in their favor. We formally define accountability (in weak and strong variants) for secret leader election protocols. We present three paradigms for adding accountability, using delay-based cryptography, enforced key revelation, or threshold committees, all of which ensure that after some time delay the result of the election becomes public. The paradigm can be chosen to balance trust assumptions, protocol efficiency, and the length of the delay before leaders are revealed. Along the way, we introduce several new cryptographic tools including re-randomizable timed commitments and timed VRFs.

2012 ACM Subject Classification Security and privacy → Cryptography

Keywords and phrases Consensus Protocols, Single Secret Leader Election, Accountability

Digital Object Identifier 10.4230/LIPIcs.AFT.2024.1

Funding Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Government, DARPA, Andreessen Horowitz, the Algorand Foundation, Google, the National Science Foundation, or any other supporting organization.
Miranda Christ: Supported by NSF grants CCF-2107187, CCF-2212233, and CCF-2312242, by LexisNexis Risk Solutions, by the Algorand Centres of Excellence programme managed by Algorand Foundation, and by a Google CyberNYC Award.
Kevin Choi: Supported by DARPA Agreement HR00112020022 and NSF grant CNS-2239975.
Joseph Bonneau: Supported by DARPA Agreement HR00112020022, NSF grant CNS-2239975, and a16z crypto research.
Tal Malkin: Supported by NSF grant CCF-2312242, by the Algorand Centres of Excellence programme managed by Algorand Foundation, and by a Google CyberNYC Award.

1 Introduction

In proof-of-stake (PoS) blockchains, an essential challenge is randomly choosing a participant as the leader. The role of leaders varies by protocol, but they may perform tasks like compiling transactions into a block to propose to the network or voting to confirm proposed blocks. A desirable property of leader election is *secrecy*: nobody knows who the leader is

until they have revealed themselves in the course of fulfilling their duty. Secrecy is important in mitigating adaptive attacks, where an adversary may attempt to selectively corrupt (or launch a denial-of-service attack on) a leader before they can fulfill their duty. Adaptive attacks can be prevented in the so-called *you only speak once* (YOSO) [21] or *erasures* model, in which elected leaders delete their signing key prior to broadcasting their election. This ensures that when leaders become publicly known, it is already too late to corrupt them.

The first secret leader election (SLE) protocols (e.g., Algorand [23]) are *probabilistic*. Each user has an equal and independent chance of being assigned as leader, meaning there is inherently some probability of multiple leaders (or none) being elected. This undermines determinacy and adds overhead, motivating Protocol Labs [30] to propose *single* secret leader election (SSLE). Boneh et al. [8] were the first to formally define and construct SSLE protocols, which ensure that only a single leader (or another precise number) is elected. They proposed several constructions which have found their way to practice. Their DDH-based scheme was later adapted into Whisk, a practical SSLE protocol designed and proposed for use in Ethereum in EIP-7441 [22]. Other SSLE constructions have also been proposed with varying efficiency-security tradeoffs, including protocols with stronger security notions such as post-quantum security [10], adaptive security [12], UC security [13]; and a higher-communication protocol that better accommodates non-uniform stake distributions [4]. As secret leader election is necessary to attain fairness in the presence of adaptive corruptions, it has been studied in recent blockchain constructions like Ouroboros Crypsinous (implicitly, in the UC framework) and Fantômette (under the name of "delayed unpredictability") [26, 2]. An even stronger notion, wherein a leader's identity is kept private even *after* they publish a block, was proposed in [20].

Withholding attacks. Unfortunately, all known secret leader election schemes have the property that if the leader fails to perform its duty and announce itself, the other parties have no way of learning who the absentee leader was. This may not seem problematic, as leaders are typically rewarded for publishing blocks so there is an opportunity cost to failing to claim leadership. However, the choice to claim leadership or withhold creates an opportunity for elected leaders to introduce bias into the randomness used to elect future leaders. Wahrstätter [32] showed that a similar attack is indeed profitable today for nodes in Ethereum's RANDAO beacon chain, a core component of its consensus protocol.[1] Withholding attacks on RANDAO may be even more profitable if combined with manipulating randomness used by application-layer protocols which rely on it as a randomness beacon. This attack is not unique to Ethereum – in fact, Ferreira et al. [19] showed that a version of it inherently exists for *all* "cryptographic self-selection protocols" in which leaders are randomly chosen based on past values of the blockchain alone.

These attacks are troubling first because they incentivize leader absenteeism, which slows down the protocol (undermining liveness). Even worse, they threaten fairness, as validators with greater stake can gain a greater advantage through this strategy. Intuitively, this is because the chance of being elected once in an epoch is linear while the chance of being elected twice is quadratic in the stake. In the limit, these attacks create a rich-get-richer effect, motivating large coalitions and undermining decentralization.

[1] In this attack, validators withhold from contributing randomness to the beacon chain, foregoing some contribution reward to improve future election chances. Ethereum does not currently employ secret leader election, so the attack is detectable; however, there are proposals to do so.

Table 1 Summary of our constructions.

Construction	Section	P/S	Approach
SSLE + RRTC	4.1	Single	Delay-based. Users generate their key material as time-lock puzzle outputs and publish the corresponding inputs.
Timed weak VRFs	4.2	Probabilistic	Delay-based. Uses a new primitive called a timed VRF, where anyone (even without the secret key) can evaluate the VRF using a slow function.
Financial punishment	5.1	Either	Incentive-based.
Indexed VRFs	5.2	Probabilistic	Incentive-based. From hash functions and requires linear precomputation.
Indexed VRFs	5.2	Probabilistic	Incentive-based. From groups of unknown order. Does not require linear precomputation and allows all users to work in the same group.
Indexed VRFs	5.2	Probabilistic	Incentive-based. From trapdoor permutations. Less precomputation but each user must maintain its own trapdoor.
SSLE + ThrPKE	6	Single	Committee-based. Uses threshold cryptography to encrypt key material to a committee, which can reconstruct this key material if it is later withheld by a leader.

Our contributions. To address the problem of withholding attacks, we propose *accountable* secret leader election, in which the other validators eventually learn the identity of a negligent leader. Of course, it is critical that they do not learn the leader's identity too early, to maintain secrecy. We define both *strong* accountability (in which a withholding leader is identified precisely) and *weak* accountability, in which all participants who deviate from the protocol are identified (although we may not know which of them was the elected leader).

We then propose concrete constructions for both single and probabilistic SLE, summarized in Table 1. Our constructions fit into three distinct approaches:

1. **Timed Accountability** (delay-based). Any party can evaluate a delay function (e.g., verifiable delay function) to learn the leader's identity after some delay. The time delay of the slow function ensures that no party can learn a leader's identity until after its slot to perform its duty has passed, even with a dishonest majority.
2. **Key-Reveal Accountability** (incentive-based). A validator must reveal its identity in all past elections in order to either claim leadership or unstake. This approach is simple, but only gives weak accountability and relies on economic incentives.
3. **Threshold Accountability** (committee-based). A quorum of validators exceeding some threshold can work together to reconstruct the identity of a past leader. This approach relies on an honest majority of validators, as a malicious majority coalition could learn the identity of upcoming leaders prematurely.

In building these schemes, we identify and construct new cryptographic primitives. We define and construct *re-randomizable timed commitments*, used for single secret leader election, and *timed VRFs*, used for Algorand-style secret leader election. We also propose novel constructions of *indexed VRFs*, which were proposed by [17] with applications to Algorand-style secret leader election.

2 Preliminaries

We use λ to denote the security parameter, and $\mathsf{poly}(\lambda)$ and $\mathsf{negl}(\lambda)$ to denote polynomial and negligible functions of λ, respectively. We use $\xleftarrow{\$}$ (or $\xrightarrow{\$}$) to denote the output of a randomized algorithm, or sampling uniformly at random from a range. We assume all

adversaries are limited to running in probabilistic polynomial time (PPT) in the security parameter λ; some adversaries are further limited to running in $\sigma(t)$ steps on at most $p(t)$ parallel processors where noted. We let $[k]$ denote the set $\{1, \ldots, k\}$, and we use (a, b) to denote the set of integers x such that $a < x < b$. We use $\mathbf{v}$ to denote a vector $(v_1, \ldots, v_n)$, and write $\mathbf{v}[i]$ for the i^{th} component of $\mathbf{v}$.

Our schemes use a number of standard cryptographic primitives, including time-lock puzzles, threshold public-key encryption, verifiable delay functions (VDFs), and non-interactive zero-knowledge proofs. We describe the syntax and properties of these primitives in the full version.

3 A Taxonomy of Leader Election Protocols

Different leader election protocols may offer different properties:

Public vs. secret. In public leader elections, all participants learn the identity of the elected leader at once. This can be achieved by running any distributed randomness protocol [14, 25] and using the output to select a random leader. In this work we are only concerned with secret leader election. In *secret* leader election, by contrast, participants do not know who the elected leader(s) is/are until they reveal themselves. Secrecy helps prevent adaptive attacks, such as targeted corruption or denial-of-service attacks against upcoming leaders.

Number of leaders. One might want the protocol to elect just a *single* leader or a committee of k leaders. If used to elect a block proposer, one leader is typically desired.

Single vs. probabilistic number of leaders. Somewhat confusingly, a *single* leader election protocol always elects the same number of leaders, which may be one or a committee of size k. Boneh et al. [8] considered the case of electing a single leader with no variance, hence the name "single," even though they noted that their techniques naturally extend naturally to electing a larger committee. *Probabilistic* leader election protocols will elect k leaders on average but might elect more or fewer due to the randomness of the protocol. While single leader election is preferable, it is challenging to guarantee when combined with secrecy.

Weighting. Unweighted leader election protocols give each participant the same probability of being elected. Weighted protocols give different participants different probabilities of election, for example, proportional to their stake in a PoS setting.

In this paper, we only consider secret election protocols where each participant should be elected with equal probability. We consider both protocols where a single leader must always be elected (Single Secret Leader Election) and protocols where the number of leaders to be elected varies randomly (Probabilistic Secret Leader Election). All of our protocols generalize to electing a committee of any size.

Below, we recall the definition of Single Secret Leader Election from [8]. We modify the syntax slightly to make it accountable; in particular, we modify $\mathsf{Register}$ to take as input a list L of all users' registrations thus far and output a list L' with the new user's registration appended. Consequently, we also modify $\mathsf{Register}$ to take this list as input. We also combine Elect_1 and Elect_2 into a single protocol for ease of presentation, as [8] notes can be done for the shuffle-based protocol we build off of. We note that this syntax does not require leader uniqueness and encompasses some Probabilistic Secret Leader Election protocols as well; thus, we define it here as simply Secret Leader Election.

▶ **Definition 1** (Secret Leader Election [8]). *A secret leader election (SLE) protocol is a tuple of PPT algorithms and protocols* $\mathsf{SLE} = (\mathsf{SLE.Setup}, \mathsf{SLE.Register}, \mathsf{SLE.Elect}, \mathsf{SLE.Verify})$

SLE.Setup$(\lambda, \ell, N) \to \mathsf{pp}, \mathsf{sk}_1, \ldots, \mathsf{sk}_N, \mathsf{st}_0$ *is an algorithm that takes in the number of parties N and an optional lower bound ℓ on the number of required participants in each election, and outputs public parameters and secret keys for all parties.*

SLE.Register$(i, \mathsf{pp}, \mathsf{st}, L) \to \mathsf{st}', L'$ *is a protocol run by all parties. It takes as input the index i of the registering party, the public parameters* pp*, the current state* st*, and the registration list L. It outputs an updated state* st' *and an updated registration list L'. The registering party i receives a nonce k_i.*

SLE.Elect$(\mathsf{pp}, \mathsf{st}, R, i, k_i, \mathsf{sk}_i) \to 1/0, \pi/\perp$ *is an algorithm run by each party i to determine if they won the election. R is a randomness beacon value, k_i is the user's nonce, and* sk_i *is the user's secret key. If user i was elected, outputs $(1, \pi)$ where π is a proof that they won. Otherwise, it outputs $(0, \perp)$.*

SLE.Verify$(i, \mathsf{pp}, \mathsf{st}, R, \pi_i) \to 1/0$ *is an algorithm run by each party to verify that user i with proof π_i indeed won the election at state* st *with randomness R.*

Informally, an SLE protocol must be *unpredictable* in that an adversary controlling some subset of the parties cannot predict with non-negligible advantage which honest party was elected (in the event that an honest party is elected).

3.1 Single Secret Leader Election

Single secret leader election (SSLE) follows the syntax defined in Definition 1 and must satisfy *uniqueness, fairness, and unpredictability* as defined first in [8]. Uniqueness ensures that only a single party can be accepted as winner of each election. Fairness ensures that an adversary corrupting c out of N parties can win the election with probability at most $\frac{c}{N}$. Unpredictability ensures that the identity of the winner cannot be predicted before they reveal themselves.

Boneh et al. [8] defined these properties in terms of security games, where an adversary and challenger engage in the SSLE protocol. The adversary controls the corrupted participants, and the challenger controls the honest participants. The adversary can request that certain honest parties register for elections, and it can specify the inputs itself for corrupted parties to register.

Catalano et al. [12] identified a shortcoming in these definitions, which they subsequently fixed in [13]. We apply their fix and provide these modified definitions below. We defer further discussion of this modification to the full version. There, we prove that for a natural class of protocols, security under the original definitions of [8] implies security under the more stringent definitions of [13].

When we say that an SSLE protocol is *secure*, we mean that is is fair, unpredictable, and unique under the modified definitions below.

▶ **Definition 2** (Uniqueness [8, 13]). *Let* $\mathsf{UNIQUE}[\mathcal{A}, \lambda, \ell, N]$ *denote the uniqueness experiment with security parameter λ, played by an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ as follows:*

Setup phase. *$\mathcal{A}$ picks a number $c < N$ as well as a set of indices $M \subsetneq [N]$, $|M| = c$ of users to corrupt. The challenger $\mathcal{C}$ runs* $\mathsf{pp}, \mathsf{sk}_1, \ldots, \mathsf{sk}_N, \mathsf{st}_0 \leftarrow \mathsf{SSLE.Setup}(\lambda, \ell, N)$ *and gives $\mathcal{A}$ the parameters* pp*, state* st_0*, and secrets* sk_i *for $i \in M$.*

Elections phase. *Adversary $\mathcal{A}$ can choose any set of users to register for elections and for any number of elections to occur, where $\mathcal{A}$ plays the role of users $\mathcal{U}_i$ for $i \in M$ and $\mathcal{C}$ plays the role of the rest of the users. The challenger $\mathcal{C}$ also generates the election randomness $R \in \mathcal{R}$. To register a (corrupted or uncorrupted) user, $\mathcal{A}$ sends the index i of the user*

to $\mathcal{C}$, and $\mathcal{C}$ and $\mathcal{A}$ together run the protocol $\mathsf{Register}(i, \mathsf{pp}, \mathsf{st}, L)$ *to update the state to* st' *and the list to L'. If the* Register *protocol aborts, the game immediately ends with output 1.*

Each election begins with $\mathcal{C}$ generating $(b_i, \pi_i) \leftarrow \mathsf{SSLE.Elect}(\mathsf{pp}, \mathsf{st}', R, i, \mathsf{sk}_i)$ *for each uncorrupted user that has registered for the election. For each uncorrupted user i that has not registered for that election, it sets $(b_i, \pi_i) = (0, \perp)$. Finally, $\mathcal{C}$ sends (b_j, π_j) for each uncorrupted user to $\mathcal{A}$.*

Output phase. *For each election in the elections phase, $\mathcal{A}$ outputs values (b_i, π_i) for each $i \in M$. The experiment outputs 0 if for each election with randomness $R \in \mathcal{R}$ and state* st*, there is at most one user $\mathcal{U}_{i^*}$ (either corrupted or uncorrupted) who outputs $b_{i^*} = 1$ and π_{i^*} such that* $\mathsf{Verify}(i^*, \mathsf{pp}, \mathsf{st}, R, \pi_{i^*}) = 1$*. Otherwise the experiment outputs 1.*

We say an SSLE scheme is unique *if no PPT adversary $\mathcal{A}$ can win the uniqueness game except with negligible probability. That is, for all PPT $\mathcal{A}$ and for any $\ell < N$ the quantity*

$$\Pr\left[\mathsf{UNIQUE}[\mathcal{A}, \lambda, \ell, N] = 1\right] \leq \mathsf{negl}(\lambda).$$

If uniqueness only holds so long as there are at least t uncorrupted *users participating in each election, we say that the protocol is t-threshold unique.*

▶ **Definition 3** (Unpredictability [8, 13])**.** *Let* $\mathsf{UNPRED}[\mathcal{A}, \lambda, \ell, N, n, c]$ *denote the unpredictability experiment with security parameter λ, played by an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ as follows:*

Setup phase. *$\mathcal{A}$ picks a set of indices $M \subsetneq [N]$, $|M| = c$ of users to corrupt. The challenger $\mathcal{C}$ runs* $\mathsf{pp}, \mathsf{sk}_1, \ldots, \mathsf{sk}_N, \mathsf{st}_0 \leftarrow \mathsf{SSLE.Setup}(\lambda, \ell, N)$ *and gives $\mathcal{A}$ the parameters* pp*, state* st_0*, and secrets* sk_i *for $i \in M$.*

Elections phase. *Adversary $\mathcal{A}$ can choose any set of users to register for elections and for any number of elections to occur, where $\mathcal{A}$ plays the role of users $\mathcal{U}_i$ for $i \in M$ and $\mathcal{C}$ plays the role of the rest of the users. The challenger $\mathcal{C}$ also generates the election randomness $R \in \mathcal{R}$. To register a (corrupted or uncorrupted) user, $\mathcal{A}$ sends the index i of the user to $\mathcal{C}$, and $\mathcal{C}$ and $\mathcal{A}$ together run the protocol* $\mathsf{Register}(i, \mathsf{pp}, \mathsf{st}, L)$ *to update the state to* st' *and the list to L'. If the* Register *protocol aborts, the game immediately ends with output 1.*

Each election begins with $\mathcal{C}$ generating $(b_i, \pi_i) \leftarrow \mathsf{SSLE.Elect}(\mathsf{pp}, \mathsf{st}', R, i, \mathsf{sk}_i)$ *for each uncorrupted user that has registered for the election. For each uncorrupted user i that has not registered for that election, it sets $(b_i, \pi_i) = (0, \perp)$. Finally, $\mathcal{C}$ sends (b_j, π_j) for each uncorrupted user to $\mathcal{A}$.*

Challenge phase. *At some point after all users $\mathcal{U}_j$ for $j \in [n]$ have registered, $\mathcal{A}$ indicates that it wishes to receive a challenge, and one more election occurs. In this election, $\mathcal{C}$* does not *send (b_j, π_j) for each uncorrupted user to $\mathcal{A}$. Let $\mathcal{U}_i$ be the winner of this election. The game ends with $\mathcal{A}$ outputting an index $i' \in [N]$. If, for $\mathcal{U}_i$ elected in the challenge phase, $i \in M$, then the output of* $\mathsf{UNPRED}[\mathcal{A}, \lambda, \ell, N, n, c]$ *is set to 0. Otherwise,* $\mathsf{UNPRED}[\mathcal{A}, \lambda, \ell, N, n, c]$ *outputs 1 iff $i = i'$.*

We say that an SSLE scheme $\mathcal{S}$ is unpredictable if no PPT adversary $\mathcal{A}$ can win the unpredictable game with greater than negligible advantage when the winner of the election is uncorrupted. That is, for all PPT $\mathcal{A}$, for any $c \leq n - 2$, $n \leq N$, and for any $\ell < N$ the quantity

$$\Pr\left[\mathsf{UNPRED}[\mathcal{A}, \lambda, \ell, N, n, c] = 1 | i \in [N] \setminus M\right] \leq \frac{1}{n - c} + \mathsf{negl}(\lambda).$$

If unpredictability only holds for $c < t$ for some $t > 0$, we say that $\mathcal{S}$ is t-threshold unpredictable.

▶ **Definition 4** (Fairness [8, 13]). *Let* $\mathsf{FAIR}[\mathcal{A}, \lambda, \ell, N, n, c]$ *denote the uniqueness experiment with security parameter* λ*, played by an adversary* $\mathcal{A}$ *and a challenger* $\mathcal{C}$ *as follows:*

Setup phase. $\mathcal{A}$ *picks a set of indices* $M \subsetneq [N]$, $|M| = c$ *of users to corrupt. The challenger* $\mathcal{C}$ *runs* $\mathsf{pp}, \mathsf{sk}_1, \ldots, \mathsf{sk}_N, \mathsf{st}_0 \leftarrow \mathsf{SSLE.Setup}(\lambda, \ell, N)$ *and gives* $\mathcal{A}$ *the parameters* pp*, state* st_0*, and secrets* sk_i *for* $i \in M$*.*

Elections phase. *Adversary* $\mathcal{A}$ *can choose any set of users to register for elections and for any number of elections to occur, where* $\mathcal{A}$ *plays the role of users* $\mathcal{U}_i$ *for* $i \in M$ *and* $\mathcal{C}$ *plays the role of the rest of the users. The challenger* $\mathcal{C}$ *also generates the election randomness* $R \in \mathcal{R}$*. To register a (corrupted or uncorrupted) user,* $\mathcal{A}$ *sends the index* i *of the user to* $\mathcal{C}$*, and* $\mathcal{C}$ *and* $\mathcal{A}$ *together run the protocol* $\mathsf{Register}(i, \mathsf{pp}, \mathsf{st}, L)$ *to update the state to* st' *and the list to* L'*. If the* Register *protocol aborts, the game immediately ends with output 1.*

Each election begins with $\mathcal{C}$ *generating* $(b_i, \pi_i) \leftarrow \mathsf{SSLE.Elect}(\mathsf{pp}, \mathsf{st}', R, i, \mathsf{sk}_i)$ *for each uncorrupted user that has registered for the election. For each uncorrupted user* i *that has not registered for that election, it sets* $(b_i, \pi_i) = (0, \perp)$*. Finally,* $\mathcal{C}$ *sends* (b_j, π_j) *for each uncorrupted user to* $\mathcal{A}$*.*

Challenge phase. *At some point after all users* $\mathcal{U}_j$ *for* $j \in [n]$ *have registered,* $\mathcal{A}$ *indicates that it wishes to receive a challenge, and one more election occurs.* $\mathsf{FAIR}[\mathcal{A}, \lambda, \ell, N, n, c]$ *outputs 1 if there is no* $i \in [n] \setminus M$ *for which* $\mathsf{Verify}(i, \mathsf{pp}, \mathsf{st}, R, \pi_i) = 1$ *in the challenge election.*

We say that an SSLE scheme $\mathcal{S}$ *is* fair *if no PPT adversary* $\mathcal{A}$ *can win the fairness game with greater than negligible advantage. That is, if for all PPT* $\mathcal{A}, n \leq N, c < n$*, and for any* $\ell < N$*,*

$$|\Pr[\mathsf{FAIR}[\mathcal{A}, \lambda, \ell, N, n, c] = 1] - c/n| \leq \mathsf{negl}(\lambda).$$

If fairness only holds for $c < t$ *for some* $t > 0$*, we say* $\mathcal{S}$ *is* t*-threshold fair.*

[8] notes that these definitions can be easily extended to accommodate elections picking a *fixed* number of multiple leaders. This is in contrast to probabilistic leader election protocols, where the number of elected leaders may vary randomly from election to election.

3.2 Accountability for Single Secret Leader Election

Here, we define an additional property: accountability. An accountable scheme features a Recover protocol that informs all parties when an elected leader withholds. This allows the protocol to impose consequences on withholding parties, whereas with standard leader election, parties could withhold undetectably.

$\mathsf{SSLE.Recover}(\mathsf{pp}, \mathsf{st}, R, L, \mathcal{U}_i) \to 1/0/\perp$ takes as input a state st, a random beacon output R, a registration list L, and a user $\mathcal{U}_i$. If the output is 1, this means that $\mathcal{U}_i$ could generate a valid proof of leadership π_i with respect to state st with randomness R. If the output is 0, $\mathcal{U}_i$ could not generate such a proof. If the output is $\perp$, $\mathcal{U}_i$ must have deviated from the protocol in some way, and it is unknown whether they could prove leadership.

Both strong and weak accountability require that for any *honest* user, Recover outputs 1 if the user can claim the election, and 0 otherwise. Strong accountability additionally requires that Recover outputs 1 whenever a (possibly misbehaving) user could claim to be the winner of the election at state st with random beacon output R. For strong accountability, Recover never outputs $\perp$.

Weak accountability has the weaker condition Recover does not output 0 for any (possibly misbehaving) user that could claim to win the election (i.e., that user i can produce a proof π such that $\mathsf{Verify}(i, \mathsf{pp}, \mathsf{st}, R, \pi_i) = 1$). It may output either 1 or $\perp$ when the winning user misbehaves. Weak accountability is useful even in the case that the output is $\perp$, as this proves that the user in question must have misbehaved.

In our delay-based approaches, Recover is a slow non-interactive algorithm that can be run by any individual party in time much longer than the election protocol takes to run. This delay ensures unpredictability. In our committee-based approaches, Recover is an interactive protocol run by the committee that should succeed as long as a threshold of them participate honestly. In key-reveal approaches, Recover requires keys to be disclosed by participants after some number of elections. As withholding parties may also withhold their keys, this approach requires some incentive for revealing.

Defining Accountability

In defining accountability, we follow the style of definitions from [8] for the properties of uniqueness, unpredictability, and fairness.

The definitions of [8] involve a game where the adversary may corrupt some subset of the parties and run the SSLE protocol while controlling these parties and interacting with the honest parties. We essentially reproduce this game from these previous definitions and modify only the output phase to capture accountability, and the elections phase to apply a fix similar to that of [13]. That is, the adversary wins the game if it causes the protocol to abort. In the accountability game, the adversary aims to cause an election round where a corrupted party is elected and Recover fails to recover their nonce.

▶ **Definition 5** ((Strong, Weak) Accountability). *We let* $\mathsf{wACCOUNT}[\mathcal{A}, \lambda, \ell, N, n]$ *and* $\mathsf{sACCOUNT}[\mathcal{A}, \lambda, \ell, N, n]$ *denote the weak and strong accountability games played between an adversary* $\mathcal{A}$ *and a challenger* $\mathcal{C}$*:*

Setup phase. *$\mathcal{A}$ picks a set of indices $M \subsetneq [N]$ of users to corrupt. $\mathcal{C}$ runs $\mathsf{pp}, \mathsf{sk}_1, \ldots, \mathsf{sk}_N, \mathsf{st}_0 \leftarrow \mathsf{SSLE.Setup}(\lambda, \ell, N)$ and gives $\mathcal{A}$ the parameters pp, state st_0, and corrupted parties' secrets sk_i for $i \in M$.*

Elections phase. *$\mathcal{A}$ can choose any set of users to register for elections and for any number of elections to occur, where $\mathcal{A}$ plays the role of users $\mathcal{U}_i$ for $i \in M$ and $\mathcal{C}$ plays the role of the rest of the users. The challenger $\mathcal{C}$ also generates the election randomness $R \in \mathcal{R}$. To register a (corrupted or uncorrupted) user, $\mathcal{A}$ sends the index i of the user to $\mathcal{C}$, and $\mathcal{C}$ and $\mathcal{A}$ together run the protocol $\mathsf{Register}(i, \mathsf{pp}, \mathsf{st}, L)$ to update the state to st' and the list to L'. If the Register protocol aborts, the game immediately ends with output 1.*

Each election begins with $\mathcal{C}$ generating $(b_i, \pi_i) \leftarrow \mathsf{SSLE.Elect}(\mathsf{pp}, \mathsf{st}', R, i, \mathsf{sk}_i)$ for each uncorrupted user that has registered for the election. For each uncorrupted user i that has not registered for that election, it sets $(b_i, \pi_i) = (0, \perp)$. Finally, $\mathcal{C}$ sends (b_j, π_j) for each uncorrupted user to $\mathcal{A}$.

Output Phase. *For each election in the elections phase, $\mathcal{A}$ outputs values (b_i, π_i) for each $i \in M$.*

We say $\mathcal{A}$ violates correctness if it falsely blames an honest user in some way. More precisely, it violates correctness if and only if for some election that occurred with randomness R, state st, and registration list L at election time, there is some uncorrupted user $\mathcal{U}_{i'}$ such that either:

Falsely blames an honest user for withholding: $b_{i'} = 0$ *where* $(b_{i'}, \pi_{i'}) \leftarrow \mathsf{SSLE.Elect}(\mathsf{pp}, \mathsf{st}', R, i', \mathsf{sk}_{i'})$ *and* $\mathsf{SSLE.Recover}(\mathsf{pp}, \mathsf{st}', R, L, \mathcal{U}_{i'}) = 1$, *or*

Falsely blames an honest user for other misbehavior: $\mathsf{SSLE.Recover}(\mathsf{pp}, \mathsf{st}', R, L, \mathcal{U}_{i'}) = \bot$.

The strong accountability experiment outputs 1 if and only if $\mathcal{A}$ *violates correctness, or for some election with randomness* R*, state* st'*, and registration list at election time* L*, there is a corrupted user* $\mathcal{U}_{i^*}$ *who outputs* $b_{i^*} = 1$ *and* π_{i^*} *such that* $\mathsf{SSLE.Verify}(i^*, \mathsf{pp}, \mathsf{st}, R, \pi_{i^*}) = 1$ *and* $\mathsf{SSLE.Recover}(\mathsf{pp}, \mathsf{st}, R, L, \mathcal{U}_{i^*}) \neq 1$.

The weak accountability experiment outputs 1 if and only if $\mathcal{A}$ *violates correcness, or for some election with randomness* R *and state* st*, and registration list at election time* L*, there is a corrupted user* $\mathcal{U}_{i^*}$ *who outputs* $b_{i^*} = 1$ *and* π_{i^*} *such that* $\mathsf{SSLE.Verify}(i^*, \mathsf{pp}, \mathsf{st}, R, \pi_{i^*}) = 1$ *and* $\mathsf{SSLE.Recover}(\mathsf{pp}, \mathsf{st}, R, L, \mathcal{U}_{i^*}) = 0$.

We say an SSLE scheme is strongly/weakly accountable*, respectively, if no PPT adversary* $\mathcal{A}$ *can win the strong/weak accountability game except with negligible probability. That is, for all PPT* $\mathcal{A}$ *and for any* $\ell < N$*,*

$$\Pr\left[(\mathsf{s/w})\mathsf{ACCOUNT}[\mathcal{A}, \lambda, \ell, N, n] = 1\right] \leq \mathsf{negl}(\lambda).$$

If accountability only holds as long as there are at least τ *uncorrupted users participating in* $\mathsf{SSLE.Recover}$*, we say that the scheme is* τ*-threshold (weakly/strongly) accountable.*

3.3 Probabilistic Secret Leader Election

Recall that in probabilistic secret leader election (PSLE), the number of elected leaders is randomly distributed. Often, this number is one in expectation, and there is a tie-breaking procedure to agree on a single leader when the protocol elects multiple. PSLE encompasses a large class of protocols that lack unifying definitions, and the SSLE definitions presented above do not apply because of differing syntax and number of elected leaders. In this paper, we focus on an approach to PSLE which we call *Algorand-style PSLE*, an abstraction of the scheme used by Algorand [23].

Algorand-style PSLE. Each party holds a VRF public-secret key pair $(K_{\mathsf{pub}}, K_{\mathsf{priv}})$. A fresh random beacon value R is generated for each election and is available to all parties. A party wins an election if $(y, \pi) \leftarrow \mathsf{VRF.Eval}(K_{\mathsf{priv}}, R)$ and $y < T$, where T is a threshold controlling the expected number of parties elected. Parties can prove they have been elected by providing their VRF proof π, which other parties can verify using their public key. In the event that multiple parties' VRFs yield values under the threshold, the winner is chosen according to some tie-breaking rule.

By pseudorandomness of the VRF, prior to the winner revealing its proof it is not possible to tell who has won the election. Thus, if a winner never reveals their VRF output, it is impossible to tell that they should have won. Furthermore, if there is a tie, it is impossible to tell that the winning party withheld.

In Section 4.2, we show how to make Algorand-style PSLE accountable by using a notion called a *timed VRF* that we define. This allows all other parties to evaluate VRF outputs using a slow function (that preserves secrecy since it takes longer to evaluate than the election takes to run).

4 The Delay-based Approach

One approach is to replace cryptographic primitives with time-based variants. In general, time-based cryptographic variants feature a fast computation function (requiring a secret key) and an equivalent slow (inherently sequential) computation function which can be computed by anybody. Timed commitments are a classic example: the original committer can efficiently open the commitment, but any party can *force open* the commitment via a slow computation.

Time-based primitives can add accountability to protocols by keeping some information (such as the identity of a leader) secret in the short term while enabling eventual public computation for accountability. Security relies on the assumed computational delay, without any economic assumptions or an honest majority.[2] We show two new time-based cryptographic tools which can be used for secret leader election: re-randomizable timed commitments and timed verifiable random functions.

4.1 Accountable SSLE from re-randomizable timed commitments

We can construct an accountable SSLE protocol by replacing the commitments from the shuffle-based protocol of [8] with *re-randomizable timed commitments (RRTCs)*, which we define and construct here. An RRTCs commits to random keys in such a way that the commitments can be re-randomized, and for a limited time period the commitments are hiding. After this time period, anyone can open the commitment to learn the key. Our RRTC construction combines the DDH-based commitment scheme used in [8] with any time-lock puzzle in a natural way. We extend the definitions of re-randomizable commitments from [10] and timed commitments from [9].

▶ **Definition 6** (Re-Randomizable Timed Commitment (RRTC))**.** *An RRTC is a tuple of algorithms* (Setup, Commit, Randomize, Test, SlowOpen) *with the following syntax and properties:*

Setup$(\lambda, t) \to$ **pp***: outputs public parameters* pp*,*

Commit$(\mathbf{pp}, t) \to (c, k, \mathbf{aux})$*: outputs a commitment* c*, a key* k*, and auxiliary information* aux*,*

Randomize$(\mathbf{pp}, c) \to c'$*: outputs a re-randomization* c' *of the commitment,*

Test$(\mathbf{pp}, c, k) \to \{\mathbf{true}, \mathbf{false}\}$*: outputs* true *or* false *depending on whether* k *is a valid key for the (possibly re-randomized) commitment* c*,*

SlowOpen$(\mathbf{pp}, c, \mathbf{aux}) \to \tilde{k}$*: if* c *is an honestly formed commitment, outputs the key* $\tilde{k} = k$ *committed to by* c*.*

Correctness: If c is an honestly-formed (and possibly re-randomized) commitment to k, $\mathsf{Test}(\mathsf{pp}, c, k) = \mathsf{true}$.

Binding: It is computationally infeasible to find c, k, k' such that $\mathsf{Test}(\mathsf{pp}, c, k)$ and $\mathsf{Test}(\mathsf{pp}, c, k')$.

Hiding for random keys: The commitment reveals nothing about k.

Honest soundness: Given an honestly-formed (and possibly honestly re-randomized) commitment, SlowOpen recovers the committed key.

Re-randomizability: Randomize outputs another valid commitment to the same key.

[2] A dishonest majority assumption may seem odd since many applications of leader election exist in scenarios like consensus protocols, which require an honest supermajority. However, we note that there may be subtle differences in the majority's honesty, for example they might collude to learn a future leader's identity early but not to actively disrupt consensus.

Unlinkability: An adversary running in sequential time at most t cannot determine if $\tilde{c}$ is a re-randomization of commitment c_1 (committing to k_1) or of commitment c_2 (committing to k_2) given that $\tilde{c}$ is a re-randomization of one of them.

$\underline{\mathsf{Setup}(\lambda, t) \to \mathsf{pp}}$
$\mathsf{TLP.pp} \xleftarrow{\$} \mathsf{TLP.Setup}(\lambda, t)$
$\mathbb{G}, g, p \xleftarrow{\$} \mathsf{GroupGen}(\lambda)$
$\mathsf{pp} \leftarrow (\mathbb{G}, g, p, \mathsf{TLP.pp})$

$\underline{\mathsf{Commit}(\mathsf{pp}, t) \to (c, k, \mathsf{aux})}$
$x, y \xleftarrow{\$} \mathsf{TLP.GenRandPuzzle}(\mathsf{TLP.pp})$
$k_L, k_R \leftarrow H(x, y)$
$r \xleftarrow{\$} \mathbb{Z}_p$
$c = (g^r, g^{rk_L})$
$k \leftarrow k_L || k_R$
$\mathsf{aux} \leftarrow x$

$\underline{\mathsf{Randomize}(\mathsf{pp}, c) \to c'}$
$(u, v) := c$
$r' \xleftarrow{\$} \mathbb{Z}_p$
$c' \leftarrow (u^{r'}, v^{r'})$

$\underline{\mathsf{Test}(\mathsf{pp}, c, k) \to \{\mathsf{true}, \mathsf{false}\}}$
$(u, v) := c$
$k_L || k_R \leftarrow k$
return $(u^{k_L} \stackrel{?}{=} v)$

$\underline{\mathsf{SlowOpen}(\mathsf{pp}, c, \mathsf{aux}) \to \tilde{k}}$
$x \leftarrow \mathsf{aux}$
$\tilde{k} \leftarrow H(x, \mathsf{TLP.Solve}(\mathsf{TLP.pp}, x))$

Figure 1 Our re-randomizable timed commitment scheme.

Honest soundness. Our *honest soundness* property is a relaxation of the soundness property defined by Boneh and Naor [9]. Soundness requires that a recipient can be convinced that an honestly generated commitment is well-formed. In contrast, we require only that if the commitment is honestly formed, SlowOpen recovers the key. This relaxation is sufficient for a weak form of accountability where one is satisfied with punishing participants for publishing malformed commitments after the fact. Furthermore, one can efficiently prove that a puzzle failed to open correctly: Simply compute $\tilde{k} \leftarrow \mathsf{SlowOpen}(\mathsf{pp}, c, \mathsf{aux})$ and show that for $(u, v) = c$, $u^{\tilde{k}} \neq v$. If the commitment were well-formed, we would have $u^{\tilde{k}} = v$.

Our RRTC construction from DDH. [8] suggests the following construction of a re-randomizable commitment based on the Decisional Diffie-Hellman (DDH) assumption. Let $\mathbb{G}$ be a cyclic group of prime order p for which the (DDH) assumption holds, and let $g \in \mathbb{G}$ be a generator for this group. Their commitment to a uniformly random key k is (g^r, g^{rk}) for a uniform r. To open a commitment (u, v) given k, one checks that $u^k = v$. Furthermore, this commitment can be re-randomized by drawing a uniform r' and computing $(u^{r'}, v^{r'})$.

We define our scheme, shown in Figure 1, to be compatible with the commitments generated in the shuffle-based protocol of [8], which results in the slightly unnatural use of terms k_L, k_R. If the above commitment is used in the SSLE protocol, two parties may submit commitments to the same k. Therefore, k_L, k_R are introduced to detect when two commitments $(g^r, g^{rk}), (g^{r'}, g^{r'k})$ have the same nonce k and prevent such a registration. Each party reveals k_{iR} at registration time, and in order to open their commitment $c_i = (u, v)$ they must provide k'_i such that $k'_{iL}, k'_{iR} \leftarrow H(k'_i)$, $k'_{iR} = k_{iR}$, and $u^{k_{iL}} = v$. Only a party that has revealed a matching k'_{iR} can claim to be leader for that commitment. Thus, if two parties submit commitments to the same k_L, the hash function ensures either that one of them can never claim an election, or they must have the same k_R. We check for duplicate k_R's at registration time to rule out this latter case. We also note that Commit could instead output (g, g^{k_L}), which would still be hiding and binding for random keys. However, this would allow an adversary to distinguish between commitments that have and have not been re-randomized, and this scheme would not achieve unlinkability as defined in [10].

Achieving soundness using NIZKs. One could modify our scheme to satisfy the stronger notion of soundness from [9] by requiring the committer to provide a non-interactive zero knowledge proof π that c is a commitment to $H(x, \mathsf{TLP.Solve}(x))$ for $x = \mathsf{aux}$. This could be achieved could use a generic zk-SNARK; for this, it is convenient to use a VDF as the time-lock puzzle to avoid heavy computation in verifying its output. Designing a sound RRTC without the use of generic SNARKs is an interesting direction for future work.

The BEHG protocol. We now briefly recall the "high-communication" variant of the shuffle-based SSLE scheme (also known as the BEHG protocol) from Boneh et al. [8], which we describe generically for any re-randomizable commitment scheme. In this variant, we maintain a public list of commitments belonging to the parties in the election. When a new user registers, they generate a nonce and corresponding commitment. They re-randomize the commitments in the list, shuffle them, and insert their own commitment at a random location. This new user posts a NIZK proof that they shuffled the list correctly: each commitment in the old list appears exactly once, re-randomized, in the new list.

The algorithms for this scheme do the following. Setup creates an empty to-be-shuffled list l, to which commitments will be added when users register; (in our modified protocol, it also creates an empty not-to-be-shuffled list L). In Register, the registering user samples a random key k_i and splits its hash into two parts $k_{iL}, k_{iR} \leftarrow H(k_i)$, then computes a re-randomizable commitment c_i to k_{iL}. It re-randomizes the commitments in l and shuffles l, then inserts c_i into l at a random location. It also provides a NIZK proof of honest shuffling. Each user then examines the current state to get the list l and checks that the list was properly shuffled using this proof. It also checks that none of the keys k_{jR} are duplicated. If either of these checks fails, the list is reverted to its most recent state and the protocol continues. Elect uses a random beacon value R given as input to choose a random commitment (i.e. the winning commitment) in l. If run by the user that submitted this commitment, including its key k_i as input, it outputs a proof π_i that includes an opening proof for that commitment, allowing the user to claim that election. Verify can be run by any user, and it checks if the revealed (by $\mathcal{U}_i$) key $\tilde{k}$ is consistent with $\mathcal{U}_i$'s k_{iR} from the registration list L and if $\tilde{k}$ is consistent with the winning commitment.

A delay-based accountable SSLE scheme. Next, we'll show that we can slightly modify this scheme to be weakly or strongly accountable. Our scheme is described below, and further detail is given in the full version.

The main modification is to use our RRTC instead of their original commitment scheme; our RRTC is exactly the same as their scheme except that our key k_i is chosen as the hash of an input-output pair to a time-lock puzzle and our k_{iL} and k_{iR} are derived directly from k_i instead of $H(k_i)$ (as there is no need to hash twice). The weakly accountable version uses our RRTC with honest soundness, and the strongly accountable version uses our RRTC with the additional proof of well-formedness to achieve full soundness. We detail our modifications below; they include a small modification to SSLE.Register and defining a Recover algorithm.

We make a small modification to SSLE.Register: when a user $\mathcal{U}_i$ registers, it must add its registration to L. L is a list of auxiliary information that each party adds to when registering that SSLE.Recover will use. L is separate from the shuffled list, and L is not shuffled or used in the elections. To be more specific, SSLE.Register generates the commitment (c, k, aux) using RRTC.Commit and appends $(k_{iR}, c, \mathsf{aux}, \mathsf{id}_i)$ to L, where id_i is a string representing its identity. It then re-randomizes c using RRTC.Randomize to obtain $(g^{r_i}, g^{r_i k_{iL}})$ and continues as in the original protocol from [8]. For the strongly accountable version, during Register all

users check the proof of commitment well-formedness and reject the registration if it fails. To claim an election for a chosen commitment (u, v), a party provides $k' = k'_L||k'_R$ and (x', y') such that $k' = H(x', y')$, $u^{k'_L} = v$, and k'_R matches the on-chain k_R from that party's initial registration.

We define $\mathsf{Recover}(\mathsf{pp}, \mathsf{st}, R, L, \mathcal{U}_i) \to 1/0/\perp$ for this scheme as follows. $\mathsf{Recover}$ first parses st to obtain the current shuffled list of entries and uses R to choose the winning commitment com^*. It then iterates through L (in case $\mathcal{U}_i$ registered multiple times). For each entry $(k_{iR}, c, \mathsf{aux}, \mathsf{id}_i)$ in L added by $\mathcal{U}_i$, it computes $\tilde{k} \leftarrow \mathsf{SlowOpen}(\mathsf{pp}, c, \mathsf{aux})$. It checks that $\mathsf{Test}(\mathsf{pp}, c, \tilde{k}) = \mathsf{true}$; if not, it moves onto the next entry in L added by $\mathcal{U}_i$. If it continues, it parses $\tilde{k_L}||\tilde{k_R} \leftarrow \tilde{k}$. If $\mathsf{Test}(\mathsf{pp}, \mathsf{com}^*, \tilde{k}) = \mathsf{true}$, it outputs 1. If $\mathsf{Test}(\mathsf{pp}, \mathsf{com}^*, \tilde{k}) = \mathsf{false}$, it continues. After it has iterated through all of L, it outputs $\perp$ if it observed that $\mathsf{Test}(\mathsf{pp}, c, \tilde{k}) = \mathsf{false}$ for any c added to L by $\mathcal{U}_i$. Otherwise, it outputs 0.

▶ **Theorem 7.** *The high-communication shuffling-based SSLE scheme from [8] is a weakly accountable SSLE scheme when instantiated with our RRTC scheme as described above (assuming an adversary that runs in sequential time less than t). It is a strongly accountable SSLE scheme when we modify our RRTC scheme to require the committer to provide a NIZK proof that the commitment was honestly generated.*

The proof of this theorem is given in the full version.

Commitment expiry. A timed commitment is no longer hiding after enough time has passed for $\mathsf{SlowOpen}$ to be evaluated. Therefore, this scheme is not secret if registrations remain in the list for time greater than t, where t is the runtime of $\mathsf{SlowOpen}$. The most natural solution is to run the protocol in epochs of length less than t. At the beginning of each epoch, the list is cleared and all users must re-register; this ensures that commitments do not stay in the list for too long. Although this re-registration increases communication, this increase can be traded off with the delay required to recover withholders' identities. If one is willing to increase this delay t, one can tolerate longer epochs and fewer re-registrations.

Protocol optimizations

Efficient TLP generation. Preparing a commitment requires generating a time-lock puzzle pair (x, y). For classic repeated-squaring time-lock puzzles, Rivest et al. proposed generating $(x, y = x^{(2^t)} \pmod N)$ by taking advantage of the trapdoor $\varphi(N)$ to compute a reduced exponent $e = 2^t \pmod{\varphi(N)}$. This approach also applies for modern VDFs in an RSA group [33]. The drawback of this approach is that each puzzle must use its own modulus N, making efficient hardware implementation more difficult.

A better approach utilizes *re-randomizable VDFs* [1]. Observe that users do not need to compute a TLP on a specific value; rather, a TLP for a random x is sufficient. Re-randomizable VDFs (of which repeated-squaring VDFs are a natural example) enable computing random input/output pairs given a single precomputed value $(g, h = g^{(2^t)})$. Observe that (g^α, h^α) is also a valid VDF input/output pair for any α. Hence, users can generate a puzzle by choosing a random α and setting $(x, y) \leftarrow (g^\alpha, h^\alpha)$.

Outsourcing TLP computation. In order to learn the nonce for a pair (c, aux), or discover that (c, aux) was malformed, one must compute the output of a TLP on aux. As this computation is slow by design, it is desirable to have a mechanism to outsource this task. Since all registrations (c, aux) are posted on-chain, we could allow any member of the public to compute these TLP outputs on the participants' behalf. This party could report a malformed commitment by posting this TLP output and a proof of correctness on-chain;

this is especially convenient if one uses a VDF as the TLP. The protocol can then *slash* (confiscate the deposited capital of) the offending participant, and the reporter could receive some of the slashed stake.[3]

4.2 Accountable PSLE from timed VRFs

In Algorand-style PSLE [23], each party holds a VRF public-secret key pair $(K_{\mathsf{pub}}, K_{\mathsf{priv}})$. K_{pub} is known to all. Here, we make the modeling assumption that a fresh random beacon value R is generated for each election; in Algorand's actual protocol, R is a function of the last block produced. R is available to all parties in the election. A party wins an election if $(y, \pi) \leftarrow \mathsf{VRF.Eval}(K_{\mathsf{priv}}, R)$ and $y < T$, where T is a threshold specifying the expected number of parties elected. Parties can prove they have been elected by providing their VRF proof π, which other parties can verify using their public key. By pseudorandomness of the VRF, prior to the winner revealing its proof it is not feasible to learn who has won the election.

In the event that multiple parties' VRFs yield values under the threshold, the party with the lowest VRF output is chosen as the winner. An unintended consequence of this tie-breaking rule is a way for malicious participants to bias the election. Because the randomness R is a function of the previous leader's identity, an adversary that controls two parties whose VRF outputs y_1 and y_2 are both below T may choose which party's output to reveal in order to generate more favorable randomness for the next election. That is, if $y_1 < y_2$, the adversary might choose *not* to reveal y_1 so that its party with y_2 can propose the next block. Because other parties cannot compute the VRF, they cannot learn that the party with y_1 withheld. This attack is therefore undetectable.

We provide accountability by replacing the VRF in Algorand-style elections with a *timed VRF*, a new primitive which we define. A timed VRF has a slow open function that can be run by anyone, without knowledge of the secret key. The slow open function requires a lengthy computation, and before this delay the VRF retains its pseudorandomness. Timed VRFs share some similarities with VDFs, but they are pseudorandom and evaluation is fast given a private key. When we replace the VRFs in Algorand-style PSLE with timed VRFs, all parties eventually learn all other parties' VRF outputs for all elections. Thus, all parties that have withheld can be identified.

Here, we delineate the formal properties of a timed VRF. We can also define timed weak VRFs (akin to weak VRFs [11]), which are only pseudorandom on randomly chosen inputs:

Our timed VRF definition is (adapted from [27, 16]):

▶ **Definition 8** (Timed VRF). *A Timed VRF is a tuple of algorithms* (KeyGen, Eval, SlowEval, Verify) *where:*

KeyGen$(\lambda, t) \to (K_{\mathsf{pub}}, K_{\mathsf{priv}})$*: generates a key pair which allows public evaluation with a time delay of* t*.*

Eval$(K_{\mathsf{priv}}, x) \to (y, \pi)$*: outputs a value* y *using the private key* K_{priv}*, and a correctness proof* π*. This function should be fast to evaluate.*

SlowEval$(K_{\mathsf{pub}}, x) \to (y, \pi)$*: outputs a value* y *for input* x *without using the private key by completing a sequential computation.*

Verify$(K_{\mathsf{pub}}, x, y, \pi) \to 1/0$*: checks if* y *is the correct evaluation of* x *under* K_{pub} *given proof* π*.*

[3] While we leave the exact incentive design as an open question, it is important that the reporter receive *some* but not all of the slashed stake. Observe that a malicious user can prove their commitment is malformed without executing a slow computation. If the reporter receives all of the slashed stake, a malicious user could report themselves and effectively suffer no penalty.

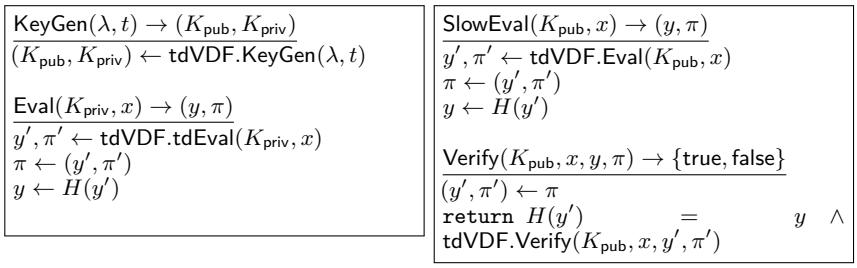

Figure 2 Our timed VRF scheme from a trapdoor VDF.

Correctness: For any honestly-formed key pair $(K_{\mathsf{pub}}, K_{\mathsf{priv}})$ and input x, given outputs $(y, \pi) \leftarrow \mathsf{Eval}(K_{\mathsf{priv}}, x)$ and $(y', \pi') \leftarrow \mathsf{SlowEval}(K_{\mathsf{pub}}, x)$, it should hold that both $\mathsf{Verify}(K_{\mathsf{pub}}, x, y, \pi)$ and $\mathsf{Verify}(K_{\mathsf{pub}}, x, y', \pi')$ return true.

Unique provability: For every K_{pub}, x, no PPT adversary can find two outputs (y_1, π_1) and (y_2, π_2) such that $y_1 \neq y_2$ and both $\mathsf{Verify}(K_{\mathsf{pub}}, x, y_1, \pi_1)$ and $\mathsf{Verify}(K_{\mathsf{pub}}, x, y_2, \pi_2)$ return true.

Strong t-pseudorandomness: For all PPT adversaries $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ where $\mathcal{A}_1$ is t-sequential, it holds that:

$$\Pr\left[b = b' \;\middle|\; \begin{array}{c} (K_{\mathsf{pub}}, K_{\mathsf{priv}}) \leftarrow \mathsf{GenKey}(\lambda, t) \\ (x_*, \sigma) \leftarrow \mathcal{A}_0^{\mathsf{Eval}(K_{\mathsf{priv}}, \cdot)}(K_{\mathsf{pub}}) \\ b \xleftarrow{\$} \{0,1\} \\ y_0 \leftarrow \mathsf{Eval}(K_{\mathsf{priv}}, x_*) \\ y_1 \xleftarrow{\$} \mathcal{Y} \\ b' \leftarrow \mathcal{A}_1^{\mathsf{Eval}(K_{\mathsf{priv}}, \cdot)}(\sigma, y_b) \end{array}\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

Weak t-pseudorandomness: For all PPT, t-sequential adversaries $\mathcal{A}$, it holds that:

$$\Pr\left[b = b' \;\middle|\; \begin{array}{c} (K_{\mathsf{pub}}, K_{\mathsf{priv}}) \leftarrow \mathsf{GenKey}(\lambda, t) \\ x_* \xleftarrow{\$} \mathcal{X} \\ b \xleftarrow{\$} \{0,1\} \\ y_0 \leftarrow \mathsf{Eval}(K_{\mathsf{priv}}, x_*) \\ y_1 \xleftarrow{\$} \mathcal{Y} \\ b' \leftarrow \mathcal{A}^{\mathsf{Eval}(K_{\mathsf{priv}}, \cdot)}(\sigma, y_b) \end{array}\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

Timed weak VRFs from trapdoor VDFs

We present an efficient construction, given in Figure 2, of a timed weak VRF from a *trapdoor VDF*, as formalized by Wesolowski [33]. Wesolowski observes that while repeated squaring is conjectured to be an inherently sequential function in a group of unknown order, given the group order it becomes efficient as the exponent 2^t can be reduced modulo the group order. Therefore, the group order serves as a trapdoor enabling efficient computation of the VDF for arbitrarily high delay parameters. The RSA group[4] $(\mathbb{Z}/N)^*$ is a natural example with the group order $\varphi(N)$ serving as the trapdoor.

[4] Note that using $(\mathbb{Z}/N)^*$ is insecure for VDFs as the low-order assumption does not hold, instead the group of quadratic residues $\mathbb{QR}_N$ or the group $\mathbb{G}^+ = (\mathbb{Z}/N)^*/\{\pm 1\}$ should be used [7, §6].

The weak pseudorandomness of the construction given in Figure 2 follows almost directly from its unpredictability when considered as a VDF. Unpredictability of a VDF requires that an adversary cannot predict the output on a random input; [6] notes that hashing the output of a function that is unpredictable in this sense yields a pseudorandom output in the random oracle model. Thus, we obtain a function whose output is pseudorandom on a random input which is exactly weak pseudorandomness. We note that in our model of Algorand-style PSLE, the input to the VRF is a random value R, and thus weak pseudorandomness is sufficient. Furthermore, as long as the distribution of the input x has λ bits of min-entropy, $H(x)$ is uniform in the random oracle model.

5 The Key-disclosure Approach

We can achieve accountability if all users reveal their secret keys after each election, enabling any party to recompute what the results should have been and detect any deviation. Of course, the critical question is how we can ensure that users actually disclose their key material.

5.1 Key disclosure via slashing

The most general approach is to compel users to publish key material under the the threat of *slashing*, or losing deposited capital, if key material is not properly disclosed. This approach works naturally for staking protocols in which users have committed a pool of deposited stake to participate. One option is for users to disclose and re-key at regular intervals (e.g. after each epoch). This adds continual overhead, but many protocols already impose a similar requirement of per-epoch setup. Another option is to have users disclose only when they attempt to withdraw their deposited capital and stop acting as participants (unstaking). This reduces the frequency and overhead of key disclosure but means it will take longer to detect misbehavior. Finally, users might disclose whenever they are elected as leader or otherwise stand to earn rewards. This approach works naturally with protocols employing the YOSO paradigm [21] to defend against adaptive adversaries, in which case keys are one-time use by design.

Whenever users disclose keys, a waiting period is needed before any withdrawal of staked capital to ensure adequate time for auditors to check for misbehavior using the disclosed key, for example by re-deriving all VRF values the user should have computed under this key and seeing if the users did not act as leader when they were expected to. They also require careful analysis of incentives; if slashing penalties are too low, attackers may be willing to absorb the loss as part of an attack. We leave detailed mechanism design of this approach as future work but note that it is a simple and potentially powerful tool.

5.2 Implicit key disclosure from indexed VRFs

We can improve the approach of disclosing key material whenever a user is elected leader by using an *indexed VRF* (iVRF) instead of a VRF in Algorand-style leader election. This idea of an iVRF was formalized by Esgin et al. [17], though the construction was used earlier by Azouvi et al [3]. The iVRF Eval function takes as input the *index* i, and proving a VRF value for index i reveals the key material for all indices $j \leq i$. In a distributed consensus setting, the index is the round number, observing that (for protocols with per-round finality), there is no downside to revealing key material for past rounds. Interestingly, while Esgin et

$\underline{\mathsf{KeyGen}(\lambda, m) \to (K_{\mathsf{pub}}, K_{\mathsf{priv}})}$
$K_{\mathsf{priv}} \xleftarrow{\$} \{0,1\}^{\lambda}$
$K_{\mathsf{pub}} = K^m = H^m(K_{\mathsf{priv}})$

$\underline{\mathsf{Eval}(K_{\mathsf{priv}}, x, i, m) \to (y, \pi)}$
$\pi = K^i := H^{m-i}(K_{\mathsf{priv}})$
$y = H(K^i, x)$

$\underline{\mathsf{Verify}(K_{\mathsf{pub}}, x, i, y, \pi) \to \{\mathsf{true}, \mathsf{false}\}}$
`return` $H^i(\pi) = K_{\mathsf{pub}} \wedge y = H(\pi, x)$

Figure 3 An indexed VRF from hash functions, due to Esgin et al. [17].

$\underline{\mathsf{Setup}(\lambda, m) \to \mathsf{pp}}$
$\mathbb{G}, g, e \xleftarrow{\$} \mathsf{GroupGen}(\lambda)$
$\mathbf{g} \leftarrow \left\{ g^{e^i} \right\}_{i=0}^{m}$
$\mathsf{pp} \leftarrow (\mathbb{G}, g, \mathbf{g})$

$\underline{\mathsf{KeyGen}(\lambda, \mathsf{pp}) \xrightarrow{\$} (K_{\mathsf{pub}}, K_{\mathsf{priv}})}$
$K_{\mathsf{priv}} = \alpha \xleftarrow{\$} \mathcal{B}$
$K_{\mathsf{pub}} = (\mathbf{g}[m])^{\alpha}$

$\underline{\mathsf{Eval}(\mathsf{pp}, K_{\mathsf{priv}}, x, i, m) \to (y, \pi)}$
$(\mathbb{G}, g, \mathbf{g}) \leftarrow \mathsf{pp}$
$\pi = K^i := (\mathbf{g}[m-i])^{\alpha}$
$y = H(K^i, x)$

$\underline{\mathsf{Verify}(K_{\mathsf{pub}}, x, i, y, \pi) \to \{\mathsf{true}, \mathsf{false}\}}$
`return` $\pi^{e^i} = K_{\mathsf{pub}} \wedge y = H(\pi, x)$

Figure 4 An indexed VRF from groups of unknown order. The space $\mathcal{B}$ must be large enough to ensure g^{α} is indistinguishable from random for $\alpha \xleftarrow{\$} \mathcal{B}$. Statistical security can be achieved with $|\mathcal{B}| \geq 2 \cdot |\mathbb{G}|$; whereas $|\mathcal{B}| \geq 2^{2\lambda}$ under the SEI assumption [15].

al. proposed indexed VRFs due to their simplicity and potential for quantum-resistance, we observe here that they also provide accountability by enabling observers to compute a user's past VRF values each time they publish a VRF output in any round.

We first recall Esgin et al.'s construction from hash chains [17] (Figure 3), then present two novel constructions of iVRFs. Our iVRF from groups of unknown order decreases precomputation cost relative to [17] and allows all users to work in the same group (Figure 4). Our construction based on a trapdoor permutation (Figure 5) offers the novel advantage that users can maintain the same key *indefinitely* (e.g. for an unlimited number of indices) with no precomputation.

5.2.1 Indexed VRFs from hash functions

The Esgin et al. [17] indexed VRFis similar to classic notions of hash chains [24, 28], as shown in Figure 3. Essentially, each user computes a chain of round-specific keys $K^i = H^{m-i}(K_{\mathsf{priv}})$ for round i. The total number of indices supported, m, should be chosen to cover, say, one epoch. Note that revealing $\pi_i = K^i$ as a proof for round i makes computing prior values easy for indices $j \leq i$: simply compute $K^j = H^{i-j}(K^i)$.

Naively, computing and verifying this proof requires computing $O(m)$ hashes each, though some tradeoffs are available if the prover stores some intermediate keys K^j to compute proofs with $O(\sqrt{m})$ computation and storage. Esgin et al. also describe tree-based variants enabling logarithmic verification costs, though all of them appear to require $O(m)$ computation during KeyGen.

5.2.2 Indexed VRFs from groups of unknown order

We present an indexed VRF construction in Figure 4 based on groups of unknown order, without assuming a trapdoor. This can have practical benefits in enabling all computation to be performed in one group. We replace the hash function in the above construction

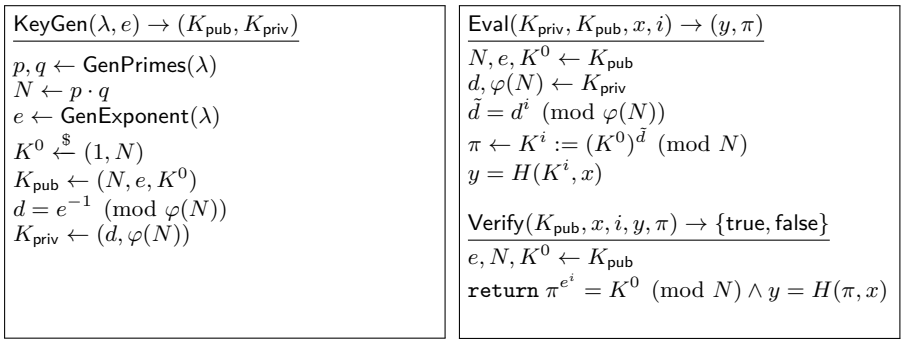

Figure 5 An indexed VRF from trapdoor permutations. We present the scheme here working in the RSA group $(\mathbb{Z}/N)^*$, though the idea is generic to any trapdoor permutation. As presented, this scheme requires linear work (in i) per verification. This can be reduced to constant cost by caching the latest value of K^i after each evaluation.

with computing e^{th} roots modulo N. However, in this protocol we do not assume users know the trapdoor, so naively they must precompute the entire chain of keys K^i, as with the hash-based indexed VRF. Implemented in this way, this approach has no clear advantage over the hash-based approach. However, notice that the precomputed chain $\mathbf{K} = \{K_{\mathsf{priv}}, (K_{\mathsf{priv}})^e, (K_{\mathsf{priv}})^{e^2}, \dots, \}$ can be computed only once in global setup, and then *randomized* by each user as needed. This randomization is straightforward: given a precomputed chain $\mathbf{g} = \{g_0 = g, g_1 = g^e, g_2 = g^{e^2}, \dots\}$, a user can sample[5] a random exponent $\alpha \leftarrow \mathcal{B}$ and compute a randomized chain $\mathbf{g}' = \{(g_0)^\alpha, (g_1)^\alpha = (g^\alpha)^e, (g_2)^\alpha = (g^\alpha)^{e^2}, \dots\}$.

This approach removes the linear precomputation and supports efficient proof computation. Assuming access to the precomputed string $\mathbf{g}$, needed elements of a user's randomized chain $\mathbf{g}'$ can be produced on-demand as $\mathbf{g}'[j] = (\mathbf{g}[j])^\alpha$. It is also possible to provide efficient proofs of correctness for any evaluation, by adding a succinct proof of exponentiation showing that $\pi^i = K_{\mathsf{pub}}$. Since the group order is unknown, either Wesolowski proofs [33] or Pietrzak proofs [29] may be used.

5.2.3 Indexed VRFs from trapdoor permutations

We present a new construction in Figure 5 based on trapdoor permutations. Essentially, we replace the hash function in the hash-based construction with a trapdoor permutation, such that computing forwards on the chain involves inverting the permutation with the help of the trapdoor, and going backwards involves evaluating the permutation. Instantiated with an RSA group, this involves computing e^{th} roots modulo N. This is easy given the trapdoor $\varphi(N)$ but otherwise believed hard for suitably chosen N under the (weak) RSA assumption [18]. This construction is similar to each user running a private STROBE randomness beacon [5].

Naively, verifying that a revealed key K^i chains back to the original key K^0 in K_{pub} requires $O(i)$ work via re-execution. This can be avoided via a succinct proof of exponentiation. However, note that since the prover knows the trapdoor $\varphi(N)$, Pietrzak proofs [29] (which

[5] The size of this exponent required for security depends on assumptions. Under the Short Exponent Indistinguishability assumption (SEI) [15], for α sampled from the range $[0, 2^{2\lambda}]$ the value of g^α will be indistinguishable from random in $\mathbb{G}$. Without this assumption, α must be sampled from the the range $[0, |\mathbb{G}| \cdot 2^{2\lambda}]$.

require $O(\log i)$ work to verify) must be used instead. Wesolowski proofs [33] are not *strongly unique* [31] if the prover knows the trapdoor and hence would make the iVRF unsound. Alternatively, a more practical approach is for verifiers to cache the most recent revealed key K_i after each epoch (instead of the originally published value K_0), leaving only constant work to verify at each index.

6 The Committee-based Approach

In a fundamentally different approach to accountable SSLE, we leverage threshold (public-key) encryption. Each participant's nonce is threshold-encrypted to the public key of a committee after a threshold cryptographic setup, and then published alongside each participant's commitment in the registration list L so that a threshold number of participants can collaborate to reconstruct the nonce of a leader that withholds later. We assume, without loss of generality, that this committee is the group of all SSLE participants, although it could even be a separate group of outsiders.

An accountable SSLE scheme using threshold encryption. The following changes are made to the BEHG "high-communication" protocol to yield an accountable SSLE scheme (denoted by ThrPKE-SSLE) that uses threshold encryption. First, SSLE.Setup also runs ThrPKE.Setup. Next, a user $\mathcal{U}_i$ registering via SSLE.Register must additionally append $\mathsf{ct}_i = \mathsf{ThrPKE.Enc}(\mathsf{pp}, k_i)$ to the registration list L. (ct_i is not included in the shuffled list).[6] This still means that the winning element of the final shuffled list ℓ is a commitment $\tilde{c}$. Given this, we modify SSLE.Verify and SSLE.Recover as follows:

SSLE.Verify$(i, \mathsf{pp}, \mathsf{st}, R, \pi)$ outputs 1 if the revealed key $\tilde{k}$ is consistent with k_{iR}, the winning commitment $\tilde{c}$, and also ct_i (with the randomness used to make the encryption also supplied by the revealer as part of π), and 0 otherwise.

SSLE.Recover$(\mathsf{pp}, \mathsf{st}, R, L, \mathcal{U}_i)$ outputs $1/0/\perp$ as follows. SSLE.Recover first parses st to obtain the current shuffled list of entries and uses R to choose the winning commitment $\tilde{c} = (\tilde{u}, \tilde{v})$. It then finds $\mathcal{U}_i$'s entry $(k_{iR}, c_i, \mathsf{ct}_i, \mathsf{id}_i)$ in L and interactively (involving at least τ out of n users) runs the algorithm ThrPKE.Dec to let $\tilde{k} = \mathsf{ThrPKE.Dec}(\mathsf{pp}, \{\mathsf{sk}_i\}_{i \in S}, \mathsf{ct}_i)$ and $\tilde{k}_L || \tilde{k}_R \leftarrow H(\tilde{k})$. It outputs 1 if both $k_{iR} = \tilde{k}_R$ and $\tilde{u}^{\tilde{k}_L} = \tilde{v}$. It outputs 0 otherwise.

After Recover is run, the registration list must be cleared and all participants must re-register. To improve efficiency, rather than running Recover after every withheld election, one could run Recover once per time period of some length, or only after m leaders have withheld. This would still return all of these leaders' identities but mitigate frequent re-registration, at the cost of learning these identities later.

▶ **Theorem 9.** *Assuming honest shuffling,* ThrPKE-SSLE *is a single secret leader election protocol that satisfies strong (and weak) accountability, given an adversary that controls less than τ participants.*

The proof is included in the full version.

[6] ct_i may be included in the shuffled list if the encryption scheme yields an unlinkable commitment as defined in [10], where no adversary can distinguish between two commitments (that include these ciphertexts), even if they have been adversarially re-randomized. Including ct_i in the shuffled list would allow the committee to decrypt only the winning ciphertext in Recover rather than the whole registration list.

7 Conclusion

We propose and define the notion of accountability for secret leader election. Our schemes take three distinct approaches to address the threat of withholding attacks in secret leader election protocols. These schemes offer a variety of trade-offs between computational overhead for participants, communication requirements, the time delay before absentee leaders will be detected, and strong-versus-weak accountability. Exploring these trade-offs in practice for concrete protocols is an important avenue for future work.

A fundamental question to ask in practice is *how promptly is accountability required?* Committee-based approaches have the advantage of enabling accountability nearly immediately after a leader fails to act during their turn in a protocol, as the committee can compute the election results whenever desired. Delay-based approaches inevitably introduce a longer waiting period, as delay functions must be parameterized conservatively to ensure that malicious attackers cannot compute them quickly enough to learn the election results early. Furthermore, honest parties might not start computing the delay function until after a leader fails to act, to avoid the cost of always computing it even when the leader is honest. Finally, key-disclosure approaches may offer an even longer waiting period for accountability: until the next time a missing leader is elected (or unstakes) or until the end of an epoch.

We also leave open the fundamental question of incentives and mechanism design. Clearly, the key disclosure approach hinges entirely on appropriately incentivizing participants to reveal keys. Delay-based approaches require incentivizing some party to compute the delay functions, which may become non-trivial if they must be computed for every participant. Even the committee-based approaches require incentivizing a committee to act when a leader fails to show up, and not to conspire to learn election results early.

Finally, all of our protocols can, at best, serve as a detection mechanism for withholding attacks (but not prevent them absolutely). Thus, it is vital to design appropriate penalties (slashing) to ensure that such attacks are not profitable. At the same time, slashing may introduce new incentive issues if attackers are incentivized to try denial-of-service attacks on leaders as they attempt to broadcast during their slot. From the point of view of an accountability mechanism there is no difference between a leader who withholds and a leader whose network connection is jammed while they are legitimately attempting to broadcast a block.

Precisely because of these open questions, we present a variety of options rather than a single approach. We hope that future work can utilize these as a toolbox to improve the security of secret leader election protocols in practice.

References

1 Arasu Arun, Joseph Bonneau, and Jeremy Clark. Short-lived zero-knowledge proofs and signatures. In *Asiacrypt*, 2022.

2 Sarah Azouvi, Patrick McCorry, and Sarah Meiklejohn. Betting on Blockchain Consensus with Fantomette. *arXiv preprint*, 2018. `arXiv:1805.06786`.

3 Sarah Azouvi, Patrick McCorry, and Sarah Meiklejohn. Winning the caucus race: Continuous leader election via public randomness. *arXiv preprint*, 2018. `arXiv:1801.07965`.

4 Michael Backes, Pascal Berrang, Lucjan Hanzlik, and Ivan Pryvalov. A framework for constructing Single Secret Leader Election from MPC. In *ESORICS*, 2022.

5 Donald Beaver, Konstantinos Chalkias, Mahimna Kelkar, Lefteris Kokoris Kogias, Kevin Lewi, Ladi de Naurois, Valeria Nicolaenko, Arnab Roy, and Alberto Sonnino. STROBE: Stake-based Threshold Random Beacons. In *AFT*, 2023.

6 Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable Delay Functions. In *CRYPTO*, 2018.

7 Dan Boneh, Benedikt Bünz, and Ben Fisch. A Survey of Two Verifiable Delay Functions. Cryptology ePrint Archive, Paper 2018/712, 2018.
8 Dan Boneh, Saba Eskandarian, Lucjan Hanzlik, and Nicola Greco. Single secret leader election. In *AFT*, 2020.
9 Dan Boneh and Moni Naor. Timed commitments. In *CRYPTO*, 2000.
10 Dan Boneh, Aditi Partap, and Lior Rotem. Post-Quantum Single Secret Leader Election (SSLE) From Publicly Re-randomizable Commitments. In *AFT*, 2023.
11 Zvika Brakerski, Shafi Goldwasser, Guy N Rothblum, and Vinod Vaikuntanathan. Weak Verifiable Random Functions. In *TCC*, 2009.
12 Dario Catalano, Dario Fiore, and Emanuele Giunta. Adaptively secure single secret leader election from DDH. In *PODC*, 2022.
13 Dario Catalano, Dario Fiore, and Emanuele Giunta. Efficient and universally composable single secret leader election from pairings. In *PKC*, 2023.
14 Kevin Choi, Aathira Manoj, and Joseph Bonneau. SoK: Distributed Randomness Beacons. In *IEEE Security & Privacy*, 2023.
15 Geoffroy Couteau, Michael Klooß, Huang Lin, and Michael Reichle. Efficient range proofs with transparent setup from bounded integer commitments. In *Eurocrypt*, 2021.
16 Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *PKC*, 2005.
17 Muhammed F. Esgin, Oguzhan Ersoy, Veronika Kuchta, Julian Loss, Amin Sakzad, Ron Steinfeld, Xiangwen Yang, and Raymond K. Zhao. A new look at blockchain leader election: Simple, efficient, sustainable and post-quantum. Cryptology ePrint Archive, Paper 2022/993, 2022.
18 Dankrad Feist. RSA Assumptions. `rsa.cash/rsa-assumptions/`, 2022.
19 Matheus VX Ferreira, Ye Lin Sally Hahn, S Matthew Weinberg, and Catherine Yu. Optimal Strategic Mining Against Cryptographic Self-Selection in Proof-of-Stake. In *Economics and Computation*, 2022.
20 Chaya Ganesh, Claudio Orlandi, and Daniel Tschudi. Proof-of-stake protocols for privacy-aware blockchains. In *Eurocrypt*, 2019.
21 Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. YOSO: You Only Speak Once: Secure MPC with Stateless Ephemeral Roles. In *CRYPTO*, 2021.
22 dapplion George Kadianakis, Justin Drake. EIP-7441: Upgrade block proposer election to Whisk. URL: `https://eips.ethereum.org/EIPS/eip-7441`.
23 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP*, 2017.
24 Neil Haller. The S/KEY one-time password system. In *NDSS*, 1994.
25 Alireza Kavousi, Zhipeng Wang, and Philipp Jovanovic. SoK: Public Randomness. Cryptology ePrint Archive, Paper 2023/1121, 2023.
26 Thomas Kerber, Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas. Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake. In *IEEE Security & Privacy*, 2019.
27 Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *FOCS*, 1999.
28 Adrian Perrig, , Ran Canetti, JD Tygar, and Dawn Song. Tesla broadcast authentication. *RSA CryptoBytes*, 5, 2002.
29 Krzysztof Pietrzak. Simple verifiable delay functions. In *ITCS*, 2018.
30 Protocol Labs. Secret single-leader election (SSLE). URL: `https://github.com/protocol/research-RFPs/blob/master/RFPs/rfp-6-SSLE.md`.
31 Philipp Schindler, Aljosha Judmayer, Markus Hittmeir, Nicholas Stifter, and Edgar Weippl. RandRunner: Distributed Randomness from Trapdoor VDFs with Strong Uniqueness. In *NDSS*, 2020.
32 Toni Wahrstätter. Selfish Mixing and RANDAO Manipulation. `ethresear.ch/t/selfish-mixing-and-randao-manipulation/16081`, 2023.
33 Benjamin Wesolowski. Efficient verifiable delay functions. In *Eurocrypt*, 2019.