Random Sample Voting: Security Proof

Editors: David Chaum, Aggelos Kiayias, Douglas Wikström, Bingsheng Zhang
Summary: Jeremy Clark

A summary of the security protocol can be found in the random sample voting whitepaper by Chaum, both in summary form (Section: INFORMAL SUMMARY OF TECHNICAL CONCEPT) and technical detail (Appendix: DETAILED ELECTION PROCESS). The security proof of random sample voting is done using the simulation-based approach of the universal composition (UC) model by Canetti, which we describe very briefly and informally now.

In the case of many cryptographic protocols, if a fully trusted party (one that follows every instruction and keeps every secrets) existed, one could accomplish the same thing as the protocol without using any (or much) cryptography at all. In the UC model, one describes what such a trusted party would do (called an ideal functionality) in order to accomplish the goals of the protocol. Since these trusted parties do not actually exist in the real world, one also describes a cryptographic protocol that sets out to accomplish the same thing. Finally, one formally proves that the behavior of the protocol realizes that of the ideal functionality.

System Components

Random sample voting is a complicated protocol, involving many components. We thus use a set of ideal functionalities to support the main functionality. We assume the existence of a bulletin board functionality, which is common to many cryptographic voting protocols, and is effectively an append-only broadcast channel. We assume functionalities for one- and two-way authenticated channels. Finally, we define an ideal functionality for a random beacon. A beacon is a public source of randomness that cannot be manipulated (e.g., lottery numbers, stock prices, weather patterns). A beacon is used in the real protocol to produce unpredictable numbers for selecting units of the protocol's data-structure for auditing. This leads to a technicality in the proof where we appear to require the simulator's ability to program the beacon, similar to how a simulator might program a random oracle.

Core Functionality

Our ideal functionality for random sample elections currently consists of the following functions (described very informally):

- `Create`: creates an instance of an election by recording a description of the ballot and the number of ballots—both real and fake—to be used in the election.
- `Sample`: selects from a list of voters a random subset to receive the real ballots and a random subset to receive the fake ballots.
- `Deliver`: send a real/fake ballot to each of selected voter after the adversary has potentially appended an annotation of his choosing.
- `Vote`: receive and record vote from voter and notify the adversary of who voted (and how if the EA is corrupt)
- `RecordVote`: receive "approval" from the adversary to finalize the vote along with a unique receipt value that is returned to the voter (to be used by the voter to perform `Audit` below)
- `Tally`: compute the result of the received real ballots and report the result to adversary
- `ReportTally`: receive a potentially modified set of votes and the resulting tally from the adversary; receive approval to finalize the tally
- `ReadTally`: report the adversary's tally to the voter and a bit indicating if this tally is the same as the one computed by `Tally` or not.
- `Audit`: receive a confirmation code and report a bit to the requester indicating if the candidate recorded in `Vote` matches the adversary's set of votes or not.

Open Problems

We are pursing further research along these lines:

- Completing the simulator to prove the protocol realizes the ideal functionality
- Tweaking the protocol, model, and adversarial assumptions to provide stronger security guarantees
- Reconsidering the format of the data structure composing the backend of the system
- Developing lighter-weight security arguments that might be more intuitive to a less technical audience to complement the full proof