Electronic Voting: Concept and Reference Implementation

Contents

Motivation

Electronic voting consistently fails to supplant conventional paper ballot due to a plethora of security shortcomings. Not only are traditional elections mediocre in terms of transparency and verifiability, they also encompass the principal-agent problem, where a state may have vested interest, and its ruling cohort – the real capabilities to tamper with the elections' outcome. It is exactly that conflict that gave rise to Capitol Hill riots we observed at Trump's departure for example. Electronic voting protocol using cryptography to deliver in zero trust environment is long overdue. Here, I propose applying Proof of Work by user devices across general population – as opposed to miners known from blockchain practices – to build a zero-trust voting system. The state would only issue single-use voting authorizations to those eligible to vote, while zero trust design would effectively convert elections into a specialized publishing process.

New Roles

There would be three roles involved in the proposed voting protocol, and their characteristics are as follows:

1. An authority – electoral commission, governmental portal or similar. We may have no better option than trusting the authority to define voters. The authority can electronically sign a one-time-pass (unique but not personally identifiable token) and share it with the voter as an authorization (mandate) to vote. The authorization identifies the voter *to the authority*; it gets published in the list of voters. Anyone could verify electronic signature on such authorization and, thus, establish that the holder was authorized to vote without knowing who they are.

2. A platform outside the authority's control. We need such platform to reliably sever the link between the authorization to vote (personally identifying the voter to the issuing authority) and the vote itself. The platform is a mere publisher of two unrelated lists: a list of voted authorizations (voters) and a list of casted votes. The platform's software is preferably open source. Concluding election's outcome from the published lists is trivial: it may or may not be in the scope of the platform.

3. A voter casts their vote with open-source software running on their device(s). This software is likely provided by the platform. It takes some time to calculate cryptographic digests over the vote being casted, a nonce, and some other data including, most notably, a unique identifier generated by voter themselves. That identifier plays a key role in end-2-end verifiability.

Architecture

Initial publication offers some grassroots insights into security standing and trustless interactions between the roles in question. The following diagram outlines what a voter sends and what gets published:
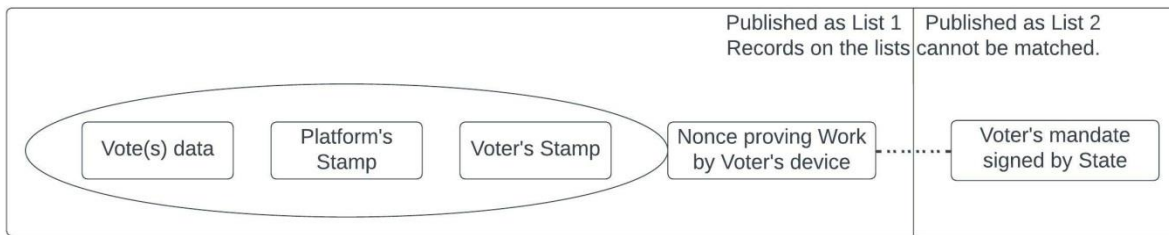
Figure 1 | A data block en route from a Voter's device to the platform. Stamps ensure the nonce proves the work within the voting period; proof of that work deters tampering at scale. Voter's stamp makes published votes end-to-end verifiable. Voter's mandate ensures that the voter voted once and was authorized. The platform splitting the data into two unrelatable lists ensures secrecy. No personally identifiable data get disclosed.
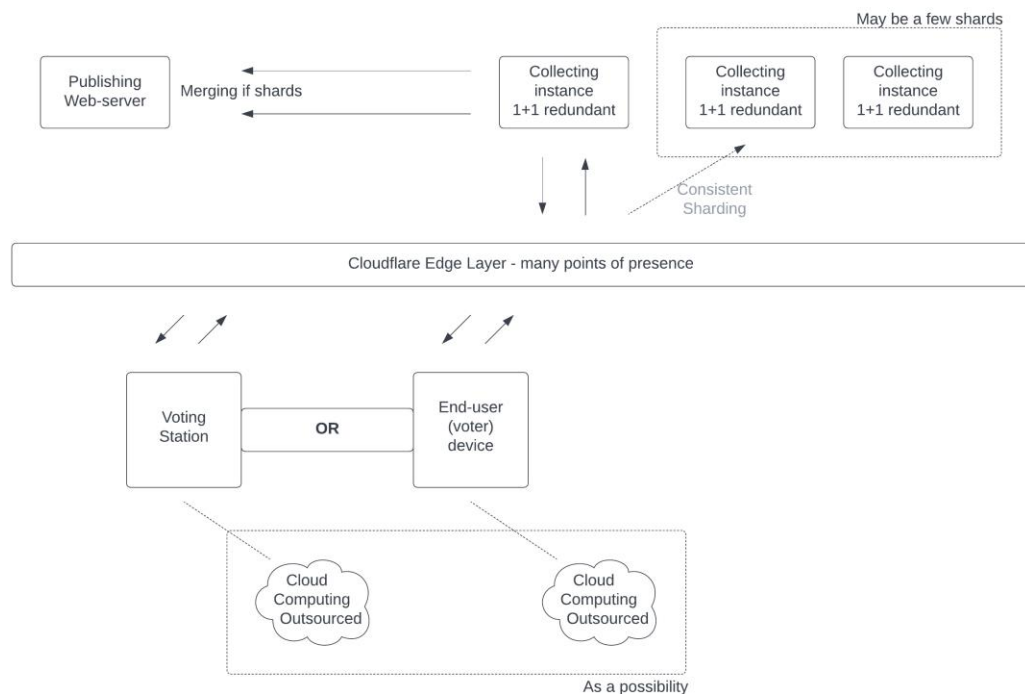
Top-level system architecture:

Figure 2 | High-level architecture of an electronic voting system with distributed integrity guard (voters' work) and isolating edge layer to mitigate potential denial-of-service attacks.

Persistent [filesystem-based hash table](#) is used to store the voter list because it is the fastest way to control for uniqueness. If sharding is employed Cloudflare selects a shard by the State-signed authorization (because it has to be controlled for uniqueness); lists from an array of collecting instances can be merged at a publishing server.

State-signed ID

As the list of voters is published, it must comprise non-personally identifiable IDs for the voters. One exploratory option was the contents of biometric passports: a State-signed hash of some unique input. Some obvious limitations here are citizens without passport and dependency on NFC capability of a mobile phone. Example from a biometric ID:
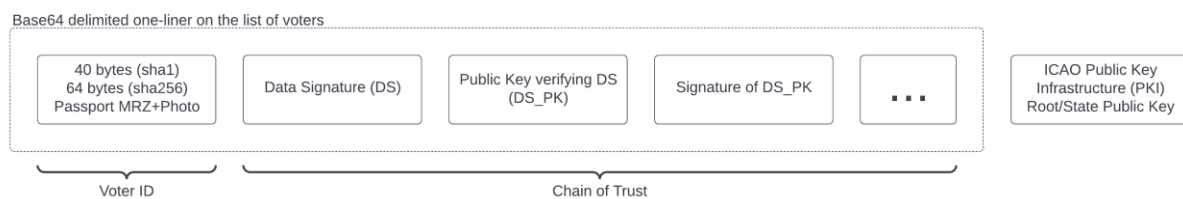

Figure 3 | State-citizen chain of trust embedded in contemporary machine-readable travel documents. A possible but not preferred option in this design.

Furthermore, elections may be misaligned with citizenship and disclosing signatures stored in biometric IDs may not be considered secure. Besides, [this investigation](#) shows cumbersomeness of x509 approach and inconsistencies between participating countries.

It is more practical for electoral authorities to issue dedicated electronically signed voting IDs, using light-weight Ed25519 signatures. In this writing I interchangeably use terms "voting ID", authorization-to-vote, and voting mandate. It constitutes a unique random sequence of 24 bytes + their Ed25519 signature (64 bytes) + public key of a signer (32 bytes).

As there may be many issuers of the mandates, the voting platform will maintain a set of valid public keys and check all incoming votes (or voting HTTP requests to be precise) against that list; that is, the trust model is flat as opposed to 2-level hierarchy in ICAO PKI.

After checking voter eligibility, a voting station's staff can print out (or take from a stock printed beforehand) a voting card. Here is an example with 128 bytes encoded:

```
Crockford's Base32 scheme with checksum
  1.    R0K    HSN    5K3    A47    MZ
  2.    77R    360    CFG    Q4D    05
  3.    VYB    HBK    FS1    M79    EA
  4.    5FQ    ZH6    YES    SKZ    6*
  5.    75X    9CZ    K1K    K6V    T$
  6.    4GY    E2A    3HN    67B    WF
  7.    896    9CS    ETY    FEP    P9
  8.    54C    R8M    KM0    EJP    PC
  9.    2HG    2MZ    K4P    QZB    CS
 10.    HM8    GPJ    VE7    NAS    WP
 11.    Z5T    9HP    18R    59F    8~
 12.    8HN    YAR    DND    1MM    J~
 13.    9EJ    ZQH    3PC    NHV    2S
 14.    YW5    4KD    G98    Y2S    68
 15.    MW1    DD3    76Y    ESP    2W
 16.    RPT    ZXY    5GF    PQC    AZ
```

Figure 4 | Crockford's Base32 representation of binary data for human-friendly transference e.g., zero and letter O count as one same character. Each line is protected by a checksum for early typo detection. Can have a QR-code alternative.

We ask voters punch these lines into their devices manually because it would buy time for their devices to perform protective work, proof of which we eventually need anyway. So rather than asking to wait – as we would have to in the case of biometric ID above – why not keep them busy while their device(s) crunch numbers?

Alternatively, with a better user experience in mind, we could give this data on a QR image and ask to wait. Moreover, we could ask to wait as if in electronic queue to see a voting station staff.

Note, that chosen implementation verifies checksum for each line, so error detection/localization is done locally in the GUI, which should be convenient for the user and excluding mere possibility of a voting transaction with erroneous mandate.

Preferred Procedure

To avoid coercion – familial or organizational peer pressure counts here – the voting moment has to be spent in a voting booth/area, where public observation is known to be the best remedy. Platform controlled TOTP (time-based one-time password) is displayed in the booths to enforce this procedure. It has to be Platform controlled because States could engage in coercion otherwise. Implementation details for TOTP are given in a later section of this document. During election, voter devices need be online; all devices used by voting stations (TOTP generators and voter ID/Authorization printers if any) need not be online.

1. A voter is asked to vote and enter TOTP on the same screen. The vote must be (due to step 10) a choice from a finite enumeration, such as a dropdown menu for example.

2. The voter's device use TOTP to requests (via Internet) a Platform stamp. Upon receipt of the stamp the screen changes. The voter is placed to an "e-queue" to see a voting station staff.

3. Devices at voter's disposal start crunching numbers to find the best nonce (proof-of-work algo). The voter is asked to go to a common area – to vacate the limited number of the booths.

4. We may want between 5 and 10 minutes spent on proof-of-work, so if the voter would scan their mandate as QR code they would need to wait this long before seeing the staff.

5. If the voter is entering their mandate manually, they can see the staff and get a card from Figure 4 straight away. Then, they would be entering their mandate during the wait.

6. The station staff checks voter's eligibility and either let the voter scan the mandate as a QR-code or give away a card shown in Figure 4 for the voter to enter manually.

7. Voter device sends a single HTTP request containing voting data protected by proof-of-work (PoW) and authorization data (their mandate).

8. Cloudflare makes edge checks on the voting block (nonce isn't cached, PoW threshold is met, Platform/Voter stamps are valid). Then, the request is sent to the platform.

9. A collecting instance verifies: data comes from Cloudflare; the voter is authorized but hasn't already voted etc. Then, the vote-voter lists are updated with respective records.

10. Platform's response has all voting options with a voter stamp against each. Only the option the voter has chosen has their stamp, other options – other voters' recent stamps.

Voter stamp is part of the voting record, it is an entropy generated by Cloudflare. Because the voter has this string as part of their receipt, they can trivially find their voting record in the vote list published online. This is to ensure end-2-end verifiability (E2EV). On the other hand, as the receipt has all other options as well, it cannot disclose how the voter voted to anyone other than the voter, because one has to know which stamp to use (which choice they made).

The voter may enter their email in step 5 to get a copy of the receipt and engage themselves in E2EV.

Booth Call-flow

A voter is asked to enter time-based one-time password (TOTP) linked to a publicly observed place – the voting booth. TOTP displayed in the booth comprises seven digits: 6 decimal digits are actual TOTP, and the 7th digit is a sum of the first six mod 11 – the checksum. 11 is the first prime number greater than 10, so the checksum character is from *[0-9, A]*. For example, a voter enters 263 917 6. Then, a software on the voter's device can verify $2 + 6 + 3 + 9 + 1 + 7 = 28$ *modulo* 11 = 6. The TOTP is accepted, and processing continues. If there were no match the voter would be asked to verify the input.

We need this trivial user-side verification because we will not be able to early detect an error the voter makes typing-in those six digits. That is, such error is a high stake, because it would lead to the voter waiting for their proof-of-work computations to complete, and then the platform would decline the vote. On the other hand, we shall not validate TOTP early, because an adversary could use our "invalid TOTP" response for brute-force extraction of a valid platform stamp. The exchange is as follows:
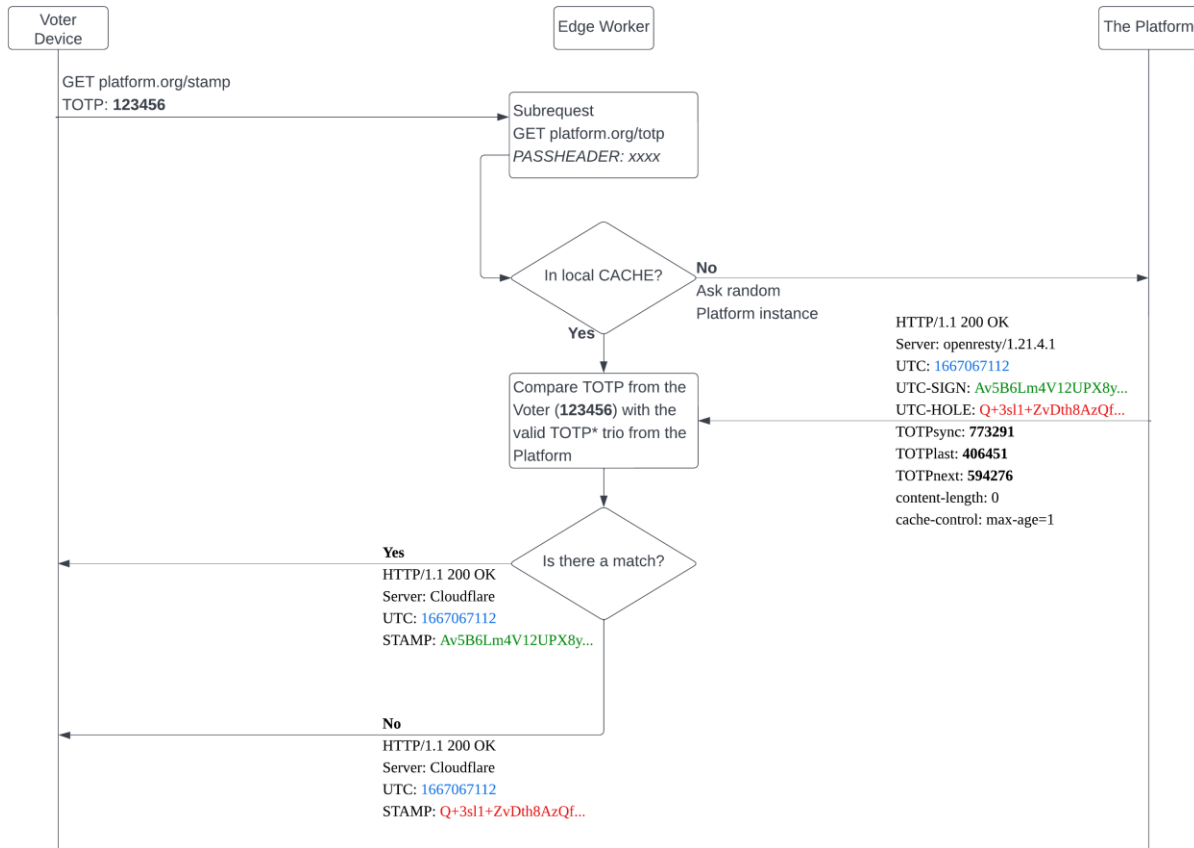
Figure 5 | Booth call-flow or "/stamp" HTTP request handling. The platform owns time-based one-time password (TOTP) that is only available at voting booths. Edge logic prevents potential attacks.

Key observations:

- Platform responses are cached in the edge for one second. Therefore, the load the platform is subjected to is limited by a number of edge machines in any one second. That's even if the edge is getting millions requests per second (RPS) in a DDoS scenario.

- Colored elements in the diagram change every second. TOTP changes every 30 seconds. Stamp requester (voter device but may be an adversary) must not be able to discriminate between a valid (green) or an invalid (red) stamps returned, so that a brute force attack lacked criteria of success.

- Stamps (UTC-SIGN, UTC-HOLE, and STAMP) are base64 encoded Ed25519 signatures of the UTC header. A public key to verify UTC-SIGN is stored as a secret in Cloudflare KV store. It needs be a secret during elections because if disclosed it would allow an adversary to tell valid and invalid stamps apart.

- *PASSHEADER* guarantees the Platform only respond to the edge – not anyone else.

- All shown transactions are encrypted: the platform uses Diffie-Hellman key exchange.

Collecting Instance (a Platform's machine)

The instance will answer three different requests:

- **/totp** corresponds to voter's **/stamp** request, where relationship is depicted above;
- **/list** returns all possible choices on the ballot;
- **/vote** casts voter's vote and returns a receipt.

Here is a **/list** example voter's software would use to identify voting options:

> GET /list HTTP/1.1
> X-Pull: Ra$q (pass-header)
>
* Processing, answer:
< HTTP/1.1 204
< Server: openresty/1.21.4.1
< Date: Thu, 24 Nov 2022 17:43:24 GMT
< Connection: keep-alive
< Say01: Cross Uranus
< Say02: Evil Neptune
< Say03: Firy Mercury
< Say04: Beauty Venus
< Say05: Legend Earth
< Say06: Warrier Mars
< Say07: Bold Jupiter
< Say08: Humble Pluto
< Say09: Shiny Saturn

< Cache-Control: max-age=3600

This campaign is themed after the solar planets. Generally, a collecting instance has campaign's configuration file containing:

- All voting options as exemplified above;
- All authoritative Ed25519 public keys to verify voting authorizations (mandates);
- Unix timestamp of the campaign's start;
- Secret Ed25519 key to sign platform UTC stamps (depicted above);
- HMAC secret to generate TOTP.

Upon getting a **/vote** request the collecting instance performs six checks (failing points); they are:

-- Bad timestamp HTTP code 485
Platform timestamp in the voting request isn't in this campaign's past.

-- Bad authority HTTP code 484
Mandate's public key in the voting request is not on the campaign's list.

-- Bad signature HTTP code 483
Mandate's signature verification fails. That is, we can't verify the voter is authorized to vote by the authority.

-- Not on ballot HTTP code 482
The "say" incoming with the request isn't on this campaign's list.

-- Already voted HTTP code 481
Mandate incoming with the request has already been seen, and associated vote was recorded earlier.

Note, other checks (Proof-of-Work, Nonce-not-recent, and stamps) are performed at the edge (Cloudflare in this PoC) – not by the platform, and we will see those checks in due course.

Vote Request as tested

The final check – Already voted HTTP code 481 – splits data supplied with the **/vote** request into two halves. One is added to a "list of votes" – that is a custom nginx log file named after campaign. Another is added to a "list of voters" – updated by a [hashtable-in-file](#) implementation called Seamagic. One of its threads sediments data from the hash-table into a log file cyclically on the best effort basis, while the other threads serve nginx. Tested with 20-bits' bucket selector (32 records-per-bucket). On a 4xCPU/4GB RAM virtual ubuntu box running on Windows 11 laptop, around 2000 votes a second were processed. Saturation of 3 million records was achieved:

```
drwxr-xr-x  2 root    root         4096 Nov 24 16:21 ./
drwxr-xr-x 11 root    root         4096 Oct 28 19:10 ../
-rw-r--r--  1 nobody  root        37050 Nov 22 17:39 access.log
-rw-r--r--  1 nobody  root        81125 Nov 24 11:43 error.log
-rw-r--r--  1 root    root            7 Nov 24 15:27 nginx.pid
-rw-r--r--  1 nobody  root    186000000 Nov 24 16:16 planet.log
----------  1 root    root   4294967297 Nov 24 16:16 plan-man.htb
-rw-r--r--  1 root    root    501000000 Nov 24 16:21 plan-man.log
----------  1 root    root            0 Nov 24 15:28 plan-man.off
-rw-r--r--  1 root    root    318558512 Nov 24 16:23 plan-man.tmp
root@ubuntu:/www/nginx/logs# wc -l plan-man.log
3000000 plan-man.log    (Voters by Seamagic)
root@ubuntu:/www/nginx/logs# wc -l planet.log
3000000 planet.log      (Votes by nginx)
```

Figure 6 | Functional and non-functional testing of a collecting instance of the Platform. Three million votes are dispatched at 2000 votes-a-second rate. Split-path handling of votes and mandates.

A **/vote** request runs like so:

> GET /vote HTTP/1.1
> X-Pull: Ra$q (pass-header)
Mandate
> auth-pkey: uBgXBIc6v49QdygE5tp79RRnnqIScyZAd9t4hkBUhjQ=
> auth-sig: QhD6qeEvSkUiYTXX84I4pgbIQp8R9sepBaoUKScIAJGpgHtn0jIsOrad2fTqxx5SP4vI02f8RIST1aUibNcdBg==
> auth-id: HgJabbZ6Zw7Nag6RAfIxTTcxC8r0uXGa
Vote
*> SAY: **Bold Jupiter***
*> Voter: **8MSD-JR3X-GF1D-GCFT-CKWP-15P0-JKY3-YTFZ***
> P-UTC: 1667950489
>

* Processing, answer:

< HTTP/1.1 204
< Server: openresty/1.21.4.1
< Date: Thu, 24 Nov 2022 20:30:20 GMT
< Connection: keep-alive
< Cache-Control: max-age=300
< Receipt01: T2YRH:YEQJ-VJQ4-AMSS-WCY7:TDDGM Cross Uranus
< Receipt02: NCFZC:NEE1-46SD-DPE0-D0S9:JHTD0 Evil Neptune
< Receipt03: 6HZ5J:P9TK-QRSN-HP8Y-KBFD:Y9MBG Firy Mercury
< Receipt04: JZVSD:H1HV-WV4D-FE59-5BYG:9GTMW Beauty Venus
< Receipt05: 5V8HB:1Z8X-689Y-ZJXQ-4J1S:DY643 Legend Earth
< Receipt06: J9G6Q:9RYN-JF7A-63VK-Q1DC:D3R94 Warrier Mars
< Receipt07: **8MSD-JR3X-GF1D-GCFT-CKWP-15P0-JKY3-YTFZ Bold Jupiter**
< Receipt08: VVK5H:HHCK-2ACD-1K5M-DJ00:2NCFS Humble Pluto
< Receipt09: CQDVJ:0T5A-T94R-HTP6-MTMC:4AHAR Shiny Saturn

Edge Function (Cloudflare)

Cloudflare is used in this PoC because it is known to mitigate the world record attacks state actors are capable of. The functionality employed can be replicated with any established CDN, such as AWS Cloudfront or Akamai.

It is the edge logic (and associated code processing every request at the edge) what securely isolates collecting instances of the Platform from common Internet of voters and adversaries. Common domain **clairvote.org** is registered for the platform. Every electoral campaign is assumed to have own *server* block in terms of nginx configuration. Accordingly, every campaign has its own hostname in terms of its REST API:

***<campaign>*.clairvote.org**

For instance, *solar.clairvote.org* is configured as a custom domain linked to a Cloudflare worker that implements the edge logic. First of all, any request not specifically configured is declined with HTTP 400 e.g.,

curl -sD- https://solar.clairvote.org/hello
HTTP/2 **400**
date: Thu, 01 Dec 2022 13:07:53 GMT
content-length: 0
server: **cloudflare**

A straightforward logic is applied to the **/list** request. The edge only abides *Cach-Control* header and caches every response accordingly. Again, caching results in guaranteed limit on request rate collecting instances handle, regardless of potential attacks on the open Internet. For *Solar* campaign:

```
curl -sD- https://solar.clairvote.org/list
HTTP/2 204
date: Thu, 01 Dec 2022 13:18:10 GMT
cache-control: max-age=3600
cf-cache-status: MISS
say01: Evil Neptune
say02: Firy Mercury
say03: Beauty Venus
say04: Legend Earth
say05: Warrier Mars
say06: Bold Jupiter
say07: Shiny Saturn
say08: Humble Pluto
say09: Cross Uranus
server: cloudflare
```

Figure 5 lays out the exact logic of a **/stamp** request:

```
curl -v -H 'TOTP: 123456' https://solar.clairvote.org/stamp
> GET /stamp HTTP/2
> Host: solar.clairvote.org
> totp: 123456
>
* Processing, answer:
< HTTP/2 204
< date: Thu, 01 Dec 2022 13:28:13 GMT
< voter: AAKQ-YAJ6-JF4A-6DMM-WEJX-5D1S-SHXX-5K8G
< p-sig: f/V+0YqqOnBz3bsYqy6SWQJlMUy2fvewxRHWDt7zVHz5ZIwg6V9LhQcBNrrJZ4fUkVsno5hukOAp9LkPIeZfAg==
< p-utc: 1669901293
< server: cloudflare
```

Response headers **P-sig** and **P-utc** together represent platform's stamp. The edge will verify the stamp is genuine in a voting request.

Voter stamp is generated by the edge as 16 bytes random + first 4 bytes of a HMAC signature of those 16 bytes. Resulting 20 bytes converted into 32-characters Crockford's scheme. The HMAC should ideally depend on the Platform's stamp and a secret owned and managed by CDN. As Cloudflare doesn't have such feature, *PASSHEADER* (Figure 5) is used as the secret.

Any valid voting request could be taken by an adversary after use and re-played back at a high rate to penetrate the edge layer and render collecting instances overloaded. To tackle this risk a nonce – that is a unique proof of work by a voter device on every voting request – must be made basis for caching. Thus, we replace **/vote** with **/<hex nonce representation>**, and the nonce can be 8-32 bytes long:

Figure 7 | Full production voting request with Proof-of-Work check disabled.

On top of the caching behavior just described, there are three failing points controlled for by the edge logic:

-- Proof-of-Work below threshold HTTP code 489

Number of leading zero bits in big-endian hash representation is lower than configured minimum. The hash function and its arguments will be described shortly.

-- Bad Platform stamp HTTP code 488

Platform's stamp could not be verified as genuine. One would end up here if the stamp was obtained with incorrect TOTP header.

-- Bad Voter ID (stamp) HTTP code 487

Voter's stamp (ID) could not be verified as genuine. Voter ID returned by a "/stamp" request contains verifiable entropy as described above. That ID must be used with the voting request.

Finally, every voting request is consistently mapped to a pair of collecting instances. Pairing ensures 1+1 redundancy in votes collection. Consistency is based on mandates: one same mandate would always be mapped to the same pair of collecting instances, in order to force only one vote having been cast. In Cloudflare-specific implementation those pairs are defined as *even* and *higher odd* in firewalled (WAF) list of hostnames e.g.,

Figure 8 | One pair of collecting instances running as mirrors in 1+1 redundancy model. The instances are only accessible through the edge layer.

PoC/API

A python script (*vote-simulator.py*) that simulates voting process and demonstrates API is in this public repo. Key variables are defined in the global scope:

- *auth_skey* is one of the secret keys an authority would use to designate a "voter" by signing their randomly unique identifier. Following UUID practice we assume 130 bits as sufficient entropy to eliminate collisions. This process would normally take place at Step 6 of "Preferred Procedure" above. Mandate given by the script is actually part signature to allow collisions between IDs signed by different secret keys (branches of the authority).

- *totp_secret* is a secret controlled by the Platform that is hidden at voting stations in reliable manner (phase 2 further on in this document). Time-based one-time password (TOTP) displayed in voting booths is generated based on this secret.

For testing purposes Cloudflare (Cf) edge is configured with two collecting instances (Platform machines). One is available, and the other is permanently down.

For **/stamp** and **/list** requests Cf picks a responding instance (origin) at random. This means that for shortly cached **/stamp** request a probability to encounter a 5xx error (origin not available) converges to 50%. On the other hand, longer cached **/list** requests return such error only on a cache-miss, which would be less frequent in test. Voter software must gracefully re-request until served.

For **/<nonce>** voting requests we implement 1+1 redundancy model, meaning that we do not expect an error to return while at least one instance of the pair is online. Voter software still must implement randomly timed re-requests in case of problems at Cf edge. Besides, returned receipt must be matched against the request to rule out virtually impossible event of colliding nonces in limited space (Cf machine) and time (cached for 5 minutes).

The work is currently defined as 25 (PoW score) leading zero bits of big-endian sha256. A concatenation sha-256 is calculated over in Figure 7's terms:

<P-UTC e.g., b'123'><P-SIG bytes><voter e.g., b'MH6'><say e.g., b'abc'><nonce bytes> where "bytes" are actual bytes decoded from Base64/hex. In tests, the PoW score was consistently achieved by a single physical CPU with 25-40 million iterations over an average 120 seconds' interval. In view of denial-of-service attacks, note, that a request cannot penetrate the Cf layer without performing this work.

Two unrelatable lists (votes and voters) can be downloaded and verified against your executed attempts. PoC instance publishes those lists (Figure 2 shows a specialized server in production implementation):

wget *https://clairvote.org/solar-vote.log* (votes)

wget *https://clairvote.org/solar-auth.log* (voters)

Figure 9 | Two lines from the list of votes. Split and annotated. Voter stamp allows end-2-end verification but not divulge voter's identity.

```
root@oceanuk:~# tail -1 solar-auth.log                    Mandate
uBgXBIc6v49QdygE5tp79RRnnqIScyZAd9t4hkBUhjQ= b27x3wjLax0/ki/6/gY38o
                ^Authority's public key^            signature                    Arbitrary ID
DaavVDd2li5yS0Brl9q1e6H+pbvIrZ5BrwN+Pf5RVtMpiD4EHpzqDeQtjMGbE9AQ== g0DRNRyRfkFIKgWIY22eGI2vwDRTcN1H
```

Figure 10 | A line from the list of voters. Split mid-signature. Voter can be identified by an ID authority chose at issuance.

The votes casted in support of a planet could be counted in a Linux terminal like so:

grep Pluto solar-vote.log |wc -l

Counting (tabulation) of votes can be done by anyone and scales well in many millions.

Phase 2. Voting Station Equipment

Voting authorizations (mandates) in steps 4-6 of the Preferred Procedure above can be generated by *mandates.py* script in our public repository. Each run of the script results in a *<base32auth_id>.png* file with QR-code and Crockford's card in *auth* folder. There is also a *~list* file containing those filenames with base64 mandate against each. Base64 better suites network protocols, while Base32 – humans, hence we use both. Base64 mandate can be looked up directly in the *voters* list as in Figure 10.

The script also spits out a token, which is a hex representation of what is recorded in the QR-code and the Crockford card. You could scan the QR-code with QRefine app on an Apple device – it would give you exactly same hex token.

An image the script produces is shown in Figure 11. It is supposed to be available from staff of a voting station or electronically if not voting in-person.

EohkV9iyqvafGVg6f46DKe

1:	Q0C1	E147	7AZR	YM3Q	N
2:	502E	DPKV	YMA6	F7N2	A
3:	29SJ	CG3Q	VDW8	CG2M	R
4:	GRT1	5234	AZCB	5AQP	D
5:	KWCN	GEKZ	HT1J	KRGC	Y
6:	Z1QS	Z1NX	KVXK	NSC8	D
7:	4Z02	3764	53ZN	C8FH	E
8:	7R3K	1PW4	JKZK	ERD4	P
9:	8200	AHPC	7ZB2	YATR	7
10:	KRN5	2S3Y	13NA	7QX8	W
11:	HMQ6	082Z	9MFA	MRPH	R
12:	VA19	166E	46GY	J3C9	H

Figure 11 | Non-fudgeable voter identification issued by an authority with two modalities of transfer (QR-scan/manual) and Base64 look-up string appeared in a published list of voters.

TOTP Generation

Time-based one-time passwords (TOTP) are displayed in voting booths/area to enforce voting action at publicly observed stations to eliminate possibility of coercion. On the other hand, TOTP generators are not controlled by the state authorities – they are controlled by the voting platform to eliminate possibility of manipulations (e.g., votes throw-in) on behalf of the state. The setup is pictured in Figure 12 in development stage:
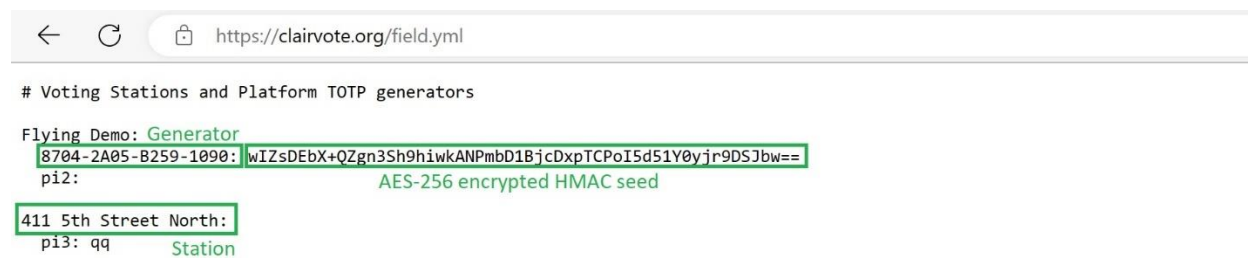
Figure 12 | Voting TOTP generator prototype (code name Flying Demo). First digit is a checksum (mod 11) for the remaining six TOTP digits. *Station*: non-changeable generator's identifier. *Updated*: when TOTP secret is updated from *clairvote.org* website.

The generator comprises temperature-compensated real time clock (RTC) that allows off-line operation for many days. The clock is [NTP](#) synchronized every time the generator gets Internet connection. Online/Offline states can easily be controlled from a mobile device by a voting station's staff.

The key component of the generator is a [secure element](#) that stores and uses involved secrets in a way they never leave the chip and, thus, cannot be extracted in a relevant time-frame. Generator's identifier ("Station" in Figure 12) and a unique wrapping secret ([RFC 3394](#)) are provisioned to the secure element prior deployment.

Deployment map is publicly available, currently as https://clairvote.org/field.yml, it is annotated in Figure 13.


```
# Voting Stations and Platform TOTP generators

Flying Demo: Generator
  8704-2A05-B259-1090: wIZsDEbX+QZgn3Sh9hiwkANPmbD1BjcDxpTCPoI5d51Y0yjr9DSJbw==
  pi2:                        AES-256 encrypted HMAC seed

411 5th Street North:
  pi3: qq        Station
```

Figure 13 | Global deployment map for voting TOTP generators. YAML file links humanly identifiable voting stations with TOTP generators deployed there (just one record in PoC). Every TOTP generator has an encrypted version of the TOTP secret.

A voting campaign has a single TOTP secret (also known as seed), so the seed is actually the same in all generators. However, as wrapping secret is unique per generator, AES-256 ciphertext of the seed is different for every generator. Seed's unwrapping happens inside the secure element, and its utilization takes place there as well. Ciphertext of TOTP secret in Figure 13 is of the cleartext provided in the voting simulator.

The purpose of this architecture is to enable election observers to verify location of each and every TOTP generator against the public deployment map, so that none is used covertly for throwing votes in. Furthermore, the generators could get seed-update before elections start on the day, so that "Updated" line (Figure 12) could guarantee the old TOTP secret isn't re-used.

Relaxation

There are circumstances, where a voter is authorized to vote outside voting station. It is best to keep such corner cases beyond this implementation to not inflate the surface of potential attacks. For instance, postal voting could be reserved for those cases. Nevertheless, if push comes to shove it is possible to handle TOTP-free votes in parallel. For example, an arbitrary ID the state issues (Figure 10) has 24 bytes of raw entropy, which is sufficient for splitting into two forms: majoritarian and TOTP-free. UTC-HOLE (Figure 5) is just another Platform's signature from a

different secret key. This "bad" signature could work together with the TOTP-free form of the voting authorization. The combination could be made passing checks at the edge (Cloudflare) and would be recorded by the collecting instances.

Audit

Checking electronic protocol can be fast and easy compared to manual paper revisions. The first link from a person to their vote sits with the authority: authorization to vote (mandate) exemplified in Figure 11 is issued by voting station's staff. As it is done electronically, it must be trivial for the authority to personally identify the voter by a voting authorization recorded on the voters list (Figure 10).

The purpose of the state-independent voting platform is to sever the link between a voter and their vote and publish two unrelatable lists: one – of the votes and the other – of the voters. Due to the lists being unrelatable the only sensible verification is comparing their sizes: number of votes must match number of voters (Figure 6). End-2-end verifiability i.e., every voter being able to check that their vote on the list is as casted, must take care of the rest, provided the ballot is secret. It is technically straightforward, nevertheless, for the platform to maintain a third – not public – list containing all vote-voter links. Although it would defeat the whole idea behind this architecture, it could play a role in trials, where secrecy can be sacrificed for full traceability in order to build the confidence and adopt this system as architected in the future.

Preferred method of audit, on top of the possibilities mentioned above, is a differential analysis of patterns the votes are accumulated by. The platform (Figure 2) is capable to aggregate data and generate contemporaneous snapshot of resulting lists every couple of minutes. That is, we would have hundreds of vote/voter list pairs by the end of any elections. Not only could we ensure that every pair has matching sizes (as alluded to above), we would also verify that the number of votes for each candidate is progressively growing – from previous pair to the next, and that the growth is reasonably smooth – without abrupt jumps. Such analysis controls for artificial addition/removal of votes because aggregates from a plethora of uncoordinated actors naturally behave smooth.

Minimum Viable Product (MVP)

Hardware for polling stations have now been productionized with custom software and a [PCB](#) tailored for [RPi microcomputers](#). Following Figure 12…
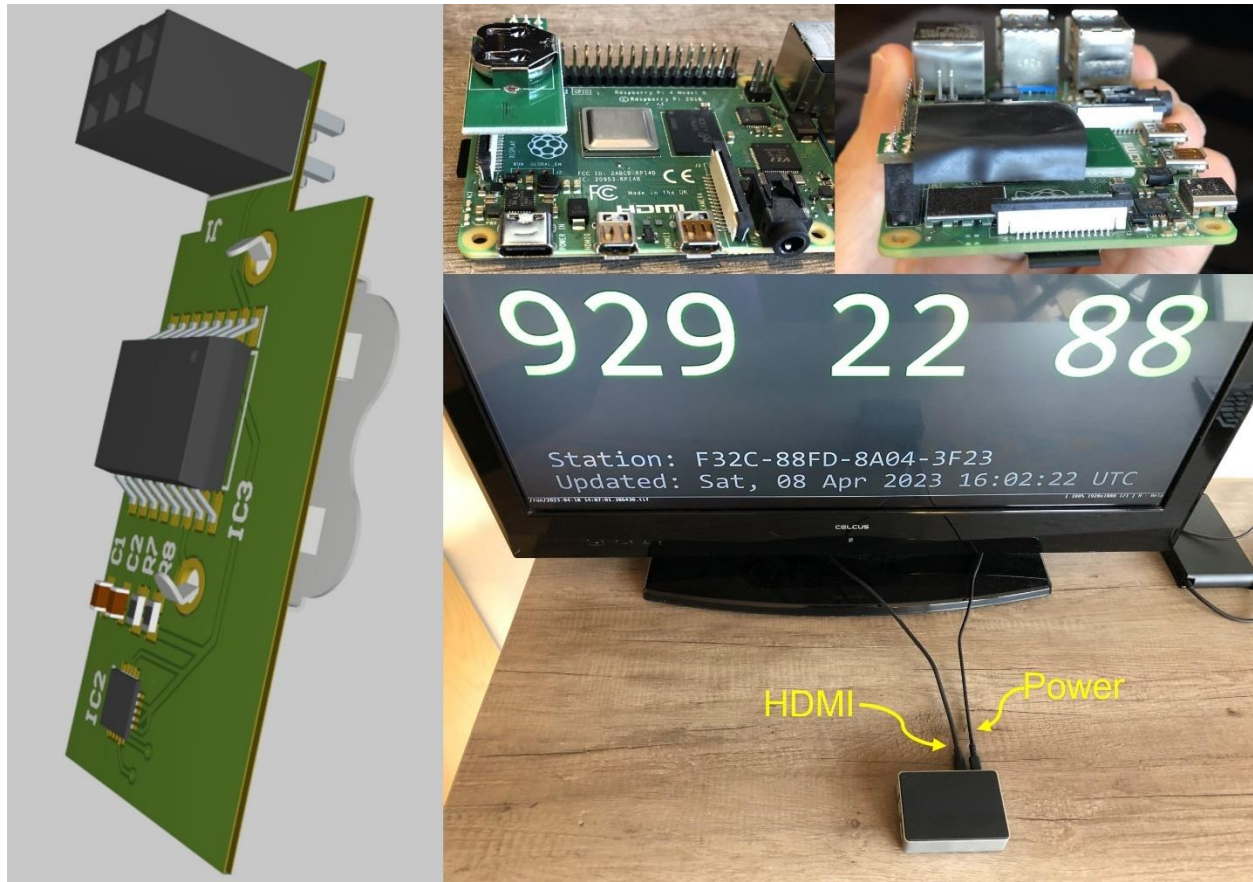

Figure 14 | Voting TOTP generator (MVP). High accuracy clock, its lithium battery, and NXP Secure Element – all reside on a custom-built PCB.

TOTP generator is mimicked by a [web page](#) for demo purposes. The web page produces both credentials to vote in *Solar* campaign. Inspectable java script, similarly to the python simulator, generates those credentials.

Anyone can vote in-browser or from corresponding [progressive web app (PWA)](#) at https://clairvote.org/go.html. Voting process can render dated equipment slow/irresponsive with CPU utilization climbing to 100% by design – we create a corpus of computational work, where

cumulative work by all voters serves as deterrent from potential meddling. The oldest browsers supported are as follows.

Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android	Deno	Node.js
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
98	98	94	84	15.4	98	94	68	15.4	18.0	98	1.14	17.0.0

Figure 15 | Compatibility table for browsers and java script environments.

Finally, a publishing server (Figure 2) can keep analyses/results near-real-time. In this MVP, however, you can verify results yourself at https://clairvote.org/check.html...
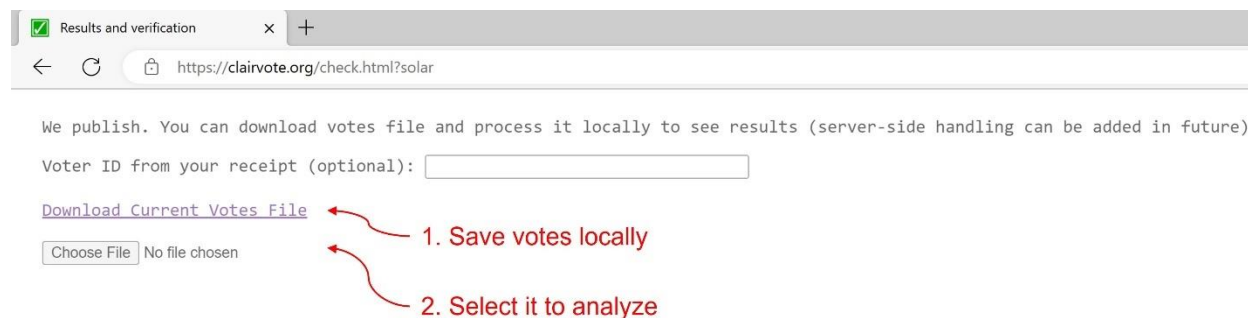

Figure 16 | Local analysis of campaign results and verification that a vote is recorded as casted.

Integral count of total number of votes is graphed with granularity of 24 hours. In a real campaign the granularity is likely to shrink to a minute, and the graph will be built individually for every option on the ballot sheet.

#