Election Eligibility with OpenID: Turning Authentication into Transferable Proof of Eligibility

Véronique Cortier, Alexandre Debant, Anselme Goetschmann, and Lucca Hirschi, *Université de Lorraine, CNRS, Inria, LORIA, France*

https://www.usenix.org/conference/usenixsecurity24/presentation/cortier

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Election Eligibility with OpenID: Turning Authentication into Transferable Proof of Eligibility

Véronique Cortier Alexandre Debant Anselme Goetschmann Lucca Hirschi

Université de Lorraine, CNRS, Inria, LORIA, France

Abstract

Eligibility checks are often abstracted away or omitted in voting protocols, leading to situations where the voting server can easily stuff the ballot box. One reason for this is the difficulty of bootstraping the authentication material for voters without relying on trusting the voting server.

In this paper, we propose a new protocol that solves this problem by building on OpenID, a widely deployed authentication protocol. Instead of using it as a standard authentication means, we turn it into a mechanism that delivers transferable proofs of eligibility. Using zk-SNARK proofs, we show that this can be done without revealing any compromising information, in particular, protecting everlasting privacy. Our approach remains efficient and can easily be integrated into existing protocols, as we have done for the Belenios voting protocol. We provide a full-fledged proof of concept along with benchmarks showing our protocol could be realistically used in large-scale elections.

1 Introduction

Internet voting is used in politically-binding elections in several countries, including Estonia, Australia, Canada, France, and Switzerland, either for local or national elections. It is more widely used in other types of elections, such as professional elections. Internet voting should aim to achieve the same level of security as traditional paper-based elections. This includes two main key properties. The first one is *vote secrecy* [12]: no one should know how a voter has voted. Internet voting heavily relies on cryptographic techniques, which poses an additional challenge: votes should remain secret even if the cryptography in use is eventually broken. This is called *everlasting privacy* [33].

This work benefited from funding managed by the French National Research Agency under the France 2030 programme with the reference ANR-22-PECY-0006. It was also partly supported by the ANR Chair IA ASAP (ANR-20-CHIA-0024) with support from the region Grand Est, France.

The second main security property is *verifiability* [24]: everyone should be able to check that the result reflects the intention of the voters. This property is usually broken down into three sub-properties.

- *Individual verifiability*: a voter should be able to check that their ballot is in the ballot box and that their ballot contains their intended vote (*cast-as-intended*).
- *Universal verifiability*: everyone should be able to check that the result matches the content of the ballot box. Such checks are often outsourced to external auditors.
- *Eligibility verifiability*: everyone should be able to check that the ballots have been cast by legitimate voters.

Universal verifiability is relatively easy to achieve, either by using mixnets that shuffle and re-randomize ballots or by using homomorphic encryption. Then a Zero-Knowledge Proof (ZKP) of correct decryption (and possibly correct shuffling) allows external observers to verify the correctness of the result. Both of these techniques have been successively deployed *e.g.,* in Helios [6]. Individual verifiability is more difficult to achieve as it requires the voter to be in the loop. Various techniques have been proposed. For example, Helios [6] proposes the Benaloh's challenge (for the cast-as-intended part) and a public bulletin board containing the ballot box. Estonia [34] is based on a two-device approach, where the voter casts a vote with their computer and checks their vote with their mobile phone. Switzerland [52] is developing a system based on return codes, which are checked by the voter against a sheet received by mail. Surprisingly, eligibility verifiability has received less attention even though a lack of eligibility verifiability enables voting servers to do *ballot stuffing*.

Related Work on Eligibility. Many systems (*e.g.,* Helios [6], Select [42], FLEP [28]) simply assume an authenticated channel between the voter and the voting server. In practice, this means that voters are given a login and a password to authenticate themselves during the voting phase. This provides weak guarantees: the voting server cannot prove that the voter has been successfully authenticated. Instead, it has full power to add ballots, *e.g.,* in the name of absentee voters.

Furthermore, the login and password can be stolen, for example when voter's email gets compromised. Worse, it may even be extremely difficult in practice to reliably and securely deliver such login and password information to all eligible voters. For example, the 2022 French Legislative election, one of the largest e-voting elections with 1.5 million eligible voters, was organized online using the FLEP protocol [28]. However, in 2 out of 11 constituencies, less than 40% of voters received their material (delivered by SMS and email), leading to a re-run of the elections in 2023 [4].

In order to guarantee some form of eligibility verifiability, several systems such as SwissPost [52], VoteAgain [43], Civitas [23], or Belenios [25] assume an external entity during setup, often called *Registrar*, that sends credentials to voters. During the voting phase, the credentials are linked to the cast ballots in a verifiable way (the exact technique varies widely depending on the protocol). However, this comes at the cost of adding an additional, ad hoc, entity, whose trustworthiness is required for having eligibility verifiability. In addition, the credentials still need to be sent to voters, with the risk of interception or simply non-delivery. Moreover, this new entity, must have communication channels with all eligible voters, and those must not be bootstrapped with the help of the possibly malicious voting server. This approach thus requires strong system and trust assumptions, which are often unrealistic in practice.

Finally, a third and simple approach [34, 49] consists in assuming a Public Key Infrastructure (PKI) for voters. In this case, voters simply sign their ballot with their private signing key. The main issue here is that in practice voters do not have a PKI. A noteworthy nationwide exception is Estonia, where voters have an electronic identity card that is used for various administrative procedures. So, naturally, the Estonian voting system [34] relies on this PKI for e-voting and provides some form of eligibility verifiability. However, ballot signature, as done in Estonia, compromises everlasting privacy: anyone with access to the ballot box knows the link between ballots and voters, hence, if encryption is broken in the future, vote secrecy will be lost as well. This is mitigated in Estonia by the fact that the ballot box is not public, which in turn does not offer *universal* eligibility verifiability. Such a PKI is anyway not available in most of the other election contexts, where this approach is not feasible.

OpenID Connect. In this paper, we propose a novel eligibility mechanism, based on OpenID Connect [50]. OpenID is a widely deployed authentication protocol that allows users to authenticate to multiple websites using their credentials associated with a single entity: the OpenID Provider (idP). For example, many websites offer users to authenticate using their Google, Microsoft, Apple, or Amazon account. This greatly simplifies account management for websites and users no longer have to manage multiple passwords. When a user wants to connect to a website or a service, called Relying Party (RP), they are redirected to the idP, who (i) informs the user of what information will be transfered to the RP and (ii) authenticates the user. If the authentication is successful, the idP issues a token that allows the RP to obtain a signature from the idP, certifying that the user has authenticated, along with additional information (*e.g.,* date of birth), as agreed by the user. A RP can decide from which idP it accepts the authentication. Well-known idPs are typically owned by big technology companies, but there are also several nationwide idPs, run by countries such as Canada, Netherlands, France, or Switzerland, as suggested by the iGOV working group [2].

Contributions. We show that it is possible to use OpenID Connect to produce privacy-protecting *transferable proof of eligibility*. Such proofs may also be of interest in other contexts outside e-voting, where a user needs to prove that they possess some required attribute, *e.g.,* their age or nationality to be eligible to some service.

The OIDEli protocols. Our first main contribution is the OpenID-Eligibility (*OIDEli*) protocol, which provides eligibility verifiability based on OpenID Connect. More specifically, we show that the existing structure of OpenID can be used to obtain a transferable signature from the idP that an authenticated and eligible voter has submitted a ballot b. It is then sufficient to add this signature to the bulletin board in order to obtain eligibility verifiability. The challenge is to use the idP as a signature oracle while ensuring that what is signed remains under the control of the voter, even for a malicious voting server. Importantly, we strictly adhere to the OpenID specification, so that our protocol can readily be used with existing implementations and idPs. Moreover, our approach is modular and can be applied to most voting protocols, provided that, at some point, a voter submits a ballot that is then published onto a bulletin board. Note that some voting systems, such as Helios [6] or Belenios [25], already offer the use of OpenID as a means of authentication. However, they use OpenID as a pure authentication layer, which does not bring any eligibility verifiability: the voting server cannot prove that a user has been authenticated. In fact, a malicious voting server can simply bypass the authentication and add ballots without being detected. For such cases, our protocol allows to weaken trust assumptions: instead of having to trust a single entity (here the voting server), *OIDEli* distributes the trust as it guarantees eligibility verifiability as long as either the idP or the voting server is honest.

As for protocols that rely on a PKI, this first protocol may compromise everlasting privacy. Indeed, the signature publicly links a ballot and its voter, which additionally leaks who voted and when, since the idP produces timestamped signatures. We address this problem with our second main contribution, the OpenID-Eligibility-ZK-ID (*OIDEli-zk*) protocol, where the voting server now proves, in zero-knowledge, that for each ballot, it has received a valid signature from the idP for a legitimate voter, but without revealing the identity

of the voter. There is no theoretical difficulty here since this remains a polynomial statement. However, it is difficult in practice since the usage in OpenID of standard cryptography (specifically SHA2) is not well suited to ZKP. In particular, we show that a naive ZKP arithmetic circuit for this statement would yield unreasonable proving time.

Optimized SNARK proof and Proof of Concept. We propose an optimized circuit design and an implementation thereof in the zero-knowledge Succinct Non-interactive Argument of Knowledge (zk-SNARK) implementation Plonky2 [53]. In particular, we show how to comply with the encoding and formatting used by OpenID (that uses JSON and Base64) and how to avoid leaking the length of identity-related information. We implement our ZKP circuit in Plonky2, although another framework could be used as well, and show that it is efficient enough to run the largest politically-binding online election to date, which is the 2022 French legislative elections to the best of our knowledge [28]. Importantly, this is done at no cost to the voter, since the ZKPs are only computed by the voting server, once it has received a valid signature from the idP. We have also integrated our *OIDEli-zk* protocol into Belenios with Google as idP, showing that it is indeed modular and readily usable with existing implementations of OpenID.

Security Analysis. Finally, we discuss the security properties met by the *OIDEli* protocols and conduct a formal security analysis using the verification tool ProVerif [17, 18]. We formally prove that both *OIDEli* and *OIDEli-zk* guarantee eligibility verifiability only assuming an honest idP. As previously explained, *OIDEli-zk* cannot provide everlasting privacy if the underlying protocol does not protect it. So instead, we show that *OIDEli-zk* always guarantees *identity-hiding*: an attacker never learns who takes part to the protocol when the voting server and the idP are honest. Other properties such as everlasting privacy may depend on the underlying voting protocol and cannot be formally proven for our protocols in isolation; we still prove them for a Helios-like voting protocol under standard trust assumptions. Moreover, we informally explain why our protocols are expected to preserve individual verifiability, universal verifiability, and vote privacy, should they hold for the underlying voting protocol.

In brief, our contributions can be summarized as follows:

- *OIDEli* protocol, that turns the OpenID authentication mechanism into a transferable proof of eligibility;
- *OIDEli-zk* additionally provides a zero-knowledge version of the transferable proof, using zk-SNARKs, to protect voter's privacy and in particular everlasting privacy;
- A PoC implementation of *OIDEli-zk*, integrated into Belenios, shows the practicability of our proposal;
- A formal security analysis performed with ProVerif.

Limitations. Note that our approach may not always be compatible with voting protocols that aim to achieve coercion resistance or receipt-freeness. A protocol is said to be receipt-free if an attacker cannot learn the voter's vote even if the voter is asked to provide all the material produced during the voting phase. A stronger property is coercion resistance: it guarantees that a voter can vote freely even if they are asked to provide all their voting material, including the authentication material. Such properties are achieved by providing voters with credentials (for authentication) but in such a way that voters can either lie about it (as in Civitas [23]) or secretly revote using the same credentials and without the adversary noticing (as in VoteAgain [43]). Our approach is not directly applicable to such fine-grained handling of authentication credentials and would require careful, dedicated modifications, which are left as future work.

We stress that eligibility verifiability always relies on trust assumptions. For example, if it relies on a PKI, then this PKI must be trusted. In our case, we assume a trusted idP. Trusting Google or Amazon for national politically-binding elections may not be acceptable but could be perfectly suitable for other types of elections. For politically-binding elections, national identity providers might be preferred. This choice should, as always, be left to the electoral authorities and citizens' representatives. One advantage of our approach is that it is based on a widely deployed protocol, allowing voters to reuse authentication means that they know and trust, and to distribute the trust to avoid a single-point-of-trust when the voting server also checks eligibility.

Related Work on OpenID. Using OpenID as a signature oracle has been explored in the past [35, 47] but most of these approaches do not protect user's privacy, which would break everlasting privacy in our case. In order to preserve privacy, CanDID [44] proposes a decentralized identity management service, that allows to gather identities from several services. Because OpenID is not privacy-friendly by default (we nonetheless make it so by the means of ZKPs), CanDID relies on oracle systems that require stronger system and trust assumptions: either MPCs with distributed verifiers or TEEs at clients. On the other hand, CanDID addresses a different problem and, in particular, does not address the challenges of e-voting eligibility. For example, it neither allows to link ballots to voters' identities nor prevents cheating voters from re-using old credentials without having to re-authenticate. A very recent and independent work [9], zkLogin, shows how to to use OpenID in order to bootstrap a fresh signature key-pair, authenticated by the OpenID identity provider. Like in our approach, zk-SNARK protect privacy and several optimizations are proposed for parsing JSON and for off-loading ZKPs computations. However, it relies on stronger trust and system assumptions that are typically unsuitable for e-voting (*e.g.,* complex setup ceremony, dedicated app embedding a RP, Trusted Execution Environment (TEE)). Moreover, it does not address eligibility proof and does not solve its related challenges that we further detail in Section 3.1 such as the signature oracle issue, enforcing cheating voters to (freshly) authenticate, or hiding the length of the voters' identity to protect voters' privacy.

2 Context

Notations. We assume several cryptographic primitives. Namely, $\mathsf{hash}(\cdot)$ is a hash function, while $\mathsf{sha}_2(.)$ specifically denotes the SHA2 hash function. We use $\cdot\|\cdot$ as a bitstring delimiter. We denote $\mathsf{sign}_{.}(\cdot)$ a signature primitive (*e.g.,* RSA) and $\mathsf{checkSign}(\cdot,\cdot)$ the associated verification function.

2.1 E-Voting Protocols

We provide eligibility verifiability in a modular way. We make only a few assumptions about the underlying *voting protocol*.

Voter list. We assume that the voter list $\mathsf{ID}_{\mathsf{el}}$ is public and known in advance. The correctness of the voter list is out of scope of the voting protocol. We also assume that voters in $\mathsf{ID}_{\mathsf{el}}$ can be represented by identifiers known to the identity providers used in OpenID, for example using their registered email address or last names and first names.

Voter. Voters use a *voting device* to cast their vote. They will need this voting device to authenticate using the OpenID protocol. They may use other devices to build their ballot and/or to perform checks. The only assumption that we make here is that they send a *ballot b* to the voting server and that it will appear on the bulletin board. This ballot should be sufficient to guarantee that the vote is properly recorded and tallied, for instance through the use of so-called *ballot tracker* and verifiable homomorphic encryption as in [25]. This ballot may be purely computed by the voter's device or in interaction with other devices or with the voting server, that may *e.g.,* re-randomize the initial ballot to produce *b*.

Bulletin Board (BB). We assume that the protocol has a notion of BB, on which the ballots are published. This board may be fully public or only available to some entities, and in particular to auditors, in charge of checking the eligibility.

Voting server. We also assume a voting server that centralizes the ballots from voters. It may authenticate voters, verify their right to vote or revote, and perform some checks. The voting server will not be trusted for eligibility. We solely assume that it publishes the voters' ballots on the BB.

Auditors. External auditors (or verifiers) access the BB and check the correctness of the cryptographic material such as signatures or zero-knowledge proofs.

Those assumptions capture a large class of protocols, that notably includes Helios-like protocols [6, 21, 25], Select [42], Selene and its variants [48, 49], Civitas [23], Polyas [45], and protocols used in national elections in Estonia [34], Switzerland [52], and France [28]. On the other hand, we discard decentralized protocols, such as self-tallying protocols [38].

Example 1. *We present a Mini-Voting protocol inspired from [14]. Before the election starts, decryption trustees compute the public key* pk_{E} *of the election in a distributed manner. Each decryption trustee has a share of the decryption key.*

- *A voter* id *encrypts their choice v with randomness r, yielding a ballot* $b = \mathsf{enc}(v, r, \mathsf{pk}_{\mathsf{E}})$ *with a homomorphic encryption scheme* enc.
- *The voter then authenticates to the voting server with identity* id*, that publishes b on the BB, possibly removing the previous ballot from* id *if any.*
- *The voter checks that their ballot appears on the BB.*

At the end of the election, the encryption of the final result of the election can be obtained by homomorphically combining the ballots. The decryption trustees collectively decrypt this final ballot and publish onto the BB the result and a ZKP of correct decryption. Auditors are in charge of checking the correctness of all the ZKPs. They should also check that no ballot disappears from the BB.

Helios [6] is an instance of the Mini-Voting protocol where enc *is the ElGamal encryption together with a ZKP of well-formedness of the ballot.*

2.2 OpenID Protocol

The OpenID protocol involves three main participants. The End User (EU) wishes to connect to a website or a service, called the RP. For this, they authenticate to an idP that issues a token that grants the EU access to the RP. The RP is registered to the idP. In particular, they share a secret key, used to authenticate their exchanges (with a MAC).

In a voting protocol setting, the EU is the voter and the RP is the voting server, while the idP is a trusted third party such as Google, Apple, or some national identity provider.

The detailed specification of OpenID Connect can be found in [50]. It offers several possible authentication flows. We provide here a high level description of the standard flow, that is, the Authorization Code Flow (See Section 3.1 of [50]). We depict this flow in black in Figure 1.

Consider an EU wishing to connect to a website, the RP, that offers OpenID with idP as an authentication means. When the EU clicks on the login button, the RP issues an **Authentication Request** $\mathsf{AuthReq}(N)$ where N is a nonce generated by the RP. This request is sent to the EU, who is redirected to the idP. The EU is then prompted to authenticate towards the idP, *e.g.,* in a pop-up page. If successful, the idP sends an **Authorization Code** back to the EU, which is transmitted to the RP. The RP can send a (MAC authenticated) request containing this Authorization Code directly to the idP to obtain a signed **ID Token** that attests that the EU successfully authenticated themselves towards the idP. The ID token is a JSON Web Token notably made of a JSON object tok and a signature $\sigma := \mathsf{sign}_{\mathsf{idP}}(\mathsf{sha}_2(\mathsf{tok}))$ with the idP's signature private key idP.[1] The JSON object $\mathsf{tok} = \mathsf{json}(a_1, \ldots, a_n)$ contains several fields a_i and in particular:

- (optionally) N, the nonce sent by the RP,

[1] The signature algorithm typically used is `RSASSA-PKCS1-v1.5` using sha_2 [37], which essentially means the token is hashed with sha_2 and then signed with RSA.

- sub, the **Subject Identifier**, a locally unique and never reassigned identifier within the EU and the RP. A EU will always have the same, unique sub when connecting to a given RP. But a different sub will be used when connecting to another RP.
- the Audience Value aud, that is, a unique identifier of the RP. This guarantees that the token cannot be used for another RP.
- the token may contain several optional fields, such as the EU's name, email address, gender, birth date, etc.

Importantly, the ID token is sent only once the idP has obtained from the login page the *consent* of the EU to communicate such information to the website/RP. The consent of the user is a key feature of OpenID Connect.

Once the RP receives a valid signed token, it can grant its user (the EU) access to the requested service (*e.g.,* a private webpage).

2.3 Zero-Knowledge Proof Systems

ZKP [20, 29, 54] allows one to prove they know some private data privZK satisfying some (public) statement $\mathcal{S}$ depending on some public data pubZK. Such a proof leaks nothing about privZK (**zero-knowledge**) and forging a proof without knowing suitable privZK is computationally infeasible (**soundness**). Usually, it is also required that the proof computation always succeeds on data satisfying $\mathcal{S}$ (**completeness**). We denote $\pi := \mathsf{ZK}_{\mathcal{S}}(\mathsf{privZK})(\mathsf{pubZK})$ the ZKP of a statement $\mathcal{S}$ with private inputs privZK and public inputs pubZK. The associated verification function is $\mathsf{checkZK}_{\mathcal{S}}(\pi, \mathsf{pubZK})$. As an illustration, one can prove the knowledge of a SHA2 preimage: $\mathsf{privZK} = x$, $\mathsf{pubZK} = h$, and $\mathcal{S} =$ "I know x such that $h = \mathsf{sha}_2(x)$".

Numerous ZKP systems exist (see *e.g.,* surveys [29, 54]) with different set of features and performance. For our use case, we seek for a system that meets the following criteria:

1. *transparent*, *i.e.,* no setup ceremony as it induces non-trivial trust assumptions in the e-voting use case. We only assume a Common Random String (CRS), made practical by efforts such as distributed randomness beacon [1].
2. *non-interactive*: verifiers should be able to verify non-interactively.
3. we must be able to compute a proof of knowledge of a SHA2 preimage with reasonable performance (matter of minutes). This will be the costliest operation of our proof (see below). Sadly, it is not possible to switch to a more suitable hash function (such as poseidon discussed below) as we inherit the choice made by OpenID.
4. the scheme should be *perfect-zero knowledge*, which intuitively means that a computationally unbounded attacker learns nothing about privZK, we will see that everlasting privacy would otherwise be put at risk.

In particular, we are not particularly interested in minimizing the verification cost, which can be distributed and spread over time. Similarly, the proof size only needs to be reasonable (in the range of MB) since the number thereof is upper-bounded by the number of eligible voters.

Putting all those constraints together led us to look at recent zk-SNARK implementations, and we selected Plonky2 [53] (based on PLONK [30] and FRI [11]) for our Proof of Concept (PoC) (see Section 6) by studying performance benchmarks [31,46,51,55,56]. However, some other systems are certainly suitable as well and our protocol is presented in a modular way. In particular, we shall specify the ZKP statement for our protocol in the more general framework of arithmetic circuits thus capturing a large class of other ZKP systems [54].

Arithmetic circuits [20, 29, 54] is a computational model for polynomials and is the input format of the statement $\mathcal{S}$ to prove for many ZKP systems [54]. The set of polynomials $\mathcal{K}[X_1, \ldots, X_n]$ is defined with respect to a given field $\mathcal{K}$ (*e.g.,* Plonky2 uses a 64-bit field), and the variables X_i correspond to the private values from privZK. The idea is to encode the statement to prove $\mathcal{S}$ into an arithmetic circuit that yields a polynomial P such that $P[v_1, \ldots, v_n] = 0$ if, and only if, $\mathsf{privZK} := (v_1, \ldots, v_n)$ satisfies the statement $\mathcal{S}$ (for simplicity, we assume pubZK is directly encoded into P).

Circuits have *wires*, carrying data to be processed, and *gates*, processing data. For the sake of simplicity and modularity, we make few assumptions about the circuit language and solely assume the following basic operations: (i) *input wires*, *i.e.,* those that are not outputs of gates, are associated to variables X_i (*i.e.,* private data) or constants (elements of $\mathcal{K}$), (ii) gates are associated to field ($\mathcal{K}$) operations (*e.g.,* addition, multiplication), (iii) equality constraints between wires (which can be easily encoded as a subtraction). This straightforwardly translates to a single constraint $P = 0$ for a polynomial $P \in \mathcal{K}[X_1, \ldots, X_n]$, where polynomial evaluation is done by associating a field element from $\mathcal{K}$ to each wire.

We assume that if one can construct the statement $\mathcal{S}$ as an arithmetic circuit, then one can use $\mathsf{ZK}_{\mathcal{S}}(\cdot)(\cdot)$ and $\mathsf{checkZK}_{\mathcal{S}}(\cdot)$. In practice, Plonky2 and possibly other systems provide such functionalities and meet all criteria listed above (see Section 6).

Even though, in theory, any NP statement can be encoded, one needs to strive to minimize the number of gates and constraints to obtain sufficiently efficient prover and verification time. As we shall see in Section 4, one of the main challenges with designing such circuits is that not all information and computation thereon can be efficiently encoded as respectively elements in $\mathcal{K}$ and $\mathcal{K}$-operations. For instance, sha_2 is notoriously costly to encode [32, 54] because it mainly consists of bitwise operations that require to encode each bit of data as a wire, associated to a field element (of 64 bits for Plonky2). When possible, "zk-friendly" hash functions such as poseidon [32] are preferable as most internal computations can directly be performed in the field.

3 OpenID-Eligibility (*OIDEli*) Protocol

While the OpenID protocol is primarily designed to offer an authentication mechanism, we show in Section 3.1 how it can be used to obtain a *transferable proof* which can convince external observers that a legitimate voter has voted.

The core idea is to obtain a signature from the OpenID provider of the fact that a voter with some identity id, included in a list of eligible identities, cast some ballot b. Such a signature (the signed ID Token) can be published on the BB and verified by anyone. However, revealing the ID token also reveals privacy-sensitive information, notably the voter's identity id and the exact timestamp of when the signature was produced, and hence when the voter voted. Therefore, anyone accessing the BB would be able to know who voted and when. Worse, this may compromise everlasting privacy, as explained in the introduction. Hence we show in Section 3.2 that this signed token may be replaced by a ZKP of knowledge of a (valid) signature, providing eligibility verifiability while still hiding voter-related information and preserving everlasting privacy.

Importantly, we do not require any modification of the OpenID protocol, hence we can readily use existing implementations. Moreover, our design fits a large class of voting protocols, as defined in Section 2.1.

3.1 *OIDEli*: Achieving Universal Eligibility Verification with OpenID

In OpenID, the idP signs an ID token tok. This token may contain several fields, notably the EU's identity id and some nonce N chosen by the RP, *i.e.,* the service provider, here the voting server. Since the idP signs an arbitrary nonce, our first key idea is to use it as a signature oracle. In particular, if a voter with identity id wants to cast a ballot b, we can obtain a signature $\mathsf{sign}_{\mathsf{idP}}(\mathsf{sha}_2(\mathsf{json}(\mathsf{id},b,\ldots)))$ by choosing $N=b$. This basic idea comes however with several issues:

• *format issue*: N is a nonce and not an arbitrary bitstring. This can be easily solved by hashing b first, that is, choosing $N=\mathsf{hash}(b)$. Importantly, the hash function hash can be any hash function of our choice. In particular, this hash function can be chosen to be well suited for a specific ZKP system.

• *cheating voters*: the RP (the voting server) is not guaranteed that the EU (the voter) has recently authenticated themselves to the idP because the voter could replay an old Authorization Code by reusing the same $N=b$.

• *signature oracle issue*: neither the idP nor the voter have control on which ballot b is actually signed. This happens "behind the scenes" for the voter and the OpenID provider is not even aware that an election is running. In particular, the voting server would be able to obtain a signed token for a voter's identity id and a ballot of its choice that can be different from the ballot cast by voter id.

• *control flow issue*: the token tok with the signature of the idP is sent to the RP (the voting server), not to the EU (the voter). Hence the voter loses control on when a valid ballot is produced in their name.

These potential threats guide the design of our protocol. Hence we propose two key principles:

1. the voting client *inspects* the authentication request $\mathsf{AuthReq}(N)$ to check that N is legitimate, *e.g.,* that it contains the ballot b the voter is willing to cast, thwarting the RP maliciously choosing N.
2. to keep control on the moment where a valid signature of the ballot is released to the voting server, the ballot b is not sent directly to the idP. Instead, the voting client *commits* to the ballot by sending $\mathsf{hash}(b,n_V^0)$. The fresh nonce n_V^0 is revealed only at the end, once the voting client has controlled the validity of the ID token.

Protocol Description. With these design principles in mind, we can now describe our *OIDEli* protocol, depicted in Figure 1. The protocol starts as soon as the voter with identifier id is ready to cast a ballot b as prescribed by the voting protocol. This ballot may be purely computed by the voting client or after some interaction with the voting server (*e.g.,* that may re-randomize the ballot).

- The voting client (the EU) first generates a fresh nonce n_V^0 and commits to the ballot, yielding $n_V=\mathsf{hash}(b,n_V^0)$ that is sent to the voting server (the RP).
- The voting server generates a fresh nonce n_S, as prescribed by OpenID: the RP must defend against replay of authentication requests. We will see in Section 3.2 that this nonce will also be used to protect privacy.
- This determines the nonce $N=\mathsf{pad}_{\mathsf{id}}(\mathsf{hash}(n_S\|n_V))$, used for the authentication request. As explained earlier, we use a hash function to comply with the size constraint of N and to hide n_S that eventually must remain private on the BB (see Section 3.2). We also add some padding that depends on the length of id with $\mathsf{pad}_{\mathsf{id}}$ (defined in Section 4.1), in order to dramatically optimize the ZKP (see Section 4).
- The authentication request $\mathsf{AuthReq}(N)$ is emitted by the voting server and sent to the voting client together with n_S. The voting client inspects the authentication request and checks that $N=\mathsf{pad}_{\mathsf{id}}(\mathsf{hash}(n_S\|n_V))$ and transmits the authentication request to the idP.
- The voter authenticates to the idP as done in OpenID.
- Upon successful authentication, the idP sends an ID token $\mathsf{tok},\sigma:=\mathsf{sign}_{\mathsf{idP}}(H)$ where $H=\mathsf{sha}_2(\mathsf{tok})$ to the voting server, as specified by OpenID. The token $\mathsf{tok}=\mathsf{json}(\mathsf{id},\mathsf{sub},N,\mathsf{aud})$ is the JSON encoding of different fields that include id, sub, N, and aud. It also contains other fields that we omit for simplicity.
- The token and signature are sent back to the voting client, which checks that everything is valid. If yes, the voting client releases n_V^0 to the voting server.

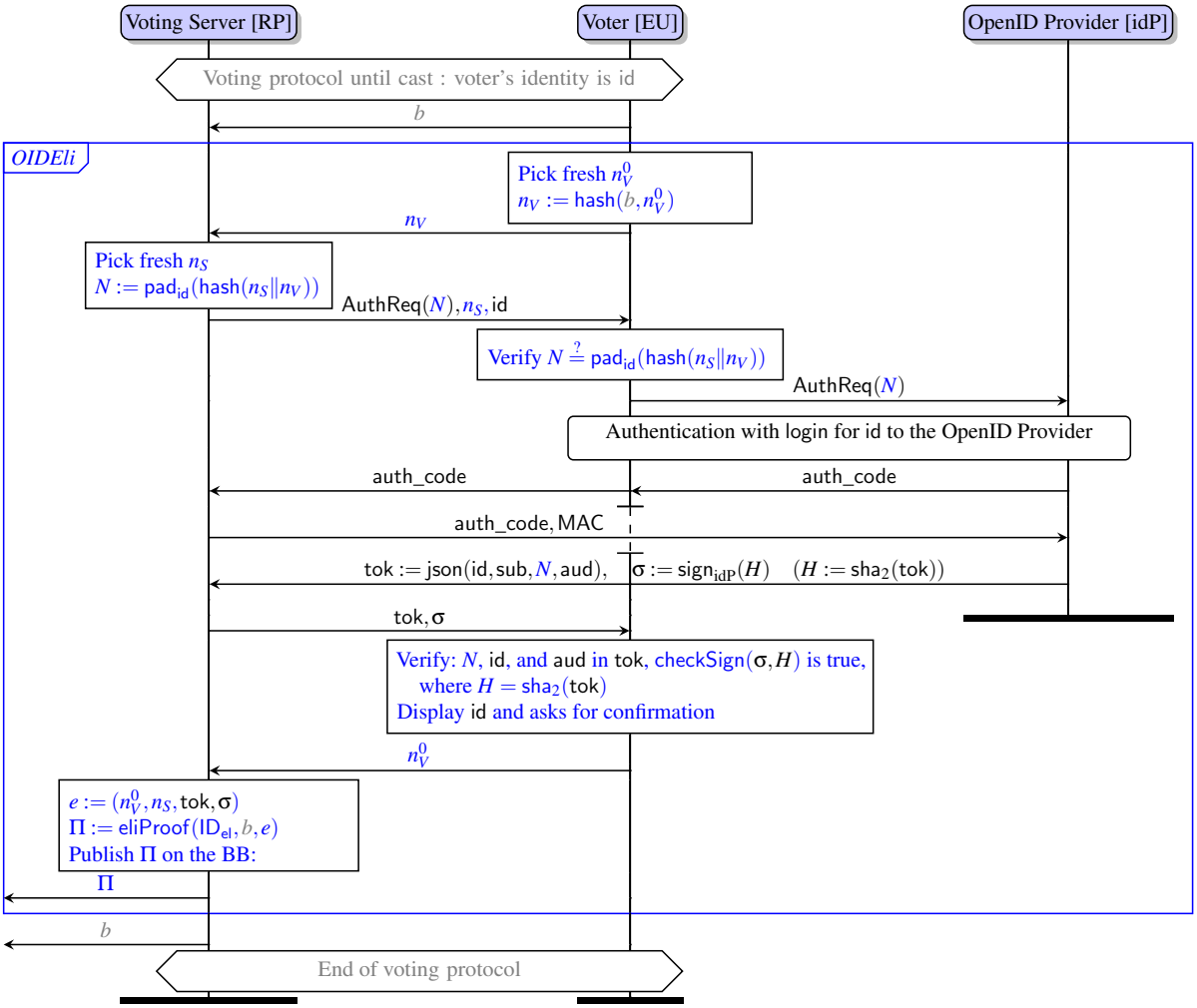

Figure 1: **OpenID-Eligibility (*OIDEli*) Protocol**. Our protocol starts when the voter is ready to submit their ballot to the voting server. We depict in blue the messages and actions of our protocol, in black the messages from the OpenID protocol flow and in gray from the voting protocol. MAC is a MAC with a symmetric key shared by the Voting Server and the OpenID Provider. $\mathsf{pad}_{\mathsf{id}}(\cdot)$ adds some padding depending on the length of id (see Section 4.1). The server produces an eligibility proof $\Pi = \mathsf{eliProof}(\mathsf{ID}_{\mathsf{el}}, b, e)$. Eligibility verifiers (auditors) can verify such a proof by running $\mathsf{checkEliProof}(\mathsf{ID}_{\mathsf{el}}, b, \Pi)$. The eligibility proof eliProof and verification checkEliProof algorithms are specified in Tables 1 and 2, depending on the protocol version we consider.

- The voting server knows the ballot b and elements $e := n_V^0, n_S, \mathsf{tok}, \sigma$ obtained through our protocol. These elements form a transferable proof of eligibility. Indeed, the voting server can prove that it has received a valid ballot from voter id. Thanks to the signature of the idP, the voting server cannot forge such a proof. For now, we assume e is published as-is on the BB.
- The voting protocol then continues as expected. For example, voters may perform extra checks, for verifiability purposes.

We denote $\mathsf{eliProof}(\mathsf{ID}_{\mathsf{el}}, b, e)$ the eligibility proof obtained from b, e. For this first version, we consider the simple case where all the elements are published, hence $\mathsf{eliProofplain}(\mathsf{ID}_{\mathsf{el}}, b, e) = e$. This proof contains, in particular, the signature of the OpenID provider, which can be verified by external observers (including auditors). The precise checks are defined by the algorithm $\mathsf{checkEliProofplain}(\mathsf{ID}_{\mathsf{el}}, b, e)$ described in Table 1. Eligibility observers must also check that all ballots come from distinct voters (that is, voters id are pairwise distinct).

Note that the publication of the ballot b on the BB of the voting protocol can be independent of, and can be done earlier than, the publication of the proof of eligibility Π. This way, individual verifiability (*e.g.,* checking b on the BB) is not impacted. However, the voting server must check it can produce Π first, *i.e.,* all checks in $\mathsf{eliProof}(\cdot, \cdot, \cdot)$ are satisfied.

$\mathsf{eliProofplain}(\mathsf{ID}_{\mathsf{el}}, b, e) := e$
$\mathsf{checkEliProofplain}(\mathsf{ID}_{\mathsf{el}}, b, (n_V^0, n_S, \mathsf{tok}, \sigma))$ is true if : • $\mathsf{tok} = \mathsf{json}(\mathsf{id}, \mathsf{sub}, N, \mathsf{aud})$ and $\mathsf{id} \in \mathsf{ID}_{\mathsf{el}}$ • $\mathsf{checkSign}(\sigma, H)$ is true with $H = \mathsf{sha}_2(\mathsf{tok})$ • $N = \mathsf{pad}_{\mathsf{id}}(\mathsf{hash}(n_S \Vert \mathsf{hash}(b, n_V^0)))$

Table 1: Eligibility proof for *OIDEli-id*. $\mathsf{ID}_{\mathsf{el}}$ is the static set of the eligible identities publicly published on the BB. For this version (plain-id), the full authentication token tok (hence the id of the voter) and n_S are provided in clear. Eligibility verifiers (auditors) must also check that all id associated to the ballots published on the BB are pairwise distinct.

Trust Assumptions. As we shall see, *OIDEli-id* guarantees eligibility verification assuming an honest idP. In contrast, the voting server (RP) can be dishonest, as well as arbitrary many voters. We also assume other dishonest RPs. In particular, an (honest) voter may be willing to authenticate to the idP to access other services provided by a potentially malicious RP'. On the other hand, we assume that, when an EU uses idP to authenticate to some RP', the user does check the displayed consent screen that specifies the context of the authentication (here with RP'). Otherwise, a malicious service RP', colluding with the voting server RP could trigger Alice to authenticate to the idP and obtain a valid signature from the idP of some ballot of its choice. Our protocol resists such threats thanks to the signature containing aud, the identifier of the RP, and the fact that the name of the RP is displayed to the EU on the consent screen. Indeed, the idP is responsible for providing such a clear consent screen to the EUs.

Security Claims. We formally prove in Section 5 that OpenID-Eligibility-clear-ID (*OIDEli-id*), implemented with eliProofplain (Table 1), provides **eligibility verifiability** for an honest idP: if a ballot b appears on the BB with a valid proof eliProofplain, then the ballot has indeed be emitted by an eligible voter $\mathsf{id} \in \mathsf{ID}_{\mathsf{el}}$, which has successfully authenticated to the OpenID provider. Furthermore, there is at most one ballot per eligible voter. Section 5 gives more details.

However, this first version of our protocol, *OIDEli-id*, may break everlasting privacy and reveal privacy sensitive voters' information. Indeed, in many voting protocols, the ballot b, published on the BB, contains the encryption of the vote. Moreover, $\mathsf{eliProofplain}(\mathsf{ID}_{\mathsf{el}}, b, e)$ (provably) reveals the identity of the voter, as well as when the voter voted (tok has a timestamped field). Such sensitive information thus leak to anyone having access to the BB. If, additionally, encryption is broken in the future, then any BB reader learns the votes of all voters, hence breaking everlasting privacy. We show in the next section how we can entirely hide on the BB both tok and its privacy-sensitive fields with the use of ZKPs.

3.2 *OIDEli-zk*: Protecting Everlasting Privacy with ZKPs

In order to protect voters' sensitive information and to preserve everlasting privacy, we describe the final version of *OIDEli* where the voting server does not publish the ID token tok in clear-text. Instead, the server computes a ZKP that it has received a valid idP's signature, for a legitimate voter.

The signature received from the OpenID provider is of the form $\sigma := \mathsf{sign}_{\mathsf{idP}}(H)$ where $n_V = \mathsf{hash}(b, n_V^0)$ and:

$$H = \mathsf{sha}_2(\mathsf{json}(\mathsf{id}, \mathsf{sub}, \mathsf{pad}_{\mathsf{id}}(\mathsf{hash}(n_S \Vert n_V)), \mathsf{aud}))$$

In principle, the server could prove that it knows a valid signature σ and values $\mathsf{id}, \mathsf{sub}, n_S, n_V^0$ such that $\mathsf{id} \in \mathsf{ID}_{\mathsf{el}}$ and b is the ballot inside the signature. This would require to encode the verification of a signature in a ZKP. Instead, we remark that we can reveal H and leave the verification of the signature to be made publicly. Similarly, we can leave public the check that $n_V = \mathsf{hash}(b, n_V^0)$.

The voting server also needs to prove that it receives signatures for *distinct* users. This may be costly to prove in zero-knowledge. Instead, we notice that the OpenID protocol guarantees that sub is unique for each voter (EU). Hence we can reveal sub and let the eligibility verifiers verify that the sub are pairwise distinct.

Since we aim to reveal H but not the token in clear on the BB, we should be careful to not allow brute-force attacks where an attacker tries to construct H for each possible $\mathsf{id} \in \mathsf{ID}_{\mathsf{el}}$ until finding a matching H found on the BB. We remark that it suffices to keep one fresh nonce included in N private to thwart such attacks. So the voting server reveals $\mathsf{pubToken} = (\mathsf{sub}, n_V, H)$, keeps id and n_S secret, and produces a ZKP $\pi := \mathsf{ZK}_{\mathcal{S}}(\mathsf{privZK})(\mathsf{pubZK})$ for $\mathsf{privZK} := (n_S, \mathsf{id})$ and $\mathsf{pubZK} := (\mathsf{ID}_{\mathsf{el}}, \mathsf{pubToken})$ that guarantees the following statement $\mathcal{S}$ (aud is considered to be a constant):

"I know private n_S and id such that :
- $\mathsf{id} \in \mathsf{ID}_{\mathsf{el}}$ and (1)
- $H = \mathsf{sha}_2(\mathsf{json}(\mathsf{id}, \mathsf{sub}, \mathsf{pad}_{\mathsf{id}}(\mathsf{hash}(n_S \Vert n_V), \mathsf{aud})))$"

Hence, *OIDEli-zk* considers $\mathsf{eliProof} = \mathsf{eliProofZK}$ as defined in Table 2. Eligibility verifiers (auditors) check the ZKP and the signatures of each ballot using $\mathsf{checkEliProof} := \mathsf{checkEliProofZK}$. They verify in addition that all sub are pairwise distinct. We explain in Section 4 how to design an efficient circuit that can prove the statement $\mathcal{S}$.

As mentioned in Section 3.1, the ZKP π does not have to be computed right away (*e.g.,* before publishing b on the BB), checking the statement $\mathcal{S}$ is enough. Indeed, by completeness of the ZKP, the server is guaranteed it will be able to effectively do it later.

Security Claims. The *OIDEli-zk* protocol (implemented with eliProofZK from Table 2) ensures eligibility verifiability, as *OIDEli-id*. In addition, it hides potentially privacy-sensitive

$\mathsf{eliProofZK}(\mathsf{ID_{el}}, b, (n_V^0, n_S, \mathsf{tok}, \sigma)) := (b, \mathsf{pubToken}, n_V^0, \pi, \sigma)$ if :
- $\mathsf{tok} = \mathsf{json}(\mathsf{id}, \mathsf{sub}, N, \mathsf{aud})$ and $\mathsf{id} \in \mathsf{ID_{el}}$
- $N = \mathsf{pad_{id}}(\mathsf{hash}(n_S \| n_V))$ and $n_V = \mathsf{hash}(b, n_V^0)$
- $\mathsf{checkSign}(\sigma, H)$ is true with $H := \mathsf{sha}_2(\mathsf{tok})$
- $\mathsf{pubToken} := (\mathsf{sub}, n_V, H)$
- $\pi := \mathsf{ZK}_{\mathcal{S}}(n_S, \mathsf{id})(\mathsf{ID_{el}}, \mathsf{pubToken})$

$\mathsf{checkEliProofZK}(\mathsf{ID_{el}}, b, (\mathsf{pubToken}, n_V^0, \pi, \sigma))$ is true if :
- $\mathsf{pubToken}$ is of the form (sub, n_V, H)
- $n_V = \mathsf{hash}(b, n_V^0)$
- $\mathsf{checkZK}_{\mathcal{S}}(\pi, (\mathsf{ID_{el}}, \mathsf{pubToken}))$ is true
- $\mathsf{checkSign}(\sigma, H)$ is true

Table 2: Eligibility proof for *OIDEli-zk*. The voter's identity is protected by the ZKP proving the statement $\mathcal{S}$ specified in Equation (1). The eligibility verifiers must also verify that all the sub published on the BB are pairwise distinct.

voter's information and in particular the link between the ballot and the identity of the voter who interacted with the idP. This guarantees voters remain *anonymous* when the voting server and the idP are honest. We call this property *identity-hiding*. Similarly, the newly added ZKP prevents breaking everlasting privacy. We summarize the security claims in Table 4 (first two lines) presented in Section 5, where the claims are formally defined and proved.

Preservation of other properties. Our *OIDEli* protocols may affect other security properties of the underlying voting protocol. In particular, as discussed in Section 3.1, *OIDEli-id* may endanger everlasting privacy, while *OIDEli-zk* preserves it, thanks to the fact that by the zero-knowledge property of ZKPs, it does not add any link, even implicit, between the voter and their ballot. We note that *OIDEli-zk* still leaks the subject identifier of the voter sub. This could allow an attacker to identify that a voter has voted in several elections, but without knowing which one. For high-stake elections, one can assume that the RP registered to the idP would anyway be renewed (*e.g.,* with a new and clear webpage associated to the election), hence fresh subs would be used.

Our *OIDEli* protocols do not compromise vote privacy. If the underlying voting protocol guarantees vote privacy, it is still the case when used with our protocol since the latter does not use any private material of the voting protocol. Similarly, other verifiability properties enjoyed by the underlying voting protocol (*e.g.,* cast-as-intended, individual, and universal verifiability) are preserved when our protocol is used since the latter does not interfere with the verifiability checks of the underlying voting protocol.

These claims regarding security preservation are informal and could be false for evilly crafted protocols. Formally proving a composition theorem is out of scope of this paper. Instead, we substantiate our claims in Section 5 by also analyzing the security of *OIDEli* on top of the Helios-like protocol from Example 1.

On the other hand, we would like to highlight the fact that the treatment of receipt-freeness, vote-buying or coercion-resistance requires more care. Voting protocols designed to offer such advanced properties often make fine grain assumptions on the communication channels (*e.g.,* anonymous channels). They may require to hide revoting (*e.g.,* Civitas [23] or VoteAgain [43]). Moreover, the exact security definition of receipt-freeness or coercion-resistance typically depend on the voting protocol. Hence, in general, *OIDEli* may break receipt-freeness and coercion-resistance. Whether or not it is possible to adapt it strongly depends on the voting protocol.

4 Zero-Knowledge Proof Design

In this section we provide a complete ZKP design for attesting the knowledge of a valid ID token privately linking the voter to their ballot, more precisely the statement $\mathcal{S}$ (Equation (1)). We first refine $\mathcal{S}$ into a precise statement specification accounting for all constraints we inherit from using the OpenID protocol, which were abstracted away in Equation (1). We then provide a complete arithmetic circuit specification, implementing this statement with a best effort at minimizing the number of gates and constraints. Such circuits are re-usable for many ZKP systems, as discussed in Section 2.3. For our Proof-of-concept, we implemented this circuit in Plonky2 (see Section 6).

4.1 Statement Specification

Figure 2 depicts the statement $\mathcal{S}$ to prove, which is structured in four main parts that we define next.

Hash H computation. As mentioned in Section 3.1, we have no other choice than to use sha_2 to compute H, since this is what the idP will use to compute the ID token, as specified by [50] (see Section 2.3). Another constraint pertaining to the nature of the ID token is the Base64 encoding of its content before the signature, which results in the modified statement $H := \mathsf{sha}_2(\mathsf{base64}(\mathsf{tok}))$. (Note that tok is a JSON Web token, which is simply serialized before the Base64 encoding.)

Token tok *checks (*lookup_i*).* We must check that the serialization of tok contains the expected (private) nonce N (field `"nonce"`) and id (field `"email"` for Google as idP) and the public sub (field `"sub"`) and aud (field `"aud"`). The serialization of tok is a sequence of ASCII characters (each encoded as a byte) of the form:

$$\mathsf{tok} = \{..,\texttt{"sub":"}\langle\mathsf{sub}\rangle\texttt{"},..,\texttt{"email":"}\langle\mathsf{id}\rangle\texttt{"},..,\\ \texttt{"aud":"}\langle\mathsf{aud}\rangle\texttt{"},...,\texttt{"nonce":"}\langle N\rangle\texttt{"},..\}.$$

A naive approach to check that tok contains id in the `"email"` field would be to: (i) impose equality between each of the bytes of id (say of 20 bytes) with the bytes of tok at the right position, say starting at position 100, so checking $\mathsf{id} = \mathsf{tok}[100..120]$, and (ii) impose equality $\texttt{"email":"} = \mathsf{tok}[91..100]$, and (iii) $\texttt{"} = \mathsf{tok}[101]$ (note that the quote character " cannot be used in the JSON values). However, this

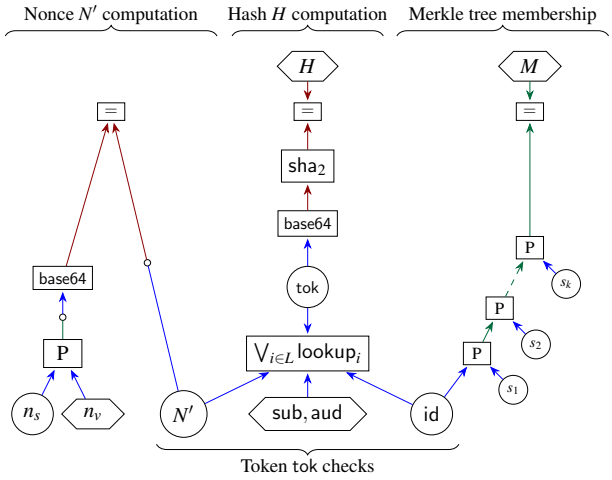

Figure 2: Abstract view of our ZKP circuit for statement $\mathcal{S}$. Large circular nodes represent private inputs while hexagonal ones denote public inputs. P stands for poseidon, L is the set of admissible lengths (in bytes) for id in $\mathsf{ID}_{\mathsf{el}}$. Each arrow represents a *group* of wires and its color indicates the type of data each wire encodes as a field $\mathcal{K}$ element (from the least to the most compact/efficient encoding): bit ($\mathbb{B}$) in red, byte ($\mathcal{B}$) in blue, and field element ($\mathcal{K}$) in green. The small circles denotes an encoding: *e.g.,* n wires encoding bytes are encoded into $n \times 8$ wires encoding bits.

statement and its encoding as a circuit would trivially leak the positions of this value in tok (here 100) and the length of id (here 20), which must remain private information.[2] Indeed, as is the case for emails, revealing $|\mathsf{id}|$ associated to a ballot can drastically reduce the *anonymity set* of associated voters.

To remedy this leakage, we must specify a statement (and later a circuit) that is independent of $|\mathsf{id}|$. In particular, the positions of the fields in tok (*e.g.,* of N) must not depend on id. We do so by letting $\mathsf{lookup} := \bigvee_{i\in L} \mathsf{lookup}_i$ where lookup_i enforces all the checks on tok using (i), (ii), and (iii) above, assuming the length of id is i and L is the set of all admissible lengths in bytes for $\mathsf{id} \in \mathsf{ID}_{\mathsf{el}}$. For our PoC, $\mathsf{ID}_{\mathsf{el}}$ was made of Gmail addresses and $L := [11, 50]$.

Another potential leak could be caused by the fact that the computation of H leaks $|\mathsf{tok}|$, which itself leaks $|\mathsf{id}|$. Therefore, we make sure $|\mathsf{tok}|$ is always independent of $|\mathsf{id}|$, and actually constant, by the means of $\mathsf{pad}_{\mathsf{id}}$ in N, included in tok. We let $\mathsf{pad}_{\mathsf{id}}(\cdot)$ be the function that adds $\mathsf{max}_{\mathsf{id}'\in L}|\mathsf{id}'| - |\mathsf{id}|$ times the character "." at the end, so that the sum $|\mathsf{id}| + |\mathsf{pad}_{\mathsf{id}}(\mathsf{hash}(\cdot,\cdot))|$ is constant and independent of id. Since all other field lengths are constant (given specific idP and RP), we thus obtain constant $|\mathsf{tok}|$ (426 bytes for our PoC). This way, all the positions of the fields in tok are at a constant position and a byte-to-byte check for the fields `nonce` (N), `aud` (aud), `email` (id), and `sub` (sub), can be enforced as explained above with the constraints (i) to (iii). The only exception is that we do not check the padding of N in tok, we solely check N', *i.e., N* without padding, is present in tok with the checks (i) and (ii) above, excluding check (iii).[3]

Remark 1. *Note that a simpler solution to the* $|\mathsf{id}|$ *leaking problem would be to first build a naive statement* $\mathcal{S}_i$ *(and circuit) leaking the length* $i = |\mathsf{id}|$ *and then takes a global statement* $\mathcal{S} := \bigvee_{i\in L} \mathcal{S}_i$*. However, this would force us to replicate* $|L|$ *times the* sha_2 *gadget in the circuit. This would come at an unreasonable high cost in terms of number of gates in the circuit and in terms of proving time (see Section 6.2). To reach reasonable proving time, we must limit ourselves to one* sha_2 *sub-circuit, we thus need* $\mathsf{pad}_{\mathsf{id}}(\cdot)$ *to make* $|\mathsf{tok}|$ *constant and the* $\vee_{i\in L}\mathsf{lookup}_i$ *optimization to allow adaptive* tok *parsing.*

Nonce N' computation. We previously defined N to be $\mathsf{pad}_{\mathsf{id}}(N')$ with $N' := \mathsf{hash}(n_S, n_V)$. We instantiate the function hash with poseidon [32] as it is optimized for ZKPs. Indeed, it requires much fewer gates than, *e.g.,* sha_2 (see Sections 2.3 and 6.2). Any secure hash function would be suitable though. Because N is then sent in an AuthReq and should thus be ASCII characters [50], the output of $\mathsf{poseidon}(n_S, n_V)$ needs to be encoded with Base64 (base64 gadget). We obtain $N' := \mathsf{base64}(\mathsf{poseidon}(n_S, n_V))$ of 44 bytes.

Merkle tree membership. The final part of the statement is the membership check of id in $\mathsf{ID}_{\mathsf{el}}$. As standard, we use a Merkle tree with the elements of $\mathsf{ID}_{\mathsf{el}}$ as leaves. The path from id to the root of the tree M is added to our circuit by successively hashing nodes on the path with their siblings $s_1, s_2, \ldots, s_k$, where k is bound by the depth of the Merkle tree. The root M becomes a public input while the path to id with the associated siblings s_i are kept private, effectively proving that $\mathsf{id} \in \mathsf{ID}_{\mathsf{el}}$ by revealing M only. We used poseidon as hashing algorithm to eventually optimize the circuit.

4.2 Arithmetic Circuit

To translate the statement specified in Section 4.1 into a proper arithmetic circuit specification, we shall specify how the various pieces of data are encoded as elements of the field $\mathcal{K}$ and decompose the required gadgets into sub-circuits.

Encoding data in $\mathcal{K}$ and converters. Initially, most inputs are given as sequences of bytes ($\mathcal{B}^*$). A compact encoding to project them onto $\mathcal{K}^*$ would be to encode as many bytes as possible in a field element (*e.g.,* 8 bytes in a 64-bit field elements in the case of Plonky2). However, not all operations on bytes (or even bits, $\mathbb{B}^*$) can be easily encoded as operations in $\mathcal{K}$ (gates) without a large number of gates. When intensive bitwise operations are required, as for computing sha_2, each bit of information shall be encoded as a field element so that bitwise operations can be easily encoded with a single

[2] Indeed, even though a ZKP does not reveal its private inputs, the statement and the circuit itself is public and can reveal some information.

[3] This is *w.l.o.g.* since the statement already proves tok has the expected length and thus so does N. The characters used for padding are irrelevant and not to be checked since the quote character " is forbidden in N.

or a few gates, trading some data density for lower circuit complexity. The same applies for operations on bytes.

In order to convert from $\mathcal{K}$ to $\mathcal{B}$ or from $\mathcal{B}$ to $\mathbb{B}$, the value is split into powers of 2 which are then grouped back.

Equality checks. Equalities in the circuit link two wires and enforce identical values. (They can be encoded with subtraction gates.) Note that lookup contains equality checks, thus the circuit has one output wire that should be checked to be 0.

base64 *gadget.* Encoding a sequence of bytes into Base64 implies splitting the sequence into a sequence of 6-bit chunks. Each 6-bit chunk is mapped to a Base64 character which is itself encoded over 8 bits (UTF8). Thus we decompose the base64 gadget into parallel b64chunk gadgets as depicted in Figure 3.

The b64chunk gadget must implement the conversion table given in Table 3. A naive implementation thereof would require 64 comparisons, one per entry in the conversion table. But since this table is composed of 5 ranges of contiguous output characters (*e.g.,* 0-25 is mapped to A-Z), the corresponding circuit can be simplified to 5 range checks as shown in Figure 4. In order to efficiently perform those checks, the 6 boolean wires are first combined into an element of the field $f \in \mathcal{K}$. For each range, f is (i) compared to the start and end of the range (resulting in either 0 or 1), and (ii) shifted to match the output range. *E.g.,* it is shifted by 65 for the first range because the character 'A', corresponding to input '0', is encoded by the number 65 in UTF8. The output of the comparison (i) and of the shift (i) are then multiplied, and the result of the 5 multiplications are summed. In short, the comparisons form a mask and only one of them can be non-zero for a value of f. Lastly, the output of the sum is converted back to a sequence of 8 bits.

lookup_i *gadget.* The sub-circuit lookup_i takes as input the ID token tok and the values it should contain, assuming $|\mathsf{id}| = i$. We made our circuit modular and robust to fields re-ordering since the specification [37] does not guarantee a fixed ordering, even though we observed no such re-ordering in practice with Google as idP. More precisely, our PoC adapts the circuit computation to the ordering used in a given tok, the ordering is not sensitive information so it can be leaked. The byte positions of the fields for sub, id, aud, and N are then computed based on the order of the fields and the assumed length of id. Given those positions, equalities between wires, encoding each one byte of information, can be added to the circuit lookup_i according to the checks (i), (ii), and (iii) specified in Section 4.1.

5 Security Analysis

In order to prove the security of the *OIDEli* protocols we conduct a security analysis using the ProVerif tool [17]. We prove that the two protocols ensure eligibility verifiability and *OIDEli-zk* additionally ensures identity-hiding. Finally, we instantiate our two protocols in the context of a Helios-like voting protocol and prove that *OIDEli-zk* ensures everlasting privacy and preserves the main security properties we expect from an e-voting protocol: vote secrecy, individual, and universal verifiability.

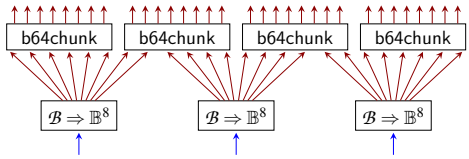

Figure 3: base64 circuit for Base64 encoding. Each arrow represents a single wire. Each b64chunk gadget has a 6-bit input and a 8-bit output, specified in Figure 4, following the conversion Table 3.

In ($\mathbb{B}^6$)	Out ($\mathbb{B}^8$)	UTF8	In ($\mathbb{B}^6$)	Out ($\mathbb{B}^8$)	UTF8
0	65	A	52	48	0
1	66	B	53	49	1
...	...	...	...	...	...
25	90	Z	61	57	9
26	97	a	62	45	-
27	98	b	63	95	_
...	...	...			
51	122	z			

Table 3: Base64 conversion table from In (6 bits) to Out (8 bits) with the corresponding output UTF8 character. Colors highlight the contiguous ranges of outputs used in our circuit (Figure 4).

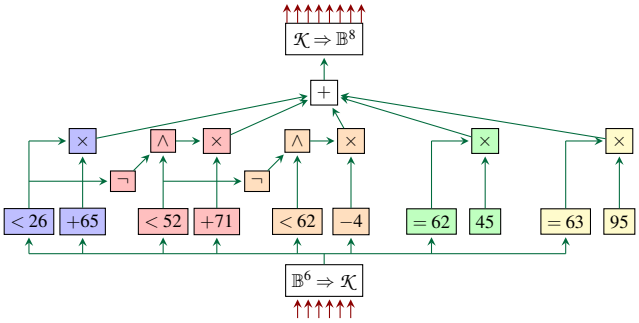

Figure 4: b64chunk circuit. Each of the sub-circuits that are eventually summed correspond to one contiguous range from Table 3.

5.1 ProVerif in a Nutshell

Two main approaches exist to formally prove the security of cryptographic protocols: computational and symbolic proofs. Computational proofs allow for more fine-grained models, where the adversary is any polynomial probabilistic Turing machine and the security of the protocol is reduced to the security of some well-known security assumptions such as computational Diffie-Hellman. In contrast, symbolic proofs explore more abstract models where the cryptographic primitives are assumed to behave as expected. This approach typically al-

lows to analyze more complex protocols and is more amendable to automation [10]. ProVerif [18] is a state-of-the-art automated prover in the symbolic setting. It has demonstrated its usefulness to find attacks in, fix, and finally prove secure widely deployed protocols such as TLS 1.3 [15], the Signal messaging protocol [39], or e-voting protocols [13,22,26]. In ProVerif, protocols are modeled in a variant of the applied pi-calculus: messages are abstracted with terms to focus on their functionality, *e.g.,* a ciphertext is perfectly hiding its plaintext as long as the attacker does not know the decryption key. Protocol agents are described with processes that model the exchanges of messages and checks thereon. Finally, the public communication channels are fully controlled by a Dolev-Yao attacker who can intercept, send, but also forge new messages.

Given some security goals, ProVerif automatically produce security proofs thereof, attacks if there are any, or does not terminate successfully (since the underlying problem is undecidable). For all the properties we analyzed, we were able to make ProVerif produce security proofs or find attacks.

5.2 Security Analysis of *OIDEli* and *OIDEli-zk*

We conduct a comprehensive security analysis of our two protocols. The results are presented in Table 4 and backs up the informal security claims stated in Section 3. More details about ProVerif and our models are given in Appendix B.

Specifically, we verify two properties, *eligibility verifiability* and *id-hiding*, in rich scenarios composed of an unbounded number of voters who may cast as many ballots as they want. We also consider an arbitrary number of (malicious) RP, one of them being the voting server.

Eligibility verifiability. The first property, *eligibility verifiability*, ensures that each ballot has been registered in the name of a legitimate voter. Moreover, to prevent *ballot stuffing*, it additionally ensures that at most one ballot can be accepted per legitimate voter.

To formally define the property, we follow the approach developed in *e.g.,* [22]. We first define an uninterpreted private symbol of function, $\mathsf{is\_eligible}(\mathsf{id})$, that is used whenever a legitimate voter id is registered. We then define three different events: $\mathsf{Corrupted}(\mathsf{id})$ is executed each time a legitimate but corrupted voter id is registered, $\mathsf{PublishBallot}(b,\Pi)$ is executed each time a ballot b and its corresponding proof Π is published onto the BB, and $\mathsf{Voted}(\mathsf{id},b)$ is executed each time a voter confirms their vote (*i.e.,* reveals n_V^0 in Figure 1). Intuitively, an event is merely an annotation in the trace, that records some specific step of the protocol. Given a trace tr and an event E, $E \in tr$ means that the event is executed in the trace.

Definition 1. A process P ensures *eligibility verifiability* if, for all execution trace tr,

$$\mathsf{PublishBallot}(b,\Pi) \in tr \Rightarrow \mathsf{GetId}(\Pi) = \mathsf{is\_eligible}(\mathsf{id}) \\ \wedge (\mathsf{inj\text{–}Voted}(\mathsf{id},b) \in tr \vee \mathsf{inj\text{–}Corrupted}(\mathsf{id}) \in tr)$$

where $\mathsf{GetId}(\Pi)$ returns the voter identity id occurring in Π.

The *injectivity* of the correspondence property (denoted by inj–before events in the conclusion) prevents ballot stuffing by ensuring that there is at most one matching $\mathsf{Voted}(\cdot,\cdot)$ or $\mathsf{Corrupted}(\cdot)$ per published ballot (see [18] for a formal definition of injective queries). In our security analysis, we prove that our protocols satisfies Definition 1 assuming everyone but the idP can be dishonest.

Id-Hiding. The second property, *id-hiding*, ensures that the protocol does not leak the identity of the voter even if all the agents, but the voting server and the idP (as they directly communicate with voters) are compromised. Intuitively, the attacker should not be able to distinguish whether a ballot is issued by Alice or a fresh unknown voter. This anonymity property is modeled, as usual in symbolic models [7,36] by an equivalence property called strong secrecy [16] (the hidden value being the voter id).

To formally define *id-hiding*, given an e-voting protocol, we note $Voter(\mathsf{id})$ the process that describes the behaviour of a given voter id. We note S the process that models the parallel composition of all the remaining processes describing the protocol under study (*e.g.,* the idP process, the voting server process, etc.). And we note $\approx$ the equivalence relation, *i.e.,* $P \approx Q$ holds when an attacker is not able to distinguish whether the P or the process Q is executed. (The exact notion of equivalence proved by ProVerif is defined in [18].)

Definition 2. An e-voting protcol $(S, Voter(\cdot))$ ensures *id-hiding* if the following equivalence property holds:

$$S \mid Voter(\mathsf{id}) \approx S \mid Voter(\mathsf{id}')$$

where id is the id of a legitimate voter known by the attacker, id' is a different id unknown from the attacker.

To compare with e-voting literature, one may notice that e-voting security analyses usually prefer the notion of *vote secrecy* [40] to the notion of anonymity to express privacy. Unfortunately, in our *OIDEli* protocols, the notion of *vote* is abstracted; only a notion of ballot is kept for the sake of generality. Therefore, *Vote secrecy* cannot be expressed in such a modular way. For our security analysis, we thus additionally verify *id-hiding* in our generic framework and prove that our protocols, when instantiated with a concrete e-voting protocol, preserve *vote secrecy* in Section 5.3. We thus first prove that *OIDEli-zk* always guarantees *id-hiding* assuming an honest voting server and idP (as they directly communicate with voters), all other agents can be dishonest. We then show that *OIDEli-id* violates *id-hiding*, for reasons we already discussed in Section 3.1. Finally, we prove that, when instantiated with an Helios-like e-voting protocol (see next section), our *OIDEli* protocols preserve *vote secrecy*.

	Voting protocol only	with *OIDEli-id*	with *OIDEli-zk*
eligibility verifiability	-	✓(<1s)	✓(1s)
id-hiding	-	✗(<1s)	✓(<1s)
With the Helios-like e-voting protocol			
individual verifiability	✓(<1s)	✓(<1s)	✓(<1s)
universal verifiability	✓(<1s)	✓(<1s)	✓(<1s)
vote secrecy	✓(<1s)	✓(10s)	✓(23s)
everlasting privacy	✓(<1s)	✗(11s)	✓(60s)

Table 4: Results of our security analysis (✓: proved, ✗: attack) with proof times (in seconds) obtained running ProVerif v2.05 on a Macbook Pro 14, M2 Pro, 32Go RAM. The ProVerif files used to conduct this analysis are provided as supplementary material [27].

5.3 Instantiating with a Helios-like Protocol

We foresee that our protocols will preserve the main security properties one may expect from an underlying e-voting protocol. To demonstrate this on a concrete example, we decided to prove the security of a Helios-like e-voting protocol implementing our *OIDEli* and *OIDEli-zk* extensions.

Assuming that the BB is publicly audited (*i.e.,* only valid proofs and signatures can be published), we prove: individual verifiability [41] when everyone but the considered voter can be dishonest, universal verifiability when everyone can be dishonest, and vote secrecy [40] when everyone but a threshold of the decryption trustees can be dishonest. The results are summarized in Table 4.

We prove that *OIDEli-zk* furthermore preserves practical everlasting privacy [8]. In this scenario, we assume that the attacker is away during the voting phase but records publicly available data to exploit it in the future when all the conditionally secure cryptographic primitives can be broken.

6 Proof of Concept

This section demonstrates the practicality of *OIDEli*. We first provide a full-fledged implementation for our ZKP design (from Section 4), distributed as a standalone Rust library, and an evaluation thereof showing proofs can be created and verified efficiently enough to be used in the largest nation-wide elections to date. We then provide a PoC implementation for our protocol *OIDEli-zk* and its integration on top of the state-of-the-art voting system Belenios [25].

6.1 ZKP Implementation

Our implementation to generate and verify eligibility proofs is available as the open source Rust library `oideli-zkp` available at [27]. It represents 3.3K lines of code with a well documented command line interface allowing direct access to its functionalities. Each gadget of the circuit is implemented in a separate module with dedicated and comprehensive unit tests following software engineering best practices, making the project accessible to anyone interested in inspecting and understanding it.

The choice of Rust as a programming language comes from the decision of using Plonky2 as ZKP system (explained in Section 2.3). The interface of Plonky2 is very close to the level of arithmetic circuits, with a `CircuitBuilder` to add `Targets` (*i.e.,* wires) and connect them to various gates, allowing us to have low level control and implement specific functions (*e.g.,* the base64 encoding). We use the standard Plonky2 configuration, notably with a 64-bit Goldilocks field, which is claimed to provide 100-bit security [53]. Our ZKP library is modular and can accommodate any idP, ID token fields, and ordering thereof. When instantiated to Google as idP, it yields the following message lengths (in bytes): $|\mathsf{tok}| = 426$, $|N'| = 44$, $|H| = |n_S| = |n_V| = 32$, $|\mathsf{sub}| = 21$, and $L = [11, 50]$.

6.2 ZKP Evaluation

We benchmark our ZKP implementation `oideli-zkp` to evaluate its practicality in the context of a large scale election. To this end, we consider the 2022 legislative election for French nationals living abroad, which had 1.5M eligible voters and where 300K of them voted online over a period of 6 days [3] for the first round [28] (out of two rounds). The other largest elections have been held with comparable figures (310K ballots cast in 7 days in Estonia [5]; 280K ballots cast in 12 days in Australia [19]). We estimate the cost of organizing such an election in terms of hardware necessary to generate the ZKPs, which is the only significant additional cost induced by using our *OIDEli-zk* protocols (see Section 6.3).

As already discussed, ballots of the underlying voting protocol can be published on the BB independently of their proof of eligibility. The voting server is only required to finish the computation of the ZKPs by the end the tally phase. In the case of our target election, this implies that the voting server is able to generate at least 300K ZKPs in 6 days, corresponding to an average rate of 2.01K proofs per hour.

Methodology. To benchmark `oideli-zkp`, we generate as many ZKPs as possible in a controlled environment for a defined amount of time. Our experiments were run on a single machine with two AMD EPYC 7F52 16-Core processors running at a maximal frequency of 3.9 GHz with 512 GB of RAM. We run the experiment on independent threads, each locked on a single CPU core. This allows evaluating the scalability of the computation and predict performances on machines with more cores. Lastly, since the circuit solely depends on the election settings, we build the circuit once-for-all for an election. Our `oideli-zkp` library then takes this circuit blueprint and private and public data as input to compute each of the ZKPs. We use a Merkle tree of depth 21 containing up to 2M eligible voters' identities. Google

was used as idP and GMail addresses as id, which yields ID tokens of 1014 bytes whose tok is 426 bytes long.

Results. Table 5 shows the performances of `oideli-zkp`, first running on a single core and then scaled up on 32 cores. The proof rate scales almost proportionally to the number of CPU cores. We also note that the verification time is, as expected, very low. As suggested by another experiment on 4 cores of a mid-range laptop, an auditor can verify the 300K eligibility proofs of our target election in under 20 minutes. Aside from that, we measure a size of 200 KB per ZKP and a maximum RAM consumption of 5GB per proof computation.

	1 core	32 cores
Proof rate (scaling factor)	$43.8\,h^{-1}\,(1)$	$1347.2\,h^{-1}$ (30.8)
Mean proof time	$82.1\,s$	$85.5\,s$
Mean verification time	$8.68\,ms$	$9.14\,ms$

Table 5: Performances of `oideli-zkp` on 1 and 32 cores.

Cost estimation. According to Table 5, twice the performance of the machine we used would be enough to process all the ballots of our target election that requires a $2010\,h^{-1}$ proof rate. The machine we used costs ca. 14K €, which yields a total first investment of 28K €, which is likely to be marginal compared to the deployment cost for such a large election. Renting CPUs for computing the ZKPs, *e.g.,* on AWS, is possible and much cheaper (ca. 500 €) but discouraged for such important, nation-wide elections since it could put everlasting privacy at risk. Indeed, the machines in the cloud would process data linking the ballot with their voters' id, thus introducing a new trust assumption for everlasting privacy, *i.e.,* the cloud provider. However, it seems to be the easiest solution for moderate-stake elections.

Larger elections. To evaluate the practicality of *OIDEli* in a larger election, we estimate its cost for casting 160M votes, such as in the US 2020 presidential primary elections[4]. Assuming a timeframe of 30 days[5], the required proof rate is $(160\times10^6)\div(30\cdot24)=2.22\times10^5$ proof per hour. Using the same price assumptions as above, the cost of purchasing hardware necessary to reach this proof rate amounts to ca. 2.3M €, which is reasonable when considering a $2B lower bound of organizing such an election[6]. This cost could be amortized by using the hardware for other purposes after the election, or alternatively CPUs could be rented at a much lower cost (ca. 44K € on AWS).

[4] https://history.house.gov/Institution/Election-Statistics/Election-Statistics/

[5] Some states opened polling stations several months in advance (https://www.ncsl.org/elections-and-campaigns/2020-state-primary-election-dates)

[6] https://electionlab.mit.edu/sites/default/files/2022-05/TheCostofConductingElections-2022.pdf

Sub-circuits evaluation. Since the generation of ZKPs is the only non-negligible additional cost induced by using *OIDEli*, we followed a best-effort approach to optimize the proving time while keeping the hardware requirements reasonable, as previously established. To identify which part of the circuit is the most complex and needs the most computing power, we compare the proving time for sha_2 to the overall proving time. The proof is for a pre-image of 671 bytes, which is the size of the sha_2 input in `oideli-zkp`, *i.e.,* the size of the Base64 encoding of tok and its header. Using the exact same machine and setup, the proving time of one core proving sha_2 only is 78.7 s which represents 96% of the proving time for the full ZKP computed by `oideli-zkp`.

In order to witness the need for the $\vee_{i\in L}\mathsf{lookup}_i$ gadget optimization and the use of $\mathsf{pad}_{\mathsf{id}}(\cdot)$ (see Remark 1), we built a circuit containing a disjunction of k sha_2 computations. Without the optimization, we would need to set $k=|L|$, that is $k=39$ for our PoC, yielding a proving time as long as 2784 s for a single proof instead of 78 s. We provide in Appendix A a more comprehensive comparison of the different circuits costs in terms of gates.

6.3 Full-fledged Proof of Concept

To demonstrate the practicality and the usability of *OIDEli*, we implemented it on top of the Belenios voting protocol and we ran a few mock elections. The source code of our PoC is publicly available [27] and made easily reproducible via a docker container. In particular, it is possible to set up (experimental) elections using our code.

The integration of *OIDEli* in the voting flow results in the addition of two steps for the voter. The voter first builds a ballot and sends it to the voting server, exactly as in the underlying voting protocol. The voting client then automatically performs the nonce construction with the voting server and redirects the voter to the authentication page of the idP. Upon successful authentication, the voting server queries the ID token from the idP and forwards it to the voting client. Finally, the voter is asked to confirm that the id from the ID token is correct (here the email), then revealing the nonce commitment n_V^0 to the voting server and concluding the role of the voter in the protocol. The voting server then proceeds with the rest of the underlying voting protocol checks and computations (here for Belenios, it notably publishes the cast ballot on the BB). Additionally, it computes and publishes the proof of eligibility at a later time.

The impact on usability thus seems marginal, since we only require voters to log-in at the idP, an operation they are used to perform in other contexts (*e.g.,* login to some service using their Google account). No costly computations are required on the voting client-side. The only additional costly computation is on the server-side with all the ZKPs. We already established in Section 6.2 that those can be computed continuously throughout the election with affordable hardware.

Acknowledgments

We are grateful to Geoffroy Couteau who helped us with useful insights about ZKPs and zk-SNARKs. We also thank Benjamin Voisin for his preliminary implementation of a ZKP with Plonky2 during his bachelor internship.

References

[1] Distributed randomness beacon. https://drand.love/.

[2] iGov working group. https://openid.net/wg/igov/.

[3] Results of the first round of the 2022 legislative french elections. https://www.diplomatie.gouv.fr/fr/services-aux-francais/voter-a-l-etranger/resultats-des-elections/article/elections-legislatives-resultats-du-1er-tour-pour-les-francais-de-l-etranger, 2022.

[4] Decision n° 2022-5813/5814 AN of the constitutional council (in french). https://www.conseil-constitutionnel.fr/decision/2023/20225813_5814AN.htm, 2023.

[5] Statistics about internet voting in estonia. https://www.valimised.ee/en/archive/statistics-about-internet-voting-estonia, 2023.

[6] Ben Adida. Helios: Web-based open-audit voting. In *USENIX Security Symposium*, 2008.

[7] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *Computer Security Foundations Symposium (CSF)*. IEEE, 2010.

[8] Myrto Arapinis, Véronique Cortier, Steve Kremer, and Mark Ryan. Practical everlasting privacy. In *International Conference on Principles of Security and Trust*. Springer, 2013.

[9] Foteini Baldimtsi, Konstantinos Kryptos Chalkias, Yan Ji, Jonas Lindstrøm, Deepak Maram, Ben Riva, Arnab Roy, Mahdi Sedaghat, and Joy Wang. zklogin: Privacy-preserving blockchain authentication with existing credentials. *arXiv preprint arXiv:2401.11735*, 2024.

[10] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *Symposium on Security and Privacy (S&P)*. IEEE, 2021.

[11] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In *International colloquium on automata, languages, and programming (ICALP)*, 2018.

[12] David Bernhard, Véronique Cortier, David Galindo, Olivier Pereira, and Bogdan Warinschi. SoK: A comprehensive analysis of game-based ballot privacy definitions. In *Symposium on Security and Privacy (SP)*. IEEE, 2015.

[13] David Bernhard, Véronique Cortier, Pierrick Gaudry, Mathieu Turuani, and Bogdan Warinschi. Verifiability analysis of CHVote. *Cryptology ePrint Archive*, 2018.

[14] David Bernhard, Véronique Cortier, Olivier Pereira, Ben Smyth, and Bogdan Warinschi. Adapting helios for provable ballot secrecy. In *European Symposium on Research in Computer Security (ESORICS)*, 2011.

[15] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the tls 1.3 standard candidate. In *Symposium on Security and Privacy (S&P)*. IEEE, 2017.

[16] Bruno Blanchet. Automatic proof of strong secrecy for security protocols. In *Symposium on Security and Privacy (S&P)*. IEEE, 2004.

[17] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends in Privacy and Security*, 1(1-2):1–135, 2016.

[18] Bruno Blanchet, Vincent Cheval, and Véronique Cortier. Proverif with lemmas, induction, fast subsumption, and much more. In *Symposium on Security and Privacy (S&P)*. IEEE, 2022.

[19] Ian Brightwell, Jordi Cucurull, David Galindo, and Sandra Guasch. An overview of the iVote 2015 voting system. New South Wales Electoral Commission, Australia, 2015. https://elections.nsw.gov.au/getmedia/4279ab0e-5db0-451e-9e3d-87d81ed82d9c/overview-of-the-ivote-2015-voting-system.pdf.

[20] ZKProof. Ed. by Daniel Benarroch, Luís Brandão, Mary Maller, and Eran Tromer. ZKProof community reference (version 0.3), 2022. https://docs.zkproof.org/reference.pdf.

[21] Pyrros Chaidos, Véronique Cortier, Georg Fuchsbauer, and David Galindo. BeleniosRF: A non-interactive receipt-free electronic voting scheme. In *Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[22] Vincent Cheval, Véronique Cortier, and Alexandre Debant. Election verifiability with proverif. In *Computer Security Foundations Symposium (CSF)*. IEEE, 2023.

[23] Michael R Clarkson, Stephen Chong, and Andrew C Myers. Civitas: Toward a secure voting system. In *Symposium on Security and Privacy (S&P)*. IEEE, 2008.

[24] Véronique Cortier, David Galindo, Ralf Küsters, Johannes Müller, and Tomasz Truderung. SoK: Verifiability notions for e-voting protocols. In *Symposium on Security and Privacy (S&P)*. IEEE, 2016.

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

[26] Véronique Cortier and Cyrille Wiedling. A formal analysis of the norwegian e-voting protocol. *Journal of Computer Security*, 25(15777):21–57, 2017.

[27] Véronique Cortier, Alexandre Debant, Anselme Goestchmann, and Lucca Hirschi. Supplementary material, 2024. https://gitlab.inria.fr/oideli/oideli-artifact.

[28] Alexandre Debant and Lucca Hirschi. Reversing, breaking, and fixing the french legislative election e-voting protocol. In *USENIX Security Symposium*, 2023.

[29] Mirco Richter et al. The MoonMath Manual to zk-SNARKs, 2022. https://github.com/LeastAuthority/moonmath-manual.

[30] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

[31] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster Zero-Knowledge for boolean circuits. In *USENIX Security Symposium*, 2016.

[32] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for Zero-Knowledge proof systems. In *USENIX Security Symposium*, 2021.

[33] Thomas Haines, Rafieh Mosaheb, Johannes Müller, and Ivan Pryvalov. Sok: Secure e-voting with everlasting privacy. *Proc. Priv. Enhancing Technol.*, 2023(1):279–293, 2023.

[34] Sven Heiberg, Tarvi Martens, Priit Vinkel, and Jan Willemson. Improving the verifiability of the estonian internet voting scheme. In *Electronic Voting (E-Vote-ID)*. Springer, 2017.

[35] Ethan Heilman, Lucie Mugnier, Athanasios Filippidis, Sharon Goldberg, Sebastien Lipman, Yuval Marcus, Mike Milano, Sidhartha Premkumar, and Chad Unrein. Openpubkey: Augmenting openid connect with user held signing keys. *Cryptology ePrint Archive*, 2023.

[36] Lucca Hirschi, David Baelde, and Stéphanie Delaune. A method for verifying privacy-type properties: the unbounded case. In *Symposium on Security and Privacy (S&P)*. IEEE, 2016.

[37] Michael B. Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). RFC 7519, May 2015.

[38] Aggelos Kiayias and Moti Yung. Self-tallying elections and perfect ballot secrecy. In *Public Key Cryptography (PKC)*, 2002.

[39] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *European symposium on security and privacy (EuroS&P)*. IEEE, 2017.

[40] Steve Kremer and Mark Ryan. Analysis of an electronic voting protocol in the applied pi calculus. In *European Symposium on Programming (ESOP)*. Springer, 2005.

[41] Steve Kremer, Mark Ryan, and Ben Smyth. Election verifiability in electronic voting protocols. In *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2010.

[42] Ralf Küsters, Johannes Müller, Enrico Scapin, and Tomasz Truderung. sElect: A lightweight verifiable remote voting system. In *Computer Security Foundations Symposium (CSF)*. IEEE, 2016.

[43] Wouter Lueks, Iñigo Querejeta-Azurmendi, and Carmela Troncoso. Voteagain: A scalable coercion-resistant voting system. In *USENIX Security Symposium*, 2020.

[44] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexander Frolov, Tyler Kell, Tyrone Lobban, Christine Moy, Ari Juels, and Andrew Miller. Candid: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability. In *IEEE Symposium on Security and Privacy (SP'21)*. IEEE, 2021.

[45] Johannes Mueller and Tomasz Truderung. Caised: A protocol for cast-as-intended verifiability with a second device. In *EvoteID*, 2023.

[46] Celer Network. The Pantheon of Zero Knowledge Proof Development Frameworks, 2023. https://blog.celer.network/2023/08/04/the-pantheon-of-zero-knowledge-proof-development-frameworks/.

[47] S. Palladino. Sign in with google to your identity contract. https://forum.openzeppelin.com/t/sign-in-with-google-to-your-identity-contract-for-fun-and-profit/1631, 2019.

[48] Peter Ryan, Peter Roenne, and Simon Rastikian. Hyperion: An enhanced version of the Selene end-to-end verifiable voting scheme. In *EvoteID*, 2021.

[49] Peter Ryan, Peter Rønne, and Vincenzo Iovino. Selene: Voting with transparent verifiability and coercion-mitigation. In *VOTING*, 2016.

[50] Natsuhiko Sakimura, John Bradley, Mike Jones, Breno De Medeiros, and Chuck Mortimore. Openid connect core 1.0. *The OpenID Foundation*, 2014. https://openid.net/specs/openid-connect-core-1_0.html.

[51] Sladuca. Sha256 prover comparison, 2022. https://github.com/Sladuca/sha256-prover-comparison.

[52] Swiss Post. e-voting system. https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation.

[53] Polygon Zero Team. Plonky2: Fast recursive arguments with PLONK and FRI, 2022. https://github.com/0xPolygonZero/plonky2/blob/136cdd053f2175134cddc61abc587f1862e76921/plonky2/plonky2.pdf.

[54] Justin Thaler et al. Proofs, arguments, and zero-knowledge. *Foundations and Trends® in Privacy and Security*, 4(2–4):117–660, 2022.

[55] Riad S Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zksnarks without trusted setup. In *Symposium on Security and Privacy (S&P)*. IEEE, 2018.

[56] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Annual International Cryptology Conference (CRYPTO)*. Springer, 2019.

A Appendix Sub-circuit Costs

We compare the cost of the different sub-circuits using the gate count metric reported by Plonky2.[7] Table 6 displays the gate count per component of our final circuit. We can see that sha_2 is the costliest part of the circuit, but optimizing it is out of the scope of this work, and the rest of the circuit represents less than 10% of the gates. Table 7 shows the infeasibility of replicating the sha_2 sub-circuit, which would be needed without the lookup_i optimization discussed in Section 4.

[7] Note that in this gate count, a gate can correspond to more than one arithmetic operation: with our settings an `ArithmeticGate` is translated into 20 constraints and a `PoseidonGate` into 123 constraints.

Gadget	Gate count
sha_2	64047
base64 (ID token)	3465
lookup ($39 \times \mathsf{lookup}_i$)	1929
base64 (N)	277
Merkle tree	29
poseidon (N)	8
conversions	457
Total	70212

Table 6: Number of gates in the Plonky2 circuit (before blinding or padding). A Merkle tree of depth 21 was used.

Number of sha_2	Number of gates	Proof time	
		1 core	32 cores
1	$64.1K$	$78.7s$	$86.2s$
2	$128.2K$	$165.5s$	$188.8s$
4	$256.5K$	$324.2s$	$397.9s$
39	$2.50M$	$2784.0s$	*out of mem.*

Table 7: Number of gates and average proof time (over 1 hour) for disjunctions of sha_2 equalities. Each sha_2 is applied on 671 bytes, which is the size of the sha_2 input in the benchmarked circuit, *i.e.,* the Base64 encoding of tok and its header. The last line corresponds to the necessary disjunction for voter ids up to 50 characters.

B How to Model in ProVerif?

ProVerif is an automatic protocol analyzer relying on a symbolic model. In this section we provide a gentle introduction to ProVerif in order to help non-familiar readers understand the scope of our security analysis. A comprehensive description of ProVerif syntax and semantics is available in [18].

Messages. In symbolic models, messages are abstracted with terms. This modeling choice allows to focus on the functional semantics of the cryptographic primitives, and abstract away the implementation details for instance.

For example, a randomized asymmetric encryption scheme is modelled by a function symbol $\mathsf{aenc}(.,.,.)$ and the term $\mathsf{aenc}(pk,m,r)$ represents the encryption of message m, using the public key pk, and the randomness r. Similarly, we can model the corresponding decryption algorithm using another function symbol: $\mathsf{adec}(sk,c)$ is the application of the decryption algorithm to the message c using the secret key

sk. Finally, we can now model the two main functionalities of an asymetric encryption scheme: (i) one can derive a public key from a secret key, (ii) applying the decryption algorithm to a valid ciphertext allows to get the plaintext. To model (i), we define a new function symbol, $\mathsf{pubKey}(\cdot)$, that will be used to define the public key corresponding to a private key. To model (ii) we finally define an equation: $\mathsf{adec}(sk, \mathsf{aenc}(\mathsf{pubKey}(sk), m, r)) = m$.

One can similarly model a digital signature scheme: $\mathsf{sign}(sk, m)$ models the signature of message m with private signing key sk. The public key can be derived from the secret key with the function symbol $\mathsf{pubSKey}(sk)$, and finally the verification algorithm is modelled by an equation: $\mathsf{checkSign}(\mathsf{pubSKey}(sk), m, \mathsf{sign}(sk, m)) = true$.

Roles. ProVerif relies on a process algebra to model protocol roles. The goal here is not to provide the complete grammar of processes. Instead, we will illustrate it through an excerpt of our model of the *OIDEli* protocol.

Consider the role of the idP: it first receives an authentication request from the EU (the voter), authenticates the voter, and confirms the authentication to the RP by sending an authorization token that allows the RP to contact the idP and obtain an ID token containing the EU information.

In the ProVerif model we again abstract the implementation and/or network details and focus on the flow of messages between the different participants. For sake of simplicity, we sometimes abstract some exchanges assuming that the exchange is correctly implemented so that an attacker would be unable to tamper the communication (*e.g.,* when using a TLS channel, but it is out of the scope of this model). Specifically, we decided to model the idP as follows:

```
ID_provider(login,pwd) =
  in(cIdP(login,pwd),(=REQ,aud,x_N));
  get idP_UserInfo_DB(=aud,=login,=pwd,xId,xSub) in (
    let data = (x_N,x_sub,x_id,x_aud) in
    new rIdT;
    let sigma = sign(ssk(idP),sha256(data),rIdT) in
    let idTokenJWT = (data, sha256(data), sigma) in
    let auth_token = authorize(idToken) in
    out(cIdP(login,pwd),(RESP, auth_token)).
```

We assume the idP, when called with some EU credentials (`login,pwd`) is waiting an input message to open a session with an EU authenticated with a login and a password. This communication must happen on a specific channel $cIdP(login, pwd)$ that depends on the two credentials and the message must contain a tag REQ and a payload that will contain the identifier of the RP (`aud`) the voter wants to authenticate to and a value that will be the nonce N in our *OIDEli* protocol. This first step is modeled by the input action:

$$\mathsf{in}(cIdP(login, pwd), (=\mathsf{REQ}, aud, x_N)).$$

Once the first message is received, the idP can look for the data associated to this user. This is modeled through a look up in an internal table (intuitively a database table) $idP\_UserInfo\_DB$ which allows to retrieve the identity xId and the subject identifier $xSub$ associated to the user for this RP. It prepares the ID token and the authorization tokens, the latter being sent back to the user. This final communication is modeled with the output action

$$\mathsf{out}(cIdP(login, pwd), (\mathsf{RESP}, auth_{token})).$$

In parallel of this process, we will define other processes that model the roles of the other participants. For instance, a process $Server()$ will be defined to model the role of the voting server and a process $Voter(\mathsf{id})$ to model the role of a voter. In the $Voter(\cdot)$ process, there are output and input actions on the channel $cIdP(login, pwd)$ that correspond to the communication occurring in the idP role. All our models with detailed explanations can be found in [27].