

S/MIME Working Group
Internet Draft
Expires: December 2006

X. Boyen
L. Martin
Voltage Security
June 2006

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

<[draft-ietf-smime-ibcs-00.txt](#)>

Status of this Memo

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with [Section 6 of BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at <http://www.ietf.org/ietf/1id-abstracts.txt>

The list of Internet-Draft Shadow Directories can be accessed at <http://www.ietf.org/shadow.html>

Abstract

This document describes the algorithms that implement Boneh-Franklin and Boneh-Boyer Identity-based Encryption. This document is in part based on IBCS #1 v2 of Voltage Security's Identity-based Cryptography Standards (IBCS) documents.

Table of Contents

1.	Introduction.....	3
2.	Notation and definitions.....	4
	2.1. Notation.....	4
	2.2. Definitions.....	6
3.	Basic elliptic curve algorithms.....	7
	3.1. The group action in affine coordinates.....	7
	3.1.1. Implementation for type-1 curves.....	7
	3.2. Point multiplication.....	9
	3.3. Special operations in projective coordinates.....	11
	3.3.1. Implementation for type-1 curves.....	11
	3.4. Divisors on elliptic curves.....	13
	3.4.1. Implementation in F_p^2 for type-1 curves.....	13
	3.5. The Tate pairing.....	15
	3.5.1. The Miller algorithm for type-1 curves.....	16
4.	Supporting algorithms.....	18
	4.1. Integer range hashing.....	19
	4.2. Pseudo-random generation by hashing.....	20
	4.3. Canonical encodings of extension field elements.....	20
	4.3.1. Type-1 curve implementation.....	21
	4.4. Hashing onto a subgroup of an elliptic curve.....	22
	4.4.1. Type-1 curve implementation.....	22
	4.5. Bilinear pairing.....	23
	4.5.1. Type-1 curve implementation.....	24
	4.6. Ratio of bilinear pairings.....	25
	4.6.1. Type-1 curve implementation.....	25
5.	The Boneh-Franklin BF cryptosystem.....	26
	5.1. Setup.....	26
	5.1.1. Type-1 curve implementation.....	27
	5.2. Public key derivation.....	28
	5.3. Private key extraction.....	28
	5.4. Encryption.....	29
	5.5. Decryption.....	30
6.	Wrapper methods for the BF system.....	32
	6.1. Private key generator (PKG) setup.....	32
	6.2. Private key extraction by the PKG.....	32
	6.3. Session key encryption.....	33
7.	Concrete encoding guidelines for BF.....	35
	7.1. Encoding of points on a curve.....	35
	7.2. Public parameters blocks.....	36
	7.2.1. Type-1 implementation.....	36
	7.3. Master secret blocks.....	38
	7.4. Private key blocks.....	38

7.5. Ciphertext blocks.....	39
8. The Boneh-Boyen BB1 cryptosystem.....	40
8.1. Setup.....	40
8.1.1. Type-1 curve implementation.....	41
8.2. Public key derivation.....	42
8.3. Private key extraction.....	43
8.4. Encryption.....	44
8.5. Decryption.....	46
9. Wrapper methods for the BB1 system.....	48
9.1. Private key generator (PKG) setup.....	48
9.2. Private key extraction by the PKG.....	49
9.3. Session key encryption.....	50
10. Concrete encoding guidelines for BB1.....	51
10.1. Encoding of points on a curve.....	51
10.2. Public parameters blocks.....	52
10.2.1. Type-1 implementation.....	53
10.3. Master secret blocks.....	54
10.4. Private key blocks.....	55
10.5. Ciphertext blocks.....	56
11. ASN.1 syntax.....	57
12. Security considerations.....	63
13. IANA considerations.....	63
14. Acknowledgments.....	63
15. References.....	64
15.1. Informative references.....	64
Authors Addresses.....	64
Intellectual Property Statement.....	64
Disclaimer of Validity.....	65
Copyright Statement.....	65
Acknowledgment.....	65

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) [3], and the first IBE system proposed by Boneh and Boyen (BB1) [2]. 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 2.3). Implementations based on other types of curves currently fall outside the scope of this document.

2. Notation and definitions

2.1. Notation

This section summarizes the essential notions and definitions regarding identity-based cryptosystems on elliptic curves. The reader is referred to [1] for the mathematical background and to [2, 3] regarding all notions pertaining to identity-based encryption.

Let F_p be a finite field of large prime characteristic p , and let F_{p^k} denote its extension field of degree k . Denote by $F_{p^k}^*$ the multiplicative group of F_{p^k} , for any $k \geq 1$.

Let $E/F_p : y^2 = x^3 + a * x + b$ be an elliptic curve over F_p . For any extension degree $k \geq 1$, the curve E/F_p defines a group $(E(F_{p^k}), +)$, which is the additive group of points of affine coordinates (x, y) in $(F_{p^k})^2$ satisfying the curve equation over F_{p^k} , with null element, or point at infinity, denoted θ . Let $\#E(F_{p^k})$ be the size of $E(F_{p^k})$.

Let q be a prime such that $E(F_p)$ has a cyclic subgroup G_1 of order q . Let k be the embedding degree or security multiplier of G_1 in $E(F_p)$, or the smallest integer greater than or equal to 1 such that q divides $p^k - 1$. Then $E(F_{p^k})$ contains a cyclic subgroup of order q , denoted G_1 , and $F_{p^k}^*$ contains a cyclic subgroup of order p , denoted G_2 .

Under these conditions, two mathematical constructions known as the Weil pairing and the Tate pairing, each provide an efficiently computable map $e : G_1 \times G_1 \rightarrow G_2$ that is linear in both arguments and believed hard to invert. If an efficiently computable isomorphism $\phi : G_1 \rightarrow G_1$ is available for the selected elliptic curve on which the Tate pairing is computed, then one can construct a function $e : G_1 \times G_1 \rightarrow G_2$, 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 1536-bit prime, being the order of the finite field F_p .

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

$\#G$ - the size of G , where G is a finite group.

G^* - the multiplicative group of the invertible elements in G ; 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 .

θ - the conventional 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 θ .

q - a 160 bit to 256-bit prime, being the order of the cyclic subgroup of interest in $E(F_p)$.

k - the embedding degree, or security multiplier, of the cyclic subgroup of order q in $E(F_p)$.

F_p^k - the extension field of the base field F_p of degree equal to the security multiplier k .

$E(F_p^k)$ - the group of points of affine coordinates in F_p^k satisfying the curve equation E/F_p , including the point at infinity θ .

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 $A + B$.

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 the IBCS#1 standard, referred to as type-1.

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 $\#E(F_p) = p + 1$, so that the p pairs of (x, y) coordinates corresponding to the p non-zero points $E(F_p) \setminus \{\theta\}$ satisfy a useful bijective relation $x \leftrightarrow y$, with $x = (y^2 - 1)^{1/3} \pmod{p}$ and $y = (x^3 + 1)^{1/2} \pmod{p}$. Type-1 curves always lead to a security multiplier $k = 2$, where

$f(x) = (x^2 + 1)$ is always irreducible, allowing the uniform representation of $F_p^2 = F[x] / (x^2 + 1)$. 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 8.1.2 (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^k elements of the extension field F_p^k are represented as k -tuples of elements of F_p . A k -tuple $(a_0, \dots, a_{(k-1)})$ is interpreted as the polynomial $a_{(k-1)} * x^{(k-1)} + \dots + a_1 * x + a_0$ in $F_p[x] / f(x)$, where $f(x)$ is an irreducible monic polynomial of order k . The actual polynomial $f(x)$ chosen depends on p and k .

Type-1 curves For type-1 curves, which are supersingular curves of equation $E/F_p : y^2 = x^3 + 1$ with p congruent to 11 modulo 12, the extension degree k is always 2 and elements of F_p^2 are represented as polynomials $a_1 * x + a_0$ in $F_p[x] / (x^2 + 1)$.

Elliptic curve points Points in $E(F_p^k)$ for $k \geq 1$ with the point $P = (x, y)$ in $F_p^k \times F_p^k$ 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^k .

2.2. Definitions

The following terminology is used to describe an IBE system.

Public parameters The public parameters are set of common systemwide parameters generated and published by the private key server (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 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 A 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 A ciphertext is an encrypted representation of any block of data, including a 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 [1].

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:

a point A in $E(\mathbb{F}_p^k)$, with $A = (x, y)$ or θ .

an elliptic curve $E/\mathbb{F}_p : y^2 = x^3 + 1$.

Output:

the point $[2]A = A + A$.

Method:

1. If $A = \emptyset$ or $y = 0$, then return \emptyset .
2. $\lambda = (3 * x^2) / (2 * y)$.
3. $x = \lambda^2 - 2 * x$.
4. $y = (x - x) * \lambda - y$.
5. Return (x, y) .

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

Input:

- a point A in $E(F_p^k)$, with $A = (x_A, y_A)$ or \emptyset ,
- a point B in $E(F_p^k)$, with $B = (x_B, y_B)$ or \emptyset ,
- an elliptic curve $E/F_p : y^2 = x^3 + 1$.

Output:

- the point $A + B$.

Method:

1. If $A = \emptyset$, return B .
2. If $B = \emptyset$, return A .
3. If $x_A = x_B$:
 - (a) If $y_A = -y_B$, return \emptyset .
 - (b) Else return [\[2\]](#)A computed using Algorithm 2.1.1 (PointDouble1).
4. Otherwise:
 - (a) $\lambda = (y_B - y_A) / (x_B - x_A)$.
 - (b) $x = \lambda^2 - x_A - x_B$.
 - (c) $y = (x_A - x) * \lambda - y_A$.
 - (d) Return (x, y) .

3.2. Point multiplication

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

Input:

an integer $l > 0$,

an integer window bit-size $r > 0$.

Output:

The unique d -element sequence $\{(b_i, e_i)\}$ for $i = 0$ to $d - 1$ such that $l = \{\text{Sum}(b_i * 2^{e_i}) \text{ for } i = 0 \text{ to } d - 1\}$ and $b_i = +/- 2^j$ for some $0 \leq j \leq r - 1$.

Method:

1. $d = 0$.
2. $j = 0$.
3. While $j \leq l$, do:
 - (a) If $l_k = 0$ then:
 - i. $j = j + 1$.
 - (b) Else:
 - i. $t = \min\{j + r - 1, l\}$.
 - ii. $h_d = (l_t, l_{(t-1)}, \dots, l_j)$ (base 2).
 - iii. If $h_d > 2^{(r-1)}$ then:
 - A. $b_d = h_d \cdot 2^r$.
 - B. $l = l + 2^{(t+1)}$.
 - iv. Else:
 - A. $b_d = h_d$.
 - v. $e_d = j$.

vi. $d = d + 1$.

vii. $j = t + 1$.

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

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

Input:

a point A in $E(F_{p^k})$,

an integer $l > 0$,

an elliptic curve $E/F_p : y^2 = x^3 + a * x + b$.

Output:

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 $l = \sum_{j=0}^l l_j * 2^j$ where $l = \{ \sum_{j=0}^l l_j * 2^j \}$ is the binary expansion of l .

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

2. (Precomputation)

(a) $A_1 = A$.

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

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

i. $A_{(2 * i + 1)} = A_{(2 * i)} + A_2$ using Algorithm 3.1.2 (PointAdd1).

(d) $Q = A_{(b_{(d-1)})}$.

3. Main loop

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

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

ii. If $b_i > 0$ then:

A. $Q = Q + A_{(b_i)}$ using Algorithm 3.1.2 (PointAdd1).

iii. Else:

A. $Q = Q - A_{(-b_i)}$ 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. Special operations in projective coordinates

3.3.1. Implementation for type-1 curves

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

Input:

- a point $(x, y, z) = A$ in $E(F_p^k)$ in projective coordinates,
- an elliptic curve $E/F_p : y^2 = x^3 + 1$.

Output:

the point $[2]A$ in projective coordinates.

Method:

1. If $z = 0$ or $y = 0$, return $(0, 1, 0) = \mathcal{O}$. Otherwise:
2. $\lambda_1 = 3 * x^2$.
3. $z = 2 * y * z$.

4. $\lambda_2 = y^2$.
5. $\lambda_3 = 4 * \lambda_2 * x$.
6. $x = \lambda_1^2 - 2 * \lambda_3$.
7. $\lambda_4 = 8 * \lambda_2^2$.
8. $y = \lambda_1 * (\lambda_3 - x) - \lambda_4$.
9. Return (x, y, z) .

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

Input:

- a point $(x_A, y_A, z_A) = A$ in $E(F_p^k)$ in projective coordinates,
- a point $(x_B, y_B) = B$ in $E(F_p^k) \setminus \{0\}$ in affine coordinates,
- an elliptic curve $E/F_p : y^2 = x^3 + 1$.

Output:

- the point $A + B$ in projective coordinates.

Method:

1. If $z_A = 0$ return $(x_B, y_B, 1) = B$. Otherwise:
2. $\lambda_1 = z_A^2$
3. $\lambda_2 = \lambda_1 * x_B$.
4. $\lambda_3 = x_A - \lambda_2$.
5. If $\lambda_3 = 0$ then return $(0, 1, 0) = 0$. Otherwise:
6. $\lambda_4 = \lambda_3^2$.
7. $\lambda_5 = \lambda_1 * y_B * z_A$.
8. $\lambda_6 = \lambda_4 - \lambda_5$.
9. $\lambda_7 = x_A + \lambda_2$.

10. $\lambda_8 = y_A + \lambda_5$.
11. $x = \lambda_6^2 - \lambda_7 * \lambda_4$.
12. $\lambda_9 = \lambda_7 * \lambda_4 - 2 * x$.
13. $y = (\lambda_9 * \lambda_6 - \lambda_8 * \lambda_3 * \lambda_4) / 2$.
14. $z = \lambda_3 * z_A$.
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:

- a point B in $E(F_p^2)$ with $B \neq \theta$.
- a point A in $E(F_p)$.
- a description of a type-1 elliptic curve E/F_p .

Output:

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

Method:

1. $r = x_B - x_A$.
2. Return r .

Algorithm 2.4.2 (EvalTangent1): evaluates the divisor of a tangent on a type-1 elliptic curve.

Input:

- a point B in $E(F_p^2)$ with $B \neq \theta$.
- a point A in $E(F_p)$.

a description of a type-1 elliptic curve E/F_p .

Output:

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 $A = \theta$ return $1 = 1 + \theta * i$.

(b) If $y_A = \theta$ return $\text{EvalVertical1}(B, A)$ using Algorithm 3.4.1 (EvalVertical1).

2. (Line computation)

(a) $a = .3 * (x_A)^2$.

(b) $b = 2 * y_A$.

(c) $c = .b * y_A . a * x_A$.

3. (Evaluation at B)

(a) $r = a * x_B + b * y_B + c$.

4. Return r .

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

Input:

a point B in $E(F_p^2)$ with $B \neq \theta$.

two points A, A in $E(F_p)$.

a description of a type-1 elliptic curve E/F_p .

Output:

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 $A = \theta$ return $\text{EvalVertical1}(B, A)$ using Algorithm 3.4.1 (EvalVertical1).

(b) If $A = \theta$ return $\text{EvalVertical1}(B, A)$ using Algorithm 3.4.1 (EvalVertical1).

(c) If $A = .A$ return $\text{EvalVertical1}(B, A)$ using Algorithm 3.4.1 (EvalVertical1).

(d) If $A = A$ return $\text{EvalTangent1}(B, A)$ using Algorithm 3.4.2 (EvalTangent1).

2. (Line computation)

(a) $a = y_A - y_A$.

(b) $b = x_A - x_A$.

(c) $c = .b * y_A - a * x_A$.

3. (Evaluation at B)

(a) $r = a * x_B + b * y_B + c$.

4. Return r.

3.5. The Tate pairing

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

Input:

a point A of order q in $E(F_p)$,

a point B of order q in $E(F_{p^k})$,

a description of an elliptic curve E/F_p such that $E(F_p)$ and $E(F_{p^k})$ have a subgroup of order q.

Output:

the value $e(A, B)$ in F_{p^k} , computed using the Miller algorithm.

Method:

1. For type-1 curve E , proceed with Algorithm 3.5.2 (TateMillerSolinas).

3.5.1. The Miller algorithm for type-1 curves

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

Input:

a point A of order q in $E(F_p)$,

a point B of order q in $E(F_{p^2})$,

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 prime order q , where $q = 2^a + s * 2^b + c$ with c and s equal to either 1 or -1.

Output:

the value $e(A, B)$ in F_{p^2} , computed using the Miller algorithm.

The following description assumes that $F_{p^2} = F_p[i]$, where $i^2 = -1$.

Elements x in F_{p^2} may be explicitly represented as $a + i * b$, with a, b in F_p .

Points in $E(F_p)$ may also be represented as coordinate pairs (x, y) with x, y in F_p .

Points in $E(F_{p^2})$ may be represented either as (x, y) , with x, y in F_{p^2} or as $(a + i * b, c + i * d)$, with a, b, c, d in F_p .

Method:

1. (Initialization)

(a) $v_{\text{num}} = 1$ in F_{p^2} .

(b) $v_{\text{den}} = 1$ in F_{p^2} .

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

(d) $t_{\text{num}} = 1$ in F_{p^2} .

(e) $t_den = 1$ in F_p^2 .

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

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

i. $t_num = t_num^2$.

ii. $t_den = t_den^2$.

iii. $t_num = t_num * \text{EvalTangent1}(B, V)$ using Algorithm 3.4.2 (EvalTangent1).

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

v. $t_den = t_den * \text{EvalVertical1}(B, V)$ using Algorithm 3.4.1 (EvalVertical1).

(b) (Normalization)

i. $V_b = (x_{(V_b)}, y_{(V_b)}) = (x_V / z_V^2, s * y_V / z_V^3)$ in $(F_p)^2$, resulting in a point V_b in $E(F_p)$.

(c) (Accumulation) Selecting on s :

i. If $s = 0$:

A. $v_num = v_num * t_den$.

B. $v_den = v_den * t_num * \text{EvalVertical1}(B, V)$ using Algorithm 3.4.1 (EvalVertical1).

ii. If $s = 1$:

A. $v_num = v_num * t_num$.

B. $v_den = v_den * t_den$.

3. (Calculation of the 2^a contribution)

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

i. $t_num = t_num^2$.

ii. $t_den = t_den^2$.

iii. $t_num = t_num * \text{EvalTangent1}(B, V)$ using Algorithm 3.4.2 (EvalTangent1).

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

v. $t_den = t_den * \text{EvalVertical1}(B, V)$ using Algorithm 3.4.1 (EvalVertical1).

(b) (Normalization)

i. $V_a = (x_{(V_a)}, y_{(V_a)}) = (x_V / z_V^2, s * x_V / z_V^3)$ in $(F_p)^2$, resulting in a point V_a in $E(F_p)$.

(c) (Accumulation)

i. $v_num = v_num * t_num$.

ii. $v_den = v_den * t_den$.

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

(a) $v_num = v_num * \text{EvalLine1}(B, V_a, V_b)$ using Algorithm 3.4.3 (EvalLine1).

(b) $v_den = v_den * \text{EvalVertical1}(B, V_a + V_b)$ using Algorithm 3.4.1 (EvalVertical1).

(c) If $c = .1$ then:

i. $v_den = v_den * \text{EvalVertical1}(B, A)$ using Algorithm 3.4.1 (EvalVertical1).

5. (Correcting exponent)

(a) Let $\eta = (p^2 - 1) / q$ (an integer).

6. (Final result)

(a) Return $(v_num / v_den)^\eta$ in F_{p^2} .

4. Supporting algorithms

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

4.1. Integer range hashing

HashToRangen(s , n) takes a string s and an integer n as input, and returns an integer in the range 0 to $n - 1$ by cryptographic hashing. The function performs a number l of SHA1 applications, with l chosen in function of n so that, for random input, the output has an almost uniform distribution in the entire range 0 to $n - 1$ with a statistical relative non-uniformity no greater than $1/\sqrt{n}$. I.e., for arbitrarily large n , for all v in 0 to $n - 1$, the probability that $\text{HashToRangen}(s, n) = v$ lies in the interval $[(1 - n^{-.1/2}) / n, (1 + n^{-.1/2}) / n]$.

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

Input:

a string s of length $|s|$ bytes,

a positive integer n represented as $\text{Ceiling}(8 * \lg(n))$ bytes.

Output:

a positive integer v in the range 0 to $n - 1$.

Method:

1. $v_0 = 0$.

2. $h_0 = 0x00$, a string of 20 null bytes.

3. $l = \text{Ceiling}((3 / 5) * \lg(n))$.

4. for each i in 1 to l , do:

(a) $t_i = h_{(i - 1)} || s$, which is the $(|s| + 20)$ -byte string concatenation of the strings $h_{(i - 1)}$ and s .

(b) $h_i = \text{SHA1}(t_i)$, which is a 20-byte string resulting from the SHA1 algorithm on input t_i .

(c) Let $a_i = \text{Value}(h_i)$ be the integer in the range 0 to $256^{20} - 1$ denoted by the raw byte string h_i interpreted in the unsigned big endian convention.

(d) $v_i = 256^{20} * v_{(i - 1)} + a_i$.

5. $v = v_1 \pmod n$.

4.2. Pseudo-random generation by hashing

HashStream(b, p) takes an integer b and a string p as input, and returns a b -byte pseudo-random string r as output. This function relies on the SHA1 cryptographic hashing algorithm, and has a 160-bit internal effective key space equal to the range of SHA1.

Algorithm 4.2.1 (HashStream): keyed cryptographic pseudo-random stream generator.

Input:

- an integer b ,
- a string p .

Output:

- a string r of size b bytes.

Method:

1. $K = \text{SHA1}(p)$.
2. $h_0 = 0x00$, a string of 20 null bytes.
3. $l = \text{Ceiling}(b / 20)$.
4. for each i in 1 to l do:
 - (a) $h_i = \text{SHA1}(h_{(i - 1)})$.
 - (b) $r_i = \text{SHA1}(h_i || K)$, where $h_i || K$ is the 40-byte concatenation of h_i and K .
5. $r = \text{LeftmostBytes}(b, r_1 || \dots || r_l)$, i.e., r is formed as the concatenation of the r_i , truncated to the desired number of bytes.

4.3. Canonical encodings of extension field elements

Canonical(p, k, o, v) takes an element v in F_p^k , and returns a canonical byte-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^k as strings.

Input:

- an element v in F_p^k ,
- a description of F_p^k ,
- a ordering parameter o , either 0 or 1.

Output:

- a fixed-length string s representing v .

Method:

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

4.3.1. Type-1 curve implementation

Canonical1(p, o, v) takes an element v in F_p^2 and returns a canonical representation of v as a byte-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:

- an element v in F_p^2 ,
- a description of p , where p is congruent to 3 modulo 4,
- a ordering parameter o , either 0 or 1.

Output:

- a string s of size $\text{Ceiling}(16 * \lg(p))$ bytes.

Method:

1. $l = 8 * \text{Ceiling}(\lg(p))$, the number of bytes needed to represent integers in Z_p .

2. $(a, b) = v$, where (a, b) in $(\mathbb{Z}_p)^2$ is the canonical representation of v in $F_p^2 = F_p / (x^2 + 1)$ as a polynomial $a + ix + b$ with $i^2 = -1$.
3. Let $a_{(256)}$ be the big-endian zero-padded fixed-length byte-string representation of a in \mathbb{Z}_p .
4. Let $b_{(256)}$ be the big-endian zero-padded fixed-length byte-string representation of b in \mathbb{Z}_p .
5. Depending on the choice of ordering o :
 - (a) If $o = 0$, then let $s = a_{(256)} || b_{(256)}$, which is the concatenation of $a_{(256)}$ followed by $b_{(256)}$.
 - (b) If $o = 1$, then let $s = b_{(256)} || a_{(256)}$, which is the concatenation of $b_{(256)}$ followed by $a_{(256)}$.
6. The fixed-length encoding of v is output as the string s .

4.4. Hashing onto a subgroup of an elliptic curve

HashToPoint(E, p, q, id) takes an identity string id and the description of a subgroup of prime order q in $E(F_p)$ or $E(F_{p^k})$ and returns a point Q_{id} of order q in $E(F_p)$ or $E(F_{p^k})$.

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

Input:

a string id ,

a description of a subgroup of prime order q on a curve E/F_p .

Output:

a point $Q_{id} = (x, y)$ of order q on E .

Method:

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

4.4.1. Type-1 curve implementation

HashToPoint1(E, p, q, id) 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 . This algorithm exploits the bijective mapping between the x and y coordinates of non-zero points on such supersingular curves.

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

Input:

a string id ,

a description of a subgroup of prime order q on a curve $E/F_p : y^2 = x^3 + 1$ where p is congruent to 11 modulo 12.

Output:

a point Q_{id} of order q on $E(F_p)$.

Method:

1. $n = q$ (compatibility mode) or p (preferred mode)
2. $y = \text{HashToRangen}(n, id)$, using Algorithm 4.1.1 (HashToRange).
3. $x = (y^2 - 1)^{1/3} = (y^2 - 1)^{(2 * p - 1) / 3}$.
4. Let $Q = (x, y)$, a non-zero point in $E(F_p)$.
5. $Q = [(p + 1) / q]Q$, a point of order q in $E(F_p)$.

4.5. Bilinear pairing

$\text{Pairing}(E, p, q, A, B)$ 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:

a description of an elliptic curve E/F_p such that $E(F_p)$ and $E(F_{p^k})$ have a subgroup of order q ,

two points A and B of order q in $E(F_p)$ or $E(F_{p^k})$.

Output:

on supersingular curves, the value of $e(A, B)$ in F_p^k 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).

4.5.1. Type-1 curve implementation

Algorithm 4.5.2 (Pairing1): computes the modified Tate pairing on type-1 curves.

Input:

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 ,

two points A and B of order q in $E(F_p)$,

Output:

the value of $e(A, B) = e(A, \phi(B))$ in $F_p^k = F_p^2$.

Method:

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

(a) Let (x, y) in $F_p \times F_p$ be the coordinates of B in $E(F_p)$.

(b) Let $\zeta = \omega^{1/3}$ in F_p^2 , with $\zeta \neq 1$. Specifically, as p is congruent to 3 modulo 4, and representing the elements of $F_p^2 = F_p[x] / (x^2 + 1)$ as polynomials $a + bx$ with $x = \omega^{1/2}$, the representation of $\zeta = (a_\zeta, b_\zeta)$ is obtained as:

i. $a_\zeta = (p - 1) / 2$.

ii. $b_\zeta = 3^{((p + 1) / 4)} \pmod{p}$.

(c) $x = x * x_\zeta$ in F_p^2 ,

(d) $B = (x, y)$ in $F_p^2 \times F_p$.

2. Compute the Tate pairing $e(A, B) = e(A, \phi(B))$ in F_p^2 using the Miller method, as in Algorithm 4.5.1 (Tate) described in [Section 4.5](#).

4.6. Ratio of bilinear pairings

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

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 4.5.2 (TateMillerSolinas), as well as the final exponentiation in step 6(a) of Algorithm 4.5.2 (TateMillerSolinas).

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

Input:

a description of an elliptic curve E/F_p such that $E(F_p)$ and $E(F_{p^k})$ have a subgroup of order q ,

four points $A, B, C,$ and $D,$ of order q in $E(F_p)$ or $E(F_{p^k})$.

Output:

on supersingular curves, the value of $e(A, B) / e(C, D)$ in F_{p^k} 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).

4.6.1. Type-1 curve implementation

Algorithm 4.6.2 (PairingRatio1). Computes the ratio of two modified Tate pairings on type-1 curves.

Input:

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 ,

four points $A, B, C,$ and $D,$ of order q in $E(F_p),$

Output:

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 $F_p^k = F_p^2 .$

Method:

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

The correct result can always be obtained, albeit more slowly, by computing the product of pairings $\text{Pairing1}(E, p, q, A, B) * \text{Pairing1}(E, p, q, C, D)$ 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 [3].

5.1. Setup

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

Input:

a curve type t (currently required to be fixed to $t = 1$),

a security parameter n (currently required to take values $n \geq 1024$).

Output:

a set of common public parameters,

a corresponding master secret.

Method:

1. Depending on the selected type t :

(a) If $t = 1,$ then Algorithm 5.1.2 (BFsetup1) is executed.

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

5.1.1. 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 of comparable security in the classical Diffie-Hellman or RSA public-key cryptosystems. For this implementation, the allowed value of n is limited to 1024, which corresponds to 80 bits of symmetric key security.

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

Input:

a security parameter n , assumed to be equal to 1024.

Output:

a set of common public parameters (t, p, q, P, P_{pub}) ,

a corresponding master secret s .

Method:

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

(a) $n_p = 512$, which will determine the size of the field F_p .

(b) $n_q = 160$, which will determine the size of the subgroup order q .

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

(a) Select an arbitrary n_q -bit prime q , i.e., such that $\text{Ceiling}(\lg(q)) = n_q$. For better performance, q is chosen as a Solinas prime, i.e., a prime of the form $q = 2^a \pm 2^b \pm 1$ where $0 < b < a$.

(b) Select a random integer r such that $p = 12 * r * q + 1$ is an n_p -bit prime, i.e., such that $\text{Floor}(\lg(p)) = n_p$.

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 $E/F_p : y^2 = x^3 + 1 \pmod{p}$.

(b) Let $P = [12 * r]P$.

(c) If $P = \theta$, then start over in step 3a.

4. 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 $P_{pub} = [s]P$.

5. (t, E, p, q, P, P_{pub}) are the common public parameters, where $E : y^2 = x^3 + 1$.

6. s is the master secret.

5.2. Public key derivation

`BFderivePubl` takes an identity string `id` and a set of public parameters, and returns a point `Q_id`.

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

Input:

an identity string `id`,

a set of common public parameters (t, E, p, q, P, P_{pub}) .

Output:

a point `Q_id` of order q in $E(F_p)$ or $E(F_{p^k})$.

Method:

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

5.3. Private key extraction

`BFextractPriv` takes an identity string `id`, and a set of public parameters and corresponding master secret, and returns a point `S_id`.

Algorithm 4.3.1 (BFextractPriv): extracts the private key corresponding to an identity string.

Input:

an identity string id ,

a set of common public parameters (t, E, p, q, P, P_{pub}) .

Output:

a point S_{id} or order q in $E(F_p)$.

Method:

1. $Q_{id} = \text{HashToPoint}(E, p, q, id)$ using Algorithm 4.4.1 (HashToPoint).
2. $S_{id} = [s]Q_{id}$.

5.4. Encryption

BFencrypt takes three inputs: a public parameter block, an identity id , and a plaintext m . The plaintext is intended to be a symmetric session key, although variable-sized short messages are allowed.

Algorithm 5.4.1 (BFencrypt): encrypts a short message or session key for an identity string.

Input:

a plaintext string m of size $|m|$ bytes,

a recipient identity string id ,

a set of public parameters.

Output:

a ciphertext tuple (U, V, W) in $E(F_p) \times \{0, \dots, 255\}^{20} \times \{0, \dots, 255\}^{|m|}$.

Method:

1. Let the public parameter set be comprised of a prime p , a curve E/F_p , the order q of a large prime subgroup of $E(F_p)$, and two points P and P_{pub} of order q in $E(F_p)$.

2. $Q_{id} = \text{HashToPoint}(E, p, q, id)$, using Algorithm 4.4.1 (`HashToPoint`), which results in a point of order q in $E(F_p)$ or $E(F_{p^k})$.
3. Select s random 160-bit vector ρ , represented as 20-byte string in big-endian convention.
4. $t = \text{SHA1}(m)$, a 20-byte string resulting from the SHA1 algorithm.
5. $l = \text{HashToRange}(q, \rho || t)$, an integer in the range 0 to $q - 1$ resulting from applying Algorithm 4.1.1 (`HashToRange`) to the 40-byte concatenation of ρ and t .
6. $U = [l]P$, which is a point of order q in $E(F_p)$.
7. $\theta = \text{Pairing}(E, p, q, P_{pub}, Q_{id})$, which is an element of the extension field F_{p^k} obtained using the modified Tate pairing of Algorithm 4.5.1 (`Pairing`).
8. Let $\theta^l = \theta^l$, which is θ raised to the power of l in F_{p^k} .
9. Let $z = \text{Canonical}(p, k, \theta, \theta^l)$, using Algorithm 4.3.1 (`Canonical`), the result of which is a canonical string representation of θ^l .
10. Let $w = \text{SHA1}(z)$ using the SHA1 hashing algorithm, the result of which is a 20-byte string.
11. Let $V = w \text{ XOR } \rho$, which is the 20-byte long bit-wise exclusive-OR of w and ρ .
12. Let $W = \text{HashStream}(|m|, \rho \text{ XOR } m)$, which is the bit-wise exclusive-OR of m with the first $|m|$ bytes of the pseudo-random stream produced by Algorithm 4.2.1 (`HashStream`) with seed ρ .
13. The ciphertext is the triple (U, V, W) .

5.5. Decryption

BFdecrypt takes three inputs: a public parameter block, a private key block, and a ciphertext parsed as (U, V, W) .

Algorithm 5.5.1 (BFdecrypt): decrypts a short message or session key using a private key.

Input:

a private key point S_{id} of order q in $E(F_p)$,

a ciphertext triple (U, V, W) in $E(F_p) \times \{0, \dots, 255\}^{20} \times \{0, \dots, 255\}^*$.

a set of public parameters.

Output:

a decrypted plaintext m , or an invalid ciphertext flag.

Method:

1. Let the public parameter set be comprised of a prime p , a curve E/F_p , the order q of a large prime subgroup of $E(F_p)$, and two points P and P_{pub} of order q in $E(F_p)$.
2. Let $\theta = \text{Pairing}(E, p, q, U, S_{id})$ by applying the modified Tate pairing of Algorithm 4.5.1 (Pairing).
3. Let $z = \text{Canonical}(p, k, \theta, \theta)$ using Algorithm 4.3.1 (Canonical), the result of which is a canonical string representation of θ .
4. Let $w = \text{SHA1}(z)$, using the SHA1 hashing algorithm, the result of which is a 20-byte string.
5. Let $\rho = w \text{ XOR } V$, the bit-wise XOR of w and V .
6. Let $m = \text{HashStream}(|W|, \rho) \text{ XOR } W$, which is the bit-wise exclusive-OR of m with the first $|W|$ bytes of the pseudo-random stream produced by Algorithm 4.2.1 (HashStream) with seed ρ .
7. Let $t = \text{SHA1}(m)$ using the SHA1 algorithm.
8. Let $l = \text{HashToRange}(q, \rho || t)$ using Algorithm 4.1.1 (HashToRange) on the 40-byte concatenation of ρ 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. Wrapper methods for the BF system

This chapter describes a number of wrapper methods providing the identity-based cryptosystem functionalities using concrete encodings. The following functions are presently given based on the Boneh-Franklin algorithms.

6.1. Private key generator (PKG) setup

Algorithm 6.1.1 (BFwrapperPKGSetup): randomly selects a PKG master secret and a set of public parameters.

Input:

- a curve type t ,
- a security parameter n .

Output:

- a common public parameter block π ,
- a corresponding master secret block σ .

Method:

1. Perform Algorithm 5.1.1 (BFsetup) on parameters t and n , producing a public parameter set and a master secret.
2. Apply Algorithm 7.2.1 (BFencodeParams) on the public parameter set obtained in step 1 to get the public parameter block π .
3. Apply Algorithm 7.3.1 (BFencodeMaster) on the master secret obtained in step 1 to get the master secret block σ .

6.2. Private key extraction by the PKG

Algorithm 5.2.1 (BFwrapperPrivateKeyExtract): extraction by the PKG of a private key corresponding to an identity.

Input:

- a master secret block σ ,
- a corresponding public parameter block π ,
- an identity string id .

Output:

a private key block κ_{id}

Method:

1. Apply Algorithm 7.2.2 (BFdecodeParams) to the public parameter block π to obtain the public parameters, comprising a prime p , a curve E/F_p , the order q of a large prime subgroup of $E(F_p)$, and two points P and P_{pub} of order q in $E(F_p)$.
2. Apply Algorithm 7.3.2 (BFdecodeMaster) on the master secret block σ to obtain the master secret s .
3. Perform Algorithm 5.3.1 (BFextractPriv) on the identity id , using the decoded parameters and secret, to produce a private key point S_{id} .
4. Apply Algorithm 7.4.1 (BFencodePrivate) to S_{id} to produce a private key block κ_{id} .

6.3. Session key encryption

Algorithm 5.3.1 (BFwrapperSessionKeyEncrypt): encrypts a short message or session key for an identity.

Input:

a public parameter block π ,

a recipient identity string id ,

a plaintext string m (possibly comprising the concatenation of a pair of random session keys for symmetric encryption and message authentication purposes on a larger plaintext).

Output:

a ciphertext block

Method:

1. Apply Algorithm 7.2.2 (BFdecodeParams) on the public parameter block π to obtain a set of public parameters, comprising a prime p , a curve E/F_p , the order q of a large prime subgroup of $E(F_p)$, and two points P and P_{pub} of order q in $E(F_p)$.

2. Perform Algorithm 5.4.1 (BFencrypt) on the plaintext m for identity id using the decoded set of parameters, to obtain a ciphertext tuple (U, V, W) .
3. Apply Algorithm 7.5.1 (BFencodeCiphertext) on (U, V, W) to obtain a serialized ciphertext string

Algorithm 6.3.2 (BFwrapperSessionKeyDecrypt): decrypts a short message or session key using a private key.

Input:

- a public parameter block π ,
- a private key block κ ,
- a ciphertext block γ .

Output:

a decrypted plaintext string m , or an error flag signaling an invalid ciphertext.

Method:

1. Apply Algorithm 7.2.2 (BFdecodeParams) on the public parameter block π to obtain the public parameters, comprising a prime p , a curve E/F_p , the order q of a large prime subgroup of $E(F_p)$, and two points P and P_{pub} of order q in $E(F_p)$.
2. Apply Algorithm 7.4.2 (BFdecodePrivate) to κ to obtain a private key point S_{id} .
3. Apply Algorithm 7.5.2 (BFdecodeCiphertext) to γ to obtain a ciphertext triple (U, V, W) .
4. Perform Algorithm 5.5.1 (BFdecrypt) on (U, V, W) using the private key S_{id} and the decoded set of public parameters, to obtain decrypted plaintext m , or an invalid ciphertext flag.
 - (a) If the decryption was successful, return the plaintext m .
 - (b) Otherwise, raise an error condition.

7. Concrete encoding guidelines for BF

This section specifies a set of concrete encoding schemes for the inputs and outputs of the previously described algorithms. ASN.1 encodings are specified in [Section 11](#) of this document.

7.1. Encoding of points on a curve

Algorithm 7.1.1 (EncodePoint): encodes a point in $E(F_p)$ in an exportable format.

Input:

a non-zero point Q in $E(F_p)$.

Output:

a fixed-length (for given p) byte-string encoding of Q .

Method:

1. Let (x, y) in $F_p \times F_p$ be the coordinates of P , where (x, y) satisfy the equation of E .
2. The point P is then encoded as a `FpPoint` using the ASN.1 rules given in the ASN.1 module given in [Section 11](#) of this document.

Algorithm 6.1.2 (DecodePoint): decodes a point in $E(F_p)$ from an exportable format.

Input:

a byte-string encoding of a non-zero point Q in $E(F_p)$.

Output:

$Q = (x, y)$.

Method:

1. The string is parsed and decoded as a pair (x, y) , where x and y are integers in Z_p .
2. Q is reconstructed as (x, y) .

7.2. Public parameters blocks

Algorithm 7.2.1 (BFencodeParams): encodes a BF public parameter set in an exportable format.

Input:

a set of public parameters (t, E, p, q, P, P_{pub}).

Output:

a public parameter block pi, represented as a byte string.

Method:

1. Separate encodings for E, p, q, P, P_{pub} are obtained as follows:

(a) If t = 1, execute Algorithm 7.2.3 (BFencodeParams1).

2. The separate encodings as well as a type indicator flag for t are then serialized in any suitable manner as dictated by the application.

Algorithm 7.2.2 (BFdecodeParams): imports a BF public parameter block from a serialized format.

Input:

a public parameter block pi, represented as a byte string.

Output:

a set of public parameters (t, E, p, q, P, P_{pub}).

Method:

1. Identify from the appropriate flag the type t of curve upon which the parameter block is based.

2. Then:

(a) If t = 1, execute Algorithm 7.2.4 (BFdecodeParams1).

7.2.1. Type-1 implementation

Algorithm 7.2.3 (BFencodeParams1): encodes a BF type-1 public parameter set in an exportable format.

Input:

a set of public parameters (t, E, p, q, P, P_{pub}) with t = 1.

Output:

separate encodings for each of the E, p, q, P, P_{pub} components.

Method:

1. E : $y^2 = x^3 + a * x + b$ is represented as a constant string, such as the empty string, since a and b are invariant for type-1 curves.
2. $p = 12 * r * q . 1$ is represented as the smaller integer r, encoded, e.g., using a big-endian byte-string representation.
3. $q = 2^a + s * 2^b + c$, where a, b are small and c and s are either 1 or -1, is compactly represented as the 4-tuple (a, b, c, s).
4. P = (x_P , y_P) in F_p x F_p is represented using the point compression technique of Algorithm 7.1.1 (EncodePoint).
5. P_{pub} is similarly encoded using Algorithm 7.1.1 (EncodePoint).

Algorithm 7.2.4 (BFdecodeParams1): decodes the components of a BF type-1 public parameter block.

Input:

separate encodings for each one of E, p, q, P, P_{pub}.

Output:

a set of public parameters (t, E, p, q, P, P_{pub}) with t = 1.

Method:

1. The equation of E is set to E = E : $y^2 = x^3 + 1$, as is always the case for type-1 curves. The actual encoding of E is ignored.
2. The encoding of q is parsed as (a, b, c, s), and its value set to $q = 2^a + s * 2^b + c$.
3. The encoding of p is parsed as the integer r, from which p is given by $p = 12 * r * q . 1$.

4. P is reconstructed from its encoding (x, y) using the point decompression technique of Algorithm 7.1.2 (DecodePoint).

5. P_{pub} is similarly reconstructed from its encoding using Algorithm 7.1.2 (DecodePoint).

7.3. Master secret blocks

Algorithm 6.3.1 (BFencodeMaster): encodes a BF master secret in an exportable format.

Input:

a master secret integer s between 2 and $q - 1$.

Output:

a master secret block σ , represented as a byte string.

Method:

1. σ is constructed as the unsigned big-endian byte-string encoding of s of length $8 * \text{Ceiling}(\lg(p))$.

Algorithm 7.3.2 (BFdecodeMaster): decodes a BF master secret from a block in exportable format.

Input:

a master secret block σ , represented as a byte string.

Output:

a master secret integer s in between 2 and $q - 1$.

Method:

1. $s = \text{Value}(\sigma)$, where σ is interpreted in the unsigned big endian convention.

7.4. Private key blocks

Algorithm 6.4.1 (BFencodePrivate): encodes a BF private key point in an exportable format.

Input:

a private key point S_{id} in $E(F_p)$.

Output:

a private key block κ , represented as a byte string.

Method:

1. κ is obtained by applying Algorithm 7.1.1 (EncodePoint) to S_{id} .

Algorithm 7.4.2 (BFdecodePrivate): decodes a BF private key point from an exportable format.

Input:

a private key block κ , represented as a byte string.

Output:

a private key point S_{id} in $E(F_p)$.

Method:

1. κ is parsed and decoded into a point S_{id} in $E(F_p)$ using Algorithm 7.1.2 (DecodePoint).

7.5. Ciphertext blocks

Algorithm 7.5.1 (BFencodeCiphertext): encodes a BF ciphertext tuple in an exportable format.

Input:

a ciphertext tuple (U, V, W) in $E(F_p) \times \{0, \dots, 255\}^{20} \times \{0, \dots, 255\}^*$.

Output:

a ciphertext block γ , represented as a byte string.

Method:

1. $U = (x, y)$ is first encoded as a fixed-length string using Algorithm 7.1.1 (EncodePoint).

2. Gamma is obtained as the encoding of U, concatenated with the fixed-length string V, and the variable length string W, both already in byte-string format.

Algorithm 7.5.2 (BFdecodeCiphertext): decodes a BF ciphertext tuple from an exportable format.

Input:

a ciphertext block gamma, represented as a byte string.

Output:

a ciphertext tuple (U, V, W) in $E(F_p) \times \{0, \dots, 255\}^{20} \times \{0, \dots, 255\}^*$.

Method:

1. Gamma is parsed as a 3-tuple comprising a fixed-length encoding of U, followed by a 20-byte string V, followed by an arbitrary-length string W.
2. U in $E(F_p)$ is then recovered by applying Algorithm 7.1.2 (DecodePoint) on its encoding.

8. The Boneh-Boyen BB1 cryptosystem

This chapter describes the algorithms constituting the first of the two Boneh-Boyen identity-based cryptosystems proposed in [2]. The description follows the practical implementation given in [2].

8.1. Setup

Algorithm 8.1.1 (BBsetup). Randomly selects a set of master secrets and the associated public parameters.

Input:

a curve type t (currently required to be fixed to $t = 1$),

a security parameter n (currently required to take values $n \geq 1024$).

Output:

a set of common public parameters,

a corresponding master secret.

Method:

1. Depending on the selected type t :

(a) If $t = 1$, then Algorithm 8.1.2 (BBsetup1) is executed.

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

8.1.1. Type-1 curve implementation

BBsetup1 takes a security parameter n as input. For type-1 curves, the scale of n corresponds to the modulus bit-size believed of comparable security in the classical Diffie-Hellman or RSA public-key cryptosystems. For this implementation, allowed values of n are limited to 1024, 2048, and 3072, which correspond to the equivalent security level ranging from 80-, 112- and 128-bit symmetric keys respectively.

Algorithm 7.1.2 (BBsetup1): randomly establishes a master secret and public parameters for type-1 curves.

Input:

a security parameter n , either 1024, 2048 or 3072.

Output:

a set of common public parameters ($t, k, E, p, q, P, P_1, P_2, P_3, v$),

a corresponding triple of master secrets (α, β, γ).

Method:

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

(a) $n_p = n / 2$, which will determine the size of the field F_p .

(b) if $n = 1024$, $n_q = 160$; if $n = 2048$, $n_q = 224$; if $n = 3072$, $n_q = 256$, which will determine the size of the subgroup order q .

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

(a) Select an arbitrary n_q -bit prime q , i.e., such that $\text{Ceiling}(\lg(p)) = n_q$. For better performance, q is chosen as a Solinas prime, i.e., a prime of the form $q = 2^a \pm 2^b \pm 1$ where $0 < b < a$.

(b) Select a random integer r such that $p = 12 * r * q + 1$ is an n_p -bit prime, i.e., such that $\text{Ceiling}(\lg(p)) = n_p$.

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 $E/F_p : y^2 = x^3 + 1 \pmod{p}$.

(b) Let $P = [12 * r]P$.

(c) If $P = 1$, then start over in step 3a.

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

(a) Select three random integers α, β, γ , 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 $v = \text{Pairing}(E, p, q, P_1, P_2)$, which is an element of the extension field F_{p^2} obtained using the modified Tate pairing of Algorithm 3.5.1 (Pairing).

5. $(t, k, E, p, q, P, P_1, P_2, P_3, v)$ are the common public parameters, where $t = 1$, $k = 2$, and $E : y^2 = x^3 + 1$.

6. (α, β, γ) constitute the master secret.

8.2. Public key derivation

`BBderivePubl` takes an identity string `id` and a set of public parameters, and returns an integer `h_id`.

Algorithm 7.2.1 (`BBderivePubl`): derives the public key corresponding to an identity string.

Input:

an identity string id ,

a set of common public parameters $(t, k, E, p, q, P, P_1, P_2, P_3, v)$.

Output:

an integer h_{id} modulo q .

Method:

1. Let $h_{id} = \text{HashToRange}(id)$, using Algorithm 3.1.1 (HashToRange).

8.3. Private key extraction

`BBextractPriv` takes an identity string id , and a set of public parameters and corresponding master secrets, and returns a private key consisting of two points D_0 and D_1 .

Algorithm 8.3.1 (`BBextractPriv`): extracts the private key corresponding to an identity string.

Input:

an identity string id ,

a set of common public parameters $(t, k, E, p, q, P, P_1, P_2, P_3, v)$.

Output:

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 $hid = \text{HashToRange}(q, id)$, using Algorithm 3.1.1 (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 .

8.4. Encryption

BBencrypt takes three inputs: a set of public parameters, an identity id , and a plaintext m . The plaintext is intended to be a short random session key, although messages of arbitrary size are in principle allowed.

Algorithm 7.4.1 (BBencrypt): encrypts a short message or session key for an identity string.

Input:

a plaintext string m of size $|m|$ bytes,

a recipient identity string id ,

a set of public parameters $(t, k, E, p, q, P, P_1, P_2, P_3, v)$.

Output:

a ciphertext tuple (u, C_0, C_1, y) in $F_q \times E(F_p) \times E(F_p) \times \{0, \dots, 255\}^{|m|}$.

Method:

1. Let the public parameter set be comprised of a prime p , a curve E/F_p , the order q of a large prime subgroup of $E(F_p)$, four points P, P_1, P_2, P_3 , of order q in $E(F_p)$, and an extension field element v of order q in F_{p^2} .

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

3. 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} .

4. Calculate the point C_0 as follows:

(a) Let $C_0 = [s]P$.

5. Calculate the point C_1 as follows:

(a) Let $hid = \text{HashToRange}_q(id)$, using Algorithm 3.1.1 (HashToRange).

(b) Let $y = s * h_{id}$ in F_q .

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

6. Obtain canonical string representations of certain elements:

(a) $\psi = \text{Canonical}(p, k, 1, w)$ using Algorithm 3.3.1 (Canonical), the result of which is a canonical byte-string representation of w .

(b) Let $l = \text{Ceiling}(8 * \lg(p))$, the number of bytes needed to represent integers in F_p , and represent each of these F_p elements as a big-endian zero-padded byte-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 .

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

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

i. Let $\zeta = \text{SHA1}(\psi)$, using the SHA1 hashing algorithm; the result is a 20-byte string.

ii. Let $\xi = \text{SHA1}(\zeta || \psi)$, using the SHA1 hashing algorithm; the result is a 20-byte string.

iii. Let $h = \xi || \zeta$, the 40-byte concatenation of the previous two SHA1 outputs.

(b) Let $y = \text{HashStream}(|m|, h) \text{ XOR } m$, which is the bit-wise exclusive-OR of m with the first $|m|$ bytes of the pseudo-random stream produced by Algorithm 3.2.1 (HashStream) with seed h .

8. 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 $\eta = \text{SHA1}(\sigma)$, using the SHA1 hashing algorithm to get a 20-byte string.

iii. Let $\mu = \text{SHA1}(\eta \parallel \sigma)$, using the SHA1 hashing algorithm to get a 20-byte string.

iv. Let $h = \mu \parallel \eta$, the 40-byte concatenation of the previous two SHA1 outputs.

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

i. Let $\rho = \text{HashToRange}(h)$ to get an integer in \mathbb{Z}_q .

ii. Let $u = s + \rho \pmod{q}$.

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

8.5. Decryption

BBdecrypt takes three inputs: a set of public parameters, a private key (D_0, D_1) , and a ciphertext parsed as (u, C_0, C_1, y) . It outputs a message m , or signals an error if the ciphertext is invalid for the given key.

Algorithm 7.5.1 (BBdecrypt): decrypts a short message or session key using a private key.

Input:

a private key given as a pair of points (D_0, D_1) of order q in $E(\mathbb{F}_p)$,

a ciphertext quadruple (u, C_0, C_1, y) in $\mathbb{Z}_q \times E(\mathbb{F}_p) \times E(\mathbb{F}_p) \times \{0, \dots, 255\}^*$.

a set of public parameters.

Output:

a decrypted plaintext m , or an invalid ciphertext flag.

Method:

1. Let the public parameter set be comprised of a prime p , a curve E/\mathbb{F}_p , the order q of a large prime subgroup of $E(\mathbb{F}_p)$, four points

P, P_1, P_2, P_3 , of order q in $E(F_p)$, and an extension field element v of order q in F_{p^2} .

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

3. Obtain canonical string representations of certain elements:

(a) $\text{psi} = \text{Canonical}(p, k, 1, w)$, using Algorithm 4.3.1 (Canonical); the result is a canonical byte-string representation of w .

(b) Let $l = \text{Ceiling}(8 * \lg(p))$, the number of bytes needed to represent integers in F_p , and represent each of these F_p elements as a big-endian zero-padded byte-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 .

4. 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 $\text{zeta} = \text{SHA1}(\text{psi})$, using the SHA1 hashing algorithm to get a 20-byte string.

ii. Let $\text{xi} = \text{SHA1}(\text{zeta} || \text{psi})$, using the SHA1 hashing algorithm to get a 20-byte string.

iii. Let $h = \text{xi} || \text{zeta}$, the 40-byte concatenation of the previous two SHA1 outputs.

(b) Let $m = \text{HashStream}(|y|, h) _XOR y$, which is the bit-wise exclusive-OR of y with the first $|y|$ bytes of the pseudo-random stream produced by Algorithm 3.2.1 (HashStream) with seed h .

5. 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 $\sigma = (y_1)_{(256^1)} || (x_1)_{(256^1)} || (y_0)_{(256^1)} || (x_0)_{(256^1)} || y || \psi$ be the concatenation of y and the five indicated strings in the specified order.

ii. Let $\eta = \text{SHA1}(\sigma)$ using the SHA1 hashing algorithm to get a 20-byte string.

iii. Let $\mu = \text{SHA1}(\eta || \sigma)$, using the SHA1 hashing algorithm to get a 20-byte string.

iv. Let $h = \mu || \eta$, the 40-byte concatenation of the previous two SHA1 outputs.

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

i. Let $\rho = \text{HashToRange}_q(h)$ to get an integer in \mathbb{Z}_q .

ii. Let $s = u - \rho \pmod{q}$.

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

(a) Test whether the equality $w = v^s$ holds in \mathbb{F}_{p^2} .

(b) Test whether the equality $C_0 = [s]P$ holds in $E(\mathbb{F}_p)$.

7. Adjudication and final output:

(a) If either of the tests performed in step 6 fails, the ciphertext is rejected, and no decryption is output.

(b) Otherwise, i.e., when both tests performed in step 6 succeed, the decrypted message is output.

9. Wrapper methods for the BB1 system

This section describes a number of wrapper methods providing the identity-based cryptosystem functionalities using concrete encodings. The following functions are presently given based on the Boneh-Franklin algorithms.

9.1. Private key generator (PKG) setup

Algorithm 9.1.1 (BBwrapperPKGSetup): randomly selects a PKG master secret and a set of public parameters.

Input:

- a curve type t ,
- a security parameter n .

Output:

- a common public parameter block π ,
- a corresponding master secret block σ .

Method:

1. Perform Algorithm 8.1.1 (BBsetup) on parameters t and n , producing a set of public parameters and master secret.
2. Apply Algorithm 10.2.1 (BBencodeParams) on the public parameters obtained in step 1 to get the public parameter block π .
3. Apply Algorithm 10.3.1 (BBencodeMaster) on the master secrets obtained in step 1 to get the master secret block σ .

9.2. Private key extraction by the PKG

Algorithm 9.2.1 (BBwrapperPrivateKeyExtract): extraction by the PKG of a private key corresponding to an identity.

Input:

- a master secret block σ ,
- a corresponding public parameter block π ,
- an identity string id .

Output:

- a private key block κ_{id} .

Method:

1. Apply Algorithm 10.2.2 (BBdecodeParams) on the public parameter block π to obtain the public parameters, comprising a prime p , the parameters of a curve E/F_p with some embedding degree k , the order q of a large prime subgroup of $E(F_p)$, four points P, P_1, P_2, P_3 , of

order q in $E(F_p)$, and an element v of order q in the extension field F_{p^k} of degree k .

2. Apply Algorithm 10.3.2 (BBdecodeMaster) on the master secret block σ to obtain the master secret (α , β , γ).
3. Perform Algorithm 8.3.1 (BBextractPriv) on the identity id , using the decoded public parameters and master secret, to produce a private key (D_0 , D_1).
4. Apply Algorithm 10.4.1 (BBencodePrivate) on the private key to produce a private key block κ_{id} .

9.3. Session key encryption

Algorithm 9.3.1 (BBwrapperSessionKeyEncrypt): encrypts a short message or session key for an identity.

Input:

a public parameter block π ,

a recipient identity string id ,

a plaintext string m (possibly comprising the concatenation of a pair of random session keys for symmetric encryption and message authentication purposes on a larger plaintext).

Output:

a ciphertext block ω .

Method:

1. Apply Algorithm 10.2.2 (BBdecodeParams) on the public parameter block π to obtain the public parameters, comprising a prime p , the parameters of a curve E/F_p with some embedding degree k , the order q of a large prime subgroup of $E(F_p)$, four points P , P_1 , P_2 , P_3 , of order q in $E(F_p)$, and an element v of order q in the extension field F_{p^k} .
2. Perform Algorithm 8.4.1 (BBencrypt) on the plaintext m for identity id using the decoded set of parameters, to obtain a ciphertext quadruple (u, C_0, C_1, y) .
3. Apply Algorithm 10.5.1 (BBencodeCiphertext) on the ciphertext (u, C_0, C_1, y) to obtain a string representation of ω .

Algorithm 9.3.2 (BBwrapperSessionKeyDecrypt): decrypts a short message or session key using a private key.

Input:

- a public parameter block π ,
- a private key block κ ,
- a ciphertext block ω .

Output:

a decrypted plaintext string m , or an error flag signaling an invalid ciphertext.

Method:

1. Apply Algorithm 10.2.2 (BBdecodeParams) on the public parameter block π to obtain the public parameters, comprising a prime p , the parameters of a curve E/F_p with some embedding degree k , the order q of a large prime subgroup of $E(F_p)$, four points P, P_1, P_2, P_3 , of order q in $E(F_p)$, and an element v of order q in the extension field F_{p^k} .
2. Apply Algorithm 10.4.2 (BBdecodePrivate) on κ to obtain the private key points (D_0, D_1) .
3. Apply Algorithm 10.5.2 (BBdecodeCiphertext) on ω to obtain a ciphertext quadruple (u, C_0, C_1, y) .
4. Perform Algorithm 8.5.1 (BBdecrypt) on (u, C_0, C_1, y) using the private key (D_0, D_1) and the decoded set of public parameters, to obtain decrypted plaintext m , or an invalid ciphertext flag.
 - (a) If the decryption was successful, return the plaintext string m .
 - (b) Otherwise, raise an error condition.

10. Concrete encoding guidelines for BB1

This section specifies a set of concrete encoding schemes for the inputs and outputs of the previously described algorithms. ASN.1 encodings are specified in [Section 11](#) of this document.

10.1. Encoding of points on a curve

We refer to the description of Algorithm 7.1.1 (EncodePoint) and Algorithm 7.1.2 (DecodePoint).

10.2. Public parameters blocks

Algorithm 10.2.1 (BBencodeParams): encodes a BB1 public parameter set in an exportable format.

Input:

a set of public parameters $(t, k, E, p, q, P, P_1, P_2, P_3, v)$.

Output:

a public parameter block pi , represented as a byte string.

Method:

1. Separate encodings for $k, E, p, q, P, P_1, P_2, P_3$ are obtained as follows:

(a) If $t = 1$, execute Algorithm 10.2.3 (BBencodeParams1).

2. The separate encodings as well as a type indicator flag for t are then serialized in any suitable manner as dictated by the application.

Algorithm 10.2.2 (BBdecodeParams): imports a BB1 public parameter block from a serialized format.

Input:

a public parameter block pi , represented as a byte string.

Output:

a set of public parameters $(t, k, E, p, q, P, P_1, P_2, P_3, v)$.

Method:

1. Identify from the appropriate flag the type t of curve upon which the parameter block is based.

2. Then:

(a) If $t = 1$, execute Algorithm 10.2.4 (BBdecodeParams1).

10.2.1. Type-1 implementation

Algorithm 10.2.3 (BBencodeParams1): encodes a BB1 type-1 public parameter set in an exportable format.

Input:

a set of public parameters $(t, k, E, p, q, P, P_1, P_2, P_3, v)$ with $t = 1$.

Output:

separate encodings for each of the $k, E, p, q, P, P_1, P_2, P_3$ components (v is redundant and omitted).

Method:

1. $E : y^2 = x^3 + a * x + b$ and $k = 2$ are represented as a constant string, such as the empty string, since the coefficients a and b and the embedding degree k are invariant for type-1 curves.
2. $p = 12 * r * q + 1$ is represented as the smaller integer r , encoded, e.g., using a big-endian byte-string representation.
3. $q = 2^a + s * 2^b + c$, where a, b are small and both c and s are either 1 or -1 is compactly represented as the 4-tuple (a, b, c, s) .
4. $P = (x_P, y_P)$ in $F_p \times F_p$ is represented using the point compression technique of Algorithm 7.1.1 (EncodePoint).
5. Each of $P_1, P_2,$ and P_3 is similarly encoded using Algorithm 7.1.1 (EncodePoint).

Algorithm 10.2.4 (BBdecodeParams1): decodes the components of a BB1 type-1 public parameter block.

Input:

separate encodings for each one of $k, E, p, q, P, P_1, P_2, P_3$.

Output:

a set of public parameters $(t, k, E, p, q, P, P_1, P_2, P_3, v)$ with $t = 1$.

Method:

1. The equation of E is set to $E : y^2 = x^3 + 1$, as is always the case for type-1 curves.
2. The embedding degree is set to $k = 2$ for type-1 curves.
3. The encoding of q is parsed as (a, b, c, s) , and its value set to $q = 2^a + s * 2^b + c$.
4. The encoding of p is parsed as the integer r, from which p is given by $p = 12 * r * q - 1$.
5. P is reconstructed from its encoding (x, y) using the point decompression technique of Algorithm 7.1.2 (DecodePoint).
6. Each of $P_1, P_2,$ and P_3 is reconstructed in a similar manner from its encoding using Algorithm 7.1.2 (DecodePoint).
7. The extension field element v is reconstructed as $v = \text{Pairing}(E, p, q, P_1, P_2)$ using Algorithm 4.5.1 (Pairing).

10.3. Master secret blocks

Algorithm 10.3.1 (BBencodeMaster): encodes a BB1 master secret in an exportable format.

Input:

a master secret triple of integers (α, β, γ) in $(\mathbb{Z}_+^q)^3$.

Output:

a master secret block σ , represented as a byte string.

Method:

1. Encode each integer as an unsigned big-endian byte-string of fixed length $\text{Ceiling}(8 * \lg(q))$, or, when q is a Solinas prime $q = 2^a \pm 2^b \pm 1$, of length $\text{Ceiling}((a + 1) / 8)$:
 - (a) σ_α to represent α .
 - (b) σ_β to represent β .
 - (c) σ_γ to represent γ .

2. $\text{Sigma} = \text{sigma_alpha} \parallel \text{sigma_beta} \parallel \text{sigma_gamma}$ is the concatenation of these strings.

Algorithm 10.3.2 (BBdecodeMaster): decodes a BB1 master secret from a block in exportable format.

Input:

a master secret block sigma , represented as a byte string.

Output:

a master secret triple of integers $(\text{alpha}, \text{beta}, \text{gamma})$ in $(\mathbb{Z}_q)^3$.

Method:

1. Parse sigma as $\text{sigma_alpha} \parallel \text{sigma_beta} \parallel \text{sigma_gamma}$, where each substring is a byte string of fixed length $\text{Ceiling}(8 * \lg(q))$, or, when q is a Solinas prime $q = 2^a \pm 2^b \pm 1$, of length $\text{Ceiling}((a + 1) / 8)$.

2. Decode each substring as an integer in unsigned big-endian byte-string representation:

(a) $\text{alpha} = \text{Value}(\text{sigma_alpha})$.

(b) $\text{beta} = \text{Value}(\text{sigma_beta})$.

(c) $\text{gamma} = \text{Value}(\text{sigma_gamma})$.

10.4. Private key blocks

Algorithm 10.4.1 (BBencodePrivate): encodes a BB1 private key in an exportable format.

Input:

a private key pair of points (D_0, D_1) in $E(F_p) \times E(F_p)$.

Output:

a private key block kappa , represented as a byte string.

Method:

1. Encode each point separately:

(a) κ_0 is obtained by applying Algorithm 7.1.1 (EncodePoint) to D_0 .

(b) κ_1 is obtained by applying Algorithm 7.1.1 (EncodePoint) to D_0 .

2. $\text{Kappa} = \kappa_0 \parallel \kappa_1$.

Algorithm 10.4.2 (BBdecodePrivate): decodes a BB1 private key from an exportable format.

Input:

a private key block kappa , represented as a byte string.

Output:

a private key pair of point (D_0, D_1) in $E(F_p) \times E(F_p)$.

Method:

1. Decode each point separately:

(a) The first prefix of kappa is parsed and decoded into a point D_0 in $E(F_p)$ using Algorithm 7.1.2 (DecodePoint).

(b) The remainder of kappa is parsed and decoded into a point D_1 in $E(F_p)$ using Algorithm 7.1.2 (DecodePoint).

10.5. Ciphertext blocks

Algorithm 10.5.1 (BBencodeCiphertext). Encodes a BB1 ciphertext tuple in an exportable format.

Input:

a ciphertext tuple (u, C_0, C_1, y) in $Z_q \times E(F_p) \times E(F_p) \times \{0, \dots, 255\}^*$.

Output:

a ciphertext block ω , represented as a byte string.

Method:

1. Let χ_0 be the fixed-length encoding of $C_0 = (x_0, y_0)$ using Algorithm 7.1.1 (EncodePoint).

2. Let chi_1 be the fixed-length encoding of $C_1 = (x_1, y_1)$ using Algorithm 7.1.1 (EncodePoint).
3. Let nu be the encoding of u as an unsigned big-endian byte-string of fixed length $\text{Ceiling}(8 * \lg(q))$, or, when q is a Solinas prime $q = 2^a \pm 2^b \pm 1$, of length $\text{Ceiling}((a + 1)/8)$.
4. $\text{Omega} = \text{chi}_0 || \text{chi}_1 || \text{nu} || y$ is the concatenation of these three strings and y .

Algorithm 10.5.2 (BBdecodeCiphertext): decodes a BB1 ciphertext tuple from an exportable format.

Input:

a ciphertext block omega , represented as a byte string.

Output:

a ciphertext tuple (u, C_0, C_1, y) in $Z_q \times E(F_p) \times E(F_p) \times \{0, \dots, 255\}^*$.

Method:

1. Omega is parsed as a quadruple comprising a fixed-length encoding of C_0 , a fixed-length encoding of C_1 , a fixed-length encoding of u , and the arbitrary-length string y :
 - (a) C_0 in $E(F_p)$ is first recovered by applying Algorithm 7.1.2 (DecodePoint) on the first parsed component of omega .
 - (b) C_1 in $E(F_p)$ is next recovered by applying Algorithm 7.1.2 (DecodePoint) on the second parsed component of omega .
 - (c) u in Z_q is then recovered from its unsigned big-endian byte-string representation in the third parsed component of omega , of length $\text{Ceiling}(8 * \lg(q))$, or, when q is a Solinas prime $q = 2^a \pm 2^b \pm 1$, of length $\text{Ceiling}((a + 1)/8)$.
 - (d) y is finally taken as the remainder of omega .

11. ASN.1 module

This section defines the ASN.1 module for the encodings discussed in sections [7](#) and [10](#).

```
IBCS { joint-iso-itu(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 of the IBCS standard only supports IBE over
-- type-1 curves. In the current version, the Curve type is
-- always set to NULL, although future versions will use it.
--

IMPORTS Curve
    FROM X9-62-module
        { iso(1) member-body(2) us(840) ansi-x9-62(10045) module(5) 1
};

ibcs OBJECT IDENTIFIER ::= {
    joint-iso-itu(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)
}
```



```
--
-- Supporting types
--

--
-- Encoding of a point on an elliptic curve E/Fp.
--

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

--
-- Encoding of a Solinas prime.
--
-- Encodes a Solinas prime of the form
--  $q = 2^a + s * 2^b + c$  with the integers a, b, c, and s.
--

SolinasPrime ::= SEQUENCE {
    a  INTEGER,
    b  INTEGER,
    c  INTEGER { positive(1), negative(-1) },
    s  INTEGER { positive(1), negative(-1) }
}

--
-- 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 1.
-- For type-1 curves, the curve is fixed, so Curve is set to NULL
-- For the BF prime p and subprime q, we have  $q * r = p + 1$ ,
-- and we encode the values of r and q in the public parameters.
-- The points P and P_pub are encoded as pointP and pointPpub
-- respectively.
--

BFPublicParameters ::= SEQUENCE {
    version      INTEGER { v1(1) },
    curve        Curve { NULL },
    r            INTEGER,
    q            SolinasPrime,
    pointP       FpPoint,
    pointPpub    FpPoint
}

--
-- A BF private key is a point on an elliptic curve,
-- which is an FpPoint.
--

BFPrivateKeyBlock ::= FpPoint
```

```
--
-- A BF master secret is an integer.
--

BFMasterSecret ::= INTEGER

--
-- BF ciphertext block
--

BFCiphertextBlock ::= SEQUENCE {
    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 1.
-- The embedding degree is currently fixed to 2.
-- For type-1 curves, curve is set to NULL.
-- For the BB1 prime p and subprime q, we have  $q * r = p + 1$ ,
-- and we encode the values of r and q in the public parameters.
--

BB1PublicParameters ::= SEQUENCE {
```

```
Version          INTEGER { v1(1) },
embedding-degree  INTEGER { degree-2(2) },
curve            Curve { NULL },
r               INTEGER,
q               SolinasPrime,
pointP          FpPoint,
pointP1         FpPoint,
pointP2         FpPoint,
pointP3         FpPoint
}

--
-- BB1 master secret block
--

BB1MasterSecret ::= SEQUENCE {
    alpha INTEGER,
    beta  INTEGER,
    gamma INTEGER
}

--
-- BB1 private Key block
--

BB1PrivateKeyBlock ::= SEQUENCE {
    pointD0 FpPoint,
    pointD1 FpPoint
}

--
-- BB1 ciphertext block
--
```

```
BB1CiphertextBlock ::= SEQUENCE {  
    pointChi0  FpPoint,  
    pointChi1  FpPoint,  
    nu         INTEGER,  
    y         OCTET STRING  
}  
END
```

12. Security considerations

This entire document discusses security considerations.

13. IANA considerations

All of the OIDs used in this document were assigned by the National Institute of Standards and Technology (NIST), so no further action by the IANA is necessary for this document.

14. 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.

15. References

15.1. Informative references

- [1] I. Blake, G. Seroussi, N. Smart, Elliptic Curves in Cryptography, Cambridge University Press, 1999.
- [2] D. Boneh, X. Boyen, Efficient selective-ID secure identity based encryption without random oracles, In Proc. of EUROCRYPT 04, LNCS 3027, pp. 223 238, 2004.
- [3] D. Boneh, M. Franklin, Identity-based encryption from the Weil pairing, In Proc. of CRYPTO 01, LNCS 2139, pp. 213 229, 2001.

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

Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in [BCP 78](#) and [BCP 79](#).

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this

specification can be obtained from the IETF on-line IPR repository at <http://www.ietf.org/ipr>.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Disclaimer of Validity

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Copyright Statement

Copyright (C) The Internet Society (2006).

This document is subject to the rights, licenses and restrictions contained in [BCP 78](#), and except as set forth therein, the authors retain all their rights.

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.

