29 July 2022 Study · Privacy & Crypto · Digital Identity · 9 min read

Privatheitsbewahrende Trust Policies

The full text is currently only available in the Deutsch version. A translation will follow.

Archived. This content is no longer updated. It remains available for reference.

Source https://technology2.a-sit.at/en/publications/privatheitsbewahrende-trust-policies/

1. The SP uses the zkTPL compiler to transform a TPL policy into a presentation request that enables a privacy-preserving authentication. This presentation request is send to the User.
2. For demonstration purposes the SP generates the cryptographic material required for the process. In practice, this can also be done by a trusted third party (TTP) or using other processes like MPC.
3. The User retrieves the cryptographic material and uses the zkTPL client to answer the presentation request by generating a verifiable presentation. Doing so they generate a proof that their attributes fulfill the requirements stated in the SP's policy. This presentation is send back to the SP.
4. After receiving the presentation, the SP uses the zkTPL verifier to ensure that the proof is valid and that the policy is fulfilled.

Demo setup

Before performing the demonstration process on your own, you need to setup the zkTPL tools as well as the ZoKrates zero-knowledge toolbox. As an alternative, jump to the next section and inspect the output of our demonstration run.

- Setup JDK and Maven

1

- Use pre-compiled `zkTPL-CLI` (e.g., `zkTPL_cli-3.0.0-SNAPSHOT.jar`)
- or: Compile and Install zkTPL components:

– `zkTPL-Compiler` → `mvn compile install`

– `zkTPL-Interpreter` → `mvn compile install`

– `zkTPL-ATV` → `mvn compile install`

– `zkTPL-CLI` → `mvn compile package`

- Install ZoKrates
- Author a TPL or SSI-TPL policy, and add a privacy-preserving predicate (or use provided example)
- use `zkTPL-CLI` (see below)

Demo run

Setup zkTPL-CLI

Setup paths to the CLI binary and configure `$USER` and `$VERIFIER` with their respective working directory to simulate the actors. Specify the relevant policy. For demonstration purposes the transaction data send is pre-generated and also configured here. The transaction contains metadata and business logic data, and is send alongside of the presentation to the SP by the User.

```
CLI="../zkTPL-CLI/target/zkTPL_cli-3.0.0-SNAPSHOT.jar"

mkdir verifier 2> /dev/null VERIFIER="java -jar $CLI --workdir verifier"

mkdir user 2> /dev/null USER="java -jar $CLI --workdir user"

POLICY="zkp_simple.tpl" TRANSACTION="transaction_zkp.json"
```

Input: TPL Policy

We inspect the example TPL policy used in this demonstration. The entry point predicate `accept` initializes the transaction parser and then calls to the `zk_accept` predicate (which is evaluated in privacy-preserving mode). In the `zk_accept` predicate we can see two types of predicates: On one hand, the parser for the credential is configured, which is needed to ensure the structure of the used credentials and to be able to operate on the credential's attributes. On the other hand, two rules are defined, which the user's attributes need to fulfill. First, the `credentialType` attribute is extracted and compared to the constant string `student_id_card` . Second, the `size` attribute is extracted and checked if it is greater-than a constant integer.

Please note that using the zkTPL system, the User can proof that they fulfill this policy without revealing the value of the involved attributes. Thus, the 2 User can proof that they are larger than 160cm without revealing how tall they actually are.

For demonstration purposes, this example policy does not contain any trust management checks. In a real use-case, the `accept` predicate will proceed with ensuring that the signature of the `Form` is valid, and that it was issued by a Issuer that the policy author considers trustworthy. The latter is done by consulting a trust scheme like eIDAS (by using trust status lists) or SSI (by using a distributed ledger-based trust authority).

```
cat zkp_simple.tpl

zk_accept(Presentation) :  extract(Presentation, verifiableCredential, Credential),
set_format(Credential, w3c_verifiableCredential), % string comparison:
extract(Credential, credentialType, student_id_card), % simple range proof
extract(Credential, size, Size), Size ⩾ 160 .

accept(Form) :  set_format(Form, registrationFormat), zk_accept(Form), print(done)
.
```

[Verifier] Compile policy and proof

After authoring a policy, the SP uses the zkTPL compiler to transform the policy from TPL format to a format suitable for privacy-preserving authentication. The generated files are named after the zk-predicate specified in the policy (e.g., `zk_accept`). If a policy author specifies several zk-predicates, this step would generate multiple proof programs and metadata for each of them.

In this step, the compiler first transforms the policy from TPL to a proof code in an intermediate format for the ZoKrates toolbox (`zk_accept.zok`). This proof code is then compiled using ZoKrates, resulting in a proof program in binary format (`zk_accept`). In addition, the compiler generates metadata (info) and information about how to interact with the proof program (ABI). This information is later used by the zkTPL client to call the proof program and to generate a cryptographic proof.

```
$VERIFIER --compile --policy $POLICY

Compiling done. Generated proof-program and ABI.
```

3

Generated Proof Program

To inspect the generated proof code in (human-readable) intermediate format, we inspect the `zk_accept.zok` file. The proof program itself (`zk_accept` file) is in binary format and therefore not human readable.

The generated proof code contains the required imports and some helper functions, as well as the checks specified in the policy (at the bottom). Additionally, it performs the encoding of the credentials needed to compute the credential's hash. In the end, the hash (commitment) will be part of the generated proof to ensure the integrity of the process, as it needs to match the hash specified in the (signed) credential.

```
cat verifier/zk_accept.zok

import "utils/pack/u32/unpack128" as unpack128 import "hashes/sha256/sha256" as
sha256 def compareHash(u32[8] hash1, u32[8] hash2) → bool: return hash1[0] ==
hash2[0] && hash1[1] == hash2[1] && hash1[2] == hash2[2] && hash1[3] def
compareString(field[4] str1, field[4] str2) → bool: return str1[0] == str2[0] &&
str1[1] == str2[1] && str1[2] == str2[2] && str1[3] == str2 def main(private
field[4] credentialType, private u32 size, u32[8] pub_hash) → bool: bool ret_val =
true u32[15] padding_hash = [2147483648, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
544] u32[4] credentialType_0 = unpack128(credentialType[0]) u32[4] credentialType_1
= unpack128(credentialType[1]) u32[4] credentialType_2 =
unpack128(credentialType[2]) u32[4] credentialType_3 = unpack128(credentialType[3])
field[4] credentialType_comp_0 = [153465906946986761447381171472992199680, 0, 0, 0]
u32[16] u32_16_0 = [credentialType_0[0], credentialType_0[1], credentialType_0[2],
crede u32[16] u32_16_1 = [size, padding_hash[0], padding_hash[1], padding_hash[2],
padding_has u32[2][16] data_hash = [u32_16_0, u32_16_1] ret_val = ret_val &&
compareString(credentialType, credentialType_comp_0) ret_val = ret_val && size ⩾
160 ret_val = ret_val && compareHash(pub_hash, sha256(data_hash)) return ret_val
```

Generated Presentation Request metadata

We also inspect the metadata files needed to call the (binary) proof program. They specify the structure in which the zkTPL client need to provide the 4 attributes when generating the verifiable presentation.

```
echo "-- Info: --"
cat verifier/info.json

echo "" echo "-- ABI: --" cat verifier/zk_accept_abi.json

-- Info: -{ "hash" : "0xb0ddbdef5b2ffd2224d25ad2067bc421", "predicateIDs" : [
"zk_accept" ], "stamp" : "2022-07-20T18:03:42.651425548", "compileBin" : true --
ABI: -{ "inputs": [ "name": "credentialType", "public": false, "type": "array",
"components": { "size": 4, "type": "field" "name": "size", "public": false, "type":
"u32" "name": "pub_hash", "public": true, "type": "array", "components": { "size":
8, "type": "u32" "outputs": [ "type": "bool"
```

5

```
```

[Verifier/TTP] Setup crypto system, generate keys

Based on the generated proof program, the SP (or another entity like a TTP) generates the cryptographic key material. The result of this step are two keys: the verification key (needed by the SP to verify presentations) and the proving key (needed by all Users to generate verifiable presentations).

```
$VERIFIER --setup

Setup done. Generated proving and verification keys.
```

[Verifier/TTP] Send proving key to User

In this step, the SP provides the proving key to all users. For demonstration purposes, we simply copy the key to the directory used by the User:

```
mv verifier/zk_accept_proving.key user/zk_accept_proving.key
```

[Verifier] Send presentation request to User

Additionally, the SP also provides the (compiled) proof program alongside metadata to all users. For demonstration purposes, we simply copy the presentation request files to the directory used by the User:

```
cp verifier/zk_accept user/zk_accept
cp verifier/info.json user/info.json
cp verifier/zk_accept_abi.json user/zk_accept_abi.json
```

[User] Perform authentication/showing

After receiving the proving key and the presentation request files, the User generates the verifiable presentation. In this step, the zkTPL client extracts the attributes from the User's credential to generate the required proof. Doing so, the zkTPL client reveals some attributes (marked as public in the metadata) but hides the rest of the attributes (private attributes).

The presentation contains a proof that the attributes (both revealed and private) fulfill the rules stated in the SP's policy. Since not all attribute values are revealed in the presentation, the presentation also contains the hash of the attributes and proves that the hash is indeed a hash of the attributes. This ensures the integrity of the presentation, as it proves that the proof was indeed computed on a credential with that hash.

```
$USER --prove

Load info from user/info.json Initializing of user attributes done.
```

6

```
Computation of witness done.
Generating of proof done.
```

Generated Proof

We now inspect the cryptographic proof, which is the hear of the verifiable presentation. The cryptographic proof consists of three elliptic curve points as well as the input and output of the proof program. In this example, the `input` field contains the `credentialType` and `size` attributes as well as the public hash of the credential (as specified in the ABI). Additionally, the `input` field also contains the boolean output of the proof program, which states whether the proof is valid (`0x1`) or not (`0x0`).

Since the zkTPL system uses succinct SNARK proofs, the length of this proof depends only on the amount and size of the (revealed) attributes. The proof size, as well as the time needed to verify the proof, are independent of other factors like the complexity of the policy or the size of the credential.

```
cat user/zk_accept_proof.json

 "proof": { "a": [
"0x21cdba75ac91e343814c6d5655ceb5296518c7d97dab4e7d1c9595884ddc8af0",
"0x0f9a6a43fed1d7fe161d78e796a724d1080d1c0a08aa5659c88902cc9d91fcc6" "b": [ [
"0x1dbe24fdca9ef505a94bfcdb907a1d54caf7e8db3daaf683ccc1098695a9276c",
"0x0788096f63e0eda460391d9fa063e869c0b4e1fcb5aeaf08bd79f093e60f72c5" [
"0x086aedc39094c94793fc2d61444fbaed1b9160267e9b4bb1791c24749401f4a9",
"0x22be4e0ee6b1890a297537b75b0d2a90e72f997915ecb7203ec310339b3d0e89" "c": [
"0x09f708bfb0ab530ddc17401f52f1ff02d9cfdcb73b45fe7295aeb0ee1b4be845",
"0x1fe02c926e47451b5d778e8ea5c0e660d941d952f4f8c86bb107f3184ebc48f4" "inputs": [
"0x0000000000000000000000000000000000000000000000000000000067bd04e4",
"0x000000000000000000000000000000000000000000000000000000000a58bb0be",
"0x0000000000000000000000000000000000000000000000000000000bc3502f9",
"0x0000000000000000000000000000000000000000000000000000000e17bd31b",
"0x0000000000000000000000000000000000000000000000000000000eb7915b3",
```

7

```
  "0x0000000000000000000000000000000000000000000000000000000e18402b",
  "0x0000000000000000000000000000000000000000000000000000000edebb1a6",
  "0x000000000000000000000000000000000000000000000000000000d4a8d85",
  "0x0000000000000000000000000000000000000000000000000000000000000001"
```

[User] Send presentation to Verifier

The User then sends the verifiable presentation back to the SP. The presentation contains the proof as well as metadata (e.g., the credential's signature), but not the credential itself or any other attributes. For demonstration purposes, we simply copy the proof file to the directory used by the SP (Verifier):

```
cp user/zk_accept_proof.json verifier/zk_accept_proof.json
```

[Verifier] Check presentation using policy

After receiving the User's access request and presentation, the SP executes the policy and check if the provided presentation fulfills their access policy. To do so, the SP uses the zkTPL verification tool is used to verify the policy, which in turn calls the zkTPL verifier to also verify the proof.

In the end, the SP learns:

- that the user is in possession of a trustworthy and valid credential of certain type,
- that the attributes from this credential fulfill the access policy,
- and that the presentation was indeed generated using attributes from that credential.

The SP learns *not*:

- the value of the private attributes,
- or any other data that was not explicitly requested in the policy.

```
$VERIFIER --verify --policy $POLICY --transaction $TRANSACTION

OK: ATV 3.0.0-SNAPSHOT OK: 3 pre-checks passed! OK: TPL Interpreter initialized!
OK: registrationFormat extraction successful. OK: Verifying zk_accept proof:
ZKProofInfo{path='verifier', hash='0x67bd04e4a58bb0be OK: Verification of zkTPL
proof done: success! PRINT: Done Verification of proof and policy done: All good!
```

8