



FIDO ECDA A Algorithm

FIDO Alliance Proposed Standard 27 September 2017

This version:

<https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-ecdaa-algorithm-v2.0-ps-20170927.html>

Editor:

[Rolf Lindemann, Nok Nok Labs, Inc.](#)

Contributors:

[Jan Camenisch, IBM](#)

[Manu Drijvers, IBM](#)

[Alec Edgington, Trustonic](#)

[Anja Lehmann, IBM](#)

[Rainer Urian, Infineon](#)

The English version of this specification is the only normative version. Non-normative [translations](#) may also be available.

Copyright © 2013-2017 [FIDO Alliance](#) All Rights Reserved.

Abstract

The FIDO Basic Attestation scheme uses attestation "group" keys shared across a set of authenticators with identical characteristics in order to preserve privacy by avoiding the introduction of global correlation handles. If such an attestation key is extracted from one single authenticator, it is possible to create a "fake" authenticator using the same key and hence indistinguishable from the original authenticators by the relying party. Removing trust for registering new authenticators with the related key would affect the entire set of authenticators sharing the same "group" key. Depending on the number of authenticators, this risk might be unacceptable high.

This is especially relevant when the attestation key is primarily protected against malware attacks as opposed to targeted physical attacks.

An alternative approach to "group" keys is the use of individual keys combined with a Privacy-CA [[TPMv1-2-Part1](#)]. Translated to FIDO, this approach would require one Privacy-CA interaction for each Uauth key. This means relatively high load and high availability requirements for the Privacy-CA. Additionally the Privacy-CA aggregates sensitive information (i.e. knowing the relying parties the user interacts with). This might make the Privacy-CA an interesting attack target.

Another alternative is the Direct Anonymous Attestation [[BriCamChe2004-DAA](#)]. Direct Anonymous Attestation is a cryptographic scheme combining privacy with security. It uses the authenticator specific secret once to communicate with a single DAA Issuer and uses the resulting DAA credential in the DAA-Sign protocol with each relying party. The DAA scheme has been adopted by the Trusted Computing Group for TPM v1.2 [[TPMv1-2-Part1](#)].

In this document, we specify the use of an improved DAA scheme based on elliptic curves and bilinear pairings largely compatible with [[CheLi2013-ECDA A](#)] called ECDA A. This scheme provides significantly improved performance compared with the original DAA and basic building blocks for its implementation

are part of the TPMv2 specification [[TPMv2-Part1](#)].

Our improvements over [[CheLi2013-ECDAA](#)] mainly consist of security fixes (see [[ANZ-2013](#)] and [[XYZF-2014](#)]) when splitting the sign operation into two parts.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current FIDO Alliance publications and the latest revision of this technical report can be found in the [FIDO Alliance specifications index](#) at <https://www.fidoalliance.org/specifications/>.

This document was published by the [FIDO Alliance](#) as a Proposed Standard. If you wish to make comments regarding this document, please [Contact Us](#). All comments are welcome.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

This document has been reviewed by FIDO Alliance Members and is endorsed as a Proposed Standard. It is a stable document and may be used as reference material or cited from another document. FIDO Alliance's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment.

Table of Contents

- 1. [Notation](#)
 - 1.1 [Conformance](#)
- 2. [Overview](#)
 - 2.1 [Scope](#)
 - 2.2 [Architecture Overview](#)
- 3. [FIDO ECDAA Attestation](#)
 - 3.1 [Object Encodings](#)
 - 3.1.1 [Encoding `BigInteger` values as byte strings \(BigIntegerToB\)](#)
 - 3.1.2 [Encoding `ECPoint` values as byte strings \(ECPointToB\)](#)
 - 3.1.3 [Encoding `ECPoint2` values as byte strings \(ECPoint2ToB\)](#)
 - 3.2 [Global ECDAA System Parameters](#)
 - 3.3 [Issuer Specific ECDAA Parameters](#)
 - 3.4 [ECDAA-Join](#)
 - 3.4.1 [ECDAA-Join Algorithm](#)
 - 3.4.2 [ECDAA-Join Split between Authenticator and ASM](#)
 - 3.4.3 [ECDAA-Join Split between TPM and ASM](#)
 - 3.5 [ECDAA-Sign](#)
 - 3.5.1 [ECDAA-Sign Algorithm](#)
 - 3.5.2 [ECDAA-Sign Split between Authenticator and ASM](#)
 - 3.5.3 [ECDAA-Sign Split between TPM and ASM](#)
 - 3.6 [ECDAA-Verify Operation](#)
- 4. [FIDO ECDAA Object Formats and Algorithm Details](#)
 - 4.1 [Supported Curves for ECDAA](#)
 - 4.2 [ECDAA Algorithm Names](#)
 - 4.3 [ecdaaSignature object](#)
- 5. [Considerations](#)
 - 5.1 [Algorithms and Key Sizes](#)
 - 5.2 [Indicating the Authenticator Model](#)
 - 5.3 [Revocation](#)

- 5.4 [Pairing Algorithm](#)
- 5.5 [Performance](#)
- 5.6 [Binary Concatentation](#)
- 5.7 [IANA Considerations](#)
- A. [References](#)
 - A.1 [Normative references](#)
 - A.2 [Informative references](#)

1. Notation

Type names, attribute names and element names are written as `code`.

String literals are enclosed in “”, e.g. “ED256”.

In formulas we use “|” to denote byte wise concatenation operations.

$X = P^x$ denotes scalar multiplication (with scalar x) of a (elliptic) curve point P.

RAND(x) denotes generation of a random number between 0 and x-1.

RAND(G) denotes generation of a random number belonging to Group G.

Specific terminology used in this document is defined in [[FIDOGlossary](#)].

The type `BigNumber` denotes an arbitrary length integer value.

The type `ECPoint` denotes an elliptic curve point with its affine coordinates x and y.

The type `ECPoint2` denotes a point on the sextic twist of a BN elliptic curve over $F(q^2)$. The ECPoint2 has two affine coordinates each having two components of type `BigNumber`

1.1 Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words **must**, **must not**, **required**, **should**, **should not**, **recommended**, **may**, and **optional** in this specification are to be interpreted as described in [[RFC2119](#)].

2. Overview

This section is non-normative.

FIDO uses the concept of attestation to provide a cryptographic proof of the **authenticator** [[FIDOGlossary](#)] model to the relying party. When the authenticator is registered to the relying party (RP), it generates a new authentication key pair and includes the public key in the attestation message (also known as key registration data object, **KRD**). When using the ECDAAs algorithm, the **KRD** object is signed using [3.5 ECDAAs-Sign](#).

For privacy reasons, the authentication key pair is dedicated to one RP (to an application identifier **AppID** [[FIDOGlossary](#)] to be more specific). Consequently the attestation method needs to provide the same level of unlinkability. This is the reason why the FIDO ECDAAs Algorithm doesn't use a basename (bsn) often found in other direct anonymous attestation algorithms, e.g. [[BriCamChe2004-DAA](#)] or [[BFGSW-2011](#)].

The authenticator encapsulates all user verification operations and cryptographic functions. An authenticator specific module (**ASM**) [[FIDOGlossary](#)] is used to provide a standardized communication interface for authenticators. The authenticator might be implemented in separate hardware or trusted execution environments. The ASM is assumed to run in the normal operating system (e.g. Android, Windows, ...).

2.1 Scope

This document describes the FIDO ECDAAs attestation algorithm in detail.

2.2 Architecture Overview

ECDAAs attestation defines [global system parameters](#) and [issuer specific parameters](#). Both parameter sets need to be installed on the host, in the [authenticator](#) and in the FIDO Server. The ECDAAs method consists of two steps:

- [ECDAAs-Join](#) to be performed *before* the first FIDO Registration
 - $n = \text{GetNonceFromECDAAsIssuer}()$
 - $(Q, c1, s1) = \text{EcdaasJoin1}(X, Y, n)$
 - $(A, B, C, D, s2, c2) = \text{EcdaasIssuerJoin}(Q, c1, s1)$
 - $\text{EcdaasJoin2}(A, B, C, D, c2, s2)$ // store cre=(A, B, C, D)
- and the pair of [ECDAAs-Sign](#) performed by the authenticator and [ECDAAs-Verify](#) performed by the FIDO Server as part of the FIDO Registration.
 - Client: Attestation = (signature, [KRD](#)) = $\text{EcdaasSign}(\text{AppID})$
 - Server: success= $\text{EcdaasVerify}(\text{signature}, \text{KRD}, \text{AppID})$

The technical implementation details of the ECDAAs-Join step are out-of-scope for FIDO. In this document we normatively specify the general algorithm to the extent required for interoperability and we outline examples of some possible implementations for this step.

The ECDAAs-Sign and ECDAAs-Verify steps and the encoding of the related ECDAAs Signature are normatively specified in this document. The generation and encoding of the [KRD](#) object is defined in other FIDO specifications.

The algorithm and terminology are inspired by [BFGSW-2011](#). The algorithm was modified in order to fix security weaknesses (e.g. as mentioned by [ANZ-2013](#) and [XYZF-2014](#)). Our algorithm proposes an improved task split for the sign operation while still being compatible to TPMv2 (without fixing the TPMv2 weaknesses in such case).

3. FIDO ECDAAs Attestation

This section is normative.

3.1 Object Encodings

We need to convert [BigInteger](#) and [ECPoint](#) objects to byte strings using the following encoding functions:

3.1.1 Encoding [BigInteger](#) values as byte strings ([BigIntegerToB](#))

We use the I2OSP algorithm as defined in [RFC3447](#) for converting big numbers to byte arrays. The bytes from the big endian encoded (non-negative) number n will be copied right-aligned into the buffer area b . The unused bytes will be set to 0. Negative values will not occur due to the construction of the algorithms.

EXAMPLE 1: Converting [BigInteger](#) n to byte string b

```
b0 b1 b2 b3 b4 b5 b6 b7
0 0 n0 n1 n2 n3 n4 n5
```

The algorithm implemented in Java looks like this:

EXAMPLE 2: Algorithm for converting [BigInteger](#) to byte strings

```
ByteArray BigIntegerToB(
    BigInteger inVal, // IN: number to convert
    int size         // IN: size of the output.
)
{
    ByteArray buffer = new ByteArray(size);
    int oversize = size - inVal.length;
    if (oversize < 0)
        return null;
    for (int i=oversize; i > 0; i--)
        buffer[i] = 0;
    ByteCopy( inVal.bytes, &buffer[oversize], inVal.length);
    return buffer;
}
```

3.1.2 Encoding `ECPoint` values as byte strings (`ECPointToB`)

We use the ANSI X9.62 Point-to-Octet-String [[ECDSA-ANSI](#)] conversion using the expanded format, i.e. the format where the compression byte (i.e. 0x04 for expanded) is followed by the encoding of the affine x coordinate, followed by the encoding of the affine y coordinate.

EXAMPLE 3: Converting `ECPoint P` to byte string

```
(x, y) = ECPointGetAffineCoordinates(P)
len = G1.byteLength
byte string = 0x04 | BigIntegerToB(x, len) | BigIntegerToB(y, len)
```

3.1.3 Encoding `ECPoint2` values as byte strings (`ECPoint2ToB`)

The type `ECPoint2` denotes a point on the sextic twist of a BN elliptic curve over $F(q^2)$, see section [4.1 Supported Curves for ECDAAs](#). Each `ECPoint2` is represented by a pair (a, b) of elements of $F(q)$.

The group zero element is always encoded (using the encoding rules as described below) as a an element having all components set to zero (i.e. $cx.a=0, cx.b=0, cy.a=0, cy.b=0$).

We always assume normalized (non-zero) `ECPoint2` values (i.e. $cz = 1$) before encoding them. Non-zero values are encoded using the expanded format (i.e. 0x04 for expanded) followed by the cx followed by the cy value. This leads to the concatenation of 0x04 followed by the first element ($cx.a$) and second element ($cx.b$) of the pair of cx followed by the first element ($cy.a$) and second element ($cy.b$) of the pair of cy . All individual numbers are padded to the same length (i.e. the maximum byte length of all relevant 4 numbers).

EXAMPLE 4: Converting `ECPoint2 P2` to byte string

```
(cx, cy) = ECPointGetAffineCoordinates(P2)
len = G2.byteLength
byte string = 0x04 | BigIntegerToB(cx.a, len) | BigIntegerToB(cx.b, len)
               | BigIntegerToB(cy.a, len) | BigIntegerToB(cy.b, len)
```

3.2 Global ECDAAs System Parameters

1. Groups G^1, G^2 and G^T , of sufficiently large prime order p
2. Two generators P^1 and P^2 , such that $G^1 = \langle P^1 \rangle$ and $G^2 = \langle P^2 \rangle$
3. A bilinear pairing $e : G^1 \times G^2 \rightarrow G^T$. We propose the use of "ate" pairing (see [[BarNae-2006](#)]). For example source code on this topic, see [BNPairings](#).
4. Hash function H with $H : \{0, 1\}^* \rightarrow Z_p$.
5. (G^1, P^1, p, H) are installed in all authenticators implementing FIDO ECDAAs attestation.

Definition of G^1, G^2, G^T , Pairings and hash function H

See section [4.1 Supported Curves for ECDAAs](#).

3.3 Issuer Specific ECDAAs Parameters

Issuer Parameters `parI`

1. Randomly generated issuer private key $isk = (x, y)$ with $[x, y = RAND(p)]$.
2. ECDAAs-Issuer public key (X, Y) , with $X = P_2^x$ and $Y = P_2^y$.
3. A proof that the issuer key was correctly computed
 1. BigInteger $r_x = RAND(p)$
 2. BigInteger $r_y = RAND(p)$

3. ECPoint2 $U_x = P_2^{r_x}$
4. ECPoint2 $U_y = P_2^{r_y}$
5. BigInteger $c = H(U_x|U_y|P_2|X|Y)$
6. BigInteger $s_x = r_x + c \cdot x \pmod{p}$
7. BigInteger $s_y = r_y + c \cdot y \pmod{p}$

$$4. \text{ ipk} = X, Y, c, s_x, s_y$$

Whenever a party uses ipk for the first time, it must first verify that it was correctly generated:

$$H(P_2^{s_x} \cdot X^{-c} | P_2^{s_y} \cdot Y^{-c} | P_2 | X | Y) \stackrel{?}{=} c$$

NOTE

$$P_2^{s_x} \cdot X^{-c} = P_2^{r_x + cx} \cdot P_2^{-cx} = P_2^{r_x} = U_x$$

$$P_2^{s_y} \cdot Y^{-c} = P_2^{r_y + cy} \cdot P_2^{-cy} = P_2^{r_y} = U_y$$

The ECDAA-Issuer public key ipk **must** be dedicated to a single authenticator model.

We use the element c of ipk as an identifier for the ECDAA-Issuer public key (called **ECDAA-Issuer public key identifier**).

3.4 ECDAA-Join

NOTE

One ECDAA-Join operation is required once in the lifetime of an authenticator prior to the first registration of a credential.

In order to use ECDAA, the authenticator must first receive ECDAA credentials from an ECDAA-Issuer. This is done by the ECDAA-Join operation. This operation needs to be performed a single time (before the first credential registration can take place). After the ECDAA-Join, the authenticator will use the ECDAA-Sign operation as part of each FIDO Registration. The ECDAA-Issuer is not involved in this step. ECDAA plays no role in FIDO Authentication / Transaction Confirmation operations.

In order to use ECDAA, (at least) one ECDAA-Issuer is needed. The approach specified in this document easily scales to multiple ECDAA-Issuers, e.g. one per authenticator vendor. FIDO lets the authenticator vendor choose any ECDAA-Issuer (similar to his current freedom for selecting any PKI infrastructure/service provider to issuing attestation certificates required for FIDO Basic Attestation).

- All ECDAA-Join operations (of the related authenticators) are performed with one of the ECDAA-Issuer entities.
- Each ECDAA-Issuer has a set of public parameters, i.e. ECDAA public key material. The related Attestation Trust Anchor is contained in the metadata of each authenticator model identified by its AAGUID.

There are two different implementation options relevant for the authenticator vendors (the authenticator vendor can freely choose them):

1. In-Factory ECDAA-Join
2. Remote ECDAA-Join and

In the first case, physical proximity is used to locally establish the trust between the ECDAA-Issuer and the authenticator (e.g. using a key provisioning station in a production line). There is no requirement for the ECDAA-Issuer to operate an online web service.

In the second case, some credential is required to remotely establish the trust between the ECDAAs-Issuer and the authenticator. As this operation is performed once and only with a single ECDAAs-Issuer, privacy is preserved and an authenticator specific credential can and should be used.

Not all ECDAAs authenticators might be able to add their authenticator model IDs (e.g. AAGUID) to the registration assertion (e.g. TPMs). In all cases, the ECDAAs-Issuer will be able to derive the exact the authenticator model from either the credential or the physically proximate authenticator. So the ECDAAs-Issuer root key **must** be dedicated to a single authenticator model.

3.4.1 ECDAAs-Join Algorithm

This section is normative.

NOTE

If this join is not in-factory, the value Q must be authenticated by the authenticator. Upon receiving this value, the issuer must verify that this authenticator did not join before.

1. The authenticator asks the issuer for a nonce.
2. The issuer chooses a nonce BigInteger $n = RAND(p)$ and sends n via the ASM to the authenticator.
3. The authenticator chooses and stores the ECDAAs private key BigInteger $sk = RAND(p)$
4. The authenticator computes its ECDAAs public key ECPoint $Q = P_1^{sk}$
5. The authenticator proves knowledge of sk as follows
 1. BigInteger $r^1 = RAND(p)$
 2. ECPoint $U^1 = P_1^{r^1}$
 3. BigInteger $c^1 = H(U^1|P_1|Q|n)$
 4. BigInteger $s^1 = r^1 + c^1 \cdot sk$
6. The authenticator sends Q, c^1, s^1 via the ASM to the issuer
7. The issuer verifies that the authenticator is "authentic" and that Q was indeed generated by the authenticator. In the case of an in-factory Join, this might be trivial; in the case of a remote Join this typically requires the use of other cryptographic methods. Since ECDAAs-Join is a one-time operation, unlinkability is not a concern for that.
8. The issuer verifies that $Q \in G^1$ and verifies $H(P_1^{s^1} \cdot Q^{-c^1} | P_1 | Q | n) \stackrel{?}{=} c^1$ (check proof-of-possession of private key).

NOTE

$$P_1^{s^1} \cdot Q^{-c^1} = P_1^{r^1 + c^1 sk} \cdot Q^{-c^1} = P_1^{r^1 + c^1 sk} \cdot P_1^{-c^1 sk} = P_1^{r^1} = U^1$$

9. The issuer creates credential (A, B, C, D) as follows
 1. BigInteger $l^J = RAND(p)$
 2. ECPoint $A = P_1^{l^J}$
 3. ECPoint $B = A^y$
 4. ECPoint $C = A^x \cdot Q^{xy l^J}$
 5. ECPoint $D = Q^{l^J y}$
10. The issuer proves that it computed this credential correctly:

1. BigInteger $r^2 = RAND(p)$
2. ECPoint $U^2 = P_1^{r^2}$
3. ECPoint $V^2 = Q^{r^2}$
4. BigInteger $c^2 = H(U^2|V^2|P_1|B|Q|D)$
5. BigInteger $s^2 = r^2 + c^2 \cdot l^J \cdot y$

11. The issuer sends A, B, C, D, c^2, s^2 to the authenticator.
12. The authenticator checks that $A, B, C, D \in G^1$ and $A \neq 1^{G^1}$
13. The authenticator checks $H(P_1^{s^2} \cdot B^{-c^2} | Q^{s^2} \cdot D^{-c^2} | P_1 | B | Q | D) \stackrel{?}{=} c^2$

NOTE

$$P_1^{s^2} \cdot B^{-c^2} = P_1^{r^2} \cdot P_1^{c^2 \cdot l^J \cdot y} \cdot B^{-c^2} = U^2 \cdot B^{c^2} \cdot B^{-c^2} = U^2$$

$$Q^{s^2} \cdot D^{-c^2} = Q^{r^2} \cdot Q^{c^2 \cdot l^J \cdot y} \cdot D^{-c^2} = V^2 \cdot D^{c^2} \cdot D^{-c^2} = V^2$$

14. The authenticator checks $e(A, Y) \stackrel{?}{=} e(B, P_2)$

NOTE

$$e(A, Y) = e(P_1^{l^J}, P_2^y); e(B, P_2) = e(A^y, P_2) = e(P_1^{y l^J}, P_2)$$

15. and the authenticator checks $e(C, P_2) \stackrel{?}{=} e(A \cdot D, X)$

NOTE

$$e(C, P_2) = e(A^x \cdot Q^{x y l^J}, P_2); e(A \cdot D, X) = e(A \cdot Q^{y l^J}, P_2^x)$$

16. The authenticator stores credential A, B, C, D

3.4.2 ECDAAs Join Split between Authenticator and ASM

This section is non-normative.

NOTE

If this join is not in-factory, the value Q must be authenticated by the authenticator. Upon receiving this value, the issuer must verify that this authenticator did not join before.

1. The ASM asks the issuer for a nonce.
2. The issuer chooses a nonce BigInteger $n = RAND(p)$ and sends n to the ASM.
3. The ASM forwards n to the authenticator
4. The authenticator chooses and stores the private key BigInteger $sk = RAND(p)$
5. The authenticator computes its ECDAAs public key ECPoint $Q = P_1^{sk}$

6. The authenticator proves knowledge of sk as follows
 1. BigInteger $r^1 = RAND(p)$
 2. ECPoint $U^1 = P_1^{r^1}$
 3. BigInteger $c^1 = H(U^1|P_1|Q|n)$
 4. BigInteger $s^1 = r^1 + c^1 \cdot sk$
7. The authenticator sends Q, c^1, s^1 to the ASM, who forwards it to the issuer.
8. The issuer verifies that the authenticator is "authentic" and that Q was indeed generated by the authenticator. In the case of an in-factory Join, this might be trivial; in the case of a remote Join this typically requires the use of other cryptographic methods. Since ECDAJ-Join is a one-time operation, unlinkability is not a concern for that.
9. The issuer verifies that $Q \in G^1$ and verifies $H(P_1^{s^1} \cdot Q^{-c^1} | P_1 | Q | n) \stackrel{?}{=} c^1$.
10. The issuer creates credential (A, B, C, D) as follows
 1. BigInteger $l^J = RAND(p)$
 2. ECPoint $A = P_1^{l^J}$
 3. ECPoint $B = A^y$
 4. ECPoint $C = A^x \cdot Q^{xyl^J}$
 5. ECPoint $D = Q^{l^Jy}$
11. The issuer proves that it computed this credential correctly:
 1. BigInteger $r^2 = RAND(p)$
 2. ECPoint $U^2 = P_1^{r^2}$
 3. ECPoint $V^2 = Q^{r^2}$
 4. BigInteger $c^2 = H(U^2|V^2|P_1|B|Q|D)$
 5. BigInteger $s^2 = r^2 + c^2 \cdot l^J \cdot y$
12. The issuer sends A, B, C, D, c^2, s^2 to the ASM. The issuer authenticates B, D, c^2, s^2 such that the authenticator can verify they were created by the issuer.
13. The ASM checks that $A, B, C, D \in G^1$ and $A \neq 1^{G^1}$
14. The ASM checks $H(P_1^{s^2} \cdot B^{-c^2} | Q^{s^2} \cdot D^{-c^2} | P_1 | B | Q | D) \stackrel{?}{=} c^2$
15. The ASM checks $e(A, Y) \stackrel{?}{=} e(B, P_2)$
16. and the ASM checks that $e(C, P_2) \stackrel{?}{=} e(A \cdot D, X)$
17. The ASM stores A, B, C, D and sends B, D, c^2, s^2 to the authenticator
18. The authenticator checks $B, D \in G^1$ and $B \neq 1^{G^1}$, and verifies that B, D, c^2, s^2 were sent by the issuer.
19. The authenticator checks $H(P_1^{s^2} \cdot B^{-c^2} | Q^{s^2} \cdot D^{-c^2} | P_1 | B | Q | D) \stackrel{?}{=} c^2$
20. The authenticator stores B, D and ignores further join requests.

NOTE

These values belong to the ECDAJ secret keys sk . They should persist even in the case of a factory reset.

3.4.3 ECDAAs-Join Split between TPM and ASM

This section is non-normative.

NOTE

The Endorsement key credential (EK-C) and TPM2_ActivateCredentials are used for supporting the remote Join.

This description is based on the principles described in [TPMv2-Part1] section 24 and [Arthur-Challener-2015], page 109 ("Activating a Credential").

1. The ASM asks the ECDAAs Issuer for a nonce.
2. The ECDAAs Issue chooses a nonce BigInteger $n = RAND(p)$ and sends n to the ASM.
3. The ASM
 1. instructs the TPM to create a restricted key by calling TPM2_Create, giving the public key template `TPMT_PUBLIC` [TPMv2-Part2] (including the public key Q in field `unique`) to the ASM.
 2. retrieves TPM Endorsement Key Certificate (EK-C) from the TPM
 3. calls TPM2_Commit(keyhandle, P1, s2, y2) where keyhandle is the handle of the restricted key generated before (see above), P1 is set to P^1 , and s2 and y2 are left empty. This call returns K, L, E, and ctr; where K and L will be empty.
 4. computes BigInteger $c^1 = H(E|P^1|Q|n)$
 5. call TPM2_Sign(c^1 , ctr), returning s^1 .
 6. sends EK-C, `TPMT_PUBLIC` (including Q in field `unique`), c^1 , s^1 to the ECDAAs Issuer.
4. The ECDAAs Issuer
 1. verifies EK-C and its certificate chain. As a result the ECDAAs Issuer knows the TPM model related to EK-C.
 2. verifies that this EK-C was not used in a (successful) Join before
 3. Verifies that the `objectAttributes` in `TPMT_PUBLIC` [TPMv2-Part2] matches the following flags: `fixedTPM = 1; fixedParent = 1; sensitiveDataOrigin = 1; encryptedDuplication = 0; restricted = 1; decrypt = 0; sign = 1.`
 4. examines the public key Q , i.e. it verifies that $Q \in G^1$
 5. checks $H(P^1 \cdot Q^{-c^1} | P^1 | Q | n) \stackrel{?}{=} c^1$
 6. generates the ECDAAs credential (A, B, C, D) as follows
 1. BigInteger $l^J = RAND(p)$
 2. ECPoint $A = P^1$
 3. ECPoint $B = A^y$
 4. ECPoint $C = A^x \cdot Q^{xyl^J}$
 5. ECPoint $D = Q^{l^Jy}$
 7. proves that it computed this credential correctly:
 1. BigInteger $r^2 = RAND(p)$
 2. ECPoint $U^2 = P^1$
 3. ECPoint $V^2 = Q^{r^2}$
 4. BigInteger $c^2 = H(U^2|V^2|P^1|B|Q|D)$

5. BigInteger $s^2 = r^2 + c^2 \cdot lJ \cdot y$
8. generates a *secret* (derived from a *seed*) and wraps the credential A, B, C, D using that *secret*.
9. encrypts the *seed* using the public key included in EK-C.
10. uses *seed* and *name* in KDFa (see [TPMv2-Part2] section 24.4) to derive HMAC and *symmetric encryption key*. Wrap the *secret* in *symmetric encryption key* and protect it with the *HMAC key*.

NOTE

The parameter *name* in KDFa is derived from `TPMT_PUBLIC`, see [TPMv2-Part1], section 16.

11. sends the credential proof c^2, s^2 and the wrapped object including the credential from previous step to the ASM.
5. The ASM instructs the TPM (by calling `TPM2_ActivateCredential`) to
 1. decrypt the *seed* using the TPM Endorsement key
 2. compute the *name* (for the ECDAAtestation key)
 3. use the *seed* in KDFa (with *name*) to derive the *HMAC key* and the *symmetric encryption key*.
 4. use the *symmetric encryption key* to unwrap the *secret*.
6. The ASM
 1. unwraps the credential A, B, C, D using the *secret* received from the TPM.
 2. checks that $A, B, C, D \in G^1$ and $A \neq 1^{G^1}$
 3. checks $H(P_1^{s^2} \cdot B^{-c^2} | Q^{s^2} \cdot D^{-c^2} | P_1 | B | Q | D) \stackrel{?}{=} c^2$
 4. checks $e(A, Y) \stackrel{?}{=} e(B, P_2)$ and $e(C, P_2) \stackrel{?}{=} e(A \cdot D, X)$
 5. stores A, B, C, D

3.5 ECDAASign

NOTE

One ECDAASign operation is required for the client-side environment whenever a new credential is being registered at a relying party.

3.5.1 ECDAASign Algorithm

This section is normative.

(signature, KRD) = Ecdasign(String AppID)

Parameters

- p : System parameter prime order of group G^1 (global constant)
- AppID: FIDO AppID (i.e. https-URL of TrustedFacets object)

Algorithm outline

1. KRD = BuildAndEncodeKRD(); // all traditional Registration tasks are here
2. BigInteger $l = RAND(p)$
3. ECPoint $R = A^l$;
4. ECPoint $S = B^l$;

5. ECPoint $T = C^l$;
6. ECPoint $W = D^l$;
7. BigInteger $r = RAND(p)$
8. ECPoint $U = S^r$
9. BigInteger $c = H(U|S|W|AppID|H(KRD))$
10. BigInteger $s = r + c \cdot sk \pmod{p}$
11. signature = (c, s, R, S, T, W)
12. return (signature, KRD)

3.5.2 ECDAASign Split between Authenticator and ASM

This section is non-normative.

NOTE

This split requires both the authenticator and ASM to be honest to achieve anonymity. Only the authenticator must be trusted for unforgeability. The communication between ASM and authenticator must be secure.

Algorithm outline

1. The ASM randomizes the credential
 1. BigInteger $l = RAND(p)$
 2. ECPoint $R = A^l$;
 3. ECPoint $S = B^l$;
 4. ECPoint $T = C^l$;
 5. ECPoint $W = D^l$;
2. The ASM sends $l, AppID$ to the authenticator
3. The authenticator performs the following tasks
 1. KRD = BuildAndEncodeKRD(); // all traditional Registration tasks are here
 2. ECPoint $S' = B^l$
 3. ECPoint $W' = D^l$
 4. BigInteger $r = RAND(p)$
 5. ECPoint $U = S^r$
 6. BigInteger $c = H(U|S'|W'|AppID|H(KRD))$
 7. BigInteger $s = r + c \cdot sk \pmod{p}$
 8. Send c, s, KRD to the ASM
4. The ASM sets signature = (c, s, R, S, T, W) and outputs (signature, KRD)

3.5.3 ECDAASign Split between TPM and ASM

This section is non-normative.

NOTE

This algorithm is for the special case of a TPMv2 as authenticator. This case requires both the TPM

and ASM to be honest for anonymity and unforgeability (see [XYZF-2014](#))).

Algorithm outline

1. The ASM randomizes the credential
 1. BigInteger $l = RAND(p)$
 2. ECPoint $R = A^l$;
 3. ECPoint $S = B^l$;
 4. ECPoint $T = C^l$;
 5. ECPoint $W = D^l$;
2. The ASM calls TPM2_Commit() with $P1$ set to S and $s2, y2$ empty buffers. The ASM receives the result values $K, L, E = S^r$ and ctr. K and L are empty since $s2, y2$ are empty buffers.
3. The ASM calls TPM2_Create to generate the new authentication key pair.
4. The ASM calls TPM2_Certify() on the newly created key with ctr from the TPM2_Commit and $E, S, W, AppID$ as qualifying data ($E = S^r$ is returned by step 2). The ASM receives signature c, s and attestation block KRD (i.e. TPMS_ATTEST structure in this case).
5. The ASM sets signature = (c, s, R, S, T, W) and outputs (signature, KRD)

3.6 ECDAAs-Verify Operation

This section is normative.

NOTE

One ECDAAs-Verify operation is required for the FIDO Server as part of each FIDO Registration.

boolean EcdasVerify(signature, AppID, KRD, ModelName)

Parameters

- p : System parameter prime order of group G^1 (global constant)
- P^2 : System parameter generator of group G^2 (global constant)
- signature: (c, s, R, S, T, W)
- AppID: FIDO AppID
- KRD: Attestation Data object as defined in other specifications.
- ModelName: the claimed FIDO authenticator model (i.e. either AAID or AAGUID)

Algorithm outline

1. Based on the claimed ModelName, look up X, Y from trusted source
2. Check that $R, S, T, W \in G^1, R \neq 1^{G^1}$, and $S \neq 1^{G^1}$.
3. $H(S^s \cdot W^{-c} | S | W | AppID | H(KRD)) \stackrel{?}{=} c$; fail if not equal

NOTE

$$B = A^y = P_1^{ly}$$

$$D = Q^{ly} = P_1^{skly} = B^{sk}$$

$$S = B^l \text{ and } W = D^l$$

$$U = S^r$$

$$\begin{aligned} S^s \cdot W^{-c} &= S^{r+csk} \cdot W^{-c} = U \cdot S^{csk} \cdot W^{-c} \\ &= U \cdot B^{lcsk} \cdot D^{-lc} = U \cdot B^{lcsk} \cdot B^{-lcsk} = U \end{aligned}$$

4. $e(R, Y) \stackrel{?}{=} e(S, P^2)$; fail if not equal

NOTE

$$e(R, Y) = e(A^l, P_2^y); e(S, P^2) = e(B^l, P^2) = e(A^{ly}, P^2)$$

5. $e(T, P^2) \stackrel{?}{=} e(R \cdot W, X)$; fail if not equal

NOTE

$$e(T, P^2) = e(C^l, P^2) = e(A^{xl} \cdot Q^{xlyl^j}, P^2); e(R \cdot W, X) = e(A^l \cdot D^l, X) = e(A^l \cdot Q^{lyl^j}, P_2^x)$$

6. for (all sk' on RogueList) do if $W \stackrel{?}{=} S^{sk'}$ fail;

7. // perform all other processing steps for new credential registration

NOTE

In the case of a TPMv2, i.e. KRD is a `TPMS_ATTEST` object. In this case the verifier must check whether the `TPMS_ATTEST` object starts with `TPM_GENERATED` magic number and whether its field `objectAttributes` contains the flag `fixedTPM=1` (indicating that the key was generated by the TPM).

8. return true;

4. FIDO ECDAAs Object Formats and Algorithm Details

This section is normative.

4.1 Supported Curves for ECDAAs

Definition of G1

G1 is an elliptic curve group $E : y^2 = x^3 + ax + b$ over $F(q)$ with $a = 0$.

Definition of G2

G2 is the p-torsion subgroup of $E'(Fq^2)$ where E' is a sextic twist of E . With $E' : y'^2 = x'^3 + b'$.

An element of $F(q^2)$ is represented by a pair (a, b) where $a + bX$ is an element of $F(q)[X] / \langle X^2 + 1 \rangle$. We use angle brackets $\langle Y \rangle$ to signify the ideal generated by the enclosed value.

NOTE

In the literature the pair (a,b) is sometimes also written as a complex number $a + b * i$.

Definition of GT

GT is an order-p subgroup of Fq^{12} .

Pairings

We propose the use of Ate pairings as they are efficient (more efficient than Tate pairings) on Barreto-Naehrig curves [DevScoDah2007].

Supported BN curves

We use pairing-friendly Barreto-Naehrig [BarNae-2006] [ISO15946-5] elliptic curves. The curves TPM_ECC_BN_P256 and TPM_ECC_BN_P638 curves are defined in [TPMv2-Part4].

BN curves have a Modulus $q = 36 \cdot u^4 + 36 \cdot u^3 + 24 \cdot u^2 + 6 \cdot u + 1$ [ISO15946-5] and a related order of the group $p = 36 \cdot u^4 + 36 \cdot u^3 + 18 \cdot u^2 + 6 \cdot u + 1$ [ISO15946-5].

- TPM_ECC_BN_P256 is a curve of form E(F(q)), where q is the field modulus [TPMv2-Part4] [BarNae-2006]. This curve is identical to the P256 curve defined in [ISO15946-5] section C.3.5.
 - The values have been generated using $u=7\ 530\ 851\ 732\ 716\ 300\ 289$.
 - Modulus $q = 115\ 792\ 089\ 237\ 314\ 936\ 872\ 688\ 561\ 244\ 471\ 742\ 058\ 375\ 878\ 355\ 761\ 205\ 198\ 700\ 409\ 522\ 629\ 664\ 518\ 163$
 - Group order $p = 115\ 792\ 089\ 237\ 314\ 936\ 872\ 688\ 561\ 244\ 471\ 742\ 058\ 035\ 595\ 988\ 840\ 268\ 584\ 488\ 757\ 999\ 429\ 535\ 617\ 037$
 - p and q have length of 256 bit each.
 - $b = 3$
 - $P^1_{256} = (x=1, y=2)$
 - $b' = (a=3, b=3)$
 - $P^2_{256} = (x,y)$, with
 - $P^2_{256}.x = (a=114\ 909\ 019\ 869\ 825\ 495\ 805\ 094\ 438\ 766\ 505\ 779\ 201\ 460\ 871\ 441\ 403\ 689\ 227\ 802\ 685\ 522\ 624\ 680\ 861\ 435, b=35\ 574\ 363\ 727\ 580\ 634\ 541\ 930\ 638\ 464\ 681\ 913\ 209\ 705\ 880\ 605\ 623\ 913\ 174\ 726\ 536\ 241\ 706\ 071\ 648\ 811)$
 - $P^2_{256}.y = (a=65\ 076\ 021\ 719\ 150\ 302\ 283\ 757\ 931\ 701\ 622\ 350\ 436\ 355\ 986\ 716\ 727\ 896\ 397\ 520\ 706\ 509\ 932\ 529\ 649\ 684, b=113\ 380\ 538\ 053\ 789\ 372\ 416\ 298\ 017\ 450\ 764\ 517\ 685\ 681\ 349\ 483\ 061\ 506\ 360\ 354\ 665\ 554\ 452\ 649\ 749\ 368)$
- TPM_ECC_BN_P638 [TPMv2-Part4] uses
 - The values have been generated using $u=365\ 375\ 408\ 992\ 443\ 362\ 629\ 982\ 744\ 420\ 548\ 242\ 302\ 862\ 098\ 433$
 - Modulus $q = 641\ 593\ 209\ 463\ 000\ 238\ 284\ 923\ 228\ 689\ 168\ 801\ 117\ 629\ 789\ 043\ 238\ 356\ 871\ 360\ 716\ 989\ 515\ 584\ 497\ 239\ 494\ 051\ 781\ 991\ 794\ 253\ 619\ 096\ 481\ 315\ 470\ 262\ 367\ 432\ 019\ 698\ 642\ 631\ 650\ 152\ 075\ 067\ 922\ 231\ 951\ 354\ 925\ 301\ 839\ 708\ 740\ 457\ 083\ 469\ 793\ 717\ 125\ 223$
 - The related order of the group is $p = 641\ 593\ 209\ 463\ 000\ 238\ 284\ 923\ 228\ 689\ 168\ 801\ 117\ 629\ 789\ 043\ 238\ 356\ 871\ 360\ 716\ 989\ 515\ 584\ 497\ 239\ 494\ 051\ 781\ 991\ 794\ 252\ 818\ 101\ 344\ 337\ 098\ 690\ 003\ 906\ 272\ 221\ 387\ 599\ 391\ 201\ 666\ 378\ 807\ 960\ 583\ 525\ 233\ 832\ 645\ 565\ 592\ 955\ 122\ 034\ 352\ 630\ 792\ 289$
 - p and q have length of 638 bit each.
 - $b = 257$
 - $P^1_{638} = (x=641\ 593\ 209\ 463\ 000\ 238\ 284\ 923\ 228\ 689\ 168\ 801\ 117\ 629\ 789\ 043\ 238\ 356\ 871\ 360\ 716\ 989\ 515\ 584\ 497\ 239\ 494\ 051\ 781\ 991\ 794\ 253\ 619\ 096\ 481\ 315\ 470\ 262\ 367\ 432\ 019\ 698\ 642\ 631\ 650\ 152\ 075\ 067\ 922\ 231\ 951\ 354\ 925\ 301\ 839\ 708\ 740\ 457\ 083\ 469\ 793\ 717\ 125\ 222, y=16)$
 - $b' = (a=771, b=1542)$

- $P^2_{638} = (x, y)$, with
 - $P^2_{638}.x = (a=192\ 492\ 098\ 325\ 059\ 629\ 927\ 844\ 609\ 092\ 536\ 807\ 849\ 769\ 208\ 589\ 403\ 233\ 289\ 748\ 474\ 758\ 010\ 838\ 876\ 457\ 636\ 072\ 173\ 883\ 771\ 602\ 089\ 605\ 233\ 264\ 992\ 910\ 618\ 494\ 201\ 909\ 695\ 576\ 234\ 119\ 413\ 319\ 303\ 931\ 909\ 848\ 663\ 554\ 062\ 144\ 113\ 485\ 982\ 076\ 866\ 968\ 711\ 247, b=166\ 614\ 418\ 891\ 499\ 184\ 781\ 285\ 132\ 766\ 747\ 495\ 170\ 152\ 701\ 259\ 472\ 324\ 679\ 873\ 541\ 478\ 330\ 301\ 406\ 623\ 174\ 002\ 502\ 345\ 930\ 325\ 474\ 988\ 134\ 317\ 071\ 869\ 554\ 535\ 111\ 092\ 924\ 719\ 466\ 650\ 228\ 182\ 095\ 841\ 246\ 668\ 361\ 451\ 788\ 368\ 418\ 036\ 777\ 197\ 454\ 618\ 413\ 255)$
 - $P^2_{638}.y = (a=622\ 964\ 952\ 935\ 200\ 827\ 531\ 506\ 751\ 874\ 167\ 806\ 262\ 407\ 152\ 244\ 280\ 323\ 674\ 626\ 687\ 789\ 202\ 660\ 794\ 092\ 633\ 841\ 098\ 984\ 322\ 671\ 973\ 226\ 667\ 873\ 503\ 889\ 270\ 602\ 870\ 064\ 426\ 165\ 592\ 237\ 410\ 681\ 318\ 519\ 893\ 784\ 898\ 821\ 343\ 051\ 339\ 820\ 566\ 224\ 981\ 344\ 169\ 470, b=514\ 285\ 963\ 827\ 225\ 043\ 076\ 463\ 721\ 426\ 569\ 583\ 576\ 029\ 220\ 880\ 138\ 564\ 906\ 219\ 230\ 942\ 887\ 639\ 456\ 599\ 654\ 554\ 743\ 732\ 087\ 558\ 187\ 149\ 207\ 036\ 952\ 474\ 092\ 411\ 405\ 629\ 612\ 957\ 921\ 369\ 286\ 372\ 038\ 525\ 830\ 610\ 755\ 207\ 588\ 843\ 864\ 366\ 759\ 521\ 090\ 861\ 911\ 494)$
- **ECC_BN_DSD_P256** [[DevScoDah2007](#)] section 3 uses
 - The values have been generated using $u=6\ 917\ 529\ 027\ 641\ 089\ 837$
 - Modulus $q = 82434016654300679721217353503190038836571781811386228921167322412819029493183$
 - The related order of the group is $p = 82434016654300679721217353503190038836284668564296686430114510052556401373769$
 - p and q have length of 256 bit each.
 - $b = 3$
 - $P^1_{DSD_P256} = (1, 2)$
 - $b' = (a=3, b=6)$
 - $P^2_{DSD_P256} = (x, y)$, with
 - $P^2_{DSD_P256}.x = (a=73\ 481\ 346\ 555\ 305\ 118\ 071\ 940\ 904\ 527\ 347\ 990\ 526\ 214\ 212\ 698\ 180\ 576\ 973\ 201\ 374\ 397\ 013\ 567\ 073\ 039, b=28\ 955\ 468\ 426\ 222\ 256\ 383\ 171\ 634\ 927\ 293\ 329\ 392\ 145\ 263\ 879\ 318\ 611\ 908\ 127\ 165\ 887\ 947\ 997\ 417\ 463)$
 - $P^2_{DSD_P256}.y = (a=3\ 632\ 491\ 054\ 685\ 712\ 358\ 616\ 318\ 558\ 909\ 408\ 435\ 559\ 591\ 759\ 282\ 597\ 787\ 781\ 393\ 534\ 962\ 445\ 630\ 353, b=60\ 960\ 585\ 579\ 560\ 783\ 681\ 258\ 978\ 162\ 498\ 088\ 639\ 544\ 584\ 959\ 644\ 221\ 094\ 447\ 372\ 720\ 880\ 177\ 666\ 763)$
- **ECC_BN_ISOP512** [[ISO15946-5](#)] section C.3.7 uses
 - The values have been generated using $u=138\ 919\ 694\ 570\ 470\ 098\ 040\ 331\ 481\ 282\ 401\ 523\ 727$
 - Modulus $q = 13\ 407\ 807\ 929\ 942\ 597\ 099\ 574\ 024\ 998\ 205\ 830\ 437\ 246\ 153\ 344\ 875\ 111\ 580\ 494\ 527\ 427\ 714\ 590\ 099\ 881\ 795\ 845\ 981\ 157\ 516\ 604\ 994\ 291\ 639\ 750\ 834\ 285\ 779\ 043\ 186\ 149\ 750\ 164\ 319\ 950\ 153\ 126\ 044\ 364\ 566\ 323$
 - The related order of the group is $p = 13\ 407\ 807\ 929\ 942\ 597\ 099\ 574\ 024\ 998\ 205\ 830\ 437\ 246\ 153\ 344\ 875\ 111\ 580\ 494\ 527\ 427\ 714\ 590\ 099\ 881\ 680\ 053\ 891\ 920\ 200\ 409\ 570\ 720\ 654\ 742\ 146\ 445\ 677\ 939\ 306\ 408\ 461\ 754\ 626\ 647\ 833\ 262\ 056\ 300\ 743\ 149$
 - p and q have length of 512 bit each.
 - $b = 3$
 - $P^1_{ISO_P512} = (x=1, y=2)$
 - $b' = (a=3, b=3)$
 - $P^2_{ISO_P512} = (x, y)$, with
 - $P^2_{ISO_P512}.x = (a=3\ 094\ 648\ 157\ 539\ 090\ 131\ 026\ 477\ 120\ 117\ 259\ 896\ 222\ 920\ 557\ 994\ 037\ 039\ 545\ 437\ 079\ 729\ 804\ 516\ 315\ 481\ 514\ 566\ 156\ 984\ 245\ 473\ 190\ 248\ 967\ 907\ 724\ 153\ 072\ 490\ 467\ 902\ 779\ 495\ 072\ 074\ 156\ 718\ 085\ 785\ 269, b=3\ 776\ 690\ 234\ 788\ 102\ 103\ 015\ 760\ 376\ 468\ 067\ 863\ 580\ 475\ 949\ 014\ 286\ 077\ 855\ 600\ 384\ 033\ 870\ 546\ 339\ 773\ 119\ 295\ 555\ 161\ 718\ 985\ 244\ 561\ 452\ 474\ 412\ 673\ 836\ 012\ 873\ 126\ 926\ 524\ 076\ 966\ 265\ 127\ 900\ 471\ 529)$
 - $P^2_{ISO_P512}.y = (a=7\ 593\ 872\ 605\ 334\ 070\ 150\ 001\ 723\ 245\ 210\ 278\ 735\ 800\ 573$

263 881 411 015 285 406 372 548 542 328 752 430 917 597 485 450 360 707 892 769
 159 214 115 916 255 816 324 924 295 339 525 686 777 569 132 644 242, b=9 131 995
 053 349 122 285 871 305 684 665 648 028 094 505 015 281 268 488 257 987 110 193
 875 868 585 868 792 041 571 666 587 093 146 239 570 057 934 816 183 220 992 460
 187 617 700 670 514 736 173 834 408)

NOTE

Spaces are used inside numbers to improve readability.

Hash Algorithms

Depending on the curve, we use $H(x) = \text{SHA256}(x) \bmod p$ or $H(x) = \text{SHA512}(x) \bmod p$ as hash algorithm $H: \{0, 1\}^* \rightarrow Z^p$.

The argument of the hash function must always be converted to a byte string using the appropriate encoding function specific in section 3.1 [Object Encodings](#), e.g. according to section 3.1.3 [Encoding ECPoint2 values as byte strings \(ECPoint2ToB\)](#) in the case of `ECPoint2` points.

NOTE

We don't use [IEEE P1363.3](#) section 6.1.1 IHF1-SHA with security parameter t (e.g. t=128 or 256) as it is more complex and not supported by TPMv2.

4.2 ECDAAs Algorithm Names

We define the following JWS-style algorithm names (see [\[RFC7515\]](#)):

ED256

`TPM_ECC_BN_P256` curve, using SHA256 as hash algorithm H.

ED256-2

`ECC_BN_DSD_P256` curve, using SHA256 as hash algorithm H.

ED512

`ECC_BN_ISOP512` curve, using SHA512 as hash algorithm H.

ED638

`TPM_ECC_BN_P638` curve, using SHA512 as hash algorithm H.

4.3 ecdaaSignature object

The fields c and s both have length N. The fields R, S, T, W have equal length (2*N+1 each).

In the case of `BN_P256` curve (with key length N=32 bytes), the fields R, S, T, W have length 2*32+1=65 bytes. The fields c and s have length N=32 each.

The `ecdaaSignature` object is a binary object generated as the concatenation of the binary fields in the order described below (total length of 324 bytes for 256bit curves):

Value	Length (in Bytes)	Description
UINT8[] ECDAAsignature_c	N	<p>The c value, $c = H(U S W KRD AppID)$ as returned by <code>AuthnrEcdaaSign</code> encoded as byte string according to <code>BigNumberToB</code>.</p> <p>Where</p> <ul style="list-style-type: none"> $U = S^r$, with $r = RAND(p)$ computed by the signer. KRD is the the entire to-be-signed object (e.g. <code>TAG_UAFV1_KRD</code> in the case of FIDO UAF). $S = B^l$, with $l = RAND(p)$ computed by the signer and $B = A^y$ computed in the ECDAAsignature-Join

Value	Length (in Bytes)	Description
UINT8[] ECDAASignature_s	N	The s value, $s=r + c * sk$ (mod p), returned by AuthnrEcdaaSign encoded as byte string according to BigIntegerToB. Where <ul style="list-style-type: none"> r = RAND(p), computed by the signer at FIDO registration (see 3.5.2 ECDAASign Split between Authenticator and ASM) p is the group order of G1 sk: is the <u>authenticator's</u> attestation secret key, see above
UINT8[] ECDAASignature_R	2*N+1	$R = A^l$; computed by the <u>ASM</u> or the <u>authenticator</u> at FIDO registration; encoded as byte string according to ECPointToB. Where <ul style="list-style-type: none"> l = RAND(p), i.e. random number $0 \leq l \leq p$. Computed by the <u>ASM</u> or the <u>authenticator</u> at FIDO registration. And where $R = A^l$ denotes the scalar multiplication (of scalar l) of a curve point A. Where A has been provided by the ECDAASign-Issuer as part of ECDAASign-Join: $A = P_1^{l^J}$, see 3.4.1 ECDAASign-Join Algorithm. Where P_1 and p are system values, injected into the <u>authenticator</u> and l^J is a random number computed by the ECDAASign-Issuer on Join.
UINT8[] ECDAASignature_S	2*N+1	$S = B^l$; computed by the <u>ASM</u> or the <u>authenticator</u> at FIDO registration encoded as byte string according to ECPointToB. Where B has been provided by the ECDAASign-Issuer on Join: $B = A^y$, see 3.4.1 ECDAASign-Join Algorithm .
UINT8[] ECDAASignature_T	2*N+1	$T = C^l$; computed by the <u>ASM</u> or the <u>authenticator</u> at FIDO registration encoded as byte string according to ECPointToB. Where <ul style="list-style-type: none"> $C = A^x \cdot Q^{xy l^J}$, provided by the ECDAASign-Issuer on Join $l^J = RAND(p)$ computed by the ECDAASign-Issuer at Join (see 3.4.1 ECDAASign-Join Algorithm) x and y are components of the ECDAASign-Issuer private key, $isk=(x,y)$. Q is the <u>authenticator</u> public key
UINT8[] ECDAASignature_W	2*N+1	$W = D^l$; computed by the <u>ASM</u> or the <u>authenticator</u> at FIDO registration encoded as byte string according to ECPointToB. Where $D = Q^{ly}$ is computed by the ECDAASign-Issuer at Join (see 3.4.1 ECDAASign-Join Algorithm).

5. Considerations

This section is non-normative.

A detailed security analysis of this algorithm can be found in [[FIDO-DAA-Security-Proof](#)].

5.1 Algorithms and Key Sizes

The proposed algorithms and key sizes are chosen such that compatibility to TPMv2 is possible.

5.2 Indicating the Authenticator Model

Some authenticators (e.g. TPMv2) do not have the ability to include their model (i.e. vendor ID and model name) in attested messages (i.e. the to-be-signed part of the registration assertion). The TPM's endorsement key certificate typically contains that information directly or at least it allows the model to be derived from the endorsement key certificate.

In FIDO, the relying party expects the ability to cryptographically verify the authenticator model.

We require the ECDAA-Issuers public key ($ipk=(X,Y,c,sx,sy)$) to be dedicated to one single authenticator model (e.g. as identified by AAID or AAGUID).

5.3 Revocation

If the private ECDAA attestation key sk of an authenticator has been leaked, it can be revoked by adding its value to a RogueList.

The ECDAA-Verifier (i.e. FIDO Server) check for such revocations. See section [3.6 ECDAA-Verify Operation](#).

The ECDAA-Issuer is expected to check revocation by other means:

1. if ECDAA-Join is done in-factory, it is assumed that produced devices are known to be uncompromised (at time of production).
2. if a remote ECDAA-Join is performed, the (remote) ECDAA-Issuer already must use a different method to remotely authenticate the authenticator (e.g. using some endorsement key). We expect the ECDAA-Issuer to perform a revocation check based on that information. This is even more flexible as it does not require access to the authenticator ECDAA private key sk .

5.4 Pairing Algorithm

The pairing algorithm e needs to be used by the ASM as part of the Join process and by the verifier (i.e. FIDO relying party) as part of the verification (i.e. FIDO registration) process.

The result of such a pairing operation is only compared to the result of another pairing operation computed by the same entity. As a consequence, it doesn't matter whether the ASM and the verifier use the exact same pairings or not (as long as they both use valid pairings).

5.5 Performance

For performance reasons the calculation of $Sig2=(R, S, T, W)$ may be performed by the ASM running on the FIDO user device (as opposed to inside the authenticator). See section [3.5.2 ECDAA-Sign Split between Authenticator and ASM](#).

The cryptographic computations to be performed inside the authenticator are limited to G1. The ECDAA-Issuer has to perform two G2 point multiplications for computing the public key. The Verifier (i.e. FIDO relying party) has to perform G1 operations and two pairing operations.

5.6 Binary Concatentation

We use a simple byte-wise concatenation function for the different parameters, i.e. $H(a,b) = H(a || b)$.

This approach is as secure as the underlying hash algorithm since the authenticator controls the length of the (fixed-length) values (e.g. U, S, W). The AppID is provided externally and has unverified structure and length. However, it is only followed by a fixed length entry - the (system defined) hash of KRD. As a consequence, no parts of the AppID would ever be confused with the fixed length value.

5.7 IANA Considerations

This specification registers the algorithm names "ED256", "ED512", and "ED638" defined in section [4. FIDO ECDAA Object Formats and Algorithm Details](#) with the IANA JSON Web Algorithms registry as defined in section "Cryptographic Algorithms for Digital Signatures and MACs" in [[RFC7518](#)].

Algorithm Name	"ED256"
Algorithm Description	FIDO ECDAAs algorithm based on TPM_ECC_BN_P256 [TPMv2-Part4] curve using SHA256 hash algorithm.
Algorithm Usage Location(s)	"alg", i.e. used with JWS.
JOSE Implementation Requirements	Optional
Change Controller	FIDO Alliance, Contact Us
Specification Documents	Sections 3. FIDO ECDAAs Attestation and 4. FIDO ECDAAs Object Formats and Algorithm Details of [FIDOEcdaaAlgorithm].
Algorithm Analysis Document(s)	[FIDO-DAA-Security-Proof]

Algorithm Name	"ED512"
Algorithm Description	ECDAAs algorithm based on ECC_BN_ISOP512 [ISO15946-5] curve using SHA512 algorithm.
Algorithm Usage Location(s)	"alg", i.e. used with JWS.
JOSE Implementation Requirements	Optional
Change Controller	FIDO Alliance, Contact Us
Specification Documents	Sections 3. FIDO ECDAAs Attestation and 4. FIDO ECDAAs Object Formats and Algorithm Details of [FIDOEcdaaAlgorithm].
Algorithm Analysis Document(s)	[FIDO-DAA-Security-Proof]

Algorithm Name	"ED638"
Algorithm Description	ECDAAs algorithm based on TPM_ECC_BN_P638 [TPMv2-Part4] curve using SHA512 algorithm.
Algorithm Usage Location(s)	"alg", i.e. used with JWS.
JOSE Implementation Requirements	Optional
Change Controller	FIDO Alliance, Contact Us
Specification Documents	Sections 3. FIDO ECDAAs Attestation and 4. FIDO ECDAAs Object Formats and Algorithm Details of [FIDOEcdaaAlgorithm].
Algorithm Analysis Document(s)	[FIDO-DAA-Security-Proof]

A. References

A.1 Normative references

[ECDSA-ANSI]

[Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm \(ECDSA\), ANSI X9.62-2005](#), November 2005. URL: <http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+X9.62%3A2005>

[RFC2119]

S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels* March 1997. Best Current Practice. URL: <https://tools.ietf.org/html/rfc2119>

[RFC3447]

J. Jonsson; B. Kaliski. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*. February 2003. Informational. URL: <https://tools.ietf.org/html/rfc3447>

[TPMv2-Part4]

Trusted Platform Module Library, Part 4: Supporting Routines URL: http://www.trustedcomputinggroup.org/files/static_page_files/8C6CABBC-1A4B-B294-D0DA8CE1B452CAB4/TPM%20Rev%202.0%20Part%204%20-%20Supporting%20Routines%2001.16-code.pdf

A.2 Informative references

[ANZ-2013]

Tolga Acar; Lan Nguyen; Greg Zaverucha. *A TPM Diffie-Hellman Oracle*. October 18, 2013. URL: <http://eprint.iacr.org/2013/667.pdf>

[Arthur-Challener-2015]

Will Arthur; David Challener; Kenneth Goldman. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. 2014. URL: <http://www.apress.com/9781430265832>

[BFGSW-2011]

D. Bernhard; G. Fuchsbauer; E. Ghadafi; N. P. Smart; B. Warinschi. *Anonymous Attestation with User-controlled Linkability*. 2011. URL: <http://eprint.iacr.org/2011/658.pdf>

[BarNae-2006]

Paulo S. L. M. Barreto; Michael Naehrig. *Pairing-Friendly Elliptic Curves of Prime Order*. 2006. URL: <http://research.microsoft.com/pubs/118425/ptcpcp.pdf>

[BriCamChe2004-DAA]

Ernie Brickell; Jan Camenisch; Liqun Chen. *Direct Anonymous Attestation*. 2004. URL: <http://eprint.iacr.org/2004/205.pdf>

[CheLi2013-ECDAAs]

Liqun Chen; Jiangtao Li. *Flexible and Scalable Digital Signatures in TPM 2.0* 2013. URL: <http://dx.doi.org/10.1145/2508859.2516729>

[DevScoDah2007]

Augusto Jun Devegili; Michael Scott; Ricardo Dahab. *Implementing Cryptographic Pairings over Barreto-Naehrig Curves*. 2007. URL: <https://eprint.iacr.org/2007/390.pdf>

[FIDO-DAA-Security-Proof]

Jan Camenisch; Manu Drijvers; Anja Lehmann. *Universally Composable Direct Anonymous Attestation*. 2015. URL: <https://eprint.iacr.org/2015/1246>

[FIDOEcdaaAlgorithm]

R. Lindemann; J. Camenisch; M. Drijvers; A. Edgington; A. Lehmann; R. Urian. *FIDO ECDAAs Algorithm*. Implementation Draft. URL: <https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-ecdaa-algorithm-v2.0-ps-20170927.html>

[FIDOGlossary]

R. Lindemann; D. Baghdasaryan; B. Hill; J. Hodges. *FIDO Technical Glossary*. Implementation Draft. URL: <https://fidoalliance.org/specs/fido-v2.0-ps-20170927/fido-glossary-v2.0-ps-20170927.html>

[ISO15946-5]

ISO/IEC 15946-5 Information Technology - Security Techniques - Cryptographic techniques based on elliptic curves - Part 5: Elliptic curve generation. URL: <https://webstore.iec.ch/publication/10468>

[RFC7515]

M. Jones; J. Bradley; N. Sakimura. *JSON Web Signature (JWS) (RFC7515)*. May 2015. URL: <http://www.ietf.org/rfc/rfc7515.txt>

[RFC7518]

M. Jones. *JSON Web Algorithms (JWA)*. May 2015. Proposed Standard. URL: <https://tools.ietf.org/html/rfc7518>

[TPMv1-2-Part1]

TPM 1.2 Part 1: Design Principles URL: http://www.trustedcomputinggroup.org/files/static_page_files/72C26AB5-1A4B-B294-D002BC0B8C062FF6/TPM%20Main-Part%201%20Design%20Principles_v1.2_rev116_01032011.pdf

[TPMv2-Part1]

Trusted Platform Module Library, Part 1: Architecture URL: http://www.trustedcomputinggroup.org/files/static_page_files/8C56AE3E-1A4B-B294-D0F43097156A55D8/TPM%20Rev%202.0%20Part%201%20-%20Architecture%2001.16.pdf

[TPMv2-Part2]

Trusted Platform Module Library, Part 2: Structures URL: http://www.trustedcomputinggroup.org/files/static_page_files/8C583202-1A4B-B294-D0469592DB10A6CD/TPM%20Rev%202.0%20Part%202%20-%20Structures%2001.16.pdf

[XYZF-2014]

Li Xi; Kang Yang; Zhenfeng Zhang; Dengguo Feng. *DAA-Related APIs in TPM 2.0 Revisited, in T. Holz and S. Ioannidis (Eds.)*. 2014. URL:

