Identity-Based Cryptography Standard (IBCS) #1:
Supersingular Curve Implementations of the BF and BB1 Cryptosystems

Status of This Memo

   This memo provides information for the Internet community.  It does
   not specify an Internet standard of any kind.  Distribution of this
   memo is unlimited.

IESG Note

   This document specifies two mathematical algorithms for identity
   based encryption (IBE).  Due to its specialized nature, this document
   experienced limited review within the IETF.  Readers of this RFC
   should carefully evaluate its value for implementation and
   deployment.

Abstract

   This document describes the algorithms that implement Boneh-Franklin
   (BF) and Boneh-Boyen (BB1) Identity-based Encryption.  This document
   is in part based on IBCS #1 v2 of Voltage Security's Identity-based
   Cryptography Standards (IBCS) documents, from which some irrelevant
   sections have been removed to create the content of this document.

Table of Contents

1.  Introduction

    This document provides a set of specifications for implementing
    identity-based encryption (IBE) systems based on bilinear pairings.
    Two cryptosystems are described: the IBE system proposed by Boneh and
    Franklin (BF) [BF], and the IBE system proposed by Boneh and Boyen
    (BB1) [BB1].  Fully secure and practical implementations are
    described for each system, comprising the core IBE algorithms as well
    as ancillary hybrid components used to achieve security against
    active attacks.  These specifications are restricted to a family of
    supersingular elliptic curves over finite fields of large prime
    characteristic, referred to as "type-1" curves (see Section 2.1).
    Implementations based on other types of curves currently fall outside
    the scope of this document.

    IBE is a public-key technology, but one which varies from other
    public-key technologies in a slight, yet significant way.  In
    particular, IBE keys are calculated instead of being generated
    randomly, which leads to a different architecture for a system using
    IBE than for a system using other public-key technologies.  An
    overview of these differences and how a system using IBE works is
    given in [IBEARCH].

    Identity-based encryption (IBE) is a public-key encryption technology
    that allows a public key to be calculated from an identity, and the
    corresponding private key to be calculated from the public key.
    Calculation of both the public and private keys in an IBE-based
    system can occur as needed, resulting in just-in-time key material.
    This contrasts with other public-key systems [P1363], in which keys
    are generated randomly and distributed prior to secure communication
    commencing.  The ability to calculate a recipient's public key, in
    particular, eliminates the need for the sender and receiver in an
    IBE-based messaging system to interact with each other, either
    directly or through a proxy such as a directory server, before
    sending secure messages.

    This document describes an IBE-based messaging system and how the
    components of the system work together.  The components required for
    a complete IBE messaging system are the following:

    o   a Private-key Generator (PKG).  The PKG contains the cryptographic
        material, known as a master secret, for generating an individual's
        IBE private key.  A PKG accepts an IBE user's private key request,
        and after successfully authenticating them in some way, returns
        the IBE private key.

o   a Public Parameter Server (PPS).  IBE System Parameters include
    publicly sharable cryptographic material, known as IBE public
    parameters, and policy information for the PKG.  A PPS provides a
    well-known location for secure distribution of IBE public
    parameters and policy information for the IBE PKG.

A logical architecture would be to have a PKG/PPS per name space,
such as a DNS zone.  The organization that controls the DNS zone
would also control the PKG/PPS and thus the determination of which
PKG/PSS to use when creating public and private keys for the
organization's members.  In this case the PPS URI can be uniquely
created by the form of the identity that it supports.  This
architecture would make it clear which set of public parameters to
use and where to retrieve them for a given identity.

IBE-encrypted messages can use standard message formats, such as the
Cryptographic Message Syntax (CMS) [CMS].  How to use IBE with CMS is
described in [IBECMS].

Note that IBE algorithms are used only for encryption, so if digital
signatures are required, they will need to be provided by an
additional mechanism.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
document are to be interpreted as described in [KEYWORDS].

1.1.  Sending a Message That Is Encrypted Using IBE

In order to send an encrypted message, an IBE user must perform the
following steps:

    1.  Obtain the recipient's public parameters.

        The recipient's IBE public parameters allow the creation of
        unique public and private keys.  A user of an IBE system is
        capable of calculating the public key of a recipient after he
        obtains the public parameters for their IBE system.  Once the
        public parameters are obtained, IBE-encrypted messages can be
        sent.

    2.  Construct and send an IBE-encrypted message.

        All that is needed, in addition to the IBE public parameters,
        is the recipient's identity in order to generate their public
        key for use in encrypting messages to them.  When this identity
        is the same as the identity that a message would be addressed
        to, then no more information is needed from a user to send

someone a secure message than is needed to send them an
unsecured message.  This is one of the major benefits of an
IBE-based secure messaging system.  Examples of identities can
be an individual, group, or role identifiers.

1.1.1.  Sender Obtains Recipient's Public Parameters

The sender of a message obtains the IBE public parameters that he
needs for calculating the IBE public key of the recipient from a PPS
that is hosted at a well-known URI.  The IBE public parameters
contain all of the information that the sender needs to create an
IBE-encrypted message except for the identity of the recipient.
[IBEARCH] describes the URI where a PPS is located, the format of IBE
public parameters, and how to obtain them.  The URI from which users
obtain IBE public parameters MUST be authenticated in some way; PPS
servers MUST support Transport Layer Security (TLS) 1.1 [TLS] to
satisfy this requirement and MUST verify that the subject name in the
server certificate matches the URI of the PPS.  [IBEARCH] also
describes the way in which identity formats are defined and a minimum
interoperable format that all PPSs and PKGs MUST support.  This step
is shown below in Figure 1.

```
                    IBE Public Parameter Request
              ----------------------------->
        Sender                                        PPS
              <----------------------------
                    IBE Public Parameters
```
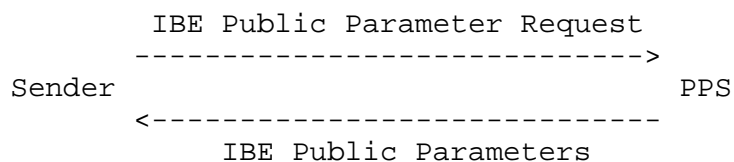
         Figure 1.  Requesting IBE Public Parameters

The sender of an IBE-encrypted message selects the PPS and
corresponding PKG based on his local security policy.  Different PPSs
may provide public parameters that specify different IBE algorithms
or different key strengths, for example, or require the use of PKGs
that require different levels of authentication before granting IBE
private keys.

1.1.2.  Construct and Send an IBE-Encrypted Message

To IBE-encrypt a message, the sender chooses a content encryption key
(CEK) and uses it to encrypt his message and then encrypts the CEK
with the recipient's IBE public key (for example, as described in
[CMS]).  This operation is shown below in Figure 2.  This document
describes the algorithms needed to implement two forms of IBE.
[IBECMS] describes how to use the Cryptographic Message Syntax (CMS)
to encapsulate the encrypted message along with the IBE information
that the recipient needs to decrypt the message.

```
       CEK ----> Sender ----> IBE-encrypted CEK

                     ^
                     |
                     |
                     |

            Recipient's Identity
          and IBE Public Parameters
```
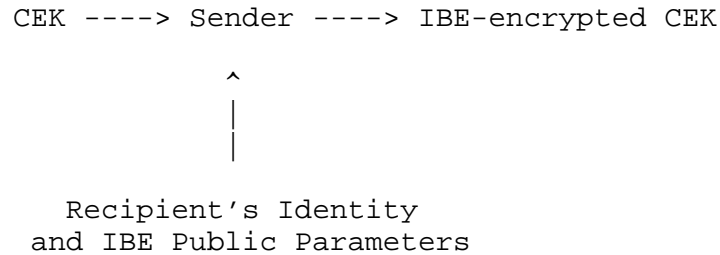
        Figure 2.  Using an IBE Public-Key Algorithm to Encrypt

1.2.  Receiving and Viewing an IBE-Encrypted Message

   In order to read an encrypted message, a recipient of an
   IBE-encrypted message parses the message (for example, as described
   in [IBECMS]).  This gives him the URI he needs to obtain the IBE
   public parameters required to perform IBE calculations as well as the
   identity that was used to encrypt the message.  Next, the recipient
   must carry out the following steps:

      1.  Obtain the recipient's public parameters.

          An IBE system's public parameters allow it to uniquely create
          public and private keys.  The recipient of an IBE-encrypted
          message can decrypt an IBE-encrypted message if he has both the
          IBE public parameters and the necessary IBE private key.  The
          PPS can also provide the URI of the PKG where the recipient of
          an IBE-encrypted message can obtain the IBE private keys.

      2.  Obtain the IBE private key from the PKG.

          To decrypt an IBE-encrypted message, in addition to the IBE
          public parameters, the recipient needs to obtain the private
          key that corresponds to the public key that the sender used.
          The IBE private key is obtained after successfully
          authenticating to a private key generator (PKG), a trusted
          third party that calculates private keys for users.  The
          recipient receives the IBE private key over an HTTPS
          connection.  The URI of a PKG MUST be authenticated in some
          way; PKG servers MUST support TLS 1.1 [TLS] to satisfy this
          requirement.

      3.  Decrypt the IBE-encrypted message.

          The IBE private key decrypts the CEK, which is then used to
          decrypt encrypted message.

The PKG may allow users other than the intended recipient to
receive some IBE private keys.  Giving a mail filtering
appliance permission to obtain IBE private keys on behalf of
users, for example, can allow the appliance to decrypt and scan
encrypted messages for viruses or other malicious features.

1.2.1.  Recipient Obtains Public Parameters from PPS

Before he can perform any IBE calculations related to the message
that he has received, the recipient of an IBE-encrypted message needs
to obtain the IBE public parameters that were used in the encryption
operation.  This operation is shown below in Figure 3.

```
                    IBE Public Parameter Request
                 ----------------------------->
        Recipient                                PPS
                 <----------------------------
                       IBE Public Parameters
```
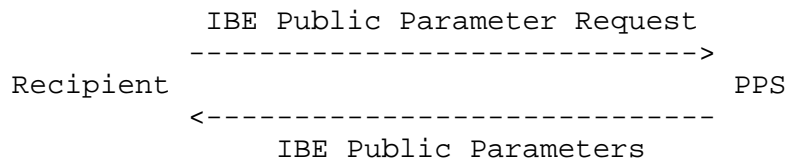
            Figure 3.  Requesting IBE Public Parameters

1.2.2.  Recipient Obtains IBE Private Key from PKG

To obtain an IBE private key, the recipient of an IBE-encrypted
message provides the IBE public key used to encrypt the message and
their authentication credentials to a PKG and requests the private
key that corresponds to the IBE public key.  Section 4 of this
document defines the protocol for communicating with a PKG as well as
a minimum interoperable way to authenticate to a PKG that all IBE
implementations MUST support.  Because the security of IBE private
keys is vital to the overall security of an IBE system, IBE private
keys MUST be transported to recipients over a secure protocol.  PKGs
MUST support TLS 1.1 [TLS] for transport of IBE private keys.  This
operation is shown below in Figure 4.

```
                    IBE Private Key Request
                 ---------------------------->
        Recipient                                PKG
                 <---------------------------
                       IBE Private Key
```

            Figure 4.  Obtaining an IBE Private Key

1.2.3.  Recipient Decrypts IBE-Encrypted Message

   After obtaining the necessary IBE private key, the recipient uses
   that IBE private key, and the corresponding IBE public parameters, to
   decrypt the CEK.  This operation is shown below in Figure 5.  He then
   uses the CEK to decrypt the encrypted message content (for example,
   as specified in [IBECMS]).

       IBE-encrypted CEK ----> Recipient ----> CEK

                                   ^
                                   |
                                   |

                            IBE Private Key
                        and IBE Public Parameters

          Figure 5.  Using an IBE Public-Key Algorithm to Decrypt

2.  Notation and Definitions

2.1.  Notation

   This section summarizes the notions and definitions regarding
   identity-based cryptosystems on elliptic curves.  The reader is
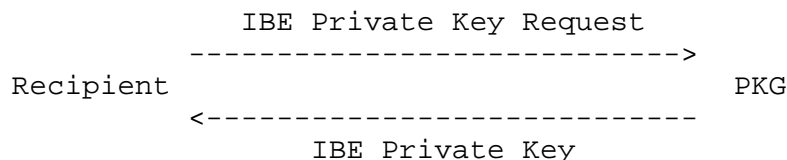   referred to [ECC] for the mathematical background and to [BF],
   [IBEARCH] regarding all notions pertaining to identity-based
   encryption.

   $F_p$ denotes finite field of prime characteristic p; $F_{p^2}$ denotes its
   extension field of degree 2.

   Let $E/F_p$: $y^2 = x^3 + a * x + b$ be an elliptic curve over $F_p$.  For
   an extension of degree 2, the curve $E/F_p$ defines a group $(E(F_{p^2})$,
   +), which is the additive group of points of affine coordinates (x,
   y) in $(F_{p^2})^2$ satisfying the curve equation over $F_{p^2}$, with null
   element, or point at infinity, denoted as 0.

   Let q be a prime such that $E(F_p)$ has a cyclic subgroup G1' of order
   q.

   Let G1'' be a cyclic subgroup of $E(F_{p^2})$ of order q, and G2 be a
   cyclic subgroup of $(F_{p^2})*$ of order p.

   Under these conditions, a mathematical construction known as the Tate
   pairing provides an efficiently computable map e: G1' x G1'' -> G2
   that is linear in both arguments and believed hard to invert [BF].
   If an efficiently computable non-rational endomorphism phi: G1' ->

G1'' is available for the selected elliptic curve on which the Tate
pairing is computed, then we can construct a function e': G1' x G1''
-> G2, defined as e'(A, B) = e(A, phi(B)), called the modified Tate
pairing.  We generically call a pairing either the Tate pairing e or
the modified Tate pairing e', depending on the chosen elliptic curve
used in a particular implementation.

The following additional notation is used throughout this document.

p - A 512-bit to 7680-bit prime, which is the order of the finite
field F_p.

F_p - The base finite field of order p over which the elliptic curve
of interest E/F_p is defined.

#G - The size of the set G.

F* - The multiplicative group of the non-zero elements in the field
F; e.g., (F_p)* is the multiplicative group of the finite field F_p.

E/F_p - The equation of an elliptic curve over the field F_p, which,
when p is neither 2 nor 3, is of the form E/F_p: $y^2 = x^3 + a * x + b$, for specified a, b in F_p.

0 - The null element of any additive group of points on an elliptic
curve, also called the point at infinity.

E(F_p) - The additive group of points of affine coordinates (x, y),
with x, y in F_p, that satisfy the curve equation E/F_p, including
the point at infinity 0.

q - A 160-bit to 512-bit prime that is the order of the cyclic
subgroup of interest in E(F_p).

k - The embedding degree of the cyclic subgroup of order q in E(F_p).
For type-1 curves this is always equal to 2.

F_p^2 - The extension field of degree 2 of the field F_p.

E(F_p^2) - The group of points of affine coordinates in F_p^2
satisfying the curve equation E/F_p, including the point at infinity
0.

Z_p - The additive group of integers modulo p.

lg - The base 2 logarithm function, so that $2^{lg(x)} = x$.

The term "object identifier" will be abbreviated "OID."

A Solinas prime is a prime of the form 2^a (+/-) 2^b (+/-) 1.

The following conventions are assumed for curve operations.

Point addition - If A and B are two points on a curve E, their sum is denoted as A + B.

Point multiplication - If A is a point on a curve, and n an integer, the result of adding A to itself a total of n times is denoted [n]A.

The following class of elliptic curves is exclusively considered for pairing operations in the present version of this document, which are referred to as "type-1" curves.

Type-1 curves - The class of curves of type-1 is defined as the class of all elliptic curves of equation $E/F_p$: $y^2 = x^3 + 1$ for all primes p congruent to 11 modulo 12.  This class forms a subclass of the class of supersingular curves.  These curves satisfy $\#E(F_p) = p + 1$, and the p points (x, y) in $E(F_p) \setminus \{0\}$ have the property that x = $(y^2 - 1)^{(1/3)}$ (mod p).  Type-1 curves always have an embedding degree k = 2.

Groups of points on type-1 curves are plentiful and easy to construct by random selection of a prime p of the appropriate form.  Therefore, rather than to standardize upon a small set of common values of p, it is henceforth assumed that all type-1 curves are freshly generated at random for the given cryptographic application (an example of such generation will be given in Algorithm 5.1.2 (BFsetup1) or Algorithm 6.1.2 (BBsetup1)).  Implementations based on different classes of curves are currently unsupported.

We assume that the following concrete representations of mathematical objects are used.

Base field elements - The p elements of the base field $F_p$ are represented directly using the integers from 0 to p - 1.

Extension field elements - The $p^2$ elements of the extension field $F_{p^2}$ are represented as ordered pairs of elements of $F_p$.  An ordered pair (a_0, a_1) is interpreted as the complex number $a_0 + a_1 * i$, where $i^2 = -1$.  This allows operations on elements of $F_{p^2}$ to be implemented as follows.  Suppose that a = (a_0, a_1) and b = (b_0, b_1) are elements of $F_{p^2}$.  Then a + b = ((a_0 + b_0)(mod p), (a_1 + b_1)(mod p)) and a * b = ((a_1 * b_1 - a_0 * b_0)(mod p), (a_1 * b_0 + a_0 * b_1)(mod p)).

Elliptic curve points - Points in E(F_p^2) with the point P = (x, y) in F_p^2 x F_p^2 satisfying the curve equation E/F_p.  Points not equal to 0 are internally represented using the affine coordinates (x, y), where x and y are elements of F_p^2.

## 2.2.  Definitions

The following terminology is used to describe an IBE system.

Public parameters - The public parameters are a set of common, system-wide parameters generated and published by the private key generator (PKG).

Master secret - The master secret is the master key generated and privately kept by the key server and used to generate the private keys of the users.

Identity - An identity is an arbitrary string, usually a human-readable unambiguous designator of a system user, possibly augmented with a time stamp and other attributes.

Public key - A public key is a string that is algorithmically derived from an identity.  The derivation may be performed by anyone, autonomously.

Private key - A private key is issued by the key server to correspond to a given identity (and the public key that derives from it) under the published set of public parameters.

Plaintext - Plaintext is an unencrypted representation, or in the clear, of any block of data to be transmitted securely.  For the present purposes, plaintexts are typically session keys, or sets of session keys, for further symmetric encryption and authentication purposes.

Ciphertext - Ciphertext is an encrypted representation of any block of data, including plaintext, to be transmitted securely.

## 3.  Basic Elliptic Curve Algorithms

This section describes algorithms for performing all needed basic arithmetic operations on elliptic curves.  The presentation is specialized to the type of curves under consideration for simplicity of implementation.  General algorithms may be found in [ECC].

3.1.  The Group Action in Affine Coordinates

3.1.1.  Implementation for Type-1 Curves

   Algorithm 3.1.1 (PointDouble1): adds a point to itself on a type-1
   elliptic curve.

   Input:

   o  A point A in E(F_p^2), with A = (x, y) or 0

   o  An elliptic curve E/F_p: y^2 = x^3 + 1

   Output:

   o  The point [2]A = A + A

   Method:

   1. If A = 0 or y = 0, then return 0

   2. Let lambda = (3 * x^2) / (2 * y)

   3. Let x' = lambda^2 - 2 * x

   4. Let y' = (x - x') * lambda - y

   5. Return (x', y')

   Algorithm 3.1.2 (PointAdd1): adds two points on a type-1 elliptic
   curve.

   Input:

   o  A point A in E(F_p^2), with A = (x_A, y_A) or 0

   o  A point B in E(F_p^2), with B = (x_B, y_B) or 0

   o  An elliptic curve E/F_p: y^2 = x^3 + 1

   Output:

   o  The point A + B

   Method:

   1. If A = 0, return B

2. If B = 0, return A

3. If x_A = x_B:

   (a) If y_A = -y_B, return 0

   (b) Else return [2]A computed using Algorithm 3.1.1 (PointDouble1)

4.  Otherwise:

   (a) Let lambda = (y_B - y_A) / (x_B - x_A)

   (b) Let x' = lambda^2 - x_A - x_B

   (c) Let y' = (x_A - x') * lambda - y_A

   (d) Return (x', y')

## 3.2.  Point Multiplication

Algorithm 3.2.1 (SignedWindowDecomposition): computes the signed
m-ary window representation of a positive integer [ECC].

Input:

o  An integer k > 0, where k has the binary representation k =
   {Sum(k_j * 2^j, for j = 0 to l} where each k_j is either 0 or 1
   and k_l = 0

o  An integer window bit-size r > 0

Output:

o  An integer d and the unique d-element sequence {(b_i, e_i), for i
   = 0 to d - 1} such that k = {Sum(b_i * 2^(e_i), for i = 0 to d -
   1}, each b_i = +/- 2^j for some 0 < j <= r - 1 and each e_i is a
   non-negative integer

Method:

1. Let d = 0

2. Let j = 0

3. While j <= l, do:

   (a) If k_j = 0, then:

            i. Let j = j + 1

        (b) Else:

            i. Let t = min{l, j + r - 1}

           ii. Let h_d = (k_t, k_(t - 1), ..., k_j) (base 2)

          iii. If h_d > 2^(r - 1), then:

                A. Let b_d = h_d - 2^r

                B. Increment the number (k_l, k_(l-1),...,k_j) (base 2) by 1

           iv.  Else:

                A.  Let b_d = h_d

            v.  Let e_d = j

           vi.  Let d = d + 1

          vii.  Let j = t + 1

    4.  Return d and the sequence {(b_0, e_0), ...,
        (b_(d - 1), e_(d - 1))}

    Algorithm 3.2.2 (PointMultiply): scalar multiplication on an elliptic
    curve using the signed m-ary window method.

    Input:

    o  A point A in E(F_p^2)

    o  An integer l > 0

    o  An elliptic curve E/F_p: y^2 = x^3 + a * x + b

    Output:

    o  The point [l]A

    Method:

    1.  (Window decomposition)

        (a) Let r > 0 be an integer (fixed) bit-wise window size,
            e.g., r = 5

   (b) Let l' = l where l = {Sum(l_j * 2^j), for j = 0 to
       len_l} is the binary expansion of l, where len_l =
       Ceiling(lg(l))

   (c) Compute (d, {(b_i, e_i), for i = 0 to d - 1} =
       SignedWindowDecomposition(l, r), the signed 2^r-ary window
       representation of l using Algorithm 3.2.1
       (SignedWindowDecomposition)

2.  (Precomputation)

   (a) Let A_1 = A

   (b) Let A_2 = [2]A, using Algorithm 3.1.1 (PointDouble1)

   (c) For i = 1 to 2^(r - 2) - 1, do:

      i.  Let A_(2 * i + 1) = A_(2 * i - 1) + A_2 using
          Algorithm 3.1.2 (PointAdd1)

   (d) Let Q = A_(b_(d - 1))

3.  Main loop

   (a) For i = d - 2 to 0 by -1, do:

      i.  Let Q = [2^(e_(i + 1) - e_i)]Q, using repeated
          applications of Algorithm 3.1.1 (PointDouble1)
          e_(i + 1) - e_i times

      ii. If b_i > 0, then:

          A.  Let Q = Q + A_(b_i) using Algorithm 3.1.2
              (PointAdd1)

      iii. Else:

          A.  Let Q = Q - A_(-(b_i)) using Algorithm 3.1.2
              (PointAdd1)

   (b) Calculate Q = [2^(e_0)]Q using repeated applications of
       Algorithm 3.1.1 (PointDouble1) e_0 times

4.  Return Q.

3.3.  Operations in Jacobian Projective Coordinates

3.3.1.  Implementation for Type-1 Curves

   Algorithm 3.3.1 (ProjectivePointDouble1): adds a point to itself in
   Jacobian projective coordinates for type-1 curves.

   Input:

   o  A point $(x, y, z) = A$ in $E(F_p^2)$ in Jacobian projective
      coordinates

   o  An elliptic curve $E/F_p$: $y^2 = x^3 + 1$

   Output:

   o  The point $[2]A$ in Jacobian projective coordinates

   Method:

   1. If $z = 0$ or $y = 0$, return $(0, 1, 0) = 0$, otherwise:

   2. Let lambda_1 $= 3 * x^2$

   3. Let $z' = 2 * y * z$

   4. Let lambda_2 $= y^2$

   5. Let lambda_3 $= 4 *$ lambda_2 $* x$

   6. Let $x' =$ lambda_1$^2 - 2 *$ lambda_3

   7. Let lambda_4 $= 8 *$ lambda_2$^2$

   8. Let $y' =$ lambda_1 $* ($lambda_3 $- x') -$ lambda_4

   9. Return $(x', y', z')$

   Algorithm 3.3.2 (ProjectivePointAccumulate1): adds a point in affine
   coordinates to an accumulator in Jacobian projective coordinates, for
   type-1 curves.

   Input:

   o  A point $(x\_A, y\_A, z\_A) = A$ in $E(F_p^2)$ in Jacobian
      projective coordinates

   o  A point (x_B, y_B) = B in E(F_p^2) \ {0} in affine
      coordinates

   o  An elliptic curve E/F_p: y^2 = x^3 + 1

   Output:

   o  The point A + B in Jacobian projective coordinates

   Method:

   1. If z_A = 0, return (x_B, y_B, 1) = B, otherwise:

   2. Let lambda_1 = z_A^2

   3. Let lambda_2 = lambda_1 * x_B

   4. Let lambda_3 = x_A - lambda_2

   5. If lambda_3 = 0, then return (0, 1, 0), otherwise:

   6. Let lambda_4 = lambda_3^2

   7. Let lambda_5 = lambda_1 * y_B * z_A

   8. Let lambda_6 = lambda_4 - lambda_5

   9. Let lambda_7 = x_A + lambda_2

   10. Let lambda_8 = y_A + lambda_5

   11. Let x' = lambda_6^2 - lambda_7 * lambda_4

   12. Let lambda_9 = lambda_7 * lambda_4 - 2 * x'

   13. Let y' = (lambda_9 * lambda_6 -

       lambda_8 * lambda_3 * lambda_4) / 2

   14. Let z' = lambda_3 * z_A

   15. Return (x', y', z')

3.4.  Divisors on Elliptic Curves

3.4.1.  Implementation in F_p^2 for Type-1 Curves

   Algorithm 3.4.1 (EvalVertical1): evaluates the divisor of a vertical
   line on a type-1 elliptic curve.

   Input:

   o  A point B in E(F_p^2) with B != 0

   o  A point A in E(F_p)

   o  A description of a type-1 elliptic curve E/F_p

   Output:

   o  An element of F_p^2 that is the divisor of the vertical line going
      through A evaluated at B

   Method:

   1. Let r = x_B - x_A

   2. Return r

   Algorithm 3.4.2 (EvalTangent1): evaluates the divisor of a tangent on
   a type-1 elliptic curve.

   Input:

   o  A point B in E(F_p^2) with B != 0

   o  A point A in E(F_p)

   o  A description of a type-1 elliptic curve E/F_p

   Output:

   o  An element of F_p^2 that is the divisor of the line tangent to A
      evaluated at B

   Method:

   1. (Special cases)

      (a) If A = 0, return 1

   (b) If y_A = 0, return EvalVertical1(B, A) using Algorithm 3.4.1
       (EvalVertical1)

2. (Line computation)

   (a) Let a = -3 * (x_A)^2

   (b) Let b = 2 * y_A

   (c) Let c = -b * y_A - a * x_A

3. (Evaluation at B)

   (a) Let r = a * x_B + b * y_B + c

4. Return r

Algorithm 3.4.3 (EvalLine1): evaluates the divisor of a line on a
type-1 elliptic curve.

Input:

o  A point B in E(F_p^2) with B != 0

o  Two points A', A'' in E(F_p)

o  A description of a type-1 elliptic curve E/F_p

Output:

o  An element of F_p^2 that is the divisor of the line going through
   A' and A'' evaluated at B

Method:

1. (Special cases)

   (a) If A' = 0, return EvalVertical1(B, A'') using Algorithm 3.4.1
       (EvalVertical1)

   (b) If A'' = 0, return EvalVertical1(B, A') using Algorithm 3.4.1
       (EvalVertical1)

   (c) If A' = -A'', return EvalVertical1(B, A') using Algorithm
       3.4.1 (EvalVertical1)

   (d) If A' = A'', return EvalTangent1(B, A') using Algorithm 3.4.2
       (EvalTangent1)

   2. (Line computation)

        (a) Let a = y_A' - y_A''

        (b) Let b = x_A'' - x_A'

        (c) Let c = -b * y_A' - a * x_A'

   3. (Evaluation at B)

        (a) Let r = a * x_B + b * y_B + c

   4. Return r

3.5.  The Tate Pairing

3.5.1.  Tate Pairing Calculation

   Algorithm 3.5.1 (Tate): computes the Tate pairing on an elliptic
   curve.

   Input:

   o  A point A of order q in E(F_p)

   o  A point B of order q in E(F_p^2)

   o  A description of an elliptic curve E/F_p such that E(F_p) and
      E(F_p^2) have a subgroup of order q

   Output:

   o  The value e(A, B) in F_p^2, computed using the Miller algorithm

   Method:

   1. For a type-1 curve E, execute Algorithm 3.5.2 (TateMillerSolinas)

3.5.2.  The Miller Algorithm for Type-1 Curves

   Algorithm 3.5.2 (TateMillerSolinas): computes the Tate pairing on a
   type-1 elliptic curve.

   Input:

   o  A point A of order q in E(F_p)

   o  A point B of order q in E(F_p^2)

o   A description of a type-1 supersingular elliptic curve E/F_p such
    that E(F_p) and E(F_p^2) have a subgroup of Solinas prime order q
    where q = 2^a + s * 2^b + c, where c and s are limited to the
    values +/-1

Output:

o   The value e(A, B) in F_p^2, computed using the Miller algorithm

Method:

1.  (Initialization)

    (a) Let v_num = 1 in F_p^2

    (b) Let v_den = 1 in F_p^2

    (c) Let V = (x_V , y_V , z_V ) = (x_A, y_A, 1) in (F_p)^3, being
        the representation of (x_A, y_A) = A using Jacobian projective
        coordinates

    (d) Let t_num = 1 in F_p^2

    (e) Let t_den = 1 in F_p^2

2.  (Calculation of the (s * 2^b) contribution)

    (a) (Repeated doublings) For n = 0 to b - 1:

         i. Let t_num = t_num^2

        ii. Let t_den = t_den^2

       iii. Let t_num = t_num * EvalTangent1(B, (x_V / z_V^2, y_V /
            z_V^3)) using Algorithm 3.4.2 (EvalTangent1)

        iv. Let V = (x_V , y_V , z_V ) = [2]V  using Algorithm 3.3.1
            (ProjectivePointDouble1)

         v. Let t_den = t_den * EvalVertical1(B, (x_V / z_V^2, y_V /
            z_V^3)using Algorithm 3.4.1 (EvalVertical1)

   (b) (Normalization)

      i. Let V_b = (x_(V_b) , y_(V_b))

         = (x_V / z_V^2, s * y_V / z_V^3) in (F_p)^2,

         resulting in a point V_b in E(F_p)

   (c) (Accumulation) Selecting on s:

      i. If s = -1:

         A. Let v_num = v_num * t_den

         B. Let v_den = v_den * t_num * EvalVertical1(B, (x_V /
            z_V^2, y_V / z_V^3))) using Algorithm 3.4.1
            (EvalVertical1)

      ii. If s = 1:

         A. Let v_num = v_num * t_num

         B. Let v_den = v_den * t_den

 3. (Calculation of the 2^a contribution)

   (a) (Repeated doublings) For n = b to a - 1:

      i. Let t_num = t_num^2

      ii. Let t_den = t_den^2

      iii. Let t_num = t_num * EvalTangent1(B, (x_V / z_V^2, y_V /
           z_V^3))) using Algorithm 3.4.2 (EvalTangent1)

      iv. Let V = (x_V , y_V , z_V) = [2]V  using Algorithm 3.3.1
          (ProjectivePointDouble1)

      v. Let t_den = t_den * EvalVertical1(B, (x_V / z_V^2, y_V /
         z_V^3))) using Algorithm 3.4.1 (EvalVertical1)

   (b) (Normalization)

      i. Let V_a = (x_(V_a) , y_(V_a)) =

         (x_V /z_V^2, s * x_V / z_V^3) in (F_p)^2,

         resulting in a point V_a in E(F_p)

        (c) (Accumulation)

            i. Let v_num = v_num * t_num

           ii. Let v_den = v_den * t_den

    4. (Correction for the (s * 2^b) and (c) contributions)

        (a) Let v_num = v_num * EvalLine1(B, V_a, V_b) using Algorithm
            3.4.3 (EvalLine1)

        (b) Let v_den = v_den * EvalVertical1(B, V_a + V_b) using
            Algorithm 3.4.1 (EvalVertical1)

        (c) If c = -1, then:

            i. Let v_den = v_den * EvalVertical1(B, A) using Algorithm
               3.4.1 (EvalVertical1)

    5. (Correcting exponent)

        (a) Let eta = (p^2 - 1) / q

    6. (Final result)

        (a) Return (v_num / v_den)^eta

4.  Supporting Algorithms

    This section describes a number of supporting algorithms for encoding
    and hashing.

4.1.  Integer Range Hashing

4.1.1.  Hashing to an Integer Range

    HashToRange(s, n, hashfcn) takes a string s, an integer n, and a
    cryptographic hash function hashfcn as input and returns an integer
    in the range 0 to n - 1 by cryptographic hashing.  The input n MUST
    be less than 2^(hashlen), where hashlen is the number of octets
    comprising the output of the hash function hashfcn.  HashToRange is
    based on Merkle's method for hashing [MERKLE], which is provably as
    secure as the underlying hash function hashfcn.

    Algorithm 4.1.1 (HashToRange): cryptographically hashes strings to
    integers in a range.

   Input:

   o  A string s of length |s| octets

   o  A positive integer n represented as Ceiling(lg(n) / 8) octets.

   o  A cryptographic hash function hashfcn

   Output:

   o  A positive integer v in the range 0 to n - 1

   Method:

   1. Let hashlen be the number of octets comprising the output of
      hashfcn

   2. Let $v_0 = 0$

   3. Let $h_0 = 0x00...00$, a string of null octets with a length of
      hashlen

   4. For i = 1 to 2, do:

      (a) Let $t_i = h_{(i-1)} \|$ s, which is the (|s| + hashlen)- octet
          string concatenation of the strings $h_{(i-1)}$ and s

      (b) Let $h_i = $ hashfcn($t_i$), which is a hashlen-octet string
          resulting from the hash algorithm hashfcn on the input $t_i$

      (c) Let $a_i = $ Value($h_i$) be the integer in the range 0 to
          $256^{hashlen} - 1$ denoted by the raw octet string $h_i$
          interpreted in the unsigned big-endian convention

      (d) Let $v_i = 256^{hashlen} * v_{(i-1)} + a_i$

   5. Let $v = v_l$ (mod n)

4.2.  Pseudo-Random Byte Generation by Hashing

4.2.1.  Keyed Pseudo-Random Bytes Generator

   HashBytes(b, p, hashfcn) takes an integer b, a string p, and a
   cryptographic hash function hashfcn as input and returns a b-octet
   pseudo-random string r as output.  The value of b MUST be less than
   or equal to the number of bytes in the output of hashfcn.  HashBytes
   is based on Merkle's method for hashing [MERKLE], which is provably
   as secure as the underlying hash function hashfcn.

   Algorithm 4.2.1 (HashBytes): keyed cryptographic pseudo-random bytes
   generator.

   Input:

   o   An integer b

   o   A string p

   o   A cryptographic hash function hashfcn

   Output:

   o   A string r comprising b octets

   Method:

   1. Let hashlen be the number of octets comprising the output of
      hashfcn

   2. Let K = hashfcn(p)

   3. Let $h_0$ = 0x00...00, a string of null octets with a length of
      hashlen

   4. Let l = Ceiling(b / hashlen)

   5. For each i in 1 to l, do:

      (a) Let $h_i$ = hashfcn($h_{(i - 1)}$)

      (b) Let $r_i$ = hashfcn($h_i$ || K), where $h_i$ || K is the (2 *
          hashlen)-octet concatenation of $h_i$ and K

   6. Let r = LeftmostOctets(b, $r_1$ || ... || $r_l$), i.e., r is formed as
      the concatenation of the $r_i$, truncated to the desired number of
      octets

4.3.   Canonical Encodings of Extension Field Elements

4.3.1.   Encoding an Extension Element as a String

   Canonical(p, k, o, v) takes an element v in $F_p^k$, and returns a
   canonical octet string of fixed length representing v.  The parameter
   o MUST be either 0 or 1, and specifies the ordering of the encoding.

   Algorithm 4.3.1 (Canonical): encodes elements of an extension field
   $F_p^2$ as strings.

   Input:

   o  An element v in F_p^2

   o  A description of F_p^2

   o  An ordering parameter o, either 0 or 1

   Output:

   o  A fixed-length string s representing v

   Method:

   1. For a type-1 curve, execute Algorithm 4.3.2 (Canonical1)

4.3.2.  Type-1 Curve Implementation

   Canonical1(p, o, v) takes an element v in F_p^2 and returns a
   canonical representation of v as an octet string s of fixed size.
   The parameter o MUST be either 0 or 1, and specifies the ordering of
   the encoding.

   Algorithm 4.3.2 (Canonical1): canonically represents elements of an
   extension field F_p^2.

   Input:

   o  An element v in F_p^2

   o  A description of p, where p is congruent to 3 modulo 4

   o  A ordering parameter o, either 0 or 1

   Output:

   o  A string s of size 2 * Ceiling(lg(p) / 8) octets

   Method:

   1. Let l = Ceiling(lg(p) / 8), the number of octets needed to
      represent integers in Z_p

   2. Let v = a + b * i, where i^2 = -1

   3. Let a_(256^l) be the big-endian zero-padded fixed-length octet
      string representation of a in Z_p

   4. Let b_(256^l) be the big-endian zero-padded fixed-length octet
      string representation of b in Z_p

   5. Depending on the choice of ordering o:

      (a) If o = 0, then let s = a_(256^l) || b_(256^l), which is the
          concatenation of a_(256^l) followed by b_(256^l)

      (b) If o = 1, then let s = b_(256^l) || a_(256^l), which is the
          concatenation of b_(256^l) followed by a_(256^l)

   6. Return s

4.4.  Hashing onto a Subgroup of an Elliptic Curve

4.4.1.  Hashing a String onto a Subgroup of an Elliptic Curve

   HashToPoint(E, p, q, id, hashfcn) takes an identity string id, the
   description of a subgroup of prime order q in E(F_p) or E(F_p^2), and
   a cryptographic hash function hashfcn and returns a point Q_id of
   order q in E(F_p) or E(F_p^2).

   Algorithm 4.4.1 (HashToPoint): cryptographically hashes strings to
   points on elliptic curves.

   Input:

   o  An elliptic curve E

   o  A prime p

   o  A prime q

   o  A string id

   o  A cryptographic hash function hashfcn

   Output:

   o  A point Q_id = (x, y) of order q n E(F_p)

   Method:

   1. For a type-1 curve E, execute Algorithm 4.4.2 (HashToPoint1)

4.4.2.  Type-1 Curve Implementation

   HashToPoint1(p, q, id, hashfcn) takes an identity string id and the
   description of a subgroup of order q in E(F_p), where E: y^2 = x^3 +
   1 with p congruent to 11 modulo 12, and returns a point Q_id of order
   q in E(F_p) that is calculated using the cryptographic hash function
   hashfcn.  The parameters p, q and hashfcn MUST be part of a valid set
   of public parameters as defined in Section 5.1.2 or Section 6.1.2.

   Algorithm 4.4.2 (HashToPoint1): cryptographically hashes strings to
   points on type-1 curves.

   Input:

   o  A prime p

   o  A prime q

   o  A string id

   o  A cryptographic hash function hashfcn

   Output:

   o  A point Q_id of order q in E(F_p)

   Method:

   1. Let y = HashToRange(id, p, hashfcn), using Algorithm 4.1.1
      (HashToRange), an element of F_p

   2. Let x = (y^2 - 1)^((2 * p - 1) / 3) modulo p, an element of F_p

   3. Let Q' = (x, y), a non-zero point in E(F_p)

   4. Let Q = [(p + 1) / q ]Q', a point of order q in E(F_p)

4.5.  Bilinear Mapping

4.5.1.  Regular or Modified Tate Pairing

   Pairing(E, p, q, A, B) takes two points A and B, both of order q,
   and, in the type-1 case, returns the modified pairing e'(A, phi(B))
   in F_p^2 where A and B are both in E(F_p).

   Algorithm 4.5.1 (Pairing): computes the regular or modified Tate
   pairing depending on the curve type.

Input:

o  A description of an elliptic curve E/F_p such that E(F_p) and
   E(F_p^2) have a subgroup of order q

o  Two points A and B of order q in E(F_p) or E(F_p^2)

Output:

o  On supersingular curves, the value of e'(A, B) in F_p^2 where A
   and B are both in E(F_p)

Method:

1. If E is a type-1 curve, execute Algorithm 4.5.2 (Pairing1)

4.5.2.  Type-1 Curve Implementation

   Algorithm 4.5.2 (Pairing1): computes the modified Tate pairing on
   type-1 curves.  The values of p and q MUST be part of a valid set of
   public parameters as defined in Section 5.1.2 or Section 6.1.2.

   Input:

o  A curve E/F_p: y^2 = x^3 + 1 where p is congruent to 11 modulo 12
   and E(F_p) has a subgroup of order q

o  Two points A and B of order q in E(F_p)

   Output:

o  The value of e'(A, B) = e(A, phi(B)) in F_p^2

   Method:

1. Compute B' = phi(B), as follows:

   (a) Let (x, y) in F_p x F_p be the coordinates of B in E(F_p)

   (b) Let zeta = (a_zeta , b_zeta), where a_zeta = (p – 1) / 2 and
       b_zeta = 3^((p + 1) / 4) (mod p), an element of F_p^2

   (c) Let x' =  x * zeta in F_p^2

   (d) Let B' = (x', y) in F_p^2 x F_p

   2. Compute the Tate pairing e(A, B') = e(A, phi(B)) in F_p^2 using
      the Miller method, as in Algorithm 3.5.1 (Tate) described in
      Section 3.5

4.6.  Ratio of Bilinear Pairings

4.6.1.  Ratio of Regular or Modified Tate Pairings

   PairingRatio(E, p, q, A, B, C, D) takes four points as input and
   computes the ratio of the two bilinear pairings, Pairing(E, p, q, A,
   B) / Pairing(E, p, q, C, D), or, equivalently, the product,
   Pairing(E, p, q, A, B) * Pairing(E, p, q, C, -D).

   On type-1 curves, all four points are of order q in E(F_p), and the
   result is an element of order q in the extension field F_p^2 .

   The motivation for this algorithm is that the ratio of two pairings
   can be calculated more efficiently than by computing each pairing
   separately and dividing one into the other, since certain
   calculations that would normally appear in each of the two pairings
   can be combined and carried out at once.  Such calculations include
   the repeated doublings in steps 2(a)i, 2(a)ii, 3(a)i, and 3(a)ii of
   Algorithm 3.5.2 (TateMillerSolinas), as well as the final
   exponentiation in step 6(a) of Algorithm 3.5.2 (TateMillerSolinas).

   Algorithm 4.6.1 (PairingRatio): computes the ratio of two regular or
   modified Tate pairings depending on the curve type.

   Input:

   o  A description of an elliptic curve E/F_p such that E(F_p) and
      E(F_p^2) have a subgroup of order q

   o  Four points A, B, C, and D, of order q in E(F_p) or E(F_p^2)

   Output:

   o  On supersingular curves, the value of e'(A, B) / e'(C, D) in F_p^2
      where A, B, C, D are all in E(F_p)

   Method:

   1. If E is a type-1 curve, execute Algorithm 4.6.2 (PairingRatio1)

4.6.2.  Type-1 Curve Implementation

   Algorithm 4.6.2 (PairingRatio1): computes the ratio of two modified
   Tate pairings on type-1 curves.  The values of p and q MUST be part
   of a valid set of public parameters as defined in Section 5.1.2 or
   Section 6.1.2.

   Input:

   o  A curve E/F_p: y^2 = x^3 + 1, where p is congruent to 11 modulo 12
      and E(F_p) has a subgroup of order q

   o  Four points A, B, C, and D of order q in E(F_p)

   Output:

   o  The value of e'(A, B) / e'(C, D) = e(A, phi(B)) / e(C, phi(D)) =
      e(A, phi(B)) * e(-C, phi(D)), in F_p^2

   Method:

   1. The step-by-step description of the optimized algorithm is omitted
      in this normative specification

   The correct result can always be obtained, although more slowly, by
   computing the product of pairings Pairing1(E, p, q, A, B) *
   Pairing1(E, p, q, -C, D) by using two invocations of Algorithm 4.5.2
   (Pairing1).

5.  The Boneh-Franklin BF Cryptosystem

   This chapter describes the algorithms constituting the Boneh-Franklin
   identity-based cryptosystem as described in [BF].

5.1.  Setup

5.1.1.  Master Secret and Public Parameter Generation

   Algorithm 5.1.1 (BFsetup): randomly selects a master secret and the
   associated public parameters.

   Input:

   o  An integer version number

   o  A security parameter n (MUST take values either 1024, 2048, 3072,
      7680, 15360)

Output:

o  A set of public parameters (version, E, p, q, P, P_pub, hashfcn)

o  A corresponding master secret s

Method:

1. Depending on the selected type t:

   (a) If version = 2, then execute Algorithm 5.1.2 (BFsetup1)

2. The resulting master secret and public parameters are separately
   encoded as per the application protocol requirements

5.1.2.  Type-1 Curve Implementation

   BFsetup1 takes a security parameter n as input.  For type-1 curves,
   the scale of n corresponds to the modulus bit-size believed [BF] of
   comparable security in the classical Diffie-Hellman or RSA public-key
   cryptosystems.

   Algorithm 5.1.2 (BFsetup1): establishes a master secret and public
   parameters for type-1 curves.

   Input:

   o  A security parameter n, which MUST be either 1024, 2048, 3072,
      7680 or 15360

   Output:

   o  A set of common public parameters (version, p, q, P, Ppub,
      hashfcn)

   o  A corresponding master secret s

   Method:

1. Set the version to version = 2.

2. Determine the subordinate security parameters $n_p$ and $n_q$ as
   follows:

   (a) If n = 1024, then let $n_p$ = 512, $n_q$ = 160, hashfcn =
       1.3.14.3.2.26 (SHA-1 [SHA]

(b) If n = 2048, then let n_p = 1024, n_q = 224, hashfcn =
    2.16.840.1.101.3.4.2.4 (SHA-224 [SHA])

(c) If n = 3072, then let n_p = 1536, n_q = 256, hashfcn =
    2.16.840.1.101.3.4.2.1 (SHA-256 [SHA])

(d) If n = 7680, then let n_p = 3840, n_q = 384, hashfcn =
    2.16.840.1.101.3.4.2.2 (SHA-384 [SHA])

(e) If n = 15360, then let n_p = 7680, n_q = 512, hashfcn =
    2.16.840.1.101.3.4.2.3 (SHA-512 [SHA])

3. Construct the elliptic curve and its subgroup of interest, as
   follows:

   (a) Select an arbitrary n_q-bit Solinas prime q

   (b) Select a random integer r such that p = 12 * r * q - 1 is an
       n_p-bit prime

4. Select a point P of order q in E(F_p), as follows:

   (a) Select a random point P' of coordinates (x', y') on the curve
       E/F_p: $y^2 = x^3 + 1 \pmod p$

   (b) Let P = [12 * r]P'

   (c) If P = 0, then start over in step 3a

5. Determine the master secret and the public parameters as follows:

   (a) Select a random integer s in the range 2 to q - 1

   (b) Let P_pub = [s]P

6. (version, E, p, q, P, P_pub) are the public parameters where E:
   $y^2 = x^3 + 1$ is represented by the OID 2.16.840.1.114334.1.1.1.1.

7. The integer s is the master secret

5.2.  Public Key Derivation

5.2.1.  Public Key Derivation from an Identity and Public Parameters

   BFderivePubl takes an identity string id and a set of public
   parameters, and it returns a point Q_id.  The public parameters used
   MUST be a valid set of public parameters as defined by Section 5.1.2.

Algorithm 5.2.1 (BFderivePubl): derives the public key corresponding
to an identity string.

Input:

o  An identity string id

o  A set of public parameters (version, E, p, q, P, P_pub, hashfcn)

Output:

o  A point Q_id of order q in E(F_p) or E(F_p^2)

Method:

1. Q_id = HashToPoint(E, p, q, id, hashfcn), using Algorithm 4.4.1
   (HashToPoint)

## 5.3.  Private Key Extraction

## 5.3.1.  Private Key Extraction from an Identity, a Set of Public
        Parameters and a Master Secret

BFextractPriv takes an identity string id, a set of public
parameters, and corresponding master secret, and it returns a point
S_id.  The public parameters used MUST be a valid set of public
parameters as defined by Section 5.1.2.

Algorithm 5.3.1 (BFextractPriv): extracts the private key
corresponding to an identity string.

Input:

o  An identity string id

o  A set of public parameters (version, E, p, q, P, P_pub, hashfcn)

Output:

o  A point S_id of order q in E(F_p)

Method:

1. Let Q_id = HashToPoint(E, p, q, id, hashfcn) using Algorithm 4.4.1
   (HashToPoint)

2. Let S_id = [s]Q_id

5.4.  Encryption

5.4.1.  Encrypt a Session Key Using an Identity and Public Parameters

   BFencrypt takes three inputs: a public parameter block, an identity
   id, and a plaintext m.  The plaintext MUST be a random symmetric
   session key.  The public parameters used MUST be a valid set of
   public parameters as defined by Section 5.1.2.

   Algorithm 5.4.1 (BFencrypt): encrypts a random session key for an
   identity string.

   Input:

   o  A plaintext string m of size |m| octets

   o  A recipient identity string id

   o  A set of public parameters (version, E, p, q, P, P_pub, hashfcn)

   Output:

   o  A ciphertext tuple (U, V, W) in E(F_p) x {0, ... , 255}^hashlen x
      {0, ... , 255}^|m|

   Method:

   1. Let hashlen be the length of the output of the cryptographic hash
      function hashfcn from the public parameters.

   2. Q_id = HashToPoint(E, p, q, id, hashfcn), using Algorithm 4.4.1
      (HashToPoint), which results in a point of order q in E(F_p)

   3. Select a random hashlen-bit vector rho, represented as (hashlen /
      8)-octet string in big-endian convention

   4. Let t = hashfcn(m), a hashlen-octet string resulting from applying
      the hashfcn algorithm to the input m

   5. Let l = HashToRange(rho || t, q, hashfcn), an integer in the range
      0 to q – 1 resulting from applying Algorithm 4.1.1 (HashToRange)
      to the (2 * hashlen)-octet concatenation of rho and t

   6. Let U = [l]P, which is a point of order q in E(F_p)

   7. Let theta = Pairing(E, p, q, P_pub, Q_id), which is an element of
      the extension field F_p^2 obtained using the modified Tate pairing
      of Algorithm 4.5.1 (Pairing)

8.  Let theta' = theta^l, which is theta raised to the power of l in
    F_p^2

9.  Let z = Canonical(p, k, 0, theta'), using Algorithm 4.3.1
    (Canonical), the result of which is a canonical string
    representation of theta'

10. Let w = hashfcn(z) using the hashfcn hashing algorithm, the
    result of which is a hashlen-octet string

11. Let V = w XOR rho, which is the hashlen-octet long bit-wise XOR
    of w and rho

12. Let W = HashBytes(|m|, rho, hashfcn) XOR m, which is the bit-wise
    XOR of m with the first |m| octets of the pseudo-random bytes
    produced by Algorithm 4.2.1 (HashBytes) with seed rho

13. The ciphertext is the triple (U, V, W)

## 5.5.  Decryption

### 5.5.1.  Decrypt an Encrypted Session Key Using Public Parameters, a Private Key

BFdecrypt takes three inputs: a public parameter block, a private key
block key, and a ciphertext parsed as (U', V', W').  The public
parameters used MUST be a valid set of public parameters as defined
by Section 5.1.2.

Algorithm 5.5.1 (BFdecrypt): decrypts an encrypted session key using
a private key.

Input:

o  A private key point S_id of order q in E(F_p)

o  A ciphertext triple (U, V, W) in E(F_p) x {0, ... , 255}^hashlen x
   {0, ... , 255}*

o  A set of public parameters (version, E, p, q, P, P_pub, hashfcn)

Output:

o  A decrypted plaintext m, or an invalid ciphertext flag

Method:

1. Let hashlen be the length of the output of the hash function
   hashlen measured in octets

2. Let theta = Pairing(E, p ,q, U, S_id) by applying the modified
   Tate pairing of Algorithm 4.5.1 (Pairing)

3. Let z = Canonical(p, k, 0, theta) using Algorithm 4.3.1
   (Canonical), the result of which is a canonical string
   representation of theta

4. Let w = hashfcn(z) using the hashfcn hashing algorithm, the result
   of which is a hashlen-octet string

5. Let rho = w XOR V, the bit-wise XOR of w and V

6. Let m = HashBytes(|W|, rho, hashfcn) XOR W, which is the bit-wise
   XOR of m with the first |W| octets of the pseudo-random bytes
   produced by Algorithm 4.2.1 (HashBytes) with seed rho

7. Let t = hashfcn(m) using the hashfcn algorithm

8. Let l = HashToRange(rho || t, q, hashfcn) using Algorithm 4.1.1
   (HashToRange) on the (2 * hashlen)-octet concatenation of rho and
   t

9. Verify that U = [l]P:

   (a) If this is the case, then the decrypted plaintext m is
   returned

   (b) Otherwise, the ciphertext is rejected and no plaintext is
   returned

6.  The Boneh-Boyen BB1 Cryptosystem

   This section describes the algorithms constituting the first of the
   two Boneh-Boyen identity-based cryptosystems proposed in [BB1].  The
   description follows the practical implementation given in [BB1].

6.1.  Setup

6.1.1.  Generate a Master Secret and Public Parameters

   Algorithm 6.1.1 (BBsetup).  Randomly selects a set of master secrets
   and the associated public parameters.

Input:

o  An integer version number

o  An integer security parameter n (MUST take values either 1024,
   2048, 3072, 7680, or 15360)

Output:

o  A set of public parameters

o  A corresponding master secret

Method:

1. Depending on the version:

   (a) If version = 2, then execute Algorithm 6.1.2 (BBsetup1)

6.1.2.  Type-1 Curve Implementation

   BBsetup1 takes a security parameter n as input.  For type-1 curves, n
   corresponds to the modulus bit-size believed [BF] of comparable
   security in the classical Diffie-Hellman or RSA public-key
   cryptosystems.  For this implementation, n MUST be one of 1024, 2048,
   3072, 7680 or 15360, which correspond to the equivalent bit security
   levels of 80, 112, 128, 192 and 256 bits respectively.

   Algorithm 6.1.2 (BBsetup1): randomly establishes a master secret and
   public parameters for type-1 curves.

   Input:

   o  A security parameter n, either 1024, 2048, 3072, 7680, or 15360

   Output:

   o  A set of public parameters (version, k, E, p, q, P, P_1, P_2, P_3,
      v, hashfcn)

   o  A corresponding triple of master secrets (alpha, beta, gamma)

   Method:

   1. Determine the subordinate security parameters $n_p$ and $n_q$ as
      follows:

   (a) If n = 1024, then let n_p = 512, n_q = 160, hashfcn =
       1.3.14.3.2.26 (SHA-1 [SHA]

   (b) If n = 2048, then let n_p = 1024, n_q = 224, hashfcn =
       2.16.840.1.101.3.4.2.4 (SHA-224 [SHA])

   (c) If n = 3072, then let n_p = 1536, n_q = 256, hashfcn =
       2.16.840.1.101.3.4.2.1 (SHA-256 [SHA])

   (d) If n = 7680, then let n_p = 3840, n_q = 384, hashfcn =
       2.16.840.1.101.3.4.2.2 (SHA-384 [SHA])

   (e) If n = 15360, then let n_p = 7680, n_q = 512, hashfcn =
       2.16.840.1.101.3.4.2.3 (SHA-512 [SHA])

2. Construct the elliptic curve and its subgroup of interest as
   follows:

   (a) Select a random n_q-bit Solinas prime q

   (b) Select a random integer r, such that p = 12 * r * q - 1 is an
       n_p-bit prime

3. Select a point P of order q in E(F_p), as follows:

   (a) Select a random point P' of coordinates (x', y') on the curve
       E/F_p: y^2 = x^3 + 1 (mod p)

   (b) Let P = [12 * r]P'

   (c) If P = 0, then start over in step 3a

4. Determine the master secret and the public parameters as follows:

   (a) Select three random integers alpha, beta, gamma, each of them
       in the range 1 to q - 1

   (b) Let P_1 = [alpha]P

   (c) Let P_2 = [beta]P

   (d) Let P_3 = [gamma]P

   (e) Let v = Pairing(E, p, q, P_1, P_2), which is an element of the
       extension field F_p^2 obtained using the modified Tate pairing
       of Algorithm 4.5.1 (Pairing)

   5. (version, E, p, q, P, P_1, P_2, P_3, v, hashfcn) are the public
      parameters

   6. (alpha, beta, gamma) constitute the master secret

6.2.  Public Key Derivation

6.2.1.  Derive a Public Key from an Identity and Public Parameters

   Takes an identity string id and a set of public parameters and
   returns an integer h_id.  The public parameters used MUST be a valid
   set of public parameters as defined by Section 6.1.2.

   Algorithm 6.2.1 (BBderivePubl): derives the public key corresponding
   to an identity string.  The public parameters used MUST be a valid
   set of public parameters as defined by Section 6.1.2.

   Input:

   o  An identity string id

   o  A set of common public parameters (version, k, E, p, q, P, P_1,
      P_2, P_3, v, hashfcn)

   Output:

   o  An integer h_id modulo q

   Method:

   1. Let h_id = HashToRange(id, q, hashfcn), using Algorithm 4.1.1
      (HashToRange)

6.3.  Private Key Extraction

6.3.1.  Extract a Private Key from an Identity, Public Parameters and a
        Master Secret

   BBextractPriv takes an identity string id, a set of public
   parameters, and corresponding master secrets, and it returns a
   private key consisting of two points D_0 and D_1.  The public
   parameters used MUST be a valid set of public parameters as defined
   by Section 6.1.2.

   Algorithm 6.3.1 (BBextractPriv): extracts the private key
   corresponding to an identity string.

Input:

o   An identity string id

o   A set of public parameters (version, k, E, p, q, P, P_1, P_2, P_3,
    v, hashfcn)

Output:

o   A pair of points (D_0, D_1), each of which has order q in E(F_p)

Method:

1. Select a random integer r in the range 1 to q - 1

2. Calculate the point D_0 as follows:

    (a) Let hid = HashToRange(id, q, hashfcn) using Algorithm 4.1.1
        (HashToRange)

    (b) Let y = alpha * beta + r * (alpha * h_id + gamma) in F_q

    (c) Let D_0 = [y]P

3. Calculate the point D_1 as follows:

    (a) Let D_1 = [r]P

4. The pair of points (D_0, D_1) constitutes the private key for id

6.4.  Encryption

6.4.1.  Encrypt a Session Key Using an Identity and Public Parameters

   BBencrypt takes three inputs: a set of public parameters, an identity
   id, and a plaintext m.  The plaintext MUST be a random session key.
   The public parameters used MUST be a valid set of public parameters
   as defined by Section 6.1.2.

   Algorithm 6.4.1 (BBencrypt): encrypts a session key for an identity
   string.

   Input:

   o   A plaintext string m of size |m| octets

   o   A recipient identity string id

   o  A set of public parameters (version, k, E, p, q, P, P_1, P_2, P_3,
      v, hashfcn)

   Output:

   o  A ciphertext tuple (u, C_0, C_1, y) in F_q x E(F_p) x E(F_p) x
      {0, ... , 255}^|m|

   Method:

   1. Select a random integer s in the range 1 to q - 1

   2. Let w = v^s, which is v raised to the power of s in F_p^2, the
      result is an element of order q in F_p^2

   3. Calculate the point C_0 as follows:

      (a) Let C_0 = [s]P

   4. Calculate the point C_1 as follows:

      (a) Let _hid = HashToRange(id, q, hashfcn) using Algorithm 4.1.1
          (HashToRange)

      (b) Let y = s * h_id in F_q

      (c) Let C_1 = [y]P_1 + [s]P_3

   5. Obtain canonical string representations of certain elements:

      (a) Let psi = Canonical(p, k, 1, w) using Algorithm 4.3.1
          (Canonical), the result of which is a canonical octet string
          representation of w

      (b) Let l = Ceiling(lg(p) / 8), the number of octets needed to
          represent integers in F_p, and represent each of these F_p
          elements as a big-endian zero-padded octet string of fixed
          length l:

          (x_0)_(256^l) to represent the x coordinate of C_0

          (y_0)_(256^l) to represent the y coordinate of C_0

          (x_1)_(256^l) to represent the x coordinate of C_1

          (y_1)_(256^l) to represent the y coordinate of C_1

   6. Encrypt the message m into the string y as follows:

      (a) Compute an encryption key h_0 as a two-pass hash of w via its
          representation psi:

            i. Let zeta = hashfcn(psi) using the hashing algorithm
               hashfcn

           ii. Let xi = hashfcn(zeta || psi) using the hashing algorithm
               hashfcn

          iii. Let h' = xi || zeta, the concatenation of the previous
               two hashfcn outputs

      (b) Let y = HashBytes(|m|, h', hashfcn) XOR m, which is the
          bit-wise XOR of m with the first |m| octets of the pseudo-
          random bytes produced by Algorithm 4.2.1 (HashBytes) with seed
          h'

   7. Create the integrity check tag u as follows:

      (a) Compute a one-time pad h'' as a dual-pass hash of the
          representation of $(w, C_0, C_1, y)$:

            i. Let sigma = $(y_1)\_(256^l)$ || $(x_1)\_(256^l)$ ||
               $(y_0)\_(256^l)$ || $(x_0)\_(256^l)$ || y || psi be the
               concatenation of y and the five indicated strings in the
               specified order

           ii. Let eta = hashfcn(sigma) using the hashing algorithm
               hashfcn

          iii. Let mu = hashfcn(eta || sigma) using the hashfcn hashing
               algorithm

           iv. Let h'' = mu || eta, the concatenation of the previous
               two outputs of hashfcn

      (b) Build the tag u as the encryption of the integer s with the
          one-time pad h'':

         i. Let rho = HashToRange(h'', q, hashfcn) to get an integer in
            $Z_q$

        ii. Let u = s + rho (mod q)

   8. The complete ciphertext is given by the quadruple $(u, C_0, C_1, y)$

6.5.  Decryption

6.5.1.  Decrypt Using Public Parameters and Private Key

   BBdecrypt takes three inputs: a set of public parameters (version, k,
   E, p, q, P, P_1, P_2, P_3, v, hashfcn), a private key (D_0, D_1), and
   a ciphertext (u, C_0, C_1, y).  It outputs a message m, or signals an
   error if the ciphertext is invalid for the given key.  The public
   parameters used MUST be a valid set of public parameters as defined
   by Section 6.1.2.

   Algorithm 6.5.1 (BBdecrypt): decrypts a ciphertext using public
   parameters and a private key.

   Input:

   o  A private key given as a pair of points (D_0, D_1) of order q in
      E(F_p)

   o  A ciphertext quadruple (u, C_0, C_1, y) in Z_q x E(F_p) x E(F_p) x
      {0, ... , 255}*

   o  A set of public parameters (version, k, E, p, q, P, P_1, P_2, P_3,
      v, hashfcn)

   Output:

   o  A decrypted plaintext m, or an invalid ciphertext flag

   Method:

   1. Let w = PairingRatio(E, p, q, C_0, D_0, C_1, D_1), which computes
      the ratio of two Tate pairings (modified, for type-1 curves) as
      specified in Algorithm 4.6.1 (PairingRatio)

   2. Obtain canonical string representations of certain elements:

      (a) Let psi = Canonical(p, k, 1, w) using Algorithm 4.3.1
          (Canonical); the result is a canonical octet string
          representation of w

      (b) Let l = Ceiling(lg(p) / 8), the number of octets needed to
          represent integers in F_p, and represent each of these F_p
          elements as a big-endian zero-padded octet string of fixed
          length l:

          $(x_0)\_(256^l)$ to represent the x coordinate of C_0

(y_0)_(256^l) to represent the y coordinate of C_0

(x_1)_(256^l) to represent the x coordinate of C_1

(y_1)_(256^l) to represent the y coordinate of C_1

3. Decrypt the message m from the string y as follows:

   (a) Compute the decryption key h' as a dual-pass hash of w via its
       representation psi:

      i. Let zeta = hashfcn(psi) using the hashing algorithm hashfcn

     ii. Let xi = hashfcn(zeta || psi) using the hashing algorithm
         hashfcn

    iii. Let h' = xi || zeta, the concatenation of the previous two
         hashfcn outputs

   (b) Let m = HashBytes(|y|, h', hashfcn)_XOR y, which is the
       bit-wise XOR of y with the first |y| octets of the pseudo-
       random bytes produced by Algorithm 4.2.1 (HashBytes) with seed
       h'

4. Obtain the integrity check tag u as follows:

   (a) Recover the one-time pad h'' as a dual-pass hash of the
       representation of (w, C_0, C_1, y):

      i. Let sigma = (y_1)_(256^l) || (x_1)_(256^l) || (y_0)_(256^l)
         || (x_0)_(256^l) || y || psi be the concatenation of y and
         the five indicated strings in the specified order

     ii. Let eta = hashfcn(sigma) using the hashing algorithm hashfcn

    iii. Let mu = hashfcn(eta || sigma) using the hashing algorithm
         hashfcn

     iv. Let h'' = mu || eta, the concatenation of the previous two
         hashfcn outputs

   (b) Unblind the encryption randomization integer s from the tag u
       using h'':

      i. Let rho = HashToRange(h'', q, hashfcn) to get an integer in
         Z_q

     ii. Let s = u - rho (mod q)

   5. Verify the ciphertext consistency according to the decrypted
      values:

      (a) Test whether the equality w = v^s holds

      (b) Test whether the equality C_0 = [s]P holds

   6. Adjudication and final output:

      (a) If either of the tests performed in step 5 fails, the
          ciphertext is rejected, and no decryption is output

      (b) Otherwise, i.e., when both tests performed in step 5 succeed,
          the decrypted message is the output

7.  Test Data

   The following data can be used to verify the correct operation of
   selected algorithms that are defined in this document.

7.1.  Algorithm 3.2.2 (PointMultiply)

   Input:

   q = 0xfffffffffffffffffffffffffffffbffff

   p = 0xbfffffffffffffffffffffffffffcffff3

   E/F_p: y^2 = x^3 + 1

   A = (0x489a03c58dcf7fcfc97e99ffef0bb4634,
   0x510c6972d795ec0c2b081b81de767f808)

   l = 0xb8bbbc0089098f2769b32373ade8f0daf

   Output:

   [l]A = (0x073734b32a882cc97956b9f7e54a2d326,
   0x9c4b891aab199741a44a5b6b632b949f7)

7.2.  Algorithm 4.1.1 (HashToRange)

   Input:

   s =
   54:68:69:73:20:41:53:43:49:49:20:73:74:72:69:6e:67:20:77:69:74
   :68:6f:75:74:20:6e:75:6c:6c:2d:74:65:72:6d:69:6e:61:74:6f:72
   ("This ASCII string without null-terminator")

   n = 0xffffffffffffffffffffefffffffffffffffffff

   hashfcn = 1.3.14.3.2.16 (SHA-1)

   Output:

   v = 0x79317c1610c1fc018e9c53d89d59c108cd518608

7.3.  Algorithm 4.5.1 (Pairing)

   Input:

   q = 0xfffffffffffffffffffffffffffffbffff

   p = 0xbffffffffffffffffffffffffffffcffff3

   E/F_p: $y^2 = x^3 + 1$

   A = (0x489a03c58dcf7fcfc97e99ffef0bb4634,
   0x510c6972d795ec0c2b081b81de767f808)

   B = (0x40e98b9382e0b1fa6747dcb1655f54f75,
   0xb497a6a02e7611511d0db2ff133b32a3f)

   Output:

   e'(A, B) = (0x8b2cac13cbd422658f9e5757b85493818,
   0xbc6af59f54d0a5d83c8efd8f5214fad3c)

7.4.  Algorithm 5.2.1 (BFderivePubl)

   Input:

   id = 6f:42:62 ("Bob")

   version = 2

   p = 0xa6a0ffd016103ffffffffff595f002fe9ef195f002fe9efb

   q = 0xffffffffffffffffffffffefffffffffffff

   P = (0x6924c354256acf5a0ff7f61be4f0495b54540a5bf6395b3d,
   0x024fd8e2eb7c09104bca116f41c035219955237c0eac19ab)

   P_pub = (0xa68412ae960d1392701066664d20b2f4a76d6ee715621108,
   0x9e7644e75c9a131d075752e143e3f0435ff231b6745a486f)

   Output:

   Q_id = (0x22fa1207e0d19e1a4825009e0e88e35eb57ba79391498f59,
   0x982d29acf942127e0f01c881b5ec1b5fe23d05269f538836)

7.5.  Algorithm 5.3.1 (BFextractPriv)

   Input:

   s = 0x749e52ddb807e0220054417e514742b05a0

   version = 2

   p = 0xa6a0ffd016103ffffffffff595f002fe9ef195f002fe9efb

   q = 0xffffffffffffffffffffffefffffffffffff

   P = (0x6924c354256acf5a0ff7f61be4f0495b54540a5bf6395b3d,
   0x024fd8e2eb7c09104bca116f41c035219955237c0eac19ab)

   P_pub = (0xa68412ae960d1392701066664d20b2f4a76d6ee715621108,
   0x9e7644e75c9a131d075752e143e3f0435ff231b6745a486f)

   Output:

   Q_id = (0x8212b74ea75c841a9d1accc914ca140f4032d191b5ce5501,
   0x950643d940aba68099bdcb40082532b6130c88d317958657)

7.6.  Algorithm 5.4.1 (BFencrypt)

     Note: the following values can also be used to test
     Algorithm 5.5.1 (BFdecrypt).

   Input:

   m = 48:69:20:74:68:65:72:65:21 ("Hi there!")

   id = 6f:42:62 ("Bob")

   version = 2

   p = 0xa6a0ffd016103ffffffffff595f002fe9ef195f002fe9efb

   q = 0xfffffffffffffffffffffffeffffffffffff

   P = (0x6924c354256acf5a0ff7f61be4f0495b54540a5bf6395b3d,
   0x024fd8e2eb7c09104bca116f41c035219955237c0eac19ab)

   P_pub = (0xa68412ae960d1392701066664d20b2f4a76d6ee715621108,
   0x9e7644e75c9a131d075752e143e3f0435ff231b6745a486f)

   Output:

   Using the random value rho =
   0xed5397ff77b567ba5ecb644d7671d6b6f2082968, we get the
   following output:

   U =
   (0x1b5f6c461497acdfcbb6d6613ad515430c8b3fa23b61c585e9a541b199e
   2a6cb,
   0x9bdfbed1ae664e51e3d4533359d733ac9a600b61048a7d899104e826a0ec
   4fa4)

   V =
   e0:1d:ad:81:32:6c:b1:73:af:c2:8d:72:2e:7a:32:1a:7b:29:8a:aa

   W = f9:04:ba:40:30:e9:ce:6e:ff

7.7.  Algorithm 6.3.1 (BBextractPriv)

    Input:

    alpha = 0xa60c395285ded4d70202c8283d894bad4f0

    beta = 0x48bf012da19f170b13124e5301561f45053

    gamma = 0x226fba82bc38e2ce4e28e56472ccf94a499

    version = 2

    p = 0x91bbe2be1c8950750784beffffffffffffff6e441d41e12fb

    q = 0xfffffffffbffffffffffffffffffffffffffff

    P = (0x13cc538fe950411218d7f5c17ae58a15e58f0877b29f2fe1,
    0x8cf7bab1a748d323cc601fabd8b479f54a60be11e28e18cf)

    P_1 = (0x0f809a992ed2467a138d72bc1d8931c6ccdd781bedc74627,
    0x11c933027beaaf73aa9022db366374b1c68d6bf7d7a888c2)

    P_2 = (0x0f8ac99a55e575bf595308cfea13edb8ec673983919121b0,
    0x3febb7c6369f5d5f18ee3ea6ca0181448a4f3c4f3385019c)

    P_3 = (0x2c10b43991052e78fac44fdce639c45824f5a3a2550b2a45,
    0x6d7c12d8a0681426a5bbc369c9ef54624356e2f6036a064f)

    v = (0x38f91032de6847a89fc3c83e663ed0c21c8f30ce65c0d7d3,
    0x44b9aa10849cc8d8987ef2421770a340056745da8b99fba2)

    id = 6f:42:62 ("Bob")

    Output:

    Using the random value r =
    0x695024c25812112187162c08aa5f65c7a2c, we get the following
    output:

    D_0 = (0x3264e13feeb7c506493888132964e79ad657a952334b9e53,
    0x3eeaefc14ba1277a1cd6fdea83c7c882fe6d85d957055c7b)

    D_1 = (0x8d7a72ad06909bb3bb29b67676d935018183a905e7e8cb18,
    0x2b346c6801c1db638f270af915a21054f16044ab67f6c40e)

7.8.  Algorithm 6.4.1 (BBencrypt)

     Note: the following values can also be used to test
     Algorithm 5.5.1 (BFdecrypt).

   Input:

   m = 48:69:20:74:68:65:72:65:21 ("Hi there!")

   id = 6f:42:62 ("Bob")

   version = 2

   E: y^2 = x^3 + 1

   p = 0x91bbe2be1c8950750784beffffffffffffff6e441d41e12fb

   q = 0xfffffffffbffffffffffffffffffffffffffffff

   P = (0x13cc538fe950411218d7f5c17ae58a15e58f0877b29f2fe1,
   0x8cf7bab1a748d323cc601fabd8b479f54a60be11e28e18cf)

   P_1 = (0x0f809a992ed2467a138d72bc1d8931c6ccdd781bedc74627,
   0x11c933027beaaf73aa9022db366374b1c68d6bf7d7a888c2)

   P_2 = (0x0f8ac99a55e575bf595308cfea13edb8ec673983919121b0,
   0x3febb7c6369f5d5f18ee3ea6ca0181448a4f3c4f3385019c)

   P_3 = (0x2c10b43991052e78fac44fdce639c45824f5a3a2550b2a45,
   0x6d7c12d8a0681426a5bbc369c9ef54624356e2f6036a064f)

   v = (0x38f91032de6847a89fc3c83e663ed0c21c8f30ce65c0d7d3,
   0x44b9aa10849cc8d8987ef2421770a340056745da8b99fba2)

   hashfcn = 1.3.14.3.2.26 (SHA-1)

   Output:

   Using the random value s =
   0x62759e95ce1af248040e220263fb41b965e, we get the following
   output:

   u = 0xad1ebfa82edf0bcb5111e9dc08ff0737c68

   C_0 = (0x79f8f35904579f1aaf51897b1e8f1d84e1c927b8994e81f9,
   0x1cf77bb2516606681aba2e2dc14764aa1b55a45836014c62)

    C_1 = (0x410cfeb0bccf1fa4afc607316c8b12fe464097b20250d684,
    0x8bb76e7195a7b1980531b0a5852ce710cab5d288b2404e90)

    y = 82:a6:42:b9:bb:e9:82:c4:57

8.  ASN.1 Module

    This section defines the ASN.1 module for the encodings discussed in
    this document.

    IBCS { joint-iso-itu-t(2) country(16) us(840) organization(1)
        identicrypt(114334) ibcs(1) module(5) version(1) }

    DEFINITIONS IMPLICIT TAGS ::= BEGIN

    --
    -- Identity-based cryptography standards (IBCS):
    -- supersingular curve implementations of
    -- the BF and BB1 cryptosystems
    --
    -- This version only supports IBE using
    -- type-1 curves, i.e., the curve $y^2 = x^3 + 1$.
    --

    ibcs OBJECT IDENTIFIER ::= {
        joint-iso-itu-t(2) country(16) us(840) organization(1)
            identicrypt(114334) ibcs(1)
    }

    --
    -- IBCS1
    --
    -- IBCS1 defines the algorithms used to implement IBE
    --

    ibcs1 OBJECT IDENTIFIER ::= {
        ibcs ibcs1(1)
    }

    --
    -- An elliptic curve is specified by an OID.
    -- A type1curve is defined by the equation $y^2 = x^3 + 1$.
    --

    type1curve OBJECT IDENTIFIER ::= {
        ibcs1 curve-types(1) type1-curve(1)
    }

```
   --
   -- Supporting types
   --


   --
   -- Encoding of a point on an elliptic curve E/F_p
   -- An FpPoint can either represent an element of
   -- F_p^2 or an element of (F_p)^2.

    FpPoint ::= SEQUENCE {
       x  INTEGER,
       y  INTEGER
   }


   --
   -- The following hash functions are supported:
   --
   -- SHA-1
   --
   -- id-sha1  OBJECT IDENTIFIER  ::= {
   --   iso(1) identified-organization(3) oiw(14)
   --   secsig(3) algorithms(2) hashAlgorithmIdentifier(26)
   -- }
   --
   -- SHA-224
   --
   -- id-sha224  OBJECT IDENTIFIER  ::= {
   --   joint-iso-itu-t(2)country(16) us(840)
   --   organization(1) gov(101)
   --   csor(3) nistAlgorithm(4) hashAlgs(2) sha224(4)
   -- }
   --
   -- SHA-256
   --
   -- id-sha256  OBJECT IDENTIFIER  ::= {
   --   joint-iso-itu-t(2)country(16) us(840)
   --   organization(1) gov(101)
   --   csor(3) nistAlgorithm(4) hashAlgs(2) sha256(1)
   -- }
   --
   -- SHA-384
   --
   -- id-sha384  OBJECT IDENTIFIER  ::= {
   --   joint-iso-itu-t(2)country(16) us(840)
   --   organization(1) gov(101)
   --   csor(3) nistAlgorithm(4) hashAlgs(2) sha384(2)
   -- }
   --
```

```
   -- SHA-512
   --
   -- id-sha512  OBJECT IDENTIFIER  ::= {
   --    joint-iso-itu-t(2) country(16) us(840)
   --    organization(1) gov(101)
   --    csor(3) nistAlgorithm(4) hashAlgs(2) sha512(3)
   -- }
   --
   --
   -- Algorithms
   --

   ibe-algorithms OBJECT IDENTIFIER ::= {
      ibcs1 ibe-algorithms(2)
   }

   ---
   --- Boneh-Franklin IBE
   ---

   bf OBJECT IDENTIFIER ::= { ibe-algorithms bf(1) }

   --
   -- Encoding of a BF public parameters block.
   -- The only version currently supported is version 2.
   -- The values p and q define a subgroup of E(F_p) of order q.
   --

   BFPublicParameters ::= SEQUENCE {
      version      INTEGER { v2(2) },
      curve        OBJECT IDENTIFIER,
      p            INTEGER,
      q            INTEGER,
      pointP       FpPoint,
      pointPpub    FpPoint,
      hashfcn      OBJECT IDENTIFIER
   }

   --
   -- A BF private key is a point on an elliptic curve,
   -- which is an FpPoint.
   -- The only version supported is version 2.
   --

   BFPrivateKeyBlock ::= SEQUENCE {
      version      INTEGER { v2(2) },
      privateKey  FpPoint
   }
```

```
   --
   -- A BF master secret is an integer.
   -- The only version supported is version 2.
   --

   BFMasterSecret ::= SEQUENCE {
      version        INTEGER {v2(2) },
      masterSecret   INTEGER
   }

   --
   -- BF ciphertext block
   -- The only version supported is version 2.
   --

   BFCiphertextBlock ::= SEQUENCE {
      version  INTEGER { v2(2) },
      u        FpPoint,
      v        OCTET STRING,
      w        OCTET STRING
   }

   --
   -- Boneh-Boyen (BB1) IBE
   --

   bb1 OBJECT IDENTIFIER ::= { ibe-algorithms bb1(2) }

   --
   -- Encoding of a BB1 public parameters block.
   -- The version is currently fixed to 2.
   --
   --

   BB1PublicParameters ::= SEQUENCE {
      version      INTEGER { v2(2) },
      curve        OBJECT IDENTIFIER,
      p            INTEGER,
      q            INTEGER,
      pointP       FpPoint,
      pointP1      FpPoint,
      pointP2      FpPoint,
      pointP3      FpPoint,
      v            FpPoint,
      hashfcn      OBJECT IDENTIFIER
   }
```

```
   --
   -- BB1 master secret block
   -- The only version supported is version 2.
   --

   BB1MasterSecret ::= SEQUENCE {
      version  INTEGER { v2(2) },
      alpha    INTEGER,
      beta     INTEGER,
      gamma    INTEGER
   }

   --
   -- BB1 private Key block
   -- The only version supported is version 2.
   --

   BB1PrivateKeyBlock ::= SEQUENCE {
      version  INTEGER { v2(2) },
      pointD0  FpPoint,
      pointD1  FpPoint
   }

   --
   -- BB1 ciphertext block
   -- The only version supported is version 2.
   --

   BB1CiphertextBlock ::= SEQUENCE {
      version     INTEGER {v2(2) },
      pointChi0   FpPoint,
      pointChi1   FpPoint,
      nu          INTEGER,
      y           OCTET STRING
   }

   END
```

9.  Security Considerations

   This document describes cryptographic algorithms.  We assume that the
   security provided by such algorithms depends entirely on the secrecy
   of the relevant private key, and for an adversary to defeat the
   security provided by the algorithms, he will need to perform
   computationally-intensive cryptanalytic attacks to recover the
   private key.

   We assume that users of the algorithms described in this document
   will require one of five levels of cryptographic strength: the
   equivalent of 80 bits, 112 bits, 128 bits, 192 bits or, 256 bits.
   The 80-bit level is suitable for legacy applications and SHOULD NOT
   be used to protect information whose useful life extends past the
   year 2010.  The 112-bit level is suitable for use in key transport of
   Triple-DES keys and should be adequate to protect information whose
   useful life extends up to the year 2030.  The 128-bit levels and
   higher are suitable for use in the transport of Advanced Encryption
   Standard (AES) keys of the corresponding length or less and are
   adequate to protect information whose useful life extends past the
   year 2030.

   Table 1 summarizes the security parameters for the BF and BB1
   algorithms that will attain these levels of security.  In this table,
   $|p|$ represents the number of bits in a prime number p, and $|q|$
   represents the number of bits in a subprime q.  This table assumes
   that a Type-1 supersingular curve is used.

   | Bits of Security | $|p|$ | $|q|$ |
   |---|---|---|
   | 80 | 512 | 160 |
   | 112 | 1024 | 224 |
   | 128 | 1536 | 256 |
   | 192 | 3840 | 384 |
   | 256 | 7680 | 512 |

   Table 1: Sizes of BF and BB1 Parameters Required to Attain Standard
   Levels of Bit Security [SP800-57].

   If an IBE key is used to transport a symmetric key that provides more
   bits of security than the bit strength of the IBE key, users should
   understand that the security of the system is then limited by the
   strength of the weaker IBE key. So if an IBE key that provides 112
   bits of security is used to transport a 128-bit AES key, then the
   security provided is limited by the 112 bits of security of the IBE
   key.

Note that this document specifies the use of the National Institute
of Standards and Technology (NIST) hashing algorithms [SHA] to hash
identities to either a point on an elliptic curve or an integer.
Recent attacks on SHA-1 [SHA] have discovered ways to find collisions
with less work than the expected 2^80 hashes required based on the
size of the output of the hash function alone.  If an attacker can
find a collision, then they could use the colliding preimages to
create two identities that have the same IBE private key.  The
practical use of such a SHA-1 [SHA] collision is extremely unlikely,
however.

Identities are typically not random strings like the preimages of a
hash collision would be.  In particular, this is true if IBE is used
as described in [IBECMS], in which components of an identity are
defined to be an e-mail address, a validity period, and a URI.  In
this case, the unpredictable results of a collision are extremely
unlikely to fit the format of a valid identity, and thus, are of no
use to an attacker.  Any protocol using IBE MUST define an identity
in a way that makes collisions in a hash function essentially useless
to an attacker.  Because random strings are rarely used as
identities, this requirement should not be unduly difficult to
fulfill.

The randomness of the random values that are required by the
cryptographic algorithms is vital to the security provided by the
algorithms.  Any implementation of these algorithms MUST use a source
of random values that provides an adequate level of security.
Appropriate algorithms to generate such values include [FIPS186-2]
and [X9.62].  This will ensure that the random values used to mask
plaintext messages in Sections 5.4 and 6.4 are not reused with a
significant probability.

The strength of a system using the algorithms described in this
document relies on the strength of the mechanism used to authenticate
a user requesting a private key from a PKG, as described in step 2 of
Section 1.2 of this document.  This is analogous to the way in which
the strength of a system using digital certificates [X.509] is
limited by the strength of the authentication required of users
before certificates are granted to them.  In either case, a weak
mechanism for authenticating users will result in a weak system that
relies on the technology.  A system that uses the algorithms
described in this document MUST require users to authenticate in a
way that is suitably strong, particularly if IBE private keys will be
used for authentication.

Note that IBE systems have different properties than other asymmetric
cryptographic schemes when it comes to key recovery.  If a master
secret is maintained on a secure PKG, then the PKG and any

   administrator with the appropriate level of access will be able to
   create arbitrary private keys, so that controls around such
   administrators and logging of all actions performed by such
   administrators SHOULD be part of a functioning IBE system.

   On the other hand, it is also possible to create IBE private keys
   using a master secret and to then destroy the master secret, making
   any key recovery impossible.  If this property is not desired, an
   administrator of an IBE system SHOULD require that the format of the
   identity used by the system contain a component that is short-lived.
   The format of identity that is defined in [IBECMS], for example,
   contains information about the time period of validity of the key
   that will be calculated from the identity.  Such an identity can
   easily be changed to allow the rekeying of users if their IBE private
   key is somehow compromised.

10.  Acknowledgments

   This document is based on the IBCS #1 v2 document of Voltage
   Security, Inc.  Any substantial use of material from this document
   should acknowledge Voltage Security, Inc.  as the source of the
   information.

11.  References

11.1.  Normative References

   [KEYWORDS]    Bradner, S., "Key words for use in RFCs to Indicate
                 Requirement Levels", BCP 14, RFC 2119, March 1997.

   [TLS]         Dierks, T. and E. Rescorla, "The Transport Layer
                 Security (TLS) Protocol Version 1.1", RFC 4346, April
                 2006.

11.2.  Informative References

   [BB1]         D. Boneh and X. Boyen, "Efficient selective-ID secure
                 identity based encryption without random oracles," In
                 Proc. of EUROCRYPT 04, LNCS 3027, pp. 223-238, 2004.

   [BF]          D. Boneh and M. Franklin, "Identity-based encryption
                 from the Weil pairing," in Proc. of CRYPTO 01, LNCS
                 2139, pp. 213-229, 2001.

   [CMS]         Housley, R., "Cryptographic Message Syntax (CMS)", RFC
                 3852, July 2004.

   [ECC]          I. Blake, G. Seroussi, and N. Smart, "Elliptic Curves in
                  Cryptography", Cambridge University Press, 1999.

   [FIPS186-2]    National Institute of Standards and Technology, "Digital
                  Signature Standard," Federal Information Processing
                  Standard 186-2, August 2002.

   [IBEARCH]      G. Appenzeller, L. Martin, and M. Schertler, "Identity-
                  based Encryption Architecture", Work in Progress.

   [IBECMS]       L. Martin and M. Schertler, "Using the Boneh-Franklin
                  and Boneh-Boyen identity-based encryption algorithms
                  with the Cryptographic Message Syntax (CMS)", Work in
                  Progress.

   [MERKLE]       R. Merkle, "A fast software one-way hash function,"
                  Journal of Cryptology, Vol. 3 (1990), pp. 43-58.

   [P1363]        IEEE P1363-2000, "Standard Specifications for Public Key
                  Cryptography," 2001.

   [SP800-57]     E. Barker, W. Barker, W. Burr, W. Polk and M. Smid,
                  "Recommendation for Key Management - Part 1: General
                  (Revised)," NIST Special Publication 800-57, March 2007.

   [SHA]          National Institute for Standards and Technology, "Secure
                  Hash Standard," Federal Information Processing Standards
                  Publication 180-2, August 2002, with Change Notice 1,
                  February 2004.

   [X9.62]        American National Standards Institute, "Public Key
                  Cryptography for the Financial Services Industry: The
                  Elliptic Curve Digital Signature Algorithm (ECDSA),"
                  American National Standard for Financial Services
                  X9.62-2005, November 2005.

   [X.509]        ITU-T Recommendation X.509 (2000) | ISO/IEC 9594-8:2001,
                  Information Technology - Open Systems Interconnection -
                  The Directory:  Public-key and Attribute Certificate
                  Frameworks.

Authors' Addresses

   Xavier Boyen
   Voltage Security
   1070 Arastradero Rd Suite 100
   Palo Alto, CA 94304

   EMail: xavier@voltage.com


   Luther Martin
   Voltage Security
   1070 Arastradero Rd Suite 100
   Palo Alto, CA 94304

   EMail: martin@voltage.com

Full Copyright Statement

Intellectual Property