

Masters in Cybersecurity

Applied Cryptography

Selective Identity Disclosure (SID)

2nd Project

João Luís - 107403

Ricardo Quintaneiro - 110056

December 26, 2024

1 Introduction

In an increasingly digital world, the management and protection of personal identity information has become a critical concern. While digital identification systems offer convenience and efficiency, they often lack granular control over what personal information is shared in each interaction. This project addresses this challenge by implementing a Selective Identity Disclosure (SID) system that enables users to maintain control over their digital identity attributes.

The system implements a Digital Citizen Card (DCC) protection mechanism with support for the Portuguese Citizen Card (CC) that allows users to selectively disclose specific identity attributes while ensuring the authenticity and ownership of the shared information. This approach solves a fundamental privacy challenge present in traditional identification systems, where all information must be disclosed regardless of the actual requirements of the verifying party.

The implementation consists of four key applications that together form a complete SID system:

- A DCC request application (*req_dcc*) for identity holders;
- A DCC generation application (*gen_dcc*) for identity issuers;
- A minimal DCC generation tool (*gen_min_dcc*) for selective attribute disclosure;
- A validation application (*check_dcc*) for verifying minimal DCCs.

This report details the implementation decisions, cryptographic approaches, and technical strategies employed in developing these applications. The system uses commitment schemes, digital signatures, and careful attribute masking to ensure both the privacy of the identity holder and the verifiability of the disclosed information.

2 gen_dcc

The DCC generation application implements a server that generates Digital Citizen Cards in response to a request, via TCP socket. This application is compatible with Portuguese CC.

We start by mapping the attributes into *Identity_Attribute* objects, which is a python data-class[1] that takes a label, a value and a pseudo-random mask to generate a commitment value through a hash function. The list of attribute names, list of masks, and the identification of the digest function used are provided by the DCC owner.

```
@dataclass(frozen=True)
class Identity_Attribute:
    pseudo_random_mask: str = ""
    label: str = ""
    value: str = ""
    commitment_value: str = field(init=False)
    digest_description: str = ""

    def __post_init__(self):
        hash = Hash(SHA384())
        hash.update((self.label + self.value + self.pseudo_random_mask).encode())
        object.__setattr__(self, 'commitment_value', hash.finalize().hex())

    def __hash__(self):
        return hash((self.label, self.value))
```

We also wrapped the received public key, that comes in hexadecimal form, in the *Public_Key* class, that also comes with a method to convert the public key into .PEM format.

```
@dataclass
class Public_Key:
    key: Ed448PublicKey = None
    algorithm: str = ""

    def to_pem(self):
        return self.key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )
```

As far as the issuer is concerned, we generate an X509 certificate and write it to a file in .pem format, starting by generating a private key and obtaining the corresponding public key, and then adding some issuer details and expiry date.

```
def generate_issuer_certificate():
    """
    Generates an RSA private key and corresponding self-signed certificate for the issuer.
    Saves the certificate to a file.
    """
```

```
global issuer_private_key
issuer_private_key = rsa.generate_private_key(public_exponent=65537, key_size=4096)
public_key = issuer_private_key.public_key()

subject = issuer = x509.Name([
    x509.NameAttribute(NameOID.COUNTRY_NAME, "PT"),
    x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, "Aveiro"),
    x509.NameAttribute(NameOID.LOCALITY_NAME, "Esgueira"),
    x509.NameAttribute(NameOID.ORGANIZATION_NAME, "CA_ISSUER"),
    x509.NameAttribute(NameOID.COMMON_NAME, "example.com"),
])

cert = x509.CertificateBuilder().subject_name(subject)
    .issuer_name(issuer)
    .public_key(public_key)
    .serial_number(x509.random_serial_number())
    .not_valid_before(datetime.datetime.now(datetime.timezone.utc))
    .not_valid_after(datetime.datetime.now(datetime.timezone.utc) +
        ↪ datetime.timedelta(days=10))
    .sign(issuer_private_key, hashes.SHA256())

with open("issuer_certificate.pem", "wb") as f:
    f.write(cert.public_bytes(serialization.Encoding.PEM))
```

Lastly, we create the DCC, signing the commitment values and owner public key with the issuer private key, which is accompanied by timestamp, a description of the asymmetric cryptosystem used for signing and the issuer certificate.

```
def sign_attributes(self):
    data_to_sign = ("".join(attr.commitment_value for attr in self.identity_attributes) +
        self.owner_public_key.to_pem().decode()
    ).encode()

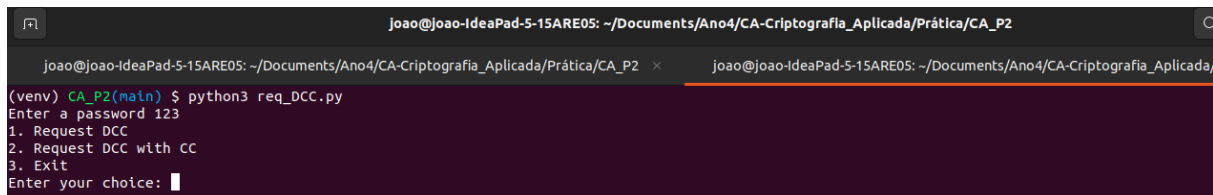
    signature = self.issuer_private_key.sign(
        data_to_sign,
        padding.PSS(
            mgf=padding.MGF1(SHA384()),
            salt_length=padding.PSS.MAX_LENGTH
        ), SHA384()
    )

    self.issuer_signature = Issuer_Signature(
        signature_value=signature.hex(),
        timestamp=datetime.datetime.now(datetime.timezone.utc).isoformat(),
        algorithm="RSA-4096",
        issuer_certificate=self.issuer_certificate)
```

This DCC will be sent back to the requester in a JSON format representation. To send this data we start by sending the length of the message because in this way the requester knows when to close the TCP connection.

3 req_dcc

To have a DCC generated, it's necessary to have a request. This request is done interacting with *req_dcc* application. The application starts for asking a password, that will serve for pseudo-random mask derivation, and then asks if the user want's to generate a DCC using Portuguese CC or not.



```
Joao@joao-IdeaPad-5-15ARE05: ~/Documents/Ano4/CA-Criptografia_Aplicada/Prática/CA_P2
Joao@joao-IdeaPad-5-15ARE05: ~/Documents/Ano4/CA-Criptografia_Aplicada/Prática/CA_P2
(venv) CA_P2(main) $ python3 req_dcc.py
Enter a password 123
1. Request DCC
2. Request DCC with CC
3. Exit
Enter your choice: █
```

Figure 1: python3 req_dcc.py

If the first option is chosen, the private key will be generated and so will the corresponding public key. The attributes for this option are declared in the code along with their values.

As for using CC, since we implemented the project in Python, we can't use the SDK mentioned in the references in the guidelines. With the help of the code found in Guide 7 from Applied Cryptography course and using PKCS#11, we obtain the name, CC serial number, country and date of birth from the X509 certificate extensions. Finally, we also get the value of the card's public key. It's important to note that it's not possible to get the private key from the card, we have to use it directly from the card.

After having the attributes, the *generate_identity_attributes()* function is going to be used to generate pseudo-random mask values, that are followed by labels, respective values and digest description, to be sent to DCC issuer.

```
def generate_identity_attributes(password: str, attributes: dict[str, str]) ->
↳ set[Identity_Attribute]:
    identity_attributes = set()
    for label, value in attributes.items():
        hash = Hash(SHA3_512())
        hash.update((label + password).encode())
        pseudo_random_mask = hash.finalize().hex()

        identity_attributes.add(Identity_Attribute(pseudo_random_mask, label,
↳ value, digest_description="SHA-3_512 for pseudo-random mask and SHA-384 for
↳ commitment value"))
    return identity_attributes
```

After this process, the data will be organised to be sent over the TCP socket in JSON format. If the DCC request is successfully generated, the owner will receive it and make a *json.dump()* to a file.

4 gen_min_dcc

As an application to show a selected set of DCC attributes, *gen_min_DCC.py* will produce a minDCC document from a given DCC. It is run solely by the DCC owner, to guarantee the selective disclosure of some of his CC information.

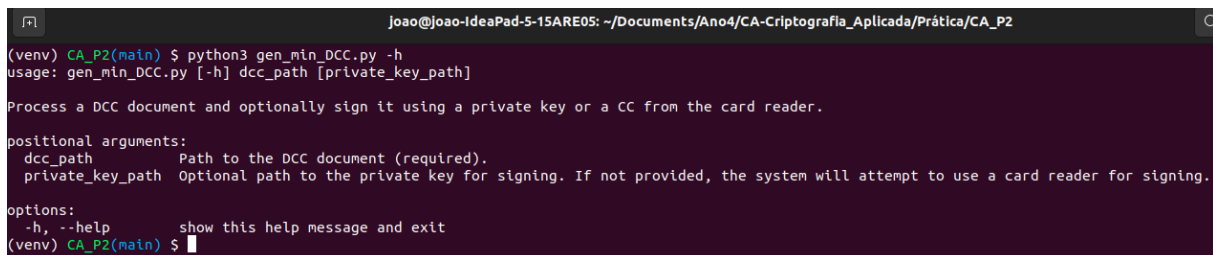
To produce a minDCC document, we first used a Python dataclass to represent what a minDCC requires:

```
@dataclass
class minDCC:
    commitment_values: list = field(default_factory=list)
    attributes_digest_description: str = "SHA-3_512 for pseudo-random mask and SHA-384 for
    → commitment values"
    identity_attributes: set = field(default_factory=set)
    owner_public_key: Public_Key = field(default_factory=Public_Key)
    owner_private_key: Ed448PrivateKey | None = None
    issuer_signature: Issuer_Signature = field(default_factory=Issuer_Signature)
    producer_signature: Signature = field(init=False)
```

Some of the fields include classes that were imported from the DCC program, the *Public_Key*, *Issuer_Signature* and *Signature* classes. This makes sure that both DCC and minDCC applications are more coupled and therefore are more compatible with each other. The producer signature can't be passed as an argument as that will be generated later, once the object is created.

To make sure we can use the Citizen Card to sign the attributes, the program can be run in two ways:

- Passing the DCC.json document **and** the PEM file of the private key of the owner
- Passing **only** the DCC.json and having the Citizen Card inserted to be able to sign



```
Joao@Joao-IdeaPad-5-15ARE05: ~/Documents/Ano4/CA-Criptografia_Aplicada/Prática/CA_P2
(venv) CA_P2(main) $ python3 gen_min_DCC.py -h
usage: gen_min_DCC.py [-h] dcc_path [private_key_path]

Process a DCC document and optionally sign it using a private key or a CC from the card reader.

positional arguments:
  dcc_path             Path to the DCC document (required).
  private_key_path     Optional path to the private key for signing. If not provided, the system will attempt to use a card reader for signing.

options:
  -h, --help           show this help message and exit
(venv) CA_P2(main) $
```

Figure 2: python3 gen_min_DCC.py -h

After handling the given files, the program extracts the information of the DCC document file, presents the identity attributes to the user and asks for input on which are meant to be revealed:

```
commitment_values = [attr["commitment_value"] for attr in
↳ dcc_document["identity_attributes"]]
digest_description = dcc_document["attributes_digest_description"]
owner_public_key = dcc_document["owner_public_key"]
issuer_signature = dcc_document["issuer_signature"]

print("The attributes in the DCC are:")
for attribute in dcc_document["identity_attributes"]:
    print(attribute["label"])
print()

lables_of_attributes_to_reveal = (input("Input the attributes you want to reveal,
↳ seperated by commas, and press enter: ")).strip().replace(" ", "").split(",")
```

Then it stores that information, asks the user for the password and produces the identity attributes using the *generate_identity_attributes* function of the DCC program, and, to make sure the producer of the minDCC is indeed the owner of the DCC attributes, compares the generated commitment values with the ones in the document, raising an error if it can't validate them.

```
revealed_attributes = dict()
for attribute in dcc_document["identity_attributes"]:
    label = attribute["label"]
    if label in lables_of_attributes_to_reveal:
        revealed_attributes[label] = attribute["value"]

password = input("Password: ")
identity_attributes = generate_identity_attributes(password, revealed_attributes)
for attribute in identity_attributes:
    if attribute.commitment_value not in commitment_values:
        raise ValueError("Generated commitment value does not match the DCC document.")
```

After that, the minDCC object is created using the class constructor, and with the help of the intermediate classes *Public_Key* and *Issuer_Signature*.

```
min_dcc = minDCC(
    commitment_values=commitment_values,
    attributes_digest_description=digest_description,
    identity_attributes=identity_attributes,
    owner_public_key=Public_Key(load_pem_public_key(owner_public_key["key"].encode()),
↳ owner_public_key["algorithm"]),
    owner_private_key=owner_private_key,
    issuer_signature=Issuer_Signature(issuer_signature["signature_value"],
↳ issuer_signature["timestamp"], issuer_signature["algorithm"],
↳ issuer_certificate=issuer_signature["issuer_certificate"])))
```

When the minDCC object is created, the `__post_init__` function will make it sign all of its selected attributes. It will realize the need to sign with the Citizen Card by having the private key of the owner or not, and it will make the signage operation function accordingly.

```
def __post_init__(self):
    self.sign_attributes()

def sign_attributes(self):
    attributes = dict()
    for attr in self.identity_attributes:
        attributes[attr.label] = {
            "value": attr.value,
            "mask": attr.pseudo_random_mask,
        }

    data_to_sign = (
        "".join(commitment_value for commitment_value in self.commitment_values) +
        "".join(label + value + mask for label, attr in attributes.items() for value, mask
        ↪ in attr.items())
        .join(self.owner_public_key.to_pem().decode())
        .join(self.issuer_signature.signature_value)
    ).encode()

    if self.owner_private_key is not None:
        try:
            private_key = load_pem_private_key(
                self.owner_private_key.encode(),
                password=None,
                backend=default_backend()
            )

            # Ensure the loaded key is Ed448
            if not isinstance(private_key, Ed448PrivateKey):
                raise ValueError("The provided private key is not an Ed448 key.")
        except Exception as e:
            raise ValueError(f"Error loading private key: {e}")

        signature = private_key.sign(
            data_to_sign
        )

        algorithm = "Ed448"

    else:
        for slot in slots:
            if 'CARTAO DE CIDADAO' in pkcs11.getTokenInfo(slot).label:
                session = pkcs11.openSession(slot)
                privKey = session.findObjects( [( CKA_CLASS , CKO_PRIVATE_KEY ),(CKA_LABEL,
                ↪ 'CITIZEN AUTHENTICATION KEY')])[0]
                signature = bytes(session.sign(
                ↪ privKey,data_to_sign,Mechanism(CKM_SHA1_RSA_PKCS)))
```



```
        session.closeSession
        algorithm = "SHA1_RSA_PKCS"
        break
    else:
        raise ValueError("No private key provided for signing.")

    self.producer_signature = Signature(
        signature_value=signature.hex(),
        timestamp = datetime.datetime.now(datetime.timezone.utc).isoformat(),
        algorithm = algorithm,
    )
```

Finally, the representation of the object, which is valid JSON and is the minDCC document that was asked in the first place, is written to a file.

```
with open("min_DCC.json", "w") as f:
    f.write(min_dcc.__repr__())
```

Here we see the code of the `__repr__` function, the representation of the object which is the minDCC document itself:

```
def __repr__(self):
    # Convert identity attributes to a list of dictionaries
    attributes = dict()
    for attr in self.identity_attributes:
        attributes[attr.label] = {
            "value": attr.value,
            "mask": attr.pseudo_random_mask,
        }

    return json.dumps({
        "commitment_values": self.commitment_values,
        "attributes_digest_description": self.attributes_digest_description,
        "revealed_identity_attributes": attributes,
        "owner_public_key": {
            "key": self.owner_public_key.to_pem().decode(),
            "algorithm": self.owner_public_key.algorithm
        },
        "issuer_signature": asdict(self.issuer_signature),
        "producer_signature": asdict(self.producer_signature),
    }, indent=2)
```

5 check_dcc

In this final program, *check_dcc*, the task is to validate if a minDCC is valid or not, and it can be validated by anyone who has a minDCC JSON document. To do that, the program must take the path to a JSON document as an argument, load the document and validate the signatures of the issuer of the DCC and the producer of the minDCC.

```
parser = argparse.ArgumentParser()
parser.add_argument("min_dcc_path", type=str)

args = parser.parse_args()

try:
    with open(args.min_dcc_path, "r") as dcc_file:
        min_dcc = json.load(dcc_file)
except Exception as e:
    print(f"Error loading DCC document: {e}")
    return

is_valid = minDCC.validate_signatures(
    commitment_values=min_dcc["commitment_values"],
    identity_attributes=min_dcc["revealed_identity_attributes"],
    owner_public_key=min_dcc["owner_public_key"],
    issuer_signature=min_dcc["issuer_signature"],
    producer_signature=min_dcc["producer_signature"]
)

if is_valid:
    print("minDCC is valid")
else:
    print("minDCC is invalid")
```

As is shown in the previous bit of code, the entire validation occurs within the *validate_signatures* static method of the minDCC dataclass.

However, to understand that validation it is necessary to first understand the validation of a DCC, which extracts the public key from the issuer of its certificate (which is included in the issuer signature object), and validates the data that was signed, commitment values and the public key of the owner.

```
@staticmethod
def validate_signature(owner_public_key: Public_Key, issuer_signature: Issuer_Signature,
    ↪ commitment_values: Iterable[str]):
    data_to_validate = (
        "".join(commitment_values) +
```

```
        owner_public_key.to_pem().decode()
    ).encode()

    try:
        certificate =
        ↪ load_pem_x509_certificate(issuer_signature.issuer_certificate.encode(),
        ↪ backend=default_backend())
        issuer_public_key = certificate.public_key()
    except Exception as e:
        print(f"Error loading issuer certificate: {e}")
        return False

    try:
        issuer_public_key.verify(
            bytes.fromhex(issuer_signature.signature_value),
            data_to_validate,
            padding.PSS(
                mgf=padding.MGF1(SHA384()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            SHA384()
        )
        return True
    except Exception:
        return False
```

The DCC validation is then used, in conjunction with the validation of the data signed by the minDCC producer, to validate the entire minDCC document.

```
@staticmethod
def validate_signatures(commitment_values: list, identity_attributes: str,
    ↪ owner_public_key: str, issuer_signature: str, producer_signature: str) -> bool:

    attributes = dict(identity_attributes)
    issuer_signature = dict(issuer_signature)
    producer_signature = dict(producer_signature)
    owner_public_key = dict(owner_public_key)

    data_to_validate = (
        "".join(commitment_value for commitment_value in commitment_values) +
        "".join(label + value + mask for label, attr in attributes.items() for value, mask
        ↪ in attr.items())
        .join(owner_public_key["key"])
        .join(issuer_signature["signature_value"])
    ).encode()

    owner_public_key = Public_Key(
        load_pem_public_key(owner_public_key["key"].encode()),
        owner_public_key["algorithm"]
```

```
)

issuer_signature = Issuer_Signature(
    signature_value = issuer_signature["signature_value"],
    timestamp = issuer_signature["timestamp"],
    algorithm = issuer_signature["algorithm"],
    issuer_certificate = issuer_signature["issuer_certificate"]
)

producer_signature = Signature(
    signature_value = producer_signature["signature_value"],
    timestamp = producer_signature["timestamp"],
    algorithm = producer_signature["algorithm"]
)

try:
    is_issuer_valid = DCC.validate_signature(owner_public_key, issuer_signature,
    ↪ commitment_values)
    print("Issuer signature is " + ("valid" if is_issuer_valid else "invalid"))
    if not is_issuer_valid:
        raise ValueError("Issuer signature is invalid.")

    if owner_public_key.algorithm == "Ed448":
        owner_public_key.key.verify(
            bytes.fromhex(producer_signature.signature_value),
            data_to_validate,
        )
    else:
        owner_public_key.key.verify(
            bytes.fromhex(producer_signature.signature_value),
            data_to_validate,
            padding.PKCS1v15(),
            hashes.SHA1()
        )
    return True
except Exception:
    return False
```

6 Choice of algorithms in the project

To comply with the use of different algorithms in the *gen_dcc* applications, we use the Python Cryptography’s implementations [2] to make use of:

1. RSA-4096 for the issuer cryptosystem;
2. Ed448 for the owner cryptosystem, if no Citizen Card is provided;
3. SHA-3 with 512 bits to generate the pseudo-random mask;
4. SHA-384 to generate the commitment values;

RSA-4096 serves as the issuer’s cryptosystem due to its position as a well-established public-key algorithm. The 4096-bit key length provides a high security margin, appropriate for a Certificate Authority role. Even if computationally intensive, this is acceptable for the issuer since key generation occurs infrequently and the issuer typically has sufficient computational resources. Additionally, RSA’s widespread support in legacy systems and PKI infrastructures makes it ideal for certificate creation and signing operations that may need to interact with existing systems.

For the owner cryptosystem, Ed448 was chosen for its excellent balance of security and performance. It provides a level of high security while being significantly faster than RSA for key generation and signing operations. Its smaller key and signature sizes, combined with resistance to side-channel attacks, make it particularly suitable for frequent operations by DCC owners on various devices.

The pseudo-random mask generation utilizes SHA-3 with 512 bits output. As the latest SHA standard, it offers strong security properties and randomness characteristics. Its different internal structure from SHA-2 provides algorithmic diversity, and its resistance to length extension attacks makes it particularly suitable for mask generation in our system [3].

For commitment values, SHA-384 was selected. The output size of 384 bits prevents collisions and length extension attacks effectively [4] while maintaining computational efficiency, making it well-suited for frequent commitment calculations.

7 References

- [1] P. Docs, *Dataclasses - data classes*. [Online]. Available: <https://docs.python.org/3/library/dataclasses.html>.
- [2] pyca/cryptography. [Online]. Available: <https://cryptography.io/en/latest/>.
- [3] Wikipedia. [Online]. Available: <https://en.wikipedia.org/wiki/SHA-3#Design>.
- [4] Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/SHA-2#Comparison_of_SHA_functions.
- [5] André-Zúquete, Nov. 2024. [Online]. Available: <https://sweet.ua.pt/andre.zuquete/Aulas/CA/24-25/docs/Ex7.pdf>.