Coercion-resistant i-voting with short PIN and OAuth 2.0

Matteo Bitussi[1], Riccardo Longo[1], Francesco Antonio Marino[2], Umberto Morelli[1], Amir Sharif[1], Chiara Spadafora[3], and Alessandro Tomasi[1]

[1]Center for Cybersecurity, Fondazione Bruno Kessler, Trento, Italy
mbitussi,rlongo, umorelli, asharif,altomasi@fbk.eu
[2]Italian Government Printing Office and Mint, IPZS, Rome, Italy
fa.marino@ipzs.it
[3]Università degli Studi di Trento, Trento, Italy
chiara.spadafora@unitn.it

Abstract

This paper presents an architecture for an OAuth 2.0-based i-voting solution using a mobile native client in a variant of the Araújo-Traoré protocol. We follow a systematic approach by identifying relevant OAuth 2.0 specifications and best practices. Having defined our framework, we identify threats applicable to our proposed methodology and detail how our design mitigates them to provide a safer i-voting process.

1 Introduction and Related Work

Electronic voting (e-voting) is becoming increasingly appealing to improve election efficiency and accuracy, and to make elections more accessible, inclusive, and transparent. As technology progresses, end-to-end verifiable voting protocols may also offer an enhanced level of trust. Several interesting proposals have been made in recent years, with particularly noteworthy examples including Helios [1], Belenios [7], and Civitas [6]. These academic proposals are of great interest, but it is non-trivial to make the leap to concrete solutions.

In this paper, we make a proposal for how to use the OAuth 2.0 framework [15] - the current standard for authorization at the application layer - for a concrete instance of a specific i-voting protocol, taking into account best current practice while attempting to ensure that the complex requirements of e-voting are still being met. The application of OAuth on its own to a concrete scenario requires knowledge of several specifications extending the core protocol, and requires effort and expertise [29].

This paper extends the protocol proposed in [23], derived from ABRTY [3, 2] with the following main contributions: (*i*) we give an architecture for an OAuth 2.0-based solution that supports i-voting through the identification of relevant OAuth 2.0 specifications and best current practices to provide a secure protocol and satisfy i-voting requirements; (*ii*) we introduce commitment access token to authorize the casting of a ballot, to prevent brute-forcing of the PIN, maintaining unlinkability between vote and voter; (*iii*) we integrate short checks for Cast-As-Intended ballot verification, following [8]; and (*iv*) we identify threats applicable to our architecture, briefly explaining how our design can mitigate them.

The i-voting protocol satisfies coercion resistance through the *fake credential* mechanism as introduced in [20]; we call it Anti-Coercion Credentials (ACCs) as in [23]. Each voter is provided with a different ACC, that will be used to validate their ballots, and can autonomously create a ruse ACC, indistinguishable from the valid one. Ballots cast with a ruse ACC do not count towards the final tally, but are indistinguishable from valid ballots, enabling anti-coercion strategies. Differently from Civitas and ABRTY [6, 3, 2], the assumptions of [23] avoid an untappable channel and try to provide a safe ACC delivery and forging in an uncontrolled environment (for further discussion see [23] and Section 4.2 on generation, issuance and forgery of ACCs; Section 5.1 on coercion and untappable channels). Moreover, the ACC is stored on the voting device masked by a PIN: to forge an ACC the voter can set up a ruse PIN. The use of a PIN to unlock voting credentials was proposed by Neumann and Volkamer [24] to improve usability, however, this mechanism is susceptible to various problems [11]. Enabling voters to set their choice of PIN remotely would enable a coercion strategy; a compromise proposed in [23] to evade coercers is to deliver the PIN at a random time, but the authors "leave considerations on PIN length and brute force countermeasures as implementation choices".

In Section 2 we provide background on OAuth; in Section 3 we describe our proposal for using commitment schemes in OAuth Access Tokens; in Section 4 we summarize the i-voting protocol with a focus on the voter's interactions, and we describe our proposal for a framework to instantiate the protocol using OAuth; in Section 5 we follow a threat modeling process limited to client-server exchanges and coercers, summarizing how our proposed mitigations preserve the e-voting requirements and coercion resistance of the base protocol.

2 Background: OAuth 2.0

The OAuth 2.0 authorization framework [15] enables a third-party application (Client) to obtain limited access to an online service hosted on a Resource Server (RS) on behalf of a Resource Owner (RO). The Client redirects the RO through the User Agent (UA) into the Authorization Server (AS), where the RO performs the authentication. After the successful authentication of the RO, the AS issues an Access Token which Clients use to access the RO's resources in the RS.

A typical Access Token may be a JSON Web Token (JWT) [18]. Unless otherwise specified, Access Tokens are bearer tokens, thus vulnerable to a range of attacks. OAuth Security best current practice recommendations [22] include:

Replay prevention: AS should constrain tokens to their intended sender, typically by binding a token to a public key held by the sender and forcing the sender to prove its possession by digital signature. Two methods for sender-constrained Access Tokens are mutual TLS (mTLS) [5] and Demonstrating Proof-of-Possession (DPoP), which is currently an Internet-Draft [12].

Client authentication: should be based on public key cryptography that enables stronger client authentication such as mTLS or signed JWT [18].

Authorization code protection: to prevent authorization code theft or injection, Proof Key for Code Exchange [27] (PKCE) is recommended.

Privilege restriction: Access Tokens should restrict privileges granted to the holder to the minimum required, in accordance with the principle of least privilege, e.g., making use of the audience, scope, and resource claims.

2.1 OAuth 2.0 Token Exchange

The OAuth 2.0 Token Exchange protocol, as defined in RFC8693 [19], is an extension to the OAuth 2.0 protocol for implementing scenarios where one token needs to be swapped for another. There are scenarios where the Client needs to access resources hosted by other downstream Resource Servers on behalf of the User and in which the usage of the direct Access Token is not possible.

RFC8693 [19] introduces a new grant type to define how a Client can request and obtain security tokens from AS that are playing the role of Security Token Service (STS) in the Token Exchange protocol. The STS validates security tokens (e.g., JWT) provided to it and issues a new security token in response, which the Client may use as access credentials for resources in heterogeneous environments or across security domains.

2.2 OAuth 2.0 Dynamic Client Registration

OAuth 2.0 Dynamic Client Registration [26] (DCR) is an extension to OAuth 2.0 that enables a Client to obtain unique dynamic client identifiers (client id) for each instance of Client. Moreover, the Client is able to register a public key with the AS that can be used for the Client authentication based on signed JWT.

RFC7591 [26] introduces a new endpoint called Dynamic Client Registration endpoint, which can be implemented by the AS as either Protected or Open - i.e., requiring an initial Access Token or not, respectively.

3 OAuth Commitment Access Token

Our proposal is based on cryptographic commitment schemes, which we provide brief background for in Section 3.1. Our proposal is described in Section 3.2.

Commitment schemes have also been used elsewhere in OAuth extensions, including one we use; we compare our proposal to significant others in Section 3.3.

3.1 Commitment schemes

A commitment scheme to a message m lets a Prover generate a commitment c and an opening o, such that c may be disclosed or published in advance, and any Verifier provided with the opening o at a later time can verify the correctness of the commitment. Commitments are *binding* if they can only be opened to a single message, and *hiding* if the commitment reveals no information about the message itself - see [4] for more details.

A simple commitment scheme to a message m based on a cryptographic hash function H may be instantiated as follows ($||$ denotes concatenation):

$\texttt{Commit}(m)$: $C(m) = (c, r)$ with the opening $o = r \xleftarrow{R} \{0,1\}^\lambda, c \leftarrow H(m||r)$.

$\texttt{Open}(c, m, r)$: $c \stackrel{?}{=} H(m||r)$.

This scheme is computationally binding and hiding if H is collision resistant and the distribution of digests is computationally indistinguishable from uniformly random output, respectively [4]. The former depends on the hashing algorithm; for the latter to be the case, the input must have sufficiently high entropy [31].

3.2 Commitment Access Token

We propose an authorization flow for a scenario in which an OAuth Client has a message b and needs an Access Token for an OAuth Client linked to b, but wants to preserve anonymityin the following sense: the AS should not learn b, and the Resource Server should not link b to the identity of the Client.

Our proposal is the following:

- to send the message b to an RS, the Client generates a random $r \xleftarrow{R} \{0,1\}^\lambda$ and computes a commitment $c \leftarrow \texttt{Commit}(b, r) = H(b||r)$ to b (steps 1-2);
- the AS authenticates the Client and provides an Access Token - a JSON Web Signature (`JWS`) with c as payload - to the Client (steps 3-4);
- the Client requests the RS to process the message b by providing the `JWS` and the commitment opening r (step 5);
- the Resource Server accepts the message b if both the `JWS` and the commitment opening are valid (steps 6-7).

The flow is shown in Figure 1, with the hash-based commitment scheme as in Section 3.1.

Trust model. In general, a Commitment Access Token (`CAT`) could be useful when (a) the Resource Owner wishes to perform a single, high-value

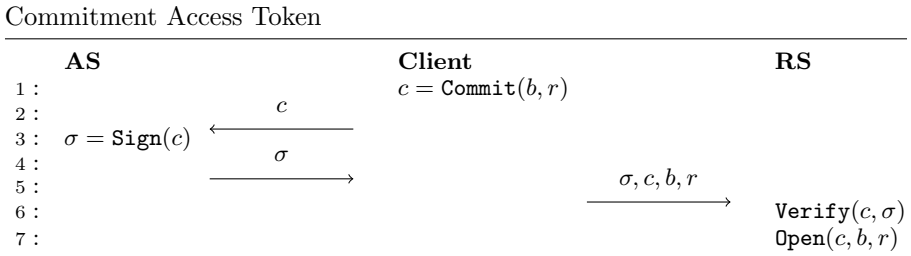

Figure 1: Commitment Access Token (`CAT`). σ represents the `JWS` digital signature, with `Sign` and `Verify` determined by the chosen JOSE algorithm [17].

operation; (b) the Authorization Server must not know the exact content of that operation, but must ensure the right audience and scope, and wishes to ensure that the rate of requests within that scope from an RO is limited without delegating this access control to the Client; (c) the Resource Server must ensure that the RO was authenticated when that specific access was authorized.

Replay protection. We allow the Client to generate the nonce r opening the commitment. This means that multiple uses of the service with the same token and message are possible. This is not an issue in our case as duplicate ballots are weeded out in a following step (see Section 4); other applications may mitigate against replay by adding further validation conditions on the token.

Holder binding. In some scenarios, there is no need to ensure holder binding to the Access Token because the commitment is binding to the payload. As long as the commitment can only be used with a specific message and the AS authorized the access, it may not matter which Client is actually accessing the RS. This decoupling helps to preserve the anonymity of the Client, and this property may be valuable for sensitive operations (e.g. ballot casting).

3.3 Comparison

A commitment scheme is used in the widely recommended OAuth extension Proof Key for Code Exchange (PKCE) [27] to mitigate against code interception or injection.

A commitment scheme is also used in the proposed OpenID Connect (OIDC) extension OpenPubKey [16] to bind an OIDC identity to a specific public key, without the AS knowing the key that will be used by the Client.

A commitment-like scheme is also used in the Goolge Play Integrity API [13] (GPIA) to provide replay protection and integrity for a high-value message.

We summarize PKCE and GPIA below, and compare the proposed CAT with PKCE and GPIA in Table 1. We omit a comparison with OpenPubKey since it is, as far as we know, at an early proposal stage and we do not make use of it.

PKCE. Two separate OAuth requests are made by a public Client to an AS: (a) an Authorization Request, to which the AS replies with an Authorization

Code (AC); and (b) an Authorization Grant, to which the AS replies with an Access Token. In exchange (a), messages are redirected via the User Agent so that the user (Resource Owner) may authenticate to the AS and grant consent; exchange (b) occurs directly between Client and AS, e.g., via TLS. Messages in exchange (a) are at higher risk of interception or disclosure.

To mitigate against AC interception using PKCE, the Client sends a commitment c to a random r to the AS in (a), and at a later time opens the commitment by revealing r in (b). Eavesdroppers intercepting the AC cannot craft a valid Authorization Grant request in (b), as they do not know the secret r.

GPIA. GPIA may be used to provide a mobile application's back-end server (Verifier) with signed and encrypted attestations about the integrity of the app (Client) and the device it runs on. Additionally, it may be used to provide replay protection and integrity for a high-value message m generated by the app, by use of a Verifier-provided nonce and a hash function, respectively. The word *commitment* is not used in the documentation, but we would argue that the protection is commitment-based, and indeed it follows a very similar flow to Figure 1, with notable differences. Firstly, the GPIA nonce r is set by the Verifier to provide replay protection, but is revealed to the GPIA attestation server. This enables GPIA to open the commitment if the message m is ever disclosed. Secondly, the GPIA commitment is computed as $c \leftarrow H(m)||r$. We observe that this commitment is binding, but not hiding, and indeed there are very clear warnings to that effect in the documentation. Which commitment to use is up to the developer: even m itself would be accepted in practice. This makes sense in the context of GPIA, which is not designed for hiding commitments, but is not sufficient in our case.

Notably, GPIA advise that requests are rate limited, and developers should be prepared to handle the absence of that service. In our application, rate limiting by the AS is a desirable feature to protect against brute force attacks.

Table 1: Comparison of PKCE, GPIA, and CAT.

Scheme	PKCE	GPIA	CAT
Commitment	$H(r)$	$H(m)\|\|r$	$H(m\|\|r)$
Nonce generated by	Client	AS	Client
Nonce revealed to AS	Yes	Yes	No
AS role	Authorization	Attestation	Authorization
RP role	Authorization	Authorization	Resource
Main mitigation	Code interception	Replay, tampering	Brute force
Hiding commitment	Yes	No	Yes

4 Our framework: an i-Voting Protocol

Here we present the i-voting protocol, introducing the entities involved and specifying their OAuth roles. For background on OAuth, see Section 2. The protocol achieves end-to-end verifiability and ballot secrecy by encryption with a threshold modified ElGamal scheme [10], zero knowledge proofs of ballot correctness, and verifiable shuffling and re-encryption [14]. We give a high-level overview of the steps from App installation to vote casting, and we also describe an adaptation of cast-as-intended verifiability following CDGT [8]. In Section 4.4 we describe the use of our proposed `CAT` during ballot casting, then in Section 4.5 we summarize the OAuth-based access control in the Client-server exchanges.

4.1 Protocol participants

Electoral Roll (ER). The ER matches authenticated voters with their right to vote. It acts as OAuth AS to i-voting RSs, regulating the distribution of both anti-coercion voting credentials (ACC, see [23]) and casting tokens. It also acts as OpenID Relying Party (RP) to the OpenID [28] Provider (OP).

eID Provider (OP). The OP is an OpenID [28] Provider, on which the ER relies on the OP to authenticate voters to properly assess their eligibility.

Voting Application (App). The App is used by voters to receive voting credentials and cast ballots. It also enables anti-coercion strategies exploiting the ACC [23]. The App is a native application, and each instance registers as a stand-alone confidential OAuth Client with the ER.

Notification Server (NS). The NS notifies Clients when resources are available to download. It acts as OAuth Resource Server.

Registration Teller (RT). The RTs issue ACCs to Apps and are involved in some anti-coercion strategies [23]. They act as OAuth Resource Server.

Ballot Box (BB). The BBs receive and collect cast ballots, sent anonymously by Client applications. They act as OAuth Resource Server.

Outside our focus on the interactions between the App and the other entities, BBs are also responsible for checking the admissibility of ballots, publishing receipts of successful casting and providing all the collected ballots for tallying. Other participants include: Tabulation Tellers (TT), which tally cast ballots but are not directly accessed by the Client; and a Web Bulletin Board (WBB), which provides a public registry to all participants, but does not require authentication or authorization by the Client.

4.2 Voting Protocol Overview

The protocol can be summarized in the following high-level steps: (i) Client and Voter registration, (ii) PIN and proof of correctness retrieval, (iii) ruse PIN request, and (iv) ballot casting and individual verification.

4.2.1 Client and Voter registration

A voter $\mathcal{V}$ starts with downloading the App from an official application store and installing it on their device. Once the App is installed, the voter can set it up and register to the election system:

1. $\mathcal{V}$ opens the App which checks device integrity (app origin, device not rooted);
2. the App fetches from WBB the election parameters - including the blank ballot structure and public keys - and the identities of ER, RTs, BBs, NS;
3. the App registers as Client to the ER, including an *ER Authentication* public key for future authentications.
4. the App directs $\mathcal{V}$ to authenticate to the ER via the OP; having verified the eligibility of $\mathcal{V}$, the ER issues a *Registration* token to the App;
5. the App generates its designated-verifier keys, and exchanges the Registration token via token exchange [19] to obtain the following tokens:
   - an *PIN request* token for each RT (containing the designated public key), this type of token is used by the App to register, manage their ACC, set up a new device;
   - a *Notification Registration* token for the NS;
6. the App registers with the NS and the RTs with these tokens;
7. each RT_i checks the validity of the authorization token sent by the App and issues its masked ACC share (see [23] for more details) to the App;
8. the App can interpolate the shares and compute the masked ACC, which is stored encrypted on the device.

4.2.2 PIN and proof of correctness retrieval

Before being able to vote, $\mathcal{V}$ has to wait to receive the mask that will unlock the ACC and from which the PIN is derived. The procedure for the PIN delivery is as follows:

1. after a random time, each RT_i tells the NS that its mask share is ready.
2. once the NS has received confirmation that enough shares are ready, it notifies the App that the mask is ready to be retrieved.
3. the App alerts $\mathcal{V}$ that the PIN is ready when the voter opens the App, they authenticate again to the ER via the OP;
4. after authentication, the ER issues three authorization tokens for the App:
   - a new *PIN request* token for each RT;

- a *PIN retrieval* token for each RT: this type of token will be used by the App to retrieve the mask shares from the RTs;
- a *proof of correctness retrieval* token for each RT: this type of token will be used by the App to retrieve the Designated-Verifier Non-Interactive Zero-Knowledge Proof (DVNIZKP) shares from the RTs (see below);

5. the App requests the mask shares to the RTs with the *PIN retrieval* tokens, each RT_i sends its share if it is ready and the token is valid;

6. with enough shares, the App can compute the mask, whose least significant digits are the PIN.

The PIN is shown to the voter, while the other part of the mask is saved and encrypted on the device. In this way, the complete ACC cannot be retrieved without entering the correct PIN.

The correctness of the ACC can be checked with the DVNIZKP, which is issued by the RTs after a random time, just like the mask. This cryptographic proof has the interesting property that it can be forged in order to feign the validity of any ACC by the holder of the designated private key.

4.2.3 Ruse PIN request

Using the App, $\mathcal{V}$ can set up a ruse PIN and forge a DVNIZKP to convince a coercer that this ruse PIN is a valid PIN.

The procedure to set up a *ruse PIN* is the following:

1. using the App, $\mathcal{V}$ requests to set a ruse PIN and types in a PIN of their choice with the same length as the valid one;

2. the App computes a forged mask and its shares so that their interpolation gives a mask identical to the original one except for the least significant digits, which are the inserted ruse PIN instead;

3. furthermore, the App computes a forged DVNIZKP and its shares so that it validates the ACC completed with the inserted ruse PIN;

4. using the *PIN request* tokens provided by ER, the app sends to each RT_i a request for the re-sending of the PIN containing the forged shares;

5. each RT_i checks the validity of the authorization token sent by the App and after a random time sends (just like for the PIN delivery) the received forged shares back: first the forged mask share, then the forged DVNIZKP share.

Note that the setup of a ruse PIN does not undermine the ability of the valid PIN to cast a valid vote, since this setup is only intended to fool a coercer.

With the *PIN request* token, the App can also request a reminder of the PIN in case $\mathcal{V}$ forgets it. To do so the procedure is similar to the ruse PIN request procedure, but the App sends empty shares instead of the forged ones and the RTs respond with the original shares instead of the received ones.

4.2.4 Ballot casting and individual verification

When the voting period starts, $\mathcal{V}$ can cast a vote with the following procedure:

1. using the App, $\mathcal{V}$ accesses the voting view where the ballot is shown, and they can select a list and a corresponding candidate or leave a blank choice;
2. $\mathcal{V}$ inserts a PIN: the valid one will produce a valid vote, while any other PIN (including the ruse one) will produce a vote that will not be counted;
3. the App encrypts the choices and computes a set of NIZKP to prove the formal correctness of the encrypted ballot[1];
4. once the ballot has been computed, the App authenticates $\mathcal{V}$ to unlock the *ER Authentication* secret key from the keystore;
5. the App uses this key to authenticate a request to the ER for the issuing of a Commitment Access Token where the message is the encrypted ballot;
6. the ER verifies the validity of the request using the *ER Authentication* public key of $\mathcal{V}$, and if the policies on vote casting are satisfied (e.g. $\mathcal{V}$ has not voted too recently or too frequently), the ER issues to the App for each BB a `CAT` (with limited time validity) tied to the ballot;
7. the app sends anonymously to each BB the ballot, authorized by the `CAT`;
8. each BB checks the authorization and the formal correctness of the received ballot; if these checks pass, the BB sends the ballot digest to the WBB, effectively time-stamping the casting;
9. once the App can confirm the digest publication on the WBB, it prompts $\mathcal{V}$ to proceed with the vote confirmation by choosing two values to disclose for *Cast-As-Intended Verifiability* (see Section 4.3);
10. the values chosen by $\mathcal{V}$ are sent by the App to the BBs to complete the disclosure, authenticating this message with a preimage of the ballot digest;
11. each BB uses the preimage to confirm that the sender knows the ballot, then partially decrypts it with the disclosure values, sending the result to WBB[2];

[1] Note that a vote that will not be counted due to the use of an invalid PIN will still pass all formal correctness checks, since these do not depend on the PIN used.

[2] Note that only the App, that has computed the encrypted ballot, is able to produce with non-negligible probability a valid disclosure, so the disclosure message is properly authenticated. Note also that a replay would have no effect because the BB would just compute and publish the same result.

12. once the App can confirm the publication of the correct data on the WBB, it displays to $\mathcal{V}$ a confirmation message and encourages to manually verify the correctness of the sent vote, i.e., that the ballot was recorded as cast and that the ACC used to construct the ballot belonged to $\mathcal{V}$.

Once the voting period has ended, the BBs publish all the confirmed encrypted ballots on the WBB. The cryptographic protocol allows to filter out ballots cast using the same ACC (only the latest is kept) or invalid PINs, with a procedure that involves the TTs and the RTs and which does not undermine voters' privacy by using a verifiable mixnet [14]. The remaining ballots are homomorphically aggregated and threshold-decrypted by the TTs to compute the final tally, which is published on the WBB alongside a set of NIZKPs that may be used by anyone to check the correctness of the tallying steps.

4.3 CDGT Cast-As-Intended Verifiability

Inspired by [8], the protocol includes checks for the Cast-As-Intended property, in which each vote is partially audited before submission, without breaking secrecy.

In [8] the checks are designed for a Helios/Belenios ballot where the preference expressed is the number of the chosen candidate. In the protocol proposed in [23], the voter can select at most one list and at most one candidate from the ones associated to the chosen list. To enable efficient checks via NIZKPs on the correctness of the vote and homomorphic tallying, the vote in our encrypted ballot is encoded as a binary choice for each list ($l_i \in \{0,1\}$) and candidate ($c_{i,j} \in \{0,1\}$). To convert these choices in two numbers that represent, respectively, the index of the chosen list and candidate, with 0 representing a blank choice, we use a ranked sum:

$$\ell = \sum_{i=0}^{n_\ell} i \cdot l_i, \qquad c = \sum_{i=0}^{n_\ell} \sum_{j=0}^{n_{c,i}} j \cdot c_{i,j},$$

where n_ℓ is the number of lists and $n_{c,i}$ is the number of candidates associated with the i-th list.

To ease the checks, enhancing user experience, we mask these values with two random shifts $s_\ell, s_c \in \{0, \ldots, 99\}$. The ballot includes the encryptions of:

$$s_\ell, \quad (s_\ell + \ell) \mod 100, \qquad s_c, \quad (s_c + c) \mod 100.$$

The App shows the voter $\mathcal{V}$ these four values, unencrypted; to check their correctness, $\mathcal{V}$ has to compute the sum of two two-digit numbers, and then consider only the last two digits. Since the number of lists and candidates is ~ 10, this sum is likely a two-digit number, so the modulo operation is not noticeable.

To confirm the vote, $\mathcal{V}$ has to choose whether to disclose the shift or the sum for both list and candidate. To disclose the two selected values after the ballot has been cast, the App sends $\mathcal{V}$'s choices to the BBs, together with the randomness that had been used to compute their encryption in the ballot.

Note that, as in [8], the disclosure of these values does not reveal anything about the vote cast; any voter auditing their ballot is guaranteed that the ballot encodes their intended vote, even if their voting device and the BBs collude.

4.4 CAT for casting authorization

During ballot casting steps 6 and 7 (Section 4.2) we use a `CAT` to authorize the casting of a ballot, to prevent brute-forcing of the PIN while maintaining unlinkability between the unsealed vote and the voter.

In this context the Client is the voting application, the Authorization Server is the Electoral Roll, and the Resource Server is the Ballot Box.

In the very specific scenario of i-voting, there is no need to ensure holder binding to the Access Token because the commitment is binding to the payload. In other words, as long as the commitment can only be used to cast one specific ballot and the AS authorized the casting of that ballot, it does not matter which Client is casting. The ballot itself is created using voting credentials, the validity of which is established when votes are tallied.

4.5 Client Resource Access Control

Here we summarize how the App interacts with the other protocol entities, and how these exchanges are protected.

Dynamic Client Registration (Client-ER)$^{\mathbf{I}}$: The App registers as OAuth Client with the ER, following [26], including a public key (JWK) for authentication to the ER and DPoP.

Notification Registration (Client-NS)$^{\mathbf{D}}$: The App registers with the NS.

Masked ACC retrieval (Client-RT)$^{\mathbf{DX}}$: The App obtains an Access Token from the ER and uses Token Exchange to obtain tokens with individual RTs as the audience, scoped to ACC retrieval.

PIN retrieval (Client-RT)$^{\mathbf{DITX}}$: The mask shares are fetched from the RTs.

Proof of correctness retrieval (Client-RT)$^{\mathbf{DTX}}$: The shares of DVNIZKP for the correctness of the ACCs are fetched from the RTs.

Ballot casting (Client-BB)$^{\mathbf{CDX}}$: The App requests a Commitment Access Token (`CAT`) from the ER, and casts ballots to BBs. Honest BBs must reject ballots without valid `CAT`. The ER may limit the frequency of issued `CAT` to mitigate against brute force attacks.

Requests are protected as follows:

$^{\mathbf{C}}$ Commitment Access Token, described in Section 3.2, required to access BBs.

$^{\mathbf{D}}$ Demonstrating Proof-of-Possession (DPoP) [12].

$^{\mathbf{I}}$The voter leverages their national electronic ID system to authenticate to the ER, establishing the right to vote. This is outside the scope of this paper.

$^{\mathbf{T}}$ At a random time, subsequent to the previous step, and following notification of availability by the NS.

[x] Token Exchange [19] used to reach multiple RS, i.e. RTs and BBs.

5 Threat model

We address the following parts of an OWASP threat modeling process [25]: the main security and functionality properties and requirements (Section 5.2), assumptions on the use case scenario and on the infrastructure (Section 5.3), attackers and their capabilities (Section 5.4), and threat mitigation enabled by the proposed solution, and how it attempts to satisfy the stated requirements and properties (Section 5.5). However, in order to better explain and justify the assumptions and attackers considered in the analysis, we first introduce the context for which the protocol has been designed in Section 5.1.

5.1 Context and application scenario

Our threat model starts with the assumption that an untappable channel is not available to voters. The i-voting protocol in [23] is developed specifically for the scenario of national elections in which citizens abroad have the right to vote, but the current paper-based postal voting system has known flaws.

In this context, voters are typically dispersed over a broad area spanning multiple countries, even considering only one constituency. This situation makes it impractical to provide enough secure voting booths so that every voter is reasonably close to one. Therefore, in order to increase voter turnout, the proposed protocol avoids to request physical presence of voters in a secure environment by making some assumptions (see Section 5.3).

This somewhat softens coercion resistance because some kinds of attacks (e.g. persistent surveillance malware on the voting device) cannot be thwarted without an untappable channel (in practice, a secure environment).

5.2 Requirements and Properties

Recommendations on standards for e-voting systems have been specified by the Council of Europe (CoE) [9]. Requirements are enumerated under four suffrage principles - universality, equality, freedom, and secrecy - and other systemic requirements - regulatory, transparency, accountability, reliability, and security. In particular, we highlight the following requirements as relevant to our proposal:

Universality: **CoE 1** The voter interface of an e-voting system shall be easy to understand and use by all voters.

Equality: **CoE 7** Unique identification of voters in a way that they can unmistakably be distinguished from other persons shall be ensured.

CoE 8 The e-voting system shall only grant a user access after authenticating them as a person with the right to vote.

Freedom: **CoE 10** The voter's intention shall not be affected by the voting system, or by any undue influence.

CoE 15 The voter shall be able to verify that their intention is accurately represented in the vote and that the sealed vote has entered the electronic ballot box without being altered. Any undue influence that has modified the vote shall be detectable.

Secrecy: **CoE 26** The e-voting process, in particular the counting stage, shall be organised in such a way that it is not possible to reconstruct a link between the unsealed vote and the voter. Votes are, and remain, anonymous.

At a technical level, the protocol (Section 4) aims to guarantee correctness and coercion resistance [20], fairness [9], accountability [21], ballot secrecy and end-to-end verifiability [32], and eligibility verifiability [7, 30].

5.3 Assumptions

An untappable channel is typically required to issue the real ACC, so that a coercer cannot intercept it. The suggested implementation for this channel is typically the physical presence of the voter, which is difficult to reconcile with voters residing abroad. Given its application context (see Section 5.1), the ACC proposal in [23] substitutes this requirement with the following assumptions:

A1: Surveillance gaps. The coercer cannot continuously oversee the voter, i.e., there are wide surveillance gaps in which the voter is free to act.

A2: A threshold of honest RS. At least one trusted RT does not collude with the coercer. More precisely, at least one trusted RT is required to construct an ACC, and at least one trusted RT is required to be available to respond to ruse PIN requests without informing a colluding coercer. Moreover a threshold of honest TTs assure that TTs do not collude to improperly decrypt ballots. The exact number of trusted RTs and TTs required depends on security vs availability trade-offs in the setup of the Secure MPC Protocols.

Note that malware could maintain continuous surveillance on a single device. See Section 5.5 for further details and mitigations.

Further assumptions, e.g., on trustworthiness of servers, the level of assurance of eID authentication, or the robustness of low-level cryptographic implementations, are inherited from the base protocol and beyond the scope of this paper.

5.4 Threats

The main attacker of concern to our proposal is the *Coercer* in the sense of [20]:

Coercer: an adversary attempting to unduly influence the result of the election.

Objectives: forcing voters to vote in a specific way and prove it (forced disclosure), or to not vote at all (forced abstention).

Capabilities: observing the voter, and requesting recordings of voter actions.

Threats: coercers mainly threaten vote freedom (CoE Principle III).

Vulnerabilities: exploit vulnerabilities to vote secrecy (CoE Principle IV).

Concretely, any i-voting solution in which votes are not cast in a secure environment is exposed to attacks such as direct oversight by the coercer in presence or by recording. The latter is enabled by the ability of the voter to record oneself while casting, or during any other stage of the voting process, such as when receiving a PIN or using the application. The unsupervised environment and the use of personal devices also increase risk of exposure to malware, which could be a more effective means of surveillance than a physically present coercer.

5.5 Mitigations

The protocol (Section 4) aims to satisfy voting requirements (Section 5.2) by ballot casting with ACC (possession factor) and PIN (knowledge factor). To mitigate against undue influence (freedom requirement CoE 10), the main anti-coercion mitigation is the ruse PIN, indistinguishable from the real PIN.

Coercion. Several coercion evasion strategies are enabled, including (i) claiming not to have received a PIN, (ii) verifying a ruse PIN with a forged DVNIZKP, (iii) casting an invalid vote by intentionally typing a ruse PIN, (iv) casting more than one ballot, and (v) using multiple devices.

Strategies (i) to (v) are effective against a physically present coercer; strategy (v) is effective against malware affecting a device after the registration phase, but surveillance malware active during registration enables a coercion strategy that reveals re-voting in tallying.

Brute force. Usability requirements (e.g., CoE 1) strongly suggest the PIN must be short, particularly if used seldom and delivered some time before usage. As pointed out in [11], a short PIN may be brute-forced: a coercer briefly in control of a voter's device could try multiple PINs, succeeding with non-negligible probability, then deprive the voter of the device. A rate-limiting measure on ballot casting is therefore advisable, but it must preserve all other requirements. In particular, anonymity (CoE 26) requires the ballot box does not authenticate the voter, and the BB cannot distinguish votes cast with the same ACC and/or PIN. Our proposed solution is the OAuth `CAT` (Section 3). Since the BB cannot do rate limiting, the AS must limit the number of cast attempts by a Client without revealing the identity of the voter to the BB: the `CAT` precisely allows anonymous authentication, thus considerably limits the impact of brute-force attacks. This countermeasure also mitigates against the "leaky duplicate removal" attack of [11], since it becomes more difficult to cast a valid (duplicate) vote.

Typos. Another problem with the PIN approach pointed out by [11] regards usability: humans are quite error-prone when entering a PIN, but the voting process cannot give feedback on its correctness to preserve coercion-resistance. In our proposal, the voter is aided in remembering the PIN by the possibility

to check its correctness with the DVNIZKP at any time, and re-requesting it[3]. Moreover we propose the usage of a visual representation - e.g., via a string of emojis - of the credential reconstructed with the inserted PIN that appears both when verifying the PIN and when voting. In this way the voter has some kind of feedback on the correctness of the inserted PIN.

The link between requirements (Section 5.2) and best practices (Section 2) with the proposed implementing measures is summarized in Table 2.

Table 2: Summary of requirements and implementing measures designed to meet them.

Requirement	Measure
CoE 1: Usability	Short PIN
CoE 7: Unique identification of voters	eID to authenticate at ER
CoE 8: Grant access after authenticating with the right to vote	ER as OAuth AS; OAuth best practices and `CAT` (Section 4.4)
CoE 10: No undue influence	`CAT` (Section 4.4), ruse PIN
CoE 15: Individual verifiability, undue influence detectability	Ruse PIN, CDGT (Section 4.3)
CoE 26: Vote anonimity - unlinkability of the unsealed vote and voter	`CAT`[4] (Section 4.4)
OAuth BCP 1: Replay prevention	DPoP
OAuth BCP 2: Client authentication	Signed JWT, DCR
OAuth BCP 3: Authorization code protection	PKCE
OAuth BCP 4: Privilege restriction	Token exchange

6 Final Remarks

In conclusion, this article offered an architecture for an OAuth 2.0-based solution that supports i-voting from a mobile native client using the protocol of [23]. The proposed method makes use of two specific OAuth 2.0 extensions: Dynamic Client Registration and Token Exchange. Dynamic Client Registration provides two important functions inside the system. First, it allows for the acquisition of a unique client identity per app instance, essentially restricting the attack

[3]Note that the possibility to set a ruse PIN allows to employ anti-coercion strategies.

[4]in addition to measures specified by the protocol - verifiable mixing and re-encryption, additively homomorphic encryption, multi-party computation etc.

surface to particular instances of the app itself. Second, it makes it easier to register a public key, which is later used for authentication against the ER to acquire the `CAT`. Furthermore, the Access Tokens are bound to this public key using the DPoP mechanism, reducing the possibility of stolen tokens being reused. The token exchange is used to obtain a more restricted Access Token for downstream resource servers, which further improves security.

A threat model was established and discussed to guarantee the robustness of the proposed solution, and we suggested how the detected threats can be addressed within our design, ensuring a safe and trustworthy i-voting process. An interesting future work is a more comprehensive and rigorous threat analysis, and a formal proof of security of the protocol, which is also missing in [23].

Acknowledgements

This work has been partially supported by "Futuro & Conoscenza Srl", jointly created by the FBK and Poligrafico e Zecca dello Stato Italiano (IPZS, the Italian Government Printing Office and Mint), as well as by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU. The second and sixth authors are members of the INdAM Research Group GNSAGA.

References

[1] Ben Adida. "Helios: Web-based Open-Audit Voting." In: *USENIX Security Symposium*. 2008. URL: https://www.usenix.org/conference/17th-usenix-security-symposium/helios-web-based-open-audit-voting.

[2] Roberto Araújo and Jacques Traoré. "A practical coercion resistant voting scheme revisited". In: *International Conference on E-Voting and Identity*. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-39185-9_12.

[3] Roberto Araújo et al. "Towards Practical and Secure Coercion-Resistant Electronic Elections". In: *Cryptology and Network Security*. Springer, 2010. DOI: 10.1007/978-3-642-17619-7_20.

[4] Dan Boneh and Victor Shoup. *A graduate course in applied cryptography*. https://toc.cryptobook.us/. 2023. (Visited on 04/18/2023).

[5] Brian Campbell et al. *OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens*. RFC 8705. Feb. 2020. DOI: 10.17487/RFC8705.

[6] Michael R Clarkson, Stephen Chong, and Andrew C Myers. "Civitas: Toward a secure voting system". In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE. 2008. DOI: 10.1109/SP.2008.32.

[7] Véronique Cortier, Pierrick Gaudry, and Stéphane Glondu. "Belenios: a simple private and verifiable electronic voting system". In: *Foundations of Security, Protocols, and Equational Reasoning*. Springer, 2019.

[8] Véronique Cortier et al. "A simple alternative to Benaloh challenge for the cast-as-intended property in Helios/Belenios". working paper or preprint. 2019. URL: https://hal.inria.fr/hal-02346420.

[9] Council of Europe. *Recommendation CM/Rec(2017)5 of the Committee of Ministers to member States on standards for e-voting*. URL: https://search.coe.int/cm/Pages/result_details.aspx?ObjectID=0900001680726f6f (visited on 10/22/2021).

[10] Yvo Desmedt and Yair Frankel. "Threshold cryptosystems". In: *Conference on the Theory and Application of Cryptology*. Springer. 1989. DOI: 10.1007/0-387-34805-0_28.

[11] Ehsan Estaji et al. "Revisiting practical and usable coercion-resistant remote e-voting". In: *Electronic Voting: 5th International Joint Conference, E-Vote-ID 2020, Bregenz, Austria, October 6–9, 2020, Proceedings 5*. Springer. 2020, pp. 50–66.

[12] Daniel Fett et al. *OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP), RFC9449*. Standards Track Draft. IETF WAP WG, Sept. 2023. URL: https://www.rfc-editor.org/rfc/rfc9449 (visited on 09/08/2023).

[13] Google Play Integrity API. *Work with integrity verdicts*. URL: https://developer.android.com/google/play/integrity/verdict (visited on 04/14/2023).

[14] Jens Groth. "A verifiable secret shuffle of homomorphic encryptions". In: *Journal of Cryptology* 23.4 (2010). DOI: 10.1007/s00145-010-9067-9.

[15] Dick Hardt. *The OAuth 2.0 Authorization Framework*. Best Current Practice. IETF RFC 6749, Oct. 2012. URL: https://datatracker.ietf.org/doc/rfc6749/ (visited on 04/07/2023).

[16] Ethan Heilman et al. *OpenPubkey: Augmenting OpenID Connect with User held Signing Keys*. 2023. URL: https://ia.cr/2023/296.

[17] IANA. *JSON Object Signing and Encryption (JOSE)*. Assignment. IANA, Feb. 2023. URL: https://www.iana.org/assignments/jose/jose.xhtml (visited on 04/18/2023).

[18] Michael B Jones, Brian Campbell, and Chuck Mortimore. *JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants*. RFC 7523. RFC - Proposed Standard. May 2015. URL: https://datatracker.ietf.org/doc/rfc7523/ (visited on 04/25/2023).

[19] Michael B Jones et al. *OAuth 2.0 Token Exchange*. Proposed Standard. IETF RFC 8693, Jan. 2020. URL: https://datatracker.ietf.org/doc/rfc8693/ (visited on 04/14/2023).

[20] Ari Juels, Dario Catalano, and Markus Jakobsson. "Coercion-resistant electronic elections". In: *Towards Trustworthy Elections*. Springer, 2010. DOI: 10.1007/978-3-642-12980-3_2.

[21] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. "Accountability: definition and relationship to verifiability". In: *Proceedings of the 17th ACM conference on Computer and communications security*. 2010.

[22] Torsten Lodderstedt et al. *OAuth 2.0 Security Best Current Practice*. Active Internet-Draft. OAuth WG, Mar. 2023. URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-security-topics/ (visited on 04/25/2023).

[23] Riccardo Longo et al. "Adaptation of an i-voting scheme to Italian Elections for Citizens Abroad". In: *E-Vote-ID 2022*. Oct. 2022. DOI: https://doi.org/10.15157/diss/027.

[24] Stephan Neumann and Melanie Volkamer. "Civitas and the real world: problems and solutions from a practical point of view". In: *Seventh International Conference on Availability, Reliability and Security*. IEEE. 2012. DOI: 10.1109/ARES.2012.75.

[25] OWASP. *Threat Modeling Process*. URL: https://owasp.org/www-community/Threat_Modeling_Process.

[26] Justin Richer et al. *OAuth 2.0 Dynamic Client Registration Protocol*. Proposed Standard. IETF RFC 7591, July 2015. URL: https://datatracker.ietf.org/doc/rfc7591/ (visited on 04/14/2023).

[27] Nat Sakimura, John Bradley, and Naveen Agarwal. *Proof Key for Code Exchange by OAuth Public Clients*. Best Current Practice. IETF RFC 7636, Sept. 2015. URL: https://datatracker.ietf.org/doc/rfc7636/ (visited on 04/04/2023).

[28] Nat Sakimura et al. *OpenID Connect Core 1.0*. Nov. 2014. URL: https://openid.net/specs/openid-connect-core-1_0.html (visited on 05/18/2023).

[29] Amir Sharif et al. "Best current practices for OAuth/OIDC Native Apps: A study of their adoption in popular providers and top-ranked Android clients". In: *Journal of Information Security and Applications* 65 (2022). DOI: https://doi.org/10.1016/j.jisa.2021.103097.

[30] Ben Smyth et al. "Towards automatic analysis of election verifiability properties". In: *Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*. Springer. 2010.

[31] Meltem Sönmez Turan et al. *NIST SP 800-90B Recommendation for the Entropy Sources Used for Random Bit Generation*. NIST, Jan. 2018. DOI: 10.6028/NIST.SP.800-90B.

[32] U.S. Election Assistance Commission. *Voluntary Voting System Guidelines (VVSG) version 2.0*. 2021. URL: https://www.eac.gov/voting-equipment/voluntary-voting-system-guidelines (visited on 10/22/2021).