SECDSA: Mobile signing and authentication under classical “sole control”[*]

Eric R. Verheul

KeyControls
The Netherlands
eric.verheul@keycontrols.nl

Version 16 March 2024

Abstract The 2014 European eIDAS regulation regulates strong electronic authentication and legally binding electronic signatures. Both require user “sole control”. Historically smartcards are used based on direct interaction between user and relying party. Here sole control is provided by giving users both physical possession and control of the cryptographic key used for signing/authentication through a PIN. Such *classical* sole control is required in the 1999 electronic signature directive by some interpretations. The eIDAS regulation repeals the directive and explicitly relaxes its sole control requirements in a trade-off between security and usability. This allows user interaction to be outsourced to intermediary parties (authentication providers, signing services). This also allows mobile applications as user friendly alternatives for smartcards. However, current mobile platforms are only equipped with limited cryptographic hardware not supporting secure knowledge factors (PINs) controlling keys. The eIDAS relaxation raises concerns on sole control; intermediary parties should not be able to act as man-in-the-middle and impersonate users. In this paper we present a simple cryptographic design for signing and authentication on standard mobile platforms providing classical sole control. We argue that our design can meet the highest eIDAS requirements, effectively introducing a new signature category in a 2016 decision of the European Commission. We also sketch a SECDSA based implementation of the European Digital Identity Wallet recently proposed by the European Commission as part of the eIDAS regulation update.

Keywords: legally binding signing, limited cryptographic hardware, mobile platforms, sole control, non-repudiation, strong authentication

[*] Patent pending.

Contents

1 Introduction

1.1 Background and motivation

The 2014 European eIDAS regulation [18] stipulates requirements for both electronic authentication and signing. This regulation introduces three assurance levels for *authenticators* i.e. the technical means of user authentication towards Service Providers, e.g. a token, smartcard or mobile application. These eIDAS assurance levels are Low, Substantial and High. For this paper the latter two levels are most relevant. These two levels require *strong* authentication, i.e. based on at least two of the three *authentication factors*: possession (something the user has), knowledge (something the user knows) and biometrics (something the user is). Strong authentication is also required in the financial sector by the European Payment Service Directive (PSD2), cf. [19,21]. This is known as Strong Customer Authentication (SCA). Although we focus on techniques meeting eIDAS requirements these are also applicable to PSD2. The eIDAS regulation also introduces various forms of electronic signatures of which *qualified* signatures are most relevant for this paper. These signatures provide the highest assurance and are legally equivalent to handwritten ones. Following the eIDAS regulation [18] we call the device (software and/or hardware) holding a private signing key the *Signature Creation Device (SCD)*. The SCD protection of the private key is crucial. Indeed if this key was stolen or copied from the SCD by a fraudster, he could sign on behalf the user.

It is fundamental in the eIDAS regulation that the user has *sole control* over its authenticator/SCD. Before the eIDAS regulation sole control was regulated in the Electronic Signatures Directive 1999/93/EC. This steered discussion among member states. Some "purist" member states, e.g. Germany, interpreted it as a requirement that the user has both physical possession and control of the cryptographic key used for signing (and by extension used for authentication). We refer to this as *classical* sole control as the eIDAS regulation (repealing the directive) explicitly relaxed sole control requirements so that physical control of the keys by the user was not (or no longer) required. As we further elaborate on later, this relaxation in the eIDAS regulation was driven by usability allowing for mobile applications. In this paper we present a cryptographic design supporting mobile strong authentication and qualified signing under classical sole control. In cryptography, signing is closely related to *non-repudiation*, i.e. the property that the user cannot denied having signed something. Although this notion is closely related to sole control, it surprisingly is not mentioned in the eIDAS regulation or the signature directive.

1.2 Role of public key cryptography and certificates

Although the eIDAS regulation is formulated technology neutrally, in practice signing and authentication is typically based on public key cryptography and certificates. The left side of Figure 1, illustrates how public key cryptography can facilitate electronic signing. Here the user generates a public-private key pair where the private key (depicted red) allows signing messages. In practice a short representation of the message (hash) is signed, cf. [53]. The public key (depicted yellow) is provided to service providers and allows for signature verification. A basic property is that the private key cannot be derived from the public key. To link the public key with the user identity, these are typically bound in a so-called public key certificate digitally signed by a Certificate Authority (CA). The public verification key of the CA is publicized, cf. [37]. Obviously, the certificate quality is dependent of its user binding. As indicated in the right side of Figure 1, public key cryptography can also facilitate authentication by letting the user sign a random challenge generated by the service provider. After signature verification, the service provider deduces the user identity from the certificate.

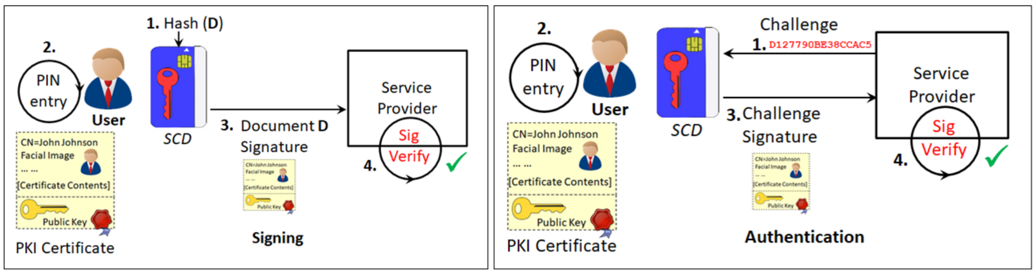


Figure 1. Two prominent PKI use cases

1.3 Classical versus eIDAS "sole control"

For best protection, private keys are historically placed in a smartcard, i.e. a separate integrated circuit card forming the possession factor. On the smartcard an application exists that allows signing with the private key but that precludes exporting it. Also, signing with the private key is under the control of a knowledge factor called Personal Identification Number (PIN). That is, a PKI smartcard is an archetype for an SCD that allows for authentication and signing under classical sole control. A PKI smartcard can conveniently meet the highest eIDAS requirements: the "eIDAS High" assurance level for authentication and qualified signing. Of course this is under the assumption that the certificates sufficiently provide in user binding (e.g. based on a face-to-face registration process). Despite it security advantages, a PKI smartcard has serious usability shortcomings for both users and service providers:

1. **PKI smartcards are user unfriendly**
   Users need to carry around smartcards and equipment (readers) interacting with them, making them user unfriendly. An SCD implementation in the form of a mobile application would be preferable from a usability perspective.
2. **Service providers want outsourcing authentication and signing**
   The direct interaction between service provider and user PKI smartcard also requires user technical support for it. Service providers want to avoid this and want to outsource this to specialized intermediate parties only providing the "happy flow" end result, i.e. an authentication or signature, to the service provider. This is also known as the *centralized approach.*

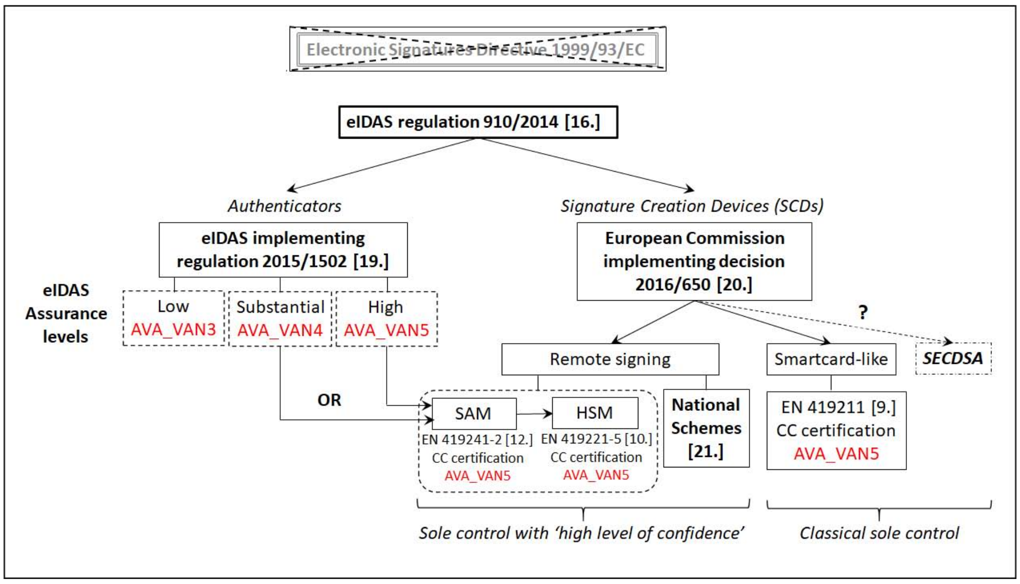


Figure 2. Legal overview combined with recommendations/standardization

These shortcomings were addressed in the eIDAS regulation and further regulations and standards by lowering the Electronic Signatures Directive requirements. Figure 2 gives an overview of the legal situation combined with the recommendiations/standarisations. The former is indicated with continuous lines and the latter with dashed lines. In contrast with this directive, the eIDAS regulation stipulates the user should have sole control [only] with "a high level of confidence", cf. [18, Article 26]. This allows for other authenticators and SCDs than based on PKI smartcards. Most notably it allows for mobile applications where an intermediary party sits between the user and the service provider handling user interaction. See Figure 3. In case of authentication such parties are known as *authentication providers* and in case of signing as *(remote) signing services*. OpenID Connect [5] and SAML [50] are protocols facilitating the centralized approach as indicated in Figure 3.

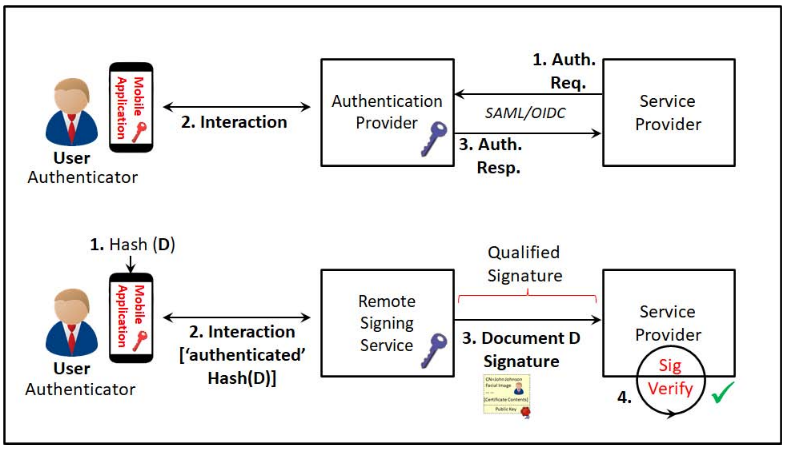


Figure 3. Setup allowed by eIDAS regulation

The SCD requirements for signing under classical sole control, e.g. with smartcard like devices, are fully specified in a 2016 eIDAS implementing decision of the European Commission [24]. These require smartcard-like SCDs to be Common Criteria (CC) certified at EAL 4+ against the protection profile in the European standard EN 419211 [10]. The (technical) requirements for remote signing indicated in Figure 3, where there is no classical control, are not yet specified at European level. This is left to the member states by [24]. To date only seven member states (Austria, France, Germany, Italy, the Netherlands, Slovakia, and Spain) have notified such national schemes [25]. We refer to [40] for further background. The national requirements in [25] vary but there seems to be consensus that the user signing keys at the signing service need to be managed in a remote Signature Creation Device (rSCDev) conforming to the European standard EN 419221-5 [11]. In practice such an rSCDev takes the form of a so-called *Hardware Security Module (HSM)*. This is further elaborated on in the upcoming European standard EN 419241-2 [13] which specifies a Signature Activation Module (SAM). Compare Figure 4. According to the standard, the SAM forms the bridge between the "signer interaction component" (SIC) in the user environment authenticating the user and the HSM holding the user signing key on the other. In practice the SIC is an authenticator from our context, allowing the user through the SAM to instruct the HSM generating qualified signatures. For qualified signing EN 419241-2 requires so-called Sole Control Assurance Level 2 (SCAL2) to be implemented between the SIC and the SAM. The SCAL2 requirements are further specified in the first of the European standard, i.e. EN 419241-1 [12]. In this standard its is noted that SCAL2 is "aimed to achieve the same sole control assurance level as what would be achieved by a stand-alone QSCD", i.e. a PKI smartcard. We conclude that European standards seem to confirm the necessity of classical sole control, at least between the authenticator and the SAM.

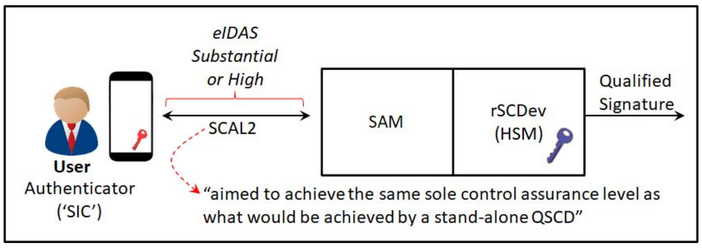

eIDAS
Substantial
or High
SCAL2
SAM
rSCDev
(HSM)
Qualified
Signature
User
Authenticator
('SIC')
"aimed to achieve the same sole control assurance level as what would be achieved by a stand-alone QSCD"

Figure 4. Setup of European standards EN 419221-5 and EN 419241-2

1.4 Link with the eIDAS assurance levels

Regarding the authentication between the user and the SAM the European standard EN 419241-1 requires at least eIDAS assurance level Substantial. The most stringent security requirement the eIDAS regulation [23] poses on authenticators is that they must be resistant against a certain *attack potential*, respectively Moderate and High for authenticators of assurance level Substantial and High. The eIDAS guidance [27] recommends this term to be interpreted in the sense of the Common Criteria methodology for IT security evaluation, i.e. as Appendix B of [36]. This means that, in Common Criteria terms, the eIDAS guidance recommends that authenticators of assurance level Substantial (High) are tested against AVA_VAN.4 (AVA_VAN.5).

For at least three reasons it seems remarkable that not eIDAS assurance level High is required in the European standard EN 419241-1. First, legally binding signing can be considered as one of the most security critical applications, thus requiring the highest assurance level available. Secondly, one can argue eIDAS assurance level High is what "a high level of confidence" refers to in Article 26 of the eIDAS regulation. Thirdly, and perhaps strongest, the CC certification against the European standard EN 419211 [10] of a smartcard-like SCD includes AVA_VAN.5 (Advanced Vulnerability Assessment level 5). This consists of testing for resistance against attacker potential High which is also the recommended basis for the eIDAS assurance level High, cf. [23,27]. The eIDAS assurance level Substantial is only recommended to provide resistance against attacker potential Moderate, i.e. AVA_VAN.4. So not only that the user authenticating part of remote signing is not required to be Common Criteria certified as the other parts are, it also requires less resistance against attacker potential than the other parts. This is illustrated in Figure 2. It also appears that the authenticating part of remote signing in [13] is not consistent with the corresponding SCD security requirements in [10]. As a chain is only as strong as its weakest link, this seems inconsistent. For consistency, one can argue that the European Commission decision [24] should strive for a better middle ground between remote and smartcard-like signing by two changes. Firstly it should only require that smartcard-like SCDs are resistant against attacker potential High, i.e. without requiring full CC certification. Secondly it should require that the authenticating part of remote signing should adhere to eIDAS assurance level High. We will argue that SECDSA can achieve all this.

Regardless of legal requirements; a remote signing service sitting between the user and the service provider should (ideally) not be able to act as man-in-the-middle. That is, it should cryptographically be precluded that the intermediary party is able to authenticate or sign on behalf of the user without his consent. This property is closely related to support of classical sole control, but also to support of *end-to-end security* between the user and the relying party (service provider). In the our context this implies protection of both integrity and authenticity of the communication between the user and the relying party. This is conceptually the simplest technical way avoiding man-in-the-middle attacks. Such end-to-end security is not required in the eIDAS regulation [18,23] or in the European standard [13]. That is, a man-in-the-middle attack on a qualified signing service is not required to be precluded cryptographically. In fact, as far we know all qualified remote signing services in practice are theoretically susceptible to such attacks. We note that such end-to-end security does seem to be required in the NIST authentication guidelines [48]. For its highest assurance level (FAL3) NIST requires an "holder of key assertion" where the user (and not any other party) proves possession of an authentication key to the relying party.

Also note that this risk does not occur in the classical setup of Figure 1 where there is direct interaction between the device and the service provider providing both classical sole control and end-to-end security. All and all, the centralized approach depicted in Figure 3 allowed by the eIDAS regulation can be considered a trade-off between security and usability.

Ideally one would not to have to make such a trade-off in the centralized setup and to still have both classical sole control and end-to-end security. In Figure 5 we have depicted the desired situation which is based on eIDAS assurance level High and in which both classical sole control and end-to-end security are supported as indicated.

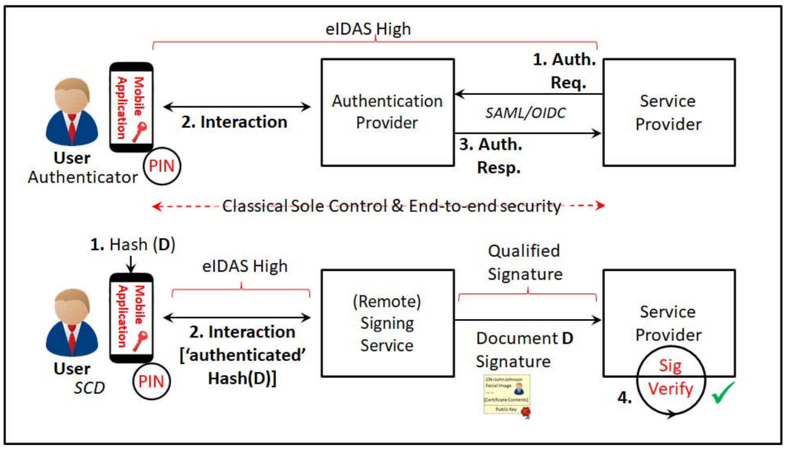


Figure 5. Cryptographic security objectives

The eIDAS guidance [27] seems to implicitly require that a possession factor of an authenticator consists of cryptographic hardware holding (signing) keys in non-exportable fashion like smartcards do.[1] Luckily standard mobile platforms (Apple/iOS, Android) typically have such cryptographic hardware. This is known as the *Secure Enclave* in the context of Apple/iOS [1] and *hardware backed keystore* in the context of Android [3]. Around 90 percent of current smartphones have such hardware a number that will increase to a 100 percent in a few years.[2] This also indicates that the use of such cryptographic hardware as possession factor is "state of the art" in strong mobile authentication. Authentication deals with disclosing user personal data to relying parties and typically also allows access to user personal data stored at relying parties. In that sense strong authentication can be seen as a technical data protection measure. Article 32 of the European General Data Protection Regulation (GDPR) [17] stipulates that such measures should take into account "state of the art". In other words, in addition to the eIDAS guidance also the GDPR seems to require the use of cryptographic hardware as a mobile authentication possession factor.

Complicating in achieving classical sole control and end-to-end security is the limited nature of this hardware. It allows managing signing keys in a non-exportable fashion like smartcards but does not similarly support PINs controlling signing keys. So one of the challenges in achieving our objectives is to design PINs in way that they support classical user sole control.

The eIDAS regulation [23] requires authenticators to be resistant against a certain *attack potential*. As indicated earlier, the eIDAS guidance [27] recommends this term to be interpreted in the sense of the Common Criteria methodology for IT security evaluation, which is not trivial to accomplish. So even if one argues that end-to-end security between user and service provider is not formally required in the eIDAS regulation, it certainly is helpful in proving resistance against (high) attack potentials.

1.5 Cryptographic objectives of this paper

In this paper we present a novel cryptographic technique called Split-ECDSA (SECDSA) achieving a mobile authentication and signing application with the objectives indicated in Figure 5. That is, classical user sole control and end-to-end security. One of the SECDSA innovations is that relying parties do not get access to the actual signatures produced by the mobile device based on two authentication factors, but only to zero-knowledge proofs that these signatures were formed and exist. SECDSA is based on the standard cryptographic hardware in mobile devices that is already required for a possession factor through binding with non-exportable signing keys, cf. Footnote 1. On this limited cryptographic hardware also an additional knowledge factor (PIN) is based. This hardware is not the bottleneck in meeting the highest eIDAS assurance level.

[1] The guidance writes: "[...] it is important that reproduction of it [possession-factor] by a third party is so difficult and unlikely that the risk of this is negligible."

[2] Based on `https://deviceatlas.com/blog/most-popular-iphones` (nearly) all iPhones used are 5s or higher which possess a Secure Enclave. Android 5 and onwards has a hardware backed keystore; starting from Android 7 support was mandatory. According to Android Studio the respective installed base percentages are 94,1 and 73,7. See `https://youtu.be/XZzLjllizYs`.

This is indicated from the successful peer review and notification of three eIDAS High mobile authenticator applications which are also based on this limited hardware. This relates to solutions from the Netherlands, Latvia and Belgium, cf. [14,15,16]. However, the (cryptographic) designs and properties of the three solutions mentioned are not made public whereas ours is openly specified in this document. We note that the European digital identity wallet recently proposed by the European Commission, also seems to require open specifications. See Section 4.

To minimize platform dependence our design does not rely on biometrics. Application of biometrics not only introduces various technical and compliance complications but also requires issuers (remotely) verifying that registered biometrics belong to users. One can also argue that "sole control" is difficult to achieve with biometric authentication as this can be coerced. These complications also emerge from the successful eIDAS High peer-review and notification of the Latvian and Belgium mobile authentication solutions. Their eIDAS notifications are conditional under disabling biometric authentication, cf. [15,16]. Despite all these objections raised against the use of biometric authentication, in Section 3.5 we do hint how this could be supported within SECDSA.

Document outline

- Section 2 contains the cryptographic prerequisites used in this paper.
- Section 3 contains the basic idea behind SECDSA and formalizes a full description in the form of a signing S-APP useable as SECDSA building block.
- In Section 4 we apply the S-APP in example SECDSA use cases in a decentralized approach allowing users selectively disclosing attributes based on eIDAS High authentication. This coincides with required functionality in a European proposal [20] for a European Digital Identity Wallet as part of an update of the eIDAS regulation [18].
- In Section 5 we apply the S-APP in example SECDSA use cases in a centralized approach.
- In SECDSA, so-called PIN-binders play an important role. Appendix A describes various PIN-binder constructions based on limited cryptographic hardware also supported in Apple or Android based mobile devices.

In this paper we also motivate that SECDSA allows implementing authentication/signing in conformity with various laws and regulations. In Figure 6 below we have indicated some of these laws and regulations and the sections where this is discussed.

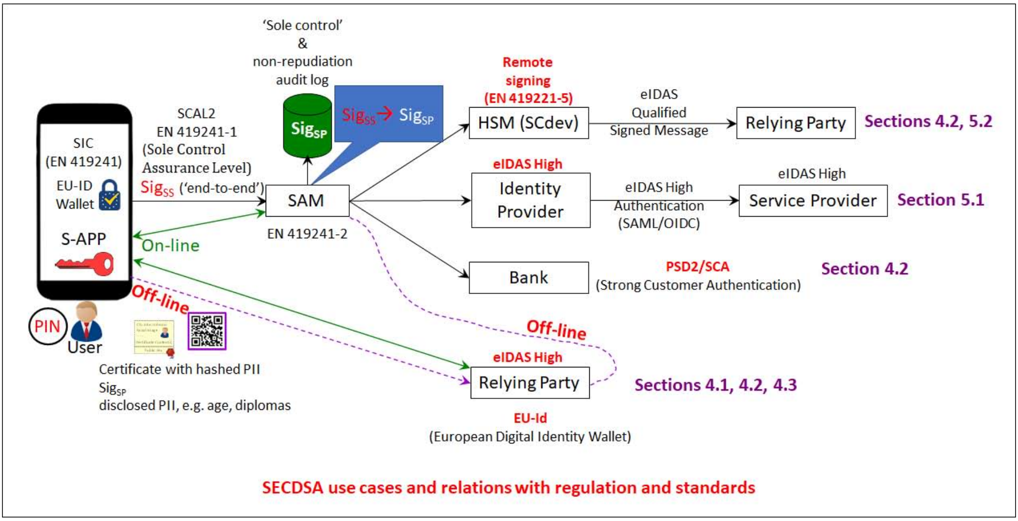


Figure 6. Laws and regulations supported by SECDSA

2 Cryptographic primitives, notation and conversions

2.1 Mathematical context and notation

We let $\mathbb{F}_r$ denote the Galois field consisting of the integers modulo a prime number r. We let $\mathbb{F}_r^*$ denote the multiplicative subgroup, i.e. the non-zero elements. See [53]. We sometimes implicitly use that $\mathbb{F}_r$, respectively $\mathbb{F}_r^*$, corresponds to the integers in the interval $[0, r-1]$, respectively $[1, r-1]$ and write operations in combination with "$\bmod r$". We let $|r| = \lceil \log_{256}(r) \rceil$ denote the size in bytes of r, i.e. the minimal number of bytes to represent r.

Central in our constructions is an additive group $\mathbb{G} = (\langle G \rangle, +)$ of order q generated by a *base point* (generator) G. We use additive notation as this is customary in the context of elliptic curve groups we deploy in practice. We assume that q is prime. For any natural scalar n and element $H \in \langle G \rangle$ we define the *(point) multiplication* nH as adding H n-times, e.g. $2H = H + H$. As $nH = mH$ if and only if $n = m \bmod q$ we can represent scalars as elements of $\mathbb{F}_q$. This allows for compact notation as $x \cdot G$, $-x \cdot G$ for $x \in \mathbb{F}_q$ and $y^{-1} \cdot G$ for $y \in \mathbb{F}_q^*$. We sometimes omit the "$\cdot$" symbol and simply write xG. A cryptographically secure (pseudo) randomly chosen element from a set is denoted by $\in_R$.

The required cryptographic security of the group $(\langle G \rangle, +)$ can be formulated in the intractability of three problems. The first one is the *Diffie-Hellman problem*: computing the values of the function $DH_G(xG, yG) = xyG$ for any $x, y \in \mathbb{F}_q$ (implicitly given but unknown). The second problem is the *Decision Diffie-Hellman* (DDH) problem: given $A, B, C \in_R \langle G \rangle$ decide whether $C = DH_G(A, B)$ or not. An equivalent definition is as follows. Any quadruple of points (G, A, B, C) in $\langle G \rangle$ can be written as (G, A, xG, yA) for some (unknown)

$x, y \in \mathbb{F}_q$. DDH amounts to deciding whether a random quadruple of points in G is a *DDH quadruple*, i.e. if $x = y$. The DH problem is at least as difficult as the DDH problem. The last related problem is the *discrete logarithm* (DL) problem in $\langle G \rangle$: given $A = xG \in \langle G \rangle$, with $x \in \mathbb{F}_q$ then find $x = DL_G(A)$. It easily follows that the DL problem is at least as difficult as the DH problem.

We assume that all three introduced problems in G are intractable which implies that the size $|q|$ of the group order should be at least 256 bits. Although strictly speaking not necessary, we assume in our constructions that $\mathbb{G}$ is a group of points over a field $\mathbb{F}_p$ on a curve with simplified Weierstrass equation

$$y^2 = x^3 + ax + b \tag{1}$$

for some suitable $a, b \in \mathbb{F}_p$. That is, each non-zero group element takes the form (x, y) where $0 \leq x, y < p$ satisfying Equation (1) modulo p. Compare [32]. We denote the zero element (point at infinity) as $\mathcal{O}$. For practical implementations one can use one of the NIST curves, e.g. P-256. Compare [44].

A *Personal Identification Number (PIN)* is a numeric string of length typically 4, 5 or 6. For generality and simplicity of presentation we choose a more generic PIN representation. A PIN is a byte array of length L, i.e. of the form $\{P_{L-1}, P_{L-2}, \ldots, P_0\}$ with all $0 < P_i < 256$, i.e. representing a non-zero byte value. Typically the byte values correspond to printable ASCII-values in which case the PIN can also be represented as a string. For example, the ASCII value of the digits 0, 1, 2, 3, 4 are 0x30, 0x31, 0x32, 0x33, 0x34 in hexadecimal representation. Consequently the PIN "01234" is a PIN of length 5 and is represented by the array {0x30, 0x31, 0x32, 0x33, 0x34}.

2.2 The Digital Signature Algorithm (DSA)

Part of our novel setup for strong authenticators is a method to strongly bind the user PIN to a digital signature. This method works for several standard digital signature algorithms including the most commonly used DSA. See [44]. We specify this method for ECDSA, the Elliptic Curve variant of DSA, from which the generic method for DSA easily follows. We first describe the working of ECDSA. The context of this is a group $\mathbb{G} = (\langle G \rangle, +)$ introduced earlier. Here the user has a private key $u \in \mathbb{F}_q^*$ and a public key $U = uG$. Algorithms 1 and 2 below specify ECDSA signing and verification following [32]. In this construction a secure hash function $\mathcal{H}(.)$ appears, cf. [53,43]. Such a function takes as input byte arrays of arbitrary input and outputs a byte array of fixed length equal to $|q|$. The latter can be accomplished by taking a secure hash function with larger output size and truncating its output.

Algorithm 1 ECDSA signature generation
Input: message M, private key u
Output signature (r, s).

```
1: Compute $\mathcal{H}(M)$ and convert this to an integer $e$.
2: Select random $k \in \{1, ..., q-1\}$.
3: Compute $kG = (x, y)$ and convert $x$ to integer $\bar{x}$.
4: Compute $r = \bar{x} \bmod q$. If $r = 0$ go to Line 1.
5: If $r \bmod q = 0$ then go to Line 1.
6: Compute $s = k^{-1}(e + u{\cdot}r) \bmod q$. If $s = 0$ go to Line 1.
7: Return $(r, s)$.
```

We remark that in the situations where cryptographic hardware is used, the calculation of the hash value of message M in Line 1 of Algorithm 1 is typically not performed by this hardware. This is typically due to communicational or computational restrictions in using the hardware. In these circumstances the hash value of message M is pre-computed in the application calling the hardware and then sent to the hardware. The hardware then converts the hash value to the integer e of Line 1 of Algorithm 1 and performs the following Lines 2-7. This setup is known as *raw* signing, i.e. generation of a signature directly on basis of a hash value without a deploying a hash operation. The cryptographic hardware in Apple and Android based platforms, cf. Section 1, also support raw signing.

Algorithm 2 ECDSA signature verification
Input: message M, signature (r, s), public key U
Output: Acceptance of rejection of the signature.

```
1: Verify that $r, s$ are integers in interval $[1, q-1]$. On failure reject the
   signature.
2: Compute $\mathcal{H}(M)$ and convert this to an integer $e$.
3: Compute $w = s^{-1} \bmod q$.
4: Compute $t_1 = e{\cdot}w \bmod q$ and $t_2 = r{\cdot}w \bmod q$.
5: Compute $X = t_1{\cdot}G + t_2{\cdot}U$.
6: If $X = \mathcal{O}$ reject the signature.
7: Convert the x-coordinate of $X$ to an integer $\bar{x}$; compute $v = \bar{x} \bmod q$.
8: If $v = r$ accept the signature otherwise reject it
```

For our design it will be convenient to have an alternative but equivalent representation of an ECDSA signature and corresponding verification algorithm. This representation, called the *full* representation, takes the form $(R, s) \in \langle G \rangle \times [1, q-1]$. The alternative verification algorithm is specified in Algorithm 3 below.

Algorithm 3 Alternative ECDSA signature verification
Input: message M, signature (R, s), public key U
Output: Acceptance of rejection of the signature.

```
1: Verify that $\mathcal{O} \neq R \in \langle G \rangle$ and that $s$ is integer in interval $[1, q-1]$. On
   failure reject the signature.
2: Compute $\mathcal{H}(M)$ and convert this to an integer $e$.
3: Compute $w = s^{-1} \bmod q$.
4: Convert the x-coordinate of $R$ to an integer $\bar{r}$; compute $r = \bar{r} \bmod q$.
5: Compute $G' = wG$ and $U' = wU$
6: Compute $X = eG' + rU'$.
7: If $X = R$ accept the signature otherwise reject it
```

Although straightforward, for further reference we prove equivalence between the two representations of ECDSA signatures and verifications algorithms.

Proposition 2.1 *Let Y be an ECDSA public key and M a message, then the following are equivalent:*

1. *One possesses a valid ECDSA signature (r, s) on message M.*
2. *One possesses a valid full ECDSA signature (R, s) on message M.*

Proof: Suppose one possesses a valid signature (r, s) on message M. Then it follows that (X, s) is a valid full ECDSA signature where X is as in Line 5 of Algorithm 2. Conversely suppose one possess a full valid signature (R, s) on message M. Then it follows that (r, s) is a valid ECDSA signature where r is as in Line 4 of Algorithm 3. □

2.3 Schnorr proofs of knowledge

To prevent certain attacks it will be fruitful to have techniques allowing parties (the user and the AP) proving they possess certain secret values (keys) without providing any information on those. Such proofs are known as *zero-knowledge proofs of knowledge (ZPK).* In this section we recall the ZKPK techniques of Schnorr, cf. [52]. Here it is convenient to also allow other generators U, V of $\mathbb{G}$ then G.

We let $D = d{\cdot}U$ be a public key in $\mathbb{G}$ with respect to U with corresponding private key $d \in \mathbb{F}_q^*$. The private key holder forms

$$E = d{\cdot}V \tag{2}$$

and sends E and V to another person (verifier) together with his public key D and generator U (implicitly defining d). Now suppose the holder wants to prove the form of E in Formula (2) to the verifier. That is, the holder wants to provide the verifier some information T allowing the later to verify this. In fact, this should be publicly verifiable (also known as "transferable"): anybody should be able to verify this. The simplest way to do this would be full disclosure of the holder private key d. However, we require that the information T provided should not leak any secret information on d.

The Schnorr proofs of knowledge [52] allow for this in an interactive protocol between the holder and the verifier. In these proofs the holder first commits to certain values related to Formula (2), the verifier then sends a challenge which the holder can only answer with a suitable response if Formula (2) holds. These proofs of knowledge do not "leak" secret information ("zero-knowledge") on d as it can be shown that the verifier essentially does not get any information he could not have generated himself.

The Schnorr protocols can also be made non-interactive (and transferable) using the Fiat-Shamir heuristic [31]. Here the verifier challenge is replaced with a secure hash of the holder commitment. That means that the holder can generate a transcript that allows the verifier (and in fact anybody) to verify that Formula (2) holds. We denote such transcript by

$$\mathcal{DT}(V \overset{d}{\rightarrow} E \,|\, D = d{\cdot}U).$$

In the following two algorithms we specify how such transcripts can be created and verified proving Formula (2). Their correctness and security follow from [52]. The transcripts not only prove Formula (2) to the verifier but also that the prover knows private key d. In this sense the transcripts can be considered an extension of the Schnorr signature algorithm based on private key d, cf. [34, Section 6.10]. As in Section 2.2 we let $\mathcal{H}(.)$ represent a hash function producing byte arrays equal to the byte length $|q|$ of the group order q. In Algorithm 4 we allow including an additional byte array A binding it to the proof of knowledge. This further strengthens the resemblance with Schnorr signatures. The byte array A can be empty, which actually is the case in the applications in this paper.

Algorithm 4 $\mathcal{DT}_c(V \overset{d}{\rightarrow} E \,|\, D = d{\cdot}U, A)$
Creation of transcript by holder of a private key d and binding to byte array A.

```
1: Select random $k \in \{1, ..., q-1\}$.
2: Compute $k{\cdot}U$, $k{\cdot}V$ $(i = 1, \ldots, n)$, and convert to byte arrays $\bar{U}, \bar{V}$.
3: Compute byte array $\mathcal{H}(\bar{U}||\bar{V}||A)$ of size $|q|$ and convert it to integer $r$
4: If $r = 0$ then go to Line 1.
5: Compute $s = k + r \cdot d \bmod q$.
6: If $s = 0$ then go to Line 1.
7: Return $(r, s)$.
```

In Line 2 of Algorithm 4 we write each of the elliptic curve points in their x- and y-coordinates and concatenate these as input to the hash function in Line 3.

Algorithm 5 $\mathcal{DT}_v(V, E, \mathcal{DT}, D, A)$
Verification of transcript $\mathcal{DT} = (r, s)$ by verifier using public key $D = d{\cdot}U$.

```
1: Verify that $V, E \in \mathbb{G}$ on failure Return False.
2: Verify that $r \in \{1, 2^{8\cdot|q|} - 1\}$ and $s \in \{1, q-1\}$, on failure Return False.
3: Compute $Q_1 = s{\cdot}U - r{\cdot}D$, $Q_2 = s{\cdot}V - r{\cdot}E$
4: if $Q_1 = \mathcal{O}$ or $Q_2 = \mathcal{O}$ Return False.
5: Convert $Q_1, Q_2$ to byte arrays $\bar{Q_1}, \bar{Q_2}$.
6: Compute byte array $\mathcal{H}(\bar{Q_1}||\bar{Q_2}||A)$ of size $|q|$ and convert it to integer $v$.
7: If $v = r$ Return True otherwise Return False.
```

We remark that by the nature of Schnorr based proofs of knowledge there is a negligible probability (in the order or $2^{-8\cdot|q|}$, i.e. 2^{-256} in the context of the

NIST curve P-256, that Algorithm 5 is erroneously successful. For simplicity we do not further stipitate that in the algorithms. In Section 3.4 we show how one can deploy ECDSA to provide alternative proofs of knowledge albeit at the expense of some theoretical security.

3 SECDSA

In this section we specify Split-ECDSA (SECDSA) allowing two factor based signing and authentication on standard mobile devices under classical sole control. In Section 3.1 we describe the SECDSA context, the basic idea behind SECDSA is in Section 3.2 from which we arrive at the full specification in Section 3.3 by first developing some further heuristics. In Sections 3.4 and 3.5 we outline some SECDSA enhancements and alternatives respectively.

3.1 SECDSA context

In the SECDSA context we will design a signing APP (S-APP) allowing users generating signatures that are publicly verifiable by relying parties. S-APP forms the basis for several applications which are outlined in Sections 4, 5. S-APP consists of the following elements (cf. Figure 7):

1. **User and Relying Party**
   A user that wants to generate SECDSA signatures on messages for a relying party that is verifiable with the user public key certificate. The signing process is based on a possession and a knowledge factor referred to as PIN.
2. **Platform**
   The physical manifestation of S-APP for the user, e.g. a mobile device such as a smartphone.
3. **Operation System (OS)**
   The OS supports interaction with the outside world including the user, forms the basis for the user application environment and the interaction of that with the Secure Cryptographic Environment (see below).
4. **Signing Application (S-APP)**
   In S-APP all the functional part of the signing application is implemented and some of the security part. S-APP has its own local storage that is typically separated by the local storage of other applications by the OS.
5. **Secure Cryptographic Environment (SCE)**
   The SCE allows applications like S-APP to securely manage cryptographic keys, e.g. for signing. An application like S-APP can instruct SCE to generate secret cryptographic keys which can then be used in certain cryptographic operations but cannot be exported from the SCE. This allows such keys to form a possession factor. The OS will typically only allow applications using their own key material and not those of others.
6. **Certificate Issuer (CI)**
   A trusted party that issues public key certificates binding user identities Id with SECDSA public keys $\mathbf{Y}$.

7. **Signing Facilitator (SF)**
   A trusted party that assists S-APP in generating the SECDSA signatures. The certificate issuer and signing facilitator roles can be combined. The necessity of and rationale for the SF which will be explained below.

In the context of Apple devices, the SCE is formed by its secure enclave [1] whereas in the context of Android based devices the SCE is typically formed by its "hardware-backed keystore" [3]. The secure enclave consists of a separate cryptographic processor whereas the "hardware-backed keystore" is typically based on a trusted execution environment based on the device processors. However, Android also supports separate cryptographic processors known as *StrongBox*, similar to the Apple secure enclave. If a StrongBox is available its use is advisable over the use of the hardware backed keystore.

Over 80-90 percent of current smartphones have such hardware, cf. Footnote 2 in the introduction. The Apple secure enclave is certified against FIPS 140-2 level 2. Also the StrongBox is required to be common criteria certified against assurance level EAL5 augmented by AVA_VAN.5, i.e. resistant to attack potential High. Compare Section 1.3.[3] Both Apple and Android SCEs support (raw) ECDSA signing based on the NIST curves [44]. In all cases ECDSA private keys are non-extractable from the SCE.

We remark that the (Apple) secure enclave should not be confused with a *secure element*. Although it has the same abbreviation, it typically has the form of an enhanced SIM card in a mobile phone, also known as UICC (universal integrated circuit card). Such enhanced UICCs can be considered as PKI smartcards and can be embedded by the device manufacturer or issued by the telecom provider (as "SIM"). Generally speaking UICC usage by third parties introduces several technical and financial (access is typically not free) issues. By contrast the Apple secure enclave and the Android hardware-backed keystore and StrongBox (if available) are freely useable by developers.

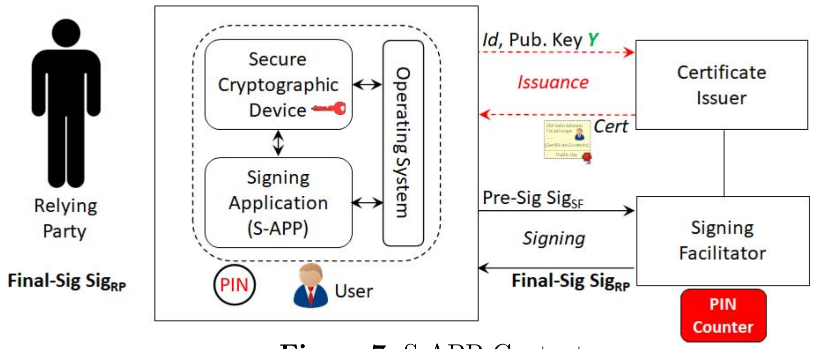


Figure 7. S-APP Context

[3] See https://support.apple.com/en-us/HT209632 and Section 9.11.2 of https://source.android.com/compatibility/11/android-11-cdd.

Establishing a possession factor in the context above is easy: generate a public-private key pair in the SCE (*SCE-key*) that allows for signing messages. So the question is how to implement an additional possession factor (PIN) in this setup. This PIN could be managed by a separate server. As part of the signing process the user would then sign the message with the SCE-key and separately authenticate to the server by entering his PIN. This server would then form a new signature with a signing key under control of the server. However, such a setup conflicts with 'sole control', moreover it will not allow for (two-factor) end-to-end security to relying parties. Indeed, how would the server prove to a relying party it really was the user that entered the PIN?

3.2 Idea behind SECDSA

The above discussion leads to the question if it is feasible to form a signing key in the user platform that is based on both a SCE-key and on a key derived from a PIN (*PIN-key*), i.e. based on some key derivation function [49]. Given the limited capabilities of a typical SCE this appears impossible in general. However the characteristics of the (EC)DSA algorithm allow for a special construction accomplishing precisely that. This construction is presented in the next algorithm and it properties are proven in the proposition following it.

Algorithm 6 Split-ECDSA (SECDSA) signature generation
Input: message M, SCE-key $u \in \mathbb{F}_q^*$, PIN-key $\sigma \in \mathbb{F}_q^*$
Output signature (r, s).

```
1: Compute $\mathcal{H}(M)$ and convert this to an integer $e$.
2: Compute $e' = \sigma^{-1} \cdot e \bmod q$
3: Select random $k \in \{1, ..., q-1\}$
4: Compute $kG = (x, y)$ and convert $x$ to integer $\bar{x}$
5: Compute $r = \bar{x} \bmod q$. If $r = 0$ go to Line 1
6: If $r \bmod q = 0$ then go to Line 1
7: Compute $s_0 = k^{-1}(e' + u \cdot r) \bmod q$. If $s_0 = 0$ go to Line 1
8: Compute $s = \sigma \cdot s_0 \bmod q$
9: Return $(r, s)$
```

Note that the pair (r, s) appearing in Lines 3-7 of Algorithm 6 is just a raw ECDSA signature on e' with respect to the SCE-key u. Compare the remarks following Algorithm 1 describing ECDSA. This means that Lines 3-7 simply consist of calling the SCE to generate a raw signature on e' with respect to the SCE-key u. In Line 2 the input of the SCE signature is modified using the PIN-key as is the outputted signature itself in Line 8. The next proposition states that this results in a signature based on both the SCE-key and the PIN-key.

Proposition 3.1 *Algorithm 6 returns an ECDSA signature (r, s) on message M based on the private key $u \cdot \sigma \bmod q$, i.e. the product of the SCE-key u and the PIN-key σ.*

Proof: We show that the ECDSA signature validation (Algorithm 2) always returns True with input M, (r, s) and public key $Y = u \cdot \sigma \cdot G$. To this end, clearly

r, s are integers in the interval $[1, q-1]$ as required in Line 1 of Algorithm 2. As we remark above (r, s_0) is a raw ECDSA signature on e' based on private key u. By inspecting Algorithm 2 (ECDSA verification) it follows that r is equal to the x-coordinate of

$$\begin{aligned} e' {\cdot} s_0^{-1} G + r {\cdot} s_0^{-1} U &= e {\cdot} \sigma^{-1} {\cdot} s_0^{-1} G + r {\cdot} s_0^{-1} U \\ &= e {\cdot} s^{-1} G + r {\cdot} \sigma {\cdot} s^{-1} U) \\ &= e {\cdot} s^{-1} G + r {\cdot} s^{-1} (\sigma {\cdot} u G) \end{aligned} \tag{3}$$

The first equality follows as $e' = \sigma^{-1} {\cdot} e \bmod q$ by the construction in Line 2 of Algorithm 6. The second equality follows as $s_0^{-1} = \sigma {\cdot} s^{-1} \bmod q$ which directly follows from the construction in Line 8 of Algorithm 6. The last equality follows from a direct verification and the definition of public key U.

It follows that (r, s) is a raw ECDSA signature on e based on private key $\sigma {\cdot} u$. That is, (r, s) is an ECDSA signature on message M based on private key $\sigma {\cdot} u$. □

SECDSA security is based on the security of raw ECDSA signing, i.e. not letting the cryptographic hardware compute the hash value itself but instead but letting it sign only the result, i.e. the hash value. We let a *hash value* be a byte array representing a big integer of the size of $|q|$, e.g. 32 bytes in the case of the NIST curve P-256. In the raw signing context, the hash value is computed by the application calling the cryptographic library/hardware. As cryptographic hardware typically has computational or communicational restraints, raw signing is typically used in practice. This is why most (if not all) cryptographic libraries/hardware including the iOS/Secure Enclave, Android/HBK+Strongbox, TPMs, PKCS11 based HSMs support this.

With *raw ECDSA signing security* we mean that the following attack is not possible: an attacker can request the SCE (cryptographic hardware) to sign a series of chosen hash values $H_1, H_2, ..., H_n$ with the SCE-key u (and public key $u {\cdot} G$) and is then able to generate a signature based on SCE private key u on a value H that was not requested by the attacker. Note that the raw ECDSA signing security notion is stronger than what is commonly used for security of a signature scheme, cf. [39]. The latter only allows the attacker to choose messages to be signed that first need to be hashed. In raw ECDSA signing security, the attacker has the freedom to choose hash values which gives him more freedom. Nevertheless, one can argue that raw ECDSA signing security is commonly accepted to hold, as ECDSA raw signing is commonly used and supported by most (if not all) cryptographic libraries/hardware. We can similarly define *raw SECDSA signing security* by replacing the SCE private key u with the SECDSA private key $\sigma {\cdot} u$ (with public key $Y = u {\cdot} \sigma {\cdot} G$). The following proposition shows that raw SECDSA signing security follows from raw ECDSA signing security.

Proposition 3.2 *If one can break raw SECDSA signing security, one can break raw ECDSA signing security.*

Proof: First of all, from the series of equalities (3) appearing in Proposition 3.1 one can conclude the following (actually the converse from what is used in this proposition): If (r, s) is an ECDSA signature on hash value e with private key $u{\cdot}\sigma$ (and public key $\sigma{\cdot}u{\cdot}G$) then $(r, s{\cdot}\sigma^{-1})$ is an ECDSA signature on hash value $e{\cdot}\sigma^{-1}$ with private key u.

Now suppose that an attacker can break SECDSA security, i.e. the attacker can call Algorithm 6 to SECDSA sign specific hashvalues $H'_1, H'_2, ..., H'_n$ leading to a SECDSA signature (r, s) on a hashvalue H' not requested. From the above conclusion it then follows that $(r, s{\cdot}\sigma^{-1})$ is signature on hashvalue $H'{\cdot}\sigma^{-1}$ with private key u. Also, in Algorithm 6, the calls for signing with private u are $H'_1{\cdot}\sigma^{-1}, H'_2{\cdot}\sigma^{-1}, ..., H'^{-1}_n{\cdot}\sigma$ which are all different from $H'{\cdot}\sigma^{-1}$ as $H'_1, H'_2, ..., H'_n$ and H' are different. We have arrived at an successful attack on raw ECDSA signing security with the underlying possession key u. □

We remark that SECDSA uses two different keys (an SCE-key and a PIN-key) to form an ECDSA signature, but works fundamentally different than ECDSA threshold signing [41] that also deploys two keys. In the latter one key is under control of the user and the other by another party, e.g. an authentication provider in our context. In SECDSA both keys are under control of the user. Moreover, by their nature threshold signing does not allow for implementation in standard cryptographic standard cryptographic hardware in mobile devices.

Proposition 3.1 now naturally gives rise to SECDSA public-private key pairs, where the private key is the product of a SCE-key and a PIN-key. To this end, it is convenient to introduce the notion of a *PIN-binder*. This is a key derivation function $\mathcal{P}(.,.)$ enabling deriving a secret cryptographic PIN-key $\mathcal{P}(K, P_P)$ from a master key K_P (also known as *PIN-binder key*) and a PIN P, i.e. an array of non-zero bytes of length L. The basic KDF security property is that, provided the master key K is suitable chosen, the derived keys are as secure as truly random keys in practice. Compare [49]. In SECDSA we require a PIN-binder to take non-zero values, i.e. in $\mathbb{F}^*_q$. The idea is that, like the SCE-key u, the PIN-binder key K_P is also generated and maintained in the SCE and that the secret cryptographic operations leading to the PIN-binder value take place inside the SCE. As the SCE needs to support this, the latter places restrictions on the PIN-binders we can use. In Appendix A we describe a few PIN-binder constructions when the SCE is formed by a Apple or Android based mobile device. If an Android device supports a StrongBox then its use as SCE is advisable over the use of the hardware backed keystore. See the discussion on page 16. In fact, when a StrongBox is available one could also base the PIN-binder on *combining* the StrongBox and the hardware backed keystore. This would consist of running the PIN based cryptographic output of one as input to the other resulting in a PIN-key dependent of keys in both the StrongBox and the hardware backed keystore. This would provide some additional security over only using the StrongBox. In the next algorithm we formalize the generation of SECDSA public-private key pairs. Here we assume that a PIN-binder is chosen in S-APP.

Algorithm 7 SECDSA key generation in S-APP
Input: user PIN
Output: SECDSA public key $\mathbf{Y} = u{\cdot}\mathcal{P}(K_P, \mathrm{PIN}){\cdot}G$ with u an SCE-key and K_P a PIN-binder key both maintained in the SCE.

```
1: S-APP instructs SCE to generate ECSDA key $u$ with public key $U = u{\cdot}G$
2: S-APP instructs the SCE to generate a PIN-binder key $K_P$
3: The user is asked to choose a PIN      // entered twice to avoid errors
4: S-APP generates $\sigma = \mathcal{P}(K_P, \mathtt{PIN})$ and computes $\mathbf{Y} = \sigma{\cdot}U$
5: S-APP deletes $\sigma$
6: S-APP returns $\mathbf{Y}$
```

We also refer to u and U appearing in Line 1 as the SCE private and public key respectively. The public key $\mathbf{Y}$ is called the SECDSA public key and $\mathbf{y} = u{\cdot}\sigma$ the SECDSA private key. By Proposition 6 it is now obvious how SECDSA signing takes place. We have fully formalized this in Algorithm 2 below.

We note an attacker with both local access to call the SCE and possession of public key $\mathbf{Y}$ would be able to mount a brute-force attack on the PIN. Indeed, such an attacker would execute lines 4-6 of Algorithm 7 for all possible PINs until he meets one with a public key equal to $\mathbf{Y}$. This is why in the full SECDSA specification (Section 3.3) the public key is treated as sensitive data that are not stored inside S-APP and encrypted outside S-APP. Also, as will be clear in the next section, the SECDSA public key $\mathbf{Y}$ is not stored (in plaintext) at either the certificate issuer or the signing facilitator. In theory an (internal) attacker of these *trusted* services could retrieve the SECDSA public key $\mathbf{Y}$ of a user by gaining unauthorized access to a certain encryption key (ZKP-key a). However, the attacker can only brute-force the user PIN under the condition the attacker has also access to the user platform/SCE. One can argue that this risk is manageable in practice as it is comparable with the risk related to PIN Unlocking Keys (PUKs). Such PUKs allowing resetting the PIN code and are commonly retained at PKI smartcard issuers. Indeed, an (internal) attacker could in theory also gain unauthorized access to a user PUK. However, such an attacker can only reset the PIN under the condition he also has access to the user smartcard. We further think that brute-forcing the PIN in SECDSA context will considerably harder for an adversary than using a PUK code. Indeed, before such PIN brute-forcing can take place the attacker first has to bypass the platform security, e.g. “jailbrake” or “root” the device without losing the data stored on it. In Section 3.4 we discuss two further controls mitigating local PIN brute-force attacks: usage of Hardware Security Modules (HSMs) and controlling the time it takes calculating a PIN-key.

3.3 Full SECDSA description

We first develop some further heuristics on the basic SECDSA idea leading to the final SECDSA specification in Protocols 1 and 2. As indicated above the public key $\mathbf{Y}$ cannot be stored on the device and also needs to appear encrypted in the user certificate to preclude the PIN is brute-forceable from the mobile device in principle. In Protocol 1 we specify how the certificate issuer encrypts a

SECDSA based public key before placing them in the certificate. This protocol also includes the signing facilitator introduced earlier.

To further explain the role and necessity of the signing facilitator; it is well known that one can recover the ECDSA public key used from a valid ECDSA signature on a known message M. Compare [9, Section 4.6]. This implies that also the SECDSA signature returned by Algorithm 6 needs to be encrypted when leaving S-APP. In Lines 12-15 of Protocol 2 it is specified how S-APP encrypts an SECDSA signature. It follows that verification of a SECDSA signature by a relying party must somehow take place on basis of an encrypted public key (in the certificate) and an encrypted signature.

Now if this encrypted signature was completely formed by S-APP alone then an attacker would be able to perform a PIN brute-force with SCE access. It follows that the encryption of the SECDSA signature must be performed in two steps: one step by the S-APP and one by another party: the signing facilitator. The facilitator will only perform this cryptographic operation if the signature sent by S-APP was correct, implicitly implying that the PIN entered was correct. To this end, the signing facilitator maintains a PIN-counter; which is incremented with each incorrect signature sent by S-APP. When the PIN-counter exceeds a certain threshold, the signing facilitator accepts no further signing requests from the user. In Lines 18-26 of Protocol 2 we specify how the signing facilitator produces the SECDSA signatures in their final form.

In the following protocols we let, as before, $\mathcal{H}(.)$ be a secure hash function with output length equal to $|q|$ bytes. We also assume that the certificate issuer and signing facilitator have securely agreed a *ZKP private key* $a \in_R \mathbb{F}_q^*$ and published the *ZKP public key* $G' = a{\cdot}G$. This public key could be wrapped in a separate certificate or could be part of the certificate holding the certificate verification public key of the certificate issuer. We further assume there is a suitable encrypted connection setup, e.g. based on HTTPS, between the partners in the protocol.

Protocol 1 Certificate issuance to User/S-APP by Certificate Issuer (CI)

```
1: User chooses PIN, S-APP generates SECDSA key-pair $\mathbf{Y}$ using Algorithm 7
2: S-APP sends CI identity $Id$, $\mathbf{Y}$ and Proof-of-Possession of $\mathbf{y}$
3: CI verifies the Proof-of-Possession of $\mathbf{y}$, on failure protocol ends
4: CI verifies $Id$, on failure protocol ends                  // out of scope
5: CI generates an array of $|q|$ random bytes $\mathbf{CId}$ (certificate identifier)
6: CI computes $\mathbf{Y}' = a{\cdot}\mathbf{Y}$.                        // encryption of $\mathbf{Y}$
7: CI generate ZKP $\mathcal{DT} = \mathcal{DT}(\mathbf{Y} \overset{a}{\to} \mathbf{Y}' \,|\, G' = a{\cdot}G)$      // $G'$ is ZKP-public key
8: CI generates certificate $C$ based on $Id, \mathbf{Y}', \mathcal{H}(\mathbf{CId})$
9: CI sends $\mathbf{CId}$, $C$, $\mathcal{DT}$, to User
10: User/S-APP verify $C$ and $\mathcal{DT}$, if successful S-APP stores $\mathbf{CId}$, $C$
11: S-APP deletes $\mathbf{Y}$ and $\mathcal{DT}$                          // sensitive data
```

The certificate in Line 8 of Protocol 1 can be based on the X.509 standard [37]. This consists of registering a SECDSA "SubjectPublicKeyInfo" ASN.1 structure holding $G', \mathbf{Y}'$ at standardisation organisations. In this way issuing and revocation of SECDSA certificates can also benefit from standard mechanisms, like

Certificate Revocation Lists (CRLs) and the Online Certificate Status Protocol (OCSP), cf. [30].

One could base the identity verification in Line 4 on an existing means of authentication, i.e. earlier issued to the user. This is also allowed by the eIDAS regulation, cf. [23, Section 2.1.2]. The personal data that the means of authentication can provide to the certificate issuer can also be placed in the issued certificate. In some cases the means of authentication is a separate eID-application placed on an electronic passport/identity card conforming to the ICAO specifications[4]. In this situation the personal data residing on the electronic passport/identity card includes a facial image of the user and is signed by the issuing government. When the certificate issuer can securely link this personal data to the identity provided by the means of authentication, the certificate issued could include the personal data residing on the electronic passport/identity card. In many situations, e.g. in the Netherlands, this linking can be done based on the user social security number that is contained in both data.

The zero-knowledge proof $\mathcal{DT}$ in Line 7 allows the user/S-APP to validate that $\mathbf{Y}'$ is correctly formed, i.e. as in Line 6. Using this zero-knowledge proof is optional as the certificate issuer is trusted by the user. It is essential that $\mathcal{DT}$ is deleted in Line 10 as it can used to check if a guess of $\mathbf{Y}$ (based on a guess of the PIN) is correct. That is, with $\mathcal{DT}$ an attacker can brute-force the PIN if he has access to the SCE which we want to avoid.

We remark that the hash value $\mathcal{H}(\mathbf{CId})$ in Line 8 could be used as the certificate serial number. Theoretically there could be two certificates then with the same serial number but the probability is about $2^{-|q|}$, thus negligible in practice.

We also envision the use of Protocol 1 when the user wants to change his PIN. In this application the verification in Line 4 is then handled by letting the certificate issuer act as relying party in Protocol 2. The user signs his request for a PIN-change (and certificate re-issuance) using his existing SECDSA certificate (and PIN). After issuance of the new certificate, the existing certificate is directly revoked by the certificate issuer. In Section 3.5 we sketch an alterative giving the users the ability to change their PIN without changing their SECDSA public key (and certificate). As is indicated in ths section this is of only limited value and also introduces a (limited) security risk.

When the user is issued a certificate by Protocol 1 he can use S-APP to generate SECDSA based signatures on messages for relying parties. As explained in the above heuristics the generated SECDSA signatures must be encrypted and S-APP and a signing facilitator. SECDSA signing by S-APP for a relying party (RP) is specified in the following protocol which includes assistance by a signing facilitator (SF).

[4] https://www.icao.int/publications/pages/publication.aspx?docnum=9303

Protocol 2 SECDSA signature generation by S-APP for RP assisted by SF
Input: message M
Output: SECDSA signature Sig_{RP}

```
1: User opens S-APP and indicates he wants to sign a message.
2: S-APP retrieves $\mathbf{CId}$ and certificate $C$ from local storage
3: S-APP sets up session with SF by sending $\mathbf{CId}$ and $C$
4: SF verifies certificate $C$                // correctly signed not revoked
5: SF computes $\mathcal{H}(\mathbf{CId})$ and verifies that certificate $C$ is based on it
6: SF looks up PIN-counter for certificate $C$, on failure initialises one
7: SF checks if PIN-counter exceeds threshold, if so returns Error to S-APP
8: SF validates $C$, on success sends random nonce $N$ to S-APP
9: S-APP retrieves $\mathbf{Y}'$ from certificate $C$
10: S-APP requests user to enter PIN
11: S-APP calls SCE to compute $\sigma = \mathcal{P}(K_P, \texttt{PIN})$.            // inside SCE
12: S-APP computes $H = \mathcal{H}(M), H' = \mathcal{H}(H||N)$ and converts $H'$ to integer $e$.
13: S-APP computes $e' = \sigma^{-1} \cdot e \bmod q$
14: S-APP calls SCE to compute raw signature $(r, s_0)$ on $e'$ with SCE-key $u$
    // inside SCE
15: S-APP computes $s = \sigma \cdot s_0 \bmod q$.
16: S-APP transforms $(r, s)$ to full form $(R, s)$ using Proposition 2.1.
17: S-APP computes $w = s^{-1} \bmod q$
18: S-APP computes $S' = w \cdot G'$, $S'' = w \cdot \mathbf{Y}'$             // $G'$ is ZKP-public key
19: S-APP Generates ZKP $\mathcal{DT}_1 = \mathcal{DT}(G' \overset{w}{\to} S' \,|\, S'' = w \cdot \mathbf{Y}')$
20: S-APP sends $Sig_{\mathtt{SF}} = \{H, (R, S', S''), N, \mathcal{DT}_1\}$ to SF
21: S-APP deletes all transient data
22: SF verifies $\mathcal{DT}_1$ on $S', S'', G', \mathbf{Y}'$, $N$ on failure returns Error to S-APP
    // also check nonce $N$ from Line 8
23: SF converts the x-coordinate of $R$ to integer $\bar{r}$; computes $r = \bar{r} \bmod q$
24: SF computes $H' = \mathcal{H}(H||N)$ and converts $H'$ to integer $e$
25: SF computes $R' = a \cdot R$                                   // ZKP-key $a$
26: Verify if $R' = e \cdot S' + r \cdot S''$
    On failure, SF increments PIN-counter with 1, returns Error to S-APP
    On success, SF resets PIN-counter to 0
27: SF generates ZKP $\mathcal{DT}_2 = \mathcal{DT}(R \overset{a}{\to} R' \,|\, G' = a \cdot G)$
28: SF sends signature $Sig_{\mathtt{RP}} = \{Sig_{\mathtt{SF}}, R', \mathcal{DT}_2\}$ to RP
    // mechanism depends on use case, cf. Sections 4, 5
```

Note that Lines 1-20 of Protocol 2 always results in an ECDSA signature sent to the signing facilitator. Depending on whether the user has entered the right PIN it will be correct or not. By including the nonce N sent by the SF in Line 8 in the SECDSA signature, it is impossible for an adversary to successfully replay the output of Line 16 in Protocol 2 to reset the PIN-counter. This is particulary important as this output is part of the final signature sent to relying parties.

The concatenation in Line 12 of Protocol 2 corresponds with a very basic implementation. Many variations exist to incorporate the message M and nonce N in the core SECDSA signature. To prevent potential subtle security issues, it is best not to use the concatenation of H, N but a suitable injective function.

For instance, one can encode H, N as a DER (Distinguished Encoding Rules, [38]) encoded SEQUENCE and take that as input for the hash function.

We remark that the implicit PIN validation by the Signing Facilitator in Line 26 of Protocol 2 can valuable as part of security monitoring. It allows, for instance, to detect PIN guessing by monitoring the number of invalid PIN entries and when they appear. This is actually an advantage of SECDSA over a PIN used in a regular PKI smartcard.

We did not specify in Protocol 2 how the SECDSA signature is sent to a relying party. In Sections 4 (respectively 5) we discuss various setups arranging for that in a decentralized (respectively centralized) way. Regardless of how the relying party receives it, Algorithm 8 specifies how it can be verified.

Algorithm 8 Encrypted SECDSA signature verification by Relying Party
Input: message M, User certificate C, Sig_{RP}
Output: True of False

```
1: Parse $Sig_{\mathrm{RP}} = \{Sig_{\mathrm{SF}}, R', \mathcal{DT}_2\}$ on failure return False        // inputcheck
2: Parse $Sig_{\mathrm{SF}} = \{H, (R, S', S''), N, \mathcal{DT}_1\}$ on failure return False// inputcheck
3: Verify certificate $C$, on failure return False // signed, not revocated
4: Retrieve $\mathbf{Y}'$ from certificate $C$
5: Verify $\mathcal{DT}_2$ on $R', R, G', G$ on failure return False        // Algorithm 5
6: Converts the x-coordinate of $R$ to an integer $\bar{r}$; compute $r = \bar{r} \bmod q$
7: Verify $\mathcal{H}(M) = H$ on failure return False
8: Compute $H' = \mathcal{H}(H||N)$ and converts $H'$ to integer $e$
9: Verify if $R' = e{\cdot}S' + r{\cdot}S''$ on failure return False
10: Verify $\mathcal{DT}_1$ on $S', S'', G', \mathbf{Y}'$ on failure return False       // Algorithm 5
11: Return True
```

In the heuristics developed at the begin of this section we concluded it is essental that the raw SECDSA signature appearing in Line 16 is suitable encrypted to prevent the SECDSA public key $\mathbf{Y}$ can be decuded from it allowing a PIN brute-force with access to the user SCE. This encryption takes place in Lines 17-18 of Protocol 2 where the encrypted raw SECDSA signature takes the form (R, S', S''). Note that the proof of knowledge $\mathcal{DT}_1$ on this encryption is zero-knowledge implying that this leaks no information on the raw SECDA signature. The following result proves that a PIN brute-force is not succesfully possible for an attacker even with valid SECDSA signatures and access to the user SCE. For simplicity we only provide an informal formulation and proof which can be further formalized.

Proposition 3.3 *In the context of Protocol 2 the following hold.*

1. *An attacker that can verify if a candidate SECDSA public Γ is correct (i.e. equal to $\mathbf{Y}$) based on the SECDSA signatures provided by Protocol 2, can also solve the Decision Diffie-Hellman (DDH) problem in $\langle G \rangle$.*
2. *As we assumed the DDH problem to be intractable in $\langle G \rangle$, an attacker cannot successfully mount a brute-force attack on the user PIN based on valid SECDSA signatures even with access to the user SCE.*

Proof: This follows as the verification indicated coincides with the verification if the quadruple (G, Γ, S', S'') is a DDH-quadruple, cf. Section 2.1. □

The following result proves that SECDSA delivers sole control and end-to-end security from the user to the relying party.

Proposition 3.4 *In the context of Protocol 2 the following hold under the assumption that SCE keys are not extractable from the SCE.*

1. *A SECDSA signature on a message M implies the existence of an ECDSA signature created by the user's SECDSA private key in S-APP.*
2. *Only through S-APP the user is able to generate a SECDSA signature on a message M. No party, even the signing facilitator, is able to impersonate the user in the context of SECDSA.*

Proof: This first result follows from the construction of the SECDSA signature Sig_{RP} on a message M which proves the existence of an ECDSA signature (r, s) generated with the plain SECDSA private key $\mathbf{y}$. Indeed, the verification of ZKP $\mathcal{DT}_2$ in Line 5 of Algorithm 8 proves that $R' = a{\cdot}R$ where a is the ZKP private key. Moreover the verification of ZKP $\mathcal{DT}_1$ in Line 10 of Algorithm 8 proves that $S' = \theta{\cdot}G'$ and $S'' = \theta{\cdot}\mathbf{Y}'$ for some $\theta \in \mathbb{F}_q^*$. By construction $G' = a{\cdot}G$ and $\mathbf{Y}' = a{\cdot}\mathbf{Y}$. We conclude that $S' = \theta{\cdot}a{\cdot}G$ and $S'' = \theta{\cdot}a{\cdot}\mathbf{Y}$. From the verification $R' = e{\cdot}S' + r{\cdot}S''$ in Line 9 of Algorithm 8 it thus follows that

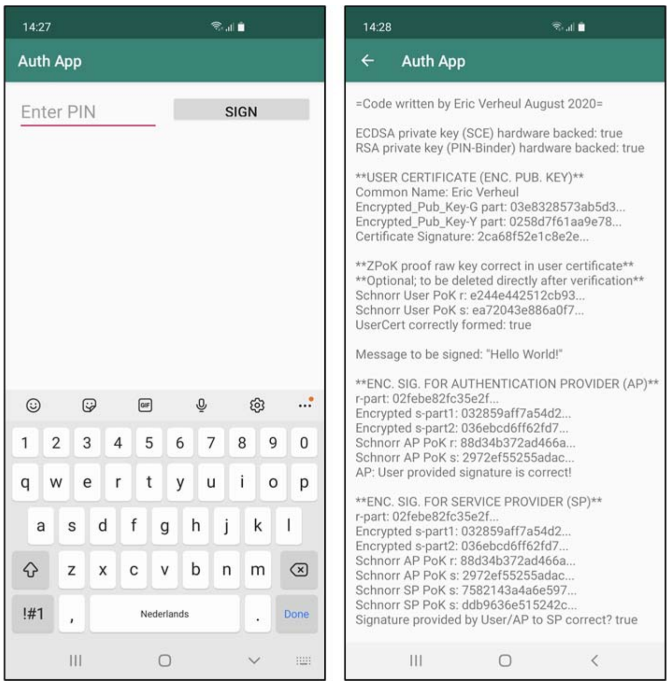
$$a{\cdot}R = e{\cdot}\theta{\cdot}a{\cdot}G + r{\cdot}\theta{\cdot}a{\cdot}\mathbf{Y} = a(e{\cdot}\theta{\cdot}G + r{\cdot}\theta{\cdot}\mathbf{Y}).$$

It thus follows that

$$R = e{\cdot}\theta{\cdot}G + r{\cdot}\theta{\cdot}\mathbf{Y}.$$

It thus follows that (R, θ^{-1}) is a plain SECDSA signature on message M based on private key $\mathbf{y}$. This signature can only be generated by the user. The second result now easily follows. □

As a proof of concept we implemented Protocol 2 and Algorithm 8 completely in one Android Application, i.e. S-APP but also the Signing Facilitator and the verification of a relying party. The PIN-binder was based on RSA in line with Algorithm 10 as illustrated in Figure 8 below. This indicates that SECDSA is easily implementable.



```
=Code written by Eric Verheul August 2020=

ECDSA private key (SCE) hardware backed: true
RSA private key (PIN-Binder) hardware backed: true

**USER CERTIFICATE (ENC. PUB. KEY)**
Common Name: Eric Verheul
Encrypted_Pub_Key-G part: 03e8328573ab5d3...
Encrypted_Pub_Key-Y part: 0258d7f61aa9e78...
Certificate Signature: 2ca68f52e1c8e2e...

**ZPoK proof raw key correct in user certificate**
**Optional; to be deleted directly after verification**
Schnorr User PoK r: e244e442512cb93...
Schnorr User PoK s: ea72043e886a0f7...
UserCert correctly formed: true

Message to be signed: "Hello World!"

**ENC. SIG. FOR AUTHENTICATION PROVIDER (AP)**
r-part: 02febe82fc35e2f...
Encrypted s-part1: 032859aff7a54d2...
Encrypted s-part2: 036ebcd6ff62fd7...
Schnorr AP PoK r: 88d34b372ad466a...
Schnorr AP PoK s: 2972ef55255adac...
AP: User provided signature is correct!

**ENC. SIG. FOR SERVICE PROVIDER (SP)**
r-part: 02febe82fc35e2f...
Encrypted s-part1: 032859aff7a54d2...
Encrypted s-part2: 036ebcd6ff62fd7...
Schnorr AP PoK r: 88d34b372ad466a...
Schnorr AP PoK s: 2972ef55255adac...
Schnorr SP PoK s: 7582143a4a6e597...
Schnorr SP PoK s: ddb9636e515242c...
Signature provided by User/AP to SP correct? true
```

Figure 8. Output of proof of concept SECDSA APP

3.4 SECDSA security enhancements

By encrypting of both the public key and signatures in SECDSA, an attacker is not able retrieve the user PIN from the user platform/SCE, cf. Proposition 3.3. An attacker could therefore try to compromise the ZKP private key a stored at either the certificate issuer or signing facilitator allowing him decrypting SECDSA public keys. With such keys and access to the user platform/SCE, the attacker would be able to brute-force the user PIN if he has also access to the user platform/SCE. In these notes following Algorithm 7 we have motivated that this risk seems manageable as it is comparable with that of handling of so-called PUK codes in the context of PKI-smartcards. We discuss two specific mitigating controls on this risk in this section. One control deals with making the brute-force attack itself less feasible and the other deals with protecting the ZKP private key a in Hardware Security Modules (HSM).

Controlling the PIN-key SCE computation time

We can control the SCE computation time for the PIN-key σ. That is, we can control the computation time in Line 4 of the SECDSA key generation algorithm (Algorithm 7) and in Line 11 of the SECDSA signature generation (Protocol 2). This can be simply achieved by letting this computation consist of several recursive calls to the SCE and the PIN-binder key, i.e. letting the output of one call be the input to the next. For instance, if we base the PIN-key on a recursive hundred RSA decryption operations (see Appendix A) then the computation time of a PIN-key will take about 3 seconds on a Samsung S10 mobile phone. Then a five digit PIN will then take 3 days to brute-force. A PIN consisting of five alphanumeric characters will take over 80 years to brute-force. Controlling the computation time can be beneficial in increasing attack potential resistance required in [23] as it increases the time to execute the attack, one of the attack factors, cf. [36]. If the execution time takes longer than the SECDSA certificate validity one could argue an attack is not practical.

Deploying cryptographic key and mobile application attestation
The SCEs of both Apple [2] and Android [4] provide support for key attestation. Key attestation allows the SCE to convey to outside parties that a generated (ECDSA) public key is actually generated inside a (trusted) SCE in a non-exportable fashion. The inner workings of Apple's key attestation are not completely clear from [2]. Android's attestation works by letting the SCE wrap an SCE generated public key inside a certificate signed with private key inside the SCE certified by Google. This private key is placed inside the SCE as part of the device manufacturing process. This key is certified by Google by issuing a certificate on the corresponding public. To mitigate linkability issues Google deploys private keys that are shared among batches of devices. Also, by allowing the generated certificate to include a challenge sent by an external party, freshness of the public key can also be guaranteed to the external party.

Attestation on the SCE public key could be implemented as part of the certificate issuance process in Protocol 1. In case of Android based devices this could be implemented as an alternative to Step 3 of Protocol 1. There are various ways to achieve this and we sketch one example. Here the user/S-APP sends the SECDSA public key $\mathbf{Y}$, the public SCE key U and the Google certificate attesting the public SCE key U to the certificate issuer. From the attestation certificate the certificate issuer can conclude U is bound to the SCE. Then the certificate issuer would then send a random challenge RC to the S-APP. Next the user/S-APP would perform a proof-of-possession of the private keys u and $\sigma = \mathcal{P}(K_P, \text{PIN})$ by sending two signatures on challenge RC. The signature related to u would be based on ECDSA related to the public key $U = u{\cdot}G$ using the SCE. ECDSA could also be used for private key $\mathbf{y}$, i.e. an ECDSA signature on challenge RC related to the public key $\mathbf{Y} = \sigma{\cdot}U$. Alternatively one could also use a Schnorr signature for the latter. From these two proofs the certificate issuer can also conclude that the user/S-APP has possession of $\mathbf{y} = u{\cdot}\sigma$ as specified in the original Step 3 of Protocol 1.

In this way assurance on the binding of the SCE key to the device, i.e. the possession factor, is provided to the certificate issuer as part of the issuance process. We remark that key attestation can be considered part of attestation of a mobile application as a whole. Here relying parties can assess that a mobile application is not tampered with, i.e. "rooted" or "jailbroken". Both Apple's devicecheck [2] as Google's Safetynet [4] provide such attestation. In SECDSA this could be deployed by both the certificate issuer as the signing facilitator in further risk mitigation.

Implementing SECDSA PIN reset
The provisioning of the SCE public key U indicated above as part of attestation could also form the basis for convenient SECDSA "PIN reset". The context here is a SECDSA user that has forgotten his PIN (or has even locked it at the signing facilitator). Issuers of PKI smartcards typically remedy this situation by providing users with a PIN Unlocking Key (PUK) allowing them to reset the PIN. In SECDSA PIN reset could be remedied by introducing some extra steps in Protocol 1 we now sketch.

First of all, the user/S-APP registers the SCE public key U as part of Protocol 1, extending its attestation use indicated. S-APP represents the PIN code as a byte array P of size L (the length of the PIN code), cf. the end of Section 2.1. Next S-APP generates a random byte array R of size L and registers this at the certificate issuer as part of Protocol 1. S-APP also provides the user with $P' = P \oplus R$ in a printable form, e.g. in hexadecimal format. Here "$\oplus$" represents the byte-wise Exclusive OR (XOR) operation. The user is urged by S-APP to store P' in a safe location (outside the device).

When the user wants to reset his PIN, he invokes a special function of S-APP. This function contacts the certificate issuer referring to **CId**. Then S-APP proves possession of the private key u corresponding to SCE key U after which the certificate issuer provides R. Next S-APP asks the user to enter P' enabling S-APP to reconstruct the PIN P as $P' \oplus R$. This allows S-APP performing a SECDSA authentication to the certificate issuer triggering a PIN change. This PIN change could be formed by rerunning Protocol 1 as indicated in the notes following this protocol or by following Section 3.5.

We remark that typically P' would be supplemented with some check bytes/digits remedying user typing errors. Additional to P' the certificate issuer could also send an additional code to the user. The user would need to provide this code to the certificate issuer after the SCE key possession proof as additional access control before R is provided to S-APP by the certificate issuer.

Using HSMs for protecting the ZKP private key
A certificate issuer or signing facilitator could manage the ZKP private key a in a Hardware Security Modules (HSM). The usage of ZKP private key a in the verification in Line 25 of Protocol 2 corresponds with a so-called Diffie-Hellman operation. This operation is commonly supported in HSM through the PKCS #11 standard [51], through the call `CKM_ECDH1_DERIVE`.

The generation of the zero-knowledge proof $\mathcal{DT}_2$ in Line 27 of Protocol 2 is not supported in the PKCS #11 standard or otherwise commonly supported in HSMs. So for this support one would need to develop a separate HSM module. However, HSMs do have support for ECDSA through the PKCS #11 standard also with respect to non-standard base points. This can used for an alternative knowledge proof based on the assumption that one can only generate ECDSA signatures on (pseudo)random messages if one has access to the corresponding private key. This assumption is an implicit requirement for all digital signature schemes. Although this is assumed to hold for ECDSA it not formally proven. So using ECDSA instead of Schnorr proofs of knowledge is theoretically less secure.

We only sketch the ECDSA based alternative for $\mathcal{DT}_2$. It takes the form of three ECDSA signatures on the signature Sig_{SF} sent by S-APP in Line 18 of Protocol 2. These are based three different public-private key pairs, corresponding to $G' = a{\cdot}G$, $R' = a{\cdot}R$ (base point R), and $G' + R' = a{\cdot}(G + R)$ (base point $G + R$). From the assumption it follows that the signing facilitator knows $\alpha, \beta, \gamma \in \mathbb{F}_q^*$ such that $G' = \alpha{\cdot}G$, $R' = \beta{\cdot}R$ and $G' + R' = \gamma{\cdot}(G + R)$. First, it follows that $\alpha = a$ by definition of the ZPK public key G'. Next, as we have

$$\alpha{\cdot}G + \beta{\cdot}R = \gamma(G + R)$$

it follows that $(\beta - \gamma){\cdot}R = (\gamma - \alpha)G$. Now suppose that α, β, γ are not all equal then it follows $\beta \neq \gamma$ and

$$R = \left(\frac{\gamma - \alpha}{\beta - \gamma}\right) G.$$

Recall that R is the commitment of the original SCE signature, i.e. the point $k{\cdot}G$ generated in Line 3 of the ECDSA signature generation, cf. Algorithm 1. It follows that $k = \frac{\gamma-\alpha}{\beta-\gamma}$ is known by the signing facilitator. As is well-known (and easily seen) one can compute the private SCE-key from the scalar k and a valid signature. So the signing facilitator can compute the private SCE-key. Now observe that Protocol 2 only uses SCE signatures so we arrive at a method for computing the private ECDSA key on basis of its signatures. This contradicts the assumed security of ECDSA. We conclude that $\gamma = \beta = \alpha = a$ as required.

By similar reasoning one could also replace the proof $\mathcal{DT}_1$ in Line 17 of Protocol 2 by three ECDSA signatures. The relying party would then only need to verify ECDSA signatures (six in total, three generated by S-APP and three by the signing facilitator). This could simplify implementation as ECDSA signature verification with non-standard base points is commonly supported in cryptographic libraries. The only relying party verification requiring explicit ECC arithmetic would then be Line 9 of Protocol 8.

3.5 An alternative SECDSA PIN change protocol

As indicated after Protocol 1 we envision that when users want to change their PIN, they simply get issued a new SECDSA certificate. In this section we sketch an alternative allowing users to change their PIN without changing their SECDSA public key and certificate. Instead of deriving the PIN-key from the user PIN and a key inside the SCE, one can also generate a random PIN-key and encrypt this with a symmetric key K_{PIN} derived from the user PIN and a key residing in the SCE. The encrypted PIN-key is then locally stored. As part of signature generation the symmetric key is re-derived from the PIN provided by the user and the key residing in the SCE. This encrypted PIN-key is then decrypted and used as part of the SECDSA signature process. The idea is that this decryption allways results in a PIN-key and depending on whether the correct PIN is provided, the correct PIN-key and SECDSA signature is formed. An advantage of this setup is that it allows changing the PIN without changing the user public key and certificate. The user then presents S-APP both the original as a new PIN. S-APP then decrypts the encrypted PIN-key with the original K_{PIN} and re-encrypts it with the new one resulting in a newly encrypted PIN-key.

As S-APP cannot validate if the original PIN was correctly entered this best involves the verification of a ("PIN-reset") signature by the signing facilitator, i.e. using Protocol 2. It thus follows that from the user perspective this alternative does not seem the have many benefits from the envisioned PIN change mechanism by issuing a new SECDSA certificate. However, this alternative does introduce a new attack scenario. The scenario consists of an attacker that has deep access to S-APP/SCE in two different time periods: during the period the old and the new PIN are active. This access should allow the attacker running a brute-force on all possible PINs, resulting in a list of all possible PIN-keys on both occasions. The correct PIN-key then presents itself as the one on both lists. This type of attack resembles the well-known *evil housekeeper attack*, where an attacker with (physical) access alters a device in some undetectable way so that they can later access the device, or the data on it. One might argue that this risk is acceptable/controllable: an attacker might be able having this kind of access *once* but not twice without alarming S-APP or the user, cf. Section 3.4. More importantly, any mobile authenticator PIN, even the archetype based on a separate (contactless) PKI smartcard, is susceptible to such attacks. Indeed, the attacker could place a key-logger on the mobile device.

Note that the sketched setup might also facilitate the usage of biometric factors as an alternative to knowledge factors (PINs). Indeed, one could encrypt the random PIN-key with a key in the cryptographic hardware that is under (local) biometric access control. It seems that Apple and Android support such functionality. However, as indicated in Section 1.5 the usage of biometrics in mobile authentication seems to be conflicting with the eIDAS High assurance level. Perhaps the usage of biometrics is reconcilable with the eIDAS Substantial assurance level. In this case one could let S-APP support two SECDSA private keys/certificates: one meeting eIDAS Substantial supporting both biometrics and a PIN and one for eIDAS High solely based on a PIN.

We finally sketch a technical design for the alternative setup. Let the PIN-key σ be based on a random byte array S of size $|q| + 8$ bytes. It is essential that S does not hold certain (trailing) bytes indicating for instance it is an unsigned integer. Consides S as a non-negative integer $\bar{S}$ and let the PIN-key be equal to $1 + (\bar{S} \bmod (q-1))$. See Appendix A for the rationale behind this. Any byte array S will result in a non-zero element in $\mathbb{F}_q^*$. The various techniques from Appendix A can also be used in deriving a pseudorandom AES key from a PIN and a key inside the SCE. Now use the byte oriented stream cipher AES-CTR [45] with the zero initialisation vector to encrypt/decrypt byte array S.

4 Decentralized SECDSA use cases (EU-ID Wallet)

In this section we apply SECDSA to three example use cases in a decentralized setting. These use cases coincide with those indicated in the European Digital Identity Wallet (EU-ID wallet) proposed by European parliament and the council as part of an amendment of the eIDAS regulation [20]. The envisioned EU-ID wallet is a mobile application for users (European citizens) of which we have outlined the ten most prominent requirements in SECDSA context in Table 1.

#	Requirements for the European Digital Identity Wallet [20,26]
1.	Allows users to securely request, obtain and store personal attributes (Article 6a paragraph 3a).
2.	Allows users to authenticate and thereby selectively disclose and combine their personal attributes (Article 6a paragraph 3a).
3.	Allows users to legally sign documents (qualified signing) in an easily accessible way throughout Europe (Article 6a paragraph 3b).
4.	Meets the eIDAS assurance level High on authentication (Article 6a paragraph 4c).
5.	Provides a “common interface” between wallet and relying parties specified on a European level (Article 6a paragraph 4a).
6.	Allows relying parties to verify the authenticity of the user attributes (Article 6a paragraph 4b).
7.	Ensures that issuers of wallet attributes cannot receive information about the use of these attributes (Article 6a paragraph 4b).
8.	Is usable in both on-line as off-line environments (Article 6a paragraph 3a).
9.	Issued attributes should be revocable within 24 hours. (Amendment (25)) *Note: this amendment lets attributes be treated as certificates.*
10.	Should be based on open standards.

Table 1. Prominent European Digital Identity Wallet requirements

The last requirement is implicitly posed in [26] as a *conditio sine qua non* for an effective European Digital Identity Framework. It forms a middle ground between the current deployment of proprietary/closed authenticaton solutions and requiring that such solutions are fully ”open source” for which there is strong societal interest too.

We note that EU-ID wallet closely resembles the “digital base identity” (Dutch “digitale bron identiteit”) envisioned by the Dutch State Secretary for the Interior and Kingdom Relations in his report to Dutch parliament[5]. In this section we outline how a EU-ID Wallet can be based on SECDSA. The basic idea is that we let the EU-ID attributes be formed by the SECDSA certificates as discussed in Section 3. In Section 4.4 we show that a SECDSA based EU-ID Wallet can meet all the EU-ID requirements from Table 1 with .

[5] https://www.rijksoverheid.nl/documenten/kamerstukken/2021/02/11/kamerbrief-over-visie-digitale-identiteit

In Sections 4.1, 4.2 we assume an on-line context: both the EU-ID wallet and service provider are on-line. In Section 4.1 we sketch how the first use case can be supported and in Section 4.2 the second use case. One of the more challenging EI-ID Wallet requirements is that the EU-ID wallet use case should also support off-line usage. In Section 4.3 we sketch two ways to accomplish this with SECDSA.

In all use cases we assume the user has been issued a SECDSA certificate, i.e. Protocol 1 has been successfully executed. We note that, due to the usage of (X.509) digital certificates in SECDSA, the "common interface" the EU-ID wallet and service provider that needs specified by the European Commission can be based on an extension of the Transport Layer Security (TLS) protocol [33] and its use of X.509 client certificates.

4.1 EU-ID wallet authentication and selective disclosure

In this use case the service provider directly communicates with the user's EU-ID wallet. The service provider requires the user to sign a challenge C as part of an authentication. This challenge will be similarly handled as the message M in protocol 2, i.e its hash will be part of Sig_{SF} et cetera. If this signing is successful the service provider then associates the user to the attributes (personal data) in his certificate. In line with the EU-ID wallet requirements, users should be to selectively disclose attributes based on eIDAS High level authentication. The SECDSA based use case indicated in Figure 9 can conveniently cater for such a EU-ID wallet using the S-APP from Section 3. Steps 2-7 are handled by Protocol 2; we have only outlined this protocol in Figure 9. By Proposition 3.4 this setup provides sole control and end-to-end security between the user/EU-ID wallet and the service provider.

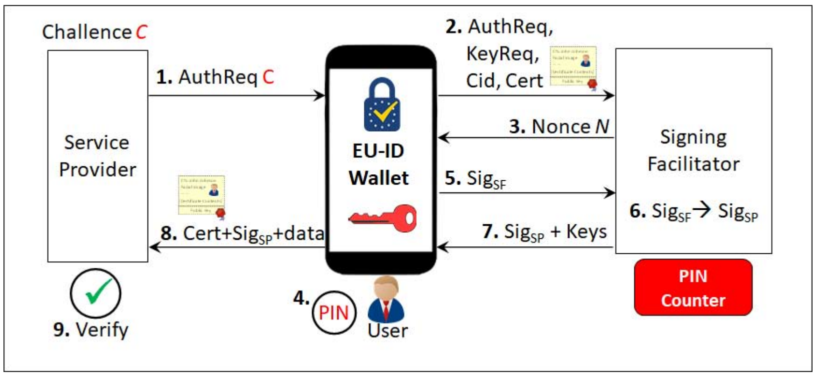


Figure 9. Authentication with selective disclosure

SECDSA also allows for selective disclosure of attributes by letting the certificate issuer only store hash values of attributes in the certificates as part of certificate issuance, i.e. Protocol 1. The attributes themselves are locally stored inside the EU-ID wallet in encrypted form. The keys giving access to the attributes are managed by the signing facilitator. As part of this use case the user first decides which of the attributes in the certificates he wants to reveal to the service provider. Next the user does not only ask the signing facilitator in Step 2 to help producing a SECDSA signature Sig_{SP} but also to provide him with the required keys. That is, the keys giving access to the attributes the user wants to reveal to the service provider. This request is also part of the signature in Step 5. If this signature is correct the signing facilitator also provides the required keys in Step 7. The user can next decrypt the attributes, inspect them and send them to the service provider in Step 8. The service provider then computes the hash values of these attributes and checks that they are present in the SECDSA signature. If so, the service provider accepts the attributes. The keys allowing attribute decryption should be deleted by the EU-ID wallet after the authentication is completed.

By simply placing hash values of attributes in SECDSA certificates makes this setup vulnerable to brute-forcing of attributes. This issue can simply be addressed by pre-pending a random byte array of fixed length, say 16 bytes, to the attribute before hashing it. The hash value H is then placed in the certificate, compare Figure 11. This byte array then also makes part of the encrypted attributes locally stored in the EU-ID wallet.

Note that by this basic attribute hashing construction, the signing facilitator does not have access to the attributes. We can also arrange that the certificate issuer does not have access to the attributes either. This can be accomplished by letting him accept attribute hash values of other issuing parties. During the issuance process, these parties are then provided with the appropriate keys to encrypt the attributes and send them to the user/EU-ID wallet to locally store them. We remark the keys controlling the attributes can be conveniently derived by the signing facilitator from the certificate serial number.

The flexibility and simplicity of using SECDSA certificates can be further improved by issuing certificates on a shared SECDSA public key as indicated in Figure 10. The basis is a “digital base identity” issued by the government holding basic identifying personal data as a conventional passport. This could correspond with the eIDAS minimum data set as specified in [22]. In this setup there are then several eIDAS UniquenessIDs corresponding to the different member states. The digital base identity would then be used by the user to authenticate to certificate issuers that then issue additional certificates based on the same SECDSA public key the digital base identity is based. This setup resembles the “digitale bron identiteit” (Dutch) envisioned by the Dutch State Secretary for the Interior and Kingdom Relations in his report to Dutch parliament. See Footnote 5.

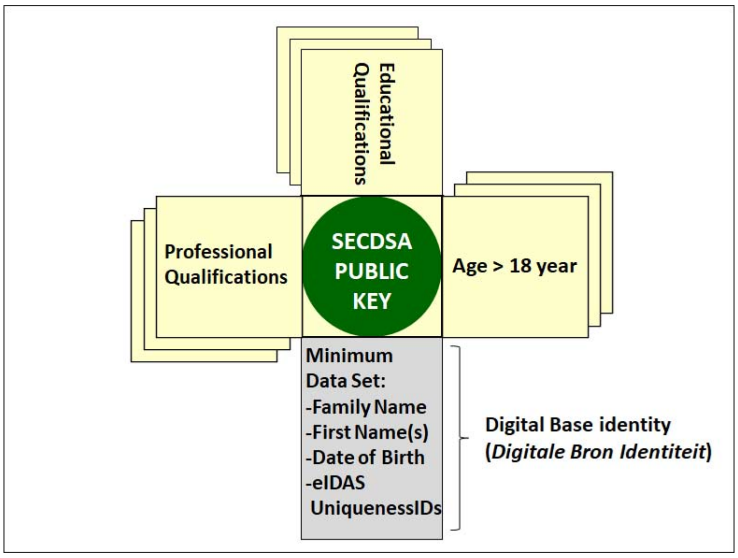


Figure 10. Flexible SECDSA certificates by shared public key use

Unlike, for instance, Camenisch–Lysyanskaya (CL) signatures [8] used in the Dutch IRMA APP[6], SECDSA does not provide for multi-show unlinkability. That is, users can be recognized/linked by service providers through their SECDSA public key and certificate. This can be considered a privacy weakness when attributes are not directly identifying themselves, e.g. only indicating age over 18 years. We remark that although multi-show unlinkability is theoretically valuable, it is only practically effective when supplemented with anonymous communications mechanisms which are not commonly used. Although this weakness can be mitigated (see below) it can be considered a trade-off between security, privacy and user-friendliness. Indeed, CL private keys are not supported in the limited hardware of standard mobile devices, hampering a solid possession authentication factor and thus security and eIDAS compliance, cf. Footnote 1. Moreover, authenticators with multi-show unlinkability are not recognizable by service providers complicating authenticator revocation further complicating eIDAS compliance. By contrast SECDSA certficates can use standard X.509 certificate revocation mechanisms (OCSP, CLRs), cf. [28,37].

SECDSA linkability can be mitigated by letting the user regularly (or even automatically) renew their certificate based on the existing one. That is, by the same mechanism envisioned for PIN change. Compare the notes following Protocol 1. To avoid linkability in this setup through the attribute hash values in the new certificate, one can deploy a two step approach. The idea is not to place the attribute hash value H in the certificate but a second hash value H_2. This is similarly formed as H using a second random byte array RBA_2 as indicated in Figure 11. As part of the attribute disclosure to the service provider, the

[6] www.irma.app

user/EU-ID wallet do not only send the attribute value including the random byte array but also RBA_2. This allows the service provider to recompute H_2 and to verify that that is in the SECDSA certificate. During the initial certificate issuance the certificate issuer provides the EU-ID wallet the second random byte array RBA_2 and uses that to compute hash value H_2 as indicated. This value H_2 is then placed in the certificate instead of H. During certificate re-issuance, the wallet sends RBA_2 and H to the certificate issuer allowing him to verify H is indeed inside the certificate. During re-issuance the certificate issuer generates a new RBA_2 that is sent to the EU-ID wallet and used to compute a new hash value H_2 placed in the new certificate.

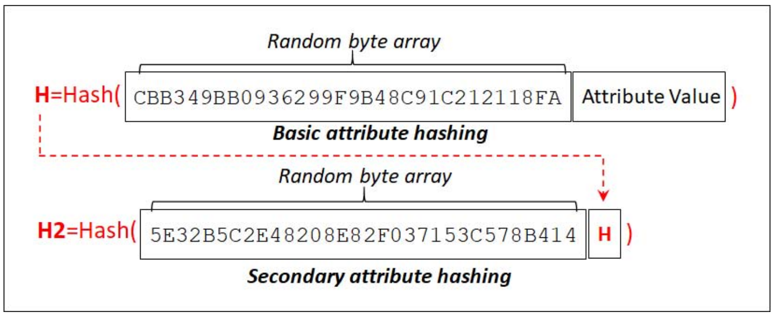

Random byte array
H=Hash(
CBB349BB0936299F9B48C91C212118FA
Attribute Value
)
Basic attribute hashing
Random byte array
H2=Hash(
5E32B5C2E48208E82F037153C578B414
H
)
Secondary attribute hashing

Figure 11. Attribute hashing

4.2 EU-ID wallet signing

In this use case the service provider directly communicates with the user's EU-ID wallet. The service provider requires the user to sign a certain message M, e.g. a contract. The SECDSA based interactions for this use case are indicated in Figure 12. Here Steps 2-7 are handled by Protocol 2; we have only outlined this protocol in Figure 12. If successful, this protocol will result in a SECDSA signature Sig_{SP} on message M. By Proposition 3.4 this setup provides sole control and end-to-end security between the user/EU-ID wallet and the service provider.

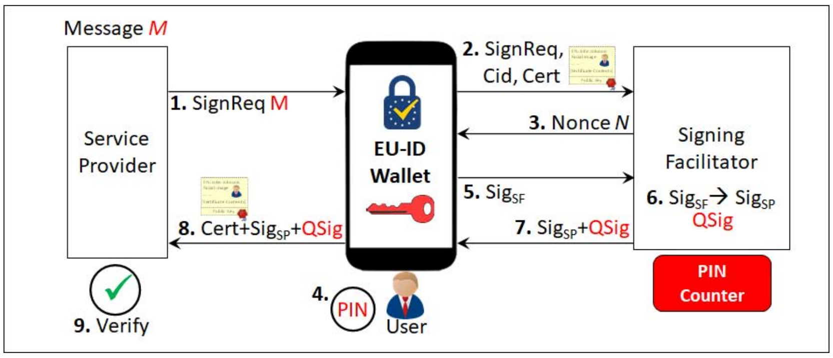


Figure 12. Qualified signing with additional SECDSA evidence

An interesting question is how this setup could allow for legally binding, i.e. qualified, signatures. A straightforward setup for this would be to only use the EU-ID wallet as a means of authentication to the Signing Facilitator and letting that implement a 'conventional' qualified remote SCD in the sense of the second paragraph of Article 1 in the 2016 European Commission decision [24]. This qualified remote SCD would then need to be certified against a national scheme where conformance to European standards EN 419221-5 [11] and EN 419241-2 [13] seems to be most future proof. The first standard regulates the Cryptographic Module (Hardware Security Module) holding the user private signing keys and the second standard regulates the Signature Activation Module (SAM) which handles user authentication and instructs the Cryptographic Module to sign on behalf of the user. Also compare Section 1.3. Formally, the EU-ID wallet would then only need meeting the authentication to the SAM on eIDAS assurance level Substantial. We think this is easily feasible as we argued in Section 1.5 one can achieve the eIDAS assurance level High. In such setup the Signing Facilitator could be implemented as part of the SAM, i.e. steps 23-28 of Protocol 2. In an extra step 29 the Signing Facilitator would then instruct the Cryptographic Module to generate a completely new signature **QSig** on the hash of message M as indicated in Figure 12. Note that Sig_{SP} includes the nonce N sent by the signing facilitator in Step 3 in Figure 12 of implying it can be considered an authentication as well.

One could also leave out the SECDSA signature all together (allowing the remote signing service to act as a conventional one) and allow relying parties retrieving it from an online registration, e.g. on basis of **QSig**. Compare Figure 13. In this way one obtains a regular signing service based on eIDAS sole control that can provide classical sole control on the demand of the relying party. The signature Sig_{SP} would also be archived at the signing service allowing for evidence in dispute handling with either the user or service provider. This registration could be part of a trust service providing long-term preservation of digital signatures, cf. [29]. Archiving the SECDSA signatures as part of this service would allow provide evidence on the actual signing by the user catering for optimal non-repudiation.

The setup indicated in Figure 13 is also beneficial in implementing Strong Customer Authentication (SCA) as required in the financial sector by the European Payment Service Directive (PSD2), cf. [19,21]. In this context the financial institution, e.g. bank, takes the role of the certificate issuer, signing facilitator and the service provider; the user is the client of the financial institution. The strong non-repudiation properties of SECDSA are useful in dispute handling on (large) financial transactions. Indeed, the archived, publicly verifiable SECDSA signatures, i.e. those of type Sig_{SP} in the above setup, can irrefutable prove ta a certain client has authorized a certain transaction. For strongest non-repudiation it is best to place the certificate issuer role at a separate trusted third party, i.e. not at the financial institution itself.

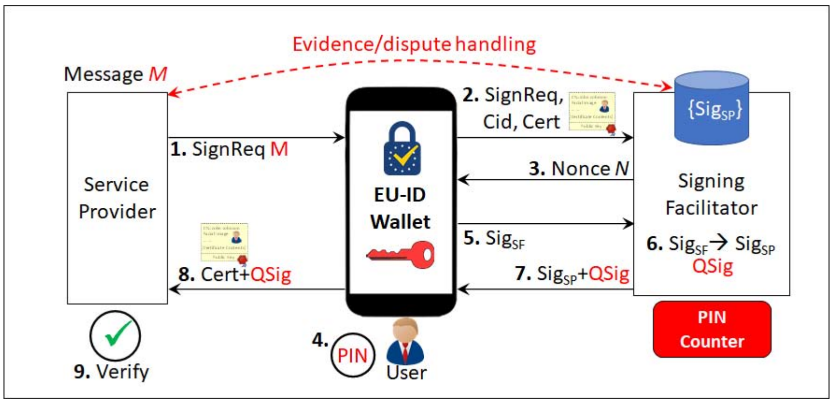


Figure 13. Qualified signing with SECDSA dispute handling

A more legally challenging approach for achieving qualified signatures is letting SECDSA signatures be certified as qualified signatures by themselves by having SECDSA included in the approved signature algorithm list [28]. Also compare Figure 14. The basis for a SECDSA signature is an ECDSA signature (generated by the EU-ID wallet) and two Schnorr signatures (one by the wallet, one by the Signing Facilitator). If we follow the alternative approach indicated at the end of Section 3.4 a SECDSA signature consists of 7 ECDSA signatures (four by the EU-ID wallet and three by the Signing Facilitator). Both of which are EU approved signature algorithms, cf. [28]. It seems there is no category in the European Commission decision [24] fitting SECDSA. Indeed, SECDSA does not fit the first paragraph of the European Commission decision as the indicated wallet does not store a PIN (as stipulated in the referenced European standard EN 419241-2 [13]). SECDSA does not fit the second paragraph either as the Signing Facilitator does not manage the SECDSA signing key. However, if the indicated wallet meets the requirements of an eIDAS assurance level High authenticator then common sense suggests that the SECDSA signatures it produces most be qualified. Indeed, these signatures form the basis of the qualified signatures in the conventional qualified remote SCD discussed above. If these can reach qualified status then a fortiori the SECDSA signatures they are based upon as a chain can never be stronger than its weakest link.

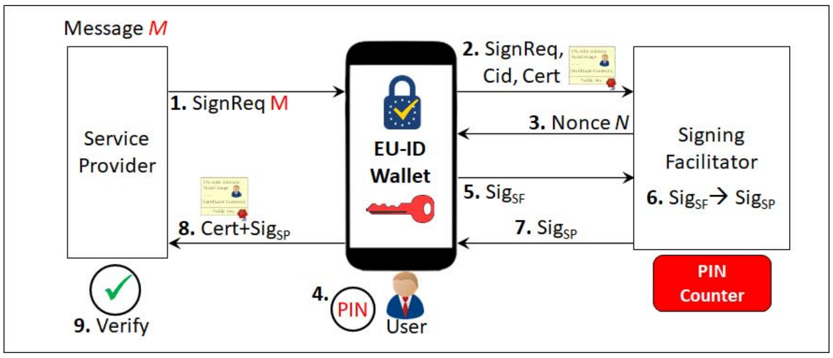


Figure 14. Qualified signing with SECDSA only

4.3 Off-line decentralized authentication and selective disclosure

In this section we sketch two approaches for letting the EU-ID wallet outlined in Section 4.1 be used in an off-line context. In the first approach one could argue we cheat a little; we do not require that the EU-ID wallet is on-line but we do require the service provider is. We thus allow the EI-ID wallet to connect to the Signing Facilitator using the service provider effectively as proxy. We have sketched this in Figure 15 below based on a variant of Protocol 2. Here ENC($\mathbf{CId}, N$) is an asymmetric encryption of the certificate identifier **CId** and nonce N under a public key of the Signing Facilitator. Without the inclusion of N an adversary could perform a replay attack, allowing him to block the EU-ID wallet of the user. To this end, the encryption scheme should be "plaintext" aware, i.e. an adversary should not be able to form the encryption without knowing both **CId** and N. For this one could use RSA with Optimal Asymmetric Encryption Padding (OAEP) [6]. Using the certificate identifier in this way allows the modified execution of Protocol 2, whereas the We assume that the data the user wants to disclose to the service provider is already available in the EU-ID wallet, e.g. by "loading" these in the wallet earlier when the wallet was on-line. As illustrated in the figure, the EU-ID wallet and service provider could for instance communicate through QR-codes.

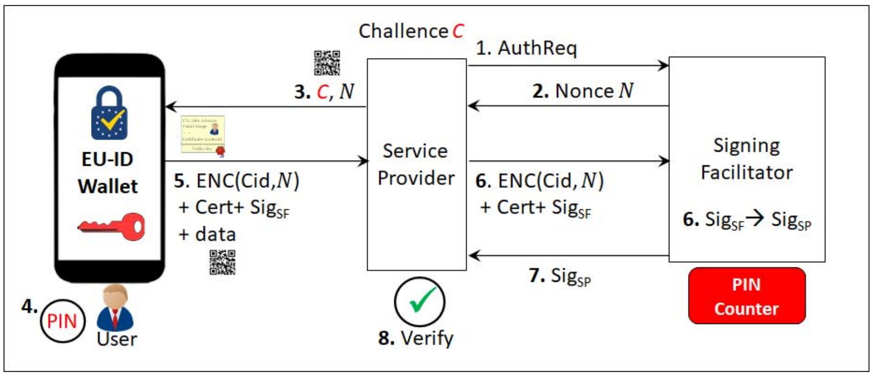


Figure 15. Off-line authentication with selective disclosure using a proxy

In the second approach we let the user/EU-ID wallet be issued short-lived certificates based on public keys based on a possession factor only, i.e. the corresponding private keys are managed in the cryptographic hardware of the device. These certificates are on-line issued to the user/EU-ID wallet based on the full SECDSA setup authenticating the user. The usage of such certificates is similar to the first approach with the only difference that the user is not required to present a knowledge factor (PIN) but only needs to consent. Of course, the usage of the short-lived certificates could be secured with a local (system) PIN or biometrically. The life-time of the short-lived certificates can be compared with the session time of a web session. This setup is indicated in Figure 16. We note that the service provider could implement further controls based on the user data disclosed, e.g., compare the disclosed facial image with the user or the disclosed user name with a regular identity document.

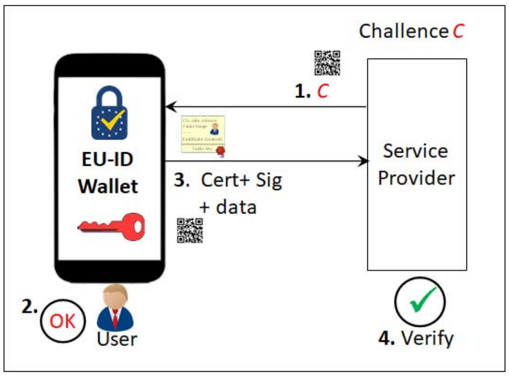


Figure 16. Off-line authentication with selective disclosure

4.4 Comparison between SECDSA and EU-ID Wallet requirements

In Table 2 below we have indicated how the requirements on the European digital identity wallet from Section 4 can be met with it SECDSA implementation outlined in Sections 4.1, 4.2 and 4.3.

#	EU-ID Wallet requirements [20,26]	SECDSA support
1.	Allows users to securely request, obtain and store personal attributes (Article 6a paragraph 3a).	Attributes are stored encrypted in the SECDSA wallet with keys managed by Signing Facilitator and access controlled by SECDSA, cf. Figure 9. See Section 4.1.
2.	Allows users to authenticate and thereby selectively disclose and combine their personal attributes (Article 6a paragraph 3a).	Attributes are attested through SECDSA certificates based on the X.509 standard [37]. Only attribute hashes are stored inside SECDSA certificates allowing selective disclosure. SECDSA certificates can also be based on a shared SECDSA public key, cf. Figures 10, 11.
3.	Allows users to legally sign documents (qualified signing) in an easily accessible way throughout Europe (Article 6a paragraph 3b).	Explained in Section 4.2. This can be based conventional remote signing (Figures 12, 13) or on conventional signing (Figure 14) by having SECDSA included in the approved signature algorithm list [28].
4.	Meets the eIDAS assurance level High on authentication (Article 6a paragraph 4c).	As argued in Section 1.5, SECDSA can form the basis for eIDAS assurance level High.
5.	Provides a “common interface” between wallet and relying parties specified on a European level (Article 6a paragraph 4a).	This can be based on an extension of the Transport Layer Security (TLS) protocol [33] and its use of X.509 client certificates.
6.	Allows relying parties to verify the authenticity of the user attributes (Article 6a paragraph 4b).	This can be based on using the X.509 standard [37] for the user certificates holding the attributes.
7.	Ensures that issuers of wallet attributes cannot receive information about the use of these attributes (Article 6a paragraph 4b).	This is the regular situation when using X.509 certificates. The revocation status service (CRL, OCSP) should be placed at an other party than the issuer which is quite common. See [28].
8.	Is usable in both on-line as off-line environments (Article 6a paragraph 3a).	Explained in Section 4.3.
9.	Issued attributes should be revocable within 24 hours. (Amendment (25))	This can be based on existing X.509 mechanisms for this (CRL, OCSP), cf. [28,37].
10.	Should be based on open standards.	SECDSA is openly specified in this document which is made public.

Table 2. Comparison between SECDSA and EU-ID Wallet requirements

5 Centralized SECDSA use cases

In this section we apply the S-APP designed in Section 3 in example SECDSA use cases in a centralized approach where a party sits between the S-APP/user and the relying party we refer to as service provider. Compare Section 1. Section 5.1 deals with authentication and Section 5.2 with signing. We assume the user has been issued a SECDSA certificate, i.e. Protocol 1 has been successfully executed.

5.1 Centralized SECDSA authentication

In this use case the service provider does not directly communicate with the user's S-APP but through an intermediary party. The SECDSA based interactions for this use case are indicated in Figure 17. Such intermediary party is usually called an authentication provider and authentication protocols as SAML [50] or OpenID Connect [5] are deployed. As in the use case of Section 4.1 the service provider requires the user to sign a challenge C as part of an authentication. If this is successful the attributes revealed in the SECDSA certificate then allow the service provider authenticate the actual user. Steps 2-7 in Figure 17 are handled by Protocol 2 with this difference that the final SECDSA signature Sig_{SP} is not sent to S-APP but to the Service Provider. We have only outlined this protocol in Figure 17. It follows from Proposition 3.4 that despite the intermediary party, this setup provides sole control and end-to-end security between the user/S-APP and the service provider.

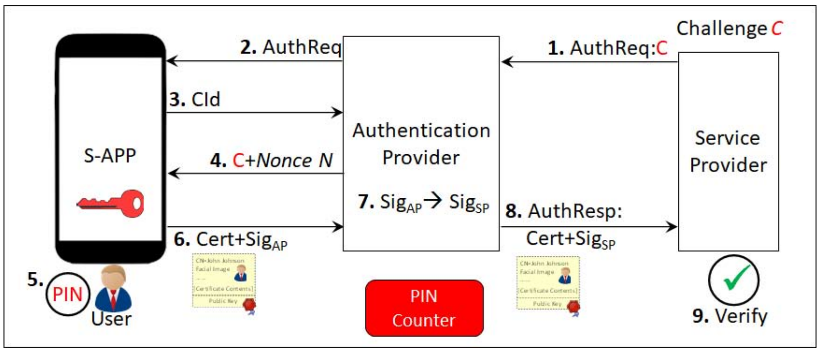


Figure 17. Centralized SECDSA authentication

In our setup we somehow place a challenge C in the authentication request of the service provider and place a SECDSA certificate and signature on C in the authentication response. Although cryptographically obvious, integration of such a "challenge-response" in authentication protocols like SAML and OpenID Connect is remarkably not common practice. However it seems this can be natively supported in OpenID connect but requires some tweaking in SAML. With respect to the latter, one could use the SAML session-id (ID) in the authentication request as this is typically generated as a pseudorandom number by the service provider. We also remark that this setup can nicely provide the "holder of key assertion" required in the NIST guidelines on strong authentication [48] for its highest assurance level (FAL3).

We finally remark that in the context of centralized authentication, one could also only let the authentication provider use SECDSA. That is, the user through his S-APP sign an authentication provider challenge and letting the SECDSA signature Sig_{SP} not be part of the authentication response to the service provider. The SECDSA signature should then be archived at the authentication provider as evidence for dispute handling with either the user or service provider.

5.2 Centralized SECDSA signing

In this use case the service provider does not directly communicate with the user's S-APP but through an intermediary party called Signing Service. As in the use case of Section 4.2 the service provider requires the user to sign a certain message M, e.g. a contract. The SECDSA based interactions for this use case are indicated in Figure 18. Here Steps 2-7 are handled by Protocol 2, with this difference that the final SECDSA signature Sig_{SP} is not sent to S-APP but to the Service Provider. We have only outlined this protocol in Figure 18. It follows from Proposition 3.4 that despite the intermediary party, this setup provides for sole control and end-to-end security between the user/S-APP and the service provider.

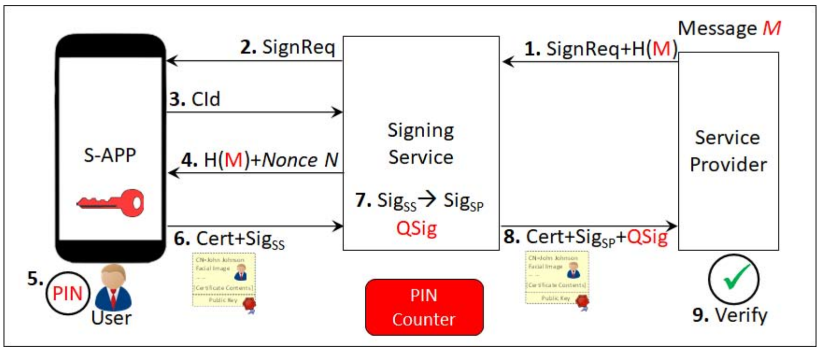


Figure 18. Centralized SECDSA signing

In Figure 18 we send both the SECDSA signature Sig_{SP} and a conventional qualifed remote signature **QSig** to most conveniently meet the formal requirements on qualified signing. Compare the SCD certification discussion in Section 4.2, where also options are discussed of only sending one of these two signatures. In case only signature **QSig** is sent, the SECDSA signature Sig_{SP} could also be archived at the signing service and online accessible for relying parties based on **QSig**. The SECDSA signature could then be used as evidence for dispute handling with either the user or service provider. This registration could be part of a trust service providing long-term preservation of digital signatures, cf. [29]. Archiving the SECDSA signatures as part of this service would allow provide evidence on the actual signing by the user catering for optimal non-repudiation.

6 References

1. https://support.apple.com/nl-nl/guide/security.
2. https://developer.apple.com/documentation/devicecheck
3. https://source.android.com/security/keystore.
4. https://developer.android.com/training/articles/security-key-attestation, https://developer.android.com/training/safetynet
5. https://openid.net.
6. M. Bellare, P. Rogaway, Optimal Asymmetric Encryption - How to Encrypt with RSA, Eurocrypt, Lecture Notes in Computer Science, Volume 950, November 1995.
7. Bundesamt für Sicherheit in der Informationstechnik (BSI), Elliptic Curve Cryptography, TR-03111, version 2.10, 2018-06-01, 2018.
8. J. Camenisch, A. Lysyanskaya, An Efficient System for Non-transferable Anonymous Credentials with Optional Anonymity Revocation, EUROCRYPT 2001. Lecture Notes in Computer Science, vol 2045. Springer, 2001.
9. Certicom Research, SEC 1: Elliptic Curve Cryptography, Version 2.0, May 21, 2009. See https://www.secg.org/sec1-v2.pdf. Modules - Part 5: Cryptographic Module for Trust Services, 2018.
10. Committee for Standardization (CEN), Protection profiles for secure signature creation device, EN 419211 (six parts), 2013-2014.
11. CEN, Protection Profiles for TSP Cryptographic Modules - Part 5: Cryptographic Module for Trust Services, EN 419221-5, 2018.
12. Committee for Standardization (CEN), Trustworthy Systems Supporting Server Signing - Part 1: General System Security Requirements, EN 419241-1, July 2018.
13. Committee for Standardization (CEN), European Trustworthy Systems Supporting Server Signing - Part 2: Protection profile for QSCD for Server Signing, EN 419241-2, February 2019.
14. Cooperation Network, Opinion No. 03/2019 on the Dutch Trust Framework for Electronic Identification, see https://ec.europa.eu/cefdigital/wiki/pages/viewpage.action?pageId=105382177
15. Cooperation Network, Opinion No. 07/2019 on on the Latvian eID scheme, https://ec.europa.eu/cefdigital/wiki/pages/viewpage.action?pageId=148898039
16. Cooperation Network, Opinion No. 8 of the Cooperation Network on the Belgian eID scheme FAS/itsme, see https://ec.europa.eu/cefdigital/wiki/pages/viewpage.action?pageId=148898042
17. European Parliament and the Council of the European Union, on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation), regulation 2016/679, 27 April 2016.
18. European Parliament and the Council of the European Union, Electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC, regulation 910/2014, 23 July 2014.
19. European Parliament and the Council of the European Union, Directive (EU) 2015/2366 (Payment Service Directive 2), 25 November 2015.
20. European Parliament and the Council of the European Union, amending Regulation (EU) No 910/2014 as regards establishing a framework for a European Digital Identity, 3 June 2021.
21. European Commission, Regulatory technical standards for strong customer authentication and common and secure open standards of communication, Commission delegated regulation (EU) 2018/389 of 27 November 2017 supplementing Directive (EU) 2015/2366.

22. European Commission, eIDAS implementing regulation 2015/1501, 8 September 2015.
23. European Commission, eIDAS implementing regulation 2015/1502, 8 September 2015.
24. European Commission, Commission Implementing Decision (EU) 2016/650 of 25 April 2016 laying down standards for the security assessment of qualified signature and seal creation devices pursuant to Articles 30(3) and 39(2) of Regulation (EU) No 910/2014, 2016.
25. European Commission, List of alternative processes notified to the Commission in accordance with Article 30.3(b) and 39.2 of the eIDAS Regulation (EU) No 910/2014, version of 27/03/2020.
26. European Commission, on a common Union Toolbox for a coordinated approach towards a European Digital Identity Framework, C(2021) 3968 final, 03/06/2021.
27. Guidance for the application of the levels of assurance which support the eIDAS Regulation. See https://ec.europa.eu.
28. European Telecommunications Standards Institute (ETSI), Electronic Signatures and Infrastructures (ESI); Cryptographic Suites, TS 119312, V1.2.1, 2017-05.
29. ETSI, Policy and security requirements for trust service providers providing long-term preservation of digital signatures or general data using digital signature techniques, TS119511, V1.1.1, 2019-06.
30. ETSI, Policy and security requirements for Trust Service Providers issuing certificates, ETSI EN 319 411 (two parts). See http://www.etsi.org.
31. A. Fiat, A. Shamir, How To Prove Yourself: Practical Solutions to Identification and Signature Problems, Crypto '86, Lecture Notes in Computer Science, Volume 263, Springer, 1986.
32. D. Hankerson, A. Menezes and S. Vanstone, Guide to Elliptic Curve Cryptography, Springer, 2004.
33. Internet Engineering Task Force, The Transport Layer Security (TLS) Protocol, RFC 5426, 2008.
34. International Organization for Standardization (ISO), Information technology - Security techniques - Digital signatures with appendix - Part 3: Discrete logarithm based mechanisms, ISO/IEC 14888-3, fourth edition, 2018.
35. ISO, Personal identification - ISO-compliant driving licence - Part 5: Mobile driving licence (mDL) application, ISO 18013-5, February 2020, draft.
36. ISO, Information technology - Security techniques - Methodology for IT security evaluation, ISO/IEC 18045, version 2014-01-15.
37. International Telecommunication Union (ITU), X.509, Public-key and attribute certificate frameworks. See https://www.itu.int.
38. ITU-T, Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), X.690, 08/2015.
39. J. Katz, Y. Lindell, Introduction to Modern Cryptography, CRC PRESS, 2008.
40. H. Leitold, D. Konrad Qualified Remote Signatures – Solutions, its Certification, and Use, Proceedings of 29th SmartCard Workshop, 20-21 February 2019, Darmstadt, Germany.
41. Y. Lindell, Fast Secure Two-Party ECDSA Signing, Crypto 2017, Lecture Notes in Computer Science, vol 10402. Springer, 2017.
42. National Institute for Standards and Technology (NIST), Advanced Encryption Standard (AES), FIPS PUB 197, November 2001.
43. NIST, Secure Hash Standard (SHS), FIPS PUB 180-4, August 2015.

44. NIST, Digital signature standard, FIPS PUB 186-4. July 2013 .
45. NIST, Recommendation for Block Cipher Modes of Operation, Special Publication 800-38A, December 2001.
46. NIST, Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, Special Publication 800-38D, November, 2007.
47. NIST, The Keyed-Hash Message Authentication Code (HMAC), FIPS PUB 198-1, July 2008.
48. NIST, Digital Authentication Guidelines, SP 800-63, June 2017.
49. NIST, Recommendation for Key Derivation Using Pseudorandom Functions, SP 800-108, October 2009.
50. OASIS, Security Assertion Markup Language (SAML), 2005.
51. OASIS, PKCS #11 Cryptographic Token Interface Base Specification, version 2.4, 14 April 2015.
52. C. P. Schnorr, Efficient signature generation for smart cards. Journal of Cryptology, 4(3):239-252, 1991.
53. D.G. Stinson, Cryptography: theory and practice, CRC press, 1995.

A Examples of PIN-binder constructions

In this appendix we present PIN-binder constructions deriving PIN-keys from the user PIN and a cryptographic key. For security assurance and ease of implementation we use standardized algorithms: the recommendations [49] of the US National Institute of Standards and Technology (NIST) combined with the recommendations of the German Bundesamt für Sicherheit in der Informationstechnik (BSI) [7]. As a building block we first define a key derivation function deriving a key $\mathcal{K}(K, D, L)$ in the form of a byte array of size L bits based on master key K and derivation byte array D. For this we deploy the NIST key derivation function construction from [49, Section 5.1] in Counter Mode whereby using the HMAC function [47] based on a secure hash function $\mathcal{H}(.)$ as pseudorandom function. This hash function can for instance be the SHA-256 hash function [43] as this is commonly supported in platforms. We further make the following choices in the NIST construction:

- the Context string is equal to D,
- the Label is the empty string, i.e. we do not use a Label string,
- the zero byte separates the Label from the output size L in binary representation, i.e. $[L]_2$, in the construction,
- the length r of the binary representation of the counter is chosen as 8.

We can now specify the first PIN-binder construction.

Algorithm 9 PIN-binder based on HMAC
Input: user PIN P, PIN-binder key: HMAC key K in SCE
Output: PIN-key σ

```
1: Compute $\mathcal{K}(K, P, 8 * |q| + 64)$ and convert to integer $x$        // in SCE
2: Return $\sigma = 1 + (x \bmod (q - 1))$
```

We generate 64 more bits in Line 1 than the length $8{\cdot}|q|$ of q in bits. If we would generate precisely $8{\cdot}|q|$ bits then a small bias would arise. By generating 64 more bits we still do not produce a formally uniform distribution modulo $q - 1$ but the deviation is believed not to be exploitable following [7]. By the addition of 1 in Line 2 we ensure that the output is always a non-zero element in $\mathbb{F}_q$.

It is convenient to base a PIN-binder on the textbook RSA decryption capabilities of the SCE. See [53]. In the RSA cryptosystem the public key consists of the product n of two large prime number p, q and the public exponent e usually taken as $2^{16} + 1 = 65537$, The private key consists of the integer $d = e^{-1} \bmod (p-1)(q-1)$. Textbook RSA encryption, respectively decryption, consists of the functions $x \to x^e \bmod n$, respectively $x \to x^d \bmod n$. Textbook RSA is not normally used in practice as it is not resistant against so-called chosen ciphertext attacks. To remedy against this kind of attacks redundancy ("padding") needs to be added to plaintext prior to encryption; that padding is then checked again as part of decryption.

Most cryptographic libraries/APIs for RSA allow the configuration of a "PADDING_NONE" mode facilitating textbook RSA. In this way flexible support for various kinds of other padding is provided to developers. The Android

SCE (hardware backed keystore) also allows for this. We can then construct a PIN-binder as sketched below. This starts with the generation of an RSA public-private key pair in the SCE of appropriate size, e.g. a modulus n with bit length $|n| = 2048$ bits, i.e. 256 byte. We assume that $|n| > |q| + 64$ which is naturally the case. We do not assume that the HMAC key K in Algorithm 10 necessarily resides in the SCE; it can also be stored in a keystore in the local storage of S-APP.

Algorithm 10 PIN-binder based on textbook RSA
Input: user PIN P, PIN-binder key: HMAC key K, RSA private key d in SCE
Output: PIN-key σ

```
1: Compute $\mathcal{K}(K, P, 8 * |n|)$ and convert to integer $x_0$
2: Let $x_1 = x \bmod n$
3: Let $x_2 = x_1^d \bmod n$                                    // in SCE
4: Return $\sigma = 1 + (x_2 \bmod (q - 1))$
```

Note that we can iterate Line 3 of Algorithm 10, i.e. let x_2 be subject to a further RSA textbook decryption and so on. As RSA decryption is a time consuming process this allows control of PIN-key computing time. This can be beneficial in mitigating PIN brute-force attacks. See Section 3.4.

As Android based devices provide for HMAC and textbook RSA in their SCE they allow for various PIN-binding mechanisms. However, Apple based devices, do not provide for either of those algorithms in their SCE. Apart from ECDSA signatures, Apple based devices typically only support the Elliptic Curve Integrated Encryption Scheme (ECIES) see [32]. ECIES yields public key based encryption allowing any party encrypting data for a user with his public key. ECIES is based on authenticated encryption meaning that ECIES decryption will only return plaintext if the ciphertext is authenticated (see below). This means that feeding the ECIES decryption function in the SCE with arbitrary ciphertext will typically result in errors and not in useful data. Despite this we show one can still base a PIN-binder on ECIES decryption. The construction presented focusses on the Apple implementation of ECIES as this is most relevant. We also outline how this construction can be extended to other ECIES implementations.

ECIES is also based on elliptic curve group $\mathbb{G} = (\langle G \rangle, +)$. A user has a private key $d \in \mathbb{F}_q^*$ and a public key $D = d{\cdot}G$, i.e. similar to ECDSA. In our context the ECIES private key d resides in the SCE, i.e. the Secure Enclave of Apple. ECIES uses the Diffie-Hellman key exchange protocol for sending a symmetric key to the recipient user which is then used in a symmetric encryption algorithm. Apple's ECIES implementation is based on three modes of the Advanced Encryption Standard (AES) [42]. We first describe the working of these and then Apple's ECIES implementation in Algorithms 11 and 12.

The first mode used is the basic AES mode, also known as Electronic Code Book (ECB). It allows encrypting a 16 byte plaintext block M using a key K resulting in a ciphertext block $C = \mathcal{E}_{\text{ECB}}(K, M)$ that is also of size 16 byte. The key K can be chosen either 128, 192 or 256 bits in length. Similarly one can decrypt the ciphertext C using key K which we denote by $M = \mathcal{D}_{\text{ECB}}(K, C)$. The second mode used is so-called Galois/Counter Mode (AES-GCM) mode [46], which is actually based on two other AES modes. The input of the AES-GCM encryption algorithm consists of an AES key K, an initialisation vector (byte array of certain length) IV, plaintext data D and optional authenticated data A. As our constructions do not use authenticated data we leave A out of the further description.

The AES-GCM encryption algorithm first derives a so-called hash-key H from K by AES encrypting the zero-block 0^{16} (array of 16 zero bytes) with K. That is i.e. $H = \mathcal{E}_{\text{AES}}(K, 0^{16})$. Next, the AES-GCM algorithm first encrypts the data D deploying AES in so-called Counter mode (AES-CTR, cf. [45]) using key K and initialisation vector IV resulting in a ciphertext C of the same length as M. We denote this by $C = \mathcal{E}_{\text{CTR}}(IV, K, M)$ and the decryption by $M = \mathcal{D}_{\text{CTR}}(IV, K, C)$. Next, AES-GCM runs the encrypted data through an AES based Message Authentication Code function called GMAC using the hash-key H. This results in an authentication tag T. We denote this by $T = \text{GMAC}_{\text{AES}}(H, C)$. The output of the AES-GCM encryption algorithm consists of the encrypted data C and the authentication tag T. The input of the AES-GCM decryption algorithm consists of a key K, an initialisation vector IV, the ciphertext data C and authentication tag T. The AES-GCM decryption algorithm first derives the hash-key H from K by AES encrypting the zero-block with K. Then it first checks the authenticity of the ciphertext C by running this through GMAC using the hash-key H. If this does not result in the authentication tag T, then the decryption returns an error. Otherwise the AES-GCM decryption algorithm decrypts C based on AES-CTR using the initialisation vector IV and K and returns the result.

In Algorithms 11 and 12 we further formalize the working of AES-GCM based ECIES as used in Apple devices.[7] We let $H[x, y]$ denote the byte range x, y (including boundaries) of a byte array H where byte 0 refers to the most significant byte. We also let SHA256(.) represent the SHA-256 secure hash function, cf. [43].

[7] The details of the Apple ECIES specification is hard to find, we base our description on `https://darthnull.org/security/2018/05/31/secure-enclave-ecies/`.

Algorithm 11 ECIES encryption based on AES-GCM
Input: Message M, recipient public key D.
Output: Ephemeral key R, AES-GCM ciphertext (C, T)

```
1: Select random $k \in \{1, ..., q-1\}$.
2: Compute $R = k{\cdot}G$ and $Z = k{\cdot}D$                    // ephemeral key $R$
3: Convert $Z$ to byte array $\bar{Z}$ and compute $H = \mathtt{SHA256}(\bar{Z})$
4: Choose Initialisation Vector $IV = H[0, 15]$ and AES-GCM key $K = H[16, 31]$
5: Compute hash-key $H = \mathcal{E}_{\text{AES}}(K, 0^{16})$              // AES-GCM
6: Compute $C = \mathcal{E}_{\text{CTR}}(IV, K, M)$                  // AES-GCM
7: Compute $T = \mathtt{GMAC}(H, C)$                        // AES-GCM
8: Return $R, (C, T)$
```

Algorithm 12 ECIES decryption based on AES-GCM
Input: Ephemeral key R, AES-GCM encrypted message (C, T), recipient private key d.
Output: Message M or rejection of the encrypted message

```
1: Compute $Z = d{\cdot}R$.
2: Convert $Z$ to byte array $\bar{Z}$ and compute $H = \mathtt{SHA256}(\bar{Z})$
3: Choose Initialisation Vector $IV = H[0, 15]$ and AES-GCM key $K = H[16, 31]$
4: Compute hash-key $H = \mathcal{E}_{\text{AES}}(K, 0^{16})$
5: Compute $T' = \mathtt{GMAC}(H, C)$                       // AES-GCM
6: If $T' \neq T$ reject message                       // AES-GCM
7: Compute $M = \mathcal{D}_{\text{CTR}}(IV, K, C)$                  // AES-GCM
8: Return $M$
```

The idea behind basing a PIN-binder on ECIES is the observation that with only possession of the ephemeral key R and the corresponding hash-key H one can generate authentication tag T on any byte array P. One can then successively feed R, (P, T) to Algorithm 12 resulting in output, namely $\mathcal{D}_{\text{CTR}}(IV, K, P)$. We further formalize this in two algorithms performed by S-APP: generating the PIN-binder key and generating the PIN-key based on that.

Algorithm 13 Generation of an ECIES-AES PIN-binder key in SCE

```
1: S-APP requests SCE to generate ECIES private key $d \in_R \mathbb{F}_q^*$   // $d$ in SCE
2: S-APP exports public key $D = d{\cdot}G$ from SCE
3: S-APP selects random $k \in \{1, ..., q-1\}$.
4: S-APP computes $Z = k{\cdot}D$
5: S-APP converts $Z$ to byte array $\bar{Z}$ and compute $H = \mathtt{SHA\text{-}256}(\bar{Z})$
6: S-APP computes AES-GCM key $K = H[16, 31]$
7: S-APP computes hash-key $H = \mathcal{E}_{\text{AES}}(K, 0^{16})$           // AES-GCM
8: S-APP locally stores $R$ and $H$ and deletes $Z$ and $K$
```

Algorithm 14 PIN-binder based on ECIES-AES
Input: user PIN P, PIN-binder key: HMAC key K, R, H, ECIES private key d (in SCE)
Output: PIN-key σ

```
1: Compute $P = \mathcal{K}(K, P, 8 * |q| + 64)$
2: Compute authentication tag $T = \texttt{GMAC}(H, P)$
3: Feed $R$, $(P, T)$ to Algorithm 12 resulting in $P'$          // ECIES decrypt
4: Convert $P'$ to integer $x$                    // same byte length $8 * |q| + 64$
5: Return $\sigma = 1 + (x \bmod (q - 1))$
```

As in Algorithm 10 we can iterate Lines 2-3, i.e. reusing P' of Line 3 in Line 2. As ECIES decryption is a time consuming process this allows control of PIN-key computing time. This can be beneficial in mitigating PIN brute-force attacks. See Section 3.4. As part of our PIN-binder construction R, $H = \mathcal{E}_{\text{AES}}(K, 0^{16})$ are locally stored by S-APP. It a basic requirement from ECIES that K can only be computed with access to the private key d. This accomplished by the hardness of the Diffie-Hellman problem, cf. Section 2.1. It is also a basic requirement of a blockcipher that K cannot be derived from H. In other words, locally storing R and H does not introduce a security vulnerability.

In Algorithms 11, 12 above we have elaborated on Apple's ECIES implementation as this is most relevant for this paper. In the general ECIES setup, one derives from Z and R an encryption key K_{ENC} and an(H)MAC key from K_{ENC}. As part of ECIES encryption one then applies key K_{ENC} to encrypt the message and key K_{MAC} to compute an authentication tag on the result. During ECIES decryption one first validates the authentication tag and when successful commence with decryption. The ECIES PIN-binder method discussed above clearly extends to the general ECIES setup by storing R and K_{MAC} in S-APP.