The use of key pools to provide both anonymity and authentication in online voting

Background:

An important source of difficulty in implementing an online voting system is the presence of two necessary but to some extent conflicting demands:

1, On one hand we need voter authentication, we want only those who have the legal right to vote in a given election to be able to do so, and we do not want those to be able to cast a lrager number of votes than that to which they are legally entitled[citation not really needed, it's pretty obvious]

2, On the other hand, we want the votes to be anonymous, it should not be possible to determine how a given voter voted, or to determine the voter by which a given vote was cast[ibid]

This paper proposes a solution based on key pools (a kind of mixer), that will allow for distribution of vote tokens, which will later be used for the actual casting of votes[1][2], in such a way that while only authenticated voters will receive vote tokens (no voter will receive more tokens than the intended number[3])

// describe key pools here?

1 The idea here is that the possession of one vote token is a necessary and sufficient to be able to cast one vote

2 This proposa; was written with a solution in mind that uses the Bitcoin block chain for the voting, with the voting tokens here being Bitcoin addresses and the passwords needed to to make transfers from these addresses, however the nothing in the pool solution here proposed necessarily relies on any particular assumptions about the voting tokens.

3 The assumption throughout this paper is one token per voter, however the proposed solution can easily be modified to allow for other numbers; in a situation where every voter gets n votes, the server may simply encrypt n tokens with each key submitted to the key pool, in a situation where different voters get a different number of votes (e.g. based on share ownership), a voters may simply submit a number of keys equal to their number of votes to the pool, and the server will not close the pool until a number of keys equal to the key total for all voters in the pool has been submitted.

Proposed solution:

Procedures, how voters and the voting server interact to ensure authenticated, anonymised distribution of voting tokens

1) **Voter authentication and pool assignment**
    1) Voter connects to voting server and authenticates themselves (using e.g. Bank ID or similar)
    2) The voting server checks that the voter has not already voted
    3) Voting server assigns voter to key pools
    4) Voting server waits for pool to be full
    5) Voting servers transmits login details (pool ID & password) to all voters in pool
        - The login details are obviously the same for all voters in the pool
2) **Anonymous pool access**
    1) Each voter generates a new asymmetric key pair[4]
    2) Voters connect to the voting server again over an anonymous channel (e.g. through Tor)
    3) Voters connect to their assigned key pool using the password and pool ID they were provided with over the authenticated channel
    4) Voters write their newly-generated, anonymous public key to the key pool, together with a random identifying nonce
    5) When the number of keys written to the pool equals the number of voters assigned, the server closes the key pool
    6) If the number of keys submitted does not reach the number of voters assigned to the pool within a given time limit, the voting server will automatically considered the pool unsuccessfully signed and redistribute the voters to new pools (see point 4, pool termination, below)
3) **Authenticated pool inspection and signing**
    1) Once the key pools has been closed, voters use the authenticated channel to inspect the contents of the key pools
    2) If and only if they find their own anonymous key among the keys submitted to the pool, voters uses some type of digital signature to confirm this (without in any way specifying which key is theirs, of course)

4 To prevent voter identification, this key pair is never used for anything else. If the pool is not successfully signed (below) and the voter is assigned to a new pool, a new key pair is generated for submission to that pool.

   3) If and only if all voters assigned to the key pool signs to confirm that their key was submitted, will the voting server consider the pool to be *successfully signed.*

4) **Pool termination**
   1) If the key pools is not successfully signed within some reasonable time limit, all data in the pool will be considered void and for each voter in the pool, the voting server will go back to step 1.2, assigning the voter to a new pool to repeat the process[5]
   2) If the key pools is successfully signed, the server will use the keys submitted to the pool as a basis for vote token distribution, below**.**

5) **Vote token distribution**
   1) If the pool I successfully signed, the server will create a list of the identifying nonces submitted to the pool, each paired with a vote token encrypted with the key submitted together with that nonce.
   2) The server then distributes this list of nonces and encrypted tokens to all voters assigned to the pool.
   3) Each voter then scans the list for their nonce, and use their private key to decrypt the corresponding vote token. The voters can not decrypt and access the tokens of other voters in the pool , since they do not have access to the corresponding private keys.

5 Randomly distributing the voters among new pools will ensure that even if one or more voters in the original pool try to sabotage the process in one way or another (e.g. by refusing to sign, or by uploading multiple keys to the pool), the other voters in the pool will eventually be assigned pools with no saboteurs. Whether this can be expected to occur within a reasonable time, and what expectations underlie that assumption, will be discussed in the next section.

Difficulties

The process above assumes that the users (voters) cooperate as intended. There are several ways an user may instead attack the system, either simply to perform a denial-of-service attack, preventing others from getting their voting tokens, or to gain access to an unauthorized amount of voting tokens for themselves.

Sabotage through refusal to sign

An obvious denial-of service-attack would be for voters simply to refuse to sign the key pools. Since all voters must sign the pool before key distribution may begin, the presence of a single saboteur will be enough to prevent the pool from being successfully signed.

Assuming an average key pools size of 1000 voters, of which 0.1% are saboteurs, the probability of a given pool containing no saboteurs is $(1-1/1000)^{1000}$, or about 37%. Below follows a table of probabilities that a given pool will be saboteur-free given various pool sizes and saboteur fractions:

Fraction of pools successfully signed (presumably unacceptably small fractions shaded)**:**

Saboteur fraction (columns): Pool size (rows):	1/10 000	1/1000	3/1000	1/100
100	0.99	0.9	0.74	0.37
500	0.95	0.6	0.22	0.0066
1000	0.9	0.37	0.05	0.000043
5000	0.6	0.0067	0.0000003	~0

The fraction of voters willing to attempt sabotaging the process in this way is of course an empirical question yet to be determined, but I find it unlikely that it would be larger than 1%.

How large the fraction of successfully signed pools has to be for the system to be functional chiefly depends on how fast a round of pool-assignment-key submission-pool inspection can be made. In principle, the whole process can be automated, in which case an important speed constraint becomes key generation times.[6] This can be compensated for by pre-generating a number of key pairs before connecting to the server.

Another speed constraint is connection times for the the anonymous channel. If running this through Tor, the safest thing would presumably be to disconnect from the Tor network entirely, and then reconnect again before accessing another key pool, which could take several seconds.

6 RSA key pair generation speeds here? Voters should not attempt to submit the same key again when assigned a new key pool, as this could be used to identify them.

It could perhaps also be argued against entirely automating the process, instead requiring voters to manually submit their keys and to inspect the key pool for them, in which case requiring the voters to do this more than a handful of times at most is likely to cause unacceptable levels of frustration.

Sabotage through submitting multiple keys to the same pool

Voters may also sabotage the process by writing more than their permitted number of keys to the key pool. As the server closes the pool when the expected total number of keys have been submitted, this would cause other voters to be unable to submit their keys. Since these voters would then not sign the key pool, this would not cause the offending voter to actually acquire more vote tokens. It would however work as a denial of service attack, and as such may be more difficult to track than mere refusal to sign (above), if the same voter repeatedly is the only one not sign multiple key pools, once can be fairly confident singling them out as an attacker, if that same voter instead attacks by submitting, say, half the total number of keys to the pool, half the other voters in the pool will successfully submit their keys (assuming the pool only has one attacker) and the other half will not, meaning the attacker will not stand out as directly on a single-pool basis, and, if the total fraction of attackers is high enough, may be hard to single out even after attacks on multiple pools.

The fraction of pools affected given different pool sizes and fraction of attackers are the same as above.

Sabotage through submitting keys to multiple pools

Hitherto the assumption has simply been that the voting server simply keeps a record of all voters who have been part of successfully signed key pools, and, once a voter connects to the server and authenticates themselves, will be assigned to a new pool only if they are not already on this list.

Assuming multiple voting servers are used concurrently, or that the same voting server accepts multiple concurrent connections from the same authenticated voter, a voter could make several such connections in rapid succession, thus being assigned to multiple key pools before any one of them has been successfully signed, potentially getting vote tokens from all of these pools.

To prevent this, the servers need not only keep track of which voters have been part of successfully signed pools, but also of which voters currently are assigned to open pools.

Note, however, that in order to carry out this attack the voter must authenticate themselves in each instance, meaning it is trivial to detect and punish those who do so, making it a very unattractive form of attack.

If an attack is discovered after the distribution of vote tokens, but before the actual voting, it should also be possible to invalidate all vote tokens from the affected pools and have the voters from those pools acquire new ones.

Other difficulties?

Attacks by other actors than voters?

Sufficient key pool sizes to guarantee an acceptable level of voter anonymity?