Univ.Prof. Dr. Dominique Schröder
Paul Gerhart
Version: 30. October 2024

Lecture: Privacy Enhancing Technologies (B.Sc.)

Challenge Sheet 2

Scenario

The EU mandates backdoors in all available messaging platforms, resulting in only insecure communication channels. We aim to maintain our privacy by creating a secure channel that can be used within these messengers. This involves encrypting and signing messages and then converting them into a compatible, sendable format. To achieve this, we will implement:

- **Hybrid encryption** combining ElGamal and AES for robust data security.
- **Schnorr signatures** to authenticate messages.
- A **message parser** to serialize and deserialize encrypted messages into a format compatible with messengers.

Setup

For the field $\mathbb{Z}_p$, we use the `curve25519_dalek::scalar::Scalar` struct, and for the group $\mathbb{G}$, we operate over `curve25519_dalek::ristretto::RistrettoPoint` elements. The generator $g \in \mathbb{G}$ is represented by `curve25519_dalek::constants::RISTRETTO_BASEPOINT_POINT`. For hashing, we use `sha2::Sha512`, and for random sampling ($\leftarrow\!\!\$$), we rely on the random number generator `rand::rngs::OsRng`.

Challenge 1: Key Generation (1P)

In this challenge, you will implement a structure for managing DLog-based cryptographic key pairs and a method for generating keys. We will use these keys later for the ElGamal public key encryption and the Schnorr signature scheme.

$\mathsf{KGen}(\lambda)$

1 : $\mathsf{sk} \leftarrow\!\!\$\ \mathbb{Z}_p$
2 : $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$
3 : **return** $(\mathsf{sk}, \mathsf{pk})$

Figure 1: Key generation pseudocode.

To cover the interfaces of this primitive, we define the following struct and method:

```
pub struct KeyPair {
    pub private_key: Scalar,
    pub public_key: RistrettoPoint,
}

pub fn generate() -> KeyPair
```

Challenge 2: ElGamal Encryption (2P)

In this challenge, you will implement the encryption and decryption functions for (signed) ElGamal encryption. Looking ahead, this public key encryption scheme will allow us to encrypt

AES keys as scalar values. You will use the recipient's public key to encrypt the message and their private key to decrypt it. We formally define (signed) ElGamal in Figure 2. Note that the key generation algorithm equals the key generation we implemented in Challenge 1. Therefore,

$\mathsf{ElGamalKGen}(\lambda)$	$\mathsf{ElGamalEncrypt}(\mathsf{pk}, m)$	$\mathsf{ElGamalDecrypt}(\mathsf{sk}, (c_1, c_2))$
1 : $\mathsf{sk} \leftarrow\$ \mathbb{Z}_p$	1 : $r \leftarrow\$ \mathbb{Z}_p$	1 : $m = c_2 - \mathsf{H}(c_1^{\mathsf{sk}})$
2 : $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$	2 : $c_1 \leftarrow g^r$	2 : **return** m
3 : **return** $(\mathsf{sk}, \mathsf{pk})$	3 : $c_2 \leftarrow \mathsf{H}(\mathsf{pk}^r) + m$	
	4 : **return** (c_1, c_2)	

Figure 2: ElGamal encryption and decryption pseudocode.

you only need to implement the following methods for the `ElGamalCiphertext` struct:

```rust
pub struct ElGamalCiphertext {
    pub c1: RistrettoPoint,
    pub c2: Scalar,
}

impl ElGamalCiphertext {
    /// Generates a new KeyPair for encryption
    pub fn keygen() -> KeyPair {

    /// Encrypts a message (represented as a Scalar) using the recipient's public key
    pub fn encrypt(message: &Scalar, public_key: &RistrettoPoint) -> ElGamalCiphertext

    /// Decrypts an ElGamal ciphertext using the recipient's private key
    pub fn decrypt(ciphertext: &ElGamalCiphertext, private_key: &Scalar) -> Scalar
}
```

Challenge 3: AES Encryption (2P)

In this challenge, you will implement AES-256-GCM encryption and decryption. This encryption scheme will allow us to securely encrypt messages using a symmetric key derived from a scalar. We will later integrate AES encryption with ElGamal encryption to create a hybrid encryption scheme. Use the library `aes_gcm::Aes256Gcm` for AES-256-GCM.

For our challenge, you need to implement the following struct and methods in Rust:

```rust
pub const AES_NONCE_SIZE: usize = 12; // Nonce size of 12 bytes (=96 bit)

pub struct AESCiphertext {
    pub nonce: [u8; AES_NONCE_SIZE], // The nonce used for encryption
    pub ciphertext: Vec<u8>,      // The actual ciphertext
}

impl AESCiphertext{
    /// Generates a random scalar to be used as an AES key
```

$\mathsf{AESKGen}(\lambda)$

1 : $\mathsf{k} \leftarrow\!\!\$\ \mathbb{Z}_p$
2 : **return** k

$\mathsf{AESEncrypt}(\mathsf{k}, m)$

1 : $\mathsf{nonce} \leftarrow\!\!\$\ \{0,1\}^{96}$
2 : $\mathsf{ciphertext} \leftarrow \mathsf{AES\text{-}256\text{-}GCM.Encrypt}(\mathsf{k}, \mathsf{nonce}, m)$
3 : **return** $(\mathsf{nonce}, \mathsf{c})$

$\mathsf{AESDecrypt}(\mathsf{k}, (\mathsf{nonce}, \mathsf{c}))$

1 : $m \leftarrow \mathsf{AES\text{-}256\text{-}GCM.Decrypt}(\mathsf{k}, \mathsf{nonce}, \mathsf{c})$
2 : **return** m

Figure 3: AES-256-GCM encryption and decryption pseudocode.

```
10      pub fn keygen() -> Scalar
11
12      /// Encrypts a plaintext message using AES-256-GCM with a Scalar as the AES
            key
13      pub fn encrypt(scalar_key: &Scalar, message: &[u8]) -> Result<AESCiphertext,
            String>
14
15      /// Decrypts a ciphertext using AES-256-GCM with a Scalar as the AES key
16      pub fn decrypt(scalar_key: &Scalar, aes_ciphertext: &AESCiphertext) ->
            Result<Vec<u8>, String>
17  }
```

Challenge 4: Hybrid Encryption (2P)

In this challenge, you will implement a hybrid encryption scheme that combines ElGamal and AES-256-GCM. Here, ElGamal encrypts an AES session key, which is then used for encrypting the main message. The resulting ciphertext includes both the ElGamal and AES-encrypted components. As a key generation, we use $\mathsf{ElGamalKGen}$ to derive the key-pair $(\mathsf{sk}, \mathsf{pk})$. We formally define hybrid encryption as follows:

$\mathsf{HybridKGen}(\lambda)$

1 : **return** $\mathsf{ElGamalKGen}(\lambda)$

$\mathsf{HybridEncrypt}(\mathsf{pk}, m)$

1 : $\mathsf{aes\_key} \leftarrow\!\!\$\ \mathsf{AESKGen}(\lambda)$
2 : $(\mathsf{nonce}, \mathsf{aes\_ciphertext}) \leftarrow \mathsf{AESEncrypt}(\mathsf{aes\_key}, m)$
3 : $\mathsf{elgamal\_ciphertext} \leftarrow \mathsf{ElGamalEncrypt}(\mathsf{pk}, \mathsf{aes\_key})$
4 : **return** $(\mathsf{elgamal\_ciphertext}, (\mathsf{nonce}, \mathsf{aes\_ciphertext}))$

$\mathsf{HybridDecrypt}(\mathsf{sk}, (\mathsf{elgamal\_ciphertext}, (\mathsf{nonce}, \mathsf{aes\_ciphertext})))$

1 : $\mathsf{aes\_key} \leftarrow \mathsf{ElGamalDecrypt}(\mathsf{sk}, \mathsf{elgamal\_ciphertext})$
2 : $m \leftarrow \mathsf{AESDecrypt}(\mathsf{aes\_key}, (\mathsf{nonce}, \mathsf{aes\_ciphertext}))$
3 : **return** m

Figure 4: Hybrid encryption and decryption pseudocode.

Lecture: Privacy Enhancing Technologies (B.Sc.)

Challenge Sheet 2

For our challenge, you need to implement the following struct and methods in Rust:

```rust
pub struct HybridCiphertext {
    pub elgamal_ciphertext: ElGamalCiphertext, // Encrypted AES key
    pub aes_ciphertext: AESCiphertext,     // AES-encrypted message
}

impl HybridCiphertext{
    pub fn keygen() -> KeyPair

    /// Encrypts a message using Hybrid Encryption with ElGamal and AES-256-GCM
    pub fn hybrid_encrypt(
        message: &[u8],
        elgamal_public_key: &RistrettoPoint,
    ) -> Result<HybridCiphertext, String>

    /// Decrypts a HybridCiphertext using the ElGamal private key to retrieve the
        AES key,
    /// then uses the AES key to decrypt the message
    pub fn decrypt(&self, private_key: &Scalar) -> Result<Vec<u8>, String>
}
```

Why Hybrid Encryption? ElGamal alone is impractical for large messages due to its inefficiency with longer data. Hybrid encryption leverages the efficiency of symmetric encryption (AES) for the message while using asymmetric encryption (ElGamal) security to protect the AES key. This combination provides both security and performance.

Challenge 5: Schnorr Signatures (2P)

In this challenge, you will implement Schnorr signatures to authenticate messages. A Schnorr signature provides a secure way to verify that a message originated from a private key holder without revealing the key itself. Note that the key generation algorithm equals the key generation we implemented in Challenge 1. However, we should *never* reuse keys for different cryptographic primitives. This means a key generated for ElGamal encryption should not also be used for signing messages, even if the keys have the same form.

$\mathsf{SchnorrKGen}(\lambda)$	$\mathsf{Sign}(\mathsf{sk}, m)$	$\mathsf{Verify}(\mathsf{pk}, m, (R, s))$
1 : $\mathsf{sk} \leftarrow\!\!\$\ \mathbb{Z}_p$	1 : $r \leftarrow\!\!\$\ \mathbb{Z}_p$	1 : $h \leftarrow \mathsf{H}(\mathsf{pk}, R, m)$
2 : $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$	2 : $R \leftarrow g^r$	2 : **return** $g^s \stackrel{?}{=} R \cdot \mathsf{pk}^h$
3 : **return** $(\mathsf{sk}, \mathsf{pk})$	3 : $h \leftarrow \mathsf{H}(\mathsf{pk}, R, m)$	
	4 : $s \leftarrow r + h \cdot \mathsf{sk}$	
	5 : **return** (R, s)	

Figure 5: Schnorr signature signing and verification pseudocode.

To implement Schnorr signatures, you will need the following struct and methods in Rust:

```rust
pub struct SchnorrSignature {
```

```
    pub R: RistrettoPoint, // Commitment point
    pub s: Scalar,       // Response scalar
}

impl SchnorrSignature {
    /// Generates a new KeyPair for signing
    pub fn keygen() -> KeyPair

    /// Sign a message with a private key
    pub fn sign(message: &[u8], signing_key: &Scalar) -> SchnorrSignature

    /// Verify a Schnorr signature
    pub fn verify(signature: &SchnorrSignature, message: &[u8],public_key:
        &RistrettoPoint) -> bool
}
```

Challenge 6: The Message Struct (2P)

In this challenge, you will implement a structure to manage the message format, allowing for encryption, signing, and verifying within our secure communication setup. We provide the serialization and deserialization functions in the file `serializers.rs`, so you do not need to implement them.

```
pub struct Message {
    pub version: u8,           // Version number of the message (1 byte)
    pub payload: Vec<u8>,      // The message content (payload)
    pub recipient: [u8; 32],   // The recipient's identifier
    pub sender: [u8; 32],      // The sender's identifier
    pub signature: SchnorrSignature, // The Schnorr signature of the payload
}

impl Message {

    /// Creates a new message with a version, payload, and recipient
        (CompressedRistretto converted to Vec<u8>)
    pub fn new(
        version: u8,
        payload: Vec<u8>,
        sender: CompressedRistretto,
        recipient: CompressedRistretto,
        signature: SchnorrSignature,
    ) -> Self

    /// Encrypts the message using hybrid encryption
    pub fn encrypt(&mut self, elgamal_public_key: &RistrettoPoint) -> Result<(),
        String>

    /// Decrypts the message payload using hybrid decryption
    pub fn decrypt(&mut self, elgamal_private_key: &Scalar) -> Result<(), String>

    /// Signs the message payload using Schnorr signatures
```

```rust
    pub fn sign(&mut self, signing_key: &Scalar)

    /// Verifies the Schnorr signature of the message
    pub fn verify(&self) -> bool
}
```

We formally define the encryption, decryption, signing, and verification processes in Figure 6.

msg.EncryptMessage(ElGamal.pk)

```
// encrypt the whole message, not just the payload; Use the provided serializer for it
hybrid_ciphertext ← HybridEncrypt(pk, Serialize(msg))
msg.payload ← hybrid_ciphertext
msg.version ← msg.version + 1
msg.sender ← default_sender
msg.recipient ← pk
msg.signature ← ∅
```

msg.DecryptMessage(ElGamal.sk)

```
hybrid_ciphertext ← msg.payload
plaintext ← HybridDecrypt(sk, hybrid_ciphertext)
decrypted_message ← Deserialize(plaintext)
msg.version ← decrypted_message.version
msg.payload ← decrypted_message.payload
msg.sender ← decrypted_message.sender
msg.recipient ← decrypted_message.recipient
msg.signature ← decrypted_message.signature
```

msg.SignMessage(Schnorr.sk)

```
vk ← g^Schnorr.sk
σ ← Sign(Schnorr.sk, msg.payload)
msg.signature ← σ
msg.sender ← vk
```

msg.VerifyMessage()

```
vk ← msg.sender
return Verify(vk, msg.payload, msg.signature)
```

Figure 6: Message struct pseudocode for encryption, decryption, signing, and verification.

Challenge 7: Sending an Encrypted and Authenticated Message (5P)

Finally, we encrypt-then-sign a plaintext message and export it to a file. The message should be hybrid-encrypted w.r.t. the public key `HIn1HpHqWUR1bzTRmCjdpbqTB5RUFu7eERX0yi/rcR8=`

and signed w.r.t. the signing key `EHeUgpnf1ymdHHcdW6e+yit5dV/dZ6UmU7uHbYCWnQ4=`. (these keys are base64 encoded and correspond to a CompressedRistretto and a Scalar. Save your message under the name `signed_encrypted_message.json`. This file already exists at the right place, just replace the content. I.e., we need the following workflow:

```
// load the signing key
let signing_key = KeyPair::sk_from_file("signing_key.txt")

// load the encryption public key
let encryption_key = KeyPair::pk_from_file("encryption_key.txt")

// create a new message
let mut message = Message::new("Group ID: 123456"); // The message should
    contain your group ID

// encrypt the message
message.encrypt(&encryption_key).expect("failed to encrypt the message");

// sign the message
message.sign(&signing_key);

let _ = message.to_file("signed_encrypted_message.json");
```

When serialized correctly, your message has the form

```
{
  "version": 1,
  "payload":
     "xqZE4xupPhpf/6zvzVEJUA1n6dyNfLKtAi/XBbpGyiXOhtWbwvZa3JhD7LYLbEucvZK13",
  "recipient": "HIn1HpHqWUR1bzTRmCjdpbqTB5RUFu7eERXOyi/rcR8=",
  "sender": "nFrRkU8AeesMMIHidGuOr4x6LAoUnZ8lsa4ywEjI9Qk=",
  "signature": {
    "R": "Wt47ZKBo2c8nQbc80WKhaT5qNj3nUIEgCJbIG3pjwQk=",
    "s": "7ph5/hmIGWEko5IRiN7nehlarbcieXWWNFPf/qwT6gQ="
  }
}
```

To get the whole five points, the sample solution must decrypt and verify the message w.r.t. the provided keys.