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 FIDO-DAA-Security-Proof] based on elliptic curves and bilinear pairings largely compatible with [CheLi2013-ECDAA] called ECDAA. 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].

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

This specification includes the fixes of the issue regarding (1) the Diffie-Hellman oracle w.r.t. the secret key of the TPM and (2) regarding the potential privacy violations by fraudulent TPMs as proposed in [CCDLNU2017-DAA].

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 Review Draft. This document is intended to become a FIDO Alliance Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

This is a Review Draft Specification and is not intended to be a basis for any implementations as the Specification may change. Permission is hereby granted to use the Specification solely for the purpose of reviewing the Specification. No rights are granted to prepare derivative works of this Specification. Entities seeking permission to reproduce portions of this Specification for other uses must contact the FIDO Alliance to determine whether an appropriate license for such use is available.

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, 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.

Table of Contents

- 1. Notation
  - 1.1 Conformance
- 2. Overview
2.1 Scope

This document describes the FIDO ECDAA attestation algorithm in detail.

2.2 Architecture Overview

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 ECDAA algorithm, the KRD object is signed using 3.5 ECDAA-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 ECDAA 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, ...).
2.2 Architecture Overview

ECDAA attestation defines global system parameters and ECDAA Issuer specific parameters. Both parameter sets need to be installed on the host, in the authenticator and in the FIDO Server. The ECDAA method consists of two steps:

- **ECDAA-Join** between the authenticator and the ECDAA Issuer to be performed before the first FIDO Registration. The ECDAA Issuer represents the authenticator vendor as it provides the credentials to attest the authenticator model.
  - (n, B, sc, yc) = GetNonceFromECDAAIssuer()
  - (D, Q, c1, s1) = EcdaaJoin(X, Y, B, sc, yc, n)
  - (A, B, C, D) = EcdaaJoin(Q, O, c1, s1)
  - EcdaaJoin2(A, C) // store cre=(A, B, C, D)

- and the pair of **ECDAA-Sign** performed by the authenticator and **ECDAA-Verify** performed by the FIDO Server of the relying party as part of the FIDO Registration.
  - Client: Attestation = (signature, KRD) = EcdaaSign(AppID)
  - Server: success=EcdaaVerify(signature, KRD, AppID)

The technical implementation details of the ECDAA-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 ECDAA-Sign and ECDAA-Verify steps and the encoding of the related ECDAA 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 ECDAA Attestation

This section is normative.

3.1 Object Encodings

We need to convert `BigNumber` and `ECPoint` objects to byte strings using the following encoding functions:

### 3.1.1 Encoding `BigNumber` values as byte strings (BigNumberToB)

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 BigNumber 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:

```java
public static ByteArray BigNumberToB(BigNumber nVal, int size)
{...
```

### 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 ECDAA. Each `ECPoint2` is represented by a pair $(a, b)$ of elements of $F(q^2)$.

The group zero element is always encoded (using the encoding rules as described below) as 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 cy followed by the cx 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)
```
3.2 Global ECDAA System Parameters

1. Groups \( G^1, G^2 \) and \( GT \), of sufficiently large prime order \( p \)
2. Two generators \( P^2 \) and \( P^3 \), such that \( G^1 = \langle P^2 \rangle \) and \( G^2 = \langle P^3 \rangle \)
3. A bilinear pairing \( e : G^1 \times G^2 \rightarrow GT \). 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. Hash function \( H^{(G)} \) with \( H^{(G)} : \{0,1\}^* \rightarrow G^1 \).
6. \( (G^1, P^2, P^3, H, H^{(G)}) \) are installed in all authenticators implementing FIDO ECDAA attestation.

**Definition of \( G^1, G^2, GT, \) Pairings, hash function \( H \) and \( H^{(G)} \)**

See section 4.1 Supported Curves for ECDAA.

3.3 Issuer Specific ECDAA Parameters

ECDAA Issuer Parameters part

1. Randomly generated ECDAA Issuer private key \( isk = (x, y) \) with \( x, y = RAND(p) \).
2. ECDAA Issuer public key \( (X, Y) \), with \( X = P^2 x \) and \( Y = P^3 y \).
3. A proof that the ECDAA Issuer key was correctly computed
   1. BigInteger \( rx = RAND(p) \)
   2. BigInteger \( ry = RAND(p) \)
   3. ECPoint2 \( Ux = P^2 x \)
   4. ECPoint2 \( Uy = P^3 y \)
   5. BigInteger \( c = H(Ux|Uy|P^2|X|Y) \)
   6. BigInteger \( s^x = rx + c \cdot x \pmod p \)
   7. BigInteger \( s^y = ry + c \cdot y \pmod p \)
4. \( ipk = X, Y, c, s^x, s^y \)

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

\[
H(P^2 x \cdot X^{-c}|P^3 y \cdot Y^{-c}|P^2 x|X|Y) = c
\]

**NOTE**

\[
\begin{align*}
P^2 x \cdot X^{-c} &= P^2 x^{x+c} \\ P^3 y \cdot Y^{-c} &= P^2 y^{x+c} \\
Ux &= P^2 x \\
Uy &= P^3 y
\end{align*}
\]

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 issue 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 ECDAA Issuer and the authenticator. As this operation is performed once and only with a single ECDAA Issuer, privacy is preserved and an authenticator specific credential can and should be used.

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

3.4.1 ECDAA-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 ECDAA Issuer must verify that this authenticator did not join before.

1. The authenticator asks the ECDAA Issuer for the B value of the credential.
2. The ECDAA Issuer chooses a nonce BigInteger m = RAND(p).
3. The ECDAA Issuer computes the B value of the credential as B = H^G(B)(m) and sends (sc,yc)=H^G(pre(m)) to the authenticator.
4. The authenticator chooses and stores the ECDAA private key BigInteger sk = RAND(p).
5. The authenticator re-computes B = (H(sc,yc))
6. The authenticator computes its ECDAA public key ECPoint Q = B^sk
7. The authenticator proves knowledge of sk as follows
   1. BigInteger r^1 = RAND(p)
   2. ECPoint U^1 = B^r^1
   3. BigInteger c^2 = H(U^1|P|^Q|m)
   4. BigInteger n = RAND(p)
   5. BigInteger c^1 = H(n|c^2)
   6. BigInteger s^1 = r^1 + c^1 \cdot sk
8. The authenticator sends Q, c^1, s^1, n via the ASM to the ECDAA Issuer
9. The ECDAA 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 ECDAA-Join is a one-time operation, unlinkability is not a concern for that.
10. The ECDAA Issuer verifies that Q ∈ G^1 and verifies H(n|H(B^s^1 \cdot Q^{-c^1}|P|^Q|m)) \neq c^1 (check proof-of-possession of private key).

NOTE

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

11. The ECDAA Issuer creates credential (A, B, C, D) as follows
   1. ECPoint A = B^1
   2. ECPoint B as computed in the beginning.
   3. ECPoint C = (A \cdot Q)^y
   4. ECPoint D = Q
12. The ECDAA Issuer sends A, C to the authenticator. The authenticator still knows B and D
13. The authenticator checks that A, C ∈ G^1 and A \neq 1^G
14. The authenticator checks e(A, Y) \neq e(B, P^y)

NOTE

\[ e(A, Y) = e(B^{1/y}, P^y) = e(B, P^y) = e(B, P^y); \]
15. and the authenticator checks e(C, P^y) \neq e(A \cdot D, X)

NOTE

\[ e(C, P^y) = e((A \cdot Q)^y, P^y); e(A \cdot D, X) = e(A \cdot Q, P^y) = e((A \cdot Q)^y, P^y) \]
16. The authenticator stores credential A, B, C, D

3.4.2 ECDAA-Join Split between Authenticator and ASM
NOTE
If this join is not in-factory, the value Q must be authenticated by the authenticator. Upon receiving this value, the ECDAA Issuer must verify that this authenticator did not join before.

1. The ASM asks the ECDAA Issuer for the B value of the credential.
2. The ECDAA Issuer chooses a nonce BigInteger m = RANDB(p)
3. The ECDAA Issuer computes the B value of the credential as \( B = H^C(m) \)
4. The ECDAA Issuer sends \((sc,yc) = H_{G1}(m)\) to the ASM.
5. The ASM forwards \((sc,yc)\) to the authenticator
6. The authenticator chooses and stores the private key BigInteger \( s_k = RANDB(p) \)
7. The authenticator re-computes \( B = (H(sc), yc) \)
8. The authenticator computes its ECDAA public key ECPoint \( Q = B^s \)
9. The authenticator proves knowledge of \( s_k \) as follows
   1. BigInteger \( r_1 = RANDB(p) \)
   2. ECPoint \( U^1 = B^{r_1} \)
   3. BigInteger \( c_2 = H(U^1 | P | Q | m) \)
   4. BigInteger \( n = RANDB(p) \)
   5. BigInteger \( c_1 = H(n | c_2) \)
   6. BigInteger \( s_1 = r_1 + c_1 \cdot s_k \)
10. The authenticator sends \( Q, c_1, s_1, n \) to the ASM, who forwards it to the ECDAA Issuer.
11. The ECDAA 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 ECDAA-Join is a one-time operation, unlinkability is not a concern for that.
12. The ECDAA Issuer verifies that \( Q \in G1 \) and verifies \( H(n | H(B^{s_1} \cdot Q^{-c_1} | P \cdot Q | m)) \equiv c_1 \).
13. The ECDAA Issuer creates credential \((A, B, C, D)\) as follows
   1. ECPoint \( A = B^{1/n} \)
   2. ECPoint \( B \) as computed in the beginning.
   3. ECPoint \( C = (A \cdot Q)^s \)
   4. ECPoint \( D = Q \)
14. The ECDAA Issuer sends \( A, C \) to the ASM. The ASM remembered \( B \) and \( D = Q \) from an earlier step.
15. The ASM checks that \( A, B, C, D \in G1 \) and \( \neq 1_G \)
16. The ASM checks \( e(A, Y) \equiv e(B, P^2) \)
17. and the ASM checks that \( e(C, P^2) \equiv e(A \cdot D, X) \)
18. The ASM stores \( A, B, C, D \) and sends \( A, C \) to the authenticator. The authenticator still knows \( B \) and \( D \).
19. The authenticator stores \( B, D \) and ignores further join requests.

NOTE
These values belong to the ECDAA secret keys \( s_k \). They should persist even in the case of a factory reset.

3.4.3 ECDAA-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 ECDAA Issuer for the B value of the credential.
2. The ECDAA Issuer chooses a nonce BigInteger m = RANDB(p).
3. The ECDAA Issuer computes the B value of the credential as \( B = H^C(m) \)
4. The ECDAA Issuer sends \((sc,yc) = H_{G1}(m)\) to the ASM.
5. 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 \( P^k \) in field unique) to the ASM.
   2. re-computes \( B = (H(sc), yc) \)
3. retrieves TPM Endorsement Key Certificate (EK-C) from the TPM
4. calls TPM2_Commit(keyhandle, P1) where keyhandle is the handle of the restricted key generated before (see above), P1 is set to (B.x,B.y), and s2 and y2 are set to B.x and B.y respectively. This call returns K, E, and ctr; where $K = B^k = Q$, $E = B^e$ is used as $U^1$ value.
5. computes BigInteger $c^2 = H(U||P||Q|m)$
6. calls TPM2_Sign(c^2, ctr), returning $s^1$, n.
7. computes BigInteger $c^1 = H(n||c^2)$
8. sends EK-C, TPMT_PUBLIC (including $Q$ in field unique), $c^1$, $s^1$, n to the ECDAA Issuer.

6. The ECDAA Issuer
1. verifies EK-C and its certificate chain. As a result the ECDAA 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; encrypt = 0; sign = 1.
4. examines the public key Q, i.e. it verifies that $Q \in G^1$
5. checks $H(n||H(B^{1/2} \cdot Q^{-1}||P||Q|m)) \equiv c^1$
6. generates the ECDAA credential $(A, B, C, D)$ as follows
   1. ECPoint $A = B^{1/y}$
   2. ECPoint $B$ as computed in the beginning.
   3. ECPoint $C = (A \cdot Q)^x$
   4. ECPoint $D = Q$
7. generates a secret (derived from a seed) and wraps the credential $A, B, C, D$ using that secret.
8. encrypts the seed using the public key included in EK-C.
9. 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.

10. sends the wrapped object including the credential from previous step to the ASM.
7. 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 ECDAA attestation 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.
8. 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 $A \neq 1_G$
3. checks $e(A, Y) \equiv e(B, P_k)$ and $e(C, P_k) \equiv e(A \cdot D, X)$
4. stores $A, B, C, D$

3.5 ECDAA-Sign

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

3.5.1 ECDAA-Sign Algorithm
This section is normative.

$(signature, KRD) = \text{EcdaaSign(String AppID)}$

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. $KRD = \text{BuildAndEncodeKRD}()$; // all traditional Registration tasks are here
2. $\text{BigNumber } l = RAND(p)$
3. $\text{ECPoint } R = A^l$
4. $\text{ECPoint } S = B^l$
5. $\text{ECPoint } T = C^l$
6. $\text{ECPoint } W = D^l$
7. BigInteger \( r = RAND(p) \)
8. ECPoint \( U = S' \)
9. BigInteger \( c2 = H(U|S|W|AppID|H(KRD)) \)
10. BigInteger \( n = RAND(p) \)
11. \( c = H(n \mid c2) \)
12. BigInteger \( s = r + c \cdot sk \mod p \)
13. signature = \((c, s, R, S, T, W, n)\)
14. return (signature, KRD)

3.5.2 ECDAA-Sign 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. BigNumber \( l = RAND(p) \)
   2. ECPoint \( R = A' \)
   3. ECPoint \( S = B' \)
   4. ECPoint \( T = C' \)
   5. ECPoint \( W = D' \)
2. The ASM sends \( l, AppID \) to the authenticator
3. The authenticator performs the following tasks
   1. \( KRD = BuildAndEncodeKRD(l); \) // all fractional Registration tasks are here
   2. ECPoint \( S' = B' \)
   3. ECPoint \( W' = D' \)
   4. BigInteger \( r = RAND(p) \)
   5. ECPoint \( U = S^r \)
   6. BigInteger \( c2 = H(U|S'|W'|AppID|H(KRD)) \)
   7. BigInteger \( n = RAND(p) \)
   8. \( c = H(n \mid c2) \)
   9. BigInteger \( s = r + c \cdot sk \mod p \)
10. Send \( c, s, KRD, n \) to the ASM
4. The ASM sets signature = \((c, s, R, S, T, W, n)\) and outputs (signature, KRD)

3.5.3 ECDAA-Sign 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. Only the TPM must be trusted for unforgeability (see [CCDLNU2017-DAA]).
```

Algorithm outline
1. The ASM randomizes the credential
   1. BigNumber \( l = RAND(p) \)
   2. ECPoint \( R = A' \)
   3. ECPoint \( S = B' \)
   4. ECPoint \( T = C' \)
   5. ECPoint \( W = D' \)
2. The ASM calls TPM2_Commit() with \( P1 \) set to \( S \) and \( s2, y2 \) empty buffers. The ASM receives the result values \( K, L, E = S' = U \) 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. The related private key might need to be protected with appropriate access control mechanisms, e.g. see section 8 of [UAFAuthnCommands].
4. The ASM calls TPM2_Certify() on the newly created key with ctr from the TPM2_Commit and \( E = U, S, W, AppID \) as qualifying data. The ASM receives signature value \( s \) and related nonce \( n \) and attestation block KRD (i.e. TPMS_ATTEST structure in this case).
5. BigInteger \( c2 = H(E|S|W|AppID|H(KRD)) \), using KRD as returned by the previous step.
6. The ASM computes: \( c = H(n \mid c2) \)
7. The ASM sets signature = (c, s, R, S, T, W, n) and outputs (signature, KRD)

3.6 ECDAA-Verify Operation

This section is normative.

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

boolean EcdaaVerify(signature, AppID, KRD, ModelName)

Parameters
- p: System parameter prime order of group G; (global constant)
- P: System parameter generator of group G; (global constant)
- signature: (c, s, R, S, T, W, n)
- 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 ∈ G, R ≠ 1G, and S ≠ 1G.
3. $H(n|H(S \cdot W^{-c}|S|W|AppID|H(KRD))) \equiv c$; fail if not equal
4. $e(R, Y) \equiv e(S, P)$; fail if not equal
5. $e(T, P) \equiv e(R \cdot W, X)$; fail if not equal
6. for (all sk’ on RogueList) do if $W \not\equiv S^{sk’}$ fail;
7. // perform all other processing steps for new credential registration
8. return true;

4. FIDO ECDAA Object Formats and Algorithm Details

This section is normative.

4.1 Supported Curves for ECDAA

Definition of G1

$B = A^y = P^b$
$D = Q^y = P^{akb} = B^k$
$S = B^l$ and $W = D^l$
$U = S^c$

$S^c \cdot W^{-c} = S^{c+ca} \cdot W^{-c} = U \cdot S^{ca} \cdot W^{-c} = U \cdot B^{ja} \cdot D^{-c} = U \cdot B^{ja} \cdot B^{-ca} = U$

$e(R, Y) \equiv e(S, P)$

$e(T, P) \equiv e(R \cdot W, X)$

$e(T, P) = e(C, P) = e(A^c \cdot Q^{b+iy}, P)$

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).
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 $E'(F_p^q)$ 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 $F(q)[X]/<X^2 + 1>$. We use angle brackets $<Y'>$ 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 $F_p^q$.

**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 [BarNae2006] [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] [BarNae2006]. This curve is identical to the P256 curve defined in [ISO15946-5] section C.3.5.

  - The values have been generated using $u = 73500851732176300289$.
  - Modulus $q = 11579208923731439678268856124474172405837587855761205198700409522629664518163$
  - Group order p = 1157920892373143967826885612447417240583595988840268584478579994295356117037
  - $p$ and $q$ have length of 256 bit each.
  - $b = 3$.
  - $P_{256} = (x=1, y=2)$
  - $b' = (a=3, b=3)$
  - $P_{256} = (x,y)$, with

- **TPM_ECC_BN_P638** is a curve of form $E(F(q))$, where $q$ is the field modulus [TPMv2-Part4]. This curve is identical to the P638 curve defined in [ISO15946-5] section C.3.5.

  - The values have been generated using $u = 365375408922443362829987474442054824230286209343$
  - Modulus $q = 6415939204630002382849232286981801117629789403238356671671968915555497239494951781$
  - Related order of the group $p = 6415939204630002382849232286981801117629789403238356671671968915555497239494951781$
  - $p$ and $q$ have length of 638 bit each.
  - $b = 257$
  - $P_{638} = (x=6415939204630002382849232286981801117629789403238356671671968915555497239494951781$
  - $b' = (a=771, b=1542)$
  - $P_{638} = (x,y)$, with

- **ECC_BN_P638** [DevScoDah2007] section 3 uses

  - The values have been generated using $u = 635375408922443362829987474442054824230286209343$
  - Modulus $q = 824340165643006797212713535031900388365717811811386228921167322412819029493183$
  - Related order of the group $p = 824340165643006797212713535031900388365717811811386228921167322412819029493183$
  - $p$ and $q$ have length of 638 bit each.
  - $b = 3$. 

---

Note: The field modulus $q$ is given in hexadecimal and its bit length is specified in the context of the discussion.
\[ P_{DSD_{P256}} = (1, 2) \]
\[ b' = (a=3, b=6) \]
\[ P_{DSD_{P256}} = (x, y), \text{ with} \]
\[ P_{DSD_{P256}}.x = (a=73481346555305118071940904527349790526214212698180576973201374397013567073039, b=2895546842622256383171634927293392194526387931861908127165887947997417463) \]
\[ P_{DSD_{P256}}.y = (a=363249105468576136812589781621624880886395445849596442201944473723702808000374339 \]
\[ \text{ECC\_BN\_ISP512 [ISO15946-5] section C.3.7 uses} \]
\[ \text{The values have been generated using} u=138919694570470098040331481282041523727 \]
\[ \text{Modulus} q = 134078079299425970995740249982058304372461533448751115804945274271475909988179584598115751660499291639750834285779043186149750164319950153126044364566323 \]
\[ \text{The related order of the group is} p = 1340780792994259709957402499820583043724615334487511158049452742714759099881680053891202040597072645741464456779393060484617566647832620560300743149 \]
\[ p \text{ and } q \text{ have length of 512 bit each.} \]
\[ b = 3 \]
\[ P_{ISO_{P512}} = (x=1,y=2) \]
\[ b' = (a=3,b=3) \]
\[ P_{ISO_{P512}} = (x, y), \text{ with} \]
\[ P_{ISO_{P512}}.x = (a=3094648157539090131026477120117259886222920557994037039545437079729084516315481514566156984245473190248967907724153072490467902779495072074156718085785269,b=3776690234788102103105760376648606783580475949014286707858034303487054633977311929555161718985244561452474412673836012873126926524076966265127900471529) \]
\[ P_{ISO_{P512}}.y = (a=75938726053347015000172324521027873580057326388141101285040372458452328752430917597485450360707709276915921115916255816324924295339525686777567132644242,b=91319950534941228532850586465664850288094505101523182488852795871110193875868586879204157166587093146239570503948181818223029048018761707065071476178333480) \]

**NOTE**
Spaces are used inside numbers to improve readability.

**Hash Algorithm \( H \)**

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

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 \text{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.

**Hash Algorithm \( H_{G1} \)**

Definition of \( H_{G1} \) (taken from [CheLi2013-ECDAA]):

\( H_{G1}: \{0, 1\}^* \rightarrow G1, \text{ where } G1 \text{ is an elliptic curve group } E: y^2 = x^3 + b \text{ over } GF(q) \text{ with cofactor } = 1. \text{ Given a message } m \in \{0, 1\}^*, \)

\( H_{G1} \) can be computed as follows:

**ECPoint p = HG1(String m)**

1. Set \( i = 0 \) be a 32-bit unsigned integer.
2. Compute \( x = H(\text{BigIntegerToB}(i,4) \mid m) \)
3. Compute \( z = x^3 + b \mod q \)
4. Compute \( y = \sqrt{z} \mod q. \text{ If } y \text{ does not exist, set } i = i+1, \text{ repeat step 2 if } i < 232, \text{ otherwise, report failure.} \)
5. Set \( y = \min(y, q - y) \)
6. return \( \text{ECPoint}(x, y) \)

**(String sc, BigNumber yc) = HG1_pre(String m)**

1. Set \( i = 0 \) be a 32-bit unsigned integer.
2. Compute \( x = H(\text{BigIntegerToB}(i,4) \mid m) \)
3. Compute \( z = x^3 + b \mod q \)
4. Compute \( y = \sqrt{z} \mod q. \text{ If } y \text{ does not exist, set } i = i+1, \text{ repeat step 2 if } i < 232, \text{ otherwise, report failure.} \)
5. Set \( y = \min(y, q - y) \)
6. Set \( \text{sc to BigIntegerToB}(i,4) \mid m. \)
7. Set \( yc \) to \( y \)
8. return \( (sc, yc) \)

The ASM on the FIDO User device platform can help the authenticator compute \( H_{G1}(m) \), yet the authenticator verifies the computation as follows:

Given \( m \), the ASM runs the above algorithm. For a successful execution, let \( sc = (i \mid m) \) and \( yc \) be the \( y \) value in the last step. The ASM sends \( sc \) and
yc to the authenticator. The authenticator computes HG1(m) = (H(sc), yc).

Given the value sc, the original message m can be recomputed by skipping the first 4 bytes.

4.2 ECDAA Algorithm Names

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ED256_2PM_ECC_BN_P256</td>
<td>curve, using SHA256 as hash algorithm H.</td>
</tr>
<tr>
<td>ED256-2_ECC_BN_PSD_P256</td>
<td>curve, using SHA256 as hash algorithm H.</td>
</tr>
<tr>
<td>ED512_ECC_BN_ISO9512</td>
<td>curve, using SHA512 as hash algorithm H.</td>
</tr>
<tr>
<td>ED638_TPM_ECC_BN_P638</td>
<td>curve, using SHA512 as hash algorithm H.</td>
</tr>
</tbody>
</table>

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 356 bytes for 256bit curves):

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Length (in Bytes)</td>
</tr>
<tr>
<td>UINT8[]</td>
<td>ECDAA_Signature_c</td>
</tr>
<tr>
<td></td>
<td>The c value, c = H(n l c2) as returned by EcdaaSign encoded as byte string according to BigNumberToB. Where</td>
</tr>
<tr>
<td></td>
<td>• c2=H(U</td>
</tr>
<tr>
<td></td>
<td>• U = S', with r = RAND(p) computed by the signer.</td>
</tr>
<tr>
<td></td>
<td>• KRD is the the entire to-be-signed object (e.g. TAG_UAFV1_KRD in the case of FIDO UAF).</td>
</tr>
<tr>
<td></td>
<td>• S = B^l, with l = RAND(p) computed by the signer and B = A^c computed in the ECDAA-Join Algorithm.</td>
</tr>
<tr>
<td>UINT8[]</td>
<td>ECDAA_Signature_s</td>
</tr>
<tr>
<td></td>
<td>The s value, s=r + c * sk (mod p), as returned by EcdaaSign encoded as byte string according to BigNumberToB. Where</td>
</tr>
<tr>
<td></td>
<td>• r = RAND(p), computed by the signer at FIDO registration (see 3.5.2 ECDAA-Sign Split between Authenticator and ASM).</td>
</tr>
<tr>
<td></td>
<td>• p is the group order of G1</td>
</tr>
<tr>
<td></td>
<td>• sk: is the authenticator's attestation secret key, see above</td>
</tr>
<tr>
<td>UINT8[]</td>
<td>ECDAA_Signature_n</td>
</tr>
<tr>
<td></td>
<td>The Nonce value n, as returned by EcdaaSign encoded as byte string according to BigNumberToB.</td>
</tr>
<tr>
<td>UINT8[]</td>
<td>ECDAA_Signature_R</td>
</tr>
<tr>
<td></td>
<td>R = A^l; computed by the ASM or the authenticator at FIDO registration; encoded as byte string according to ECPointToB. Where</td>
</tr>
<tr>
<td></td>
<td>• l = RAND(p), i.e. random number 0≤l≤p. Computed by the ASM or the authenticator at FIDO registration.</td>
</tr>
<tr>
<td></td>
<td>• R = A^l denotes the scalar multiplication (of scalar l) of a curve point A.</td>
</tr>
<tr>
<td></td>
<td>• A has been provided by the ECDAA Issuer as part of ECDAA-Join: A = B^1/y, see 3.4.1 ECDAA-Join Algorithm.</td>
</tr>
<tr>
<td></td>
<td>• Where p is a system value, injected into the authenticator and y is part of the ECDAA Issuer private key isk=(x,y).</td>
</tr>
<tr>
<td>UINT8[]</td>
<td>ECDAA_Signature_S</td>
</tr>
<tr>
<td></td>
<td>S = B^l; computed by the ASM or the authenticator at FIDO registration encoded as byte string according to ECPointToB. Where</td>
</tr>
<tr>
<td></td>
<td>• B has been provided by the ECDAA Issuer on Join: B = HG1(m) = (H(sc), yc), see 3.4.1 ECDAA-Join Algorithm.</td>
</tr>
<tr>
<td>UINT8[]</td>
<td>ECDAA_Signature_T</td>
</tr>
<tr>
<td></td>
<td>T = C^x; computed by the ASM or the authenticator at FIDO registration encoded as byte string according to ECPointToB. Where</td>
</tr>
<tr>
<td></td>
<td>• C = (A * Q)^x, provided by the ECDAA Issuer on Join</td>
</tr>
<tr>
<td></td>
<td>• x is a components of the ECDAA Issuer private key, isk=(x,y).</td>
</tr>
<tr>
<td></td>
<td>• Q is the authenticator public key</td>
</tr>
<tr>
<td>UINT8[]</td>
<td>ECDAA_Signature_W</td>
</tr>
<tr>
<td></td>
<td>W = D^x; computed by the ASM or the authenticator at FIDO registration encoded as byte string according to ECPointToB. Where</td>
</tr>
<tr>
<td></td>
<td>• D = Q is computed by the ECDAA Issuer at Join (see 3.4.1 ECDAA-Join Algorithm).</td>
</tr>
</tbody>
</table>
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 \(\text{Sig}2=(R, S, T, W)\) 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 Concatenation

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 KRID. 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].

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“alg”</td>
<td>i.e. used with JWS.</td>
</tr>
<tr>
<td>JOSE Implementation Requirements</td>
<td>Optional</td>
</tr>
<tr>
<td>Change Controller</td>
<td>FIDO Alliance, Contact Us</td>
</tr>
<tr>
<td>Specification Documents</td>
<td>[FIDO-DAA-Security-Proof]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“alg”</td>
<td>i.e. used with JWS.</td>
</tr>
<tr>
<td>JOSE Implementation Requirements</td>
<td>Optional</td>
</tr>
<tr>
<td>Change Controller</td>
<td>FIDO Alliance, Contact Us</td>
</tr>
</tbody>
</table>
A. References

A.1 Normative references

[ECDSA-ANSI]

[RFC2119]

[RFC7515]

[ISO15946-5]

[FIDO-DAA-Security-Proof]
Algorithm Analysis Document(s) Sections 3. FIDO ECDAA Attestation and 4. FIDO ECDAA Object Formats and Algorithm Details of FIDO ECDAA Algorithm

Algorithm Name
“ED638”

Algorithm Description

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 ECDAA Attestation and 4. FIDO ECDAA Object Formats and Algorithm Details of FIDO ECDAA Algorithm.

Algorithm Analysis Document(s) Sections 3. FIDO ECDAA Attestation and 4. FIDO ECDAA Object Formats and Algorithm Details of FIDO ECDAA Algorithm.

A.2 Informative references

[ANZ-2013]

[Arthur-Challener-2015]

[BFGSW-2011]

[BazNae-2006]

[BricCamche2004-DAA]

[FIDO-DAA-Security-Proof]
Algorithm Analysis Document(s) Sections 3. FIDO ECDAA Attestation and 4. FIDO ECDAA Object Formats and Algorithm Details of FIDO ECDAA Algorithm

Algorithm Name

[FIDOEcdaaAlgorithm]
R. Lindemann; J. Camenisch; M. Drivers; A. Edgington; A. Lehmann; R. Urban. FIDO ECDAA Algorithm. Implementation Draft. URL: https://fidoalliance.org/specs/fido-ecdaa-1.2-20171128/fido-ecdaa-algorithm-v1.2-20171128.html

[FIDO-Glossary]
R. Lindemann; D. Baghdasaryan; B. Hill; J. Hodges. FIDO Technical Glossary. Implementation Draft. URL: https://fidoalliance.org/specs/fido-ecdaa-1.2-20171128/fido-glossary-v1.2-20171128.html

[ISO15946-5]

[RFC7515]

[RFC7518]

[TPMv2-Part1]

[TPMv2-Part2]

[UAAuthenticatorCommands]
D. Baghdasaryan; J. Kemp; R. Lindemann; R. Sasson; B. Hill. FIDO UAF Authenticator Commands v1.0 Implementation Draft. URL: https://fidoalliance.org/specs/fido-uaa-v1.2-20171128/fido-uaa-cmds-v1.2-20171128.html

[XZY-2014]
Li X; Kang Yang; Zhoufen Zhang; Dengguo Feng. DAA-Related APIs in TPM 2.0 Revisited, in T. Holz and S. Ioannidis (Eds) 2014. URL: http://www.trustedcomputinggroup.org/files/static_page_files/8C56AE3E-1A4B-B294-D055-B08C6F2F0/TPM%20v1.2-%20Architecture%2001_16.pdf