A Secure Voting System for Score Based Elections

Lihi Dery
lihid@ariel.ac.il
Ariel Cyber Innovation Center
Ariel University
Israel

Tamir Tassa
tamirta@openu.ac.il
The Open University
Israel

Avishay Yanai
yanaia@vmware.com
VMware Research
Israel

Arthur Zamarin
arthurzam@gmail.com
The Open University
Israel

ABSTRACT

Dery et al. recently proposed [3] a secure voting protocol for score-based elections, where independent talliers perform the tallying procedure. The protocol offers perfect ballot secrecy: it outputs the identity of the winner(s), but keeps all other information secret, even from the talliers. This high level of privacy, which may encourage voters to vote truthfully, and the protocol's extremely lightweight nature, make it a most adequate and powerful tool for democracies of any size. We have implemented that system and in this work we describe the system's components – election administrators, voters and talliers – and its operation. Our implementation is in Python and is open source. We view this demo as an essential step towards convincing decision makers in communities that practice score-based elections to adopt it as their election platform.

KEYWORDS

Electronic Voting, Secure Multiparty Computation, Perfect Ballot Secrecy, Voting Protocols, Computational Social Choice

ACM Reference Format:
Lihi Dery, Tamir Tassa, Avishay Yanai, and Arthur Zamarin. 2021. A Secure Voting System for Score Based Elections. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3460120.3485343

1 INTRODUCTION

Ballot secrecy is an essential goal in the design of voting systems, since when voters are concerned for their privacy, they might decide to vote differently from their real preferences, or even abstain from voting altogether.

Dery et al. [3] presented a secure protocol with perfect ballot secrecy to compute election results for score-based voting rules. Perfect ballot secrecy [1] means that, given any coalition of voters, the protocol does not reveal any information on the ballots, beyond what can be inferred from the published results.

CSS'21, November 15–19, 2021, Seoul, South Korea

ACM ISBN 978-1-4503-8454-4/21/11.
https://doi.org/10.1145/3460120.3485343

In score-based elections over M candidates, $\mathbf{C} = \{C_1, \ldots, C_M\}$, each voter, $V_n \in \mathbf{V} = \{V_1, \ldots, V_N\}$, submits a *ballot vector*, $\mathbf{w}_n := (\mathbf{w}_n(1), \ldots, \mathbf{w}_n(M))$, that holds the scores that she gives to each of the M candidates. The winner is the candidate that receives the highest aggregated score from all voters (or the K highest, if the elections need to determine $K \geq 1$ selected candidates).

Each rule in this family defines the allowed ballot vectors. E.g., in the Plurality rule, each ballot vector must contain a single 1-entry, while the remaining $M-1$ entries are 0; the 1-entry is placed in the position corresponding to the voter's favorite candidate. Other rules in this family are: Approval (the ballot vector includes 1-entries for candidates that the voter approves, and 0-entries otherwise); Veto (the ballot vector includes a 1-entry for the voter's least preferred candidate, and 0-entries for all others); Range (the ballot vector contains scores for the candidates, where the scores are in a preset range $[0, L]$); and Borda (the ballot vector $\mathbf{w}_n$ is a permutation of $\{0, 1, \ldots, M-1\}$ which describes V_n's ranking of the candidates).

The protocol involves a set of talliers, $\mathbf{T} = \{T_1, \ldots, T_D\}$, to whom the voters send shares in their ballot vectors. The talliers validate the legality of the cast ballots, aggregate them, and eventually compute the final voting results, where all those computations are carried out by invoking secure multiparty sub-protocols, so that the talliers never obtain access to the actual ballots or other computational results such as the final scores of candidates. The protocol is secure under the assumption that the talliers have an *honest majority*. Employing more talliers (higher values of D) will imply higher costs, but at the same time it will provide enhanced security against coalitions of corrupted talliers. Such perfect ballot privacy, by which the ballots and aggregated scores are not disclosed even to the talliers, may increase the voters' confidence and, consequently, encourage them to vote according to their true preferences.

2 THE PROTOCOL

Protocol 1 is a high level description of the protocol presented in [3]. All computations are carried out in some finite field $\mathbf{Z}_p$.

The heart of the protocol is in Steps 4 and 6. In Step 4 the talliers run an MPC (Multi-Party Computation) sub-protocol for validating the legality of the ballot $\mathbf{w}_n$, in which they received shares in Step 3 (the vectors $\mathbf{w}_{n,d}$, $d \in [D] := \{1, \ldots, D\}$), without actually constructing it. The process of validating a ballot consists of computing products and sums of shared values. For example, for Plurality, each of the ballot entries must be either 0 or 1, which can be validated by computing the product $\mathbf{w}_n(m) \cdot (\mathbf{w}_n(m) - 1)$,

for all $m \in [M] := \{1, \ldots, M\}$. Those validations, for each of the five rules, are carried out by invoking the MPC sub-protocol of Damgård and Nielsen [2] for computing products of shared values.

In Step 6, the talliers sort the vector of aggregated scores ($\mathbf{w} = \sum \mathbf{w}_n$, where the sum goes over all legal ballots), in which they only hold shares ($\hat{\mathbf{w}}_d$), in order to find the K largest entries in it. This is done by computing an arithmetic circuit, presented in [5], which performs an MPC comparison of two secret values $u, v \in \mathbf{Z}_p$. Let u and v be two entries in the aggregated vector of scores $\mathbf{w}$. Each tallier T_d, $d \in [D]$, holds shares u_d and v_d in u and v, respectively, in a Shamir's D'-out-of-D secret sharing scheme [6], where $D' := \lfloor (D+1)/2 \rfloor$. The circuit outputs the bit that indicates whether $u < v$. That way, the talliers can find the winning candidates without learning any other information on the ballots or on the aggregated vector of scores.

Protocol 1 A protocol for secure score-based voting

Input: Ballot vectors, $\mathbf{w}_n$, $n \in [N] := \{1, \ldots, N\}$.
Output: The K candidates with highest aggregated scores.

1: Each voter V_n, $n \in [N]$, constructs a ballot vector $\mathbf{w}_n$ according to the voting rule.
2: Each voter V_n, $n \in [N]$, generates a random polynomial $g_{n,m}$ of degree $D' - 1$, where $D' = \lfloor (D+1)/2 \rfloor$ and $g_{n,m}(0) = \mathbf{w}_n(m)$, $\forall m \in [M]$. Then, she creates the share vector $\mathbf{w}_{n,d} = (g_{n,1}(d), \ldots, g_{n,M}(d))$, for each $d \in [D]$.
3: V_n, $\forall n \in [N]$, sends $\mathbf{w}_{n,d}$ to T_d, $\forall d \in [D]$.
4: The talliers $\mathbf{T}$ jointly validate the legality of each of the cast ballots $\mathbf{w}_n$, without revealing them.
5: T_d, $\forall d \in [D]$, computes $\hat{\mathbf{w}}_d = \sum \mathbf{w}_{n,d}$, where the sum is over all $n \in [N]$ for which $\mathbf{w}_n$ is legal.
6: The talliers $\mathbf{T}$ jointly find the K indices with largest entries in $\mathbf{w}$ (the sum of all legal ballot vectors) and output them.

3 IMPLEMENTATION

We have implemented a demo of a voting system that is based on Protocol 1. The demo illustrates the three modules of the system: election administrator, voters and talliers, and it allows users to experience a simple interface for the administrator and voters, and to witness the efficiency of the protocol.

The system is implemented in Python. We chose this programming language for best cross-platform support. In addition, it renders the code easier to read, to verify, and to modify as required. The full source code of our demo is open source and can be found at GitHub: https://github.com/arthurzam/SecureVoting.

3.1 The election administrator

The first module of the system is the election administrator. That is the module which initiates a new election campaign. The administrator determines:

- The election title (e.g., "Electing a new faculty dean").
- The set of candidates $\mathbf{C} = \{C_1, \ldots, C_M\}$ and their indexing.
- The set of eligible voters, $\mathbf{V} = \{V_1, \ldots, V_N\}$, together with their email addresses and unique identifiers.
- The voting rule.
- The number K of candidates out of $\mathbf{C}$ that need to be elected.
- The number D of talliers, and their TCP/IP ports/addresses.
- The election termination condition (e.g., a specified time).

After establishing a new election campaign, the election administrator will receive a `config.json` file that will hold all configuration details of the election. The administrator will send that file to all talliers, and a shortened version of that file (without the details of all voters) to each of the voters.

3.2 The voter module

The voter module can be executed on various devices (laptops, tablets, smartphones), and is intended to be operated by non-expert voters. Hence, the main design goal here was to keep the module simple, so that it will be easy to migrate it between platforms, and to keep the interface simple and clear.

The simple graphical interface presents the voter with the list of candidates, and instructs her on how to enter her ballot. For example, in Plurality, the voter is asked to select exactly one candidate. In Range, the voter needs to insert a score, out of a given range, for each of the candidates. In Borda, the voter is instructed to rearrange the order of the candidates from her most favorite to her least favorite.

Once the voter has finished entering her ballot, the voter's module will use the `config.json` file in order to create the proper ballot vector (Step 1 in Protocol 1), create shares in them (Step 2), and send those shares, signed and encrypted using SSL, to the talliers' addresses as listed in the configuration file `config.json` (Step 3).

In order to support various types of voting devices, we developed a specialized version of the voter module, implemented on fully client-side web code. It receives the election information through the URL parameters, presents them to the user, manages the vote casting and the connection to the talliers. This module is implemented using HTML and JavaScript, and can thus be run on any browser in any device.

3.3 The tallier module

The tallier module is the most complex one: it establishes the synchronized network with other peer talliers, collects the ballot shares from all voters (Step 3 in Protocol 1), validates their legality, together with the other tallier modules (Step 4), aggregates shares of all legal ballots (Step 5), and consequently takes part in an MPC sub-protocol (with the other tallier modules) in order to determine the winner(s) (Step 6).

As this module handles heavy network transport and a significant load of mathematical computations, our implementation uses concurrency, as provided by Python's coroutines from the `asyncio` library. Our implementation is self-contained and does not depend on external libraries. In particular, we fully implemented the MPC sub-protocols for multiplication [2] and number comparison [5].

To improve the performance of an MPC evaluation of a given (arithmetic) circuit, it is usually better to construct the whole circuit (which is a directed acyclic graph over addition and multiplication gate nodes), and then to compute in parallel all gates in a given layer of the circuit, since those gates are independent. However, such implementations result in great inflexibility, as they are hard to read, maintain and modify. Therefore, we decided to depend on the asynchronous nature of the network, and use network coroutines for asynchronous development of each gate by itself, and

D	M	Voters latency (msec)		Max load (per sec)		Time to compute the winner (sec)		Network transport (MB)	
		L	W	L	W	L	W	L	W
3	2	14	1,679	300	100	0.12	5.99	0.06	0.02
3	8	108	1,787	111	67	0.80	21.89	0.34	0.11
3	32	683	1,721	26	19	5.12	41.85	2.53	0.77
9	2	112	2,132	90	77	0.39	7.29	0.28	0.19
9	8	469	2,018	37	30	2.13	24.03	1.73	0.60
9	32	2,211	2,565	9	6	13.92	51.44	10.77	4.19

Table 1: Election measurements for D talliers and M candidates (L and W stand for LAN and WAN)

pass the responsibility of scheduling the network buffers to the operating system's network schedulers. Such schedulers run efficient algorithms for managing network traffic, which are based on heavily-researched network queueing disciplines. One possible consequence of our approach is that it might result in large numbers of small packets that could reduce efficiency. We address that potential problem by tweaking Nagle's algorithm [4] as needed.

Using the `config.json` file that was sent by the election administrator at the initiation of the election campaign, the talliers set the synchronous network and start accepting ballots from the voters and replying with the vote validation result. After the voting period ends, the talliers engage in an MPC protocol that computes the winner(s).

The reader is referred to the `ReadMe` file in our GitHub project for more explanations about the system. In particular, it includes instructions on how to set up a new election campaign, how to vote, and how the results are announced. The readme file includes also screenshots of the different module interfaces.

4 EXPERIMENTS

We evaluated our system on both Local and Wide Area Networks (LAN and WAN). All benchmarks were performed with $D \in \{3, 9\}$ talliers, $M \in \{2, 8, 32\}$ candidates, and one ($K = 1$) winner. The underlying field was $\mathbf{Z}_p$ with $p = 2^{31} - 1$. We implemented all five voting rules, but in the presented benchmarks the voting rule was Range with $L = 10$. We simulated ballots from $N = 10^4$ voters, using two extreme voting throughput scenarios: one with sequential voting and one in which all ballots were submitted simultaneously. We stress that our current implementation can be applied efficiently for up to roughly 500M voters, and can handle even larger scenarios by simple code adjustments.

When running local elections (say, selecting a faculty dean or members to a company's board), the talliers' servers are expected to be nodes in a LAN. Such settings result in short Round Trip Time. We ran the LAN tests on AWS EC2 m5d.4xlarge (16 cores, 3.1 GHz Intel Xeon Platinum 8175M processors, with 64GB memory). The machines were selected in the same availability zone in a data center in Ohio, over a network with bandwidth 7.5Gbps.

To simulate elections with voters that are spread over a wide area (say, national elections), we tested the performance of our voting system when the talliers are nodes in a WAN. To that end we set up a WAN over Amazon servers (16 cores, 3.1 GHz Intel Xeon Platinum 8175M processors, with 64GB memory), with talliers that are spread evenly in Ohio, Ireland and Singapore. Such settings result in longer Round Trip Time, which significantly affect the runtime of MPC computations. The network bandwidth was around 26-40 MBps.

In the sequential voting scenario, we generated a slow and steady stream of votes, and measured the *voter's latency*, i.e., the time that elapsed since the voter had submitted her ballot until she received a confirmation that the ballot was legal and processed (Table 1, third column). On the other hand, in the simultaneous voting scenario, where all $N = 10^4$ ballots were submitted at the same time, we measured the *max load* per second, i.e., the number of ballots that could be processed simultaneously (Table 1, fourth column). In addition, we measured in both scenarios the *time to compute the winner* of the election, at the completion of the election period (Table 1, fifth column), and the total *network transport* usage for each tallier (Table 1, sixth column). All measured values are reported for both LAN and WAN (left and right sub-columns, respectively).

The measurements on LAN show that even our current implementation of the demo can hold against high loads of voters, so that it can be readily deployed in real life elections.

When measuring on WAN, the CPU usage was very low (around 5-10%) because the main bottleneck was the network. Memory usage was high (up to 260 MB), mainly due to network buffers. We witnessed a better transport layer utilization owing to our configuration of the operating system that uses Nagle's algorithm [4].

On the voter's end, the time to generate secret shares from the voter's ballot vector with $M = 8$ candidates and $D = 9$ talliers was around 25 ± 1 μsec on a PC (4 cores, 3.5 GHz Intel Core i5-6600K processors, with 8GB memory), and around 30 ± 2 μsec on a mobile phone (OnePlus 3T, 4 cores, 2.35 GHz Qualcomm MSM8996 Snapdragon 821 processors, with 6GB memory). Those are clearly practical runtimes.

5 CONCLUSION

We demonstrate that the secure electronic voting protocol of Dery et al. [3], based on secure multiparty computation (MPC), is practical, even with a relatively large number of talliers and over WAN. One of our conclusions is that in application scenarios where the main bottleneck is the network (rather than CPU time), it is beneficial to write the system in a high level language, like Python, which increases openness, readability, and mutability, and hence potentially also increases the trust in the system. We encourage developers of similar applications to follow that direction.

ACKNOWLEDGMENTS

Lihi Dery's work was supported by the Ariel Cyber Innovation Center in conjunction with the Israel National Cyber Directorate in the Prime Minister's Office.

REFERENCES

[1] David Chaum. 1988. Elections with Unconditionally-Secret Ballots and Disruption Equivalent to Breaking RSA. In *EUROCRYPT*. 177–182.
[2] Ivan Damgård and Jesper Buus Nielsen. 2007. Scalable and Unconditionally Secure Multiparty Computation. In *CRYPTO*. 572–590.
[3] Lihi Dery, Tamir Tassa, and Avishay Yanai. 2021. Fear not, vote truthfully: Secure Multiparty Computation of score based rules. *Expert Syst. Appl.* 168 (2021), 114434.
[4] John Nagle. 1995. Congestion control in IP/TCP internetworks. *Comput. Commun. Rev.* 25 (1995), 61–65.
[5] Takashi Nishide and Kazuo Ohta. 2007. Multiparty Computation for Interval, Equality, and Comparison Without Bit-Decomposition Protocol. In *PKC*. 343–360.
[6] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22 (1979), 612–613.