UNIVERSITY OF TARTU
Institute of Computer Science
Cyber Security Curriculum

Eduard Iltšuk

Two-Party ECDSA Protocol for Smart-ID

Master's Thesis (30 ECTS)

Supervisor: Arnis Paršovs, MSc

Tartu 2020

Two-Party ECDSA Protocol for Smart-ID

Abstract:
Smart-ID is a digital signature solution based on threshold cryptography where two parties (mobile device and server) collaborate in key generation and signing process. The current solution uses RSA-based two-party signature scheme suggested by Buldas et al. in 2017 paper. This thesis proposes a Smart-ID-like solution based on Two-Party ECDSA protocol suggested by Lindell Yehuda in his 2017 paper. The thesis finds that the suggested ECDSA solution is able to provide the same security features as the current RSA-based Smart-ID solution, but with improved efficiency – more efficient key exchange, smaller signature size and does not require scalable secure storage on the server side. The security proof of the suggested ECDSA solution is not provided. However, the thesis provides a brief security analysis of the solution and the intuition why the suggested solution might be secure. The prototype implementation of the solution is also provided.

Keywords:
Smart-ID, Two-Party ECDSA, digital signature, mobile device, threshold cryptography

CERCS: T-120 Systems engineering, computer technology

Kahepoolne ECDSA protokoll Smart-ID jaoks

Lühikokkuvõte: Smart-ID on digitaalallkirja lahendus, mis põhineb lävendi krüptograafial, kus kaks osapoolt (mobiilseade ja server) teevad omavahel koostööd võtmete genereerimise ja allkirjastamise protsessis. Praegune Smart-ID lahendus kasutab RSA-l põhinevat kahepoolset allkirjastamise skeemi, millele panid aluse Buldas et al. oma artiklis aastal 2017. Käesolevas magistritöös pakub autor välja Smart-ID-le sarnase lahenduse, mis põhineb Lindell Yehuda poolt 2017. aastal oma artiklis välja toodud kahepoolsel ECDSA protokollil. Magistritöö tulemusel selgub, et töös välja pakutud ECDSA-l põhinev lahendus on võimeline tagama samal tasemel turvalisust nagu praegune RSA-l põhinev Smart-ID lahendus, olles seejuures veelgi efektiivsem järgnevates aspektides: efektiivsem võtmete vahetus; väiksem allkirja maht ning see ei vaja serveri poolelt skaleeruvat turva salvestit. Välja pakutud ECDSA lahenduse turvalisuse tõendust ei ole töös käsitletud, kuid autor on koostanud lühida turvaanalüüsi ja toob välja nägemuse, miks see lahendus võiks osutuda turvaliseks. Töös on esitatud ka lahenduse prototüübi implementatsioon.

Võtmesõnad:
Smart-ID, Kahepoolne ECDSA, digitaalne allkiri, mobiilseade, läve krüptograafia

CERCS: T120 Süsteemitehnoloogia, arvutitehnoloogia

Contents

1 Introduction

Secure storage of the keys has always been a relevant topic in digital signing. In order to provide high security, secret keys are usually held in specific hardware devices named Hardware Security Module (HSM). These devices provide physical protection as well as enable to securely perform cryptographic operations with the keys such as encryption and decryption. Modern smartphones rarely provide secure storage on a hardware level, because such protection is costly. A useful approach to eliminate such necessity is to use threshold cryptography. Threshold cryptography is a good method to significantly increase security of the secret keys by distributing them among multiple parties and requiring the parties to collaborate when performing cryptographic operations. Although threshold cryptography is not new, several novel approaches have been proposed within the past years for its two-party case, which fits best digital signing with smartphones [6, 7, 10, 19].

One of the companies to implement a two-party digital signature algorithm in an actual product is Estonian company SK ID solutions. They have used a Two-Party Rivest–Shamir–Adleman (RSA) scheme that was proposed in 2016 by Ahto Buldas, Aivo Kalu, Peeter Laud and Mart Oruaas in the paper "Server-Supported RSA Signatures for Mobile Devices" [6]. The scheme has then been referred to by the name of the product – the Smart-ID scheme. Apart from distributed key and signature generation, the Smart-ID scheme included special additive sharing for the mobile device's private key. This private key sharing obliged mobile device and server to collaborate not only in the signature creation but also in the ownership claiming of the mobile device's private key. The Smart-ID has been acknowledged to provide a digital signature with smartphones of the same level of security as is provided with different hardware security devices such as smart cards.

Since the beginning of 2000s, Elliptic Curve Cryptography (ECC) has gained popularity, because it yielded the same amount of security as RSA but using significantly shorter keys. However, it is hard to achieve an efficient threshold protocol for Elliptic Curve Digital Signature Algorithm (ECDSA), especially in the two-party case, which has no honest majority. In 2017, Yehuda Lindell proposed an efficient Two-Party ECDSA protocol in his paper "Fast Secure Two-Party ECDSA Signing" [19]. This method is particularly interesting for the Smart-ID scheme since it requires significantly less storage on the server side as well as does not need the secret sharing of the mobile device's private key.

This thesis adapts the protocol proposed by Yehuda Lindell to the Smart-ID scheme with several modifications in order to fulfill the security requirements Smart-ID scheme currently provides. The thesis analyzes several aspects of the new protocol as well as proposes different decisions based on the implementation expectations. This work also includes a proof of concept implementation of the proposed Two-Party ECDSA protocol that could serve as a starting point in order to integrate the protocol into an actual product.

The first chapter of the thesis describes the cryptographic preliminaries that are required in order to understand different concepts behind the Two-Party RSA protocol used by the Smart-ID scheme and the proposed Two-Party ECDSA protocols. In the second chapter, a brief overview of the current Smart-ID Two-Party RSA protocol is given. The third chapter provides the description of the Two-Party ECDSA protocol proposed by Yehuda Lindell. In the fourth chapter, the Two-Party ECDSA protocol that is proposed to be used for the Smart-ID scheme is described. The chapter also includes analysis of different security aspects of the protocol in the context of Smart-ID. Finally, the last chapter provides comparison of the current RSA-based Smart-ID protocol with the proposed ECDSA-based protocol, taking into account different aspects – storage, performance, implementation and security.

2 Cryptographic preliminaries

This chapter intends to briefly cover cryptographic preliminaries that are relevant for Two-Party ECDSA as well as for current Smart-ID RSA scheme.

2.1 RSA cryptosystem

RSA cryptosystem is one of the first cryptosystems to use an asymmetric cryptographic algorithm, where a party would have two different keys for encryption and decryption, one that can be shared publicly and one that should be kept secret. It was first published in 1977 in the paper “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems” [28].

RSA cryptosystem relies on the difficulty of factoring a large composite number. In other words, it is hard to decompose such a number into a product of two smaller numbers, in case of RSA, prime numbers. Integer factorization is currently the most efficient method known to break the RSA cryptosystem. However, there could also be other methods. This problem of finding out plaintext given only RSA public key and ciphertext has become referred to as the RSA problem. There is no proof that the RSA problem is computationally difficult as well as there is no known method to efficiently solve it.

This section intends to give a brief overview of RSA key generation, signing and verification algorithms.

2.1.1 Key generation

In order to generate an RSA key pair a party has to do the following steps:

1. Generate a cryptographically secure two different random large prime numbers p and q of equivalent length;

2. Calculate $n = p \cdot q$;

3. Calculate $\varphi(n) = (p-1)(q-1)$;

4. Choose e from range $1 < e < \varphi(n)$ and such that satisfies $gcd(e, \varphi(n)) = 1$. The standard value used today is $e = 65537$;

5. Calculate $d = e^{-1} \mod \varphi(n)$;

6. (n, e) is a public key and (n, d) is a private key.

2.1.2 Signing

A party has a private key (n, d). Party intends to sign a message m. Sign message m with private key (n, d):

1. Using a hash function calculate the hash of the message m to be signed:

$$h = hash(m)$$

2. Pad the resulting hash h to the byte-length of modulus (PKCS#1 v1.5 padding scheme is usually used in practice);

3. Calculate signature using the padded h and the private key (n, d):

$$s = h^d \mod n$$

2.1.3 Verification

To verify the signature, the verifier needs a public key (n, e) and signature s. Successful verification process assures that the message m was indeed signed by the corresponding private key. Verify signature s is correct against public key (n, e):

1. Using a hash function calculate the hash of the message m that was signed:

$$h = hash(m)$$

2. Decrypt the message using public key (n, e):

$$h = s^e \mod n$$

3. Remove padding from the decrypted message;

4. Compare the resulting hash with the hash calculated in step 1. If they are equal, verification is successful.

2.2 Paillier cryptosystem

Paillier cryptosystem is a probabilistic asymmetric algorithm for public key cryptography that is based on the hardness of factoring the product of two primes as in RSA. It was invented and named after Pascal Paillier in 1999. In his paper [26], Pascal Paillier proposed to use Composite Residuosity Class Problem for creating a new encryption scheme, which is why Paillier cryptosystem security relies upon the Decisional Composite Residuosity Assumption (DCRA). This means that given an integer z and a composite of two primes n, it is not possible to identify whether z is an n-residue modulo n^2. In other words, find x such that $z = x^n \mod n^2$.

The Paillier encryption scheme is most known for its homomorphic properties. The homomorphic encryption enables computations while protecting the confidentiality of the data. This property is especially important for the protocol described later in this work.

2.2.1 Key generation

The described key generation includes optimization described in Section 13.2 of the book "Introduction to Modern Cryptography" (2014) by Jonathan Katz and Yehuda Lindell [16]. This optimization allows much simpler calculation of the key pair, if p and q are chosen of equivalent length. In order to generate a Paillier key pair a party has to do the following steps:

1. Generate a cryptographically secure two different random large prime numbers p and q of equivalent length;
2. Calculate $\varphi(n) = (p-1)(q-1)$;
3. Calculate $n = p \cdot q$;
4. Calculate $g = n + 1$;
5. Calculate $\lambda = \varphi(n)$;
6. Calculate $\mu = \lambda - 1 \mod n$;
7. (n, g) is a public key and (λ, μ) is a private key.

In order to raise the performance of the encryption and decryption operations, n^2 can be precomputed and then reused when needed.

2.2.2 Encryption

A party has a public key (n, g). In order to encrypt the message m a party has to do the following:

1. Check that message m is in range $0 \leq m < n$ or reject;
2. Generate a cryptographically secure random number r from range $0 < r < n$ and such that satisfies $gcd(r, n) = 1$;
3. Calculate ciphertext c of message m:

$$c = g^m r^n \mod n^2$$

Since a random number is used in the encryption operation, the resulting ciphertext will be nondeterministic unlike in plain RSA. This is an important property of Paillier encryption scheme.

2.2.3 Decryption

A party has a private key (λ, μ) and a public key (n, g). In order to decrypt the ciphertext c a party has to do the following:

1. Check that ciphertext $c < n^2$ or reject;
2. Calculate m from ciphertext c:

$$L(x) = \frac{x-1}{n}$$

$$m = L(c^{\lambda} \mod n^2) \cdot \mu \mod n$$

2.2.4 Homomorphic properties of Paillier cryptosystem

The Paillier cryptosystem is most notable for its homomorphic properties. It is a partially homomorphic cryptosystem since it only has additive property to the full extent. This means that having two ciphertexts of two numbers, a party can compute the ciphertext of the sum of these two numbers. For instance, having ciphertexts of $a = 5$ and $b = 7$, it is possible to compute ciphertext of 12:

$$enc(a) + enc(b) = enc(a) \cdot enc(b) \mod n^2 = enc(a + b)$$

It is also possible to perform addition of a scalar to ciphertext by doing the following computation:

$$enc(a) + b = enc(a) \cdot g^b \mod n^2 = enc(a + b)$$

The Paillier encryption does not allow to compute the ciphertext of a multiplication of two numbers, having only ciphertexts of these numbers, however, it allows a scalar multiplication. This homomorphic scalar multiplication is as important for the protocol described in this thesis as the homomorphic addition. It is achieved by performing the following computation:

$$enc(a) \cdot b = enc(a)^b \mod n^2 = enc(a \cdot b)$$

2.3 Elliptic Curve Cryptography

ECC is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. The base assumption of ECC is that finding the discrete logarithm of a random elliptic curve element with respect to a publicly known base point is infeasible. This is known as Elliptic Curve Discrete Logarithm Problem (ECDLP). Elliptic curve cryptographic schemes were proposed independently in 1985 by Neal Koblitz [18] and Victor Miller [22], however, they entered wide use only in the beginning of 2000s. Elliptic curves are applicable for key agreement, digital signatures, encryption, pseudo-random generators and other tasks.

Elliptic curve is a plane curve over a finite field which consists of the points satisfying the equation:

$$y^2 = x^3 + ax + b$$

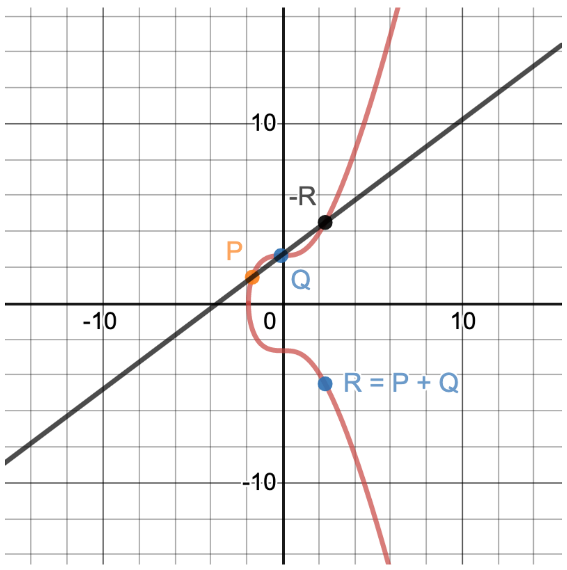


Figure 1. Point addition on secp256k1.

Figure 1 shows elliptic curve $secp256k1$ as well as one of the most fundamental operations of the ECC – point addition. The values of a and b determine the shape of the curve. The coordinates have to be chosen from a fixed finite field. The security of ECC relies on the ability to compute multiplication of a point, but inability to compute the multiplicand given the original and product points. The size of the elliptic curve determines the difficulty of the problem. [18, 22]

To be able to use ECC, first, all parties have to agree on the elements that define the elliptic curve. Such elements are known as the domain parameters of the scheme. Elliptic curve is defined over a finite field. This field could either be a primary field or a binary field. In the prime case the finite field contains p elements, where p is an odd prime number. In the binary case the field contains 2^m elements and is represented by the irreducible binary polynomial $f(x)$ of degree m. Binary fields are not as efficient in software implementations compared to prime fields where large multipliers are available [23]. The constants a and b come from the defining equation of the curve. The cyclic subgroup is defined by its base point G. The n parameter is the order of G, such that $n \times G$ would be equal to the point at infinity of the curve. Finally, h is the cofactor, which is commonly an integer smaller or equal to 4. The result is that in the prime case, the domain parameters are (p, a, b, G, n, h), while in the binary case, they are (m, f, a, b, G, n, h). [4, 15]

The domain parameters must be validated before use if there is no assurance that they were generated by a trusted party. Since generation of domain parameters is a time-consuming process that requires non-trivial implementation, several standard bodies published domain parameters of elliptic curves for several common field sizes. These domain parameters are commonly known as “standard curves” or “named curves”. Such named curves are referenced by their name or the unique object identifier defined in the standard documents published by National Institute of Standards and Technology (NIST) [17], Standards for Efficient Cryptography Group (SECG) [5] and ECC Brainpool [20].

Here are the domain parameters of one of the most popular elliptic curves suggested by SECG – $secp256k1$ [5]. The elliptic curve domain parameters over F_p associated with a Koblitz curve $secp256k1$ are specified by the sextuple $T = (p, a, b, G, n, h)$ where the finite field F_p is defined by:

$$\begin{aligned} p &= \text{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF} \\ &\quad \text{FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F} \\ &= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1 \end{aligned}$$

The curve $E : y^2 = x^3 + ax + b \mod n$ is defined by:

$$a = 00$$

$$b = 07$$

The base point G coordinates are:

$$\begin{aligned} x = \text{79BE667E F9DCBBAC 55A06295 CE870B07} \\ \text{029BFCDB 2DCE28D9 59F2815B 16F81798} \end{aligned}$$

$$\begin{aligned} y = \text{483ADA77 26A3C465 5DA4FBFC 0E1108A8} \\ \text{FD17B448 A6855419 9C47D08F FB10D4B8} \end{aligned}$$

Finally, the order n of G and the cofactor h are:

$$\begin{aligned} n = \text{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE} \\ \text{BAAEDCE6 AF48A03B BFD25E8C D0364141} \end{aligned}$$

$$h = 01$$

One of the main properties of ECC that made it so popular is to yield the same amount of security as RSA but using shorter keys. As can be seen by the security strength comparison in Table 1, a 256-bit elliptic curve key provides the same level of security as the RSA key of 3072 bits.

Table 1. Comparison of security level provided by RSA and ECC [12].

Security Strength	**Factoring Modulus (RSA)**	**Elliptic Curve**
80 bits	1024 bits	160 bits
112 bits	2048 bits	224 bits
128 bits	3072 bits	256 bits
192 bits	7680 bits	384 bits
256 bits	15360 bits	512 bits

2.4 Elliptic Curve Diffie-Hellman

Although it is only in the 2000s that the elliptic curves started to be adopted in practice, already in 1985 Victor S.Miller proposed to use elliptic curves for Diffie-Hellman (DH) key exchange protocol. Victor S.Miller claimed that DH using elliptic curve would be approximately 20% faster than the original implementation by Whitfield Diffie and Martin Hellman [22]. Elliptic Curve Diffie-Hellman (ECDH), just as its predecessor DH, is a key agreement protocol that allows two parties to establish a shared secret over an insecure channel. The prerequisites are that both parties have to agree on all the elements defining the elliptic curve, that is, the domain parameters of the scheme, and each has to generate an elliptic curve key pair. The agreed secret, a secret key, is most commonly then used to encrypt subsequent communications using a symmetric-key cipher.

ECDH is designed to provide a variety of security goals depending on its application. It can provide unilateral implicit key authentication, mutual implicit key authentication, known-key security, and forward secrecy, depending on factors such as whether public keys are exchanged in an authentic manner or not, and whether key pairs are ephemeral or static. [4]

The ECDH protocol can be seen in Figure 2. Alice and Bob represent two parties who participate in the protocol. Below is a detailed description of ECDH protocol steps:

1. Alice and Bob agree on domain parameters of the elliptic curve (p, a, b, G, n, h);
2. Alice and Bob each generate an Elliptic Curve (EC) key pair based on the previously agreed domain parameters. EC key pair consists of a private key d, which is a randomly selected integer in range $0 < d < n$, and a public key Q, which is

computed by multiplication of the point on the curve G with the scalar d:
Alice: $d_1 \in 0 < d_1 < n;\ Q_1 = G \times d_1$
Bob: $d_2 \in 0 < d_2 < n;\ Q_2 = G \times d_2$

3. Alice sends its public key Q_1 to Bob;
4. Bob validates public key Q_1. The instructions for public key validation can be found in Section 6.2 of [15];
5. Bob computes the point $Q = Q_1 \times d_2$. Since the result Q is a point on the curve, Bob takes its x coordinate and uses it as a resulting shared secret;
6. Bob sends its public key Q_2 to Alice;
7. Alice validates public key Q_2. The instructions for public key validation can be found in Section 6.2 of [15].
8. Alice verifies that Q_1 is a point on the agreed curve and computes the point $Q = Q_2 \times d_1$. Since the result Q is a point on the curve, Alice takes its x coordinate and uses it as a resulting shared secret. Because the order in which the points are multiplied does not affect the result, Alice and Bob end up with the same number as a shared secret.

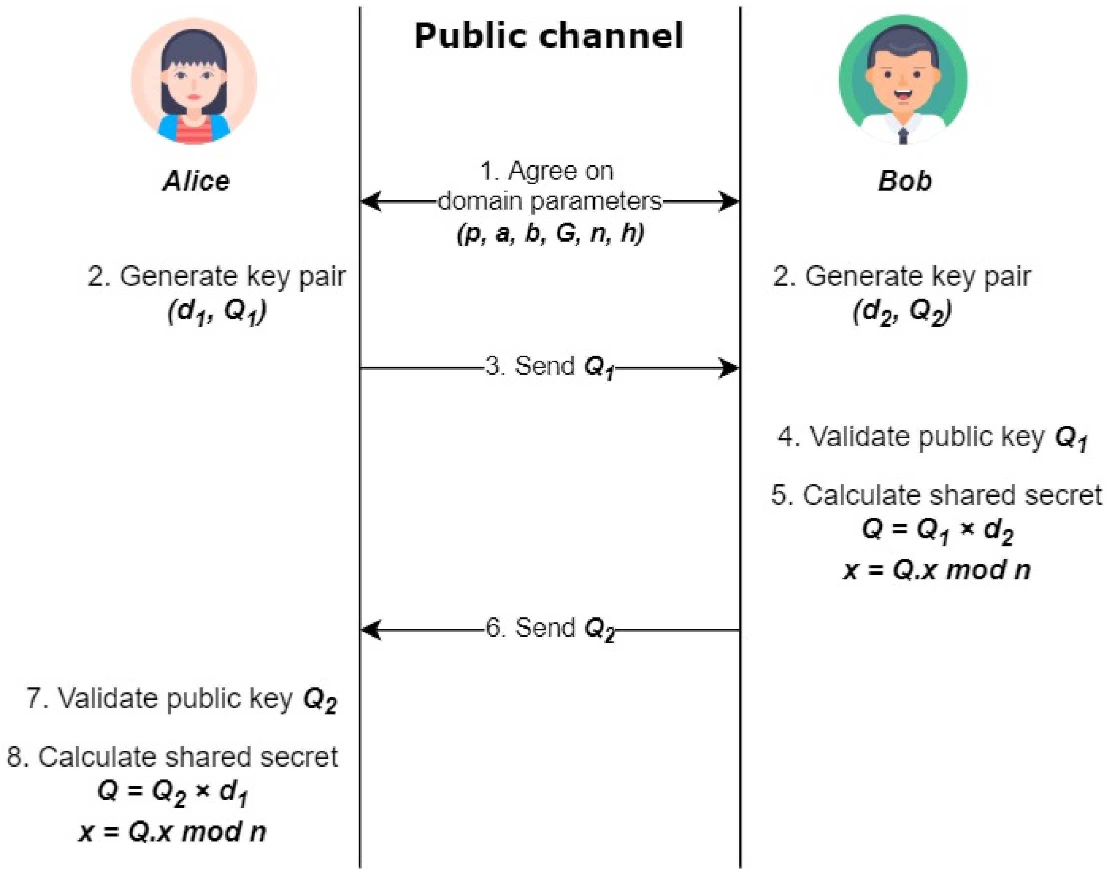

Public channel
Alice
Bob
1. Agree on domain parameters (p, a, b, G, n, h)
2. Generate key pair (d_1, Q_1)
2. Generate key pair (d_2, Q_2)
3. Send Q_1
4. Validate public key Q_1
5. Calculate shared secret $Q = Q_1 \times d_2$ $x = Q.x \bmod n$
6. Send Q_2
7. Validate public key Q_2
8. Calculate shared secret $Q = Q_2 \times d_1$ $x = Q.x \bmod n$

Figure 2. ECDH protocol.

2.5 Elliptic Curve Digital Signature Algorithm

ECDSA is the elliptic curve analogue of the Digital Signature Algorithm (DSA), that was first proposed in 1992 by Scott Vanstone in response to NIST's request for public comments on their first proposal for DSS [32]. In the end of 1990s it was accepted as a standard by different organizations such as ISO, ANSI, IEEE, NIST and FIPS [15].

In order to describe Two-Party ECDSA it is necessary to first understand the standard ECDSA's processes of key generation, signing and verification. Below is a brief overview of ECDSA in accordance with ANSI X9.62 standard [1].

2.5.1 Key generation

A party intends to generate a EC key pair. First, party has to choose an elliptic curve, for instance, $secp256k1$. Party retrieves curve's domain parameters (p, a, b, G, n, h) provided by SECG [5]. The key generation includes the following steps:

1. Generate a cryptographically secure random secret number d from range $0 < d < n$;

2. Calculate a point on the curve Q with coordinates (x, y). This is done by multiplying the base point G with the private key d:

$$Q = G \times d$$

3. (d, Q) is a resulting EC key pair, where d is a private key and Q is a public key. Q can be shared with other parties, while d should be kept secret.

2.5.2 Signing

A party has a private key d. Party intends to sign the message m. The ECDSA signing process consists of the following steps:

1. Using a hash function calculate the hash of the message m to be signed:

$$e = hash(m)$$

2. Calculate z by taking $len(n)$ leftmost bits of e. The value z can be greater than the group order n, but not longer in bits;

3. Generate a cryptographically secure random secret number k from range $0 < k < n$. This secret number shall be protected with the same care as the private key d;

4. Calculate the point R by multiplying the base point G with the scalar k:

$$R = G \times k$$

5. Calculate value r by taking x coordinate of R and reducing it modulo n:
$$r = R.x \mod n$$
In case $r = 0$, return back to step 3 and try again;

6. Calculate value s:
$$s = k^{-1}(z + r \cdot d) \mod n$$
In case $s = 0$, return back to step 3 and try again;

7. Choose the minimum valid s parameter. In ECDSA, there are always two valid signatures (r, s) and $(r, n - s)$. It makes sense to choose the one that is the smallest:
$$s = min(s, n - s)$$
The final ECDSA signature is (r, s).

It is crucial for k to be random and unpredictable, otherwise, the private key d can be recovered. Furthermore, it is crucial to select a different secret nonce k for different messages signed with the same private key. One way how such reliance on randomness can be removed is by deriving the secret nonce k from private key and message, which results in a deterministic signature. This technique is described in RFC6979 [27].

2.5.3 Verification

To verify the signature, the verifier needs a public key Q and signature (r, s). Successful verification process assures that the message m was indeed signed by the corresponding private key. Verify signature (r, s) is correct against public key Q:

1. Validate public key Q (see Section 2.4);
2. Verify that r and s are integers in the intervals $0 < r < n$ and $0 < s < n$, or else fail verification;
3. Calculate $hash(m)$ using the same hash function as was used for signing and convert the resulting bytes to an integer e;
4. Calculate $w = s^{-1} \mod n$;
5. Calculate $u_1 = e \cdot w \mod n$ and $u_2 = r \cdot w \mod n$;
6. Calculate point $X = u_1 \times G + u_2 \times Q$. If resulting X point is equal to identity element O (identity element is $n \times G$), the signature is rejected;
7. Take the integer value of x coordinate of X and reduce it modulo n;
8. Verification is successful if the resulting x is equal to r:
$$x = r \mod n$$

3 Two-Party RSA protocol used by Smart-ID

In 2016 Ahto Buldas, Aivo Kalu, Peeter Laud and Mart Oruaas in their paper "Server-Supported RSA Signatures for Mobile Devices" proposed a method for shared RSA signing between the mobile device and the server [6]. This scheme modified the RSA scheme of the Distributed Multiprime RSA scheme proposed by Damgard, Mikkelsen and Skeltved [29]. The main difference was that the proposed scheme addressed the vulnerability to brute-force attacks by additively sharing the mobile device's RSA private exponent between the server and the mobile device. The proposed Two-Party RSA scheme included the following features:

1. The server alone is unable to create valid signatures;
2. Having the mobile device's share, it is not possible to create a signature without the server;
3. The server detects the cloned mobile device's shares and blocks the service;
4. Having the password-encrypted mobile device's share, the brute-force attacks cannot be performed without alerting the server;
5. The composite RSA signature is an ordinary RSA signature and verifies with standard crypto-libraries.

Damgard, Mikkelsen and Skeltved proved in their paper that the hardness of the Multiprime RSA problem would be as hard as the RSA problem, therefore, the proposed protocol would also rely on the difficulty of factoring large composite integers.

The Estonian company SK ID Solutions AS has implemented the modified Multiprime RSA scheme in their product Smart-ID [31] in order to provide authentication and signing services, which is why it is referred to as Smart-ID scheme. For the purpose of simplification, the roles on mobile device and server sides are generalized into Mobile and Server throughout this work, ignoring the distinction, for instance, between Smart-ID Secure-Zone, Core and HSM. Also, this work completely ignores the cloning prevention measures that take place in Smart-ID protocol, because these measures do not affect the two-party scheme.

3.1 Key exchange

The first stage of the RSA scheme is the key exchange, which can be seen in Figure 3. The key exchange consists of the steps described below:

1. Mobile generates the 3072-bit RSA key pair with the private key (n_m, d_m) and public key $(n_m, e = 65537)$;

2. During key generation Mobile also calculates Euler's totient function of the modulus n_m and temporarily stores it. The function looks as follows:

$$\varphi(n_m) = (p-1)(q-1)$$

   where p and q are distinct prime numbers chosen randomly in the RSA key generation process;

3. Mobile generates a random 3072-bit number d_{m_1};

4. Mobile calculates d_{m_2}:

$$d_{m\_2} = d_m - d_{m\_1} \mod \varphi(n_m)$$

   The result of this operation is secret sharing of the Mobile's private exponent d_m by splitting it into two parts – d_{m_1} and d_{m_2}. These parts alone do not contain any information about d_m. One part (d_{m_1}) will be used by Mobile and another one (d_{m_2}) by Server. Mobile deletes the private exponent d_m;

5. Mobile converts d_{m_1} to 384-byte value and encrypts it with 128-bit Advanced Encryption Standard (AES) in Cipher Block Chaining (CBC) mode without padding – $enc_{PIN}(d_{m\_1})$. The AES key is derived from the Mobile user's PIN code. Mobile deletes the share d_{m_1};

6. Mobile sends its public key modulus n_m and the share of its private exponent d_{m_2} to Server. Mobile deletes the share d_{m_2};

7. Server generates the 3072-bit RSA key pair with private key (n_s, d_s) and public key $(n_s, e = 65537)$;

8. Server calculates the constants α and β using the Extended Euclid's algorithm, such that:

$$\alpha \cdot n_m + \beta \cdot n_s = 1$$

9. Server calculates the composite public modulus:

$$n = n_m \cdot n_s$$

   Therefore, the composite public key will be $(n, e = 65537)$.

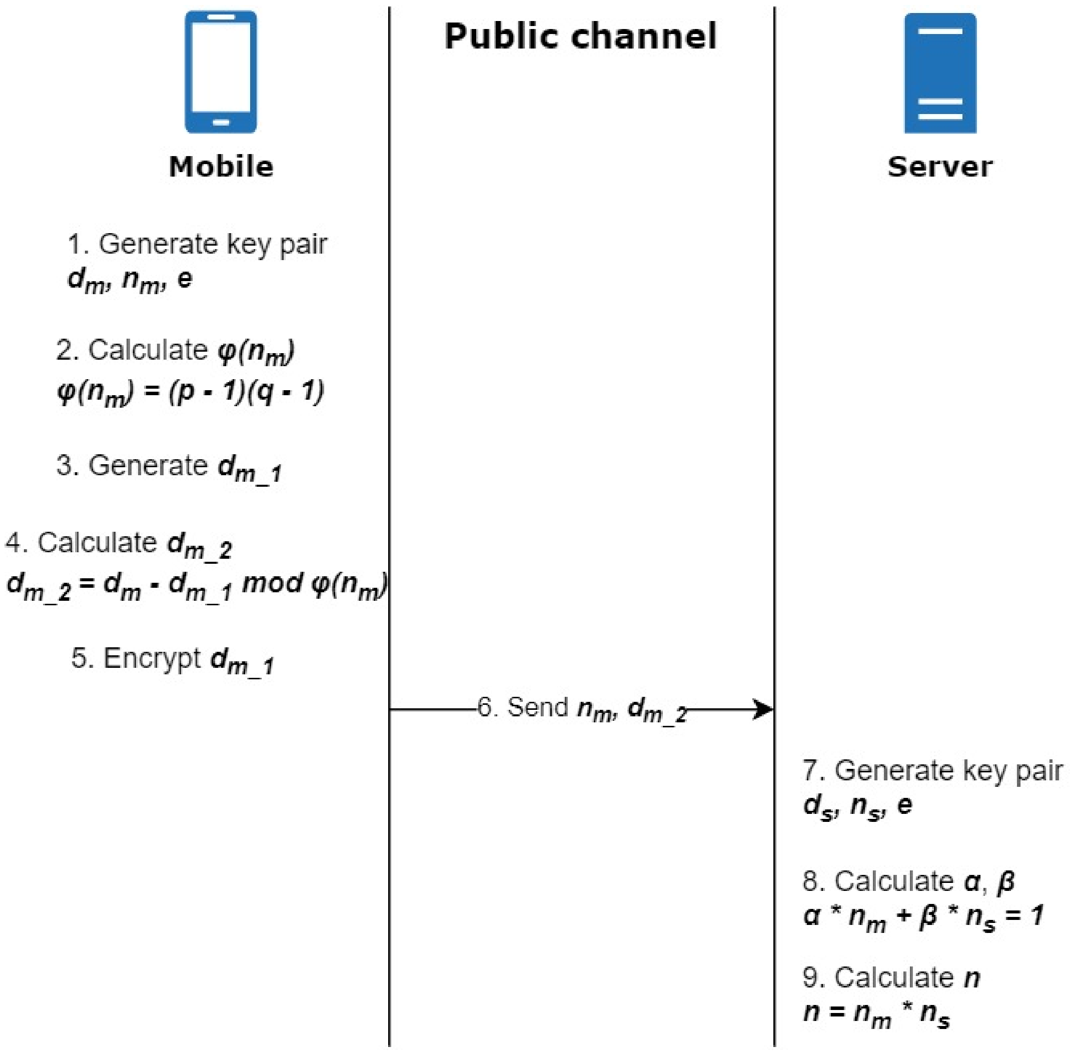


Figure 3. Two-Party RSA key exchange protocol used by Smart-ID.

3.2 Signing

Once the key exchange protocol has successfully finished, it is possible to proceed with the signing. The signing protocol can be seen in Figure 4 and detailed description of its steps is presented below:

1. Server sends message m to be signed to Mobile;

2. Mobile prompts the user to enter PIN code. Once entered, the share of the Mobile's private key exponent d_{m_1} is decrypted;

3. Mobile calculates the signature share:

   $$s_{m\_1} = m^{d_{m\_1}} \mod n_m$$

4. Mobile sends the signature share s_{m_1} to Server;

5. Server calculates other share of the Mobile's signature using Mobile's private exponent share d_{m_2}:

   $$s_{m\_2} = m^{d_{m\_2}} \mod n_m$$

6. Server combines Mobile's signature shares to calculate the complete Mobile's signature:

   $$s_m = s_{m\_1} \cdot s_{m\_2} \mod n_m$$

7. Server verifies the Mobile's signature s_m using the Mobile's public key (n_m, e):

   $$m = s_m^e \mod n_m$$

   If the verification is successful, the correct PIN code was used on the Mobile to decrypt Mobile's private exponent share d_{m_2}. If incorrect PIN has been entered the server blocks the Mobile's account or introduces delays;

8. Server calculates its signature:

   $$s_s = m^{d_s} \mod n_s$$

9. Server calculates the composite signature:

   $$s = \beta \cdot n_s \cdot s_m + \alpha \cdot n_m \cdot s_s \mod n$$

10. Server verifies the composite signature s using the composite public key (n, e). In order to verify the signature Server performs the following operation:

    $$m = s^e \mod n$$

    This calculated m can now be compared with the message that was signed.

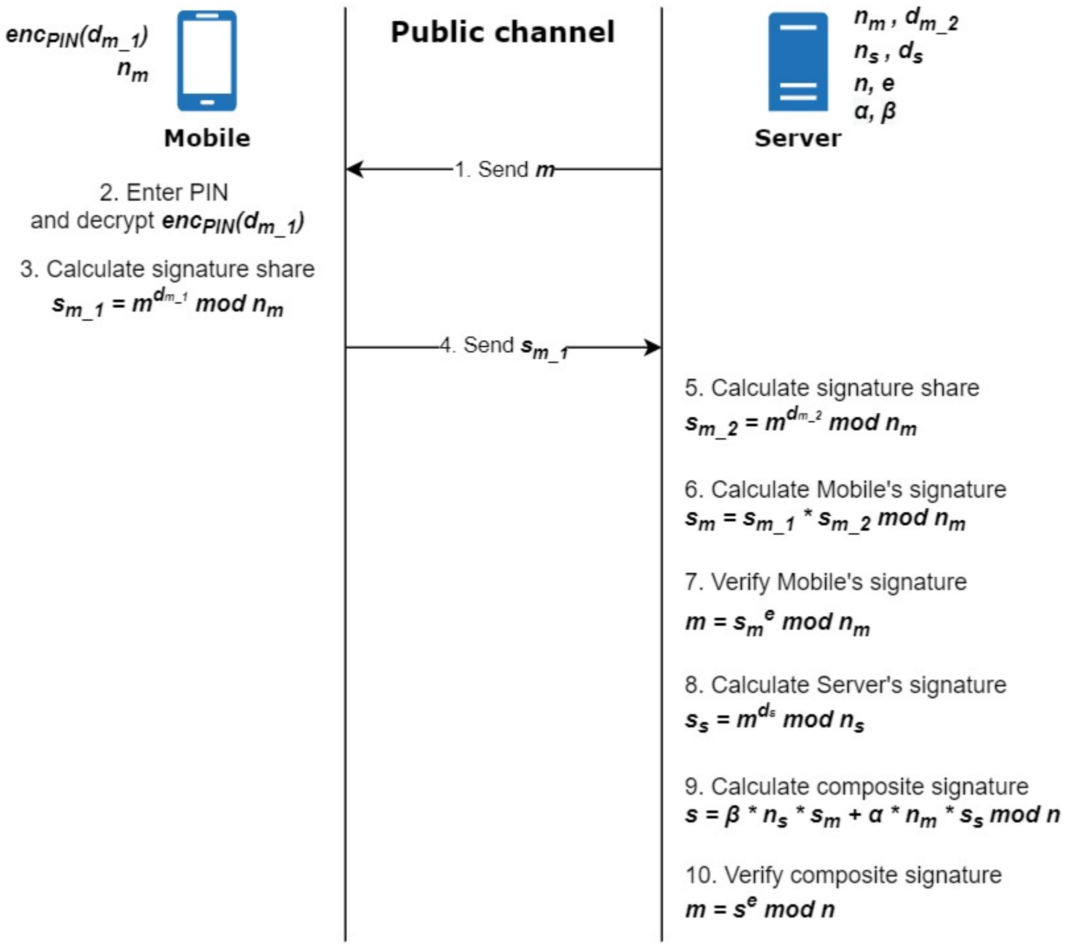

$enc_{PIN}(d_{m\_1})$
n_m
Mobile
Public channel
Server
n_m , d_{m_2}
n_s , d_s
n, e
α, β
1. Send m
2. Enter PIN and decrypt $enc_{PIN}(d_{m\_1})$
3. Calculate signature share $s_{m\_1} = m^{d_{m\_1}} \bmod n_m$
4. Send s_{m_1}
5. Calculate signature share $s_{m\_2} = m^{d_{m\_2}} \bmod n_m$
6. Calculate Mobile's signature $s_m = s_{m\_1} * s_{m\_2} \bmod n_m$
7. Verify Mobile's signature $m = s_m^e \bmod n_m$
8. Calculate Server's signature $s_s = m^{d_s} \bmod n_s$
9. Calculate composite signature $s = \beta * n_s * s_m + \alpha * n_m * s_s \bmod n$
10. Verify composite signature $m = s^e \bmod n$

Figure 4. Two-Party RSA signing protocol used by Smart-ID.

4 Two-Party ECDSA protocol by Yehuda Lindell

In the paper "Fast Secure Two-Party ECDSA Signing" Yehuda Lindell proposed an efficient Two-Party ECDSA protocol [19]. The purpose of the protocol is to securely share keys between two parties that can together efficiently generate a standard ECDSA signature. The problem of threshold cryptography in ECDSA is solved by using homomorphic properties of the Paillier cryptosystem. One party generates a partial ECDSA signature by applying its computations homomorphically on the pre-shared Paillier ciphertext, the other party is able to decrypt the resulting ciphertext and apply its part of the computations.

4.1 Key exchange

This section describes Two-Party ECDSA key exchange according to protocol by Yehuda Lindell. Some details have been left out from the protocol, because they are not relevant in the context of the implementation proposed in this thesis.

4.1.1 EC key generation and exchange

The key exchange is divided into two phases for the purpose of the transparency, although in the original protocol described by Yehuda Lindell there is no such division.

First phase is the generation of EC keys. The main goal of this phase is to generate a private key for each party and to compute a composite public key without revealing any information that could potentially expose any of the private keys. In order to achieve such a goal this phase implements ECDH protocol that is described in Section 2.4. Although the composite public key is not a secret in this scenario, as in the standard ECDH case, ECDH protocol still fits to solve the problem of establishing the shared value based on two secret values.

In addition to the ECDH, the protocol also includes a commitment scheme and zero-knowledge proof for a discrete logarithm. The details of the implementation for these two are described in Yehuda Lindell's paper as well as summarized here. According to Yehuda Lindell for the zero-knowledge proof it is possible to use standard Schnorr signature [30]. As for the commitment scheme, any secure universally-composable scheme can be utilized.

Before starting with the key exchange, the parties need to agree on an elliptic curve and store its domain parameters. Both parties must also have the ability to generate cryptographically secure random numbers. The protocol consists of the following steps (Figure 5):

1. Alice generates a cryptographically secure random secret number d_1 from range $0 < d_1 < \frac{n}{3}$;

2. Alice calculates Q_1 point on the chosen curve by multiplying domain parameter base point G with a scalar value of d_1:

$$Q_1 = G \times d_1$$

3. Alice calculates a zero-knowledge proof of knowledge of d_1, the discrete logarithm of EC point Q_1, which will be referred to as π_1;

4. Alice calculates c_1, which is the commitment for Q_1 and π_1;

5. Alice sends c_1 to Bob, hence committing to Q_1 and π_1;

6. Bob generates a cryptographically secure random secret number d_2 from range $0 < d_2 < n$;

7. Bob calculates the public key Q_2 point on the chosen curve by multiplying domain parameter base point G with a scalar value of d_2:

$$Q_2 = G \times d_2$$

8. Bob calculates a zero-knowledge proof of knowledge of d_2, the discrete logarithm of EC point Q_2, which will be referred to as π_2;

9. Bob sends Q_2 and π_2 to Alice;

10. Alice verifies the proof π_2. In other words, Alice verifies that Bob indeed knows d_2, and that Q_2 is the result of $G \times d_2$;

11. Alice decommits c_1 by sending Q_1 and π_1;

12. Bob verifies the proof π_1. In other words, Bob verifies that Alice indeed knows d_1, and that Q_1 is the result of $G \times d_1$;

13. Alice and Bob separately calculate the composite public key point Q by multiplying the public key point received from the other party with their private key. They end up with the same point on the curve because essentially they both multiply base point G with scalar values of d_1 and d_2, but in different order.
Alice: $Q = Q_2 \times d_1 = (G \times d_2) \times d_1$
Bob: $Q = Q_1 \times d_2 = (G \times d_1) \times d_2$
The resulting EC point Q is the composite public key. The d_1 and d_2 are corresponding private keys. All the remaining variables apart from the composite public key and private keys should now be deleted. After the first phase of key exchange, Alice ends up with (d_1, Q) and Bob with (d_2, Q).

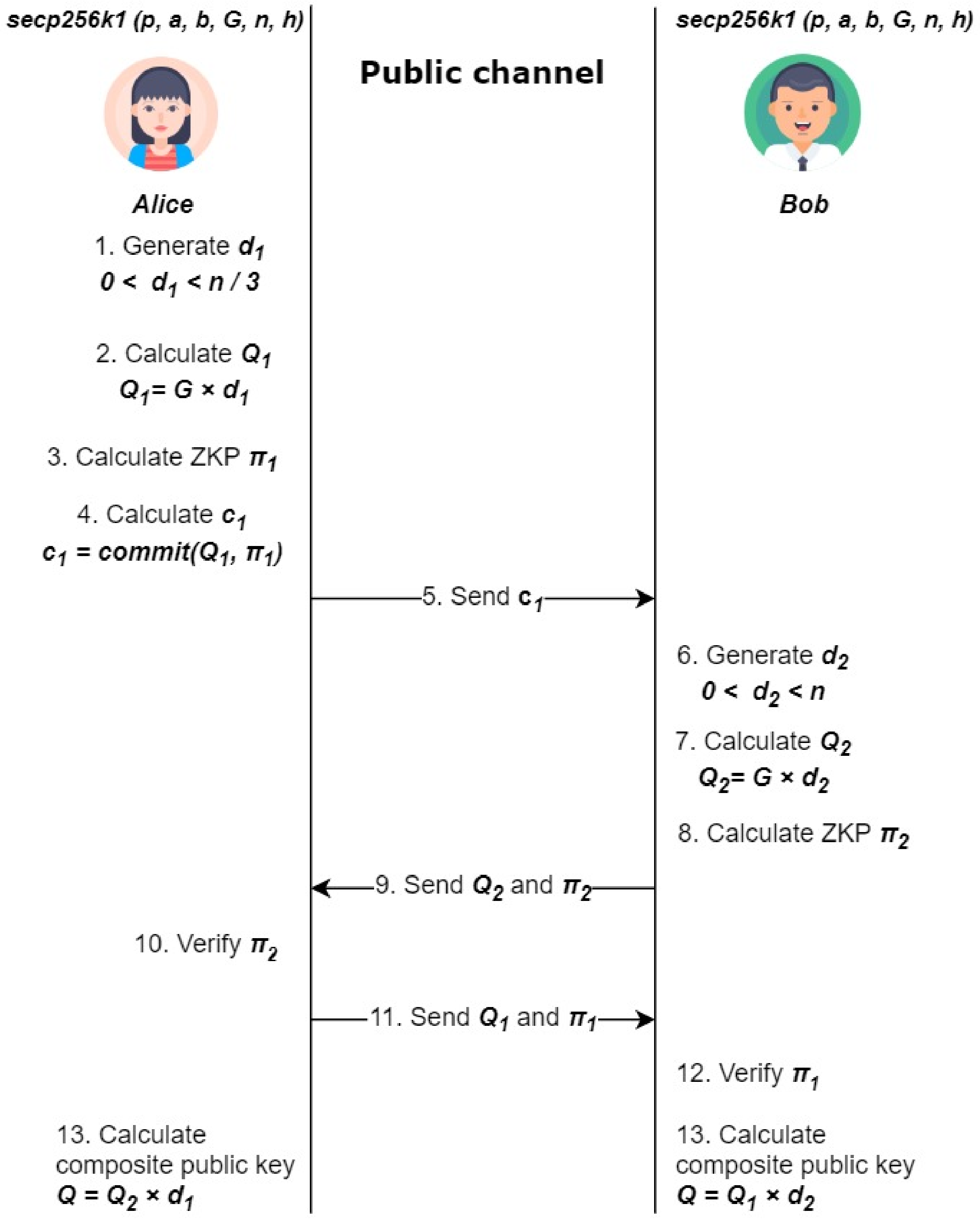


Figure 5. EC key generation and exchange protocol by Yehuda Lindell.

4.1.2 Paillier key generation and exchange

Once the first phase is finished both parties have the corresponding private keys and the composite public key stored. The second phase is concerned with the Paillier related actions. Only one party needs to generate a Paillier key pair and store it for future use. The other party needs to receive the Paillier public key as well as the Paillier-encrypted EC private key of the Paillier key pair holder. Paillier key generation and exchange by Yehuda Lindell can be seen in Figure 6. The steps of this protocol are the following:

1. Bob generates a Paillier key pair with public key (n_p, g_p) and private key (λ, μ). Paillier public key n_p and g_p parameters should not be confused with EC domain parameters n and G, they are not related in any way. The Paillier key length should be calculated as follows:

   $$key\_length = max(3 \log a + 1, b)$$

   where a is the length of the order of the curve's base point and b is the minimal secure length for Paillier to achieve the same security strength. For instance, in case of the elliptic curve $secp256k1$ the a would be 256, and b would be 3072, and therefore, the resulting Paillier key length would be 3072 bits [12];

2. Bob sends Paillier public key parameters (n_p, g_p) to Alice. Alice stores (n_p, g_p) for future use;

3. Bob encrypts its EC private key d_2 with Paillier public key (n_p, g_p). Paillier encryption is described in Section 2.2.2;

4. Bob proves to Alice that a Paillier public key (n_p, g_p) was generated correctly as well as that the encrypted value is Bob's EC private key d_2. This is done via a novel highly efficient zero-knowledge proof that bridges Paillier encryption and EC groups (page 32 in [19]);

5. Bob sends the resulting ciphertext of encrypted EC private key d_2 to Alice. Alice stores the ciphertext $enc(d_2)$ and will repeatedly use it during the signature generation process.

After the key generation and exchange, Alice has the following values:

- (p, a, b, G, n, h) – $secp256k1$ EC domain parameters;
- d_1 – Alice's EC private key;
- Q – composite EC public key;
- $enc(d_2)$ – Bob's EC private key encrypted by Paillier;

- (n_p, g_p) – Paillier public key.

Bob's values:

- (p, a, b, G, n, h) – $secp256k1$ EC domain parameters;
- d_2 – Bob's EC private key;
- Q – composite EC public key;
- (n_p, g_p) – Paillier public key;
- (λ, μ) – Paillier private key.

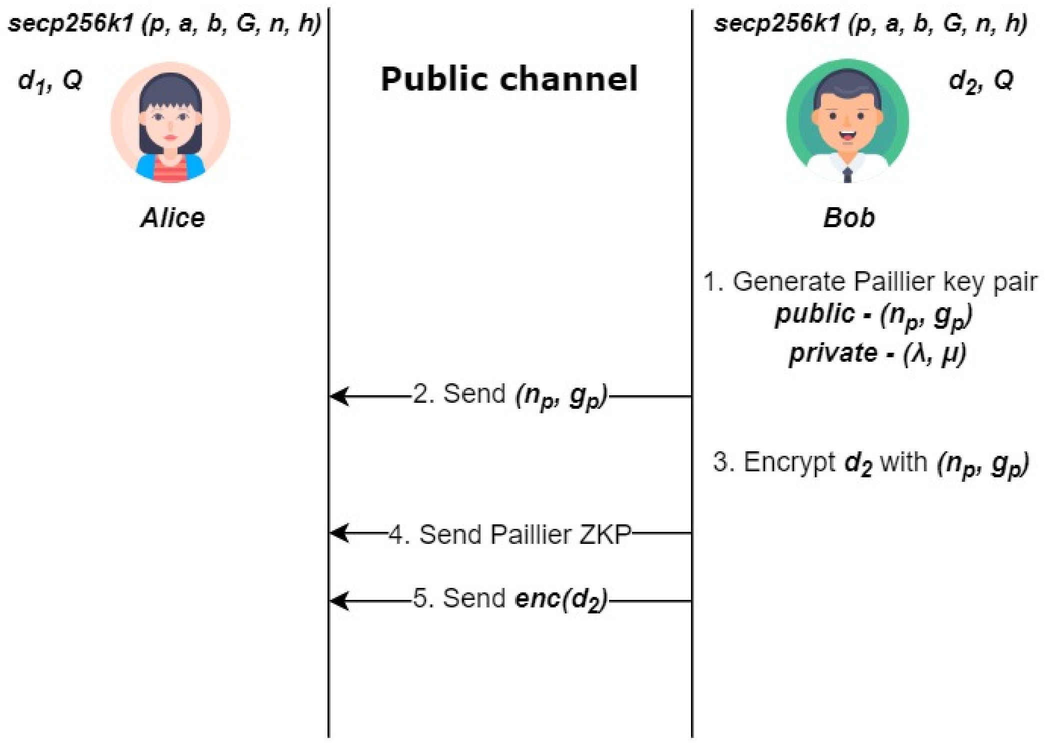


Figure 6. Paillier key generation and exchange protocol by Yehuda Lindell.

4.2 Signing

The signing protocol is divided into two phases for the purpose of the transparency, although in the original protocol described by Yehuda Lindell [19] there is no such division. The first phase is the agreement on ECDSA signature's r value and the second is the collaborative calculation of ECDSA signature's s value. To be able to start with the signing process parties must have gone through the key exchange process that was described previously.

4.2.1 Generation of ECDSA signature's r value

The first phase of the signature generation is the agreement on the signature's value r. This is achieved via ECDH in the same way as in the Two-Party ECDSA key exchange process (Figure 7):

1. Alice generates a cryptographically secure random secret number k_1 from range $0 < k_1 < n$;
2. Alice calculates the R_1 point on the chosen curve by multiplying the base point G with a scalar value of k_1; This operation is exactly the same as when calculating the R point in the standard ECDSA signature protocol:

$$R_1 = G \times k_1$$

3. Alice calculates a zero-knowledge proof of knowledge of k_1, the discrete logarithm of EC point R_1, which will be referred to as π_1;
4. Alice calculates c_1, which is the commitment for R_1 and π_1;
5. Alice sends c_1 to Bob, hence committing to R_1 and π_1;
6. Bob generates a cryptographically secure random secret number k_2 from range $0 < k_2 < n$;
7. Bob calculates the R_2 point on the chosen curve by multiplying domain parameter base point G with a scalar value of k_2:

$$R_2 = G \times k_2$$

8. Bob calculates a zero-knowledge proof of knowledge of k_2, the discrete logarithm of EC point R_2, which will be referred to as π_2;
9. Bob sends R_2 and π_2 to Alice;
10. Alice verifies the zero-knowledge proof for discrete logarithm π_2;
11. Alice decommits c_1 by sending R_1 and π_1;
12. Bob verifies the zero-knowledge proof for discrete logarithm π_1;

13. Alice and Bob separately calculate the composite R point by multiplying the received point from the other party with their secret number. They end up with the same point on the elliptic curve because essentially they both multiply base point G with scalar values of k_1 and k_2, but in different order.
    Alice: $R = R_2 \times k_1 = (G \times k_2) \times k_1$
    Bob: $R = R_1 \times k_2 = (G \times k_1) \times k_2$
    Once that is done, x coordinate of the point R is extracted and reduced modulo n. It would be the r value of the ECDSA signature. The r value must not be zero. If it is, all steps starting from step 1 have to be repeated again (although such occurrence is utterly improbable). All the remaining variables apart from the signature's r value and secret numbers k_1 and k_2 , should now be deleted.

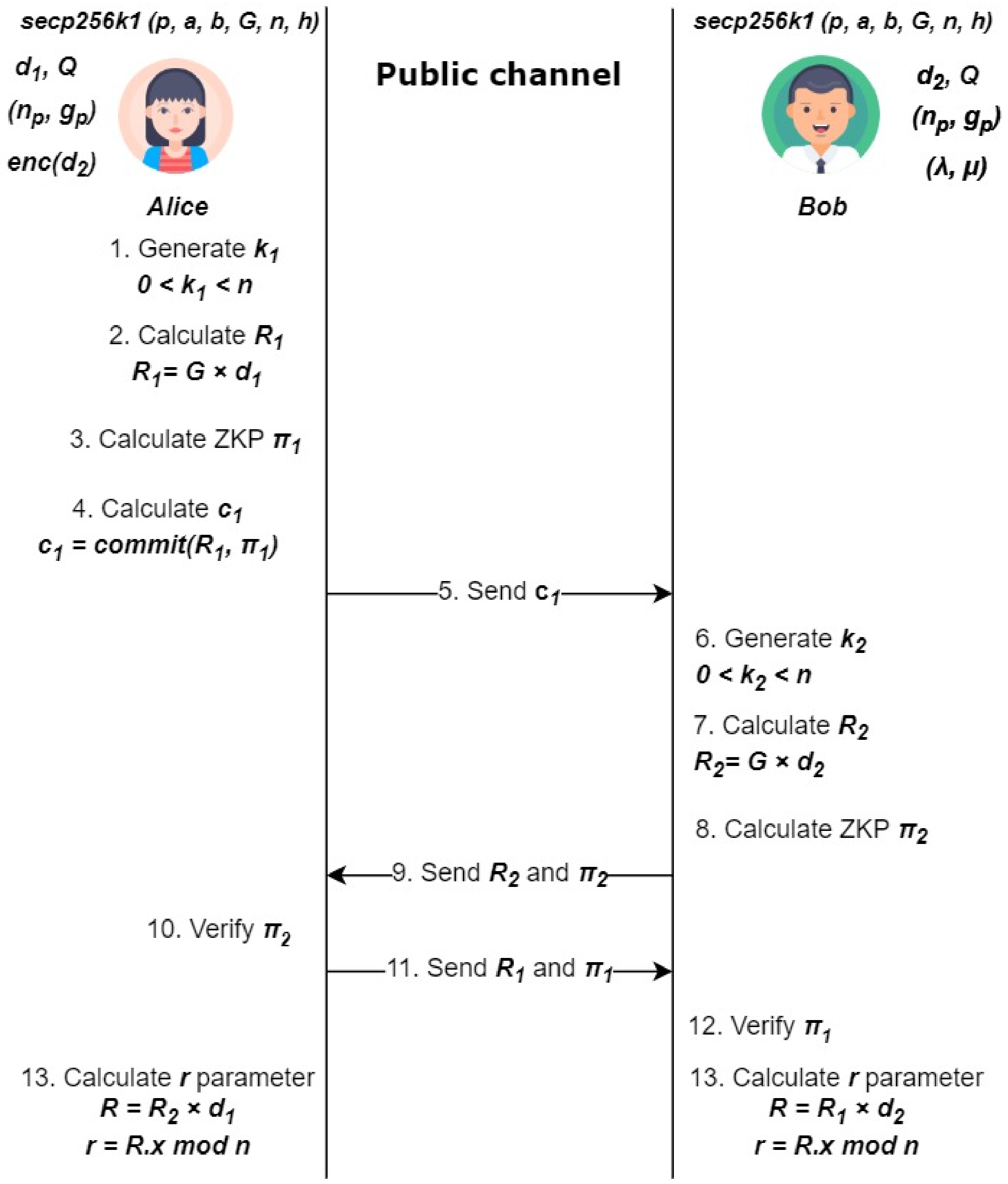


Figure 7. Signature's r value agreement protocol by Yehuda Lindell.

4.2.2 Generation of ECDSA signature's s value

In this process, ECDSA signature's s value is calculated by both parties collaborating (Figure 8):

1. Bob calculates the z value of the ECDSA. In order to do that, Bob calculates the hash of the content intended to be signed and takes its $|n|$ leftmost bits, where $|n|$ is the bit length of the curve's domain parameter n;

2. Bob sends to Alice the z value;

3. Alice generates a cryptographically secure random secret number p from range $0 < p < n^2$;

4. Alice uses previously stored (received during key exchange) Paillier ciphertext of d_2 and calculates the Paillier ciphertext of partial signature s' by performing the following operations:

   (a) Calculate s'_1:
   $$s'_{1\_1} = p \cdot n$$
   $$s'_{1\_2} = z \cdot k_1^{-1} \mod n$$
   $$s'_1 = s'_{1\_1} + s'_{1\_2}$$

   (b) Calculate $enc(s'_1)$ by encrypting s'_1 using Paillier public key (n_p, g_p). Paillier encryption is described in Section 2.2.2;

   (c) Calculate s'_3:
   $$s'_3 = r \cdot d_1 \cdot k_1^{-1} \mod n$$

   (d) Calculate $enc(s'_2)$ using homomorphic multiplication of $enc(d_2)$ by scalar value s'_3:
   $$enc(s'_2) = enc(d_2)^{s'_3} \mod {n_p}^2$$

   (e) Calculate $enc(s')$ using homomorphic addition of $enc(s'_1)$ and $enc(s'_2)$:
   $$enc(s') = enc(s'_1) \cdot enc(s'_2) \mod {n_p}^2$$

   It is important to mention here that this particular way of computation has been chosen on purpose, because it allows the maximum amount of computations in plaintext, therefore, having an effect on performance. Alice deletes the secret number k_1;

5. Alice sends the encrypted partial signature $enc(s')$ to Bob;

6. Bob decrypts the ciphertext of s' with Paillier private key (λ, μ) and public key (n_p, g_p) obtaining value s' in plain form. Paillier decryption is described in Section 2.2.3;

7. Bob calculates ECDSA signature's s value by multiplying s' with the inverse of k_2. It is important to remember that after Paillier decryption all operations should be done with modulus of n (EC domain parameter, not the Paillier public key modulus n_p here):

   $$s = s' \cdot k_2^{-1} \mod n$$

   Choose the smallest s value:

   $$s = min(s, n - s)$$

   The final ECDSA signature is (r, s). Bob deletes the secret number k_2;

8. Bob performs signature verification. If it is not successful then the signing process has to start all over from step 1. The process of signature verification is exactly the same as in standard ECDSA and is described in Section 2.5.3;

9. Bob sends to Alice the final ECDSA signature (r, s).

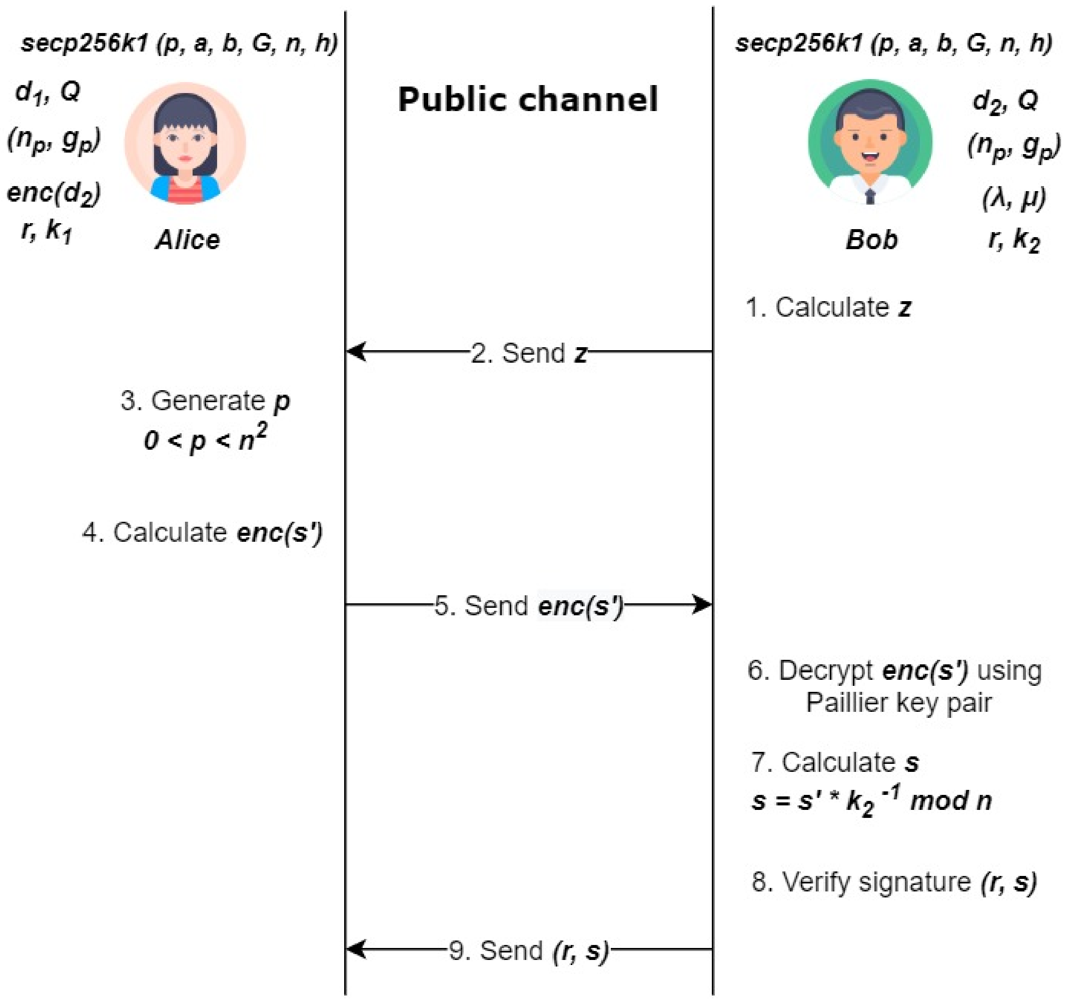


Figure 8. Signature's s value generation protocol by Yehuda Lindell.

5 Two-Party ECDSA protocol for Smart-ID

This chapter describes the Two-Party ECDSA protocol for the Smart-ID scheme providing implementation details and modifications to the original protocol to satisfy the security requirements of the current Smart-ID scheme.

5.1 Protocol

Before describing the protocol, several protocol parameters have to be fixed that later will be used in the implementation. The algorithms and key sizes used in the implementation provide the security strength of 128 bits [12] which is equivalent to the current Smart-ID Two-Party RSA solution. Parties have agreed on the elliptic curve based on the chosen security strength.

The elliptic curve that will be used for the protocol implementation is $secp256k1$. Its domain parameters can be taken from the SECG published document [5]. Elliptic curve $secp256k1$ is a Koblitz curve that is most notable for its usage in Bitcoin transaction signing. The elliptic curve domain parameters typically consist of examples of two different types of parameters – parameters associated with a Koblitz curve and parameters chosen verifiably at random. Because of its special non-random way Koblitz curves enable especially efficient computation. If the implementation is sufficiently optimized Koblitz curves can perform more than 30% faster than other curves of the same size. In addition to that, $secp256k1$ constants were selected in a predictable way, which significantly lowers the possibility of any sort of backdoors inserted into the curve by its creator. Elliptic curve $secp256k1$ is among the recommended elliptic curves specified by SECG and matches the chosen security strength of 128 bits. It is important to mention that there are other elliptic curves that may be chosen as an alternative. [33, 5]

For the purpose of the simplification, this thesis changes the parties of the protocol to more general ones, i.e., Mobile and Server, leaving out such details as whether the action is in fact performed by, for example, mobile application.

5.1.1 Key exchange

This section describes Two-Party ECDSA key exchange protocol, which is divided into two phases.

EC key generation and exchange. The first phase of the Two-Party ECDSA key exchange is to generate EC keys and calculate the composite EC public key using ECDH (Figure 9):

1. Mobile generates EC key pair (d_m, Q_m) using curve $secp256k1$;

2. Mobile derives 256-bit encryption key from PIN code by hashing the PIN with SHA-256 hash function. Mobile performs encryption of the 256-bit EC private key d_m by performing XOR operation with the derived 256-bit encryption key:

   $$enc_{PIN}(d_m) = d_m \oplus \text{SHA-256(PIN)}$$

   The chosen encryption scheme is discussed later in Section 5.3;

3. Mobile stores the encryption of its EC private key $enc_{PIN}(d_m)$;

4. Mobile deletes the plaintext value of its EC private key d_m;

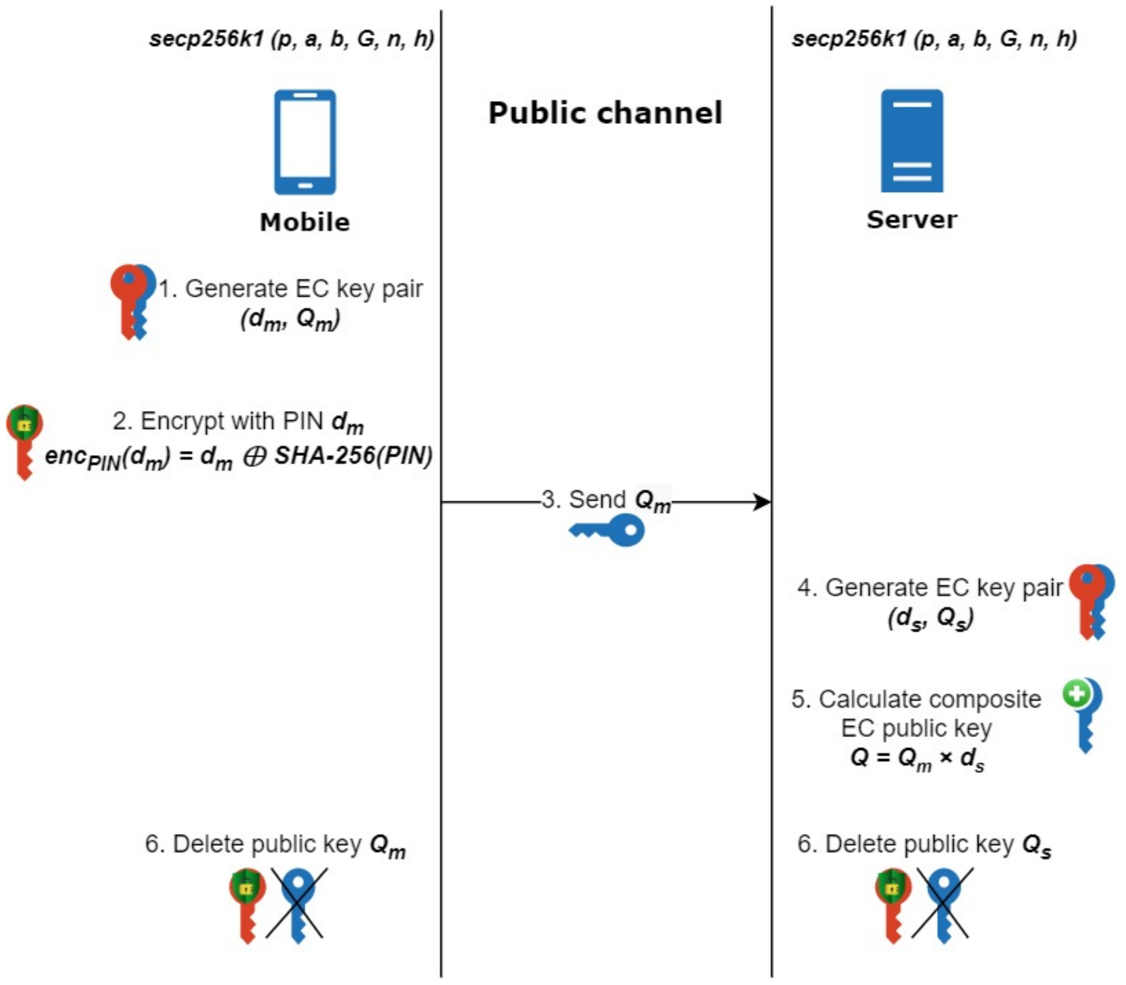
5. Mobile sends its EC public key Q_m to Server;

6. Server generates EC key pair (d_s, Q_s) using curve $secp256k1$;

7. Server calculates the composite EC public key:

   $$Q = Q_m \times d_s$$

8. Mobile and Server delete their corresponding public keys Q_m and Q_s. Mobile has to delete it to prevent the Mobile's EC public key to be used as a reference point in PIN brute-force attacks. Server can delete it to save space.



secp256k1 (p, a, b, G, n, h)
Public channel
secp256k1 (p, a, b, G, n, h)
Mobile
Server
1. Generate EC key pair (d_m, Q_m)
2. Encrypt with PIN d_m
$enc_{PIN}(d_m) = d_m \oplus SHA\text{-}256(PIN)$
3. Send Q_m
4. Generate EC key pair (d_s, Q_s)
5. Calculate composite EC public key $Q = Q_m \times d_s$
6. Delete public key Q_m
6. Delete public key Q_s

Figure 9. EC key generation and exchange protocol for Smart-ID.

Paillier key generation and exchange. The second phase of the Two-Party ECDSA key exchange is to generate the Paillier key pair on Server, encrypt Server's EC private key and send the ciphertext to Mobile (Figure 10):

1. Server generates Paillier public key (n_p, g_p) and private key (λ, μ) with key length of 3072 bits. The key length is calculated according to Lindell's protocol. It is important to remember here that the chosen security strength is 128 bits;
2. Server encrypts its EC private key d_s with Paillier public key (n_p, g_p). Paillier encryption is described in Section 2.2.2;
3. Server sends Paillier public key (n_p, g_p) and its encrypted EC private key $enc_{PAILLIER}(d_s)$ to Mobile;
4. Server deletes its EC private key d_s.

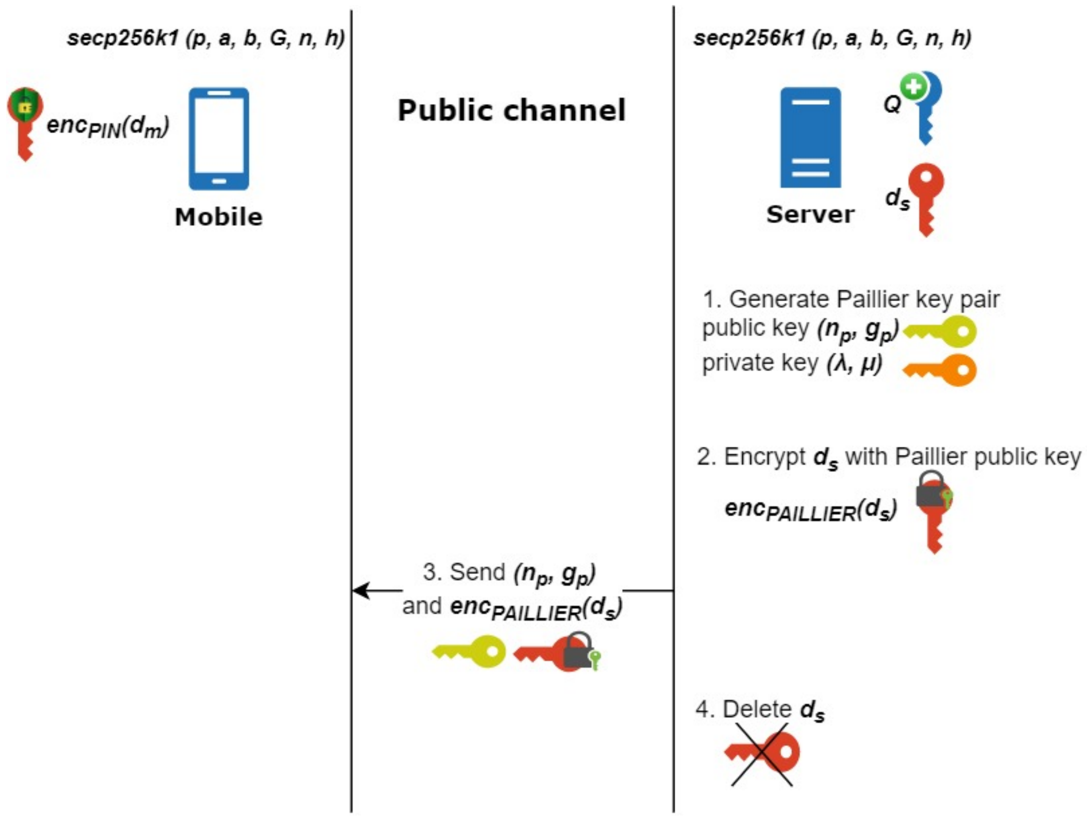
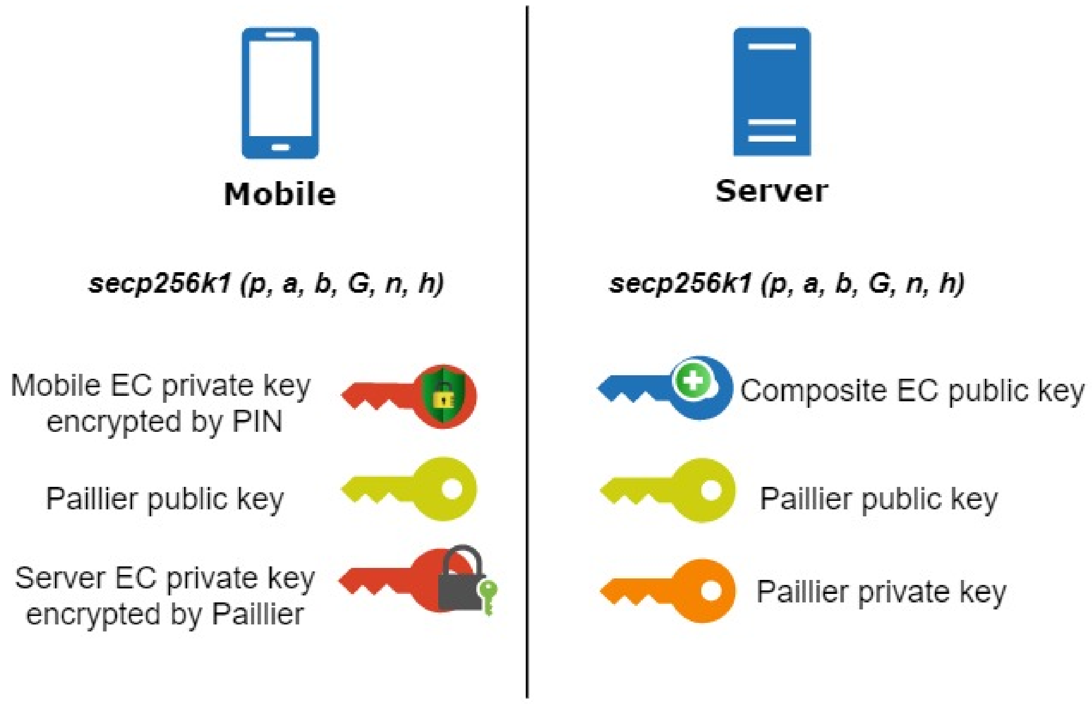

secp256k1 (p, a, b, G, n, h)
$enc_{PIN}(d_m)$
Mobile
Public channel
secp256k1 (p, a, b, G, n, h)
Q
Server
d_s
1. Generate Paillier key pair
public key (n_p, g_p)
private key (λ, μ)
2. Encrypt d_s with Paillier public key
$enc_{PAILLIER}(d_s)$
3. Send (n_p, g_p)
and $enc_{PAILLIER}(d_s)$
4. Delete d_s

Figure 10. Paillier key generation and exchange protocol for Smart-ID.

Final state after key exchange. Once key exchange is successfully finished, Mobile and Server end up with stored values that are shown in Figure 11.



Mobile
secp256k1 (p, a, b, G, n, h)
Mobile EC private key encrypted by PIN
Paillier public key
Server EC private key encrypted by Paillier
Server
secp256k1 (p, a, b, G, n, h)
Composite EC public key
Paillier public key
Paillier private key

Figure 11. Final state after Two-Party ECDSA key exchange protocol for Smart-ID.

Mobile has the following values stored:

- (p, a, b, G, n, h) – $secp256k1$ EC domain parameters;
- $enc_{PIN}(d_m)$ – Ciphertext of Mobile's EC private key encrypted by PIN;
- $enc_{PAILLIER}(d_s)$ – Ciphertext of Server's EC private key encrypted using Paillier. Used in signature generation calculations;
- (n_p, g_p) – Paillier public key.

Server's values:

- (p, a, b, G, n, h) – $secp256k1$ EC domain parameters;
- Q – composite EC public key that is able to verify signature only when it is signed by both EC private keys;
- (n_p, g_p) and (λ, μ) – Paillier key pair.

5.1.2 Signing

In an actual Smart-ID protocol scenario the ECDSA r value can be calculated and validated simultaneously with the computation of s value, in case the commitment protocol and zero-knowledge proof for discrete logarithm are left out. This improves the overall performance of the signing protocol, because interaction between Mobile and Server is reduced to minimum. It is enough for Server to notify Mobile, then for Mobile to make one request to Server and handle the response.

The changes between the original Yehuda Lindell's protocol and the proposed implementation are described in more detail in Section 5.2. The protocol description omits some of the details that are provided in the Yehuda Lindell's signing protocol description above (Section 4.2). Below is a detailed description of the steps involved in the signing protocol (Figure 12):

1. Server calculates the z value using the function SHA-256 on the content intended to be signed. In ECDSA algorithm it is possible to sign a value up to bit length of n, therefore, it makes sense to use the hash function that would be exactly as long in bits as the curve's n parameter. For the elliptic curve $secp256k1$, that is chosen for this protocol, it makes sense to use the hash function that produces 256-bit hash value, such as SHA-256. If longer hash values are signed then it is necessary to truncate them to 256 bits reducing the security strength to the same as provided by SHA-256;

2. Server generates a cryptographically secure random secret number k_s from range $0 < k_s < n$ and temporarily stores it for the next steps;

3. Server calculates R_s point based on previously generated secret number k_s:

$$R_s = G \times k_s$$

4. Server sends a notification message with the value z and the point R_s to Mobile. This triggers the process of an encrypted partial signature generation on Mobile;

5. Mobile generates a cryptographically secure random secret number k_m from range $0 < k_m < n$;

6. Mobile calculates the r value of ECDSA signature by using previously received R_s and generated k_m:

$$R = R_s \times k_m$$

$$r = R.x \mod n$$

7. Mobile validates that r is not equal to zero, or otherwise, the process has to return back to step 5. The probability for this to occur is negligible;

8. Mobile calculates its R_m point on the chosen curve based on the previously generated secret number k_m:

$$R_m = G \times k_m$$

9. Mobile prompts the user to enter PIN code. Once entered, Mobile's EC private key d_m is decrypted by applying XOR operation with the SHA-256 hash derived from the entered PIN code:

$$d_m = enc_{PIN}(d_m) \oplus \text{SHA-256(PIN)}$$

10. Mobile generates a cryptographically secure random number p from range $0 < p < n^2$;

11. Mobile calculates ciphertext of partial signature $enc_{PAILLIER}(s')$ using the same method as described in Yehuda Lindell's protocol (Section 4.2);

12. Mobile deletes k_m from the device;

13. Mobile sends a message to Server with the ciphertext of partial signature $enc_{PAILLIER}(s')$, R_m and signature's r value;

14. Server calculates the r value of the ECDSA signature:

$$R = R_m \times k_s$$

$$r = R.x \mod n$$

15. Server verifies that the resulting r is the same as received from Mobile. If not, Server replies with an error;

16. Server decrypts $enc_{PAILLIER}(s')$. Paillier decryption is described in Section 2.2.3. Server obtains partial signature s';

17. Server applies the remaining operations in order to compute s value of the ECDSA signature:

$$s = s' \cdot k_s^{-1} \mod n$$

$$s = min(s, n - s)$$

18. Server deletes its secret number k_s;

19. Server verifies the resulting ECDSA signature (r, s). If the verification is successful, Server replies with success response, otherwise, Server replies with an error and reduces the amount of remaining PIN attempts.

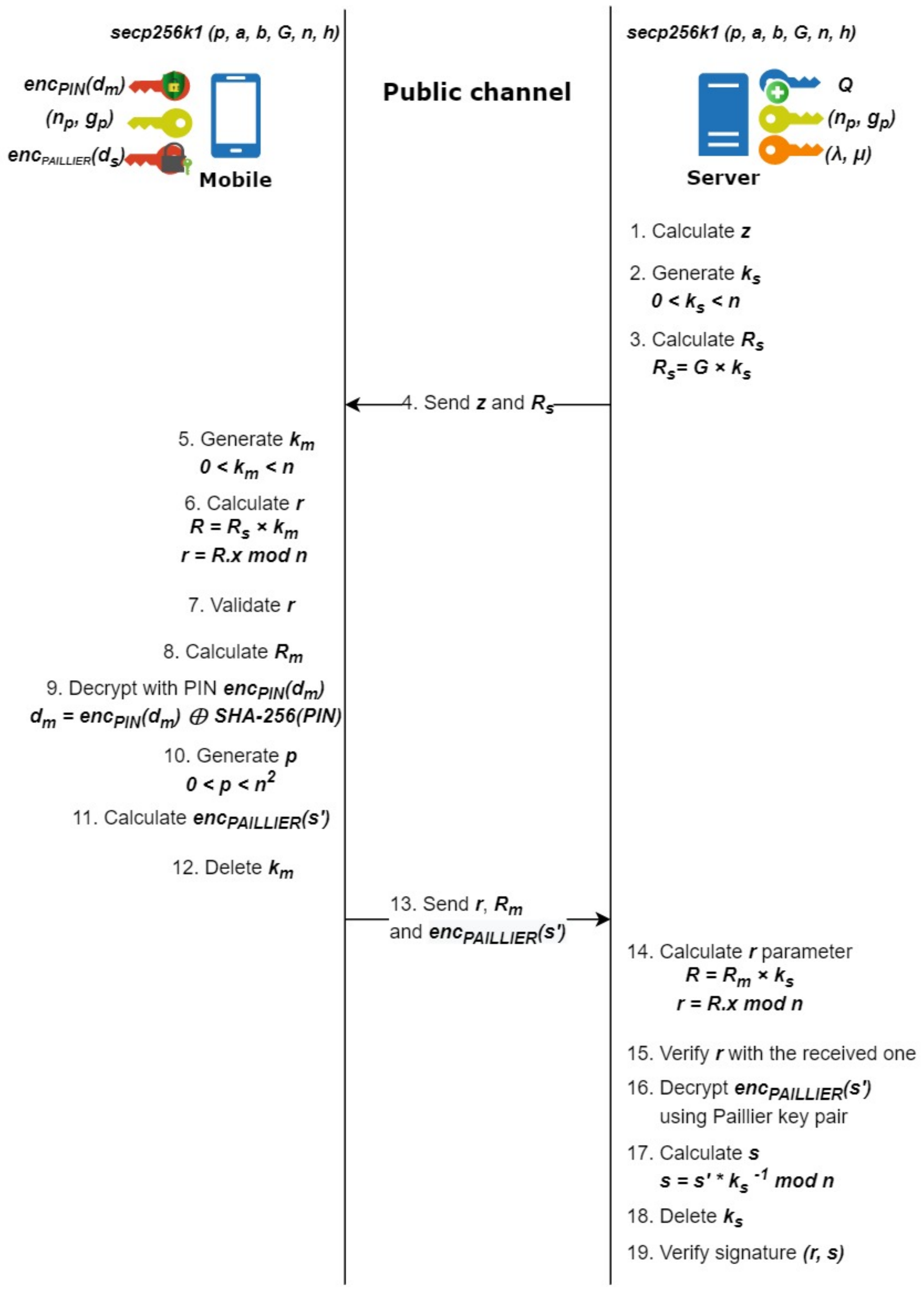

secp256k1 (p, a, b, G, n, h)
$enc_{PIN}(d_m)$
(n_p, g_p)
$enc_{PAILLIER}(d_s)$
Mobile
Public channel
secp256k1 (p, a, b, G, n, h)
Q
(n_p, g_p)
(λ, μ)
Server
1. Calculate z
2. Generate k_s $0 < k_s < n$
3. Calculate R_s $R_s = G \times k_s$
4. Send z and R_s
5. Generate k_m $0 < k_m < n$
6. Calculate r $R = R_s \times k_m$ $r = R.x \bmod n$
7. Validate r
8. Calculate R_m
9. Decrypt with PIN $enc_{PIN}(d_m)$ $d_m = enc_{PIN}(d_m) \oplus SHA\text{-}256(PIN)$
10. Generate p $0 < p < n^2$
11. Calculate $enc_{PAILLIER}(s')$
12. Delete k_m
13. Send r, R_m and $enc_{PAILLIER}(s')$
14. Calculate r parameter $R = R_m \times k_s$ $r = R.x \bmod n$
15. Verify r with the received one
16. Decrypt $enc_{PAILLIER}(s')$ using Paillier key pair
17. Calculate s $s = s' * k_s^{-1} \bmod n$
18. Delete k_s
19. Verify signature (r, s)

Figure 12. Two-Party ECDSA signing protocol for Smart-ID.

5.2 Comparison to the original Two-Party ECDSA protocol

The proposed implementation of Two-Party ECDSA protocol for Smart-ID is different from Yehuda Lindell's protocol in several aspects. This section describes the rationale behind these decisions. However, the assumptions made below may have to be reconsidered before using this protocol in production as these suggestions have not received sufficient security analysis from the cryptanalysis community.

Firstly, in the original protocol Yehuda Lindell proposes to use a zero-knowledge proof for a discrete logarithm, which proves that both parties have the private keys corresponding to the exchanged EC public keys. On top of this zero-knowledge proof required from both sides, there is also a commitment protocol in place that makes sure that the party, who initiates the key exchange, at first simply commits to its corresponding public key as well as the previously described zero-knowledge proof without sending the actual values until the corresponding values are received from the other party. This ensures that a party cannot choose its value based on the value received from the other party.

The author of this work did not find any evidence that such measures are associated with any particular attack vectors. ECDH protocol has been used since the beginning of the 2000s, yet no such measures were proposed to be implemented for its improvement in the software development world. It was confirmed by Yehuda Lindell in the email exchange that the main purpose of these countermeasures is to be able to cryptographically prove the security of the protocol. Although the author of the original protocol considers these measures important, he also does not have any particular attack scenarios in mind that could be relevant when such measures are ignored. Therefore, these two security measures may be considered as additional precautions and implemented only in order to raise the security level even further. However, the author of this work chose to exclude these zero-knowledge proofs for a discrete logarithm and the commitment protocol in both key exchange and signing phases, since they significantly raise the amount of communication steps between Mobile and Server.

Secondly, the original protocol includes proving in zero-knowledge that a Paillier public key was generated correctly as well as that the encrypted value is the private key corresponding to Server's EC public key. This is done via zero-knowledge proof which bridges Paillier encryption and EC groups. This zero-knowledge proof has been designed and proved by Yehuda Lindell (page 32 in [19]).

Similarly as before, it is not clear if removal of these proofs enable any practical attack by Server. Server could try to construct values of Paillier public key and Paillier encryption of Server's EC private key such that when Mobile applies partial signature calculation, the value of partial signature would expose Mobile's EC private key or the value of Mobile's secret number k_m. To be useful, such an attack would also have to produce a valid ECDSA signature as a result. While the author of this work was not able to come up with a specific attack vector, to ensure that the removal of zero-knowledge proofs do not make the proposed protocol vulnerable, the security proof should be made. This task, however, was not in the scope of this work, therefore the author of this work introduced a reasonable assumption that at least in the key generation phase Server is honest and follows the protocol. This Paillier encryption related zero-knowledge proof would require several steps of communication during key exchange according to the original protocol. The proposed implementation ignored this security measure due to the lack of known attack vectors and significant complication of communication during key exchange phase.

Another difference from the original protocol is that in the key exchange phase Mobile uses an ordinary $secp256k1$ EC key pair, where private key is from range $0 < d < n$, while Yehuda Lindell's protocol suggests that one party (represented by Mobile) may choose a private key from range $0 < d < \frac{n}{3}$. As stated in the Yehuda Lindell's paper, this is chosen for technical reasons that become apparent in the specially designed Paillier related zero-knowledge proof. However, no security as well as significant performance implications are associated with it. Also, in implementation it is easier to do the EC key generation via cryptographic libraries, which obviously would not support such a custom case. For those reasons, ordinary $secp256k1$ EC key pair is used for Mobile in the proposed implementation.

Furthermore, in order to reduce the amount of communication steps between Mobile and Server, which affect performance in the actual implementation, the author of this work decided to combine ECDH phase of agreeing on the signature's r value together with computation of signature s value. This change should not have any security implications, because computation of the signature's nonce still happens independently. It is worth pointing out that it will only be possible if the above-mentioned commitment protocol is ignored.

Finally, the proposed protocol is modified to enable protection for Mobile's EC private key by encrypting it with PIN code - the feature that is provided by the current RSA-based Smart-ID protocol (Section 5.3).

5.3 Mobile's private key protection against brute-force attacks

One of the most important problems that the Smart-ID protocol addresses is protection of the private key on the mobile device without the hardware security storage. Nowadays, some mobile devices have a built-in hardware security storage, for instance, Android devices have Strongbox and iOS devices have Secure Enclave. However, that is far from being always the case [9, 21, 14]. This thesis intends to propose a solution, based on Smart-ID example, how such a problem could be addressed using the proposed Two-Party ECDSA protocol. However, it is necessary to mention that there could be various alternative solutions that may provide the same security.

It is clear that the Mobile's EC private key should not be stored in a clear form on the device. At the same time, the Mobile's EC private key should not be disclosed to

Server at any point, because that would enable Server to produce signatures without the participation of Mobile. Therefore, the Mobile's EC private key can be stored in an encrypted form on the device, but there are certain requirements that this kind of encryption method should meet.

The encryption password must be short enough to be conveniently entered, but at the same time its shortness should not be exploitable in brute-force attacks. Smart-ID uses PIN code to serve as that password. A PIN code is a sequence of up to 5 digits. This PIN code is used to encrypt and decrypt the Mobile's EC private key. The fact that PIN code is easily brute-forced compels Mobile to not possess any information that could be used as a reference point by the attacker to learn whether the decryption of Mobile's EC private key was successful. This is the main reason why in the proposed protocol Mobile's EC public key should be deleted as soon as possible. Otherwise, it would serve as a hint on which private key is actually correct by verifying whether the decrypted result is a private key that corresponds to the Mobile's EC public key. Therefore, when a user enters the PIN code, Mobile has no information that could be used by the attacker to check whether the decryption is valid. Server is able to understand the correctness of the PIN code by verifying the signature with the composite public key. Server regulates the amount of unsuccessful PIN attempts according to the chosen policy. To provide protection against brute-force attacks, the decrypted result using each wrong PIN code candidate has to be indistinguishable from the actual private key that is obtained by performing decryption with the correct PIN code.

In Smart-ID, the user has 9 attempts all together with the time-delay locks to guess the PIN code. So in case attacker obtains a mobile device with Smart-ID application, the attacker will have the below calculated probability of guessing the PIN code before getting blocked.
AUTHENTICATION 4-digit PIN:

$$\frac{9}{10^{len(PIN)}} = \frac{9}{10^4} = \frac{9}{10000} = 0.0009 = 0.09\%$$

SIGNING 5-digit PIN:

$$\frac{9}{10^{len(PIN)}} = \frac{9}{10^5} = \frac{9}{100000} = 0.00009 = 0.009\%$$

If there is absolutely no information hinting which PIN code or private key may have more probability of success, then, as can be seen from the calculations, the chance to guess the correct PIN code is very small.

Smart-ID uses symmetric block cipher algorithm AES with CBC mode of operation and no padding. It is crucial not to use any padding mechanisms for encryption of the Mobile's EC private key, because padding has a structure that is distinguishable from the random value obtained when decrypting using a key derived from an incorrect PIN code.

Since EC private key is a random 256-bit number, it is indistinguishable from a random value obtained as a result of decryption using a wrong PIN code. However, it is important to remember that not every 256-bit number may be a valid EC private key, because EC private key is a number that is smaller than the elliptic curve's order n. In the case of elliptic curve $secp256k1$ there would be $2^{256} - n$ out of 2^{256} decryption variations that could be identified as invalid. However, in the context of Smart-ID's 4-digit and 5-digit PINs, this probability is so negligible (less than $\frac{1}{10^{37}}$) that it will not improve probability of guessing the PIN code.

In the current RSA Smart-ID protocol the Mobile's private key exponent had to be shared with Server as otherwise the Mobile's RSA modulus could have been used as a reference point in brute-force attack. Since in the proposed Two-Party ECDSA scheme Mobile does not have to use Mobile's EC public key in the calculation of partial signature, Mobile can simply delete the public key thus eliminating the reference point.

As an alternative to encryption using block cipher such as AES, the proposed scheme uses simple XOR operation encrypting the 256-bit EC private key with 256-bit key stream obtained by hashing the PIN code using SHA-256:

$$enc_{PIN}(d_m) = d_m \oplus \text{SHA-256(PIN)}$$

This optimization is possible because EC private key is relatively short. As an alternative, the EC private key could have been split into two 16-byte blocks and encrypted using AES similarly as done in the current Smart-ID protocol.

XOR operation would be exceptionally lightweight in terms of performance for 256 bit numbers as well as very simple to implement. If the attacker obtains $enc_{PIN}(d_m)$, he would have 2^{256} possible valid private key values d_m to guess. However, since hash is derived from PIN code, the attacker will know that only $10^{len(PIN)}$ out of them can be potentially valid PIN codes, and therefore, private keys. These $10^{len(PIN)}$ potentially valid PIN codes retain the property of having an equal probability because each will produce a random value indistinguishable from others. Therefore, this technique fulfills the requirement. Because of that, such a method is very straightforward and convenient to use for the purpose of encrypting the Mobile's EC private key.

5.4 Possible protocol modifications

This section discusses possible protocol modifications that can be considered when implementing the Two-Party ECDSA protocol.

5.4.1 Reversing the roles of the parties

In the proposed protocol the roles of Alice and Bob of the original Two-Party ECDSA protocol by Lindell have been assigned to Mobile and Server, respectively. The roles could be reversed (Mobile is Bob and Server is Alice). However, in that case, Paillier private key has to be protected on Mobile using some other method than the one described in Section 5.3. The amount of communication steps in both scenarios is the same.

5.4.2 Storage of Server's EC private key

Server can delete its EC private key d_s or store in encrypted or plain form. This depends on the recovery strategy that will be used. If the system is expected to be built in such a manner that Mobile can restore its EC private key (for instance, with mnemonic seed), then the Server should be eligible to provide again its encrypted EC private key $enc_{PAILLIER}(d_s)$. Since Smart-ID currently does not provide such a use case, then the key can be safely deleted.

5.4.3 Multiplication of private keys during key exchange

Another interesting optimization is that Mobile can precompute $d_s \cdot d_m$ in Paillier-encrypted form after receiving Paillier ciphertext of Server's EC private key $enc_{PAILLIER}(d_s)$ and store it instead of two values - $enc_{PAILLIER}(d_s)$ and $enc_{PIN}(d_m)$. In order to do that Mobile should do the PIN code encryption only when it receives $enc_{PAILLIER}(d_s)$. Then, Mobile can perform a homomorphic multiplication of $enc_{PAILLIER}(d_s)$ by a scalar d_m due to Paillier property described in Section 2.2.4. Once $enc_{PAILLIER}(d_s \cdot d_m)$ is calculated, it should be encrypted with the PIN code. It is important to remember that the encryption must match all the requirements described in Section 5.3.

Instead of storing $enc_{PAILLIER}(d_s)$ and $enc_{PIN}(d_m)$ Mobile will store only a single value $enc_{PIN}(enc_{PAILLIER}(d_s \cdot d_m))$. As well as having one less value to store, this also brings a minor performance improvement since during the signing process Mobile can now skip one ciphertext multiplication by a scalar.

However, since Paillier ciphertext is not just a random number as EC private key is, it can potentially enable an attacker to distinguish valid decryption results, and, therefore, eliminate some of the PIN codes.

5.4.4 Reuse of Paillier key pair

In an actual implementation of the protocol, Paillier key pair may be generated only once and then reused across key exchanges with different Mobile devices, therefore improving Server's key exchange performance. This change should not introduce any new security risks (at least if Paillier encryption related zero-knowledge proof is excluded).

A compromised Mobile may only be able to collect encrypted but not the plaintext value of partial signature s' since it gets multiplied by the Server's secret number k_s. Other than that, the only Paillier-related operation is the encryption of Server's EC private key, which should not expose any details of the Server's Paillier private key even if done repeatedly.

5.4.5 Deterministic ECDSA

As mentioned in Section 2.5.2, if the same secret number k is used twice for signing different messages, then it is possible to calculate the private key d. For the proposed implementation of Two-Party ECDSA this risk also applies. In standard ECDSA, if the same k is selected to sign different messages using the same private key, then given two signatures (r, s_1) and (r, s_2) as well as their corresponding messages m_1 and m_2, it is possible to calculate k and then the private key d using the calculations below:

1. Calculate z values of messages by using hash function and taking $len(n)$ leftmost bits;

2. Then k value can be calculated using the following calculation:
$$k = \frac{z_2 - z_1}{s_2 - s_1}$$

3. Finally, retrieve private key:
$$d = \frac{s \cdot k - z_1}{r}$$
It does not matter which z value (z_1 or z_2) to use here.

In Two-Party ECDSA if one party reuses the same k for different signatures and the other party is interested in abusing it, then it could lead to the similar consequences. If one party has a faulty Random Number Generator (RNG) and the other party is interested in abusing it, then the latter one can reuse the same k value until there are two signatures for different messages with the same r value. Once that is achieved, a composite EC private key can be retrieved. In theory, it may also be possible that both parties have a faulty RNG but such occurrence is unlikely.

One solution how such reliance on randomness can be partly mitigated is by using the deterministic algorithm of ECDSA described in RFC6979 [27]. In this algorithm k value is generated based on message and private key, therefore, it is always unique for a particular message. In Two-Party ECDSA this method can also be applied, but the role of p value, generated by Mobile in the signing process, becomes more important. If p value is exposed or can be predicted and Mobile is compromised in such a way that the attacker has access to its storage to retrieve Mobile's PIN-encrypted EC private key ($enc_{PIN}(d_m)$) and the outbound communication to retrieve Paillier-encrypted partial signature ($enc_{PAILLIER}(s')$), then the attacker can brute-force the Mobile's EC private key d_m. The use of RFC6979 prevents the possibility that the nonce k is reused, but Mobile still has to have access to RNG to generate p.

6 Smart-ID RSA and ECDSA comparison

This chapter intends to provide an overview for comparison of the current Smart-ID RSA-based scheme with the proposed ECDSA-based scheme, taking into account storage, performance, security and implementation considerations.

6.1 Implementation

The proposed ECDSA-based Smart-ID scheme is implemented in Java programming language and is provided in the public BitBucket repository [13]. For standard ECDSA operations the third-party cryptographic Java library BouncyCastle v1.65 [24] is used. The class `TwoPartyEcdsa.java` can be run to demonstrate generation of Two-Party ECDSA signature described in this paper. The classes `Mobile.java` and `Server.java` represent the corresponding parties mentioned in this work. Benchmarking related implementations of both RSA and ECDSA are done in separate classes and are located in a package `ee.ut.msc.ecdsa2p.benchmark`.

As can be seen later in Section 6.2, the difference between average and maximum performance of different operations is quite significant which means that the calculations using secret values are not done in constant time. This could be used to perform timing attacks potentially leaking the secret values used in the calculations. Due to that fact, it is important that in an actual production implementation the operations performed with secret values are done in constant time in order to not make the implementation vulnerable to timing side channel attacks.

Both schemes are very straightforward to implement and can use cryptographic libraries for key generation purposes as well as signature verification. Although the Paillier cryptosystem implementation is not provided in many popular programming languages, it is relatively easy to implement it with key generation optimization proposed by Jonathan Katz and Yehuda Lindell [16]. The Paillier cryptosystem in the provided proof of concept implementation is implemented using math operations of Java's `java.math.BigInteger` class.

The provided implementation may be used as a reference implementation to create an actual production solution. However, it is important to mention that this implementation is missing several important aspects of the actual Smart-ID protocol such as protection of secret values, clone detection, protection against side-channel attacks and so on.

6.2 Performance

Table 2 describes how different phases of Smart-ID ECDSA and RSA schemes implemented in Java programming language perform on the laptop computer with the Intel i7 7500u processor using the key lengths described in this work. Every operation was performed 1000 times with new keys being generated each time in order to provide better variation.

Table 2. Comparison of Smart-ID RSA and ECDSA performance.

	RSA			**ECDSA**		
	Minimum	**Maximum**	**Average**	**Minimum**	**Maximum**	**Average**
Key exchange	565 ms	22349 ms	4484 ms	215 ms	5182 ms	963 ms
Signing	89 ms	320 ms	102 ms	179 ms	477 ms	202 ms
Verification	0 ms	2 ms	0 ms	0 ms	14 ms	1 ms

As can be seen in Table 2, the RSA scheme significantly loses in performance in key generation and exchange, however, is tangibly faster in signing. Approximately half of the time necessary for ECDSA-based signing comes from Paillier decryption operation on Server. The author of this work was not able to optimize this implementation further, however, that may as well be possible. Regardless of that, the signing time should be more than enough for an actual production solution. The verification time is so insignificant for both schemes that it is not worth mentioning. The signing in ECDSA-based scheme would result in the same amount of communication rounds as in the Two-Party RSA protocol, and therefore should not have any additional complexity. Benchmarking implementations are provided in the source code repository [13] together with the proof of concept implementation.

6.3 Storage

The Server related values for security strength of 128 bits would require the amount of storage described in Table 3.

Table 3. Comparison of Smart-ID RSA and ECDSA storage.

Value	**RSA**	**ECDSA**
signature	384 bytes	64 bytes
composite public key	768 bytes	64 bytes
α	384 bytes	-
β	384 bytes	-
n_m	384 bytes	-
n_s	384 bytes	-
d_{m_2}	384 bytes	-
d_s	384 bytes	-

Based on Table 3, in the Two-Party RSA scheme Server would need at least 3072 bytes per each boarded device and 384 bytes per each signature. In the Two-Party ECDSA scheme these values are significantly lower – 64 bytes per each boarded device and 64 bytes per each signature. Also, in the Two-Party RSA scheme 768 bytes per boarded device (d_{m_2} and d_s) need to be stored securely on the Server side. In the Two-Party ECDSA scheme Paillier private key also requires 768 bytes of secure storage, but since it is reused its storage requirements remain the same regardless of the amount of devices, therefore, it does not affect storage in the context of scaling. Due to the above, Paillier key pair values are not included in Table 3.

Although for today's storage capabilities these numbers may seem insignificant, on a big scale they can actually provide a tangible advantage in costs. The client related values are not listed since the amount of required storage is insignificant for modern mobile devices.

6.4 Security

The security of the current RSA-based Smart-ID scheme relies on the difficulty of the RSA problem, while the proposed ECDSA scheme relies on both ECDLP and DCRA. Also, the proposed ECDSA-based scheme requires good randomness in both key generation and signing phases, while the RSA scheme only in the key generation phase. Although the requirement of randomness in signing process can be reduced using RFC6979 [27], the reliance on two different public key cryptography assumptions is not in favor of the ECDSA-based scheme.

One of the main motivations for creating the ECDSA scheme in Smart-ID context is the fact that ECDSA-based scheme does not require the secret sharing of the Mobile's private key. This is because ECDSA signature only requires private key d to produce a digital signature, while in RSA also the public key modulus is needed. Therefore, in the ECDSA-based scheme the Mobile's public key can be deleted right after the composite public key during key exchange has been calculated. It is arguable whether this has an actual advantage in any particular attack scenario. This is due to the fact that if an attacker is able to eavesdrop communication during key generation, then Mobile's EC public key can be obtained the same way as Mobile's private key in RSA scheme to later successfully do the brute-force attack against Mobile's encrypted private key.

7 Conclusion and future work

This thesis proposed Two-Party ECDSA protocol based on Yehuda Lindell's paper [19] to be used for the Smart-ID-like scheme. The key exchange and signing protocols have been described in detail for the security strength of 128 bits. The proposed Two-Party ECDSA protocol has been analysed in several aspects providing different solutions that should be considered before an actual production implementation. Finally, the proof of concept implementation for the protocol was provided.

The proposed Two-Party ECDSA protocol provides significant storage optimization in comparison to the current Smart-ID's RSA-based one (6 times less per signature) and does not require scalable secure storage on the server side. The experimentation results proved that the proposed Two-Party ECDSA protocol is multiple times faster in key exchange, however, slightly loses in signing performance within an acceptable amount. The verification in both cases happens to be a matter of at max a few milliseconds. The author believes it is possible to optimize the implementation further to achieve even better performance results.

One of the biggest motivations to research Two-Party ECDSA was that it does not require the secret sharing of a mobile device's private key during key exchange as in Smart-ID's Two-Party RSA solution. Although Two-Party ECDSA enables a more elegant solution by using ECDH, this feature does not eliminate any particular attack vectors that would be relevant for the Smart-ID scheme.

Paillier decryption has the most impact on the performance of the described Two-Party ECDSA protocol. Choi et al. in [11] proposed an approach to modify Paillier cryptosystem in such a way to be able to perform decryption more efficiently and without the need of μ parameter. Let m be the message in plaintext and c its Paillier ciphertext, then the decryption operation will be $m = L(c^{\lambda} \mod n^2)$. This is achieved by a specially chosen public key that satisfies the equation $g^{\lambda} = 1 + n \mod n^2$. However, there are certain attack vectors discovered by Kouichi Sakurai and Tsuyoshi Takagi [29] that need to be taken into consideration before implementing this optimization.

The Paillier cryptosystem is not the only cryptosystem with homomorphic properties that could fit the Two-Party ECDSA scheme. In fact, any additively homomorphic cryptosystem may also be suitable, because the ciphertext addition property also implies the ciphertext multiplication by a scalar, since multiplication by a scalar can be done as ciphertext addition times of the scalar value. Among additively homomorphic systems there are such cryptosystems as Boneh–Goh–Nissim [3], Okamoto–Uchiyama [25] and Benaloh cryptosystem [2]. It would be interesting to see how these cryptosystems would compare in performance with the Two-Party ECDSA Paillier cryptosystem covered in this work.

After Yehuda Lindell's paper [19] was published, there has been published several works that also addressed the problem of Two-Party ECDSA. Castagnos et al. proposed in their paper [7] a very similar protocol to Lindell's that uses encryption techniques from

Hash Proof Systems of Ronald Cramer and Victor Schoup [8] instead of Paillier cryptosystem. However, the performance of that scheme for 128-bit security is marginally slower than that of Lindell's scheme. Another interesting protocol was proposed by Doerner et al. in paper [10]. This protocol does not rely on any additional assumptions apart from ECDLP and provides exceptionally fast signing time. However, the understanding and implementation of the protocol is very complex due to the complicated calculations and proofs used in the protocol.

The Two-Party ECDSA case has had a significant development during the last few years, mainly motivated by the need to protect Bitcoin private keys from theft. The scheme proposed by Yehuda Lindell was a big step towards deploying such a scheme in practice. At this development rate, it would not be a surprise if far more efficient and secure Two-Party ECDSA schemes would appear very soon.

References

[1] X9 ANSI. 62: public key cryptography for the financial services industry: the elliptic curve digital signature algorithm (ECDSA). *Am. Nat'l Standards Inst*, 1999.

[2] Josh Benaloh. Dense probabilistic encryption. In *Proceedings of the workshop on selected areas of cryptography*, pages 120–128, 1994.

[3] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-DNF formulas on ciphertexts. In *Theory of cryptography conference*, pages 325–341. Springer, 2005.

[4] Daniel RL Brown. SEC 1: Elliptic curve cryptography. *Certicom Research*, 2009.

[5] Daniel RL Brown. Sec 2: Recommended elliptic curve domain parameters. *Standars for Efficient Cryptography*, 2010.

[6] Ahto Buldas, Aivo Kalu, Peeter Laud, and Mart Oruaas. Server-supported RSA signatures for mobile devices. In *European Symposium on Research in Computer Security*, pages 315–333. Springer, 2017.

[7] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Two-party ECDSA from hash proof systems and efficient instantiations. In *Annual International Cryptology Conference*, pages 191–221. Springer, 2019.

[8] Ronald Cramer and Victor Shoup. Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 45–64. Springer, 2002.

[9] Android Developers. Android keystore system. `https://developer.android.com/training/articles/keystore#HardwareSecurityModule`. (2020.08.04).

[10] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 980–997. IEEE, 2018.

[11] Choi Dug-Hwan, Seungbok Choi, and Dongho Won. Improvement of probabilistic public key cryptosystems using discrete logarithm. In *International Conference on Information Security and Cryptology*, pages 72–80. Springer, 2001.

[12] Damien Giry. NIST Report on Cryptographic Key Length and Cryptoperiod (2020), May 2020.

[13] Eduard Iltšuk. Two-Party ECDSA proof of concept. `https://bitbucket.org/eduard_iltsuk/two-party-ecdsa/src/master/`. (2020.08.04).

[14] Apple Inc. Apple Platform Security. `https://support.apple.com/guide/security/secure-enclave-overview-sec59b0b31ff/1/web/1`. (2020.08.04).

[15] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International journal of information security*, 1(1):36–63, 2001.

[16] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2014.

[17] Cameron F Kerry and Charles Romine Director. FIPS PUB 186-4 federal information processing standards publication digital signature standard (DSS). 2013.

[18] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[19] Yehuda Lindell. Fast secure two-party ECDSA signing. In *Annual International Cryptology Conference*, pages 613–644. Springer, 2017.

[20] Manfred Lochter and Johannes Merkle. Elliptic curve cryptography (ECC) brainpool standard curves and curve generation. Technical report, RFC 5639, March, 2010.

[21] Daniel Micay. Auditor overview. `https://attestation.app/about`. (2020.08.04).

[22] Victor S Miller. Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques*, pages 417–426. Springer, 1985.

[23] Dana Neustadter and Tom St Denis. Elliptic Curves over Prime and Binary Fields in Cryptography. 2008.

[24] Legion of the Bouncy Castle Inc. BouncyCastle. `https://www.bouncycastle.org/`. (2020.08.04).

[25] Tatsuaki Okamoto and Shigenori Uchiyama. A new public-key cryptosystem as secure as factoring. In *International conference on the theory and applications of cryptographic techniques*, pages 308–318. Springer, 1998.

[26] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*, pages 223–238. Springer, 1999.

[27] Thomas Pornin. Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA). *Internet Engineering Task Force RFC*, 6979:1–79, 2013.

[28] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[29] Kouichi Sakurai and Tsuyoshi Takagi. On the security of a modified Paillier public-key primitive. In *Australasian Conference on Information Security and Privacy*, pages 436–448. Springer, 2002.

[30] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Conference on the Theory and Application of Cryptology*, pages 239–252. Springer, 1989.

[31] SK ID Solutions. Smart-ID Technical Overview. `https://www.smart-id.com/wordpress/wp-content/uploads/2017/01/smart-id-technical-overview-v0.6.html`. (2020.08.04).

[32] Scott Vanstone. Responses to NIST's proposal. *Communications of the ACM*, 35(7):50–52, 1992.

[33] Bitcoin Wiki. Secp256k1. `https://en.bitcoin.it/wiki/Secp256k1`. (2020.08.04).

Appendix

I. Glossary

AES Advanced Encryption Standard. 20, 45, 46

CBC Cipher Block Chaining. 20, 45

DCRA Decisional Composite Residuosity Assumption. 9, 51

DH Diffie-Hellman. 14

DSA Digital Signature Algorithm. 17

EC Elliptic Curve. 14, 17, 24, 25, 27–29, 33, 35, 36, 38, 39, 41, 43–48

ECC Elliptic Curve Cryptography. 6, 12, 14

ECDH Elliptic Curve Diffie-Hellman. 14, 24, 29, 35, 43, 44, 52

ECDLP Elliptic Curve Discrete Logarithm Problem. 12, 51, 53

ECDSA Elliptic Curve Digital Signature Algorithm. 6–8, 17, 18, 24, 29, 30, 32, 33, 35, 38, 40, 41, 43, 44, 46, 48–53

HSM Hardware Security Module. 6, 19

NIST National Institute of Standards and Technology. 13, 17

RNG Random Number Generator. 48

RSA Rivest–Shamir–Adleman. 6–9, 14, 19, 20, 35, 44, 46, 49–52

SECG Standards for Efficient Cryptography Group. 13, 17, 35

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public