A framework for constructing Single Secret Leader Election from MPC^{*}

Michael Backes¹, Pascal Berrang², Lucjan Hanzlik¹, and Ivan Pryvalov^{1,3}

¹ CISPA Helmholz Center for Information Security {backes,hanzlik}@cispa.de ² University of Birmingham P.P.Berrang@bham.ac.uk ³ University of Luxembourg ivan.pryvalov@uni.lu

Abstract. The emergence of distributed digital currencies has raised the need for a reliable consensus mechanism. In proof-of-stake cryptocurrencies, the participants periodically choose a closed set of validators, who can vote and append transactions to the blockchain. Each validator can become a leader with the probability proportional to its stake. Keeping the leader private yet unique until it publishes a new block can significantly reduce the attack vector of an adversary and improve the throughput of the network. The problem of Single Secret Leader Election (SSLE) was first formally defined by Boneh et al. in 2020.

In this work, we propose a novel framework for constructing SSLE protocols, which relies on secure multi-party computation (MPC) and satisfies the desired security properties. Our framework does not use any shuffle or sort operations and has a computational cost for N parties as low as O(N) of basic MPC operations per party. We improve the state-of-theart for SSLE protocols that do not assume a trusted setup. Moreover, our SSLE scheme efficiently handles weighted elections. That is, for a total weight S of N parties, the associated costs are only increased by a factor of $\log S$. When the MPC layer is instantiated with techniques based on Shamir's secret-sharing, our SSLE has a communication cost of $O(N^2)$ which is spread over $O(\log N)$ rounds, can tolerate up to t < N/2of faulty nodes without restarting the protocol, and its security relies on DDH in the random oracle model. When the MPC layer is instantiated with more efficient techniques based on garbled circuits, our SSLE requires all parties to participate, up to N-1 of which can be malicious, and its security is based on the random oracle model.

1 Introduction

In 2008, Bitcoin [27] laid the foundation for the increasingly important areas of cryptocurrencies and distributed ledgers. One of the main advantages of distributed ledgers is that there is no single central authority that controls the

 $^{^{\}star}$ This is the full version of the paper, an extended abstract is to appear at ESORICS 2022.

transaction flow (*censorship resistance*). Anyone can access the public ledger, which is a sequence of blocks that contains transactions. For example, in Bitcoin, participants called "miners" are randomly selected to produce and append a new block to the chain. This selection process relies on the "proof-of-work" concept (PoW). To append a block to the chain, the participant has to find a value, such that a cryptographic hash function is evaluated below some threshold.

To avoid extreme energy consumption induced by PoW protocols [29], an alternative approach, "proof-of-stake" (PoS), has been proposed. Here, the probability of being selected for appending the chain depends on the stake (i.e., coins) a party owns. It does not matter whether the party owns an account with some stake v, or several accounts whose accumulated stake amounts to v. The protocol consensus works as long as the majority of all stake is controlled by honest users.

In cryptocurrencies based on proof-of-stake [23, 26, 18, 19], a single party that produces a block is chosen randomly from a set of participants, called validators (which is the equivalent to miners in a PoW protocol). In a PoS cryptocurrency there could be potentially thousands or millions users, who may come and go. It is up to a PoS protocol to determine and fix a relatively small (typically tens or hundreds) set of validators [23] from which a validator is selected that can append a block within a given time frame. To create a consistent picture for all validators, this selection has to be deterministic, but pseudo-random – properties often achieved by relying on Verifiable Random Functions (VRF). However, if an adversary knows in advance which of the validators is selected, it can launch a targeted attack and cause a denial-of-service.

Previous approaches to solving this issue aim to run the selection process in private, with the selected participant publishing a proof alongside the block. Until recently, these approaches failed to guarantee only a single participant to be chosen [23]. After much interest in a solution that provides such a guarantee [35], Boneh et al. proposed a formal definition and several instantiations of a Single Secret Leader Election [4].

The primary motivation of having a single leader is a simple consensus design, as there are no forks in the blockchain (assuming some reasonable connectivity between parties). This property encourages the leader to solely perform heavy computations, which may even exceed the running time of SSLE and/or require multiple cores. For example, the leader's task may consist of prover-heavy computations, whereas verification is very fast (SNARKs). Many protocols (e.g., [26, 18]) assume uniqueness, and it is easy to update them with a SSLE solution. They may require a full redesign if the uniqueness assumption no longer holds.

1.1 Our contribution

1. In this work, we propose a framework for constructing an efficient Single Secret Leader Election (SSLE), which relies on secure multi-party computation (MPC). We formulate a simulation-based definition of the SSLE problem.

2. We present two instantiations of our framework, which improve the stateof-the-art for SSLE protocols that do not require a trusted setup. The first instantiation is a *t*-threshold SSLE scheme that is based on Shamir's secret sharing in the random oracle model. We prove that our construction is secure in the honest-but-curious and malicious adversary models. For the latter, we additionally assume DDH. For N parties, the leader election requires $O(\log N)$ communication rounds and O(N) of basic operations on the underlying primitives. Furthermore, we instantiate our SSLE scheme using the MPC framework by Wang et al. [40], which is secure against any number of malicious parties and is more scalable, but requires all parties to be online.

3. Our SSLE framework can efficiently handle arbitrary stake distributions. For N parties and the overall sum of their stake units S, our construction achieves $O(N \log S)$ cost of the election. Compared with a standard multi-registration technique, in which a party registers multiple times for the election proportionally to her stake, this cost may go up to O(S), which makes our solution exceptionally efficient if $N \ll S$.

4. We implemented and microbenchmarked our solution using two different MPC frameworks. The performance evaluation indicates that our DDH-based SSLE protocol can be used in practical scenarios up to 30-40 parties when instantiated with the textbook $O(N^2)$ techniques using the verifiable secret sharing scheme (VSS). Furthermore, we implemented our SSLE in the MPC framework based on garbled circuits [40]. The overall time to set up and complete the protocol for 128 parties in a practical scenario is less than 7 minutes.

Note that, due to space limitations, we refer to the full-version [3] for most proofs. Only the security analysis can be found in the appendix.

1.2 Background

The idea of proof-of-stake was first discussed on the Bitcoin forum⁴ in 2011. Kiayaias et al. presented a provably-secure PoS protocol "Ouroboros" at CRYPTO 2017 [26], in which the participants that produce the blocks are elected publically. Such a leader election may be public as in Ouroboros or private as in Algorand [23]. In a private leader election, each node needs to check whether it will be the next leader using its private information but then can prove to others using only public information that it is indeed the next leader. Such a design makes it impossible for others to predict and carry out DoS attacks against the next leader until it is too late.

Algorand achieves this private leader election using Verifiable Random Functions, for which a participant has to prove the outcome to be below a certain threshold. This, however, can result in either no participant or multiple participants being elected. Another protocol employing a private leader election has been formalized by Ganesh et al., whose protocol Ouroboros Praos [19] does not guarantee existence and uniqueness of the leader either.

To mitigate these shortcomings of previous private leader elections, a problem statement of a single secret leader election was first posed at a GitHub page [35] in the form of a research proposal in the context of the Filecoin cryptocurrency. The protocol's goal is to elect a *single* leader among a finite set of participants.

⁴ https://bitcointalk.org/index.php?topic=27787.0 (accessed 31.01.2022)

Informally, such a protocol consisting of n participants has to meet the following requirements: fairness – the probability of a particular party being elected should be proportional to her power or stake, secrecy – only the elected leader should learn the result of the protocol, (public) verifiability – the elected leader should be able to prove the leadership to the other participants or observers by showing a proof-string, unpredictability – no set of participants smaller than a threshold m of n should be able to predict the outcome of the protocol with a probability greater than a negligible factor. To tolerate sporadic drop outs, the protocol should satisfy liveness – it should terminate as long as the honest majority is participating. Moreover, the protocol should be reasonably efficient, i.e., on-chain $O(\log n)$ bits per block, O(n) communication complexity (per active party).

In order to let anyone to verify transactions in a chain, the leader has to append new transactions along with a proof of leadership, and the verification algorithm should only use the data stored on the blockchain. Therefore, in the context of PoS systems, we distinguish on-chain and off-chain messages sent by the parties according to a SSLE scheme. A multi-round SSLE protocol is not required to post intermediate messages in a blockchain, as long as the parties agree on the final on-chain message. If in a PoS protocol the parties' stakes are public, an SSLE scheme can be naturally used out of the box.

1.3 Related work

Following this call, Boneh et al. [4] formalized the problem of Single Secret Leader Election (SSLE) and presented three constructions: 1) a feasibility result based on indistinguishability obfuscation, 2) a construction based on threshold fully homomorphic encryption (TFHE), and 3) a construction based on DDH that achieves a weaker notion of security. Subsequently, Catalano et al. [9] proposed a UC-secure SSLE based on public key encryption with keyword search (PEKS).

We begin by first comparing how arbitrary stake distributions are handled in previous and our work. While a scenario with equal stakes is easier to analyze, in practice one has to also account for arbitrary stake distributions and how they affect the overall performance of the scheme. Boneh et al. [4] suggest a multi-registration technique (one registration corresponds to one unit of stake) to address arbitrary stake distributions, which makes the associated costs grow linearly with the user's stake. In contrast, our construction offers a more efficient tree-based solution to this setting with the associated costs grow logarithmically in the total stake S of participating parties.

The TFHE-based SSLE [4] uses TFHE [5] as a building block, which in turn is based on fully homomorphic encryption (FHE) [22]. In contrast to our approach, its security relies on the learning with errors assumption (LWE) [34] and requires a trusted setup. Depending on how the underlying building blocks are instantiated, the TFHE-based SSLE scheme offers various trade-offs in terms of assumptions and space/runtime. For more details, we refer the reader to [4, 5]. After a random beacon R is revealed, the parties engage in one round of communication to determine a leader. The DDH-based construction [4] relies on more lightweight components than the TFHE-based one, but achieves only a weaker security notion of unpredictability. More specifically, for N parties, a potential leader is picked from a subset of data elements representing $O(\sqrt{N})$ parties, and an adversary is asked to predict a leader within this subset (excluding parties controlled by the adversary), whereas in the full notion of unpredictability an adversary has to guess a purported leader from the set of N parties. To register for an election, a party has to update and shuffle $O(\sqrt{N})$ group elements available and provide a NIZK proof of honest shuffling and re-randomization. These messages have to be considered on-chain, so that everyone could verify the outcome of an election. One can trade efficiency for security in this scheme by changing the number of elements that has to be reshuffled during registration. After a random beacon R is revealed, a leader can be determined locally, thus requiring no further communication. This construction can achieve a full security notion when shuffling all N group elements, at the cost of degrading its efficiency.

The PEKS-based construction [9] uses PEKS as a building block. The latter is a notion of functional encryption [6, 31], in which a holder of a secret key skassociated with a keyword w can locally check whether a ciphertext encrypts wor not. On a high level, the SSLE construction works as follows: each party ireceives a secret PEKS-key sk_i associated with the number i. During an election, a new ciphertext c is generated, which encodes a random keyword $j \in \{0, N-1\}$. A party that can decrypt c is the leader and can compute a NIZK proof of leadership. The SSLE construction requires a trusted party to generate the keys, for which the authors show a protocol to distribute this trust assuming $t < \lfloor N/2 \rfloor$ of parties are corrupted; this instantiation is based on the functional encryption for orthogonality (OFE) [25, 41] and assumes SXDH is hard in bilinear groups.

Our SSLE is based on MPC, runs $O(\log N)$ rounds of communication, and does not use a randomness beacon R. A party is required to post as little as O(1)of information on-chain during registration and for proving leadership. From [4], we use game-based definitions, as a starting point for our definitions, and the technique to prevent duplicate key attacks; we then propose a simulation-based definition in the UC framework. Our UC-formulation of the SSLE problem differs from [9]; we find that our definition better facilitates a more restrictive setting, in which each party fully controls its registration procedure of the SSLE and the leader reveals the registration material as a proof of leadership. In such a model, we are able to seamlessly instantiate SSLE with any secure MPC framework. We prove security for a sequential execution of the protocol.

We compare our constructions with Boneh et al. [4] and Catalano et al. [9] in Table 1. For completeness, we included the obfuscation-based feasibility result in [4]. This construction is the only non-interactive among the discussed ones, i.e., the outcome of the election is known right after the randomness beacon is revealed. By *pub*, we denote the number of public key operations such as exponentiation, by MPC op. we denote basic MPC operations such as multiplication. The most notable differences are that (1) our scheme does require neither a

Construction	$\begin{array}{c} Assump-\\ tions \end{array}$	$Security \\ notion$	Setup	Rounds	$Computation \ / \ Communication$	On- $chain$
Obfuscation- based [4]	iO	game-based	trusted	0 + beacon	$O(\lambda)$, feasibility result	O(1)
TFHE-based [4]	TFHE, weak PRF	game-based, t-threshold	trusted	1 + beacon	depends on a particular instance	O(N)
Shuffle-based [4]	ROM, DDH	game-based, weak unpre- dictability	-	1 + beacon	$O(\sqrt{N})$ pub. / group el.	$O(\sqrt{N})$
Shuffle-based [4]	ROM, DDH	game-based	-	1 + beacon	O(N) pub. / group el.	O(N)
PEKS-based [9]	ROM, SXDH	UC, t -treshold	trusted	$\leq 2 + beacon$	O(N) pub. / group el.	$O(\log^2 N)$
Our Construction 1	ROM, DDH	UC, t-threshold	-	$O(\log N)$	O(N) MPC op.	O(1)
Our Construction 2	ROM	UC	-	$O(\log N)$	O(N) MPC op.	O(1)

Table 1: Comparison of SSLE protocols, assuming all N users participate in election, amortized per one election. On-chain asymptotics include a security parameter λ ; PEKS-based on-chain asymptotic is shown assuming the parameter choice suggested in [9].

trusted setup nor a randomness beacon, and (2) requires only a constant amount of data to be posted on-chain.

In the discussed schemes except for iO- and PEKS-based the leader has to re-register before next election, since she reveals a secret that was generated and used for the registration.

Concurrently to our work, Catalano et al. [10] revisit the shuffle-based SSLE realization from [4] and propose two UC-secure SSLE constructions from DDH. Their first construction is secure against static adversaries and their second achieves adaptive security with erasures.

On the practicality of our SSLE framework The number of validators depends on the PoS protocol and can vary from dozens to a few hundred and in limited cases thousands. It does not necessarily correlate with the total number of users. Stake disbalances also vary, and therefore they need to be approximated in our framework by a tree of a sufficient height (Section 7.1). Our tree optimization technique has a better effect when applied to a smaller set of validators.

In our SSLE framework, we rely on existing MPC techniques. If a more efficient MPC protocol than the ones used in our constructions emerges, it will help to further improve the running time of the SSLE.

2 Definitions

2.1 Preliminaries

DDH Assumption [16] Let g be a generator of a group G of a prime order q. For any probabilistic polynomial time (PPT) machine \mathcal{A} and $(x, y, z) \leftarrow (\mathbb{Z}_q)^3$, $|Pr[\mathcal{A}(g, g^x, g^y, g^{xy}) = 1] - Pr[\mathcal{A}(g, g^x, g^y, g^z) = 1]| \leq negl(\lambda).$

Secret Sharing Secret sharing schemes allow a dealer to share a secret s among parties such that later a qualified set of parties can jointly reconstruct s, whereas a non-qualified set of parties learns no information about it. We use Shamir's Secret Sharing [37], which is a t-threshold scheme. Let $P_1, ..., P_N$ be N parties and there is a threshold t < N/2. In Shamir's secret sharing scheme, a secret can be shared among N parties such that t + 1 parties can reconstruct it, whereas t parties learn no information about the secret. We denote [x] a Shamir sharing of x in a prime field \mathbb{Z}_q , for which each party P_i gets a secret share $x_i \in \mathbb{Z}_q$. For this scheme to work, it is required that N < q. We denote Share a protocol to share a secret x as [x], and Rec to reconstruct x from [x]. Whereas Shamir's Secret Sharing is only secure against passive adversaries, Verifiable Secret Share (VSS) schemes [32] can protect against active.

Communication and adversary models We assume secure point-to-point communication channels between parties. An adversary is allowed to corrupt up to t < N parties. We consider two models of adversaries: honest-but-curious and malicious. In the honest-but-curious model, adversaries follow the protocol honestly and try to learn as much as possible from observed communication by corrupted parties. In the malicious model, the parties controlled by an adversary can stop communicating or send arbitrary messages to other parties, not necessarily following the prescribed protocols.

Secure Multi-Party Computation (MPC) MPC allows a set of parties $\mathcal{P} = \{P_1, ..., P_N\}$ to jointly compute a function on their private inputs in a privacy-preserving manner [42]. Our SSLE scheme is based on MPC.

We borrow the standard definitions of VIEW and t-Privacy from [1].

Definition 1 (VIEW). Let $\mathcal{P} = \{P_1, ..., P_N\}$ engage in a protocol Π that computes function $f(in_1, ..., in_N) = (out_1, ..., out_N)$. Let $VIEW_{\Pi}(P_i)$ denote the view of participant P_i during the execution of protocol Π . More precisely, P_i 's view is formed by its input and internal random coin tosses r_i , as well as messages $m_1, ..., m_l$ passed between the parties during protocol execution: $VIEW_{\Pi}(P_i) = (in_i, r_i; m_1, ..., m_l)$.

We denote the combined view of a set of participants $\mathcal{I} \subseteq \mathcal{P}$ (i.e., the union of the views of the participants in I) by $VIEW_{\Pi}(\mathcal{I})$.

Definition 2 (*t*-**Privacy**). We say that protocol Π is *t*-private in the presense of honest-but-curious adversaries if for all $\mathcal{I} \subset \mathcal{P}$ with $|\mathcal{I}| \leq t < N$ there exist a PPT simulator $S_{\mathcal{I}}$ such that $\{S_{\mathcal{I}}(in_{\mathcal{I}}, f(in_1, ..., in_N))\} \equiv \{VIEW_{\Pi}(\mathcal{I}), out_{\mathcal{I}}\},$ where $in_{\mathcal{I}} = \bigcup_{P_i \in \mathcal{I}} \{in_i\}, out_{\mathcal{I}} = \bigcup_{P_i \in \mathcal{I}} \{out_i\}, and \equiv denotes computational$ indistinguishability.

In the malicious setting, the subset \mathcal{I} of honest-but-curious parties in Def. 1 and 2 is replaced with an equal-sized subset of malicious PPT parties \mathcal{I}^M , and the protocol Π is replaced with Π^M , its maliciously-secure version.

We instantiate our SSLE scheme using the following underlying protocols:

- 1. the VSS-based MPC protocols [32, 33, 21, 20, 15, 11], in which secrets are shared between the parties using Shamir's secret sharing scheme:
 - protocols for adding shares, substracting, and multiplying by a scalar: $[x] + [y], [x] - [y], [\alpha \cdot x],$
 - RndFld to generate a share of a random field element in \mathbb{Z}_p ,
 - RndBit to generate a share of a random bit,
 - Mul to compute $[x \cdot y]$ given [x] and [y].
- 2. garbled circuit based MPC [40] on boolean circuits, where each party can privately input her input to a computing circuit.

2.2 Single Secret Leader Election

We consider the following problem. Given a set of N parties. The parties do some interactive pre-computation. Then, each party can run a local function that takes the transcript as input to determine whether it is the leader or not. The leader can show a proof that it is the leader.

Game-based formulation of the SSLE problem Our syntax and security properties of SSLE are based on that of [4], with a slight difference that we do not have an external source of randomness (random beacon) and we allow multiple rounds of communication between the parties during the election, whereas the definition of SSLE in [4] allows a single round of communication.

Informally, we capture the following security properties:

- 1. Uniqueness an adversary wins this experiment if in at least one election in a series of consecutive elections there is more than one verifiable leader.
- 2. Unpredictability the adversary asks for a challenge election after a series of elections. The challenger does not send to the adversary the outcome of this election. The adversary has to guess the leader in this challenge election. If some honest party is the leader, the adversarial chances to correctly guess the leader should not be significantly greater than pure guessing.
- 3. Fairness the adversary asks for a challenge election after a series of elections. The probability of winning this challenge election by one of the corrupted parties should not be significantly greater than c/n, where c is the number of corrupted parties, and n is the number of parties registered for the challenge election.

Due to page limits, we postpone the formal game-based definition to Appendix A.

Simulation-based definition of the SSLE problem We now formulate the SSLE problem as an ideal functionality $\mathcal{F}_{SSLE}^{N,\ell,c}$, which is presented in Figure 1. In the description of the ideal functionality, we denote election id as *eid*, and registration numbers as C_i . We then show that the simulation-based definition implies the game-based one.

 $\mathcal{F}_{SSLE}^{N,\ell,c}$ for a set of parties $\mathcal{P} = \{P_1, \ldots, P_N\}$, c of which are corrupted by an adversary, consists of the following steps:

- Upon receiving a message (*eid*, register, C) from P_i , check if (*eid*, P_i , ·) or (*eid*, *elected*, ·, ·) is stored. If so, ignore the message. Otherwise, store (*eid*, P_i , C). When storing tuples, we write P_i to denote the party's unique identifier. Send (*eid*, *registered*, P_i) to all parties and the environment.
- Upon receiving a message (eid, regVerify) from P_i , reply 0 if there exist two stored tuples (eid, P_j, C_j) and (eid, P_k, C_k) such that $j \neq k$ and $C_j = C_k$. Otherwise, reply 1.
- Upon receiving a message (eid, elect) from P_i , check if there are at least ℓ registered parties that have corresponding stored tuples (eid, \cdot). If not, ignore the message, otherwise proceed. Check if (eid, elected, P_u, C_u) is stored. If not, pick one of the stored tuples (eid, \cdot , \cdot) uniformly at random as (eid, P_u, C_u), append it as (eid, elected, P_u, C_u), and send (eid, elected, C_u) to the environment. Send (eid, elected, C_u) to P_i .
- Upon receiving a message $(eid, verify, P_j, C)$ from P_i , check if $(eid, elected, P_u, C_u)$ is stored. If such a tuple exists, reply 1 if $P_u = P_j$ and $C_u = C$. In all other cases, reply 0.



Our modeling of the ideal functionality $\mathcal{F}_{\mathsf{SSLE}}^{N,\ell,c}$ for N parties with an adversary statically corrupting up to t of them is influenced by the corresponding gamebased definition (Definition 3), which defines the registration and verification algorithms that surround the election itself. We follow the same approach and define messages in the ideal functionality for registration, election, and their verification.

In $\mathcal{F}_{\mathsf{SSLE}}^{N,\ell,c}$, the parties send messages to the ideal functionality that correspond to a specific stage of the election. First, the parties register for an election with id *eid* via sending register messages containing the registration number C. They receive notifications from the ideal functionality for every registered party. To verify registration, the parties send messages regVerify to the ideal functionality, which outputs 1 if all registered numbers are distinct, otherwise it outputs 0 and the execution of $\mathcal{F}_{\mathsf{SSLE}}^{N,\ell,c}$ stops. If regVerify returned 1, the parties participate in the election by sending messages elect to $\mathcal{F}_{\mathsf{SSLE}}^{N,\ell,c}$, which returns one of the registered numbers as the elected number. Finally, the parties can verify whether some party P_i is the elected leader by sending a message verify with the identifier for P_i and the elected number.

Next, we discuss some of the design choices that we made in $\mathcal{F}_{SSLE}^{N,\ell,c}$:

- 1. With the explicit inputs associated to parties, the definition naturally captures the adversarial ability to register multiple parties using the same private material and thereby break the uniqueness property.
- 2. The result of the election is returned to the parties as one of the numbers, used for the registration. In this way we model the information leakage, which suggests an efficient way of running multiple elections by the same parties. To run a subsequent election, the leader has to simply re-register, while other parties can keep their previously registered numbers.

Intuitively, the security properties from the game-based definitions are captured in the ideal functionality $\mathcal{F}_{\mathsf{SSLE}}^{N,\ell,c}$ as follows:

- 10 M. Backes et al.
- 1. Uniqueness provided by answering regVerify messages, which excludes the case that two parties register the same number, and elect messages are answered with exactly one number.
- 2. Unpredictability provided by answering elect messages with one of n registered numbers, which are known only to the respective parties. In the beginning, party P_i sends her input C_i only to the ideal functionality and never discloses C_i to other parties until the election is finished. P_i discloses her registered number only when P_i is the elected leader.
- 3. Fairness provided by answering elect messages by *uniformly at random* selecting one of *n* registered distinct numbers as the elected value.

We formally prove that the ideal functionality implies the game-based definitions by showing the non-existence of a simulator given any of the game-based attackers.

Proposition 1. The ideal functionality $\mathcal{F}_{SSLE}^{N,\ell,c}$ implies the game-based definitions for uniqueness (Definition 5), unpredictability (Definition 6), and fairness (Definition 7).

We refer to Appendix B for the proof of Proposition 1 and subsequent theorems.

In this work, we only consider SSLE schemes with *expiring registration*. In such schemes, in a single SSLE instance elections are run sequentially and the eventual leader has to re-register for subsequent elections. In the remainder of the paper we will only consider the modified ideal functionality that ensures sequentiality. To this end, the ideal functionality keeps track of the current election id eid^* . As soon as it receives a message with $eid' \neq eid^*$, it stops responding to any further messages with eid^* and updates the current election id to eid'. In contrast to the real world, in the ideal world non-leaders have to register for subsequent elections explicitly using the same registration number C.

3 (Non-secret) single leader election constructions

In this section, we start by discussing how naive solutions to the problem of SSLE fail in keeping the leader secret. We then gradually introduce the basis for our final SSLE protocol. Note that, while the constructions in this section do not yet meet our requirements and are considered non-secret, they will form the basis of the protocol presented in Section 4.

Naive attempts Designing a secure SSLE protocol is not a trivial task. Below, we briefly mention three naive leader selection protocols and discuss where they fail to meet our requirements.

Protocol 1 Run any secure MPC protocol for N parties to generate a fresh random number (selection phase) and later reveal it and take it modulo N to determine which party is selected (reveal phase).

Protocol 2 Each party commits to a number and sends the commitment to all parties (selection phase). To determine the leader, each party opens the commitment, and the leader is computed as a sum of these N numbers modulo N (reveal phase).

Protocol 3 Let an array of distinct numbers represent the participants in the election. To determine a leader, we randomly permute this array (or sort it according to some unpredictable criteria) (selection phase) and pick the first number as the leader and discard the rest (reveal phase).

Problem The three naive protocols defined above are *not* secret leader election protocols, as they follow a two-phase pattern: the *selection* phase and the *reveal* phase. After the selection phase is over, the parties have already committed to some leader, which is not yet known to anyone. After the reveal phase, everyone knows who the leader is. The missing intermediate point (the "check" phase) is the one that would allow the leader to learn the outcome exclusively.

Nevertheless, these protocols can serve as the basis for a secret leader election protocol. Our construction is inspired by the idea in Protocol 3. While there exist cryptographic protocols for multi-party sorting ([24, 30]) that all rely on a pairwise comparison subroutine, we take a more efficient approach: We observe that the order of the discarded numbers does not matter and take this into account when designing our solution. This observation allows us to eliminate the requirement for this comparison subroutine.

Oblivious Select We begin by defining a *two-party Oblivious Select* (OSelect) protocol, whose goal is to secretly select one out of two commitments. Once the commitment is selected, the parties can open the selected commitment. This subprotocol essentially makes use of the observation from the previous section that we can discard any information except the chosen leader. Let PSwap be an algorithm that on input commitments C_0 and C_1 computes $(C'_i = \text{Com}(C_i, r_i))_{i \in \{0,1\}}$ and outputs (C'_b, C'_{1-b}) for a random bit b. Let PSelect be an algorithm that on input commitments C_0 and C_1 outputs $C' = \text{Com}(C_b, r)$. It is easy to see that if the commitment scheme is hiding, then an adversary cannot find the value of b significantly better than pure guessing.

We now describe **OSelect** between Alice and Bob. The protocol consists of the select and the opening phases. In the select phase, Alice publishes her commitment C_A and Bob C_B , then Alice performs **PSwap** on (C_A, C_B) and sends the result (C_0, C_1) to Bob; Bob now performs **PSelect** on those values and outputs C'. In the opening phase, the two parties reveal their randomness so that the complete transcript of computing C' could be reconstructed by anyone.

The protocol can be naturally extended to N parties, where N is a power of two; let us call the resulting protocol $\mathsf{OSelect}_N$. It consists of $(\log N)$ rounds; in the first round N/2 pairs of parties are formed that run $\mathsf{OSelect}$, thereby reducing two commitments into one. In the following round, N/4 pairs of parties are formed, etc., until there is a single commitment left. We will use the logical tree-like structure used in $\mathsf{OSelect}_N$ as the basis for our final SSLE construction.

Leader Election based on Oblivious Select We define LeaderElection, our intermediate non-secret protocol, which essentially uses $OSelect_N$ in a black-box manner. In the selection phase, each user U_i initially holds a distinct number m_i and commits to it as $C_i = Com(m_i; r_i)$. Then, the users run $OSelect_N$. Thanks to the properties of $OSelect_N$, its output \overline{C} is a commitment to one of the user's inputs. If \overline{C} is a commitment to m_i , then U_i is the elected user. Since there are in total N - 1 calls to OSelect, we achieve an amortized cost O(1) per party. In the opening phase, all users broadcast their input message and randomness, so that the execution of $OSelect_N$ could be verified by anyone.

Problem The resulting protocol is still a *non-secret* leader election, as the leader does not learn the output of the protocol exclusively. Moreover, the *unpredictability* property does not hold: an adversary controlling two parties in a single instance of **OSelect** can exclude certain parties as potential leaders. Lastly, all parties are required to participate in the protocol in at least one instance of **OSelect**, which makes it impossible to tolerate a single faulty party. In the next section, we will address these problems and present our secure SSLE protocol.

Upgrading to secret leader We now modify LeaderElection by adding an intermediate representation layer in order to let the secret leader actually check whether she is the elected leader. Here, we make use of a distributed key generation and threshold decryption. The resulting secret leader election protocol does not satisfy all our requirements to SSLE but serves as an intermediate point towards our final construction in Section 4.

Distributed Key Generation (DKG) [32] allows several parties to agree on a joint secret key. The corresponding public key is computed and published jointly by the honest majority of the parties. In a t-out-of-N DKG protocol [20], the secret key is shared according to Shamir's secret sharing scheme. The protocol can be efficiently simulated against passive and active adversaries, which can corrupt up to t parties. In threshold cryptography, parties jointly generate a group public key to encrypt messages and a qualified subset of parties can collaboratively decrypt ciphertexts encrypted using that key. We consider Shamir's t-out-of-N threshold ElGamal-based decryption schemes, for which any coalition of t parties cannot decrypt a given ciphertext or learn any information about the plaintext, whereas any coalition of t+1 parties can recover it, even if the remaining N-t-1 parties stop communicating.

Let g be a generator of a group G of a prime order \mathbb{Z}_p . User U_i registers for the election by generating a registration key $k_i \in \mathbb{Z}_p$ and computing a registration token as $e_i \leftarrow (g^r, g^{k_i \cdot r})$ for some random r. The values k_i are e_i are kept private. Next, the users generate a temporary shared public key using as t-out-of-N DKG protocol, $pk_{\mathcal{G}} = g^{sk_{\mathcal{G}}}$. The corresponding group secret key, $sk_{\mathcal{G}}$, is shared between N parties, such that t + 1 parties have to collaborate to decrypt a ciphertext C.

Instead of OSelect, we use a new subroutine OSelectD, which is a two-party verifiable oblivious select protocol in the discrete log setting. Unlike OSelect, the users can publicly verify that a OSelectD instance was executed correctly without learning which input was selected. The input to OSelectD is an Elgamal

13

encryption of two group elements $e_i := (g^r, g^{k_i \cdot r})$ under a group public key $y_{\mathcal{G}}$ for some user's registration key k_i and randomness r; these two encryptions can be represented as a tuple $(g^{r'}, (y_{\mathcal{G}})^{r'} \cdot g^r, (y_{\mathcal{G}})^{r'} \cdot g^{k_i \cdot r}) \in (G_q)^3$, for some r', and we will call such tuples valid. OSelectD relies on the discrete log variants of PSwap and PSelect, which we call PSwapD and PSelectD. Let PSwapD be an algorithm that on input two tuples C_0 and C_1 computes $C'_0 = (C_0)^{r_0}$, $C'_1 = (C_1)^{r_1}$ and outputs (C'_b, C'_{1-b}) for a random bit b, accompanied with appropriate NIZK proofs that computation is done correctly. Let PSelectD be an algorithm that on input two tuples C_0 and C_1 outputs $C' = (C_b)^r$ and appropriate NIZK proofs. These proofs are generalizations [14, 7] of Schnorr signature [36] and can be efficiently instantiated in the random oracle model using the Fiat-Shamir transform [17]. It is straightforward to see that if the inputs to OSelectD are valid tuples w.r.t. k_i and k_j , then so is the output of OSelectD w.r.t. $k \in \{k_i, k_j\}$.

Expanding OSelectD to N users, we get OSelectD_N. Users jointly run OSelectD_N and decrypt its output to obtain \overline{C} . There will be a unique pair (e_i, \overline{e}) , which forms a valid DDH tuple, for which the elected leader knows an exponent; all other pairs (e_j, \overline{e}) , where $j \neq i$, are random tuples. The leader presents the exponent as proof of leadership. She will have to re-register to get a fresh k_i before participating in another election.

Problem While the leader can learn the outcome of the election in private, there remain several problems to address. First, an adversary can run a *duplicate key* attack [4], where she obtains multiple registration tokens that correspond to a single registration key, and thus break fairness. To see why the protocol suffers from this attack, we observe that for a fixed registration key k_i , two valid tokens $(g^{r_1}, g^{k_i \cdot r_1})$ and $(g^{r_2}, g^{k_i \cdot r_2})$ are indistinguishable from two valid tokens that correspond to different keys, because of the DDH assumption. The mitigation measures proposed in [4] work in our setting, too. Second, a malicious adversary can use biased coins when computing OSelectD. If both parties are under her control, she can break the obliviousness of OSelectD and, in turn, the unpredictability and fairness of the SSLE. Finally, even an *honest-but-curious* adversary, who controls both parties in OSelectD and follows the protocol, exactly knows which input has been selected, thus *breaking* unpredictability of SSLE. Since our goal is to satisfy *all* the three properties (uniqueness, unpredictability, and fairness), we will need one more modification to our current construction, which we present in the following section.

4 Our SSLE from DDH

In this section, we define our full SSLE construction; to this end we modify the secret leader election from Section 3 by replacing OSelect with its MPC variant, OSelectM. Thereby we ensure that no adversary in our model can learn the outcome of a OSelectM protocol instance. The extension of OSelectM to N inputs, which we call OSelectM_N, retains the (binary) tree layout of inputs and outputs. Each OSelectM instance is now executed by all parties simultaneously. This modification incurs additional communication costs compared to

 $\begin{array}{l} \hline \textbf{OSelectM}([C_A], [C_B]) \\ \hline [b] \leftarrow \mathsf{RandBit}() \\ \textbf{do} \quad // \text{ run in parallel} \\ [b \cdot C_A] \leftarrow \mathsf{Mul}([b], [C_A]) \\ [(1-b) \cdot C_B] \leftarrow \mathsf{Mul}([1-b], [C_B]) \\ [C'] \leftarrow [b \cdot C_A] + [(1-b) \cdot C_B] \\ \textbf{output} \quad [C'] \end{array}$

```
Fig. 2: OSelectM: Oblivious Select in the MPC setting.
```



Fig. 3: OSelect M_N : Extension OSelect M to N inputs. Example for N = 4.

the previous (insecure) version of our SSLE construction. Fortunately, the number of communication rounds needed for a leader election remains $O(\log N)$, as OSelectM instances on the same level in the tree can run in parallel.

OSelectM is an oblivious select protocol in the MPC setting, which can be completed as long as at least t+1 parties remain online and honestly execute the protocol. It takes two secret shares $[C_A], [C_B]$ as input and outputs a new secret share [C'] such that the output secret C' is either C_A or C_B with equal probability, depending on the selection bit b. The description of OSelectM protocol is shown in Fig. 2. OSelectM extension to N inputs, called OSelectM_N, follows a binary-tree structure of inputs and outputs to OSelectM; see Fig. 3.

To prevent duplicate key attacks, we incorporate into our SSLE scheme a technique used in [4]. The technique works as follow. The registration key k_i is now used to produce a secret part k_{iL} and a public fingerprint k_{iR} using a cryptographic hash function H, where $(k_{iL}, k_{iR}) \leftarrow H(k_i)$. Before the election starts, each user verifies that there are no duplicate fingerprints in the public state *st*. The security properties of the hash function ensure that chances for an adversary to succeed in a duplicate key attack are negligible.

The election proceeds as follows. For each $i \in \{1, \ldots, N\}$, the parties jointly generate $[C_i]$, a MPC version of the secret part k_{iL} of the registration key k_i , which is $[k_{iL}]$. In the MPC setting we do not need to additionally hide the key using Elgamal encryption, since secret sharing already hides the results of the computation.

The parties then proceed with $OSelectM_N$ and obtain $[\bar{C}]$, which is a secret share of one of the secret inputs to $OSelectM_N$. The parties jointly reconstruct two group elements (\bar{e}_1, \bar{e}_2) from $[\bar{C}]$, which turn out to be a randomization of the secret part of a party participated in the election, which we denote \bar{k}_L . If P_i is the elected leader, the following equation will hold $\bar{k}_L = k_{iL}$, i.e. each party learns the secret key k_{iL} of the leader, but does not know which one. The leader P_i sends the registration key k_i as a proof of leadership. To verify a proof π , one recomputes the secret part π_L of the registration key and its fingerprint π_R and checks that the computed fingerprint matches the one stored as st_i , and that the equation $\pi_L = \bar{k}_L$ holds.

$Setup(1^\lambda,t,N)$	$RegisterVerify(i,k_i)$	$Elect(i,k_i)$		
$\begin{array}{lll} 1: & p, g \leftarrow FindParam(1^{\lambda}) \\ 2: & \mathbf{for} \ i \in 1N \\ 3: & st_i \leftarrow \bot \\ 4: & \mathbf{return} \ p, g, st_1,, st_N \end{array}$	$\begin{array}{llllllllllllllllllllllllllllllllllll$	$1: \text{ for } i \in 1N$ $2: [C_i] := [k_{iL}]$ $3: [\bar{C}] \leftarrow OSelectM_{N}([C_1],$ $4: \dots, [C_N])$ $5: \bar{k}_L \leftarrow Rec([\bar{C}])$		
$\frac{\text{Register}(i)}{1: k_i \leftarrow \mathbb{Z}_p}$ $2: k_{iL}, k_{iR} \leftarrow H(k_i)$ $3: \text{st} \leftarrow k_{iD}$	$6: \mathbf{if} \ k_{iR} \neq st_i$ $7: \mathbf{return} \ 0$ $8: \mathbf{return} \ 1$	$6: \mathbf{if} \ \bar{k}_L \neq k_{iL}$ $7: \qquad \mathbf{return} \perp$ $8: \mathbf{return} \ \pi := k_i$		
$ \begin{array}{ll} 3 & \mathbf{st}_i \leftarrow \mathbf{k}_{iR} \\ 4 & [k_{iL}] \leftarrow Share(k_{iL}) \\ 5 & \mathbf{return} \ k_i \end{array} $	$\begin{aligned} & \frac{Verify(i,\pi)}{1: & \pi_L, \pi_R \leftarrow H(\pi)} \\ & 2: & if \ \pi_L = \bar{k}_L \ and \ \pi_R = st_i \\ & 3: & return \ 1 \\ & 4: & return \ 0 \end{aligned}$			

Fig. 4: Single Secret Leader Election construction SSLE instantiated with $\mathsf{OSelect}\mathsf{M}_\mathsf{N}.$

In the malicious adversary model, we can use standard techniques [21, 13] to protect the underlying MPC primitives used in the scheme against active adversaries.

We now formally define our fully-fledged SSLE construction.

Construction 1 (Single secret leader election (SSLE)) Our (N, N, t)-SSLE scheme is a tuple of PPT algorithms SSLE = (Setup, Register, RegisterVerify, Elect, Verify) that use a group G of a prime order p. Let g be a generator of G, let H be a function that maps $\{0,1\}^{\lambda}$ to $\mathbb{Z}_p \times \{0,1\}^{r(\lambda)}$. The description of the algorithms is shown in Figure 4.

Theorem 1. Assuming the underlying MPC primitives are secure in the honestbut-curious adversary model, H is a random oracle, then Construction 1 implements functionality \mathcal{F}_{SSLE} .

5 Our SSLE based on garbled circuits

In this section, we present our SSLE protocol, instantiated in the MPC framework by Wang et al. [40], which can tolerate up to N - 1 corrupted parties.

Preliminaries Wang et al. proposed an efficient secure constant-round MPC on boolean circuits by extending a two-party protocol [39] to the multi-party setting [40]. The protocol uses an ideal functionality $\mathcal{F}_{\mathsf{Pre}}$ as a preprocessing step to set up correlated randomness between the parties. At a high level, $\mathcal{F}_{\mathsf{Pre}}$ generates authenticated shares on random bits x, y, z such that $z = x \wedge y$, using

information-theoretic MACs [28]. Those authenticated shares are then used to distributively construct a single, "authenticated" garbled circuit, which is evaluated by one of the parties. Wang et al. show the security of their protocol against a malicious adversary that compromises N-1 parties in the $\mathcal{F}_{\mathsf{Pre}}$ -hybrid model and assuming a random oracle (ROM). We refer the reader to [40] for the description of $\mathcal{F}_{\mathsf{Pre}}$ and further details.

Construction We use the MPC protocol [40] to instantiate our SSLE in a blackbox manner. The SSLE construction shown in Figure 4 needs to be updated to account for the technical details specific to the MPC part in the **Elect** algorithm.

To implement the Oblivious Select, we use a part of the input as selection bits. Each party contributes to these bits, via bitwise-xor; the selection bits are therefore secret-shared. The modified version of the Elect algorithm and a pseudocode of OSelectM instantiated in the framework [40] are shown in Fig. 5. Each party P_i provides her input of (rBits + aBits) bits, which is stored in arrays of instances of MPC's class Integer, R and A. We require that $rBits \ge$ $\log(aBits)$. Array R is used to compute selection bits R[0]. Then $\log(N)$ rounds follow, in which depending on a selection bit bit, A[i] is assigned to either A[2i]or A[2i + 1]. We use an existing function select of MPC's class Integer for this specific computation. Finally, A[0] is revealed.

Construction 2 (Single secret leader election (SSLE)) Our SSLE scheme is a tuple of PPT algorithms SSLE = (Setup, Register, RegisterVerify, Elect, Verify). Let H be a function that maps $\{0,1\}^{\lambda}$ to $\{0,1\}^{l(\lambda)} \times \{0,1\}^{r(\lambda)}$. The description of the algorithms is shown in Fig. 4, and Elect is appropriately modified, as shown in Fig. 5.

Theorem 2. Assuming the underlying MPC primitives are secure in the malicious adversary model, H is a random oracle, then Construction 2 implements functionality \mathcal{F}_{SSLE} .

6 Evaluation

6.1 Experimental setup

We evaluate our SSLE framework, we implemented Constructions 1 and 2 and ran two kind of tests: in a local setting (LAN) and in a global setting (WAN). In the LAN setting, we used machines located in the same Amazon EC2 region. In the WAN setting, we used machines located in four different regions (Europe, North America, South America, and Asia). If not specified otherwise, each machine is a t2.large instance with 2 cores Intel Xeon E5-2686v4 2.3 GHz, 8Gb of RAM, and installed Ubuntu 20.04. In some regions t2.large instances are not available; instead we used t3.large instances with 2 cores Xeon Platinum 8175 2.5 GHz, 8Gb of RAM.

In our experiments, we evaluate a complete OSelect tree in our SSLE framework, that is the number of users being a power of two, starting from 8 parties,

```
\mathsf{oselect}_{\mathsf{N}}
```

1: **for** $i \in 1..N$ R[i-1] = Integer(bits : rBits, party : i)2:A[i-1] = Integer(bits : aBits, party : i)3: for $i \in 1..N - 1$ 4: $R[0] = R[0] \oplus R[i]$ 5:rem = N6: 7: round = 0while rem > 18: bit = R[0][round]9: 10: for $i \in 0..rem/2 - 1$ A[i] = A[2i].select(bit, A[2i+1])11: round=round+112:rem = rem/213:14: A[0].reveal() $\mathsf{Elect}(i, k_i)$ 1: $R_i, A_i \leftarrow k_{iL}$ 2: $\bar{A} \leftarrow \text{oselect}_{\mathsf{N}}(party \ i : R_i, A_i)$ 3: if $\bar{A} \neq A_i$ $\mathbf{return} \perp$ 4: 5: return $\pi := k_i$

Fig. 5: Elect algorithm and Oblivious Select instantiated in the MPC framework by Wang et al. [40]

and each party holding one unit of stake. For each experiment we take average of 10 runs, except that for lengthy experiments with a running time more than 1 minute we perform a single run. Next, we present implementation details and the evaluation results individually for each construction.

N	t	Algorithm	HbC time, sec.	Mal. time, sec.				
8	3	Register	< 0.01	0.11				
	RegisterVerify		< 0.01	< 0.01	N	$I(\lambda)$	LAN time,	WAN time,
		Elect	0.1	3.56	11	$\iota(\lambda)$	sec.	sec.
		Verify	< 0.01	< 0.01	8	48	2.73	23.42
16	7	Register	0.01	0.56		64	2.76	24.03
		RegisterVerify	< 0.01	< 0.01		80	2.80	24.27
		Elect	0.34	28.1	16	48	4.28	38.95
		Verify	< 0.01	< 0.01		64	4.50	39.92
32	15	Register	0.02	3.83		80	4.86	40.61
		RegisterVerify	< 0.01	< 0.01	32	48	8.25	73.34
		Elect	1.45	356.6		64	8.35	75.93
		Verify	< 0.01	< 0.01		80	8.80	77.81
64	31	Register	0.08	n.a.	64	48	17.64	145.87
		RegisterVerify	< 0.01			64	18.62	153.34
		Elect	7.63			80	23.90	150.67
		Verify	< 0.01		128	48	64.33	300.77
128	63	Register	0.21	n.a.		64	74.54	326.09
		RegisterVerify	< 0.01			80	83.54	317.46
		Elect	54.4					
		Verify	< 0.01					

Table 2: Experimenal results for Construction 1 in the honest-but-curious and malicious adversary models (left), and for Construction 2 in the malicious adversary model (right).

6.2 Construction 1 (Section 4)

Implementation details We implemented our Construction 1 in C++ in the honest-but-curious and malicious adversary models. We implemented the underlying MPC primitives for secret sharing, adding shares, substracting, multiplying by a scalar: $[x] + [y], [x] - [y], [\alpha \cdot x]$, protocols RndFld, RndBit, Mul [32, 33, 21, 20, 15, 11]. In the malicious adversary model, these primitives are accompanied with verifiable secret sharing (VSS). We set the threshold t = N/2 - 1 in all experiments. Our implementation uses the Relic toolkit [2] for operations on elliptic curves in groups of a prime order of 256 bits, the Boost and OpenSSL libraries for secure communication.

Experimental results We performed LAN tests for up to 128 parties in the honestbut-curious adversary model, and up to 32 parties in the malicious model. Timings are shown in Table 2.

Analysis The experimental results show that up to 128 parties can complete Elect protocol in under a minute. The running time grows rapidly as the number



Fig. 6: Comparison of timings for Oblivious Select in Construction 2 in the LAN and WAN settings.

of parties increases. This is due to expensive public key operations for generating and reconstructing Shamir's secret shares. The explosion of running time is more visible in the malicious adversary model. In order to protect against such adversaries, we have to use verifiable secret sharing, which requires $O(N^2)$ public key operations in the textbook implementation. While Elect is the most heavy algorithm, the rest of the SSLE protocol is essentially for free. We conclude that Construction 1 offers a practical *t*-robust solution to the SSLE problem for a small number of parties (up to 32, according to our evaluation).

6.3 Construction 2 (Section 5)

Implementation details We implemented and evaluated Oblivious Select part of the Elect algorithm, as it is the most heavy part of the SSLE protocol (see experimental results for Construction 1 in the honest-but-curious adversary model in Section 6.2). Our implementation fully relies on the implementation of the MPC framework by Wang el at. [40], which is available as [38]. We can trade-off security for efficiency by controlling how many bits each party inputs to Oblivious Select(see Lemma 8).

Experimental results In the MPC framework, the evaluator of the garbled global circuit requires more RAM than any other party. Therefore, we set up one machine as a m5a.4xlarge instance with 16 cores and 64G of RAM, while the rest of machines remain t2.large or t3.large instances. We run experiments for each N up to 128. For the trade-off, we choose the length of user inputs to Oblivious Select, $l(\lambda)$, as 48, 64, and 80 bits. Additionally, each party provides 8 bits of selection bits, which satisfies the constraint that it should be at least as big as $\log(N)$ in all test cases. Timings the LAN and WAN settings are shown in Table 2 and in Fig. 6.

Analysis The experimental results show that the running time of Oblivious Select algorithm (and in turn, Elect) grows almost linearly as the number of parties gets increased. As we ran only 1 iteration for long test cases, we can see some unexpected fluctuations in the running time, which we think are caused by fluctuations in the network and normally should be eliminated after averaging multiple iterations.

The LAN and WAN settings have identical computational and communication cost, as they only differ in the location of machines. We suspect that higher latency between machines in the WAN settings accounts for the increased running time. In the LAN setting, 128 parties can compute a leader in under 1.5 minutes, where as in the WAN setting, this number approaches 7 minutes.

7 Practical considerations

There are several constraints in Constructions 1 and 2 that affect its practicality. First, the definition of SSLE says that the probability for a party being elected should be equal among all participants. In practice, the stakeholders may have different stakes, and the probability for a party to be elected should be proportional to her stake. A straightforward solution to this constraint would be to adapt our SSLE construction to work with stake units and let each party control several units. If implemented naively, this approach results in a linear blow-up in computation and required storage (in the number of stake units). In the following, we will show an efficient technique to extend Construction 1 to support arbitrary (non-uniform) probability distributions in the election.

Second, we assumed the number of parties to be a power of two, in order to construct a complete binary tree in Oblivious Select. However, if the number of parties is arbitrary, the tree structure will likely unbalance the tree leaves, as some inputs will not be matched on the first level with other inputs. Therefore, such inputs would proceed to the next round without competition, i.e., with the probability of 1, whereas input C_i in a binary tree will proceed with the probability of 1/2. We will show that the technique from the previous point addresses this concern, too.

Moreover, we will show how to reduce the number of communication rounds in Construction 1 by using appropriate MPC primitives. Due to page limits, we present this extension in Appendix D.

7.1 Non-uniform distributions

We observe that it is possible to unbalance almost-for-free the probability of being selected (among two parties) if the sum of the weights is a *power of two*. To illustrate this idea, assume that the weights are (1, 3), i.e., the probabilities for two parties being selected are determined by the ratio 1:3. We can construct a tree-structure with the probabilities 1/4 and 3/4, as shown in ??.

Basically, we introduce a special case for OSelectM when handling shares of the same secret for free, OSelectM(ShareC, [C]) $\rightarrow [C]$. The resulting tree can be optimized significantly by dropping the nodes with the same inputs.

Using this technique, we can handle weights of the form $(w, 2^L - w)$ with a logarithmic overhead, for some $L \ge 1$ and $1 \le w < 2^L$. However, we cannot naturally handle arbitrary weight ratios. For example, weights such as (1, 2) are problematic. Nevertheless, we can approximate the probabilities in the election according to any weights (a, b) by having a tree of sufficient depth. Arbitrary N and stakes Let N be the number of parties participating in the election with their stakes $(s_1, ..., s_N)$, and let $S = \sum_{i=1}^N s_i$ be the sum of parties' stakes. The multi-registration solution may lead to O(S) complexity of the election algorithm. We extend our technique to an arbitrary number of users.

We start with a similar idea: each party has a sequence of stake units on the first level in a OSelectM_N tree. If $N \ll S$, there will be many pairs of inputs that represent the same party. We observe that in this case, there is no need to run OSelectM on such inputs. Instead, we can pick any input and advance it to the next level in the tree. The worst case complexity (the number of OSelectM instances) of this technique is $O(N \log S)$, since each party P_i 's inputs will be matched in a tree of depth $O(\log S)$ at most two times, against P_{i-1} and P_{i+1} . With a tree of depth L we can get the absolute precision up to $2^{-L} \cdot S$.

8 Conclusion

In current proof-of-stake cryptocurrencies, the a-priori knowledge of who will append the blockchain may be easily exploited by an adversary and lead to a denial-of-service of the system. The importance of hiding the identity of the block appender was recently recognized, and first solutions to the problem have been proposed by Boneh et al. [4]. In this work, we took a step further and presented an efficient SSLE protocol, whose security relies on MPC. We implemented our solution and microbenchmarked it in a real-world scenario. Our security analysis and performance evaluation indicate that our solution is practical and can be deployed into existing proof-of-stake cryptocurrencies.

Nevertheless, there are open questions that we leave for future work. First, our construction elects a unique leader. If the leader does not show up within some timeout, the parties have to restart the protocol from scratch. It would be interesting to come up with a more efficient solution for this scenario. The secret committee election of size > 1 can be seen as a generalization of the SSLE problem and could solve the mentioned problem. Such a primitive could be used, for example, to privately electing a committee for making treasury decisions as in [43]. However, the generalization of our techniques for this setting is not straightforward. If we naively skip the last OSelect subroutine and declare both candidates as leaders, we know that one candidate is from the left subtree, another from the right, which would contradict the committee's uniform selection. We leave this interesting problem as future work.

And second, the precision of our construction w.r.t. the stake distribution directly affects the communication and computational costs. It would be interesting to design an SSLE scheme that is cost-stable with regard to stake distributions.

Acknowledgements We thank anonymous reviewers for their helpful reviews. This work has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC grant

agreement no. 610150-imPACT, was supported by the German Federal Ministry of Education and Research (BMBF) through funding for CISPA and the CISPA-Stanford Center for Cybersecurity (FKZ: 16KIS0762), and by the Luxembourg National Research Fund (FNR) under the CORE Junior project FP2 (C20/IS/14698166/FP2/Mueller).

Bibliography

- Aliasgari, M., Blanton, M., Zhang, Y., Steele, A.: Secure computation on floating point numbers. In: NDSS (2013)
- [2] Aranha, D.F., Gouvêa, C.P.L., Markmann, T., Wahby, R.S., Liao, K.: RELIC is an Efficient LIbrary for Cryptography. https://github.com/relictoolkit/relic (2014)
- [3] Backes, M., Berrang, P., Hanzlik, L., Pryvalov, I.: A framework for constructing single secret leader election from MPC (full version). eprint 2022/XYZ (2022)
- [4] Boneh, D., Eskandarian, S., Hanzlik, L., Greco, N.: Single secret leader election. In: Proceedings of the 2nd ACM Conference on Advances in Financial Technologies. p. 12–24 (2020)
- [5] Boneh, D., Gennaro, R., Goldfeder, S., Jain, A., Kim, S., Rasmussen, P.M., Sahai, A.: Threshold cryptosystems from threshold fully homomorphic encryption. In: CRYPTO 2018. pp. 565–596
- [6] Boneh, D., Sahai, A., Waters, B.: Functional encryption: Definitions and challenges. In: TCC 2011. pp. 253–273
- [7] Camenisch, J., Stadler, M.: Proof systems for general statements about discrete logarithms. Technical report/Dept. of Computer Science, ETH Zürich 260 (1997)
- [8] Canetti, R.: Security and composition of multiparty cryptographic protocols. Journal of CRYPTOLOGY 13(1), 143–202 (2000)
- [9] Catalano, D., Fiore, D., Giunta, E.: Efficient and universally composable single secret leader election from pairings. IACR Cryptol. ePrint Arch. 2021, 344 (2021)
- [10] Catalano, D., Fiore, D., Giunta, E.: Adaptively secure single secret leader election from DDH. In: ACM PODC 2022. pp. 430–439 (2022)
- [11] Catrina, O., Saxena, A.: Secure computation with fixed-point numbers. In: International Conference on Financial Cryptography and Data Security. pp. 35–50 (2010)
- [12] Cramer, R., Damgård, I., Ishai, Y.: Share conversion, pseudorandom secretsharing and applications to secure computation. In: TCC 2005. pp. 342–362
- [13] Cramer, R., Damgård, I., Maurer, U.: General secure multi-party computation from any linear secret-sharing scheme. In: EUROCRYPT 2000. pp. 316–334 (2000)
- [14] Cramer, R., Damgård, I., Schoenmakers, B.: Proofs of partial knowledge and simplified design of witness hiding protocols. In: CRYPTO '94. pp. 174–187 (1994)

- [15] Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: TCC 2006. pp. 285–304 (2006)
- [16] Diffie, W., Hellman, M.: New directions in cryptography. IEEE transactions on Information Theory 22(6), 644–654 (1976)
- [17] Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Conference on the theory and application of cryptographic techniques. pp. 186–194 (1986)
- [18] França, B., Wissfeld, M., Berrang, P., von Styp-Rekowsky, P., Trinkler, R.: Albatross: An optimistic consensus algorithm. arXiv preprint arXiv:1903.01589 (2019)
- [19] Ganesh, C., Orlandi, C., Tschudi, D.: Proof-of-stake protocols for privacyaware blockchains. In: EUROCRYPT 2019. pp. 690–719 (2019)
- [20] Gennaro, R., Jarecki, S., Krawczyk, H., Rabin, T.: Secure distributed key generation for discrete-log based cryptosystems. Journal of Cryptology 20(1), 51–83 (2007)
- [21] Gennaro, R., Rabin, M.O., Rabin, T.: Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In: PODC '98. pp. 101–111 (1998)
- [22] Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: CRYPTO 2013. pp. 75–92
- [23] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: Proceedings of the 26th Symposium on Operating Systems Principles (SOSP). pp. 51–68 (2017)
- [24] Goodrich, M.T.: Randomized shellsort: A simple oblivious sorting algorithm. In: Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms. pp. 1262–1277. Society for Industrial and Applied Mathematics (2010)
- [25] Katz, J., Sahai, A., Waters, B.: Predicate encryption supporting disjunctions, polynomial equations, and inner products. In: EUROCRYPT 2008. pp. 146–162
- [26] Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: CRYPTO 2017. pp. 357–388 (2017)
- [27] Nakamoto, S.: Bitcoin: A peer-to-peer electroniccash system. https://bitcoin.org/bitcoin.pdf (2008)
- [28] Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: CRYPTO 2012. pp. 681– 700
- [29] O'Dwyer, K.J., Malone, D.: Bitcoin mining and its energy footprint. In: InISSC 2014/CIICT 2014. pp. 280–285. IET (2014)
- [30] Ohrimenko, O., Goodrich, M.T., Tamassia, R., Upfal, E.: The Melbourne shuffle: Improving oblivious storage in the cloud. In: International Colloquium on Automata, Languages, and Programming. pp. 556–567 (2014)

- 24 M. Backes et al.
- [31] O'Neill, A.: Definitional issues in functional encryption. IACR Cryptol. ePrint Arch. 2010, 556 (2010)
- [32] Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: CRYPTO '91. pp. 129–140 (1991)
- [33] Rabin, T., Ben-Or, M.: Verifiable secret sharing and multiparty protocols with honest majority. In: Proceedings of the twenty-first annual ACM symposium on Theory of computing (STOC). pp. 73–85 (1989)
- [34] Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing (STOC). pp. 84–93 (2005)
- [35] Single-Leader Election (SSLE). https://github.com/protocol/researchgrants/blob/master/RFPs/rfp-006-SSLE.md (2019)
- [36] Schnorr, C.P.: Efficient identification and signatures for smart cards. In: CRYPTO '89. pp. 239–252 (1989)
- [37] Shamir, A.: How to share a secret. Communications of the ACM 22(11), 612–613 (1979)
- [38] Wang, X., Malozemoff, A.J., Katz, J.: EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit (2016)
- [39] Wang, X., Ranellucci, S., Katz, J.: Authenticated garbling and efficient maliciously secure two-party computation. In: CCS '17. pp. 21–37
- [40] Wang, X., Ranellucci, S., Katz, J.: Global-scale secure multiparty computation. In: CCS '17. pp. 39–56 (2017)
- [41] Wee, H.: Attribute-hiding predicate encryption in bilinear groups, revisited. In: TCC 2017. pp. 206–233
- [42] Yao, A.C.: Protocols for secure computations. In: 23rd Annual Symposium on Foundations of Computer Science (SFCS). pp. 160–164 (1982)
- [43] Zhang, B., Oliynykov, R., Balogun, H.: A treasury system for cryptocurrencies: Enabling better collaborative intelligence. In: NDSS 2019

A Postponed definitions

Definition 3 (Single secret leader election (SSLE)). A single secret leader election scheme is a tuple of PPT algorithms $SSLE = (SSLE.Setup, SSLE.Register, SSLE.RegisterVerify, SSLE.Elect_1, SSLE.Elect_2,$

SSLE.Verify) with the following behavior:

- $SSLE.Setup(1^{\lambda}, \ell, N) \rightarrow (pub, sk_1, ..., sk_N, st_0)$: The setup process generates public parameters pub, a number of secrets, and an initial state st_0 . N is an upper bound on the number of participants supported by the scheme, and ℓ is a lower bound on the number of required users per election. SSLE.Setup is a one-time setup process, followed by a series of elections.
- $SSLE.Register(i, pub, st) \rightarrow (rk_i, rt_i, st')$: Each user registers with a unique public identity $i \in [N]$, the public parameters pub, and the current state st. Registration outputs a registration key rk_i , gives a user a registration token rt_i , and modifies the state to st'. SSLE.Register must be run to participate in a series of elections. The eventual leader may be required to re-register for subsequent elections.

- SSLE.RegisterVerify $(i, rk_i, rt_i, pub, st) \rightarrow \{0, 1\}$:
 - SSLE.RegisterVerify is run by previously registered users after a new user registers to verify that the registration was carried out correctly. Verification can use the user's registration key rk_i , registration token rt_i , the public parameters pub, and current state st. The user's registration token rt_i can be modified as the result of SSLE.RegisterVerify.
- $SSLE.Elect_1(pub, st, i, sk_i) \rightarrow (p_i, l_i)$: Leader election begins by taking public parameters pub, current state st, a user U_i 's secret key sk_i , and outputting intermediate values p_i and l_i .
- SSLE.Elect₂(pub, st, i, p_i , l_1 , ..., l_m , sk_i , rk_i , rt_i) \rightarrow (p'_i , l'_i) $\cup \pi \cup \{\bot\}$: Leader election proceeds by taking public parameters pub, current state st, intermediate values p_i and l_1 , ..., l_m , user U_i 's secrets sk_i , rk_i , rt_i , and outputting either 1) new intermediate values p'_i and l'_i , in which case the algorithm will be executed one more time on appropriate inputs, or eventually 2) a proof of leadership π in case user U_i has been chosen as the leader, or a distinguishable symbol \bot otherwise.
- SSLE. Verify(pub, $i, st, \pi_i; p_i$) $\rightarrow \{0, 1\}$: Given index i, the state st, and optionally an intermediate value p_i from the election, the verification algorithm accepts or rejects the proof that user U_i has been elected leader. SSLE. Verify is used to check the authenticity of a participant who claims to be the leader when it is time for the leader to reveal herself.

If the eventual leader is required to re-register for subsequent elections, we say that an SSLE scheme with such a property has expiring registration. Otherwise, if the parties may re-use their registration for multiple elections regardless of the outcome, an SSLE scheme has non-expiring registration. Note that reregistration will change the public state st.

We could also formally include a revoke algorithm, to indicate that a user no longer wishes to participate. We refrain from such a formalism since it does not significantly impact the security properties, but our scheme can be modified to account for it.

Definition 3 specifies algorithms for setting up an SSLE instance, registering participants for the elections, verifying that the registration is performed correctly, electing the leader among registered parties, and verifying a proof of leadership. The setup algorithm generates private keys for the parties and introduces some initial state st_0 , and the state can be updated by the registration algorithm. The state is public and accessible by all parties during the entire execution of an SSLE instance. The registration algorithm provides a party P_i with a registration key rk_i and a registration token rt_i , which are kept private and used during elections. The difference between the key and the token is that the token depends on other parties participating in elections and therefore can be altered by the party holding it as the result of SSLE.RegisterVerify. The election proceeds in several rounds. In each round, party P_i computes private p_i and public l_i intermediate values, and public output values from all participating parties are used as input for subsequent rounds (SSLE.Elect₂). After the final round, each party holds either a proof of leadership π or \bot . The public state

and, optionally, private value p_i from the last communication round are used to verify a proof of leadership.

Next, we define an experiment for N parties between the challenger and an adversary, where the adversary is controlling c parties. This experiment will serve as the initial step in the security games.

Definition 4 (Experiment). We define an experiment $\mathsf{EXPR}[\mathcal{A}, \lambda, \ell, N, c]$ with security parameter λ , which is played between an adversary \mathcal{A} and a challenger \mathcal{C} as follows:

- Setup phase. A picks⁵ a set of indices $M \subset [N]$, |M| = c, of users to corrupt. C runs (pub, $sk_1, ..., sk_N, st_0$) \leftarrow SSLE.Setup $(1^{\lambda}, \ell, N)$ and gives \mathcal{A} the parameters pub, state st_0 , and secrets sk_i for $i \in M$.
- **Elections phase.** A chooses a set of users to register for elections and for any polynomial number of elections to occur, where \mathcal{A} plays the role of users U_i for $i \in M$ and \mathcal{C} plays the role of the rest of the users.

To register an uncorrupted user, \mathcal{A} sends the index i of the user to \mathcal{C} , and \mathcal{C} runs $(rk_i, rt_i, st') \leftarrow SSLE.Register(i, pub, st)$. To register a corrupted user, \mathcal{A} sends the index i of the user to C along with an updated state st'. In either case, C then runs SSLE.RegisterVerify (j, rk_i, rt_i, pub, st) for any previously registered user U_j , where $j \in [N] \setminus M$. If any call to SSLE.RegisterVerify returns 0, the game immediately ends with output 0. Otherwise, the state is updated to st'. Each election begins with C generating $(p_i, l_i) \leftarrow SSLE.Elect_1$ (pub, st, i, sk_i) on behalf of each uncorrupted registered user and A sending values l_i for any subset of corrupted registered users. Let $l_1, ..., l_t$ be the set of intermediate values l_i generated in this step. Then, elections may proceed with C generating $(p'_i, l'_i) \leftarrow SSLE.Elect_2(\mathsf{pub}, st, i, p_i, l_1, ..., l_t, sk_i, rk_i, rt_i)$ on behalf of each uncorrupted registered user and \mathcal{A} sending values l'_i for any subset of corrupted registered users. The intermediate values $l'_1, ..., l'_t$ are used as input for subsequent calls to SSLE.Elect₂. Let $l''_1, ..., l''_t$ be the last intermediate values generated by SSLE.Elect₂, and p''_i be the last private intermediate value of an uncorrupted registered user U_j .

Then, for all uncorrupted users, C sets $\pi_j \leftarrow SSLE.Elect_2(pub, st, j, p''_j, l''_1, ..., l''_t, sk_j, rk_j, rt_j)$ if user U_j has registered for that election or \perp otherwise. C sends π_j for each uncorrupted user to A.

If the SSLE scheme has expiring registration, the leader of an election should repeat the registration procedure after the election is over, and uncorrupted parties should verify registration. During the elections phase, multiple elections can take place.

The experiment in Definition 4 allows the adversary to corrupt parties statically during the setup. Then, the elections phase takes place, where an adversary chooses which parties will participate in an election. Those parties then register for the election. Each time a new party registers, all registered parties have to

⁵ We define the experiment is a way that c is given. Should \mathcal{A} pick this number, we will denote such an experiment as $\mathsf{EXPR}[\mathcal{A}, \lambda, \ell, N]$.

re-run SSLE.RegisterVerify to ensure that the registration data stored in the public state st is not malformed. During the election, the challenger plays the role of honest users, whereas a (malicious) adversary can send arbitrary messages on behave of the corrupted parties. As the outcome of an election, all non-registered (but participating in the election) parties receive \perp symbol, whereas registered parties may receive either \perp or a proof π .

Definition 5 (Uniqueness). The uniqueness experiment

UNIQUE[$\mathcal{A}, \lambda, \ell, N$] between an adversary \mathcal{A} and a challenger \mathcal{C} with security parameter λ extends $\mathsf{EXPR}[\mathcal{A}, \lambda, \ell, N]$ as follows:

- **Output phase.** For each election in the elections phase, \mathcal{A} outputs values π_i for each $i \in M$. The experiment outputs 0 if for each election with state st, there is at most one user U_{i^*} who wins that election. Otherwise the experiment outputs 1. We say user U_{i^*} wins an election if it outputs $\pi_{i^*} \neq \bot$ such that SSLE.Verify(pub, i^* , st, π_{i^*}) = 1.

We say an SSLE scheme is unique if no PPT adversary \mathcal{A} can win the uniqueness game expect with negligible probability. That is, for all PPT \mathcal{A} and for any $\ell < N$, $Pr[\mathsf{UNIQUE}[\mathcal{A}, \lambda, \ell, N] = 1] \leq negl(\lambda)$. If uniqueness only holds so long as there are at least t uncorrupted users participating in each election, we say that the scheme is t-threshold unique.

Informally, an adversary wins the uniqueness experiment if in at least one election in a series of consecutive elections there is more than one verifiable leader.

Definition 6 (Unpredictability). The unpredictability experiment UNPRED[$\mathcal{A}, \lambda, \ell, N, n, c$] between an adversary \mathcal{A} and a challenger \mathcal{C} with security parameter λ extends EXPR[$\mathcal{A}, \lambda, \ell, N, c$] as follows:

- Challenge phase. At some point after the elections phase, \mathcal{A} indicates that it wishes to receive a challenge, and one more election occurs. In this election, \mathcal{C} does not send (π_j) for each uncorrupted user to \mathcal{A} . Let U_i be the winner of this election. The game ends with \mathcal{A} outputting an index $i' \in [N]$. If, for 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 S is unpredictable if no PPT adversary A can win the unpredictability game with greater than negligible advantage. That is, for all PPT A, for any $c \leq n-2$, $n \leq N$, and for any $\ell < N$,

$$Pr[\mathsf{UNPRED}[\mathcal{A},\lambda,\ell,N,n,c] \mid i \in [N] \setminus M] \le \frac{1}{n-c} + nelg(\lambda).$$

If \mathcal{A} wins with advantage $\alpha + negl(\lambda)$ for $\alpha > \frac{1}{n-c}$, with α potentially depending on c, n, or N, we say that \mathcal{S} is α -unpredictable. If the value of α depends on N, then we require that n = N. If unpredictability only holds for c < t for some t > 0, we say that \mathcal{S} is t-threshold unpredictable.

Informally, in the unpredictability experiment the adversary asks for a challenge election after a series of elections. In this special election, the challenger

does not send to the adversary the outcome of elections. The adversary has to guess, who is the leader in this challenge election. If some corrupted party turns out to be the leader, the experiment is trivial and the result of the experiment will be always 0. Otherwise, if some honest party is the leader, the adversarial chances to correctly guess the leader should not be significantly greater than pure guessing.

Definition 7 (Fairness). The fairness experiment $\mathsf{FAIR}[\mathcal{A}, \lambda, \ell, N, n, c]$ between an adversary \mathcal{A} and a challenger \mathcal{C} with security parameter λ extends $\mathsf{EXPR}[\mathcal{A}, \lambda, \ell, N, c]$ as follows:

- Challenge phase. At some point after the elections phase, \mathcal{A} indicates that it wishes to receive a challenge, and one more election occurs. FAIR $[\mathcal{A}, \lambda, \ell, N, n, c]$ outputs 1 if there is no $i \in [n] \setminus M$ for which SSLE. Verify(pub, $i, st, \pi_i) = 1$ in the challenge election.

We say that an SSLE scheme S is fair if no PPT adversary A can win the fairness game with greater than negligible advantage. That is, for all PPT A, $n \leq N, c < n$, and for any $\ell < N$, $|Pr[\mathsf{FAIR}[\mathcal{A}, \lambda, \ell, N, n, c] = 1] - c/n| \leq negl(\lambda)$. If fairness only holds for c < t for some t > 0, we say S is t-threshold fair.

Weaker selectively secure definitions of unpredictability and fairness are not considered in this paper. A viable SSLE scheme must satisfy all the definitions above.

Informally, in the fairness experiment, the adversary asks for a challenge election after a series of elections. The probability of winning this challenge election by one of the corrupted parties should not be significantly greater than c/n, where c is the number of corrupted parties, and n is the number of parties registered for the challenge election.

B Security analysis

Lemma 1. Let $[C_A]$ and $[C_B]$ be the inputs to OSelectM protocol, and let [C'] be the output. Then, assuming the underlying secret sharing scheme is linearly homomorphic and the primitives for multiplication secret shares and generating a random shared bit are secure, it holds that $C' \in \{C_A, C_B\}$.

Proof. The underlying RandBit primitive produces shares of a random bit [b]. By homomorphic properties of the secret sharing scheme and security of the multiplication primitive, it follows that, if b = 0, C' evaluates to C_A , otherwise, if b = 1, C' evaluates to C_B .

Lemma 2. Let $[C_1], ..., [C_N]$ be the inputs to $OSelectM_N$ protocol, and let $[\overline{C}]$ be the output. Then, it holds that $\overline{C} \in \{C_1, ..., C_N\}$.

Proof. It follows from Lemma 1 and the binary tree structure of OSelectM instances in $OSelectM_N$.

Lemma 3. Assuming secret sharing is secure, algorithm Register in Construction 1 called by some party, securely implements sending a (register) message in the ideal model.

Proof. The party uses the output value from Register as input to the (register) message in the ideal model. The proof follows from simulatability of the secret sharing scheme.

Lemma 4. Assuming H is a random oracle, algorithm RegisterVerify in Construction 1 securely implements sending a (regVerify) message in the ideal model.

Proof. By the properties of the random oracle, we have that the probability that $C_i \neq C_j$ in the ideal model and $k_{iR} = k_{jR}$ is $1/2^{\lambda}$, which is negligible in λ .

Lemma 5. Algorithm Elect in Construction 1 securely implements sending a (elect) message in the ideal model.

Proof. We construct a simulator S for an ideal adversary A. S recovers the adversarial input from party P_i by reconstructing it from the shares available to the simulator (S controls enough honest parties to reconstruct any shared secret).

S sends all inputs from honest parties and the recovered adversarial inputs and receives $C_{U_1^N}$ from the ideal functionality as the result of the election. It is the same for all parties, including those controlled by the adversaries, so the simulator forwards this value to \mathcal{A} . In order to let the adversary believe it interacts with the real protocol, the simulator has to produce a transcript of the OSelectM_N protocol that will result in a specific value U_N to be chosen and output. To do that, the simulator fixes the shares of the honest parties for random bits [b] in OSelectM instances so that the reconstruction would output the specific fixed b, that will result OSelectM_N to select precisely the U_N -th element of the sequence (st_1, \ldots, st_N) . The underlying secret sharing scheme allows to simulate the transcripts for the honest parties that share a simulator-chosen secret.

In the real protocol, it is possible that for some $i \neq j$, $k_{iL} = k_{jL}$, while $k_{iR} \neq k_{jR}$ and so the parties would pass the registration. However, this only happens with a low probability that we can control.

Lemma 6. Assuming H is a random oracle, Construction 1 produces a unique leader with the probability at least $1 - e^{-\frac{N(N-1)}{2p}}$.

Proof. The probability that there exist two parties P_i and P_j such that $k_{iL} = k_{jL}$ and $k_{iR} \neq k_{jR}$ can be estimated by the birthday paradox. Specifically, this probability is bounded by $e^{-\frac{N(N-1)}{2p}}$.

Lemma 7. Algorithm Verify in Construction 1 securely implements sending a (verify) message in the ideal model.

Proof. It follows by a similar argument as in the proof of Lemma 4.

Proof (Proof of Theorem 1). Since we only consider sequential execution, we need to show that the adversarial view in the real and the ideal worlds is indistinguishable for one instance of the protocol, and the security of the whole protocol will follow by Canetti's composition theorem [8]. To this end, we construct a simulator for a real-world adversary as follows.

- For the registration, the real-world adversary and honest users use a call to the random oracle H for some (random) input and then share a string. Sharing algorithm can be simulated by a secure secret sharing scheme. Moreover, a *t*-private secret sharing scheme for t < N allows S to reconstruct the input used by the corrupted user. Hence, all the numbers shared by the corrupted and honest users during registration are known to S.
- To simulate the verification of registration, S verifies that there is no duplicate numbers recorded during registration. If this is the case, it outputs 1, otherwise 0.
- To simulate the election, S first consults the ideal functionality to elect the leader and then we use Lemma 5 to simulate the corresponding transcript.
- To simulate the verify algorithm, S compares the elected number with the number registered by the user (honest or malicious) and outputs 1 if the numbers are equal, otherwise it outputs 0.

We argue that any PPT environment \mathcal{Z} cannot distinguish between the ideal world and the real world significantly better than negligible probability via a series or games.

 G_0 - the real-world experiment.

 G_1 - same as G_0 , except that the election always returns a single leader. $G_1 \approx G_0$, since the probability of collisions during registration is negligible by Lemma 6.

 G_2 - same as $\mathsf{G}_1,$ except that MPC protocols (secret sharing, secret multiplication, and secret addition) are simulated by Lemma 5. We have that $\mathsf{G}_0\approx\mathsf{G}_1$ due to the properties of the secure secret sharing scheme, which allows for efficient simulation.

 G_2 - same as $\mathsf{G}_1,$ except that $\mathcal S$ consults the ideal functionality to determine the leader on the submitted numbers during registration. Since we have at least one honest user in the election who generates a random number for the registration and that the transcript can be simulated by the MPC functionality, we have that $\mathsf{G}_2=\mathsf{G}_1.$

 G_3 - same as G_2 , except that during the registration, parties submit random numbers (instead of calls to H), and for the verification they send theses numbers to S. We have that $G_3 \approx G_2$ by the properties of the random oracle.

 G_4 - the ideal-world experiment. We have that $G_3 = G_4$ by construction (parties use random number for the registration and are consulted by the ideal functionality to determine the leader).

Proof (Proof of Theorem 2). Since both our constructions Construction 1 and Construction 2 rely on secure MPC primitives and differ only in specifics of the used MPC frameworks, we simply follow the steps in the proof of Theorem 1 to prove the theorem.

C Postponed security analysis

Proof (Proof of Proposition 1). The goal is to demonstrate that any scheme that UC-realizes $\mathcal{F}_{\mathsf{SSLE}}$ fulfills all three game-based notions of (a) uniqueness, (b) unpredictability and (c) fairness. One needs to show the non-existence of a simulator given any of the game-based attackers: For some protocol π , assume an adversary against (a) and construct from it an environment distinguishing π and $\mathcal{F}_{\mathsf{SSLE}}$ (and same for (b) and (c)).

In the following experiments, assume to the contrary that some protocol π UC-realizes $\mathcal{F}_{\mathsf{SSLE}}$ and there is an ideal-world simulator \mathcal{S} of the real-world adversary. Before an experiment starts, a random bit b is drawn; \mathcal{Z}^* interacts with π in case b = 0, otherwise it interacts with \mathcal{S} in case b = 1. Let \mathcal{B} be an adversary successfully attacking some game-based property of π with non-negligible advantage $Adv_{\mathcal{B}}$. We construct an environment \mathcal{Z}^* as follows: it uses \mathcal{B} to set up malicious parties and communicate with the honest parties in the real world and with \mathcal{S} in the ideal world. Let *Exec* denote the output bit of \mathcal{Z}^* in the experiment.

Uniqueness. An adversary wins the uniqueness experiment if there are at least two users U_i, U_j for which the verification algorithm outputs 1 in the challenge election. At the end of the experiment, \mathcal{Z}^* runs the verification algorithm for each user on the final output of the election (running SSLE.Verify in the real world or interacting with \mathcal{S} that simulates the corresponding (verify) queries). If there is more than one user for which the verification algorithm (or \mathcal{S}) returns true, \mathcal{Z}^* outputs 0 (\mathcal{Z}^* believes it interacts with π). Otherwise, \mathcal{Z}^* outputs 1 (\mathcal{Z}^* believes it interacts with \mathcal{S}). On the one hand, we have that $Pr[Exec[\mathcal{F}_{SSLE}, \mathcal{S}, \mathcal{Z}^*] = 0] = 0$ due to the definition of \mathcal{F}_{SSLE} , which records only one leader for a single election and never changes it afterwards. On the other hand, $Pr[Exec[\pi, \mathcal{A}, \mathcal{Z}^*] = 0] = Adv_{\mathcal{B}}$, which is non-negligible. Hence, \mathcal{Z}^* can distinguish between the real and ideal worlds with non-negligible probability, which contradicts the assumption we made about π .

Unpredictability. An adversary breaks unpredictability if it can predict the leader (among honest parties) significantly better than pure guessing. At the end of the experiment, \mathcal{Z}^* computes the prediction as whatever \mathcal{B} does. If the prediction was correct, \mathcal{Z}^* outputs 0 (\mathcal{Z}^* believes it interacts with π), otherwise it outputs 1 (\mathcal{Z}^* believes it interacts with \mathcal{S}). On the one hand, we have that $Pr[Exec[\mathcal{F}_{SSLE}, \mathcal{S}, \mathcal{Z}^*] = 0] = \frac{1}{n-c}$ due to the definition of \mathcal{F}_{SSLE} , which uniformly at random chooses the leader among *n* registered parties, where *c* of them are controlled by the adversary. On the other hand, $Pr[Exec[\pi, \mathcal{A}, \mathcal{Z}^*] = 0] \geq \frac{1}{n-c} + Adv_{\mathcal{B}}$, where $Adv_{\mathcal{B}}$ is non-negligible. We have that \mathcal{Z}^* can distinguish between the real and ideal worlds with non-negligible probability, which contradicts the initial assumption about π .

Fairness. An adversary breaks fairness if it can significantly increase the chances for the corrupted parties to be elected. At the end of the fairness experiment, \mathcal{Z}^* learns whether one of the *c* controlled users (among *n* registered) is the leader. If this is the case, \mathcal{Z}^* outputs 0 (\mathcal{Z}^* believes it interacts with π), otherwise it outputs 1 (\mathcal{Z}^* believes it interacts with \mathcal{S}). On the one hand, we have that

 $Pr[Exec[\mathcal{F}_{\mathsf{SSLE}}, \mathcal{S}, \mathcal{Z}^*] = 0] = \frac{c}{n}$ due to the definition of $\mathcal{F}_{\mathsf{SSLE}}$, which uniformly at random chooses the leader among *n* registered parties, where *c* of them are controlled by the adversary. On the other hand, $Pr[Exec[\pi, \mathcal{A}, \mathcal{Z}^*] = 0] \geq \frac{c}{n} + Adv_{\mathcal{B}}$, where $Adv_{\mathcal{B}}$ is non-negligible. We have that \mathcal{Z}^* can distinguish between the real and ideal worlds with non-negligible probability.

Theorem 3. Assuming the underlying primitives are secure in the malicious adversary model, the statement of Theorem 1 holds in the malicious adversary model.

Proof. The theorem follows from Canetti's composition theorem [8] and Theorem 1.

Lemma 8. Assuming H is a random oracle, Construction 2 produces a unique leader with the probability at least $1 - e^{-\frac{N(N-1)}{2^{l(\lambda)}+1}}$.

Proof. Analogously to Lemma 6 with the difference that k_{iL} has $2^{l(\lambda)}$ possible choices.

D Extensions

D.1 Extensions to DDH-based MPC

It is possible to generate random bits in a fully non-interactive manner [12]. This will require an initial (trusted) setup for distributing randomness between the parties. For the base case where stakes are equal and N is a power of two, this improvement will not be significant, as the parties have to generate only $O(\log N)$ random bits, while requiring O(N) interactive multiplications of shares. In a general case, for arbitrary stakes, it will show a better reduction in cost.