USENIX Security '24 Artifact Appendix: Election Eligibility with OpenID: Turning Authentication into Transferable Proof of Eligibility

Véronique Cortier Alexandre Debant Anselme Goetschmann Lucca Hirschi

Inria, CNRS, Université de Lorraine, France

A Artifact Appendix

A.1 Abstract

To complement the Security Analysis and the Proof of Concept sections of our paper on transferable proofs of eligibility, we provide a set of machine-checkable symbolic proofs as well as a proof-of-concept implementation of our protocol. The artifact contains three parts:

1. ProVerif files modeling our protocol and steps to reproduce our mechanized security analysis of the *OIDEli* protocols with ProVerif,
2. a Rust implementation of the ZKPs required by *OIDEli-zk* along with some benchmarks, and
3. a full-fledged proof-of-concept implementation of *OIDEli-zk* integrated in the Belenios voting system.

A.2 Description & Requirements

The three parts composing the artifact can be considered independently.

ProVerif files: Symbolic proofs that can be verified using the ProVerif tool and ensuring that the *OIDEli* protocols guarantee the claimed security properties.

`oideli-zkp` library and benchmarks: A Rust library implementing the ZKP design specified in our paper. It is accompagnied by some benchmarks to evaluate its performance.

PoC integration in Belenios: A patch of the Belenios voting system to integrate the *OIDEli-zk* protocol, using `oideli-zkp` to generate ZKPs and Google as OpenID provider. The proof-of-concept is bundled as a docker container allowing simple execution.

A.2.1 Security, privacy, and ethical concerns

Since the experiments are run locally, none of the three parts of the artifact raise any security, privacy or ethical concerns. We only note the need for the creation of an OpenID service on Google Cloud, which could be abused if the corresponding client id or secret was published.

A.2.2 How to access

The latest version of the artifact is accessible on the following Git repository: https://gitlab.inria.fr/oideli/oideli-artifact/-/releases/v1.0.

A.2.3 Hardware dependencies

ProVerif files: No particular requirement, the proofs can be verified on any standard laptop.

`oideli-zkp`: The execution of the benchmarks requires a notable amount of memory. For the experiment described below that runs on 32 cores, 160 GB of RAM are necessary, corresponding to 5 GB per core. Further, we used a machine equiped with two AMD EPYC 7F52 16-Core processors running at a maximal frequency of 3.9 GHz.

PoC: A single ZKP is generated during the experiment, requiring around 5 GB of memory.

A.2.4 Software dependencies

ProVerif files: The version 2.05 of the ProVerif tool has to be installed to verify the proofs. Instructions to install the tool can be found on the following page: https://bblanche.gitlabpages.inria.fr/proverif/.

`oideli-zkp`: A Rust environment is required to build the library and run the benchmarks. The version 1.76.0-nightly of the toolchain needs to be installed with the following commands.

```
rustup toolchain install nightly-2023-12-01
rustup default nightly-2023-12-01
```

PoC: The execution of Belenios relies on docker as container platform. Additionally, packages are downloaded from the opam package manager (https://opam.ocaml.org/) during the build of the docker image.

The experiments were successfully run on Debian 12 and Fedora 40.

A.2.5 Benchmarks

No additional data is needed to run the experiments contained in this artifact.

A.3 Set-up

A.3.1 Installation

We refer to A.2.4 for the installation of the required environment and to the **Preparation** step of each experiment for the remaining set-up.

A.3.2 Basic Test

ProVerif files: The first ProVerif file can be executed with the following command and should result in two `true` queries.

```
proverif -lib default.pvl oideli_id-
    eligibility-and-others.pv
```

`oideli-zkp`: The following command should print the help of the `prove` command provided by `oideli-zkp`:

```
cargo run --release prove --help
```

PoC: Once the docker container is built, the following command should start a local instance of Belenios accessible from :

```
docker run --network host  --name belenios-
    oideli --rm -it belenios-oideli
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our *OIDEli* protocols ensures the *eligibility verifiability* and *id-hiding* properties as described by experiment (E1) in Section 5.2 and illustrated in Table 4 of our paper.

(C2): Our `oideli-zkp` Rust library can generate above 40 proofs per hour and per core on hardware specified in Appendix A.2.3. This is shown by experiment (E2) described in Section 6.2 of our paper and with results are reported in Table 5.

(C3): *OIDEli-zk* can be intergrated in a state-of-the art voting system using an existing OpenID provider to ensure eligibility verifiability. In particular, this can be done for the Belenios voting system and Google as voting system, as described by experiment (E3) in Section 6.3 of our paper.

A.4.2 Experiments

(E1): ProVerif proofs [10 human-minutes + 5 compute-minutes]: verify the symbolic proofs of the *OIDEli* protocols using the ProVerif prover.
Preparation: Navigate to the `proverif-files` directory in a local copy of the artifact and make sure ProVerif is installed as detailed in A.2.4.
Execution: Run ProVerif for each file with the following command:

```
proverif -lib default.pvl <file.pv>
```

For more detailed instructions, refer to the README in the corresponding directory.
Results: Each ProVerif run is concluded by a "Verification summary" indicating which queries could be verified or not. The README file details the expected output for each ProVerif file to obtain the results reported in Table 4 of our paper.

(E2): `oideli-zkp` benchmarks [5 human-minutes + 5 compute-hours + 160GB RAM]: evaluate the performance of `oideli-zkp` by generating ZKPs on test data.
Preparation: Install the Rust environment described in A.2.4 to execute the benchmarks. Navigate to the `oideli-zkp` directory in a local copy of the artifact.
Execution: The benchmark generating ZKPs on 32 cores for 5 hours can be executed with the following command:

```
PARALLELISM=32 RUNNING_TIME_MINUTES=300 cargo
    run --release --example parallel_proofs
```

When executing the benchmark on a machine with less memory, the `PARALLELISM` parameter allows to perform a similar execution but without using as many workers, each needing up to 5GB of RAM. More details about the benchmark execution are given in the README_benchmarks file.
Results: At the end of the benchmark execution, the number of completed proof generation is reported. This number was 6736 in our case, yielding a proof rate of 1347.2 proof per hour. It might vary depending on the hardware and other condition of the environment but should be similar for hardware close to the one described in A.2.3.

(E3): PoC integration of *OIDEli-zk* in Belenios [20 human-minutes + 2 compute-minutes]: cast a ballot in a test election on Belenios using Google as OpenID provider.
Preparation: The README of the artifact provides detailed instructions on how to prepare the execution of this proof-of-concept implementation. On one hand, the docker container with Belenios and `oideli-zkp` has to be built, which might take a few minutes. On the other, an OpenID service has to be set up in the Google Cloud console and a Gmail address has to be added as test user.
Execution: Once the docker container is running, a test election can be created with via the command line, as described in the README. A ballot can then be cast by the voter with the test Gmail address. Finally, the verifiability checks can be executed to ensure that the eligibility proof in the cast ballot is valid.
Results: The steps above should be successful and the raw ballot visible in the ballot box should contains a `oideli_data` field with an eligibility proof `eliproof`.

A.5 Notes on Reusability

ProVerif files: Beyond proofs for the protocols in scope, the ProVerif files provide an example of protocol modeling which can be useful to produce symbolic proofs in other similar contexts.

`oideli-zkp`: The Rust library is accompagnied by 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.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.