PKOC Specification

Security Classification:	Protected
Version:	1.0
Revision:	Rev1
Control:	Uncontrolled when printed
Date	11/05/2021

Document History

Version	Date	Author	Description
0.0-.99	Oct 8 2021	Mohammad Soleimani	Initial Document
1.0RC1	Oct 22, 2021	Mohammad Soleimani, Alex Lammers	Changed to PSIA Stationary
1.0RC2	Oct 27, 2021	Mohammed Soleimani	Added ECDHE Perfect Forward Secrecy Flow Updated drawings for ECDHE to remove Normal Flow section
1.0	Nov 5, 2021		Released Specification

Disclaimer

THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING

ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Without limitation, PSIA disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and PSIA disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

You are hereby granted a license to copy and distribute (but not to modify) the information set forth in this document; and to make and sell, including commercially, products using the specification. Any rights that PSIA has not expressly granted are hereby reserved.

PSIA encourages you to explore membership in the organization to obtain the full set of benefits to be received from use of its specifications.

Table of Contents

Introduction 5

Glossary 5

Recommended App Registration Process: 6

Registration Sequence Diagram: 6

PKOC Requirements: 7

Reader 7

Mobile Device 7

Cryptographic Functions 7

PKOC BLE Exchange: 7

Initial Advertisement and GATT Service setup 7

Message Types 8

PKOC Transmission Flows 9

Common Handshake for all Flows 9

Normal Flow 10

SourceGUID Flow 10

ECDHE Fast Flow 12

ECDHE with Perfect Forward Secrecy Flow 14

High Level Interaction Diagrams 17

Normal Flow 17

SourceGUID Flow 18

ECDHE Fast Flow 19

ECDHE with Perfect Forward Secrecy Flow 20

Sequence Diagram: 21

OSDP Reader and panel PKOC authentication 21

OSDP/WIEGAND Reader AND Reader PKOC authentication 22

Lock PKOC authentication (Fast ECDHE) 23

Lock PKOC authentication (Forward Secrecy ECDHE) 24

Android PKOC Sample Code 25

iOS PKOC Sample Code: 26

Introduction

This document describes an open air-interface protocol for Bluetooth Low Energy (BLE) interactions for a secure credential exchange with a lock or a Reader for access control. This specification is based on Public Key Open Credential (PKOC) concept, which is a system solution for secure, interoperable credentials by utilizing distributed asymmetric encryption key creation. A phone/smart device will generate an asymmetric key pair using EC (Elliptic curve) algorithm and store it in a secure element (secure enclave in iOS and key store in android). The Reader/lock will receive the public key from the phone and authenticate it via challenge-response before granting access.

Glossary

Android – An operating system for mobile devices developed and maintained by Google, Inc..

Application (App) – A software application developed to run on a mobile operating system.

Asymmetric Key Cryptography – A method of communicating where two keys are used, one public which can be transmitted securely in the open, and the other private which should be kept secret. A message can then be either encrypted and/or signed using one key to be decrypted and/or verified by the corresponding key.

Bluetooth Low Energy (BLE) – An subset of the Bluetooth specification designed for low power wireless data exchange without the need to pair devices or maintain a persistent connection.

Challenge-Response – An authentication protocol where a trusted device sends a single use challenge that an untrusted device performs a cryptographic operation with using a secret to generate a response that establishs that the untrusted device possesses and controls the secret.

Credential – A physical or digital token which contains a unique identifier for the individual to whom it was issued. The credential can then be used to assert the identity of the individual to whom it was issued to systems that recognize the issuer.

Digital Signature – Additional data attached to a message that uses encryption and decryption algorithms to verify the message's origin and contents

Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) - is an anonymous key agreement protocol that allows two parties, each having an elliptic curve public-private key pair, to establish a shared secret over an insecure channel.

iOS – An operating system for mobile devices developed and maintained by Apple, Inc.

Lock – A physical device designed to prevent access without without the use of a corresponding key.

Mobile Device – a portable telephone or other network connected computer designed to be carried or worn by the owner.

Nonce – a single use number used as a challenge with Challenge-Response.

Panel – A computing device that is maintained within the secured perimeter of a building that electronically locks and unlocks doors or devices based on messages received at the point of access from a Reader.

Public Key Open Credential (PKOC) – A method of permitting the secure authentication and transmission of the credential of an individual using public keys which can be securely passed in the open which are not issued or created by a centralized authority.

Reader – A small device that is mounted near or embedded within a Lock that permits the transmission of a credential to a Panel using a magnetic stripe, RFID, NFC, or Bluetooth to exchange data with a credential issued to an individual.

Secure Element - a hardware component that provides all cryptographic operations for authenticating the user and is designed to be secure even if the operating system kernel is hacked (e.g. Secure Enclave in iOS and Key Store in Android.)

Universally Unique Identifier (UUID) – a 128-bit number used to identify a unique object on a computer system

Recommended App Registration Process:

In the app, we would have a registration where

- The app will generate the asymmetric key pair with Elliptic Curve (EC) cryptography and save it to the secure element.
- It will send the public key to the cloud at the time of registration. The public key must be transferred in the encrypted format.

Registration Sequence Diagram:

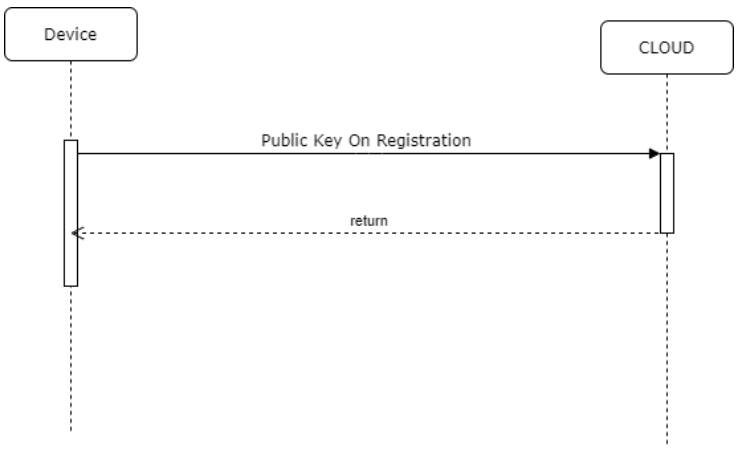


Figure 1 - Registration Sequence

PKOC Requirements:

Reader

To meet the data transmission requirements for PKOC all Readers must support BLE 4.2[1] and utilize the Data Packet Length Extension to permit up to 242 bytes of data to be transmitted. Any reference to Reader in this specification can also refer to a passthrough from a Reader to a Panel where the Panel performs the Reader's actions.

Mobile Device

To interface with a PKOC compliant Reader a mobile device must also support BLE 4.2. Of the most common mobile operating systems in use today this would requires IOS 9.2.1 or later, and Android 6.0.1 or later[2]. Developers are strongly encouraged to support Android 9.0 and later to utilize the operating system level secure enclave.

Cryptographic Functions

PKOC utilizes asymmetric cryptographic functions to securely transmit public keys and other data to Readers. To generate digital signatures on the most common mobile operating systems EcdsaSignatureMessageX962Sha256 is preferred for IOS and SHA256WithECSDA for Android. The signature must follow DER x9.62 encoding but transmit only the 64-byte "real numbers" from the 72-74 byte DER encoded signature. For the Reader, cryptographic functions must be performed by a hardware secure element.

PKOC BLE Exchange:

Initial Advertisement and GATT Service setup

The BLE beacon advertisement and service UUID for PKOC Readers will be 41fb60a1-d4d0-4ae9-8cbb-b62b5ae81810

There would be a PKOC GATT Service (UUID: 41fb60a1-d4d0-4ae9-8cbb-b62b5ae81810) which will have two characteristics:

1. **PKOC Write** (UUID: fe278a85-89ae-191f-5dde-841202693835) which will supports: write with notification, write without response.
2. **PKOC Read** (UUID: e5b1b3b5-3cca-3f76-cd86-a884cc239692) which will supports: read, notify, indicate.

NOTE: The PKOC GATT service UUID, and Write and Read characteristic UUIDs are static for PKOC Readers.

[1] https://www.bluetooth.com/specifications/specs/core-specification-4-2/
[2] https://devzone.nordicsemi.com/f/nordic-q-a/11678/ble-4-2-and-smart-phones-os-compatibility

Message Types

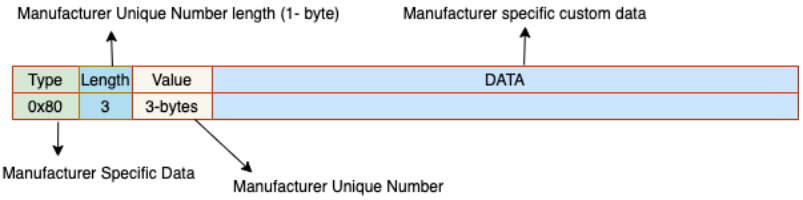
- Messages will utilize a Type/Length/Value (TLV) command structure.
  - Type will be fixed at 1 byte.
  - Length will be fixed at 1 byte permitting a maximum 245 bytes to be usable for the value.
  - Value will contain the data needed to fill but not exceed the length specified.
- Packets can contain more than one message using concatenation.
  - Additional messages will begin at the byte immediately following the last message.
  - There is no limit on the number of messages that can be concatenated subject to the maximum transmission unit size of 247 bytes.
- Support for the following messages is mandatory to be PKOC compliant.

GATT Specific Type

Type	Details	
0x01h	Uncompressed Public Key ECC P-256 (65 Bytes)	PKOC Write
0x02h	Nonce	PKOC Read/Write
0x03h	Digital Signature	PKOC Write
0x04h	Response	Signature Verification
0x05h	Source GUID / Owner GUID	PKOC Read
0x06h	Obfuscated Public Key	PKOC Write
0x07h	Phone ECDHE Establish	PKOC Write
0x08h	Reader ECDHE Accept	PKOC Read
0x09h	Encryption Error	PKOC Read/Write
0x0Ah	(reserved, debug)	PKOC Read/Write
0X0Bh	Request Reader use epheremal keys for ECDHE	PKOC Write
0x40h	Encrypted data follows	PKOC Write
0x80h	Manufacturer Specific Data	PKOC Read/Write

- **Manufacturer Specific Data:** Type 0x80h will be reserved to allow Manufacturer Specific Data (GATT) for the transmission of non-standard data within the standard messaging protocol. This will be in TLV format.
  - Type must be 1 byte with a value of 0x80h for Manufacturer Specific Data.
  - Length will be 1 byte with a value of 0x03h for the Manufacturer Unique Number length

- Value must be 3 bytes for the Manufacturer Unique Number[3] followed by the rest of the data.



- Reader/Panel will perform data verification steps and will send the acknowledge byte to the phone.

PKOC Transmission Flows

PKOC supports four (4) types of data communication between the PKOC Reader and an App on a mobile device. The App will scan for a PKOC Service UUID when a Reader is discovered. The App will be able to communicate with the Reader using the Normal Flow, the SourceGUID Flow, and two ECDHE Flows. The Normal Flow permits communication if the App has neither the public key nor the ObfuscationGUID corresponding to the Reader. The SourceGUID can be used if the App possesses the ObfuscationGUID corresponding to the SourceGUID of the Reader. The ECDHE flows can be used if the App has the public key for the Reader.

NOTE: While all four flows are required in the Reader to support the PKOC specification, the app will select one of the four flows.

Common Handshake for all Flows

The below steps are the handshake procedures before the app selects which flow it will authenticate with.

1. App will scan for PKOC Service UUID, if the Reader is discovered.
2. App will make connection with the Reader.
3. App will discover the PKOC GATT Service
4. App will discover the PKOC Characteristic UUIDs
5. App will send Nonce request to the Reader
6. Reader will send Nonce (16 bytes) +Source GUID (16 bytes) to the App.

Type	Length	Value	Type	Length	Value
2	1-byte	Nonce	5	1-byte	Source GUID

Based on the Reader id in advertisement data, the App will make a decision whether it can follow the Normal Flow, SourceGUID Flow, or ECDHE Flow.

[3] Manufacturer's Unique Number is obtained from the IEEE https://standards.ieee.org/products-services/regauth/oui/index.html

Note: If the App has an Obfuscation GUID that matches the Source GUID for the Reader the Normal Flow and Source GUID Flow are available. If the App has the Reader's Public Key the Normal Flow and the ECDHE Flows are available. The App is not required to answer if the Source GUID is unknown.

Normal Flow

The Normal Flow transmits the PKOC without any encryption or obfuscation. Because the PKOC is a public cryptographic key there is no reduction in security.

1. Using the Private Key of the app, generate the digital signature for the Nonce that Reader sent.
2. Remove the ASN header from #1
3. Fetch uncompressed public key from secure enclave.
4. App sends public key and digital signature to the Reader.
5. Reader will verify the signature and send back response
    a. 040101 – Success
    b. 040100 – Failed

NOTE:- The last byte [Least significant] suggests the Success or Failure and the reason.

We will use 3 least significant bits of the last byte and below are the details:

1. The least significant bit indicates Success (if true) or Failure (if false)
2. The 2nd least significant bit indicates Access Denied (if true)
3. The 3rd least significant bit indicates Signature not verified (if true)

Bits (Least Significant Byte)	Hex Value	Status
00000001	0x01	Success
00000000	0x00	Failure (Unknown reason)
00000010	0x02	Failure (Signature Verified but Access denied)
00000110	0x06	Failure (Signature not verified and Access denied)

SourceGUID Flow

The SourceGUID Flow transmits the PKOC use prearranged obfuscation. Although the PKOC is a public cryptographic key and there is no reduction in security through clear transmission, the use of the obfuscation key provides increased privacy for the PKOC owner.

1. App will send the obfuscated EC Public key and Digital signature to the Reader
    - Logic to create Obfuscated Public Key
    1. Append Nonce + Obfuscated GUID corresponding to the source GUID that Reader sent.
    2. Generate SHA256 of #1, it will give out 32 bytes of hashed data.
    3. Concatenate hashed data (#2) two times to make it 64 bytes of data.
    4. Prepend #3 with 00 to make it 65 bytes.
    5. Now as both hashed data (#4) and the Public key are 65 bytes, do the XOR of both the strings, this will be the Obfuscated public key.
    - Logic to create Digital Signature
    1. Using the Private key of the app, generate the digital signature for the Nonce that Reader sent.
    2. Remove the ASN header from #1
2. Reader will un-obfuscate the Public key
    - **Logic to un-obfuscate Public Key** (same 4 step that were used while creating obfuscated Public Key)
    1. Append Nonce + Obfuscated GUID corresponding to the source GUID that Reader has.
    2. Generate SHA256 of #1, it will give out 32 bytes of hashed data.
    3. Concatenate hashed data (#2) two times to make it 64 bytes of data.
    4. Prepend #3 with 00 to make it 65 bytes.
    5. Reader will now do the XOR of the Obfuscated Public Key that the app sent and the string generated in #4. It will give the Public key of the app
3. Reader will verify the signature and send back the status to app
    - Reader will verify the signature with the public key that was retrieved from the obfuscated public key.
    c. 040101 – Success
    d. 040100 – Failed

NOTE:- The last byte [Least significant] suggests the Success or Failure and the reason.

We will use 3 least significant bits of the last byte and below are the details:

1. The least significant bit indicates Success (if true) or Failure (if false)
2. The 2nd least significant bit indicates Access Denied (if true)
3. The 3rd least significant bit indicates Signature not verified (if true)

Bits (Least Significant Byte)	Hex Value	Status
00000001	0x01	Success
00000000	0x00	Failure (Unknown reason)

00000010	0x02	Failure (Signature Verified but Access denied)
00000110	0x06	Failure (Signature not verified and Access denied)

ECDHE Fast Flow

The ECDHE Flow transmits the PKOC using Elliptic Curve encryption. This flow provides the maximum amount of privacy but requires prior knowledge of the public key of the Reader.

A Reader is manufactured with an embedded Elliptic Curve key pair key pair. This key pair can be used as an ECDH key for secure key exchange and as an ECDSA key for authentication and signing. The Reader's flash memory is the only place where the private portion of the key pair exists.

To establish a secure communications channel, the Phone and Reader interact as follows:

1. The Phone has recognized the Reader from the Reader's advertisement, and already has the Reader's PK public key
2. The Phone will generate a random 16 to 32 byte "nonce" (default 16 byte) challenge to the Reader's embedded key, and transmit it as TLV 0x02.
    a. (This is distinguished from the "PKOC challenge" 0x02 code by virtue that it's being transmitted from the phone to the Reader)
3. Phone will then generate an "ephemeral key pair" for ECDHE, and transmit a **65 byte** TLV 0x07 to the Reader, containing the ECDH parameter block for ECDHE shared-secret generation.
    a. Note that the TLV 0x07 should follow the TLV 0x02
4. (The phone may calculate the Shared Secret concurrently, while waiting for the Reader's response)
5. The Reader will receive the TLV 0x02 "nonce", and perform the following:
    a. The Reader will perform a SHA256 function over the hash to produce a 32 byte hash
    b. The Reader will "sign" the hash using it's embedded PK as ECDSA key
    c. It will transmit the TLV 0x08, giving the Nonce digital signature (**64 bytes**) to authenticate itself
6. The Reader will receive the TLV 0x07, and perform the following:
    a. Use its embedded PK as an ECDH key to complete the ECDH interaction; the Reader may now calculate the Shared Secret
7. App will receive the TLV 0x08, and verify the digital signature; if the digital signature is
    a. Not Verified, it will throw error
8. Both the Reader and the Phone use SHA256 to turn the exact same Shared Secret value into the exact same AES256 key
9. The Reader and Phone are now prepared to communicate securely, using AES256-CBC with an IV of 0.

With an AES256 key now available, the Phone or App may insert a “begin encrypted” TLV indicator in any messages:

1. T = 0x40 “begin encryption”
2. L = 1, one data byte follows
3. V = sequence number, begins at 1 and counts for each “begin encryption” TLV that is transmitted
    a. AES-CBC requires that encrypted blocks are strictly ordered. The sequence number allows a recipient to know if the message flow was interrupted (BLE is a datagram service after all).
    b. If a sequence error occurs, the BLE connection must be closed
        i. except, the recipient may choose to ignore a duplicate
    c. If the connection becomes closed, a new connection must re-establish ECDHE.
    d. Sequence number increment wraps from 255 to 0
4. The remainder of the received characteristic buffer, following 0x40 0x01 0xnn, requires decryption via the ECDHE AES256-CBC key
5. The buffer requires padding to a multiple of 16 bytes
    a. The encrypted form of the padding is random. In decrypted form, the padding must be 0s
6. In case of Success or Failure (due to signature not verified or Access Denied), below data will be returned from the Reader

    a. 040101 – Success
    b. 040100 – Failed

NOTE:- The last byte [Least significant] suggests the Success or Failure and the reason.

We will use 3 least significant bits of the last byte and below are the details:

1. The least significant bit indicates Success (if true) or Failure (if false)
2. The 2nd least significant bit indicates Access Denied (if true)
3. The 3rd least significant bit indicates Signature not verified (if true)

Bits (Least Significant Byte)	Hex Value	Status
00000001	0x01	Success
00000000	0x00	Failure (Unknown reason)
00000010	0x02	Failure (Signature Verified but Access denied)
00000110	0x06	Failure (Signature not verified and Access denied)

ECDHE with Perfect Forward Secrecy Flow

The ECDHE Flow transmits the PKOC using Elliptic Curve encryption. This flow provides the maximum amount of privacy with Forward Secrecy but requires prior knowledge of the public key of the Reader.

A Reader is manufactured with an embedded Elliptic Curve key pair key pair. This key pair can be used as an ECDH key for secure key exchange and as an ECDSA key for authentication and signing. The Reader's flash memory is the only place where the private portion of the key pair exists.

To establish a secure communications channel, the Phone and Reader interact as follows:

1. The Phone has recognized the Reader from the Reader's advertisement, and already has the Reader's PK public key
2. Phone generated ephemeral ECDH Keys
3. The Phone will generate a random 16 to 32 byte "nonce" (default 16 byte) challenge in TLV 0x02 and send to the Reader along with its ephemeral Public key 65 byte TLV 0x07 and message 0x0Bh with a value of 0x01h to prompt Reader to also generate ephemeral keys, transmitted it in the TLV formats.

Note: This is distinguished from the "PKOC challenge" 0x02 code by virtue that it's being transmitted from the phone to the Reader. The TLV 0x07 should follow the TLV 0x02

4. The Reader will receive the TLV 0x02 "nonce", and perform the following:
    a. The Reader will perform a SHA256 function over the hash to produce a 32 byte hash
    b. The Reader will "sign" the hash using it's embedded PK as ECDSA key
5. The Reader will receive the TLV 0x0B "Request for ephemeral PK", and generates the ephemeral keys, containing the ECDH parameter block for ECDHE shared-secret generation.
6. The Reader will receive the TLV 0x07, and uses its ephemeral Private Key as an ECDH key along with Phone's ephemeral Public Key to complete the ECDH interaction; the Reader may now calculate the Shared Secret.
7. The reader transmits the TLV 0x08 digital signature and TLV 0x07 reader's ephemeral public key.
8. The phone receives TLV 0x07 "Reader's ephemeral public key", and will create a shared secret key with its ephemeral Private Key as an ECDH key along with reader's ephemeral Public Key concurrently
9. The phone receives TLV 0x08 "Digital Signature" and will verify the digital signature, if not verified will throw error.
10. Both the Reader and the Phone use SHA256 to turn the exact same Shared Secret value into the exact same AES256 key
11. The Reader and Phone are now prepared to communicate securely, using AES256-CBC with an IV of 0.

With an AES256 key now available, the Phone or App may insert a “begin encrypted” TLV indicator in any messages:

1. T = 0x40 “begin encryption”
2. L = 1, one data byte follows
3. V = sequence number, begins at 1 and counts for each “begin encryption” TLV that is transmitted
    a. AES-CBC requires that encrypted blocks are strictly ordered. The sequence number allows a recipient to know if the message flow was interrupted (BLE is a datagram service after all).
    b. If a sequence error occurs, the BLE connection must be closed
        i. except, the recipient may choose to ignore a duplicate
    c. If the connection becomes closed, a new connection must re-establish ECDHE.
    d. Sequence number increment wraps from 255 to 0
4. The remainder of the received characteristic buffer, following 0x40 0x01 0xnn, requires decryption via the ECDHE AES256-CBC key
5. The buffer requires padding to a multiple of 16 bytes
    a. The encrypted form of the padding is random. In decrypted form, the padding must be 0s
6. In case of Success or Failure (due to signature not verified or Access Denied), below data will be returned from the Reader

    c. 040101 – Success
    d. 040100 – Failed

NOTE:- The last byte [Least significant] suggests the Success or Failure and the reason.

We will use 3 least significant bits of the last byte and below are the details:

4. The least significant bit indicates Success (if true) or Failure (if false)
5. The 2nd least significant bit indicates Access Denied (if true)
6. The 3rd least significant bit indicates Signature not verified (if true)

Bits (Least Significant Byte)	Hex Value	Status
00000001	0x01	Success
00000000	0x00	Failure (Unknown reason)
00000010	0x02	Failure (Signature Verified but Access denied)
00000110	0x06	Failure (Signature not verified and Access denied)

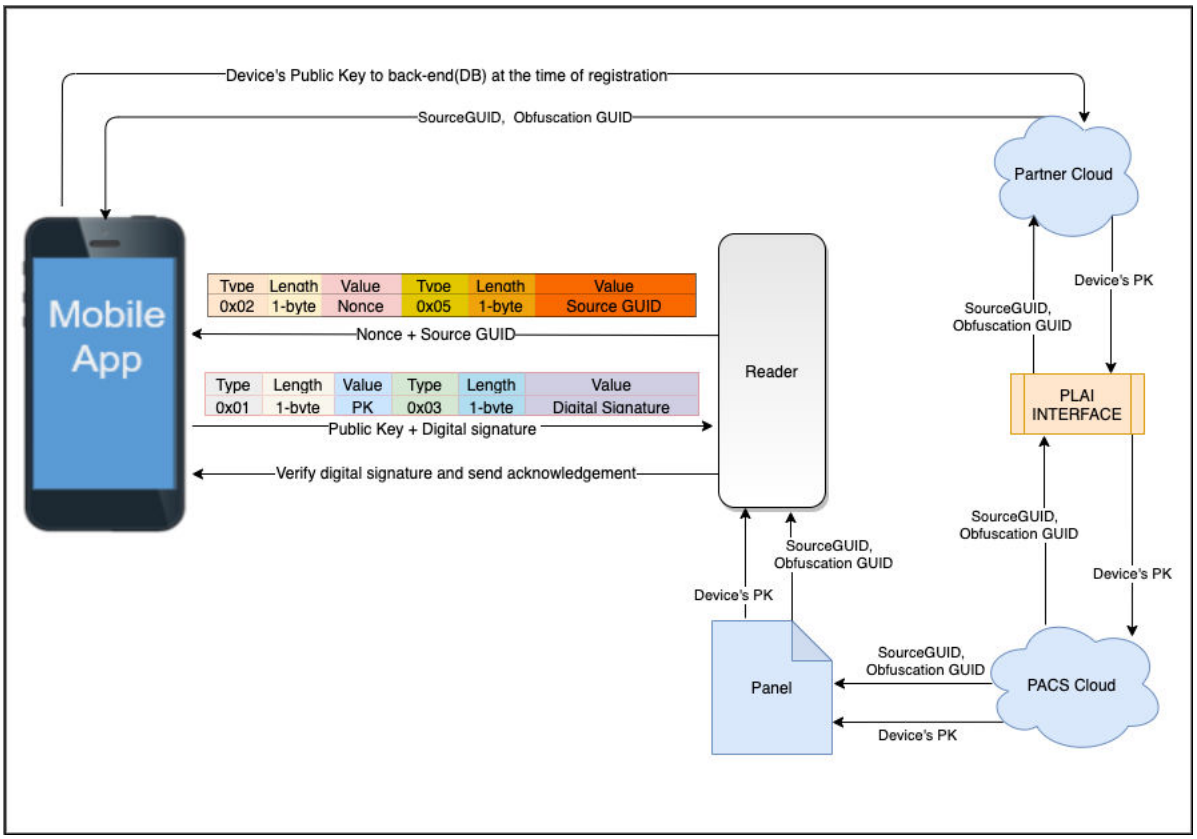
7. In case of failure related to encryption, TLV code 0x09 will be sent from the Reader to the Phone, or from the Phone to the Reader. The TLV will be composed of "0x09 0x01 0xnn", where 0x01 is the payload length, and 0xnn will be a sub-code mention below in the table:

Least Significant Byte	Hex Value	Failure Reason
00000001	0x01	Sequence number error (did not begin at 1, or non-consecutive)
00000010	0x02	Length following 0x40 "tlv_encrypted" is not a multiple of 16
00000011	0x03	ECDHE encryption key not yet established
00000100	0x04	Internal error

Upon receipt of an encryption error, the Phone and Reader *must* close the current connection and re-connect (including new ECDHE negotiation, if necessary).

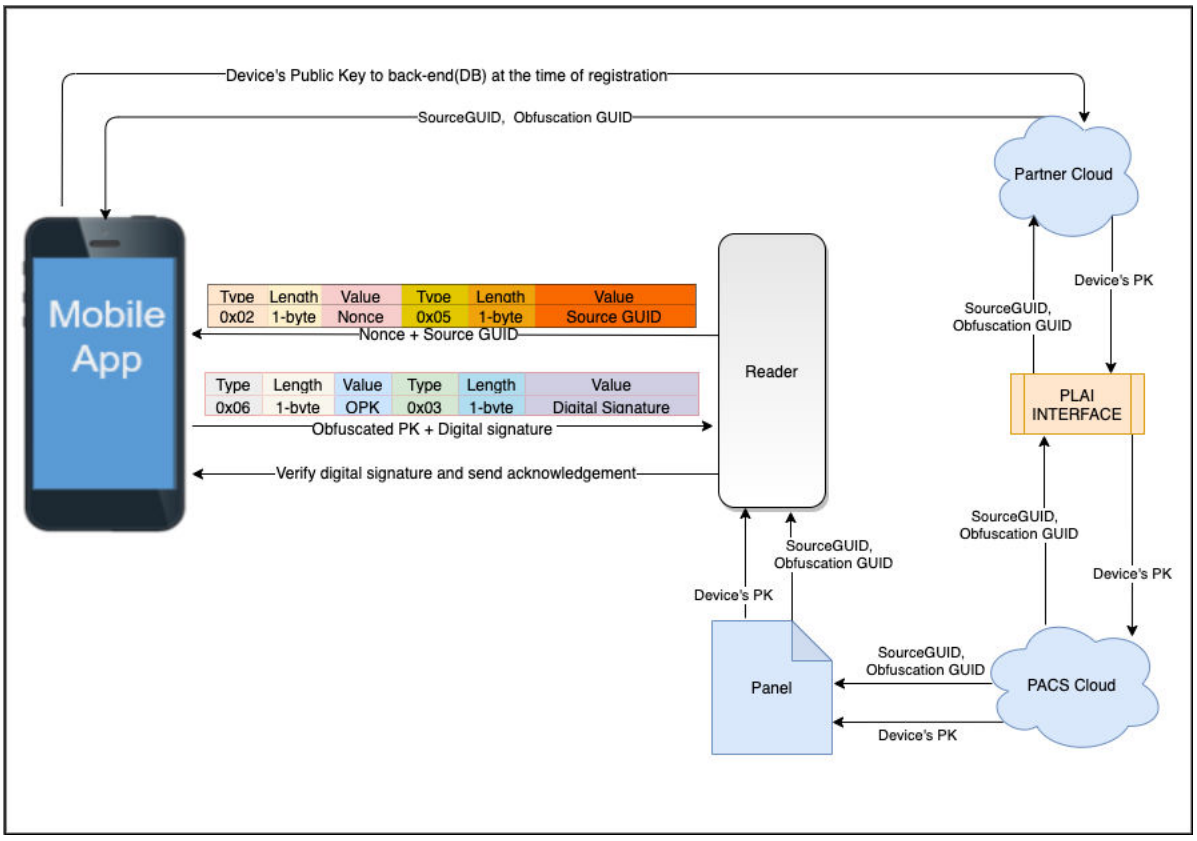
High Level Interaction Diagrams

Normal Flow



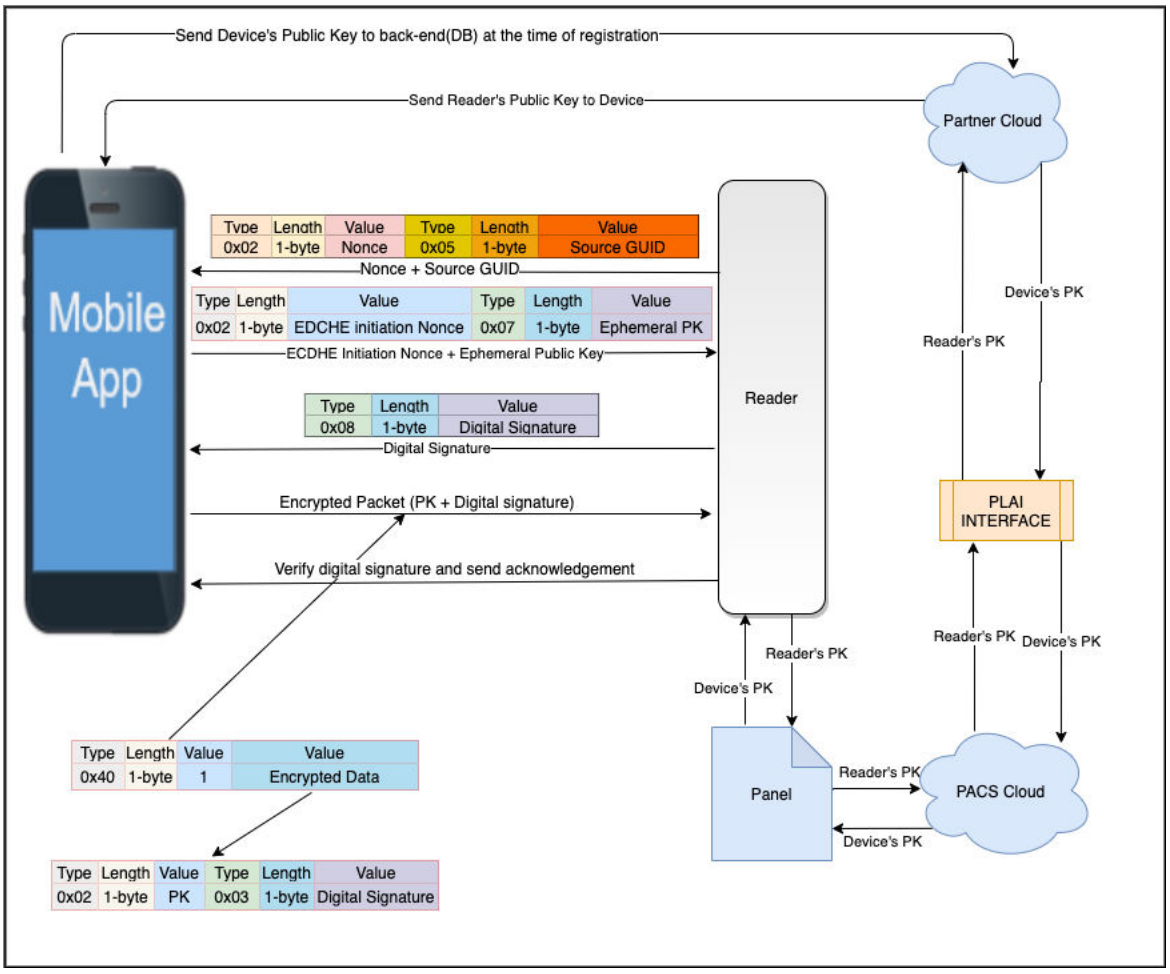
Normal Flow - Mobile Lock Interaction

SourceGUID Flow



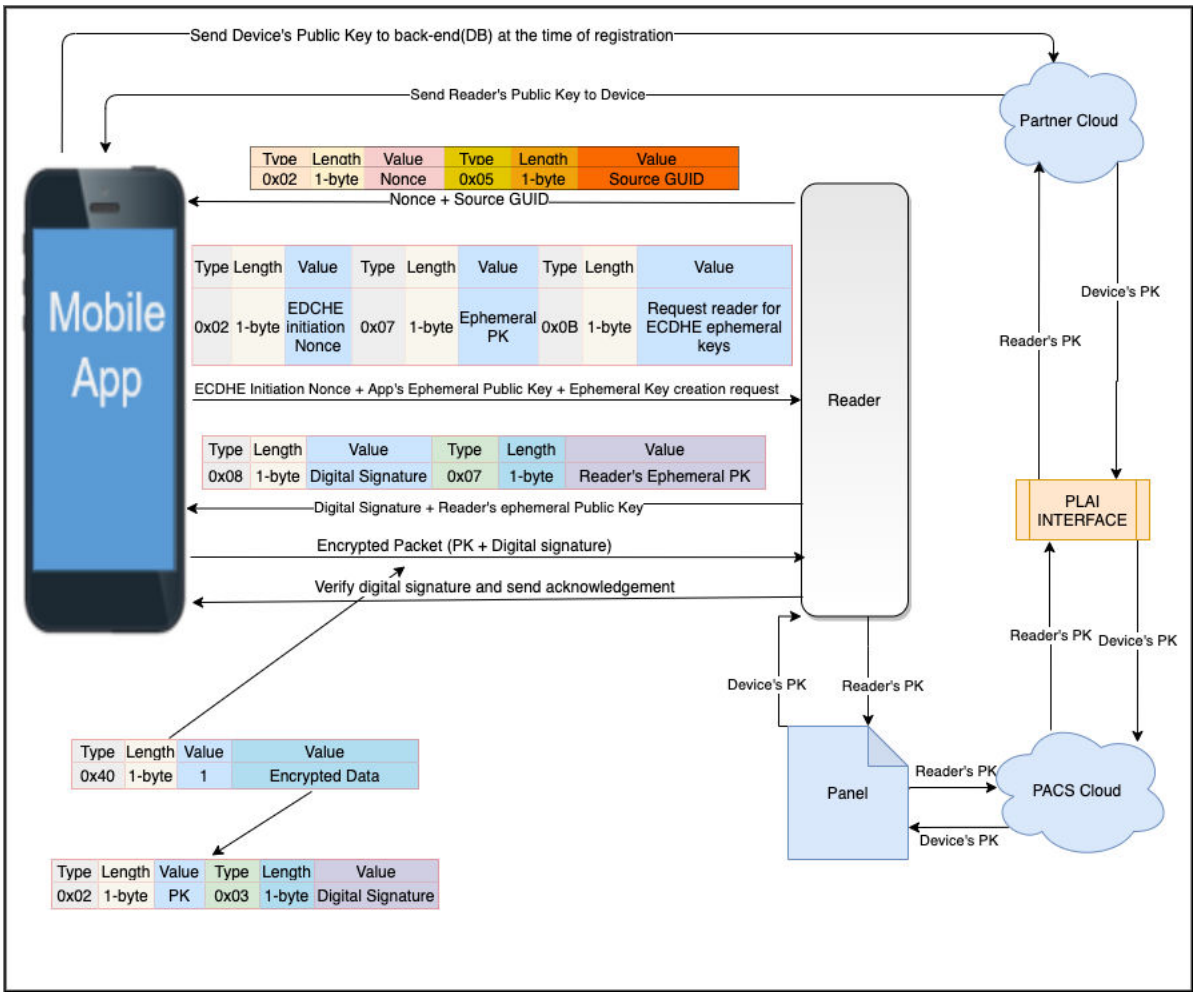
Source GUID- Mobile Lock Interaction

ECDHE Fast Flow



ECDHE Fast Flow - Mobile Lock Interaction

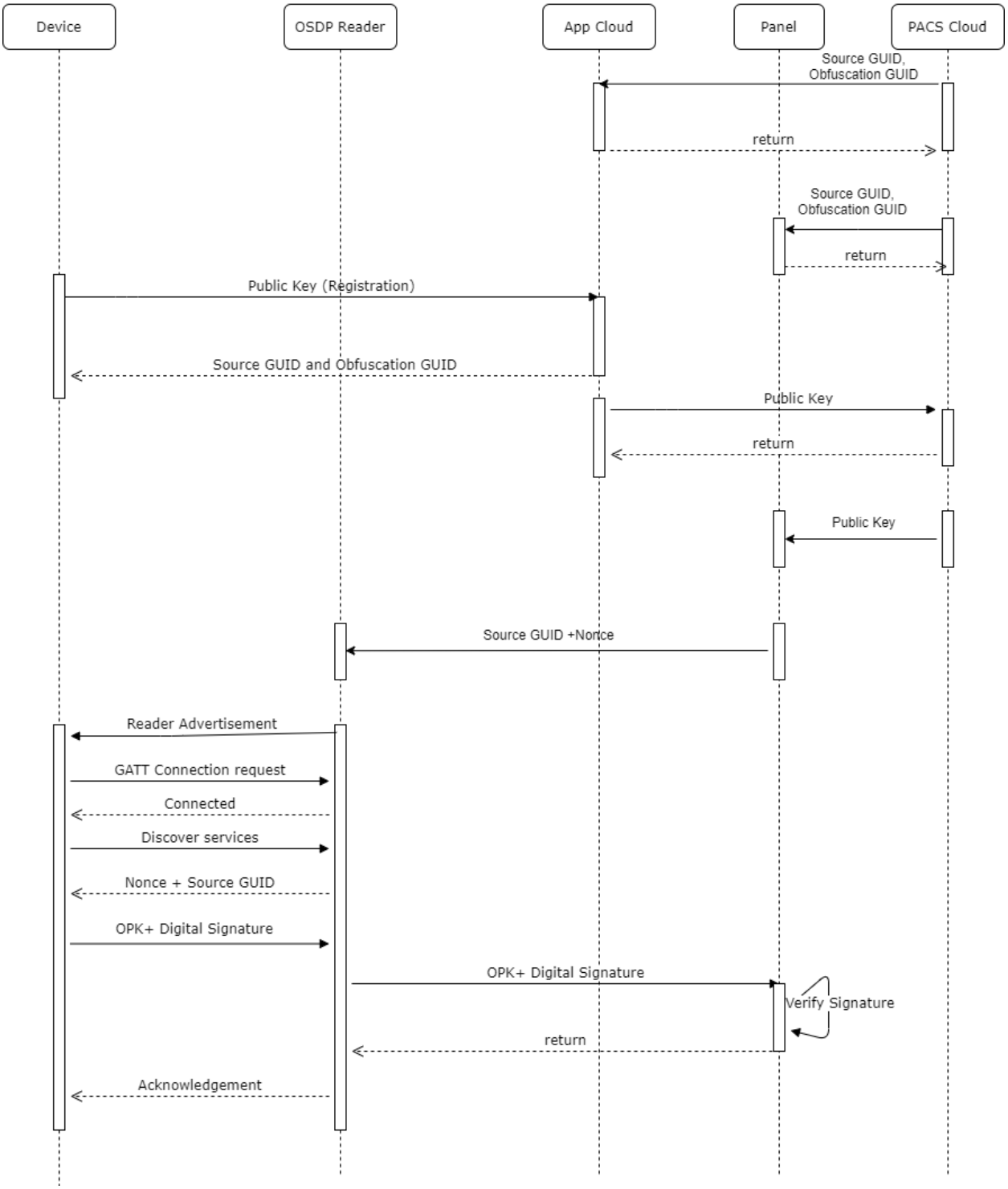
ECDHE with Perfect Forward Secrecy Flow



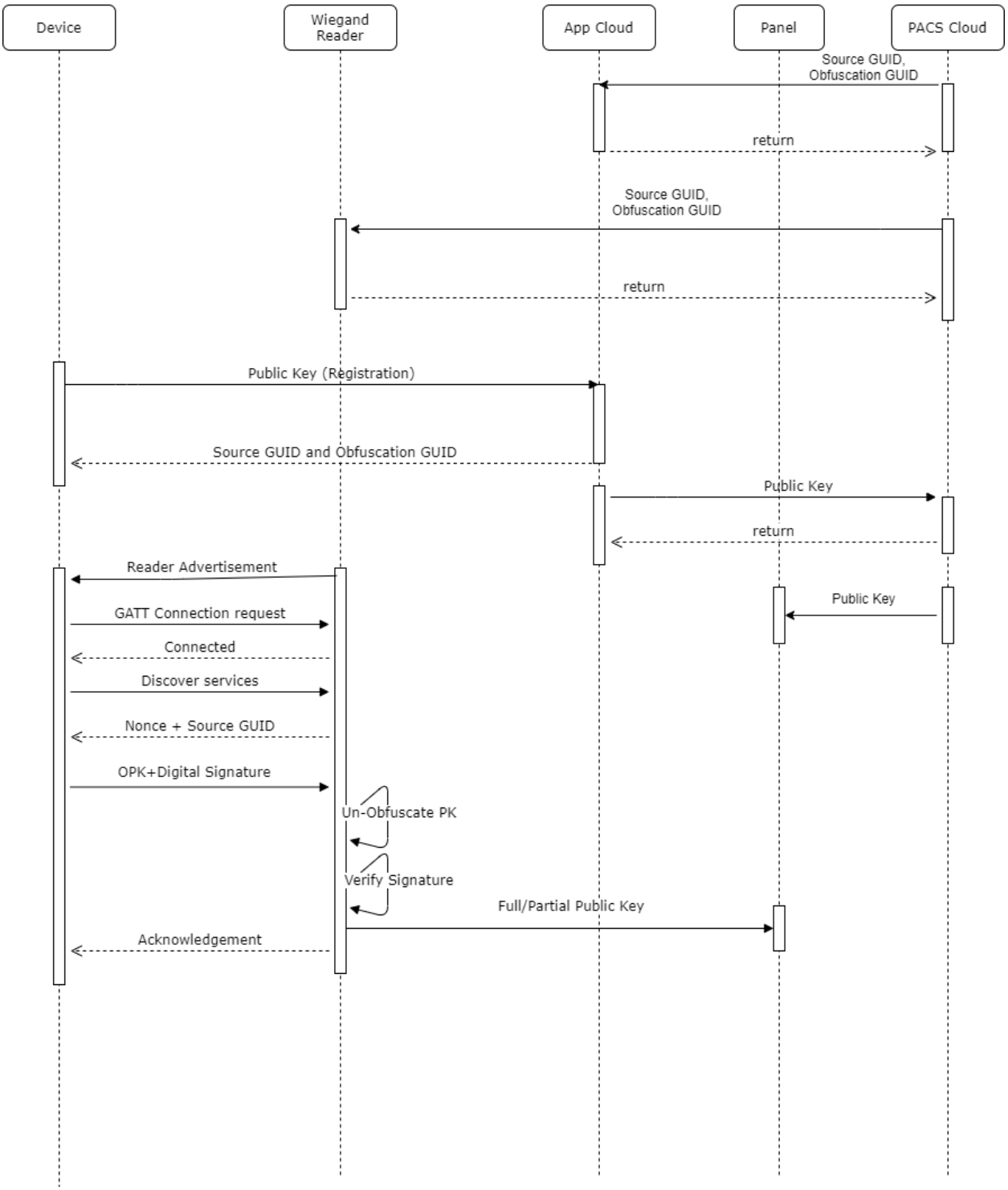
ECDHE with Perfect Forward Secrecy Flow – Mobile Lock Interaction

Sequence Diagram:

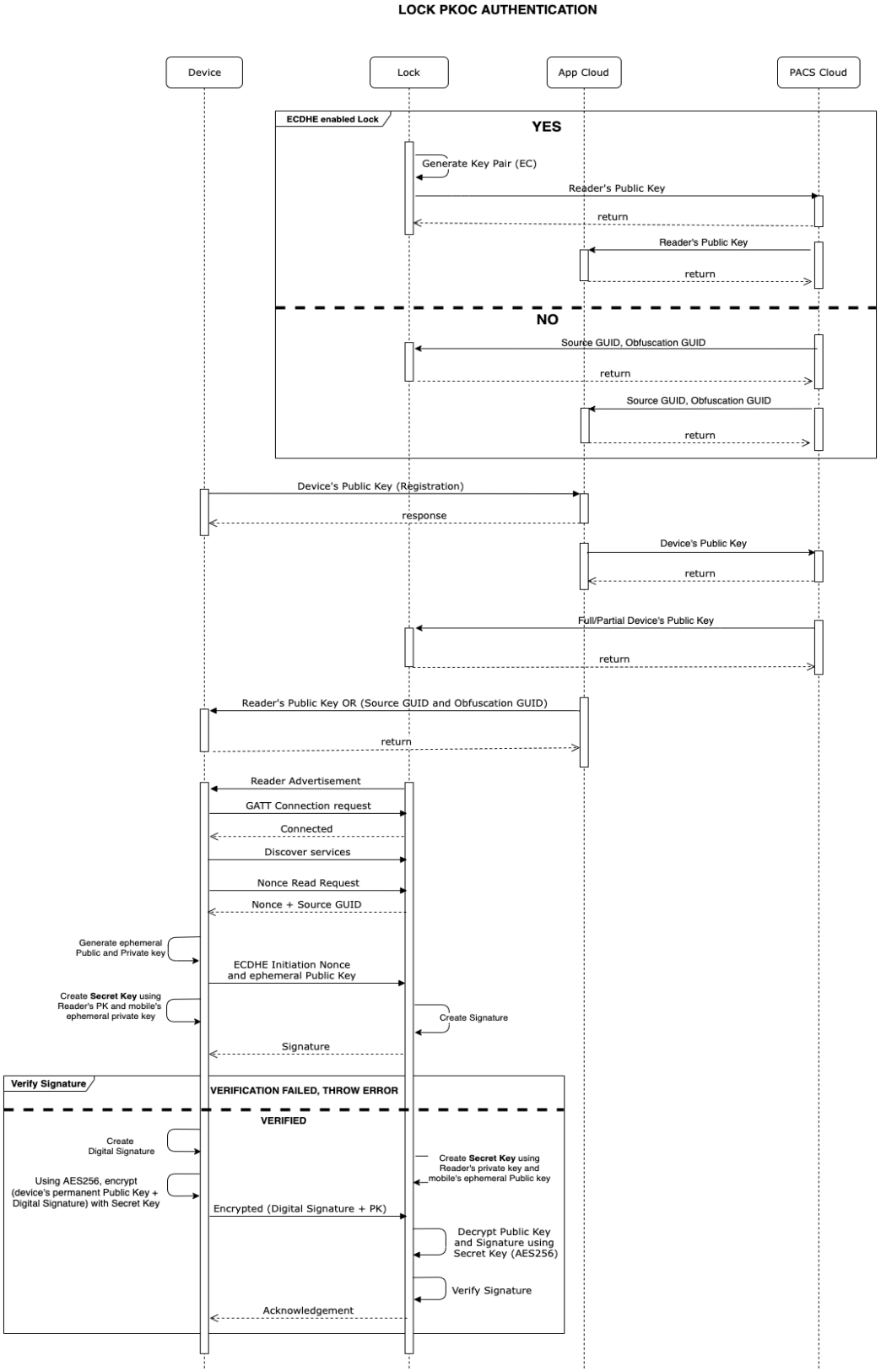
OSDP Reader and panel PKOC authentication



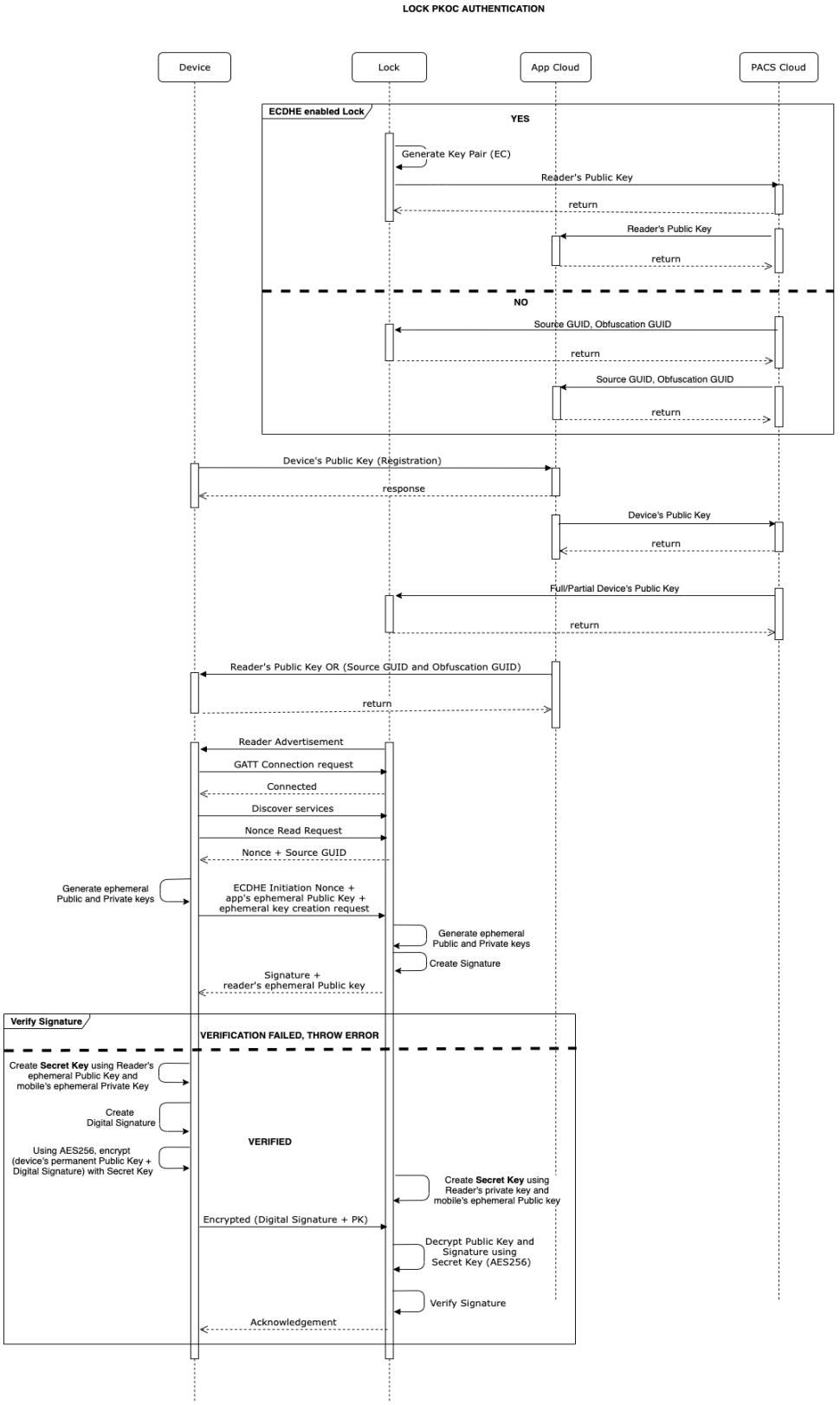
OSDP/WIEGAND Reader AND Reader PKOC authentication



Lock PKOC authentication (Fast ECDHE)



Lock PKOC authentication (Forward Secrecy ECDHE)



Android PKOC Sample Code

1. Create a P-256 ECC public/private key pair

```
public void CreateKeyPair1()
{
            KeyPairGenerator keyGenerator =
KeyPairGenerator.GetInstance(KeyProperties.KeyAlgorithmEc, "AndroidKeyStore");

            ECGenParameterSpec ecParam = new ECGenParameterSpec("secp256r1");
            var builder = new KeyGenParameterSpec.Builder(_keyName, KeyStorePurpose.Sign |
KeyStorePurpose.Verify)
                    .SetUserAuthenticationRequired(false)
                    .SetDigests(KeyProperties.DigestSha256)
                    .SetRandomizedEncryptionRequired(false)
                    .SetKeySize(256)
  .SetAlgorithmParameterSpec(ecParam);
            keyGenerator.Initialize(builder.Build())
            keyGenerator.GenerateKeyPair();
}
```

2. Get the public key

```
public IKey GetPublicKey()
  {

      if (!_androidKeyStore.ContainsAlias(_keyName))
                return null;
       return _androidKeyStore.GetCertificate(_keyName)?.PublicKey;
  }
```

3. Get the signature from PK and the Hash value

```
// Create Signature instance
Signature s = Signature.GetInstance("SHA256withECDSA");
// Get Private Key
var key = ((PrivateKeyEntry)entry).PrivateKey;
// initialization signature with private key.
s.InitSign(privateKey);
// Update the data to be verified.
s.Update(data);
// create signature
var signature = s.Sign();
```

4. Extract 64 Byte of signature

```
public static byte[] RemoveASNHeaderFromSignature(byte[] signature)
{
               if (signature.Length == 64)
               {
                 return signature;
               }
               var calculatedSignature = new byte[64];
               int xLength = signature[3];
               var arrX = new byte[32];
               // copy 32 bytes in arrayX
               Array.Copy(signature, xLength == 32 ? 4 : 5, arrX, 0, 32);
               // copy 32 bytes in aaryaY
               int yLength = signature[3 + xLength + 2];
```



```
                var arrY = new byte[32];
                Array.Copy(signature, yLength == 32 ? (3 + xLength + 2 + 1) :
(3 + xLength + 2 + 1 + 1), arrY, 0, 32);
                // copy arrayX and arrayY in signature bytes.
                Array.Copy(arrX, 0, calculatedSignature, 0, arrX.Length);
                Array.Copy(arrY, 0, calculatedSignature, arrX.Length, arrY.Length);

                return calculatedSignature;
}
```

iOS PKOC Sample Code:

1. Create a P-256 ECC public/private key pair

```
private bool GenerateKECKeyPair() {
using (var access = new SecAccessControl(SecAccessible.AfterFirstUnlockThisDeviceOnly,
SecAccessControlCreateFlags.PrivateKeyUsage) {
                        var secKeyAttributes = new SecPublicPrivateKeyAttrs {
                    ApplicationTag = KeyStoreName,
                    CanDecrypt = true,
                    CanEncrypt = true,
                    CanSign = true,
                    CanVerify = true,
                    IsPermanent = true,

                };
 var secStatusCode = SecKey.GenerateKeyPair(SecKeyType.EC, 256, secKeyAttributes, out SecKey
publicKey, out SecKey privateKey);
 if (secStatusCode == SecStatusCode.Success)
{ return true; }
                else{ return false; }
                }
}
```

2. Get the public key:

```
private  SecKey GetPublicKey()
var Key = SecKeyChain.QueryAsConcreteType(
                new SecRecord(SecKind.Key){
                    KeyType = SecKeyType.EC,
                    ApplicationTag = KeyStoreName,
                },
                out var code);
 code == SecStatusCode.Success ? Key as SecKey : null;
return key.GetPublicKey();
```

3. Get the signature from PK and the Hash value

```
NSData generateDigitalSignature (NSData hashData)
{                   NSData signedData = null;
                    try {
        SecKey privateKey = getPrivateKey ();
        Console.WriteLine (privateKey);
        NSError err;
        if (privateKey == null) {
          GetSecureEnclavePublicKey ();
          privateKey = getPrivateKey ();
        }
```



```
 signedData = privateKey?.CreateSignature (SecKeyAlgorithm.EcdsaSignatureMessageX962Sha256,
hashData, out err);
       Console.WriteLine (signedData);
      } catch (Exception ex) {
      }
      return signedData;
  }
```

4. Extract 64 Byte of signature

```
private byte[] RemoveASNHeaderFromSignature(byte[] signature)
        {
            if (signature.Length == 64)
                return signature;
            var calculatedSignature = new byte[64];
            int xLength = signature[3];
            var arrX = new byte[32];
            Array.Copy(signature, xLength == 32 ? 4 : 5, arrX, 0, 32);
            int yLength = signature[3 + xLength + 2];
            var arrY = new byte[32];
           Array.Copy(signature, yLength == 32 ? (3 + xLength + 2 + 1) : (3 + xLength + 2 + 1
+ 1), arrY, 0, 32);
            Array.Copy(arrX, 0, calculatedSignature, 0, arrX.Length);
            Array.Copy(arrY, 0, calculatedSignature, arrX.Length, arrY.Length);
            return calculatedSignature;
        }
```


END OF DOCUMENT

