Intel® Enhanced Privacy ID Specifications

Document Owner: Jiangtao Li

Contributors: Igor Astakhov, Ernie Brickell, Xiaozhu Kang, Sergey Khlystov, Sergey Kirillov, Michael Kounavis, Marcin Maka, Chris McConnell, Daniel Nemiroff, Jody Pfotenhauer, Michal Piotrewicz, Keith Shippy, Andrey Somsikov, Adam Thomas, William Stevens, Peter Tang, Marc Valle, Matthew Wood

Revision 1.0

June 23, 2009

Intel Confidential

Disclaimers

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppels or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Any prototypes may contain design defects or errors known as errata that may cause the product to deviate from published specifications. Currently characterized errata are available on request.

*Third-party brands and names are the property of their respective owners

Revisions

Revision	Date	Description
0.1	07/23/2007	Initial draft
0.2	08/06/2007	Add introduction section
0.3	08/07/2007	Add arithmetic for finite fields
0.4	08/08/2007	Add arithmetic for Tate pairing
0.5	08/21/2007	Update efficient multiplication method for finite fields
0.6	08/26/2007	Update protocol descriptions
0.7	03/28/2008	• Update sign and verify algorithms • Update protocol performance enhancement • Add key provisioning method
0.8	04/28/2008	• Update sign and verify algorithms • Finalize elliptic curve parameters
0.8.5	05/09/2008	Add method for signature pre-computation
0.9	09/09/2008	• Update Tate pairing algorithm • Update sign algorithm
1.0	06/23/2009	• Add more math descriptions • Add serialization • Add details on private-key based revocation

Table of Contents

1 Introduction

This document specifies the Intel® Enhanced Privacy ID (INTEL® EPID) scheme. The goal of the Intel® Enhanced Privacy ID scheme is to enable one to prove that he is a member in good standing in a group without revealing any information about his identity. In an INTEL® EPID group, there is a group public key. Each member of the group has a unique private key that he uses to prove membership in the group.

In the INTEL® EPID scheme, there are three types of entities: issuer, platforms, and verifiers. The issuer is the entity that issues a unique INTEL® EPID private key to each member of the group. The platform is an entity who is trying to prove membership in the group. If the platform is indeed a member in the group in good standing (i.e., the platform has a valid INTEL® EPID private key and has not been revoked), the proof should be successful. The verifier is the entity who is trying to establish whether the platform is a member of the group. In our context, the issuer is Intel, the platform is an Intel hardware device (e.g., PCH, CPU, TPM, or LRB), and the verifier could be a software on the host, a server on the Internet, or another hardware device.

For each INTEL® EPID group, there is a group public key and a group issuing private key (master key). The group issuing private key is used for generating a unique private key for each group member. The issuing private key should be kept securely by the issuer. Each platform can use its INTEL® EPID private key to digitally sign a message, and the resulting signature is called an INTEL® EPID signature. The verifier can use the group public key to verify the correctness of an INTEL® EPID signature, i.e., to verify that the signature was indeed created by a platform with a valid INTEL® EPID private key. The INTEL® EPID signature however does not reveal any information about which unique private key was used to create the signature.

Note that the issuer does not keep a database of all the INTEL® EPID private keys that he issues. Given an INTEL® EPID signature, even the issuer cannot identify which group member signed the signature.

The INTEL® EPID scheme must handle revocation. The issuer can revoke a platform based on the platform's INTEL® EPID private key. This revocation method is called private-key based revocation and is used when a platform's private key gets extracted from the secure storage of the platform and published widely. We use PRIV-RL to denote the revocation list corresponding to this method. For this revocation method, it is the verifier's responsibility to check whether the platform has been revoked in PRIV-RL.

Our INTEL® EPID scheme has two components: one is key provisioning scheme; the other is proof of membership scheme. For key provisioning, there are

1. Issuer Setup: In this procedure, the issuer creates a group public key and a group issuing private key. The issuer publishes and distributes the group public key to everyone (e.g., every platform and every verifier).
2. Issue: The issuer assigns a unique INTEL® EPID private key to each platform.
3. Platform Initial Setup: If the compressed INTEL® EPID private key is stored in the fuses of the device, the platform needs to perform this step to decompress the INTEL® EPID private key. The platform can also perform some initial pre-computations.

For proof of membership scheme, there are the following two operations:

1. Sign: The platform generates an INTEL® EPID signature on a message using its INTEL® EPID private key.
2. Verify: The verifier verifies the correctness of an INTEL® EPID signature using the group public key. The verifier also verifies that the creator of the INTEL® EPID signature has not been revoked in PRIV-RL.

A simple usage example of proof of membership is as follows: the verifier sends a challenge message to the platform. The platform signs the message using his INTEL® EPID private key and sends the INTEL® EPID signature back to verifier. The verifier verifies the signature and verifies that the member has not been revoked. We may use the sign and verify operations as part of a key agreement protocol.

In the INTEL® EPID scheme, the computational cost of the verify algorithm grows linearly on the size of the revocation list. In general, we do not expect a large number of platforms that need to be revoked. However, to prevent the size of the revocation list from becoming too large, the issuer creates multiple INTEL® EPID groups. For example, suppose there are 10 million platforms. The issuer can create 100 INTEL® EPID groups and each group has 100,000 group members. To issue an INTEL® EPID private key for a platform, the issuer first chooses an INTEL® EPID group for him, and then generates an INTEL® EPID private key in that group. To conduct proof of membership, the platform needs to first reveal which INTEL® EPID group he belongs to, and then proves that he is a valid member of that group. Of course, the issuer needs to maintain a revocation list for each INTEL® EPID group. Note that the group size of an INTEL® EPID group could vary depending on the setting.

In an INTEL® EPID scheme, there are two options for generating an INTEL® EPID signature: one is random base option; the other is name base option. In random base option, the INTEL® EPID signature is truly anonymous (i.e., two INTEL® EPID signatures generated by the same platform are totally different and cannot be linked together). In name base option, the verifier sends his basename to the platform, and the platform creates an INTEL® EPID signature based on that basename. Given a basename, the platform always uses the same pseudonym; therefore, all his signatures with that basename are linkable. The name base option is useful when a platform only interacts with a verifier for only once or a few times, while the verifier wants to prevent a corrupted platform from abusing the anonymity.

2 Notations and Data Formats

In the Intel® Enhanced Privacy ID (INTEL® EPID) scheme, we use the notion of an algebra group extensively. Note that the group we refer in this section is not an INTEL® EPID group but a group in mathematics. A group is a set of elements with certain operations. Each group has its parameters and a number of functions associated with this group. For example, there is a multiply function in a group. Given two elements a, b of a group, the multiply function outputs a·b, where a·b is another element of the group. Each group has a generator, a special element in the group.

In the rest of this specification, we use big integers (integers more than 32-bit or 64-bit) extensively. When we store those integers in memory, fuses, or transfer over the network, we always use big endian format unless explicitly specified.

In the INTEL® EPID protocol, we use random number generator (RNG) or pseudorandom number generator (PRNG) for choosing random elements in a group.

Unless specifically noted otherwise, one the following RNG or PRNG: (1) PRNG based on a design described in ANSI X9.31, (2) PRNG defined in FIPS PUB 186-2, or (3) RNG or PRNG of equal or higher quality that passes the tests described in NIST Special Publication 800-22.

2.1 Basic Notations

In this document, we use the following terminologies:

DSA	Digital Signature Algorithm
EC-DSA	Elliptic Curve DSA
ISK	Intel Signing Key
IVK	Intel Verification Key
ISV	Independent Software Vendor
PCH	Platform Controller Hub
PRIV-RL	Private-key based Revocation

We use the following notations:

p	A large prime (256-bit*)
q	A large prime (256-bit*)
p’	A large prime (256-bit)
q’	A large prime (256-bit)
gid	AN INTEL® EPID group ID (32-bit)
n1	Number of entries in PRIV-RL (32-bit)
G1	A special group of p elements
G2	A special group of p elements
G3	A special group of p’ elements
GT	A special group of p elements
g1	Generator of G1 (512-bit)
g2	Generator of G2 (1536-bit)
g3	Generator of G3 (512-bit)
sver	INTEL® EPID version number (16-bit unsigned integer)
blobid	Blob identifier (16-bit unsigned integer)
RLver	Revocation list version number (32-bit unsigned integer)
slen	A security parameter. In this version of INTEL® EPID, slen = 80.

Hash(m)	A hash of arbitrary-length message m (256-bit). Unless specified otherwise, we use SHA-256 as the hash function and the output is used as a big endian number.
a || b || c || …	Concatenate a, b, c, … into a single message
0x	Indicator of hexadecimal representation of an octet or an octet string. For example, "0x 25" denotes the octet of hexadecimal value 25. "(0x) 25 4a 0e" denotes the string of three octets with hexadecimal value 25, 4a, and 0e, respectively.

* Note that the actual size of p and q may be smaller than 256-bit. In this case, we simply pad p and q with 0s in the most significant bits to make them 256-bit.

2.2 Functions

In this section we describe the functions of a group without giving the details on how to implement them. The implementation of these functions is described in Section 7. Given a group G, we have the following functions.

1. G.mul(P, Q). This program takes two group elements P and Q in G. It outputs R = P·Q, another element in G.
2. G.exp(P, b). This program takes a group element P in G and a positive integer b. It outputs R = P^b, another element in G.
3. G.multiexp(P[0], b[0], …, P[m-1], b[m-1]). This program takes m group elements P[0], …, P[m-1] in G and m positive integers b[0], …, b[m-1], where m is a small positive integer. It outputs R = $P[0]^{b[0]} \cdot \ldots \cdot P[m-1]^{b[m-1]}$, another element in G. This function is equivalent to compute $P[0]^{b[0]}$, …, $P[m-1]^{b[m-1]}$, and then multiply them together. This function is only available for G1, G3, and GT.
4. G.getRandom(). This program outputs a random element of G that is chosen uniformly at random. The RNG or PRNG used for choosing the random element must be robust (see the beginning of this section). This function is only available for G1, G3, and GT.
5. G.inGroup(P). This program takes a value P as input. If P is indeed a group element of G, it outputs true, otherwise, it outputs false.
6. G.hash(m). This program hashes an arbitrary message m to an element in G. This program takes a message m as input. It outputs an element of G. This function is only available for G1 and G3.
7. G.makePoint(x). This program takes an integer x as input. It outputs R, an element of G. R is constructed based on the x value. This function is only available for G1 and G3.
8. G.inverse(P). This program takes a group element P as input. It outputs R, the inverse of P. R is an element of G.
9. G.isEqual(P, Q). This program takes two group elements P and Q as input. If P = Q, it outputs true, otherwise, it outputs false.
10. G.isIdentity(P). This program takes a group element P as input. It outputs true if P is the identity element of G, otherwise, it outputs false.

11. pairing(P, Q). This program takes an element P in G1 and an element Q in G2 as input. It outputs the R = e(P, Q), an element in GT.

A note on G.exp() and G.multiexp(): Let p be the total number of elements in G. To compute G.exp(P, b) where b could be an arbitrary (positive or negative) integer, we always do the following steps: (1) first compute b' = b mod p, and then (2) compute G.exp(P, b'). Now when we implement G.exp(P, b'), we only need to consider the case where b' is an integer between [0, p-1]. Same methodology applies to the use and implementation of G.multiexp().

2.3 Data Format and Group Parameters

In this specification, we use big integer or multi-precision integer intensively. We have four mathematical groups G1, G2, G3, and GT. The groups G1, G2, G3 are elliptic curve groups. The group GT is a finite field group. Most of the operations of the INTEL® EPID protocol are based on these groups. We now give an overview of data structure of each element in G1, G2, G3, and GT.

- Element of G1: takes the format of (x, y) where x, y are big integers between [0, q-1].
- Element of G2: takes the format of (x[0], x[1], x[2], y[0], y[1], y[2]), where x[i] and y[i] are big integers between [0, q-1]
- Element of G3: takes the format of (x, y) where x, y are big integers between [0, q'-1].
- Element of GT: It takes the format of (x[0], x[1], …, x[5]), where x[i] is a big integer between [0, q-1].

Note that for elliptic curve groups G1, G2, and G3, there is a special element O, called the point of infinity. We treat this element separately. In general, the point of infinity O should not appear in any INTEL® EPID private keys or INTEL® EPID signatures.

We have a summary in below

Group Elements	Size
G1	512-bit
G2	1536-bit
G3	512-bit
GT	1536-bit

We now present the parameters of each group.

- G1.param: (p, q, h, a, b, g1)
  - p (256-bit), a prime
  - q (256-bit), a prime
  - h (32-bit), a small integer, also denoted as cofactor

  - a (256-bit), an integer between [0, q-1]
  - b (256-bit), an integer between [0, q-1]
  - g1 (512-bit), a generator (an element) of G1
- G2.param:
  - p (256-bit), same as in G1
  - q (256-bit), same as in G1
  - a (256-bit), same as in G1
  - b (256-bit), same as in G1
  - coeff (768-bit), the coefficients of an irreducible polynomial
    - coeff[0], coeff[1], coeff[2]: 256-bit integers between [0, q-1]
  - qnr (256-bit), a quadratic non-residue (an integer between [0, q-1])
  - orderG2(768-bit), the total number of points in G2 elliptic curve
  - g2 (1536-bit), a generator (an element) of G2
- G3.param: (p', q', h', a', b', g3)
  - p' (256-bit), a prime
  - q' (256-bit), a prime
  - h' (32-bit), a small integer, usually 1, also denoted as cofactor'
  - a' (256-bit), an integer between [0, q'-1]
  - b' (256-bit), an integer between [0, q'-1]
  - g3 (512-bit), a generator (an element) of G3
  - Note that we use the standard NIST P-256 curve as the parameters of G3
- GT.param: (q, coeff, qnr)
  - q (256-bit), same as in G1
  - coeff (768-bit), same as in G2
  - qnr (256-bit), same as in G2
- The overall parameters are: (p, q, h, a, b, coeff, qnr, orderG2, p', q', h', a', b', g1, g2, g3). The size of the INTEL® EPID public parameters of G1, G2, G3, and GT is 6464 bits or 808 bytes. We use param(G1, G2, G3, GT) to denote the overall INTEL® EPID parameters.

2.4 Format of Public Key, Private Key, Signature, and Revocation List

In the INTEL® EPID scheme, we have the following data structure of a group public key, issuing private key, private key, signature, revocation list, pre-computed signature, platform pre-computation blob, and verifier pre-computation blob.

- Intel Verification Key (IVK): (C)

  - IVK is an EC-DSA public key used for verifying INTEL® EPID group certificates, parameters, and revocation lists
  - We use the NIST standard P-256 elliptic curve as the parameters. The parameters are exact the same as the parameters of G3. We do not include the parameters of the elliptic curve as part of IVK, as the parameters are standard and publicly available.
  - C: an element in G3
  - total size: 512-bit
- Intel Signing Key (ISK): (d)
  - ISK is the EC-DSA private key that corresponds to IVK
  - d: 256-bit integer
  - total size: 256-bit
- Group certificate: (gid, h1, h2, w, sig)
  - gid: group ID
  - h1: an element in G1
  - h2: an element in G1
  - w: an element in G2
  - The non-signature portion (gid, h1, h2, w) of a group certificate is referred as group public key in the rest of this specification.
  - sig: 512-bit EC-DSA signature on group public key signed by the issuer using ISK
  - total size: 3104-bit
- Issuing private key (gid, u1, u2, gamma)
  - gid: group ID
  - u1: an integer between [0, p-1]
  - u2: an integer between [0, p-1]
  - gamma: an integer between [0, p-1]
  - total size: 800-bit
- INTEL® EPID parameters certificate: (param(G1, G2, G3, GT), sig)
  - param(G1, G2, G3, GT): INTEL® EPID parameters
  - sig: 512-bit EC-DSA signature on the INTEL® EPID parameters signed by the issuer using ISK
  - total size: 6976-bit
- Private key: (gid, A, x, y, f)
  - gid: group ID
  - A: an element in G1
  - x: an integer between [0, p-1]

  - y: an integer between [0, p-1]
  - f: an integer between [0, p-1]
  - total size: 1312-bit
- Compressed private key: (gid, A.x, seed)
  - gid: group ID
  - A.x: an integer between [0, q-1]
  - seed: a 256-bit string
  - total size: 544-bit
- Signature: (B, K, T1, T2, c, nd, sx, sy, sf, sa, sb, salpha, sbeta)
  - B, K: elements of G3
  - T1, T2: elements of G1
  - c: a 256-bit integer
  - nd: an 80-bit integer
  - sx, sy, sa, sb, salpha, sbeta: integers between [0, p-1]
  - sf: a 593-bit integer (round up to 600-bit)
  - total size: 4520-bit
- Pre-computed signature: (B, K, T1, T2, a, b, rx, ry, rf, ra, rb, ralpha, rbeta, R1, R2, R3, R4)
  - B, K, R3: elements of G3
  - T1, T2, R1, R2: elements of G1
  - R4: element of GT
  - a, b, rx, ry, ra, rb, ralpha, rbeta: integers between [0, p-1]
  - rf: a 592-bit integer
  - total size: 7760-bit
- Platform pre-computation blob: (e12, e22, e2w, ea2)
  - e12, e22, e2w, ea2: elements of GT
  - total size: 6144-bit
- Verifier pre-computation blob: (gid, e12, e22, e2w)
  - gid: group ID
  - e12, e22, e2w: elements of GT
  - total size: 4640-bit
- Private-key based revocation list PRIV-RL: (gid, RLver, n1, f[1], …, f[n1], sig)
  - gid: group ID
  - RLver: revocation list version number (32-bit)
  - n1: number of entries in PRIV-RL (32-bit)

- f[i]: integer between [0, p-1], for i=1, …, n1.
- sig: 512-bit EC-DSA signature on the non-signature portion of the revocation list signed by the issuer using ISK
- total size: (96 + 256 · n1 + 512)-bit

We will explain how signatures are computed on PRIV-RL, INTEL® EPID group certificate, and INTEL® EPID parameters certificate in the next subsection.

2.5 Data Serialization

Given INTEL® EPID data (e.g., INTEL® EPID private key, signature), often times we need to serialize the data so that it can be stored in the memory or in the disk, or transferred over the network to other entities. To make sure that the receiver of the data can read and parse the data correctly and that the parsing software is compatible with future generations of INTEL® EPID, we define the following simple serialization method.

1. Suppose INTEL® EPID data has a format of (a, b, c, …).
2. The serialized INTEL® EPID data is (sver || blobid || a || b || c || …) where,
   a. sver is 16-bit INTEL® EPID version number. sver = 1 for this specification.
   b. blobid is 16-bit blob ID of the data type, defined in the following table.
   c. The 4-byte (sver || blobid) is also called a serialization header.
   d. All the elements are converted to big endian format and concatenated into a single message.

blobid	Data type	Size in bits
0	INTEL® EPID parameters certificate	7008
1	*“Reserved for future use”*	*N/A*
2	Issuing private key	832
3	Private key	1344
4	Compressed private key	576
5	Signature	4552
6	Pre-computed signature	7792
7	Platform pre-computation blob	6176
8	Verifier pre-computation blob	4672
9	Intel verification key	544
10	Intel signing key	288
11	*“Reserved for future use”*	*N/A*
12	Group certificate	3136

13	Private-key based revocation list	640 + 256·n1
14	*"Reserved for future use"*	*N/A*

Table 2-1: Table of blobid values and corresponding total data size in bits including the serialization header

Given serialized data, we can de-serialize the data as the following:

1. First read the first 16-bit as sver. If sver ≠ 1, discard the data.
2. Then read the next 16-bit as blobid. Look up the blobid table and identify the type of the data (whether it is a private key or a signature, etc).
3. As we already know the exact data size and data format in Table 2-1 and Section 2.4, we can parse the data properly.

Example: Give an INTEL® EPID private key (gid, A, x, y, f).

1. The serialization is (1 || 3 || gid || A || x || y || f).
2. The size of the serialized private key is 1344 bits (1312-bit + 32-bit serialization header).
3. To de-serialize a private key, we first read 16-bit sver and 16-bit blobid, and then read 32-bit gid, 512-bit A, 256-bit x, 256-bit y, 256-bit f sequentially.

Recall that in G1, G2, G3, there is a special element O (point at infinity). We can serialize O as zero, e.g., encode O as [0, 0] in G1 and G3, or [0, 0, 0, 0, 0, 0] in G2. To de-serialize an element in G1 or G2 or G3, we first check whether it is zero, if so, we parse the element as O. Note that, it is acceptable to encode O as zero in our setting, because the elliptic curves we choose in INTEL® EPID parameters have the property that [0, 0] is not a valid point on the elliptic curves.

If the sver value is larger than that supported by the implementation, the blob should not be processed and an error should be reported.

2.6 Signatures Computed using ISK

Now we describe how the signatures are computed in group certificate, INTEL® EPID parameters certificate, and PRIV-RL. Let ISK be the Intel signing key. To compute the signature in a group certificate, an INTEL® EPID parameters certificate, or a PRIV-RL, the issuer first serializes the non-signature part, then computes the SHA-256 hash value over the serialized data, and finally computes an EC-DSA signature over the hash value using ISK.

Example: To create a group certificate for group public key (gid, h1, h2, w).

1. The issuer first serializes non-signature part of the group certificate by adding a serialization header (sver || blobid), where sver = 1 and blobid = 12.
2. The issuer computes h = Hash(sver || blobid || gid || h1 || h2 || w).
3. The issuer computes sig = EC-DSA-Sign(ISK, h), an EC-DSA signature on h.
4. The serialized group certificate is (sver || blobid || gid || h1 || h2 || w || sig).

3 Key Provisioning

In this section, we describe the INTEL® EPID key provisioning. This section has four parts: INTEL® EPID public key generation and provisioning, fuses based INTEL® EPID private key provisioning, generic INTEL® EPID private key provisioning, and verifier pre-computation. We intentional omit of details of elliptic curve operations. We shall describe the detailed elliptic curve operations in the later sections. In the rest of this document, we often convert between binary strings and integers. When we convert between binary strings and integers, we always use the big endian format unless explicitly specified.

In all cases, one particular private key may not be embedded into more than one instance of a platform. The platform should have a good protection on the private key according to the associated license agreement. Failure to protect the private key may result in revocation of the private key.

3.1 Public Key Generation and Provisioning

In this step, the issuer (Intel) performs the following steps to generate INTEL® EPID public keys:

1. The issuer has an EC-DSA key pair (IVK, ISK). This EC-DSA key is used for signing INTEL® EPID group certificate, INTEL® EPID parameters, and revocation list.
2. The issuer loads param(G1, G2, G3, GT), the INTEL® EPID parameters. Note that the INTEL® EPID parameters are the same for every INTEL® EPID group with regarding to a particular INTEL® EPID version.
3. The issuer signs the param(G1, G2, G3, GT) using ISK and produces an INTEL® EPID parameters certificate.
4. Assume the issuer needs to create n groups. For i=1, …, n, the issuer creates the i-th group as follows:
    a. The issuer picks a 32-bit unsigned integer gid[i] as the group ID. The issuer needs to make sure gid[i] has not been used before.
    b. The issuer chooses three random integers u1[i], u2[i], gamma[i] between [2, p-1].
    c. The issuer computes h1[i] = G1.exp(g1, u1[i]).
    d. The issuer computes h2[i] = G1.exp(g1, u2[i]).
    e. The issuer computes w[i] = G2.exp(g2, gamma[i]).
    f. The group public key for the INTEL® EPID group is (gid[i], h1[i], h2[i], w[i]).
    g. The group issuing private key for the INTEL® EPID group is (gid[i], u1[i], u2[i], gamma[i]).
    h. The issuer signs the INTEL® EPID group public key using ISK and produces an INTEL® EPID group certificate.
5. The issuer publishes the INTEL® EPID parameters certificate and all the INTEL® EPID group certificates online or distributes to ISVs. The issuer may also embed the INTEL® EPID parameters certificate and INTEL® EPID group

certificates in the firmware or driver of each hardware device. Every hardware device and every verifier can verify the correctness of the INTEL® EPID parameters and the INTEL® EPID public keys using the IVK.

3.2 Fuses based Private Key Provisioning

This section describes fuses based INTEL® EPID private key provisioning that is specially designed for PCH and LRB. This provisioning method is used for the scenarios where the hardware device has limited fuses to store the INTEL® EPID private key and we need to compress the INTEL® EPID private key as much as possible.

3.2.1 Private Key Generation and Compression

In this process, the issuer generates a unique INTEL® EPID private key for each hardware device. The issuer then compresses the INTEL® EPID private key and stored the compressed private key in the fuses of each hardware device. The issuer does the following steps. In the following steps, we use temporary integers t1, t2, and t3.

1. Let FKey be the unique 256-bit fuse key of the hardware device. The issuer chooses a random 256-bit FKey and then derives a 256-bit INTEL® EPID random seed value from FKey. The derivation function is platform specific and is out of the scope of this specification.
2. The issuer chooses an INTEL® EPID group for the device. Let gid be the chosen group ID. Let (gid, h1, h2, w) be the group public key and (gid, u1, u2, gamma) be the group issuing private key.
3. The issuer computes

   x = (Hash(seed || "(0x) 00 45 43 43 2d 53 61 66 65 49 44") ||
   Hash(seed || "(0x) 01 45 43 43 2d 53 61 66 65 49 44")) mod p,

   y = (Hash(seed || "(0x) 02 45 43 43 2d 53 61 66 65 49 44") ||
   Hash(seed || "(0x) 03 45 43 43 2d 53 61 66 65 49 44")) mod p,

   f = (Hash(seed || "(0x) 04 45 43 43 2d 53 61 66 65 49 44") ||
   Hash(seed || "(0x) 05 45 43 43 2d 53 61 66 65 49 44")) mod p.
4. The issuer computes integer t1 = (1 + u1 · f + u2 · y) mod p.
5. The issuer computes integer t2 = (gamma + x) mod p.
6. If t1 = 0 or t2 = 0, the issuer discards FKey, chooses a new FKey, and starts over from step 1. The probability of discarding FKey is extremely small.
7. The issuer computes integer t3 = inverse(t2) mod p, the inverse of t2 modulo p. Refer to Section 7.1 for the inverse operation over a prime field.
8. The issuer computes integer t1 = (t1 · t3) mod p.
9. The issuer computes A = G1.exp(g1, t1). A is an element of G1.
10. The value (gid, A, x, y, f) is the INTEL® EPID private key of the device.
11. The issuer now compresses the private key as follows. Let A = (A.x, A.y) where A.x and A.y are two 256-bit integers. The compressed INTEL® EPID

private key is (gid, A.x, seed). Note that A cannot be the point of infinity, given the step 6 above.

12. The issuer blows (gid, A.x, FKey) to the fuses of the device. Note that, in practice, the issuer may encrypt A.x and FKey or pad an integrity check value on the compressed private key before blowing the key into the fuses. However, this is out of the scope of this specification.
13. The issuer erases the fuse key FKey and the INTEL® EPID private key (gid, A, x, y, f) from its memory.

For math readers, the key generation steps are as follows:

1. The issuer derives integers x, y, f from seed as in the step 3 above.
2. The issuer computes $A = g1^{(1 + u1 \cdot f + u2 \cdot y) / (gamma + x)}$.
3. The value (gid, A, x, y, f) is the INTEL® EPID private key for the device.

3.2.2 Private Key Decompression and Initial Setup

A hardware device performs the following initial setup process as follows. In this process, the device derives the compressed INTEL® EPID private key from its fuses, then performs a decompression operation, and finally verifies the correctness of the INTEL® EPID private key. The device also performs the one-time pre-computation to calculate platform pre-computation blob needed for the sign algorithm. In the end, the device stores the INTEL® EPID private key and the platform pre-computation blob in its secure storage (e.g., using ME's blob service).

In the following process, we use temporary variables t1 (an element of G1), t2, t3 (elements of GT).

1. The device derives the compressed INTEL® EPID private key (gid, A.x, seed) from its fuses, where gid and A.x are read from the fuses and the seed value is derived from the fuse key FKey.
2. Let IVK be the Intel verification key embedded in the device. Given gid in the fuses, the device obtains the INTEL® EPID parameters certificate and the INTEL® EPID group certificate corresponding to gid. It verifies the correctness of the INTEL® EPID parameters and the INTEL® EPID public key using IVK.
3. Let the INTEL® EPID public key be (gid, h1, h2, w).
4. The device computes

   x = (Hash(seed || "(0x) 00 45 43 43 2d 53 61 66 65 49 44") ||
   Hash(seed || "(0x) 01 45 43 43 2d 53 61 66 65 49 44")) mod p,

   y = (Hash(seed || "(0x) 02 45 43 43 2d 53 61 66 65 49 44") ||
   Hash(seed || "(0x) 03 45 43 43 2d 53 61 66 65 49 44")) mod p,

   f = (Hash(seed || "(0x) 04 45 43 43 2d 53 61 66 65 49 44") ||
   Hash(seed || "(0x) 05 45 43 43 2d 53 61 66 65 49 44")) mod p.
5. The device computes A = G1.makePoint(A.x).

6. The device tests whether (A, x, y, f) is a valid INTEL® EPID private key as follows:
    a. It computes t1 = G1.exp(A, x).
    b. It computes t2 = pairing(t1, g2).
    c. It computes t3 = pairing(A, w).
    d. It computes t3 = GT.mul(t2, t3).
    e. It computes t1 = G1.multiexp(h1, f, h2, y).
    f. It computes t1 = G1.mul(t1, g1).
    g. It computes t2 = pairing(t1, g2).
    h. If GT.isEqual(t2, t3) = false
        i. It sets A = G1.inverse(A). *Note that A is modified here*.
        ii. It computes t3 = GT.exp(t3, p-1).
        iii. If GT.isEqual(t2, t3) = false again, it reports bad INTEL® EPID private key and exits.
7. The device's INTEL® EPID private key is (gid, A, x, y, f).
8. The device computes e12 = pairing(h1, g2).
9. The device computes e22 = pairing(h2, g2).
10. The device computes e2w = pairing(h2, w).
11. The device computes ea2 = pairing(A, g2).
12. The device's platform pre-computation blob is (e12, e22, e2w, ea2).
13. The device stores the platform pre-computation blob (e12, e22, e2w, ea2) along with its INTEL® EPID private key (gid, A, x, y, f) in its secure storage. The platform pre-computation blob will be used in the sign algorithm.

For math readers, the INTEL® EPID private key decompression steps are as follows:

1. The device derives the compressed INTEL® EPID private key (gid, A.x, seed) from the fuses of the device.
2. The device derives integers x, y, f from seed as in the step 4 above.
3. The device computes A = G1.makePoint(A.x). Note that either (A, x, y, f) or (A^{-1}, x, y, f) is the correct INTEL® EPID private key.
4. The device verifies that pairing(g1·$h1^{f}$·$h2^{y}$, g2) = pairing(A, $g2^{x}$·w). Note that pairing(A, $g2^{x}$·w) = pairing(A^{x}, g2) · pairing(A, w).
5. If the above equation not holds, the device sets A = G1.inverse(A).
    a. The device verifies pairing(g1·$h1^{f}$·$h2^{y}$, g2) = pairing(A, $g2^{x}$·w) again.
    b. If the above equation fails again, reports failure.
6. The value (gid, A, x, y, f) is the INTEL® EPID private key.
7. The device computes e12, e22, e2w, and ea2 as steps 8-11 above.

8. The device stores INTEL® EPID private key and platform pre-computation blob securely.

3.3 Generic Private Key Provisioning

This section describes INTEL® EPID private key provisioning for the generic setting. We assume the hardware device has enough storage to store the INTEL® EPID private key without the necessity to compress the private key.

3.3.1 Private Key Generation

In this process, the issuer generates a unique INTEL® EPID private key for each hardware device. In the following steps, we use temporary integers t1, t2, and t3.

1. The issuer chooses an INTEL® EPID group for the device. Let gid be the chosen group ID. Let (gid, h1, h2, w) be the group public key and (gid, u1, u2, gamma) be the group issuing private key.
2. The issuer chooses x, y, f randomly from [1, p-1].
3. The issuer computes integer t1 = (1 + u1 · f + u2 · y) mod p.
4. The issuer computes integer t2 = (gamma + x) mod p.
5. If t1 = 0 or t2 = 0, the issuer repeats from step 2.
6. The issuer computes integer t3 = inverse(t2) mod p, the inverse of t2 modulo p. Refer to Section 7.1 for the inverse operation over a prime field.
7. The issuer computes integer t1 = (t1 · t3) mod p.
8. The issuer computes A = G1.exp(g1, t1). A is an element of G1.
9. The value (gid, A, x, y, f) is the INTEL® EPID private key of the device.
10. The issuer erases the private key (gid, A, x, y, f) from its memory.

3.3.2 Private Key Verification and Platform Pre-Computation

A hardware device performs the following initial setup process after it receives an INTEL® EPID private key (gid, A, x, y, f) from the issuer. In the following process, we use temporary variables t1 (an element of G1), t2, t3 (elements of GT).

1. Let the INTEL® EPID public key be (gid, h1, h2, w) and the INTEL® EPID private key be (gid, A, x, y, f). The device tests whether (A, x, y, f) is a valid INTEL® EPID private key as follows:
    a. It computes t1 = G1.exp(A, x).
    b. It computes t2 = pairing(t1, g2).
    c. It computes t3 = pairing(A, w).
    d. It computes t3 = GT.mul(t2, t3).
    e. It computes t1 = G1.multiexp(h1, f, h2, y).
    f. It computes t1 = G1.mul(t1, g1).
    g. It computes t2 = pairing(t1, g2).

   h. If GT.isEqual(t2, t3) = false, it reports bad INTEL® EPID private key and exits.

2. The device computes e12 = pairing(h1, g2).
3. The device computes e22 = pairing(h2, g2).
4. The device computes e2w = pairing(h2, w).
5. The device computes ea2 = pairing(A, g2).
6. The device's platform pre-computation blob is (e12, e22, e2w, ea2).
7. The device stores the platform pre-computation blob (e12, e22, e2w, ea2) along with its INTEL® EPID private key (gid, A, x, y, f) in its secure storage. The platform pre-computation blob will be used in the sign algorithm.

3.4 Verifier Pre-Computation

A verifier can perform the following pre-computation after he obtains a group public key certificate from the issuer. In the end, the verifier produces a verifier pre-computation blob.

1. Let IVK be the Intel root verifier key known to the verifier.
2. The verifier obtains the INTEL® EPID parameters certificate and an INTEL® EPID group certificate from the issuer. The verifier verifies the correctness of the INTEL® EPID parameters and the INTEL® EPID public key using IVK.
3. Let the INTEL® EPID public key be (gid, h1, h2, w).
4. The verifier computes e12 = pairing(h1, g2).
5. The verifier computes e22 = pairing(h2, g2).
6. The verifier computes e2w = pairing(h2, w).
7. The verifier pre-computation blob for the INTEL® EPID group corresponding to gid is (gid, e12, e22, e2w).
8. The verifier stores the verifier pre-computation blob along with the INTEL® EPID public key.

4 Proof of Membership

In this section, we describe how to perform a proof of membership using the Intel® Enhanced Privacy ID (INTEL® EPID) scheme. We first present the sign and verify algorithms, then present a method such that the device can pre-compute most of an INTEL® EPID signature without knowing which message to sign.

In all cases, members and verifiers must confirm the signature of the following data items before their use

- Intel® EPID parameters certificate
- Group certificate

In all cases, verifiers must confirm the signature of the following data items before their use

- Private-key based revocation list

4.1 Basic Proof of Membership Algorithm

In this section, we present two INTEL® EPID algorithms: sign and verify. Before we present them, we sketch some ideas on how to use them in proof of membership. The proof of membership has the following steps.

1. The device sends gid to the verifier.
2. The verifier obtains the INTEL® EPID group public key and the revocation list PRIV-RL corresponding to gid.
3. The verifier chooses an arbitrary message m. The verifier sends m and possibly his basename to the device. The basename is a message with arbitrary length.
4. The device runs the sign algorithm based on the INTEL® EPID public key, its INTEL® EPID private key, the message m, and the verifier's basename (optional), and produces an INTEL® EPID signature σ. The device sends σ to the verifier.
5. Given the group public key, the message m, PRIV-RL, its basename, and the INTEL® EPID signature σ, the verifier runs the verify algorithm.

4.1.1 Sign algorithm

Input

(gid, h1, h2, w): INTEL® EPID public key

(gid, A, x, y, f): INTEL® EPID private key

m: the message from the verifier

mSize: size of m in bytes, 4-byte integer

bsn: verifier's basename (optional)

bsnSize: size of bsn in bytes, 4-byte integer

Output

σ = (B, K, T1, T2, c, nd, sx, sy, sf, sa, sb, salpha, sbeta): an INTEL® EPID signature

Steps

1. We use the following variables T1, T2, R1, R2 (elements of G1), R4 (elements of GT), B, K, R3 (elements of G3), a, b, alpha, beta, rx, ry, ra, rb, ralpha, rbeta, nrx, re22, t3, c, sx, sy, sa, sb, salpha, sbeta (256-bit big integers), nd (80-bit big integer), rf (592-bit big integer), and sf(600-bit big integer).
2. The device reads the platform pre-computation blob (e12, e22, e2w, ea2) from its secure storage. Refer to Section 3.3 for the computation of these values.
3. The device verifies gid in public key and private key all match.
4. If bsnSize = 0, the device computes B = G3.getRandom().
5. If bsnSize > 0, the device computes B = G3.hash(bsn).
6. The device computes K = G3.exp(B, f).

7. The device chooses two random integers a, b from [0, p-1].
8. The device computes T1 = G1.exp(h2, a).
9. The device computes T1 = G1.mul(T1, A).
10. The device computes T2 = G1. multiexp(h1, a, h2, b).
11. The device computes alpha = (a · x) mod p.
12. The device computes beta = (b · x) mod p.
13. The device chooses rx, ry, ra, rb, ralpha, rbeta randomly from [0, p-1].
14. The device chooses a random 592-bit integer rf. Here, the reason for choosing 592-bit integer is as follows: 592 = length(p) + length(hash) + slen = 256+256+80.
15. The device computes nrx = (- rx) mod p.
16. The device computes re22 = ((nrx · a) + ry + ralpha) mod p.
17. The device computes R1 = G1.multiexp(h1, ra, h2, rb).
18. The device computes R2 = G1.multiexp(h1, ralpha, h2, rbeta, T2, nrx).
19. The device computes R3 = G3.exp(B, rf).
20. The device computes R4 = GT.multiexp(ea2, nrx, e12, rf, e22, re22, e2w, ra).
21. The device computes t3 = Hash(p || g1 || g2 || g3 || h1 || h2 || w || B || K || T1 || T2 || R1 || R2 || R3 || R4).
22. The device chooses a random 80-bit unsigned integer nd.
23. The device computes c = Hash(t3 || nd || mSize || m).
24. The device computes sx = (rx + c · x) mod p.
25. The device computes sy = (ry + c · y) mod p.
26. The device computes sf = rf + c · f *over integer* (sf at most 593-bit*)*.
27. The device computes sa = (ra + c · a) mod p.
28. The device computes sb = (rb + c · b) mod p.
29. The device computes salpha = (ralpha + c · alpha) mod p.
30. The device computes sbeta = (rbeta + c · beta) mod p.
31. The device outputs σ = (B, K, T1, T2, c, nd, sx, sy, sf, sa, sb, salpha, sbeta).

For math readers, the steps are as follows:

1. The device follows steps 1-6 as above.
2. The device chooses two random integers a, b from [0, p-1].
3. The device computes T1 = $A \cdot h^a$ and T2 = $h1^a \cdot h2^b$.
4. The device computes alpha = (a · x) mod p and beta = (b · x) mod p.
5. The device chooses rx, ry, ra, rb, ralpha, rbeta randomly from [0, p-1].
6. The device chooses a random 592-bit integer rf.

7. The device computes R1 = $h1^{ra} \cdot h2^{rb}$.
8. The device computes R2 = $h1^{ralpha} \cdot h2^{rbeta} \cdot T2^{-rx}$.
9. The device computes R3 = B^{rf}.
10. The device computes R4 = $ea2^{-rx} \cdot e12^{rf} \cdot e22^{ry+ralpha-rx \cdot a} \cdot e2w^{ra}$.
11. The device chooses a random 80-bit unsigned integer nd.
12. The device computes c = Hash(Hash(p || g1 || g2 || g3 || h1 || h2 || w || B || K || T1 || T2 || R1 || R2 || R3 || R4) || nd || msize || m).
13. The device computes sx = (rx + c · x) mod p.
14. The device computes sy = (ry + c · y) mod p.
15. The device computes sf = rf + c · f *over integer* (sf at most 593-bit).
16. The device computes sa = (ra + c · a) mod p.
17. The device computes sb = (rb + c · b) mod p.
18. The device computes salpha = (ralpha + c · alpha) mod p.
19. The device computes sbeta = (rbeta + c · beta) mod p.
20. The device outputs σ = (B, K, T1, T2, c, nd, sx, sy, sf, sa, sb, salpha, sbeta).

Note on basename: as we described earlier, if a platform uses random base (bsnSize = 0), then the signature generated by the platform is anonymous. If the platform use name base (bsnSize > 1), then all the signatures created by the platform using the same bsn are linkable to the verifier. By default, the platform should use random base. However, in certain scenarios, the platform could use the name base option. To prevent loss of privacy, the platform may keep a list of authorized verifiers and corresponding base names in its local policy database. The platform could use a basename only if the verifier and the basename are in the platform's policy database.

4.1.2 Verify algorithm

Input

(gid, h1, h2, w): INTEL® EPID public key

(gid, RLver, n1, f[1], …, f[n1], sig): PRIV-RL

m: the message chosen by the verifier

mSize: size of m in bytes, 4-byte integer

bsn: verifier's basename (optional)

bsnSize: size of bsn in bytes, 4-byte integer

σ = (B, K, T1, T2, c, nd, sx, sy, sf, sa, sb, salpha, sbeta): an INTEL® EPID signature

Output

true (signature valid) or false (signature invalid)

Steps

1. We use the following variables T1, T2, R1, R2, t1, t2 (elements of G1), R4, t3 (elements of GT), B, K, R3, t5 (elements of G3), c, sx, sy, sa, sb, salpha, sbeta, nc, nc', nsx, syalpha, t4 (256-bit big integers), nd (80-bit big integer), and sf (600-bit big integer).
2. The verifier reads the verifier pre-computation blob (gid, e12, e22, e2w) from its storage. Refer to Section 3.4 for the computation of these values.
3. The verifier verifies gid in the public key, PRIV-RL, and the verifier pre-computation blob all match.
4. The verifier verifies the signature of PRIV-RL using IVK.
5. The verifier verifies that G3.isIdentity(B) is false.
6. If bsnSize = 0, the verifier verifies G3.inGroup(B) = true.
7. If bsnSize > 0, the verifier verifies B = G3.hash(bsn).
8. The verifier verifies G3.inGroup(K) = true.
9. The verifier verifies G1.inGroup(T1) = true.
10. The verifier verifies G1.inGroup(T2) = true.
11. The verifier verifies sx, sy, sa, sb, salpha, sbeta in [0, p-1].
12. The verifier verifies that sf is an (at-most) 593-bit unsigned integer, in other words, sf < 2^{593}.
13. The verifier computes nc = (-c) mod p.
14. The verifier computes nc' = (-c) mod p'.
15. The verifier computes nsx = (-sx) mod p.
16. The verifier computes syalpha = (sy + salpha) mod p.
17. The verifier computes R1 = G1.multiexp(h1, sa, h2, sb, T2, nc).
18. The verifier computes R2 = G1.multiexp(h1, salpha, h2, sbeta, T2, nsx).
19. The verifier computes R3 = G3.multiexp(B, sf, K, nc').
20. The verifier computes t1 = G1.multiexp(T1, nsx, g1, c).
21. The verifier computes t2 = G1.exp(T1, nc).
22. The verifier computes R4 = pairing(t1, g2).
23. The verifier computes t3 = pairing(t2, w).
24. The verifier computes R4 = GT.mul(R4, t3).
25. The verifier compute t3 = GT.multiexp(e12, sf, e22, syalpha, e2w, sa).
26. The verifier compute R4 = GT.mul(R4, t3).
27. The verifier compute t4 = Hash(p || g1 || g2 || g3 || h1 || h2 || w || B || K || T1 || T2 || R1 || R2 || R3 || R4).
28. The verifier verifies c = H(t4 || nd || mSize || m).
29. For i = 1, …, n1, the verifier computes t5 =G3.exp(B, f[i]) and verifies that G3.isEqual(t5, K) = false.

30. If all the above verifications succeed, the verifier outputs true. If any of the above verifications fails, the verifier immediately aborts and outputs false.

For math readers, the steps are following:

1. The verifier follows steps 1-4 as above.
2. If bsn is given, the verifier verifies B = G3.hash(bsn).
3. If bsn is not given, the verifier verifies that B $\in$ G3 and B is not the identity element of G3.
4. The verifier verifies K $\in$ G3, T1 $\in$ G1, and T2 $\in$ G1.
5. The verifier verifies sx, sy, sa, sb, salpha, sbeta in [0, p-1].
6. The verifier verifies that sf is an (at-most) 593-bit unsigned integer.
7. The verifier computes $R1 = h1^{sa} \cdot h2^{sb} \cdot T2^{-c}$.
8. The verifier computes $R2 = h1^{salpha} \cdot h2^{sbeta} \cdot T2^{-sx}$.
9. The verifier computes $R3 = B^{sf} \cdot K^{-c}$.
10. The verifier computes

    $R4 = \text{pairing}(T1^{-sx} \cdot g1^{c}, g2) \cdot \text{pairing}(T1^{-c}, w) \cdot e12^{sf} \cdot e22^{sy+salpha} \cdot e2w^{sa}$.
11. The verifier verifies c = Hash(Hash(p || g1 || g2 || g3 || h1 || h2 || w || B || K || T1 || T2 || R1 || R2 || R3 || R4) || nd || mSize || m).
12. For i = 1, …, n1, the verifier verifies that $K \neq B^{f[i]}$.
13. If all the above verifications succeed, the verifier outputs true. If any of the above verifications fails, the verifier immediately aborts and outputs false.

4.2 Pre-computation of Signatures

To speed up the performance of INTEL® EPID, we could pre-compute an INTEL® EPID signature without knowing the message to be signed. After the device knows what message to sign, it only needs to perform 2% or 5% of the total sign computation, depending on random base option or name base option. Members must not use the same pre-computed signature to generate more than one signature. If we use a pre-computed signature for two signatures, the attacker could learn the INTEL® EPID private key. We add two new algorithms: pre-sign, and sign-with-pre-computation.

4.2.1 Pre-Sign algorithm

Input

(gid, h1, h2, w): INTEL® EPID public key

(gid, A, x, y, f): INTEL® EPID private key

Output

pre-σ = (B, K, T1, T2, a, b, rx, ry, rf, ra, rb, ralpha, rbeta, R1, R2, R3, R4): basic pre-computed signature

Steps

1. We use the following variables T1, T2, R1, R2 (elements of G1), R4 (elements of GT), B, K, R3 (elements of G3), a, b, alpha, beta, rx, ry, ra, rb, ralpha, rbeta, nrx, re22 (256-bit big integers), and rf (592-bit big integer).
2. The device reads the platform pre-computation blob (e12, e22, e2w, ea2) from its secure storage. Refer to Section 3.3 for the computation of these values.
3. The device verifies gid in INTEL® EPID public key and private key all match.
4. The device computes B = G3.getRandom().
5. The device computes K = G3.exp(B, f).
6. The device chooses two random integers a, b from [0, p-1].
7. The device computes T1 = G1.exp(h2, a).
8. The device computes T1 = G1.mul(T1, A).
9. The device computes T2 = G1. multiexp(h1, a, h2, b).
10. The device chooses rx, ry, ra, rb, ralpha, rbeta randomly from [0, p-1].
11. The device chooses a random 592-bit integer rf.
12. The device computes nrx = (- rx) mod p.
13. The device computes re22 = ((nrx · a) + ry + ralpha) mod p.
14. The device computes R1 = G1.multiexp(h1, ra, h2, rb).
15. The device computes R2 = G1.multiexp(h1, ralpha, h2, rbeta, T2, nrx).
16. The device computes R3 = G3.exp(B, rf).
17. The device computes R4 = GT.multiexp(ea2, nrx, e12, rf, e22, re22, e2w, ra).
18. The device sets and outputs pre-σ = (B, K, T1, T2, a, b, rx, ry, rf, ra, rb, ralpha, rbeta, R1, R2, R3, R4).
19. The device stores pre-σ in the secure storage of the device.

4.2.2 Sign-with-Pre-Computation algorithm

Input

(gid, h1, h2, w): INTEL® EPID public key

(gid, A, x, y, f): INTEL® EPID private key

(B, K, T1, T2, a, b, rx, ry, rf, ra, rb, ralpha, rbeta, R1, R2, R3, R4): pre-computed signature

m: the message from the verifier

mSize: size of m in bytes, 4-byte integer

bsn: verifier's basename (optional)

bsnSize: size of bsn in bytes, 4-byte integer

Output

σ = (B, K, T1, T2, c, nd, sx, sy, sf, sa, sb, salpha, sbeta): an INTEL® EPID signature

Steps

1. We use the following variables T1, T2, R1, R2 (elements of G1), R4 (elements of GT), B, K, R3 (elements of G3), a, b, alpha, beta, rx, ry, ra, rb, ralpha, rbeta, t3, c, sx, sy, sa, sb, salpha, sbeta (256-bit big integers), nd (80-bit big integer), rf (592-bit big integer), and sf(600-bit big integer).
2. The device loads (B, K, T1, T2, a, b, rx, ry, rf, ra, rb, ralpha, rbeta, R1, R2, R3, R4) from the pre-computed signature.
3. If bsnSize > 0, the device does the following:
    a. The device computes B = G3.hash(bsn).
    b. The device computes K = G3.exp(B, f).
    c. The device computes R3 = G3.exp(B, rf).

    Note that the device over-writes the B, K, and R3 values.
4. If bsnSize = 0, the device uses B, K, R3 from the pre-computed signature.
5. The device computes alpha = (a · x) mod p.
6. The device computes beta = (b · x) mod p.
7. The device compute t3 = H(p || g1 || g2 || g3 || h1 || h2 || w || B || K || T1 || T2 || R1 || R2 || R3 || R4).
8. The device chooses a random 80-bit integer nd.
9. The device computes c = H(t3 || nd || mSize || m).
10. The device computes sx = (rx + c · x) mod p.
11. The device computes sy = (ry + c · y) mod p.
12. The device computes sf = rf + c · f *over integer* (sf at most 593-bit*)*.
13. The device computes sa = (ra + c · a) mod p.
14. The device computes sb = (rb + c · b) mod p.
15. The device computes salpha = (ralpha + c · alpha) mod p.
16. The device computes sbeta = (rbeta + c · beta) mod p.
17. The device deletes the pre-computed signature from its secure storage.
18. The device outputs σ = (B, K, T1, T2, c, nd, sx, sy, sf, sa, sb, salpha, sbeta).

5 Revocation Handling

We now describe how to revoke corrupted INTEL® EPID private keys. Each INTEL® EPID group has a private-key based revocation list. Initially, there is no private key that needs to be revoked. Therefore, PRIV-RL takes the following format at the beginning: PRIV-RL = (gid, RLver=0, n1=0, sig).

We now describe how to revoke a corrupted INTEL® EPID private key.

1. Someone reports a corrupted INTEL® EPID private key (gid, A, x, y, f) or a compressed private key (gid, A.x, seed) to the issuer.
2. Let PRIV-RL = (gid, RLver, n1, f[1], …, f[n1], sig) be the latest version of PRIV-RL for INTEL® EPID group corresponding to gid.
3. The issuer verifies the correctness of the private key as follows
    a. If the private key is compressed, decompress the private key as described in Section 3.2.2.
    b. Pick a random 256-bit message m.
    c. Compute an INTEL® EPID signature over m using the private key and using random base option.
    d. Verify the signature using the INTEL® EPID public key and PRIV-RL.
    e. If the signature verification fails, discard the private key. This means that the private key is not valid or has already been revoked.
4. The issuer updates the new PRIV-RL as follows
    a. Update the RL version number, i.e., compute RLver = RLver + 1.
    b. Update the total entry of the PRIV-RL, i.e., compute n1 = n1 + 1.
    c. Append f in the private key to PRIV-RL, i.e., set f[n1] = f.
    d. Update the signature on PRIV-RL using ISK.

In the above steps, the issuer revokes one private key at a time. Of course, the issuer can revoke multiple private keys before updating a new version of PRIV-RL.

The verifier can periodically download PRIV-RL from the issuer and use the latest PRIV-RL during the signature verification. Once the verifier obtains a new PRIV-RL, he always verifies the signature using IVK to make sure the PRIV-RL is indeed issued by Intel. He also needs to make sure that he gets a newer version of PRIV-RL by checking the RLver, revocation list version number.

6 Parameters of Intel® Enhanced Privacy ID Scheme

In this document, we provide a set of INTEL® EPID parameter we recommend. The parameters of G1, G2, and GT are generated by Pairing-Based Crypto (PBC) library. The parameters of G3 are the standard NIST P-256 elliptic curve.

We now present the parameters of each group. All the following numbers are in *hexadecimal* format.

```
p =
00008957 3F174730 8C43D5EE 41979619 72BB8688 ED4BEF04 ABAEC38E EC51C3D3
q =
09F924E5 D9BC677F 810DF025 58F75313 A98AA610 47655D73 9EF194EB 05B1A711
h =
00001297
a =
0553D7C8 81F778C2 2C37B6C0 163E6824 3A84781C 0ADF9BB3 ED21C446 E5A7A392
b =
```

003A2E39 0E10D8AC 47CB29C8 F12C7F11 992A18B7 EF7348A6 BE70A68B 97348AB1

coeff[0] =

02167A61 53DDF6E2 8915A094 F1B5DC65 211562E1 7DC54389 EEB4EFC8 A08E340F

coeff[1] =

048227E1 EB9864C2 8D8FDD0E 8240AED4 3163D646 3216857A B71868B8 170281A6

coeff[2] =

062076E8 545453B4 A9D8444B AAFB1CFD AE15CA29 79A624A4 0AF61EAC EDFB1041

qnr =

0866A767 366E6271 B7A65294 8FFB259E E64F25E5 269A2B6E 7EF8A639 AE46AA24

orderG2 =

0003DFFC BE2F5C2E 45497A2A 91BAD13E 01EC5FC2 151410B3 285E56CC 26512493

0E6C9996 38E07D68 8CB79723 F4AC4DBC 5E0115FF 45600813 CD59D773 B00C205E

ABAA2431 E22AA253 8AF786D5 1978C555 9C08B7E2 F4D03774 9356627B 95CC2CB0

p' =

FFFFFFFF 00000000 FFFFFFFF FFFFFFFF BCE6FAAD A7179E84 F3B9CAC2 FC632551

q' =

FFFFFFFF 00000001 00000000 00000000 00000000 FFFFFFFF FFFFFFFF FFFFFFFF

h′ =

00000001

a' =

FFFFFFFF 00000001 00000000 00000000 00000000 FFFFFFFF FFFFFFFF FFFFFFFC

b' =

5AC635D8 AA3A93E7 B3EBBD55 769886BC 651D06B0 CC53B0F6 3BCE3C3E 27D2604B

g1.x =

07783B0D FE4AA319 49B0CEAF 3F740F32 160C8B46 945BA5B0 E48ADAD8 88329053

g1.y =

08F7A2AA BA62B3FE 2980C95B 6353C824 3C7C1F4C DACDE55F A2369304 3C3ABC2E

g2.x[0] =

02109AF4 06323089 CB95E955 0E9DAF0E 98CDCADC B1FFFCD1 4566BB86 461E8C30

g2.x[1] =

047853E1 3F96C5E4 15237B1F 3F2CD395 40BC7A31 1F14389E 1AA5D663 1091E4D3

g2.x[2] =

00B402BC 47FAA629 820BB1D5 FFF2E6B0 C6AEE87B 91D9EE66 071FFDA2 E70266DD

g2.y[0] =

052EF8C6 C16AEF3C C195F626 CE5E55D1 641328B1 1857D81B 84FAEC7E 5D990649

g2.y[1] =

057335A9 A7F2A192 5F3E7CDF ACFE0FF5 08D03CAE CD58005F D0847EEA 6357FEC6

g2.y[2] =

0156DAF3 7261DAC6 93B0ACEF AAD4516D CA711E06 73EA83B2 B1994A4D 4A0D3507

```
g3.x =
6B17D1F2 E12C4247 F8BCE6E5 63A440F2 77037D81 2DEB33A0 F4A13945 D898C296
g3.y =
4FE342E2 FE1A7F9B 8EE7EB4A 7C0F9E16 2BCE3357 6B315ECE CBB64068 37BF51F5
```

7 Math Primitives and Group Operations

Note that: this section is very math intensive.

To implement the Intel® Enhanced Privacy ID (INTEL® EPID) scheme, we need to implement G1, G2, G3, GT. G1 is an elliptic curve group E(Fq) over a prime field Fq. G2 is an elliptic curve group E(Fq^3) over a finite field Fq^3. G3 is an elliptic curve group E(Fq') over a prime field Fq'. GT is an order-p subgroup of finite field Fq^6.

As for implementation, here is the order of what we need to build: a prime field, then a finite field, then finite field extension, then elliptic curve over prime field, then elliptic curve over finite field, and finally the Tate pairing operation. In our implementation, we use IPP cryptography library to construct INTEL® EPID. In the rest of this section, we sometimes use IPP functions; however, we try to be as independent to IPP as possible when we describe these math primitives.

7.1 Prime Field Fq

We first present the data format for each element of Fq and the parameter of Fq.

1. Each element in Fq is a positive big integer between [0,q-1]
2. param(Fq) = q, where q is a large prime.

We have the following functions associated with this prime field.

Fq.init(q)

Input: q

Steps:

1. Set q as the parameter of Fq.
2. If needed, we could do a prime testing on q to make sure q is indeed a prime number.

d = Fq.add(a, b)

Input: a, b (elements in Fq)

Output: d (an element in Fq) where d = a + b

Step: Compute d = (a + b) mod q.

d = Fq.subtract(a, b)

Input: a, b (elements in Fq)

Output: d (an element in Fq) where d = a - b

Step: Compute d = (a – b) mod q.

d = Fq.negate(a)

Input: a (an element in Fq)

Output: d (an element in Fq) where d = - a

Step: if a = 0, d =0, otherwise, compute d = q – a.

d = Fq.mul(a, b)

Input: a, b (elements in Fq)

Output: d (an element in Fq) where d = a · b

Step: Compute d = (a · b) mod q.

d = Fq.exp(a, b)

Input: a (an element in Fq), b (a non-negative integer)

Output: d (an element in Fq) where d = a^b

Step: Compute d = (a^b) mod q.

d = Fq.inverse(a)

Input: a (an element in Fq)

Output: d (an element in Fq) where a·d = 1

Step: Compute ippsModInv_BN(a, q, d).

d = Fq.isEqual(a, b)

Input: a, b (elements in Fq)

Output: d (Boolean). If a = b, then d = true, otherwise d = false

Step: If a = b, set d = true, else set d = false.

d = Fq.getRandom()

Output: d (an element in Fq). d is chosen randomly from Fq

Steps:

1. Choose a random integer t of length(q) + slen bits. In out setting, length(q) = 256, thus we choose a 336-bit integer t.
2. Compute d = t mod q.

d = Fq.squareRoot(a)

Input: a (an element in Fq)

Output: d (an element in Fq) where d is a square root of a, i.e., a = d·d mod q

Steps:

1. Choose an element g in Fq.
2. Check whether $g^{(q-1)/2}$ mod q = q-1. If not, go to step 1.
3. Set t = q-1, s = 0.
4. While (t is even number)

   t = t/2, s = s+1.
5. Note that g, s, t can be pre-computed and used for all future computations. Also note that q-1 = 2^s·t where t is an odd integer.
6. e = 0.
7. For i = 2, ..., s

   j = 2^i,

   if $(a \cdot g^{-e})^{(q-1)/j}$ mod q ≠ 1, then set e = e + j/2.
8. Compute h = $(a \cdot g^{-e})$ mod q.
9. Compute d = $(g^{e/2} \cdot h^{(t+1)/2})$ mod q.
10. Verify whether a = d^2 mod q. If so, return d, otherwise, return fail.

Note that the above algorithms for prime field Fq is mathematically correct, but may not be the most efficient way to implement. For performance enhancement, we could use Montgomery multiplication and exponentiation. Given an element in Fq, we could always keep the element in the Montgomery representation until we need to output this element.

7.2 Finite Field Fq^d

We first present the data format for each element of Fq^d and the parameters of Fq^d. In our INTEL® EPID setting, d is equal to 3.

1. Each element a in Fq^d takes the format of (a[0], …, a[d-1]) where a[0], …, a[d-1] are elements of prime field Fq, or integers between [0, q-1].
2. param(Fq^d) = (q, d, coeff[0], …, coeff[d-1])
   - q is a large prime and is the characteristic of the finite field.
   - d = 3.
   - coeff[0], …, coeff[d-1] are elements in Fq and represent an irreducible polynomial.

We have the following operations for this finite field Fq^d. In the following functions, we often treat each element in Fq as a big integer.

For math readers, we have the following notations:

1. For each element a in Fq^d with format of (a[0], …, a[d-1]), element a can be seemed as a polynomial $a[d-1] \cdot t^{d-1} + \ldots + a[1] \cdot t + a[0]$.
2. We have an irreducible polynomial p(t) defined as: $p(t) = t^d + coeff[d-1] \cdot t^{d-1} + \ldots + coeff[1] \cdot t + coeff[0]$.

Fq^d.init(q, d, coeff[d-1], …, coeff[0])

Input: q (a prime), d (16-bit unsigned integer), coeff[0], …, coeff[d-1] (big integers),

Steps:

1. Set (q, d, coeff[0], …, coeff[d-1]) as parameters of Fq^d.
2. If needed, we could do a prime testing on q to make sure q is indeed a prime number, and test p(t) is indeed an irreducible polynomial.

e = Fq^d.add(a, b)

Input: a, b (elements in Fq^d)

Output: e (an element in Fq^d) where e = a + b

Steps:

1. Let a = (a[0], …, a[d-1]), b = (b[0], …, b[d-1]), and e = (e[0], …, e[d-1]).
2. For i = 0, …, d-1, set e[i] = (a[i] + b[i]) mod q.

e = Fq^d.add(a, b)

Input: a (an element in Fq^d), b (big integer)

Output: e (an element in Fq^d) where e = a + b

Steps:

1. Let a = (a[0], …, a[d-1]), and e = (e[0], …, e[d-1]).
2. For i = 1, …, d-1, set e[i] = a[i].
3. e[0] = (a[0] + b) mod q.

e = Fq^d.subtract(a, b)

Input: a, b (elements in Fq^d)

Output: e (an element in Fq^d) where e = a - b

Steps:

1. Let a = (a[0], …, a[d-1]), b = (b[0], …, b[d-1]), and e = (e[0], …, e[d-1]).
2. For i = 0, …, d-1, set e[i] = (a[i] - b[i]) mod q.

e = Fq^d.negate(a)

Input: a (an element in Fq^d)

Output: e (an element in Fq^d) where e = -a

Steps:

1. Let a = (a[0], …, a[d-1]) and e = (e[0], …, e[d-1]).
2. For i = 0, …, d-1, set e[i] = Fq.negate(a[i]).

e = Fq^d.mul(a, b)

Input: a, b (elements in Fq^d)

Output: e (an element in Fq^d) where e = a · b

Steps:

1. Let a = (a[0], …, a[d-1]), b = (b[0], …, b[d-1]), and e = (e[0], …, e[d-1]).
2. Let t[0], …, t[2d-2] be temporary big integers
3. For generic algorithm, we have
    a. For i = 0, …, 2d-2, set t[i] = 0.
    b. For i = 0, …, d-1, and for j = 0, …, d -1,
        Compute t[i+j] = (t[i+j] + a[i] × b[j]) mod q.
4. Alternatively, for d = 3, we can do the following for performance improvement (in this case, skip step 3):
    a. Compute p[0] = (a[0] × b[0]) mod q.
    b. Compute p[1] = (a[1] × b[1]) mod q.
    c. Compute p[2] = (a[2] × b[2]) mod q.
    d. Compute p[01] = ((a[0] + a[1]) × (b[0] + b[1])) mod q.
    e. Compute p[02] = ((a[0] + a[2]) × (b[0] + b[2])) mod q.
    f. Compute p[12] = ((a[1] + a[2]) × (b[1] + b[2])) mod q.
    g. Compute t[0] = p[0].
    h. Compute t[1] = (p[01] – p[0] – p[1]) mod q.
    i. Compute t[2] = (p[02] – p[0] – p[2] + p[1]) mod q.
    j. Compute t[3] = (p[12] – p[1] – p[2]) mod q.
    k. Compute t[4] = p[2].
5. For i = 2d-2, …, d
    For j = 0, …, d-1,
        Compute t[i+j−d] = (t[i+j−d] − t[i] × coeff[j]) mod q.
6. For i = 0, …, d-1, set e[i] = t[i].
7. Return e.

For math readers, we have the following steps:

1. Let a = (a[0], …, a[d-1]), b = (b[0], …, b[d-1]).
2. e = ((a[d-1] · t^{d-1} + … + a[0]) · (b[d-1] · t^{d-1} + … + b[0])) mod p(t)

e = Fq^d.mul(a, b)

Input: a (an element in Fq^d), b (a big integer)

Output: e (an element in Fq^d) where e = a · b

Steps:

1. Let a = (a[0], …, a[d-1]) and e = (e[0], …, e[d-1]).
2. For i = 0, …, d-1, set e[i] = (a[i] · b) mod q.

e = Fq^d.exp(a, b)

Input: a (an element in Fq^d), b (a non-negative integer)

Output: e (an element in Fq^d) where e = a^b

Steps:

1. Let $b_n…b_1b_0$ be the binary representation of b.
2. Set e = a.
3. For i = n-1, …, 0, do the following:

   e = Fq^d.mul(e, e),

   If b_i = 1, compute e = Fq^d.mul(e, a).
4. Return e.

e = Fq^d.inverse(a)

Input: a (an element in Fq^d)

Output: e (an element in Fq^d) where a · e = 1

Steps:

1. This algorithm is described for math readers only.
2. Let a = (a[0], …, a[d-1]).
3. Let lastrem, rem, quo, lastaux, aux, temp be polynomials with coefficients in Fq. The degrees of these polynomials are at most d.
4. Set lastaux = 0, aux = 1.
5. Set lastrem = p(t), the irreducible polynomial.
6. Set rem = a[d-1] · t^{d-1} + … + a[1] ·t + a[0].
7. while (the degree of rem > 0)

   a. temp = rem, // save rem to temp

   b. (quo, rem) = Fq^d.div(lastrem, rem),

c. lastrem = temp, // lastrem is the previous rem value
d. temp = aux, // save aux to temp
e. aux = lastaux – quo · aux,
f. lastaux = temp, // lastaux is the previous aux value

8. t = Fq.inverse(rem). // rem has degree 0, i.e., rem is in Fq
9. e = Fq^d.mul(aux, t).
10. Return e.

(q, r) = Fq^d.div(a, b)

Input: a, b (polynomials with coefficients in Fq)

Output: q, r (polynomials with coefficients in Fq) such that a = b · q + r

Steps:

1. Let a = a[m] · t^m + … + a[1] ·t + a[0].
2. Let b = b[n] · t^n + … + a[1] ·t + a[0].
3. Set r = a, q = 0.
4. If m < n, and return.
5. For i = m–n, …, 0, do the following
   a. Compute temp = (r[m+i] / b[n]) mod q,
   b. Set q = q + temp · t^i,
   c. For j = 0, ..., n, Set r[i+j] = r[i+j] – temp · b[j].
6. Return (q, r).

For math readers:

1. r = remainder(a / b).
2. q = quotient(a / b).

e = Fq^d.isEqual(a, b)

Input: a, b (elements in Fq^d)

Output: e (Boolean). If a = b, then e = true, otherwise e = false

Steps:

1. Let a = (a[0], …, a[d-1]) and b = (b[0], …, b[d-1]).
2. If a[i] = b[i] for all i = 0, …, d-1, return e = true.
3. Otherwise, return e = false.

e = Fq^d.getRandom()

Output: e (an element in Fq^d). e is chosen randomly from Fq^d

Steps:

1. Let e = (e[0], ..., e[d-1]).
2. For i = 0, …, d-1, compute e[i] = Fq.getRandom().

7.3 Quadratic Field Extension Fq^k

We first present the data format for each element of Fq^k and the parameters of Fq^k. In our INTEL® EPID scheme, $Fq^k = Fq^d[\sqrt{qnr}]$, a quadratic extension of field Fq^d, where d=3, k=6, and qnr is a positive integer and a quadratic non-residue in Fq^d. In general case, qnr could be any quadratic non-residue in Fq^d, but not limited to elements in Fq.

1. Each element a in Fq^k takes the format of (a[0], a[1]) where a[0], a[1] are elements of Fq^d.
2. param(Fq^k) = (param(Fq^d), qnr)
   - qnr is a positive integer and is a quadratic non-residue in Fq^d.
   - param(Fq^d) is the parameters of Fq^d.

We have the following operations for Fq^k.

For math readers, given a finite field Fq^d that has already been defined, the quadratic field extension Fq^k can be seemed as a field extension to Fq^d, where

1. For each element a = (a[0], a[1]), it can be seemed as a polynomial $a[1] \cdot t + a[0]$.
2. The irreducible polynomial $p(t) = t^2 - qnr$.

Fq^k.init(param(Fq^d), qnr)

Input: param(Fq^d), qnr (a positive integer)

Steps:

1. Set param(Fq^k) = (param(Fq^d), qnr).
2. If needed, we could test that qnr is indeed a quadratic non-residue.

e = Fq^k.add(a, b)

Input: a, b (elements in Fq^k)

Output: e (an element in Fq^k) where d = a + b

Steps:

1. Let a = (a[0], a[1]), b = (b[0], b[1]), and e = (e[0], e[1]).
2. For i = 0 and 1, set e[i] = Fq^d.add(a[i], b[i]).

e = Fq^k.subtract(a, b)

Input: a, b (elements in Fq^k)

Output: e (an element in Fq^k) where e = a - b

Steps:

1. Let a = (a[0], a[1]), b = (b[0], b[1]), and e = (e[0], e[1]).
2. For i = 0 and 1, set e[i] = Fq^d.subtract(a[i], b[i]).

e = Fq^k.negate(a)

Input: a (an element in Fq^k)

Output: e (an element in Fq^k) where e = -a

Steps:

1. Let a = (a[0], a[1]) and e = (e[0], e[1]).
2. For i = 0 and 1, set e[i] = Fq^d.negate(a[i]).

e = Fq^k.mul(a, b)

Input: a (an element in Fq^k), b (a big integer)

Output: e (an element in Fq^k) where e = a · b

Steps:

1. Let a = (a[0], a[1]) and e = (e[0], e[1]).
2. For i = 0 and 1, set e[i] = Fq^d.mul(a[i], b).

e = Fq^k.mul(a, b)

Input: a, b (elements in Fq^k)

Output: e (an element in Fq^k) where e = a · b

Steps:

1. Let a = (a[0], a[1]), b = (b[0], b[1]), and e = (e[0], e[1]).
2. All the following arithmetic operations are in Fq^d.
3. e[0] = a[0] · b[0] + a[1] · b[1] · qnr.
4. e[1] = a[0] · b[1] + a[1] · b[0].
5. Note that for performance reason, we can compute e[1] as follows

   e[1] = (a[0] + a[1]) · (b[0] + b[1]) – a[0] · b[0] – a[1] · b[1].
6. Return e = (e[0], e[1]).

For math readers, we have the following steps:

1. Let a = (a[0], a[1]), b = (b[0], b[1]).
2. e = ((a[1] · t + a[0]) · (b[1] · t + b[0])) mod p(t)

e = Fq^k.exp(a, b)

Input: a (an element in Fq^k), b (a non-negative integer)

Output: e (an element in Fq^k) where $e = a^b$

Steps:

1. Let $b_n \ldots b_1 b_0$ be the binary representation of b.
2. Set e = a.
3. For i = n-1, …, 0, do the following:

   e = Fq^k.mul(e, e),

   If $b_i = 1$, compute e = Fq^k.mul(e, a).
4. Return e.

e = Fq^k.inverse(a)

Input: a (an element in Fq^k)

Output: e (an element in Fq^k) where $a \cdot e = 1$

Steps:

1. Let a = (a[0], a[1]) and e= (e[0], e[1]).
2. temp = a[1] · a[1] · qnr – a[0] · a[0].
3. temp = Fq^d.inverse(temp).
4. e[0] = Fq^d.mul(a[0], temp).
5. e[0] = Fq^d.negate(e[0]).
6. e[1] = Fq^d.mul(a[1], temp).
7. Return e.

For math readers, we have the following steps:

1. Let a = (a[0], a[1]) and e= (e[0], e[1]).
2. e[0] = – a[0] / (a[1] · a[1] · qnr – a[0] · a[0]).
3. e[1] = a[1] / (a[1] · a[1] · qnr – a[0] · a[0]).

e = Fq^k.isEqual(a, b)

Input: a, b (elements in Fq^k)

Output: e (Boolean). If a = b, then e = true, otherwise e = false

Steps:

1. Let a = (a[0], a[1]) and b = (b[0], b[1]).
2. If a[0] = b[0] and a[1] = b[1], then return e = true.

3. Otherwise, return e = false.

7.4 Elliptic Curve Group E(Fq) over a Prime Field

We now describe the elliptic curve arithmetic for E(Fq) over a prime field Fq. For efficiency reason, we describe all the functions in this subsection using IPP API. We first present the data format for each element of E(Fq) and the parameters of E(Fq).

1. Each element P in E(Fq) takes the format of (P.x, P.y) where P.x, P.y are elements of Fq. There is a special element in E(Fq), denoted as O, the point of infinity. We treat this element separately.
2. param(E(Fq)) = (p, q, h, a, b, x, y)
    - p is a large prime. p is the total number of elements in the group E(Fq).
    - q is a large prime. q is the parameter of the prime field Fq.
    - h is an integer such that p·h = #E(Fq), total number of points.
    - a and b are coefficients of the curve E: $y^2 = x^3 + a \cdot x + b$, where a and b are integers between [0, q-1].
    - x and y are big integers in Fq such that (x, y) forms a generator of this ECC group.

We have the following operations for E(Fq).

E(Fq).init(p, q, h, a, b, x, y)
Input: p, q, a, b, x, y (big integers), h (32-bit integer)
Step: Call ippsECCPSet(q, a, b, x, y, p, h, ecc).

R = E(Fq).mul(P, Q)
Input: P, Q (elements of E(Fq))
Output: R (an element of E(Fq)) where R = P·Q
Step: Call ippsECCPAddPoint(P, Q, R, ecc).

For math readers: E(Fq).mul() is point addition.

R = E(Fq).exp(P, b)
Input: P (an element of E(Fq)), b (a non-negative integer)
Output: R (an element of E(Fq)) where R = P^b
Step: Call ippsECCPMulPointScalar(P, b, R, ecc).

For math readers: E(Fq).exp() is point multiplication.

R = E(Fq).inverse(P)
Input: P (an element of E(Fq))
Output: R (an element of E(Fq)) where R = -P
Step: Call ippsECCPNegativePoint(P, R, ecc).

R = E(Fq).makePoint(P.x)

Input: P.x (an element of Fq)

Output: R (an element of E(Fq))

Steps:

1. Compute temp = ($P.x^3$ + a · P.x + b) mod q.
2. Compute P.y = Fq.squareRoot(temp).
3. Set R = (P.x, P.y) by calling ippsECCPSetPoint(P.x, P.y, R, ecc).

d = E(Fq).inGroup(P)

Input: P (an element of E(Fq))

Output: d (Boolean). If P is an element in E(Fq), d = true, otherwise, d = false

Steps:

1. Call ippsECCPCheckPoint(P, result, ecc).
2. If result = ippECPointIsNotValid, return d = false.
3. If h = 1,
    a. Return d = true.
4. else
    a. Compute Q = E(Fq).exp(P, p), where p is the order the group
    b. If E(Fq).isIdentity(Q) = true, return d = true,
    c. Otherwise, return d = false.

Note that steps 3 and 4 may be skipped if ippsECCPCheckPoint already performs them.

d = E(Fq).isIdentity(P)

Input: P (an element of E(Fq))

Output: d (Boolean). If P is point at infinity, d = true, otherwise, d = false

Steps:

1. Call ippsECCPCheckPoint(P, result, ecc).
2. If result = ippECPointIsAtInfinite, set d = true, else set d = false.

R = E(Fq).getRandom()

Output: R (an element of E(Fq)), a random element in E(Fq)

Steps:

1. Compute R.x = Fq.getRandom().
2. Compute t1 = ($R.x^3$ + a · R.x + b) mod q.
3. Compute t2 = Fq.squareRoot(t1).
4. If computing square root fails, go to step 1.
5. Let y[0] = min(t2, q-t2), y[1] = max(t2, q-t2).
6. Choose a random bit b and set R.y = y[b].

7. Set R = E(Fq).exp(R, h).

Alternative method (maybe slower):

1. Choose a random integer a between [0, p-1].
2. Compute R = E(Fq).exp(g, a).

R = E(Fq).hash(m)

Input: m (arbitrary length message)

Output: R (an element of E(Fq)) where R = Hash(m)

Steps:

1. Set i = 0.
2. Compute Hash(i || m) || Hash(i+1 || m) where i is 32-bit unsigned integer.
3. Let b be the first bit of the above hash result.
4. Let t be the next 336-bit of the above hash result. Here the reason for choosing 336-bit is as follows: 336 = length(q) + slen = 256 + 80.
5. Compute R.x = t mod q.
6. Compute t1 = $(R.x^3 + a \cdot R.x + b) \bmod q$.
7. Compute t2 = Fq.squareRoot(t1).
8. If computing square root fails, set i = i+2, go to step 2.
9. Let y[0] = min(t2, q-t2), y[1] = max(t2, q-t2).
10. Set R.y = y[b].
11. Set R = E(Fq).exp(R, h).

d = E(Fq).isEqual(P, Q)

Input: P, Q (elements of E(Fq))

Output: d (Boolean). If P = Q, d = true, otherwise, d = false

Steps:

1. Call ippsECCPComparePoint(P, Q, result, ecc).
2. If result = ippECPointIsEqual, set d = true, else set d = false.

7.5 Elliptic Curve Group E(Fq^d) over a Finite Field

We now describe the elliptic curve arithmetic for E(Fq^d) over a finite field Fq^d. We already described the arithmetic of Fq^d in Section 7.2. Again, in our setting d = 3. We first present the data format for each element of E(Fq^d) and the parameters of E(Fq^d).

1. Each element P in E(Fq^d) takes the format of (P.x, P.y) where P.x and P.y are elements in Fq^d. Again, there is a special element O in E(Fq^d), the point of infinity.
2. param(E(Fq^d)) = (orderG2, param(Fq^d), a, b, x, y)

- orderG2 is the total number of points in $E(Fq^d)$.
- param(Fq^d) is the parameters of the finite field Fq^d.
- a and b are coefficients of the curve E: $y^2 = x^3 + a \cdot x + b$. Note that, in general, a and b could be arbitrary elements in Fq^d. In our setting, a and b are elements in Fq.
- x and y are elements of Fq^d such that (x, y) forms a generator of this elliptic curve group.

We have the following operations for $E(Fq^d)$.

$E(Fq^d)$.init(orderG2, param(Fq^d), a, b, x, y)

Input: orderG2, a, b (big integers), x, y (elements of Fq^d)

Steps:

1. Set param($E(Fq^d)$) = (orderG2, param(Fq^d), a, b, x, y)
2. Set g.x = x, g.y = y. Now g = (g.x, g.y) is a generator of $E(Fq^d)$.

R = $E(Fq^d)$.mul(P, Q)

Input: P, Q (elements of $E(Fq^d)$)

Output: R (an element of $E(Fq^d)$) where $R = P \cdot Q$

Steps:

1. If P = O, set R = Q and return.
2. If Q = O, set R = P and return.
3. Now let P = (P.x, P.y), Q = (Q.x, Q.y), and R = (R.x, R.y)
4. If P.x = Q.x and $P.y \neq Q.y$

   Set R = O and return.
5. lambda = $E(Fq^d)$.computeLambda(P, Q).
6. R.x = lambda · lambda – P.x – Q.x.
7. R.y = (P.x – R.x) · lambda – P.y.
8. Return R.

lambda = $E(Fq^d)$.computeLamda(P, Q)

Input: P, Q (elements of $E(Fq^d)$)

Output: lambda (an element of Fq^d)

Steps:

1. Let P = (P.x, P.y) and Q = (Q.x, Q.y).
2. If P = Q,

   lambda = (3 · P.x · P.x + a) / (2 · P.y), where a is the parameter of E(Fq).
3. If P != Q,

   lambda = (Q.y – P.y) / (Q.x – P.x).

4. Return lambda.

Note that for the following two algorithms, only one of them needs to be implemented. The method using Jacobian projective coordinates is more efficient.

R = E(Fq^d).exp(P, b) with Affine space

Input: P (an element of E(Fq^d)), b (a non-negative integer)

Output: R (an element of E(Fq^d)) where R = P^b

Steps:

1. Let $b_n \ldots b_1 b_0$ be the binary representation of b.
2. Set R = P.
3. For i = n-1, …, 0, do the following:

   R = E(Fq^d).mul(R, R),

   If b_i = 1, compute R = E(Fq^d).mul(R, P).
4. Return R.

R = E(Fq^d).exp(P, b) using Jacobian Projective Coordinates

Input: P (an element of E(Fq^d)), b (a non-negative integer)

Output: R (an element of E(Fq^d)) where R = P^b

Steps:

1. Let $b_n \ldots b_1 b_0$ be the binary representation of b.
2. Let X, Y, Z, X′, Y′, Z′, w, v be elements in Fq^d.
3. Let P = (P.x, P.y).
4. Set X = P.x, Y = P.y, and Z = 1.
5. For i = n-1, …, 0, do the following with arithmetic in Fq^d:
   a. Compute $w = 3 \cdot X^2 + a \cdot Z^4$,
   b. Compute $v = 4 \cdot X \cdot Y^2$,
   c. Compute $X' = w^2 - 2 \cdot v$,
   d. Compute $Y' = w \cdot (v - X') - 8 \cdot Y^4$,
   e. Compute $Z' = 2 \cdot Y \cdot Z$,
   f. Set X = X′, Y = Y′, Z = Z′,
   g. If b_i = 1, compute the following
      i. Compute $w = P.x \cdot Z^2 - X$,
      ii. Compute $v = P.y \cdot Z^3 - Y$,
      iii. Compute $X' = v^2 - w^3 - 2 \cdot X \cdot w^2$,
      iv. Compute $Y' = v \cdot (X \cdot w^2 - X') - Y \cdot w^3$,
      v. Compute $Z' = w \cdot Z$,

vi. Set X = X', Y = Y', Z = Z',

6. If Z = 0, return R = O, point at infinity.
7. Let R = (R.x, R.y).
8. Set R.x = X · $(Z^{-1})^2$,
9. Set R.y = Y · $(Z^{-1})^3$,
10. Return R.

Note: the exponentiations described in this document are all based on binary method; we can always enhance the performance using sliding windows techniques.

R = E(Fq^d).inverse(P)

Input: P (an element of E(Fq^d))

Output: R (an element of E(Fq^d)) where R = -P

Steps:

1. If P = O, set R = O and return.
2. Set R.x = P.x.
3. Set R.y = Fq^d.negate(P.y).

e = E(Fq^d).isIdentity(P)

Input: P (an element of E(Fq^d))

Output: e (Boolean). If P is point at infinity, e = true, otherwise, e = false

Step: If P = O, set e = true, else, set e = false.

e = E(Fq^d).isEqual(P, Q)

Input: P, Q (elements of E(Fq^d))

Output: e (Boolean). If P = Q, e = true, otherwise, e = false

Steps:

1. If P = Q = O, set e = true and return.
2. If one of P, Q is equal O, the other is not, set e = false and return.
3. If Fq^d.isEqual(P.x, Q.x) = true and Fq^d.isEqual(P.y, Q.y) = true, set e = true.
4. Otherwise set e = false.

7.6 Operations for G1, G2, G3, GT

The overall parameters of G1, G2, G3, and GT are: (p, q, h, a, b, coeff[0], coeff[1], ceoff[2], qnr, orderG2, p', q', h', a', b', g1.x, g1.y, g2.x[0], g2.x[1], g2.x[2], g2.y[0], g2.y[1], g2.y[2], g3.x, g3.y), where h and h' are 32-bit integers, orderG2 is a 768-bit integer, the rest of variables are 256-bit integers.

G1 is an elliptic curve group E(Fq). It can be initialized as follows:

1. Set G1 = Fq.init(p, q, h, a, b, g1.x, g1.y).
2. Set g1 = (g1.x, g1.y), an element of G1.

G3 is an elliptic curve group E(Fq) as well (but with different parameters). It can be initialized as follows:

1. Set G3 = Fq.init(p', q', h', a', b', g3.x, g3.y).
2. Set g3 = (g3.x, g3.y), an element of G3.

GT is a quadratic field extension Fq^k. It can be initialized as follows:

1. Set Fq^d = Fq^d.init(q, 3, coeff[0], coeff[1], coeff[2]).
2. Set GT = Fq^k.init(param(Fq^d), qnr).

G2 is an elliptic curve group E(Fq^d). It can be initialized as follows:

1. Set Fq^d = Fq^d.init(q, 3, coeff[0], coeff[1], coeff[2]).
2. Set g2.x = (g2.x[0], g2.x[1], g2.x[2]) an element of Fq^d.
3. Set g2.y = (g2.y[0], g2.y[1], g2.y[2]) an element of Fq^d.
4. twista = (a · qnr · qnr) mod q.
5. twistb = (b · qnr · qnr · qnr) mod q.
6. Set G2 = E(Fq^d).init(orderG2, param(Fq^d), twista, twistb, g2.x, g2.y).
7. Set g2 = (g2.x, g2.y), an element of G2.

Operations of G1 and G3 can be referred to Section 7.4. Operations of G2 can be referred to Section 7.5. Operations of GT can be referred to Section 7.3.

7.7 Operations for Multi-Exponentiation

In INTEL® EPID algorithm, we use multi-exponentiation to speed up the performance. We now describe the generic multi-exponentiation algorithms. In what follows, we assume we have a group G with basic operations such as multiplication.

R = G.multiexp(P[0], b[0], …, P[m-1], b[m-1])

Input: P[0], …, P[m-1] (elements of G), b[0], …, b[m-1] (positive integers), m>1

Output: R (an element of G) where $R = P[0]^{b[0]} \cdot \ldots \cdot P[m-1]^{b[m-1]}$

Steps:

1. Assume b[0], …, b[m-1] are equal length. If not, pad the short ones with 0's in the most significant bits to make b[0], …, b[m-1] equal length.
2. Assume m is small. If m is large, e.g., m > 5, then the following algorithm may not be efficient.
3. Let t[1], …, $t[2^m-1]$ be 2^m-1 elements in G need to be computed as follows.

4. For j = 1, …, 2^m-1,
    a. Let j[m-1] … j[1] j[0] be the binary representation of j.
    b. Set $t[i] = P[0]^{j[0]} \cdot P[1]^{j[1]} \cdot \ldots \cdot P[m-1]^{j[m-1]}$.
5. Note that we need at most 2^m multiply operations for the above step.
6. For i = 1, …, m,
    a. Assume that b[i] is (n+1) bits in length.
    b. Let $b[i]_n \ldots b[i]_1\ b[i]_0$ be the binary representation of b[i].
7. Let $j = b[0]_n + 2 \cdot b[1]_n + \ldots + 2^{m-1} \cdot b[m-1]_n$.
8. Set R = t[j].
9. For i = n-1, …, 0, do the following:
    R = G.mul(R, R),
    Compute $j = b[0]_i + 2 \cdot b[1]_i + \ldots + 2^{m-1} \cdot b[m-1]_i$,
    If $j \neq 0$, compute R = G.mul(R, t[j]).
10. Return R.

7.8 Operations for Pairing

A Tate pairing function takes an element of G1 and an element of G2 and outputs an element in GT. Before we present the algorithm pairing function, we describe some building blocks. In what follows, we assume that G1, G2, and GT have already been initialized. *The following algorithm is for math readers only*.

d = pairing(P, Q)

Input: P (an elements in G1), Q (an element in G2)

Output: d (an element in GT), result of Tate pairing

Steps:

1. Let Q = (Q.x, Q.y), where Q.x and Q.y are elements in Fq^d.
2. Compute inv = Fq.inverse(qnr).
3. Now we compute Qx, Qy, two elements in GT, as follows.
4. Compute Qx = (inv · Q.x, 0).
5. Compute Qy = (0, inv^2 · Q.y).
6. Call d = pairing(P, Qx, Qy).
7. Return d.

d = pairing(P, Qx, Qy)

Input: P (an elements in G1), Qx, Qy (elements in GT)

Output: d (an element in GT), result of Tate pairing

Steps:

1. Let $p_n \ldots p_1 p_0$ be the binary representation of p.
2. If P = O, point at infinity, then return r = 1.
3. Let P = (px, py), where px, py are big integers.
4. Let X, Y, Z, X', Y', Z', w, v, ty, ry be elements in Fq.
5. Let tx, rx be elements in GT.
6. Set X = px, Y = py, Z = 1, ry = 1.
7. Set rx = 1, identity element of GT.
8. For i = n-1, …, 0 do
    a. Compute $w = (3 \cdot X^2 + a \cdot Z^4) \bmod q$, where a is parameter of G1,
    b. Compute $v = (4 \cdot X \cdot Y^2) \bmod q$,
    c. Compute $X' = (w^2 - 2 \cdot v) \bmod q$,
    d. Compute $Y' = (w \cdot (v - X') - 8 \cdot Y^4) \bmod q$,
    e. Compute $Z' = (2 \cdot Y \cdot Z) \bmod q$,
    f. Compute $ty = (Z' \cdot Z^2) \bmod q$,
    g. Compute $tx = ty \cdot Qy - 2 \cdot Y^2 - w \cdot Z^2 \cdot Qx + w \cdot X$, (arithmetic in GT)
    h. Set X = X', Y = Y', Z = Z',
    i. Compute rx = tx · rx · rx, (arithmetic in GT)
    j. Compute ry = (ty · ry · ry) mod q,
    k. If $p_i = 1$ and $i \neq 0$ then
        i. Compute $w = (px \cdot Z^2 - X) \bmod q$,
        ii. Compute $v = (py \cdot Z^3 - Y) \bmod q$,
        iii. Compute $X' = (v^2 - w^3 - 2 \cdot X \cdot w^2) \bmod q$,
        iv. Compute $Y' = (v \cdot (X \cdot w^2 - X') - Y \cdot w^3) \bmod q$,
        v. Compute $Z' = (w \cdot Z) \bmod q$,
        vi. Compute ty = Z',
        vii. Compute $tx = Z' \cdot (Qy - py) - v \cdot (Qx - px)$, (arithmetic in GT)
        viii. Set X = X', Y = Y', Z = Z',
        ix. Compute rx = tx · rx, (arithmetic in GT)
        x. Compute ry = (ty · ry) mod q,
9. Compute $ry = ry^{-1} \bmod q$,
10. Compute r = rx · ry, (arithmetic in GT).
11. Compute d = finalExp(r).
12. Return d.

d = finalExp(r)

Input: r (an element in GT)

Output: d (an element in GT) where d = GT.exp(r, $(q^6-1)/p$)

Steps:

1. If we do not care about performance, we can simply do the following
   d = GT.exp(r, $(q^6 - 1)/p$).
2. Alternatively, we could do the following:
   a. Let r = (r[0], r[1]), where r[0] and r[1] are elements in Fq^d,
   b. Compute x = transform(r[0]), where x is an element in Fq^d,
   c. Compute y = transform(r[1]), where x is an element in Fq^d,
   d. Let t1, t2, t3, t4 be four variables in GT,
   e. t1 = (x, y), t2 = (r[0], –r[1]), t3 = (x, –y), t4 = (r[0], r[1]),
   f. d = (t1 · t2) / (t3 · t4),
   g. Compute d = GT.exp(d, $(q^2 - q + 1)/p$).
3. Return d.

b = transform(a)

Input: a (an element in Fq^d)

Output: b (an element in Fq^d)

Steps:

1. Assume a = (a[0], a[1], a[2]).
2. Let a = a[2] · t^2 + a[1] · t + a[0].
3. Compute b = (a[2] · t^{2q} + a[1] · t^q + a[0]) mod p(t), where p(t) is the irreducible polynomial defined in the parameters of Fq^d.
4. Return b.