Hashing to Elliptic Curves
draft-irtf-cfrg-hash-to-curve-03

Abstract

This document specifies a number of algorithms that may be used to encode or hash an arbitrary string to a point on an Elliptic Curve.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

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."

This Internet-Draft will expire on September 12, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
# Table of Contents

1. Introduction ............................................. 3  
1.1. Requirements ....................................... 3  
2. Background ............................................. 3  
  2.1. Terminology ...................................... 5  
    2.1.1. Encoding ................................... 5  
    2.1.2. Serialization ................................ 5  
    2.1.3. Random Oracle ................................. 6  
3. Algorithm Recommendations ............................... 6  
4. Utility Functions ........................................ 7  
5. Deterministic Encodings ................................ 8  
  5.1. Interface ......................................... 8  
  5.2. Notation .......................................... 8  
  5.3. Encodings for Weierstrass curves ..................... 9  
    5.3.1. Icart Method ................................. 9  
    5.3.2. Shallue-Woestijne-Ulas Method ................. 10  
    5.3.3. Simplified SWU Method ......................... 13  
    5.3.4. Boneh-Franklin Method ......................... 14  
    5.3.5. Fouque-Tibouchi Method ....................... 16  
  5.4. Encodings for Montgomery curves ..................... 19  
    5.4.1. Elligator2 Method ............................ 19  
6. Random Oracles ........................................... 22  
  6.1. Interface .......................................... 22  
7. Curve Transformations .................................... 22  
8. Ciphersuites ............................................ 22  
9. IANA Considerations ..................................... 24  
10. Security Considerations .................................. 25  
11. Acknowledgements ...................................... 25  
12. Contributors .......................................... 25  
13. Normative References ................................... 25  

Appendix A. Related Work .................................... 28  
  A.1. Probabilistic Encoding ................................ 28  
  A.2. Naive Encoding ................................... 29  
  A.3. Deterministic Encoding .............................. 29  
  A.4. Supersingular Curves ................................ 30  
  A.5. Twisted Variants ................................... 30  

Appendix B. Try-and-Increment Method .......................... 30  

Appendix C. Sample Code ..................................... 31  
  C.1. Icart Method ....................................... 31  
  C.2. Shallue-Woestijne-Ulas Method ....................... 32  
  C.3. Simplified SWU Method ................................ 34  
  C.4. Boneh-Franklin Method ................................ 34  
  C.5. Fouque-Tibouchi Method .............................. 35  
  C.6. Elligator2 Method .................................. 36  
  C.7. hash2base .......................................... 37  
  C.7.1. Considerations .................................. 38  

Appendix D. Test Vectors ..................................... 39
1. Introduction

Many cryptographic protocols require a procedure which maps arbitrary input, e.g., passwords, to points on an elliptic curve (EC). Prominent examples include Simple Password Exponential Key Exchange [Jablon96], Password Authenticated Key Exchange [BMP00], Identity-Based Encryption [BF01] and Boneh-Lynn-Shacham signatures [BLS01].

Unfortunately for implementors, the precise mapping which is suitable for a given scheme is not necessarily included in the description of the protocol. Compounding this problem is the need to pick a suitable curve for the specific protocol.

This document aims to address this lapse by providing a thorough set of recommendations across a range of implementations, and curve types. We provide implementation and performance details for each mechanism, along with references to the security rationale behind each recommendation and guidance for applications not yet covered.

Each algorithm conforms to a common interface, i.e., it maps a bitstring \((0, 1)^*\) to a point on an elliptic curve \(E\). For each variant, we describe the requirements for \(E\) to make it work. Sample code for each variant is presented in the appendix. Unless otherwise stated, all elliptic curve points are assumed to be represented as affine coordinates, i.e., \((x, y)\) points on a curve.

1.1. Requirements

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 [RFC2119].

2. Background

Here we give a brief definition of elliptic curves, with an emphasis on defining important parameters and their relation to encoding.
Let $F$ be the finite field $GF(p^k)$. We say that $F$ is a field of characteristic $p$. For most applications, $F$ is a prime field, in which case $k=1$ and we will simply write $GF(p)$.

Elliptic curves can be represented by equations of different standard forms, including, but not limited to: Weierstrass, Montgomery, and Edwards. Each of these variants correspond to a different category of curve equation. For example, the short Weierstrass equation is 
\[ y^2 = x^3 + Ax + B. \]
Certain encoding functions may have requirements on the curve form, the characteristic of the field, and the parameters, such as $A$ and $B$ in the previous example.

An elliptic curve $E$ is specified by its equation, and a finite field $F$. The curve $E$ forms a group, whose elements correspond to those who satisfy the curve equation, with values taken from the field $F$. As a group, $E$ has order $n$, which is the number of points on the curve. For security reasons, it is a strong requirement that all cryptographic operations take place in a prime order group. However, not all elliptic curves generate groups of prime order. In those cases, it is allowed to work with elliptic curves of order $n = qh$, where $q$ is a large prime, and $h$ is a short number known as the cofactor. Thus, we may wish an encoding that returns points on the subgroup of order $q$. Multiplying a point $P$ on $E$ by the cofactor $h$ guarantees that $hP$ is a point in the subgroup of order $q$.

Summary of quantities:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Relevance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>Order of finite field, $F = GF(p)$</td>
<td>Curve points need to be represented in terms of $p$. For prime power extension fields, we write $F = GF(p^k)$.</td>
</tr>
<tr>
<td>$n$</td>
<td>Number of curve points, $#E(F) = n$</td>
<td>For map to $E$, needs to produce $n$ elements.</td>
</tr>
<tr>
<td>$q$</td>
<td>Order of the largest prime subgroup of $E$, $n = qh$</td>
<td>If $n$ is not prime, may need mapping to $q$.</td>
</tr>
<tr>
<td>$h$</td>
<td>Cofactor</td>
<td>For mapping to subgroup, need to multiply by cofactor.</td>
</tr>
</tbody>
</table>
2.1. Terminology

In the following, we categorize the terminology for mapping bitstrings to points on elliptic curves.

2.1.1. Encoding

In practice, the input of a given cryptographic algorithm will be a bitstring of arbitrary length, denoted \(\{0, 1\}^*\). Hence, a concern for virtually all protocols involving elliptic curves is how to convert this input into a curve point. The general term "encoding" refers to the process of producing an elliptic curve point given as input a bitstring. In some protocols, the original message may also be recovered through a decoding procedure. An encoding may be deterministic or probabilistic, although the latter is problematic in potentially leaking plaintext information as a side-channel.

Suppose as the input to the encoding function we wish to use a fixed-length bitstring of length \(L\). Comparing sizes of the sets, \(2^L\) and \(n\), an encoding function cannot be both deterministic and bijective. We can instead use an injective encoding from \(\{0, 1\}^L\) to \(E\), with \(L < \log_2(n) - 1\), which is a bijection over a subset of points in \(E\). This ensures that encoded plaintext messages can be recovered.

In practice, encodings are commonly injective and invertible. Invertible encodings allow computation of input bitstrings given a point on the curve.

2.1.2. Serialization

A related issue is the conversion of an elliptic curve point to a bitstring. We refer to this process as "serialization", since it is typically used for compactly storing and transporting points, or for producing canonicalized outputs. Since a deserialization algorithm can often be used as a type of encoding algorithm, we also briefly document properties of these functions.

A straightforward serialization algorithm maps a point \((x, y)\) on \(E\) to a bitstring of length \(2\log(p)\), given that \(x, y\) are both elements in \(GF(p)\). However, since there are only \(n\) points in \(E\) (with \(n\) approximately equal to \(p\)), it is possible to serialize to a bitstring of length \(\log(n)\). For example, one common method is to store the \(x\)-coordinate and a single bit to determine whether the point is \((x, y)\) or \((x, -y)\), thus requiring \(\log(p)+1\) bits. This method reduces storage, but adds computation, since the deserialization process must recover the \(y\) coordinate.
2.1.3. Random Oracle

It is often the case that the output of the encoding function Section 2.1.1 should be (a) distributed uniformly at random on the elliptic curve and (b) non-invertible. That is, there is no discernible relation existing between outputs that can be computed based on the inputs. Moreover, given such an encoding function F from bitstrings to points on the curve, as well as a single point y, it is computationally intractable to produce an input x that maps to a y via F. In practice, these requirement stem from needing a random oracle which outputs elliptic curve points: one way to construct this is by first taking a regular random oracle, operating entirely on bitstrings, and applying a suitable encoding function to the output.

This motivates the term "hashing to the curve", since cryptographic hash functions are typically modeled as random oracles. However, this still leaves open the question of what constitutes a suitable encoding method, which is a primary concern of this document.

A random oracle onto an elliptic curve can also be instantiated using direct constructions, however these tend to rely on many group operations and are less efficient than hash and encode methods.

3. Algorithm Recommendations

In practice, two types of mappings are common: (1) Injective encodings, as can be used to construct a PRF as F(k, m) = k*H(m), and (2) Random Oracles, as required by PAKEs [BMP00], BLS [BLS01], and IBE [BF01]. (Some applications, such as IBE, have additional requirements, such as a Supersingular, pairing-friendly curve.)

The following table lists recommended algorithms for different curves and mappings. To select a suitable algorithm, choose the mapping associated with the target curve. For example, Elligator2 is the recommended injective encoding function for Curve25519, whereas Simple SWU is the recommended injective encoding for P-256. Similarly, the FFSTV Random Oracle construction described in Section 6 composed with Elligator2 should be used for Random Oracle mappings to Curve25519. When the required mapping is not clear, applications SHOULD use a Random Oracle.
### Utility Functions

Algorithms in this document make use of utility functions described below.

- **hash2base(x)**: This method is parametrized by p and H, where p is the prime order of the base field $\mathbb{F}_p$, and H is a cryptographic hash function which outputs at least $\left\lfloor \log_2(p) \right\rfloor + 1$ bits. The function first hashes x, converts the result to an integer, and reduces modulo p to give an element of $\mathbb{F}_p$. We provide a more detailed algorithm in Appendix C.7.

- **CMOV(a, b, c)**: If $c = 1$, return a, else return b.

  Common software implementations of constant-time selects assume $c = 1$ or $c = 0$. CMOV may be implemented by computing the desired selector (0 or 1) by ORing all bits of c together. The end result will be either 0 if all bits of c are zero, or 1 if at least one bit of c is 1.

- **CTEQ(a, b)**: Returns $a = b$. Inputs a and b must be the same length (as bytestrings) and the comparison must be implemented in constant time.

- **Legendre(x, p)**: $x^{(p-1)/2}$. The Legendre symbol computes whether the value x is a "quadratic residue" modulo p, and takes values 1, -1, 0, for when x is a residue, non-residue, or zero, respectively. Due to Euler’s criterion, this can be computed in constant time, with respect to a fixed p, using the equation $x^{(p-1)/2}$. For clarity, we will generally prefer using the formula directly, and annotate the usage with this definition.

---

<table>
<thead>
<tr>
<th>Curve</th>
<th>Inj. Encoding</th>
<th>Random Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-256</td>
<td>Simple SWU Section 5.3.3</td>
<td>FFSTV(SWU) Section 6</td>
</tr>
<tr>
<td>P-384</td>
<td>Icart Section 5.3.1</td>
<td>FFSTV(Icart) Section 6</td>
</tr>
<tr>
<td>Curve25519</td>
<td>Elligator2 Section 5.4.1</td>
<td>FFSTV(Elligator2) Section 6</td>
</tr>
<tr>
<td>Curve448</td>
<td>Elligator2 Section 5.4.1</td>
<td>FFSTV(Elligator2) Section 6</td>
</tr>
</tbody>
</table>
Internet-Draft                hash-to-curve                   March 2019

o  \( \sqrt{x, p} \): Computing square roots should be done in constant time
   where possible.

When \( p = 3 \mod 4 \), the square root can be computed as "\( \sqrt{x, p} := x^{(p+1)/4} \)". This applies to P256, P384, and Curve448.

When \( p = 5 \mod 8 \), the square root can be computed by the following algorithm, in which "\( \sqrt{-1} \)" is a field element and can be precomputed. This applies to Curve25519.

\[
\begin{align*}
\sqrt{x, p} := & \quad x^{(p+3)/8} \quad \text{if } x^{(p+3)/4} = x \\
& \sqrt{-1} \times x^{(p+3)/8} \quad \text{otherwise}
\end{align*}
\]

The above two conditions hold for most practically used curves, due to the simplicity of the square root function. For others, a suitable constant-time Tonelli-Shanks variant should be used as in [Schoof85].

5. Deterministic Encodings

5.1. Interface

The generic interface for deterministic encoding functions to elliptic curves is as follows:

\[
\text{map2curve}(\alpha)
\]

where \( \alpha \) is a message to encode on a curve.

5.2. Notation

As a rough style guide for the following, we use \((x, y)\) to be the output coordinates of the encoding method. Indexed values are used when the algorithm will choose between candidate values. For example, the SWU algorithm computes three candidates \((x_1, y_1), (x_2, y_2), (x_3, y_3)\), from which the final \((x, y)\) output is chosen via constant time comparison operations.

We use \( u, v \) to denote the values in \( \mathbb{F}_p \) output from \( \text{hash2base} \), and use as initial values in the encoding.

We use \( t_1, t_2, \ldots \), as reusable temporary variables. For notable variables, we will use a distinct name, for ease of debugging purposes when correlating with test vectors.

The code presented here corresponds to the example Sage [SAGE] code found at [github-repo]. Which is additionally used to generate
intermediate test vectors. The Sage code is also checked against the hacsproj implementation.

Note that each encoding requires that certain preconditions must hold in order to be applied.

5.3. Encodings for Weierstrass curves

The following encodings apply to elliptic curves defined as E: y^2 = x^3+Ax+B, where 4A^3+27B^2 ≠ 0.

5.3.1. Icart Method

The map2curve_icart(alpha) implements the Icart encoding method from [Icart09].

*Preconditions*

A Weierstrass curve over F_{p^n}, where p>3 and p^n = 2 mod 3 (or p = 2 mod 3 and for odd n).

*Examples*

- P-384

*Algorithm*: map2curve_icart

Input:

- alpha: an octet string to be hashed.
- A, B: the constants from the Weierstrass curve.

Output:

- (x, y), a point in E.

Operations:

u = hash2base(alpha)

v = ((3A - u^4) / 6u)

x = (v^2 - B - (u^6 / 27))^(1/3) + (u^2 / 3)

y = ux + v

Output (x, y)

*Implementation*
The following procedure implements Icart’s algorithm in a straight-line fashion.

map2curve_icart(alpha)

Input:

alpha - value to be hashed, an octet string

Output:

(x, y) - a point in E

Precomputations:

1. $c_1 = (2 * p) - 1$
2. $c_1 = c_1 / 3$ \hspace{1cm} // $c_1 = (2p-1)/3$ as integer
3. $c_2 = 3^{(-1)}$ \hspace{1cm} // $c_2 = 1/3 \pmod{p}$
4. $c_3 = c_2^3$ \hspace{1cm} // $c_3 = 1/27 \pmod{p}$

Steps:

1. $u = \text{hash2base}(alpha)$ \hspace{1cm} // $\{0,1\}^* \rightarrow \mathbb{F}_p$
2. $u^2 = u^2$ \hspace{1cm} // $u^2$
3. $u^4 = u^2 \cdot u^2$ \hspace{1cm} // $u^4$
4. $v = 3 \cdot A$ \hspace{1cm} // $3A$ in $\mathbb{F}_p$
5. $v = v - u^4$ \hspace{1cm} // $3A - u^4$
6. $t_1 = 6 \cdot u$ \hspace{1cm} // $6u$
7. $t_1 = t_1^((-1))$ \hspace{1cm} // modular inverse
8. $v = v \cdot t_1$ \hspace{1cm} // $(3A - u^4)/(6u)$
9. $x_1 = v^2$ \hspace{1cm} // $v^2$
10. $x_1 = x - B$ \hspace{1cm} // $v^2 - B$
11. $u^6 = u^4 \cdot c_3$ \hspace{1cm} // $u^4 / 27$
12. $u^6 = u^6 \cdot u^2$ \hspace{1cm} // $u^6 / 27$
13. $x_1 = x_1 - u^6$ \hspace{1cm} // $v^2 - B - u^6/27$
14. $x_1 = x^c_1$ \hspace{1cm} // $(v^2 - B - u^6/27)^{(1/3)}$
15. $t_1 = u^2 \cdot c_2$ \hspace{1cm} // $u^2 / 3$
16. $x = x + t_1$ \hspace{1cm} // $(v^2 - B - u^6/27)^{(1/3)} + (u^2 / 3)$
17. $y = u \cdot x$ \hspace{1cm} // $ux$
18. $y = y + v$ \hspace{1cm} // $ux + v$
19. Output $(x, y)$

5.3.2. Shallue-Woestijne-Ulas Method

The map2curve_swu(alpha) implements the Shallue-Woestijne-Ulas (SWU) method by Ulas [SWU07], which is based on Shallue and Woestijne [SW06] method.
*Preconditions*

This algorithm works for any Weierstrass curve over $\mathbb{F}_{p^n}$ such that $A \neq 0$ and $B \neq 0$.

*Examples*

- P-256
- P-384
- P-521

*Algorithm*: map2curve_swu

Input:

- $\alpha$: an octet string to be hashed.
- $A, B$: the constants from the Weierstrass curve.

Output:

- $(x, y)$, a point in $E$.

Operations:

1. $u = \text{hash2base}(\alpha || 0x00)$
2. $v = \text{hash2base}(\alpha || 0x01)$
3. $x_1 = v$
4. $x_2 = (-B / A)(1 + 1 / (u^4 * g(v)^2 + u^2 * g(v)))$
5. $x_3 = u^2 * g(v)^2 * g(x_2)$
6. If $g(x_1)$ is square, output $(x_1, \sqrt{g(x_1)})$
7. If $g(x_2)$ is square, output $(x_2, \sqrt{g(x_2)})$
8. Output $(x_3, \sqrt{g(x_3)})$

The algorithm relies on the following equality:

$$u^3 * g(v)^2 * g(x_2) = g(x_1) * g(x_2) * g(x_3)$$

The algorithm computes three candidate points, constructed such that at least one of them lies on the curve.

*Implementation*

The following procedure implements SWU’s algorithm in a straight-line fashion.
map2curve_swu(alpha)

Input:
  alpha - value to be hashed, an octet string

Output:
  (x, y) - a point in E

Precomputations:
1. $c_1 = -B / A \mod p$  // Field arithmetic
2. $c_2 = (p - 1)/2$  // Integer arithmetic

Steps:
1. $u = \text{hash2base}(\alpha || 0x00)$  // $(0,1)^* \to \mathbb{F}_p$
2. $v = \text{hash2base}(\alpha || 0x01)$  // $(0,1)^* \to \mathbb{F}_p$
3. $x_1 = v$  // $x_1 = v$
4. $g_v = v^3$
5. $g_v = g_v + (A * v)$  // $g_v = g(v)$
6. $g_{x_1} = g_v$  // $g_{x_1} = g(x_1)$
7. $u_2 = u^2$
8. $t_1 = u_2 * g_v$  // $t_1 = u^2 * g(v)$
9. $t_2 = t_1^2$
10. $t_2 = t_2 + t_1$  // $t_2 = 1/(u^4g(v)^2 + u^2g(v))$
11. $n_1 = 1 + t_2$
12. $x_2 = c_1 * n_1$  // $x_2 = -B/A * (1 + 1/(t_1^2 + t_1))$
13. $g_{x_2} = x_2^3$
14. $t_2 = A * x_2$
15. $g_{x_2} = g_{x_2} + t_2$  // $g_{x_2} = g(x_2)$
16. $t_2 = t_2^2$
17. $g_{x_3} = g_{x_2} + A * x_3$  // $g_{x_3} = g(x_3)$
18. $g_{x_3} = g_{x_3} + B$  // Legendre($g_{x_3}$)
19. $l_1 = g_{x_1}^c_2$  // Legendre($g_{x_1}$)
20. $l_2 = g_{x_2}^c_2$  // Legendre($g_{x_2}$)
21. $x = \text{CMOV}(x_2, x_3, l_2)$  // If $l_2 = 1$, choose $x_2$, else choose $x_3$
22. $x = \text{CMOV}(x_1, x, l_1)$  // If $l_1 = 1$, choose $x_1$, else choose $x$
23. $g_x = \text{CMOV}(g_{x_2}, g_{x_3}, l_2)$  // If $l_2 = 1$, choose $g_{x_2}$, else choose $g_{x_3}$
24. $g_x = \text{CMOV}(g_{x_1}, g_x, l_1)$  // If $l_1 = 1$, choose $g_{x_1}$, else choose $g_x$
25. $y = \sqrt{g_x}$
26. Output $(x, y)$
5.3.3. Simplified SWU Method

The map2curve_simple_swu(alpha) implements a simplified version of Shallue-Woestijne-Ulas algorithm given by Brier et al. [SimpleSWU].

*Preconditions*

This algorithm works for any Weierstrass curve over F_{p^n} such that A≠0, B≠0, and p=3 mod 4.

*Examples*
- P-256
- P-384
- P-521

*Algorithm*: map2curve_simple_swu

Input:
- alpha: an octet string to be hashed.
- A, B: the constants from the Weierstrass curve.

Output:
- (x,y), a point in E.

Operations:
1. Define \( g(x) = x^3 + Ax + B \)
2. \( u = \text{hash2base}(\text{alpha}) \)
3. \( x1 = -B/A \times (1 + (1 / (u^4 - u^2))) \)
4. \( x2 = -u^2 \times x1 \)
5. If \( g(x1) \) is square, output \((x1, \sqrt{g(x1)})\)
6. Output \((x2, \sqrt{g(x2)})\)

*Implementation*

The following procedure implements the Simple SWU’s algorithm in a straight-line fashion.
map2curve_simple_swu(alpha)

Input:

alpha - value to be encoded, an octet string

Output:

(x, y) - a point in E

Precomputations:

1.  c1 = -B / A mod p           // Field arithmetic
2.  c2 = (p - 1)/2              // Integer arithmetic

Steps:

1.    u = hash2base(alpha)  // {0,1}^* -> Fp
2.   u2 = u^2
3.   u2 = -u2                // u2 = -u^2
4.   u4 = u2^2
5.   t1 = u4 + u2
6.   t1 = t1^(-1)
7.   n1 = 1 + t2             // n1 = 1 + (1 / (u^4 - u^2))
8.   x1 = c1 * n1            // x1 = -B/A * (1 + (1 / (u^4 - u^2)))
9.  gx1 = x1 ^ 3
10.  t1 = A * x1
11.  gx1 = gx1 + t1
12.  gx1 = gx1 + B           // gx1 = x1^3 + Ax1 + B = g(x1)
13.  x2 = u2 * x1            // x2 = -u^2 * x1
14.  gx2 = x2^3
15.  t1 = A * x2
16.  gx2 = gx2 + B           // gx2 = x2^3 + Ax2 + B = g(x2)
17.  e = gx1*c2
18.  x = CMOV(x1, x2, l1)     // If l1 = 1, choose x1, else choose x2
19.  gx = CMOV(gx1, gx2, l1)  // If l1 = 1, choose gx1, else choose gx2
20.   y = sqrt(gx)
21. Output (x, y)

5.3.4. Boneh-Franklin Method

The map2curve_bf(alpha) implements the Boneh-Franklin method [BF01] which covers the case of supersingular curves "E: y^2=x^3+B". This method does not guarantee that the resulting a point be in a specific subgroup of the curve. To do that, a scalar multiplication by a cofactor is required.
*Preconditions*

This algorithm works for any Weierstrass curve over "F_q" such that "A=0" and "q=2 mod 3".

*Examples*

- "y^2 = x^3 + 1"

*Algorithm*: map2curve_bf

Input:

- "alpha": an octet string to be hashed.
- "B": the constant from the Weierstrass curve.

Output:

- "(x, y)": a point in E.

Operations:

1. u = hash2base(alpha)
2. x = (u^2 - B)^((2 * q - 1) / 3)
3. Output (x, u)

*Implementation*

The following procedure implements the Boneh-Franklin’s algorithm in a straight-line fashion.
map2curve_bf(alpha)

Input:

alpha: an octet string to be hashed.
B    : the constant from the Weierstrass curve.

Output:

(x, y): a point in E

Precomputations:

1.  c = (2 * q - 1) / 3    // Integer arithmetic

Steps:

1.  u = hash2base(alpha)  // {0,1}^* -> F_q
2.  t0 = u^2                // t0 = u^2
3.  t1 = t0 - B             // t1 = u^2 - B
4.  x = t1^c               // x  = (u^2 - B)^((2 * q - 1) / 3)
5. Output (x, u)

5.3.5. Fouque-Tibouchi Method

The map2curve_ft(alpha) implements the Fouque-Tibouchi’s method [FT12] (Sec. 3, Def. 2) which covers the case of pairing-friendly curves "E : y^2 = x^3 + B". Note that for pairing curves the destination group is usually a subgroup of the curve, hence, a scalar multiplication by the cofactor will be required to send the point to the desired subgroup.

*Preconditions*

This algorithm works for any Weierstrass curve over "F_q" such that "q≡7 mod 12", "A=0", and "1+B" is a non-zero square in the field. This covers the case "q≡1 mod 3" not handled by Boneh-Franklin’s method.

*Examples*

- SECP256K1 curve [SEC2]
- BN curves [BN05]
- KSS curves [KSS08]
- BLS curves [BLS01]
*Algorithm*: map2curve_ft

Input:

- "alpha": an octet string to be hashed.
- "B": the constant from the Weierstrass curve.
- "s": a constant equal to $\sqrt{-3}$ in the field.

Output:

- (x, y): a point in $E$.

Operations:

1. $t = \text{hash2base}(\alpha)$
2. $w = \frac{s \cdot t}{1 + B + t^2}$
3. $x_1 = \frac{(-1 + s)}{2} - t \cdot w$
4. $x_2 = -1 - x_1$
5. $x_3 = 1 + \frac{1}{w^2}$
6. $e = \text{Legendre}(t)$
7. If $x_1^3 + B$ is square, output $(x_1, e \cdot \sqrt{x_1^3 + B})$
8. If $x_2^3 + B$ is square, output $(x_2, e \cdot \sqrt{x_2^3 + B})$
9. Output $(x_3, e \cdot \sqrt{x_3^3 + B})$

*Implementation*

The following procedure implements the Fouque-Tibouchi’s algorithm in a straight-line fashion.
map2curve_ft(alpha)

Input:

- alpha: an octet string to be encoded
- B: the constant of the curve

Output:

- \((x, y)\): - a point in \(E\)

Precomputations:

1. \(c_1 = \sqrt{-3}\) // Field arithmetic
2. \(c_2 = (-1 + c_1) / 2\) // Field arithmetic

Steps:

1. \(t = \text{hash2base}(\alpha)\) // \((0,1)^* \rightarrow \mathbb{F}_p\)
2. \(k = t^2\) // \(t^2\)
3. \(k = k + B + 1\) // \(t^2 + B + 1\)
4. \(k = 1 / k\) // \(1 / (t^2 + B + 1)\)
5. \(k = k * t\) // \(t / (t^2 + B + 1)\)
6. \(k = k * c_1\) // \(\sqrt{-3} * t / (t^2 + B + 1)\)
7. \(x_1 = c_2 - t * k\) // \((-1 + \sqrt{-3}) / 2 - \sqrt{-3} * t^2 / (t^2 + B + 1)\)
8. \(x_2 = -1 - x_1\)
9. \(r = k^2\)
10. \(r = 1 / r\)
11. \(x_3 = 1 + r\)
12. \(fx_1 = x_1^3 + B\)
13. \(fx_2 = x_2^3 + B\)
14. \(s_1 = \text{Legendre}(fx_1)\)
15. \(s_2 = \text{Legendre}(fx_2)\)
16. \(x = x_3\)
17. \(x = \text{CMOV}(x_2, x, s_2 > 0)\) // if \(s_2=1\), then \(x\) is set to \(x_2\)
18. \(t_2 = \text{Legendre}(t)\)
19. \(y = t_2 * \sqrt{y}\) // TODO: determine which root to choose
20. Output \((x, y)\)

Additionally, "map2curve_ft(alpha)" can return the point "\((c_2, \sqrt{1 + B})\)" when "u=0".
5.4. Encodings for Montgomery curves

A Montgomery curve is given by the following equation E:
By^2=x^3+Ax^2+x, where B(A^2 - 4) ≠ 8800; 0. Note that any curve
with a point of order 2 is isomorphic to this representation. Also
notice that E cannot have a prime order group, hence, a scalar
multiplication by the cofactor is required to obtain a point in the
main subgroup.

5.4.1. Elligator2 Method

The map2curve_elligator2(alpha) implements the Elligator2 method from
[Elligator2].

*Preconditions*

Any curve of the form y^2=x^3+Ax^2+Bx, which covers all Montgomery
curves such that A ≠ 0 and B=1 (i.e. j-invariant != 1728).

*Examples*

  o Curve25519
  o Curve448

*Algorithm*: map2curve_elligator2

Input:

  o alpha: an octet string to be hashed.
  o A,B=1: the constants of the Montgomery curve.
  o N: a constant non-square in the field.

Output:

  o (x,y), a point in E.

Operations:
1. Define \( g(x) = x(x^2 + Ax + B) \)
2. \( u = \text{hash2base}(\alpha) \)
3. \( v = -A/(1 + N*u^2) \)
4. \( e = \text{Legendre}(g(v)) \)
5.1. If \( u \neq 0 \), then
5.2. \( x = ev - (1 - e)A/2 \)
5.3. \( y = -e*\sqrt{g(x)} \)
5.4. Else, \( x=0 \) and \( y=0 \)
6. Output \((x,y)\)

Here, \( e \) is the Legendre symbol defined as in Section 4.

*Implementation*

The following procedure implements elligator2 algorithm in a straight-line fashion.
map2curve_elligator2(alpha)

Input:

alpha - value to be encoded, an octet string
A,B=1 - the constants of the Montgomery curve.
N - a constant non-square value in Fp.

Output:

(x, y) - a point in E

Precomputations:

1. c1 = (p - 1)/2    // Integer arithmetic
2. c2 = A / 2 (mod p) // Field arithmetic

Steps:

1. u = hash2base(alpha)
2. t1 = u^2
3. t1 = N * t1
4. t1 = 1 + t1
5. t1 = t1^(-1)
6. v = A * t1
7. v = -v        // v = -A / (1 + N * u^2)
8. gv = v + A
9. gv = gv * v
0. gv = gv + B
11. gv = gv * v   // gv = v^3 + Av^2 + Bv
12. e = gv^c1    // Legendre(gv)
13. x = e*v
14. ne = -e
15. t1 = 1 + ne
16. t1 = t1 * c2
17. x = x - t1   // x = ev - (1 - e)*A/2
18. y = x + A
19. y = y * x
20. y = y + B
21. y = y * x
22. y = sqrt(y)  // y = -e * sqrt(x^3 + Ax^2 + Bx)
23. y = y * ne    // y = -e * sqrt(x^3 + Ax^2 + Bx)
24. x = CMOV(0, x, 1-u)
25. y = CMOV(0, y, 1-u)
26. Output (x, y)

Elligator2 can be simplified with projective coordinates.
6. Random Oracles

Some applications require a Random Oracle (RO) of points, which can be constructed from deterministic encoding functions. Farashahi et al. [FFSTV13] showed a generic mapping construction that is indistinguishable from a random oracle. In particular, let \( f : (0,1)^* \rightarrow E(F) \) be a deterministic encoding function, and let \( H_0 \) and \( H_1 \) be two hash functions modeled as random oracles that map bit strings to elements in the field \( F \), i.e., \( H_0, H_1 : (0,1)^* \rightarrow F \). Then, the "hash2curveRO(\( \alpha \))" mapping is defined as

\[
\text{hash2curveRO}(\alpha) = f(H_0(\alpha)) + f(H_1(\alpha))
\]

where \( \alpha \) is an octet string to be encoded as a point on a curve.

6.1. Interface

Using the deterministic encodings from Section 5, the "hash2curveRO(\( \alpha \))" mapping can be instantiated as

\[
\text{hash2curveRO}(\alpha) = \text{hash2curve}(\alpha || 0x02) + \text{hash2curve}(\alpha || 0x03)
\]

where the addition operation is performed as a point addition.

7. Curve Transformations

Every elliptic curve can be converted to an equivalent curve in short Weierstrass form ([BL07] Theorem 2.1), making SWU a generic algorithm that can be used for all curves. Curves in either Edwards or Twisted Edwards form can be transformed into equivalent curves in Montgomery form [BL17] for use with Elligator2. [RFC7748] describes how to convert between points on Curve25519 and Ed25519, and between Curve448 and its Edwards equivalent, Goldilocks.

8. Ciphersuites

To provide concrete recommendations for algorithms we define a hash-to-curve "ciphersuite" as a four-tuple containing:

- Destination Group (e.g. P256 or Curve25519)
- hash2base algorithm
- HashToCurve algorithm (e.g. SSWU, Icart)
- (Optional) Transformation (e.g. FFSTV, cofactor clearing)
A ciphersuite defines an algorithm that takes an arbitrary octet string and returns an element of the Destination Group defined in the ciphersuite by applying HashToCurve and Transformation (if defined).

This document describes the following set of ciphersuites:

- H2C-P256-SHA256-SSWU-
- H2C-P384-SHA512-Icart-
- H2C-SECP256K1-SHA512-FT-
- H2C-BN256-SHA512-FT-
- H2C-Curve25519-SHA512-Elgator2-Clear
- H2C-Curve448-SHA512-Elgator2-Clear
- H2C-Curve25519-SHA512-Elgator2-FFSTV
- H2C-Curve448-SHA512-Elgator2-FFSTV

H2C-P256-SHA256-SSWU- is defined as follows:

- The destination group is the set of points on the NIST P-256 elliptic curve, with curve parameters as specified in [DSS] (Section D.1.2.3) and [RFC5114] (Section 2.6).

- hash2base is defined as {#hashtobase} with the hash function defined as SHA-256 as specified in [RFC6234], and p set to the prime field used in P-256 ($2^{256} - 2^{224} + 2^{192} + 2^96 - 1$).

- HashToCurve is defined to be {#sswu} with A and B taken from the definition of P-256 (A=-3, B=4105836372515214212932612978004726840 911441015993725554835256314039467401291).

H2C-P384-SHA512-Icart- is defined as follows:

- The destination group is the set of points on the NIST P-384 elliptic curve, with curve parameters as specified in [DSS] (Section D.1.2.4) and [RFC5114] (Section 2.7).

- hash2base is defined as {#hashtobase} with the hash function defined as SHA-512 as specified in [RFC6234], and p set to the prime field used in P-384 ($2^{384} - 2^{128} - 2^96 + 2^{32} - 1$).

- HashToCurve is defined to be {#icart} with A and B taken from the definition of P-384 (A=-3, B=2758019355995970587784901184038904809).
H2C-Curve25519-SHA512-Eligator2-Clear is defined as follows:

- The destination group is the points on Curve25519, with curve parameters as specified in [RFC7748] (Section 4.1).
- hash2base is defined as (#hashtobase) with the hash function defined as SHA-512 as specified in [RFC6234], and p set to the prime field used in Curve25519 (2^255 - 19).
- HashToCurve is defined to be (#elligator2) with the curve function defined to be the Montgomery form of Curve25519 (y^2 = x^3 + 486662x^2 + x) and N = 2.
- The final output is multiplied by the cofactor of Curve25519, 8.

H2C-Curve448-SHA512-Eligator2-Clear is defined as follows:

- The destination group is the points on Curve448, with curve parameters as specified in [RFC7748] (Section 4.1).
- hash2base is defined as (#hashtobase) with the hash function defined as SHA-512 as specified in [RFC6234], and p set to the prime field used in Curve448 (2^448 - 2^224 - 1).
- HashToCurve is defined to be (#elligator2) with the curve function defined to be the Montgomery form of Curve448 (y^2 = x^3 + 156326x^2 + x) and N = -1.
- The final output is multiplied by the cofactor of Curve448, 4.

H2C-Curve25519-SHA512-Eligator2-FFSTV is defined as in H2C-Curve25519-SHA-512-Eligator2-Clear except HashToCurve is defined to be (#ffstv) where F is (#elligator2).

H2C-Curve448-SHA512-Eligator2-FFSTV is defined as in H2C-Curve448-SHA-512-Eligator2-Clear except HashToCurve is defined to be (#ffstv) where F is (#elligator2).

9. IANA Considerations

This document has no IANA actions.
10. Security Considerations

Each encoding function variant accepts arbitrary input and maps it to a pseudorandom point on the curve. Points are close to indistinguishable from randomly chosen elements on the curve. Not all encoding functions are full-domain hashes. Elligator2, for example, only maps strings to "about half of all curve points," whereas Icart's method only covers about 5/8 of the points.

11. Acknowledgements

The authors would like to thank Adam Langley for this detailed writeup up Elligator2 with Curve25519 [ElligatorAGL]. We also thank Sean Devlin and Thomas Icart for feedback on earlier versions of this document.

12. Contributors

- Armando Faz
  Cloudflare
  armfazh@cloudflare.com

- Sharon Goldberg
  Boston University
  goldbe@cs.bu.edu

- Ela Lee
  Royal Holloway, University of London
  Ela.Lee.2010@live.rhul.ac.uk

13. Normative References


[ECOPRF] "EC-OPRF - Oblivious Pseudorandom Functions using Elliptic Curves", n.d..


[FFSTV13] "Indifferentiable deterministic hashing to elliptic and hyperelliptic curves", n.d..


[Jablon96] "Strong password-only authenticated key exchange", n.d.


Appendix A. Related Work

In this chapter, we give a background to some common methods to encode or hash to the curve, motivated by the similar exposition in [Icart09]. Understanding of this material is not required in order to choose a suitable encoding function - we defer this to Section 3 - the background covered here can work as a template for analyzing encoding functions not found in this document, and as a guide for further research into the topics covered.

A.1. Probabilistic Encoding

As mentioned in Section 2, as a rule of thumb, for every \( x \) in \( \text{GF}(p) \), there is approximately a 1/2 chance that there exist a corresponding \( y \) value such that \((x, y)\) is on the curve \( E \).

This motivates the construction of the MapToGroup method described by Boneh et al. [BLS01]. For an input message \( m \), a counter \( i \), and a standard hash function \( H: \{0, 1\}^* \rightarrow \text{GF}(p) \times \{0, 1\} \), one computes \((x, b) = H(i || m)\), where \( i || m \) denotes concatenation of the two values. Next, test to see whether there exists a corresponding \( y \) value such that \((x, y)\) is on the curve, returning \((x, y)\) if successful, where \( b \) determines whether to take +/- \( y \). If there does not exist such a \( y \), then increment \( i \) and repeat. A maximum counter
value is set to I, and since each iteration succeeds with probability approximately 1/2, this process fails with probability $2^{-I}$. (See Appendix B for a more detailed description of this algorithm.)

Although MapToGroup describes a method to hash to the curve, it can also be adapted to a simple encoding mechanism. For a bitstring of length strictly less than $\log_2(p)$, one can make use of the spare bits in order to encode the counter value. Allocating more space for the counter increases the expansion, but reduces the failure probability.

Since the running time of the MapToGroup algorithm depends on $m$, this algorithm is NOT safe for cases sensitive to timing side channel attacks. Deterministic algorithms are needed in such cases where failures are undesirable.

A.2. Naive Encoding

A naive solution includes computing $H(m) \cdot G$ as map2curve($m$), where $H$ is a standard hash function $H : \{0, 1\}^* \rightarrow GF(p)$, and $G$ is a generator of the curve. Although efficient, this solution is unsuitable for constructing a random oracle onto $E$, since the discrete logarithm with respect to $G$ is known. For example, given $y_1 = map2curve(m_1)$ and $y_2 = map2curve(m_2)$ for any $m_1$ and $m_2$, it must be true that $y_2 = H(m_2) / H(m_1) \cdot map2curve(m_1)$. This relationship would not hold (with overwhelming probability) for truly random values $y_1$ and $y_2$. This causes catastrophic failure in many cases. However, one exception is found in SPEKE [Jablon96], which constructs a base for a Diffie-Hellman key exchange by hashing the password to a curve point. Notably the use of a hash function is purely for encoding an arbitrary length string to a curve point, and does not need to be a random oracle.

A.3. Deterministic Encoding

Shallue, Woestijne, and Ulas [SW06] first introduced a deterministic algorithm that maps elements in $F_{q}(q)$ to a curve in time $O(\log^4 q)$, where $q = p^n$ for some prime $p$, and time $O(\log^3 q)$ when $q = 3 \mod 4$. Icart introduced yet another deterministic algorithm which maps $F_{q}(q)$ to any EC where $q = 2 \mod 3$ in time $O(\log^3 q)$ [Icart09]. Elligator (2) [Elligator2] is yet another deterministic algorithm for any odd-characteristic EC that has a point of order 2. Elligator2 can be applied to Curve25519 and Curve448, which are both CFRG-recommended curves [RFC7748].

However, an important caveat to all of the above deterministic encoding functions, is that none of them map injectively to the entire curve, but rather some fraction of the points. This makes

them unable to use to directly construct a random oracle on the curve.

Brier et al. [SimpleSWU] proposed a couple of solutions to this problem. The first applies solely to Icart’s method described above, by computing \( F(H_0(m)) + F(H_1(m)) \) for two distinct hash functions \( H_0, H_1 \). The second uses a generator \( G \), and computes \( F(H_0(m)) + H_1(m)\cdot G \). Later, Farashahi et al. [FFSTV13] showed the generality of the \( F(H_0(m)) + F(H_1(m)) \) method, as well as the applicability to hyperelliptic curves (not covered here).

A.4. Supersingular Curves

For supersingular curves, for every \( y \) in \( GF(p) \) (with \( p>3 \)), there exists a value \( x \) such that \((x, y)\) is on the curve \( E \). Hence we can construct a bijection \( F : GF(p) \to E \) (ignoring the point at infinity). This is the case for [BF01], but is not common.

A.5. Twisted Variants

We can also consider curves which have twisted variants, \( E^d \). For such curves, for any \( x \) in \( GF(p) \), there exists \( y \) in \( GF(p) \) such that \((x, y)\) is either a point on \( E \) or \( E^d \). Hence one can construct a bijection \( F : GF(p) \times \{0, 1\} \to E \cup E^d \), where the extra bit is needed to choose the sign of the point. This can be particularly useful for constructions which only need the \( x \)-coordinate of the point. For example, \( x \)-only scalar multiplication can be computed on Montgomery curves. In this case, there is no need for an encoding function, since the output of \( F \) in \( GF(p) \) is sufficient to define a point on one of \( E \) or \( E^d \).

Appendix B. Try-and-Increment Method

In cases where constant time execution is not required, the so-called try-and-increment method may be appropriate. As discussion in Section 1, this variant works by hashing input \( m \) using a standard hash function ("Hash"), e.g., SHA256, and then checking to see if the resulting point \((m, f(m))\), for curve function \( f \), belongs on \( E \). This is detailed below.
1. ctr = 0
2. h = "INVALID"
3. While h is "INVALID" or h is EC point at infinity:
   4.1   CTR = I2OSP(ctr, 4)
   4.2   ctr = ctr + 1
   4.3   attempted_hash = Hash(m || CTR)
   4.4   h = RS2ECP(attempted_hash)
   4.5   If h is not "INVALID" and cofactor > 1, set h = h * cofactor
5. Output h

I2OSP is a function that converts a nonnegative integer to octet string as defined in Section 4.1 of [RFC8017], and RS2ECP(h) = OS2ECP(0x02 || h), where OS2ECP is specified in Section 2.3.4 of [SECG1], which converts an input string into an EC point.

Appendix C. Sample Code

This section contains reference implementations for each map2curve variant built using [hacspec].

C.1. Icart Method

The following hacspec program implements map2curve_icart(alpha) for P-384.

```python
from hacspec.speclib import *

prime = 2**384 - 2**128 - 2**96 + 2**32 - 1
felem_t = refine(nat, lambda x: x < prime)
affine_t = tuple2(felem_t, felem_t)

@typechecked
def to_felem(x: nat_t) -> felem_t:
    return felem_t(nat(x % prime))

@typechecked
def fadd(x: felem_t, y: felem_t) -> felem_t:
    return to_felem(x + y)

@typechecked
def fsub(x: felem_t, y: felem_t) -> felem_t:
    return to_felem(x - y)
```

def fmul(x: felem_t, y: felem_t) -> felem_t:
    return to_felem(x * y)

@typechecked
def fsqr(x: felem_t) -> felem_t:
    return to_felem(x * x)

@typechecked
def fexp(x: felem_t, n: nat_t) -> felem_t:
    return to_felem(pow(x, n, prime))

@typechecked
def finv(x: felem_t) -> felem_t:
    return to_felem(pow(x, prime-2, prime))
a384 = to_felem(prime - 3)
b384 = to_felem(2758019355995970587784901184038904809305690585636156852142870730
1988699241309860865136260764883745107765439761230575)

@typechecked
def map2p384(u:felem_t) -> affine_t:
    v = fmul(fsub(fmul(to_felem(3), a384), fexp(u, 4)), finv(fmul(to_felem(6), u )))
    u2 = fmul(fexp(u, 6), finv(to_felem(27)))
    x = fsub(fsqr(v), b384)
    x = fsub(x, u2)
    x = fexp(x, (2 * prime - 1) // 3)
    x = fadd(x, fmul(fsqr(u), finv(to_felem(3))))
    y = fadd(fmul(u, x), v)
    return (x, y)

C.2. Shallue-Weiostijne-Ulas Method

The following hacspec program implements map2curve_swu(alpha) for P-256.
from p256 import *
from hacspec.speclib import *

a256 = to_felem(prime - 3)
b256 = to_felem(410583637251521421293261297800472684091144101599372555483525631
4039467401291)

@typechecked
def f_p256(x:felem_t) -> felem_t:
    return fadd(fexp(x, 3), fadd(fmul(to_felem(a256), x), to_felem(b256)))

@typechecked
def x1(t:felem_t, u:felem_t) -> felem_t:
    return u

@typechecked
def x2(t:felem_t, u:felem_t) -> felem_t:
    coefficient = fmul(to_felem(-b256), finv(to_felem(a256)))
t2 = fsqr(t)
t4 = fsqr(t2)
gu = f_p256(u)
gu2 = fsqr(gu)
denom = fadd(fmul(t4, gu2), fmul(t2, gu))
return fmul(coefficient, fadd(to_felem(1), finv(denom)))

@typechecked
def x3(t:felem_t, u:felem_t) -> felem_t:
    return fmul(fsqr(t), fmul(f_p256(u), x2(t, u)))

@typechecked
def map2p256(t:felem_t) -> felem_t:
    u = fadd(t, to_felem(1))
x1v = x1(t, u)
x2v = x2(t, u)
x3v = x3(t, u)

    exp = to_felem((prime - 1) // 2)
e1 = fexp(f_p256(x1v), exp)
e2 = fexp(f_p256(x2v), exp)

    if e1 == 1:
        return x1v
    elif e2 == 1:
        return x2v
    else:
        return x3v
C.3. Simplified SWU Method

The following hacspec program implements map2curve_simple_swu(alpha) for P-256.

```python
from p256 import *
from hacspec.speclib import *

a256 = to_felem(prime - 3)
b256 = to_felem(41058363725152142129326129780047268409114441015993725554835256314039467401291)

def f_p256(x:felem_t) -> felem_t:
    return fadd(fexp(x, 3), fadd(fmul(to_felem(a256), x), to_felem(b256)))

def map2p256(t:felem_t) -> affine_t:
    alpha = to_felem(-(fsqr(t)))
    frac = finv((fadd(fsqr(alpha), alpha)))
    coefficient = fmul(to_felem(-b256), finv(to_felem(a256)))
    x2 = fmul(coefficient, fadd(to_felem(1), frac))
    x3 = fmul(alpha, x2)
    h2 = fadd(fexp(x2, 3), fadd(fmul(a256, x2), b256))
    h3 = fadd(fexp(x3, 3), fadd(fmul(a256, x3), b256))
    exp = fmul(fadd(to_felem(prime), to_felem(-1)), finv(to_felem(2)))
    e = fexp(h2, exp)

    exp = to_felem((prime + 1) // 4)
    if e == 1:
        return (x2, fexp(f_p256(x2), exp))
    else:
        return (x3, fexp(f_p256(x3), exp))
```

C.4. Boneh-Franklin Method

The following hacspec program implements map2curve_bf(alpha) for a supersingular curve "y^2=x^3+1" over "GF(p)" and "p = (2^250)(3^159)-1".

```python
...
```
from hacspec.speclib import *

prime = 2**250*3**159-1

a503 = to_felem(0)
b503 = to_felem(1)

@typechecked
def map2p503(u:felem_t) -> affine_t:
    t0 = fsqr(u)
    t1 = fsub(t0,b503)
    x = fexp(t1, (2 * prime - 1) // 3)
    return (x, u)

C.5. Fouque-Tibouchi Method

The following hacspec program implements map2curve_ft(alpha) for a BN
curve "BN256 : y^2=x^3+1" over "GF(p(t))", where "p(x) = 36x^4 +
36x^3 + 24x^2 + 6x + 1", and "t = -(2^62 + 2^55 + 1)".
from hacspec.speclib import *

t = -(2**62 + 2**55 + 1)
p = lambda x: 36*x**4 + 36*x**3 + 24*x**2 + 6*x + 1
prime = p(t)

aBN256 = to_felem(0)
bBN256 = to_felem(1)

@typechecked
def map2BN256(u:felem_t) -> affine_t:
    ZERO = to_felem(0)
    ONE = to_felem(1)
    SQRT_MINUS3 = fsqrt(to_felem(-3))
    ONE_SQRT3_DIV2 = fmul(finv(to_felem(2)), fsub(SQRT_MINUS3, ONE))

    fcurve = lambda x: fadd(fexp(x, 3), fadd(fmul(to_felem(aBN256), x), to_felem(bBN256)))
    flegendre = lambda x: fexp(u, (prime - 1) // 2)

    w = finv(fadd(fadd(fsqr(u), B), ONE))
    w = fmul(fmul(w, SQRT_MINUS3), u)
    e = flegendre(u)

    x1 = fsub(ONE_SQRT3_DIV2, fmul(u, w))
    fx1 = fcurve(x1)
    s1 = flegendre(fx1)
    if s1 == 1:
        y1 = fmul(fsqrt(fx1), e)
        return (x1, y1)

    x2 = fsub(ZERO, fadd(ONE, x1))
    fx2 = fcurve(x2)
    s2 = flegendre(fx2)
    if s2 == 1:
        y2 = fmul(fsqrt(fx2), e)
        return (x2, y2)

    x3 = fadd(finv(fsqr(w)), ONE)
    fx3 = fcurve(x3)
    y3 = fmul(fsqrt(fx3), e)
    return (x3, y3)

C.6. Elligator2 Method

The following hacspec program implements map2curve_elligator2(alpha) for Curve25519.
from curve25519 import *
from hacspect.speclib import *

a25519 = to_felem(486662)
b25519 = to_felem(1)
u25519 = to_felem(2)

@typechecked
def f_25519(x:felem_t) -> felem_t:
    return fadd(fmul(x, fsqr(x)), fadd(fmul(a25519, fsqr(x)), x))

@typechecked
def map2curve25519(r:felem_t) -> felem_t:
    d = fsub(to_felem(p25519), fmul(a25519, finv(fadd(to_felem(1), fmul(u25519, fsqr(r))))))
    power = nat((p25519 - 1) // 2)
    e = fexp(f_25519(d), power)
    x = 0
    if e != 1:
        x = fsub(to_felem(-d), to_felem(a25519))
    else:
        x = d
    return x

C.7. hash2base

The following procedure implements hash2base.
hash2base(x)

Parameters:

- H - cryptographic hash function to use
- hbits - number of bits output by H
- p - order of the base field Fp
- label - context label for domain separation

Preconditions:

floor(log2(p)) + 1 >= hbits

Input:

- x - an octet string to be hashed

Output:

- y - a value in the field Fp

Steps:

1. t1 = H("h2c" || label || I2OSP(len(x), 4) || x)
2. t2 = OS2IP(t1)
3. y = t2 mod p
4. Output y

where I2OSP, OS2IP [RFC8017] are used to convert an octet string to and from a non-negative integer, and a || b denotes concatenation of a and b.

C.7.1. Considerations

Performance: hash2base requires hashing the entire input x. In some algorithms/ciphersuite combinations, hash2base is called multiple times. For large inputs, implementers can therefore consider hashing x before calling hash2base. I.e. hash2base(H'(x)).

Most algorithms assume that hash2base maps its input to the base field uniformly. In practice, there will be inherent biases. For example, taking H as SHA256, over the finite field used by Curve25519 we have p = 2^255 - 19, and thus when reducing from 255 bits, the values of 0 .. 19 will be twice as likely to occur. This is a standard problem in generating uniformly distributed integers from a bitstring. In this example, the resulting bias is negligible, but for others this bias can be significant.
To address this, our hash2base algorithm greedily takes as many bits as possible before reducing mod p, in order to smooth out this bias. This is preferable to an iterated procedure, such as rejection sampling, since this can be hard to reliably implement in constant time.

The running time of each map2curve function is dominated by the cost of finite field inversion. Assuming $T_i(F)$ is the time of inversion in field $F$, a rough bound on the running time of each map2curve function is $O(T_i(F))$ for the associated field.

Appendix D. Test Vectors

This section contains test vectors, generated from reference Sage code, for each map2curve variant and the hash2base function described in Appendix C.7.

D.1. Elligator2 to Curve25519

Input:

alpha =

Intermediate values:

\begin{align*}
  u &= 140876c725e59a161990918755b3eff6a9d5e75d9ea20f9a4ebcf94e69ff013 \\
  v &= 6a262de4dca3a094ceb2d307fd985a018f55d1c7da5a3416423b462c8aaff893 \\
  gv &= 5dc09f578dca7bffeac3ec4ad2792c9822cd1d881839e823d26cd338f6ddc3e
\end{align*}

Output:

\begin{align*}
  x &= 15d9d21b245c5f6b314d2cf80267a5fe70aa2e382505cbe9bdc4b9d375489a54 \\
  y &= 1f132cbbfbb17d3f80e8a862a6fb437650775de0b86624f5a40d3e17739a07ff
\end{align*}
Input:
alpha = 00

Intermediate values:
\[ u = 10a97c83dec52945a72fe18511ac9741234de3fb62fa0fec399df \]
\[ v = 6ff5b9893b26c0c8b68adb3d653b335a8e810b4abbbec13348e828 \]
\[ gv = 2d1599d36275c36c6b34c07c62934e940c5248a9d275041f3724819d7e8b22 \]

Output:
\[ x = 6ff5b9893b26c0c8b68adb3d653b335a8e810b4abbbec13348e828 \]
\[ y = 553455c1c56434494c47dcfa9c7983c07fcb908f7a38717ba869a2469 \]

Input:
alpha = ff

Intermediate values:
\[ u = 59c48eefc872abc09321ca7231ed6c754c6524486a6e315e9e230716ed67ad3 \]
\[ v = 20392de0e96030c4a37cd6f650a86c6bc390bec21919d9c54f35f2a2534b2b \]
\[ gv = 0951a0c55b92e231494695cb775a0653a23f41635e11f97168e231095dd5c30c \]

Output:
\[ x = 5fc6d21f169fcf3b5c832909af5793943c6f4313de6e6263abb0ca0d5da547bc \]
\[ y = 2b6bf1b3322717ed564d04659757c86615c0dee954fbd695e8ac9d97e24d1 \]
Input:

alpha = ff0011223344112233441122334411223344556677885566778855667788

Intermediate values:

\begin{align*}
  u &= 380619de15c80fe3668bac96be51b0fd17129f6cf084a250cfaa76 \\
  v &= 2f3d9063e573c522d8f20c752f15b114f810b53d880154e2f30cde \\
  yv &= 4ce282b7cfdca2db63cec91a08b947f10fcf03bc69bafcd1c60b7d \\
  dv &= 305baaf
\end{align*}

Output:

\begin{align*}
  x &= 2f3d9063e573c522d8f20c752f15b114f810b53d880154e2f30cde \\
  y &= 5e43ab6a0590c11547b910d06d37c96e4cc3fc91ad8a54494d74b \\
  12de6ae45d
\end{align*}

D.2. Icart to P-384

Input:

alpha =

Intermediate values:

\begin{align*}
  u &= 287d7ef77451ecd3c1c0428092a70b5ed870ca22681c81ac52037d \\
    &a7e22a3657d3538fa5ce30488b8e5fb95eb58da86 \\
  u4 &= 56ae4e7e72dbae15bd0d5a8462d0228a5db9093268639e1cd015 \\
    &4aa3e63d81eea72c2d5fa4998f7ca971bb50b44d6f \\
  v &= eaa16e82d5a88eb9ff1866640c34693d4e32fdca72921ed2fe4d \\
    &cfce3b163dea8ec9528f7e3b5ca3e27cba5c97db9 \\
  x1 &= cbc52f2bf7f194a47fd88e3fa4f68fc41cdde8a8c47f79c225ad80 \\
    &455c4db0e5db47209754764929327edf339c19203b \\
  u6 &= 5af8bcb067c1fc0bf3c7115481f3bd78af70e35a9d067060c6e2 \\
    &164620d477e3247a55e514d0a790a7dd58e7482fa \\
  x1 &= 871a993757d3aa90b7261aa76fc1d7488b4dcbfc8505f1170e3707 \\
    &1ab59c93e88ca9d633173053d2b4f94a592b147
\end{align*}

Output:

\begin{align*}
  x &= b4e57fc7f87adbd52ab843635313cfd5fb356550b6fbde5741f6b \\
    &51b12b33a104bfe2c68bef24139332c7e213f145d5 \\
  y &= bd3980b713d51ac0f719b6cc045e2168717b74157f6fd0e36d4501 \\
    &3e2b5c7e0d70dacbb2fb826ad125d3f8a0dc5dc801f
\end{align*}
Input:

\(\alpha = 00\)

Intermediate values:

\[
\begin{align*}
  u &= 5584733e5ee080c9dbfa4a91c5c8da5552ccee17c74fae9d28380e6623493df985a7827f02538929373de483477b23521 \\
  u_4 &= 3f8451733c017a3e5acd8a310f5594ae539c74b009fc75aecda7f1abd42b3a47b1bd8b2b29eb3dd01db0a1bf67f5c15e \\
  v &= a20ff29b0a3d0067cb8a53e132753a46f598aa568efe00f9e286a5e4300c9010f58e3ed97b4b7b356347048f122ca2b8 \\
  x_1 &= d8fcadbcb05829f3d7d12493f8720514e2f125751f0d9c91ba8ee5de43456528c1e155cc93ac525562d9c3fcb3e49d3e3 \\
  u_6 &= 35340edd3abbe78fe33fd955e9126d67c6352db6ecbcbcf3abbaa530ffa37724d3a51d9d046057d0fa76278f916fa10c \\
  x_1 &= 382ba470b52fede86ed48a824ae3827a738b8cad54c9473d1eeee18b548b8f12389dce7c47893e18aafad06ab8ff52
\end{align*}
\]

Output:

\[
\begin{align*}
  x &= a15fe3979721e717f173c5d38882c011be02499d26a070a3bed825fcac5a251a1297a9593254a50f8aa243c6191976a \\
  y &= 641d1cb53087208240a935769ca1b99c3a97a492526e5b3cfae8c20bebde9345c4dd549e2d01d5417918ce039451f4d7
\end{align*}
\]
Input:

\[ \alpha = ff \]

Intermediate values:

\[ u = 72b21e3c82881ab0a2845ec645dd9d6fd4f3c74cb3 \]
\[ u_4 = 60cbd41d32d7588ff3634655bd5e5ef6ab9077b7629bb648669cf8 \]
\[ v = f3e63b1b10195a28833f391d480df124be3c1cbbaa0c7b5b0252db \]
\[ x_1 = 9d4c43b595deb99138eb0f7688695abe8a7145d4b8f1f911b8384b \]
\[ u_6 = bb44318a26c920aa39270421eb8ff73aac89637d01e6b32697fbd2 \]
\[ x_1 = aa283d625fdb4d12761e359d6bd6a2de63f036a2d9d1373c11d9 \]
\[ \text{Output:} \]

\[ x = 26536b1be6480de4e8d3232e17312085d2fc5b4ad18aae3edfe1f6 \]
\[ y = 7533cf819fa713699f4919f79fc0f01c9b632f2e08a5a34de7d9e \]
Input:

\[ \alpha = \text{ff0011223344112233441122334455667788556677885566778855667788} \]

Intermediate values:

\[ u = \text{e1a5025e8e9b677623767613cd90b685a46fe462c914aaf7dab3b} \]
\[ u4 = \text{be47baa8671fb710a0cf58c85d95ea9cecf2a7d6a6d859f3dcb52be} \]
\[ v = \text{24ed8cb050c045f6401a6221b85c37d482197f54a7340303449c13} \]
\[ x1 = \text{24ed8cb050c045f6401a6221b85c37d482197f54a7340303449c13} \]
\[ u6 = \text{e806b407aef7874ad4ded43a46bc002e0ddea39a5754cf09dfcb9} \]
\[ x1 = \text{e806b407aef7874ad4ded43a46bc002e0ddea39a5754cf09dfcb9} \]

Output:

\[ x = \text{810096c7dec85367fa04f706c2e456334325202b9fcbbc34970d9fd} \]
\[ y = \text{ddde061cec6efc0cfdadbc0241fd00ab2ad2b8f8e00dc0d45f8} \]

D.3. SWU to P-256
Input:

\[ \alpha = \]

Intermediate values:

\[ u = d8e1655d6562677a74be47c33ce9edcbefd5596653650e5758c8aaab65a99db3 \]
\[ v = 7764572395df002912b7cbb93c9c287f325b57afa1e7e82618ba579b796e6ad1 \]
\[ x1 = 7764572395df002912b7cbb93c9c287f325b57afa1e7e82618ba579b796e6ad1 \]
\[ g_v = 0d8af0935d993caaeefca7ef912e06415cbe7e00a93cca295237c667f0cc2f941 \]
\[ g_{x1} = 0d8af0935d993caaeefca7ef912e06415cbe7e00a93cca295237c667f0cc2f941 \]
\[ n_1 = ef66b409fa309a99e4dd4a1922711dea389925d4a5947b3a0e3fe34efdfc0cf \]
\[ x_2 = 2848af84de537f96c3629d93a78b37413a8b07c72248be8eac61faa058cedf96 \]
\[ g_{x2} = 3ae6b1a6a81f78b9176847f84ab7987f361cb486864d4dbf3e45af2d9354fb36a \]
\[ x_3 = 4331af86e99e4fc7a3e5f0ca7b8a62c3c9f0146dacef75b6900fe60b8293e8e \]
\[ g_{x3} = 1d78aa2bd9ff7c11c53807622c4d476ed67ab3c93206225ae437f086ebaa2982 \]
\[ y_1 = 574e9564a28b9104b9dfb104a976f5f6a07c5c5b69e901e596df26e4f571e369 \]

Output:

\[ x = 7764572395df002912b7cbb93c9c287f325b57afa1e7e82618ba579b796e6ad1 \]
\[ y = 574e9564a28b9104b9dfb104a976f5f6a07c5c5b69e901e596df26e4f571e369 \]
Input:

\[ \text{alpha} = 00 \]

Intermediate values:

\[
\begin{align*}
\text{u} &= \text{c4188ee0e554daea559d04d45982d6b184eff86c4a910a4324744d6fb3c62} \\
\text{v} &= \text{0e82c0c07eb17c24c84f4a83fdd6195c23f76d455ba7a8d5bc3f620ce20caf9} \\
\text{x1} &= \text{0e82c0c07eb17c24c84f4a83fdd6195c23f76d455ba7a8d5bc3f620ce20caf9} \\
\text{g1} &= \text{4914f49c40c561bfed65762d4bbf652e236f890ae752ea10460be2939c3a} \\
\text{g2} &= \text{4914f49c40c561bfed65762d4bbf652e236f890ae752ea10460be2939c3a} \\
\text{n1} &= \text{ae5000e861347ff29e3368597174b1a0a04b9b08019f59936baa6f7e3176cf03} \\
\text{x2} &= \text{331a4d8dead57f3d36e239e9cfaaaf6804354a5897da421db73a795c3f9af7} \\
\text{g2} &= \text{b3dda8702e046be4e2bd42e29f09fddbc98a3fe04bd91ca8a19045684be9d81} \\
\text{x3} &= \text{1133498ac9e9b658271586be95ca43a946aa320eb32e796624766ac7d1cc60} \\
\text{g3} &= \text{7cd39b42a3b487dc6c2782a5aebd123502b9fecc849be21766c8a00ca16c318f} \\
\text{y2} &= \text{66cf6a249077e13be24cf2cfab67dfcc8407a299e69c817785b8b9a23eecefe734} 
\end{align*}
\]

Output:

\[
\begin{align*}
\text{x} &= \text{331a4d8dead57f3d36e239e9cfaaaf6804354a5897da421db73a795c3f9af7} \\
\text{y} &= \text{66cf6a249077e13be24cf2cfab67dfcc8407a299e69c817785b8b9a23eecefe734} 
\end{align*}
\]
Input:

alpha = ff

Intermediate values:

\[ u = 777b56233c4db9fe7de8b046189d39e0b2c2add660221e7c4a2d458c3034df2 \]
\[ v = 51a60aecd0ade7769bd04a4a3241130e00c7adaa91f76f1e115f1d082902b02 \]
\[ x1 = 51a60aecd0ade7769bd04a4a3241130e00c7adaa91f76f1e115f1d082902b02 \]
\[ gv = f7ba284fd26c0cb7b678f71caecbd9bf88890daba48b596927c70bf805f5e5ebaf805f5e5ebea \]
\[ gx1 = f7ba284fd26c0cb7b678f71caecbd9bf88890daba48b596927c70bf805f5e5ebea \]
\[ n1 = a437e699818d87069a6e4d5298f26f19fd301835eb33b0a3936e3bd1507680 \]
\[ x2 = 7236d245e18dfd43d756a2d048c6e491bb9ebfc2ca627e315d49b1e02957fc \]
\[ gx2 = 9d6ebf27637ca38ee894e5052b989021b7d76fa2b01053ce05429554a205c047 \]
\[ x3 = 90553fad8a170464497621e7f2ffcc35d17af4107b79dab6d2a126ea692c9db \]
\[ gx3 = d7d141749e2e8e4b2253d4ef22e3ba7c7970e604e03b59277aed1032f02ca11 \]
\[ y1 = 4115534ea22d3b46a9c541a25e72b3f37a2ac7635a6bebb16ff504c3170fb69a \]

Output:

\[ x = 51a60aecd0ade7769bd04a4a3241130e00c7adaa91f76f1e115f1d082902b02 \]
\[ y = 4115534ea22d3b46a9c541a25e72b3f37a2ac7635a6bebb16ff504c3170fb69a \]
Input:

\[ \alpha = \text{ff0011223344112233441122334455667788556677885566778855} \]

Intermediate values:

\[ u = 87541ffa2efec46a38875330f66a6a53b99edce4e407e06cd0ccaf39f8208aa6 \]
\[ v = 3dbb1902335f823df0d4fe0797456bfee25d0a2016ae6e357197c4122bf7e310 \]
\[ x1 = 3dbb1902335f823df0d4fe0797456bfee25d0a2016ae6e357197c4122bf7e310 \]
\[ g_v = 2704056d76b889ce788ab5cc68fd932f3d7cb125d0dbe0afba9dd7655d0651ed \]
\[ g_x1 = 2704056d76b889ce788ab5cc68fd932f3d7cb125d0dbe0afba9dd7655d0651ed \]
\[ n_1 = 43b52359e2739c205b2e4c8a0b3cd6842feb2ed131ec37fc0788eb264dc1999b \]
\[ x_2 = 39150bd341015403c27154093cd0382d61d27dafef1dbe7083683223bc3e1b2a \]
\[ g_x2 = 0985d428671b570b3c94dbaa2c4f160095db0a3d79b738c488ca8b45971d03 \]
\[ x_3 = 30cf2e681176c3e50b36842e3e7623ba0577f6a1a0572448ab5ba4bcf9c3d71 \]
\[ g_x3 = ea7c1f13e2ab39240dd74e884f0878d21020fd73b7f4f84c79ad72d009ae0 \]
\[ y_2 = 71b6dea4bc8dcae3dab695b69f25a7dbdc4e00f4926407bad89a80ab12655340 \]

Output:

\[ x = 39150bd341015403c27154093cd0382d61d27dafef1dbe7083683223bc3e1b2a \]
\[ y = 71b6dea4bc8dcae3dab695b69f25a7dbdc4e00f4926407bad89a80ab12655340 \]

D.4. Simple SWU to P-256
Input:

alpha =

Intermediate values:

\[ u = 650354c1367c575b44d039f35a05f2201b3b3d2a93bf4ad6e5535bb5838c24e \]
\[ n1 = 88d14bad9d79058c1427aa778892529b513234976ce84015c795f3b3c1860963 \]
\[ x1 = c55836cadcb8c4f99e345c88aa0af67db2d3e6e527de7a5b7a859a3f6a2d3 \]
\[ gx1 = 9104bf247de931541fedfd4a483ced90fd3ac32f4bb8b0de021a21f770fccc7ae \]
\[ x2 = 0243b55837314f184ed8eca38b733945ec124ffd079850de608c9d175aed9d29 \]
\[ gx2 = 0f522f68139c6a8ff028c5c24536069441c3eae8a68d49939b20190a87e2f170 \]
\[ y2 = 29b59b5c656bdf740b3ea8efad626a1f072eb384f2db56903f67fe4fbb6ff82 \]

Output:

\[ x = 0243b55837314f184ed8eca38b733945ec124ffd079850de608c9d175aed9d29 \]
\[ y = 29b59b5c656bdf740b3ea8efad626a1f072eb384f2db56903f67fe4fbb6ff82 \]
Input:

\[ \alpha = 00 \]

Intermediate values:

\[
\begin{align*}
  u &= 54acd0c1b3527a157432500fc3403b6f8a0aa0103d6966b783614a8e41c9c5b1 \\
  n1 &= bb27567ea0729adc2b7af65a85b7f599559b107ce0d2495c4d26d8a1ce842372 \\
  x1 &= 6ae899e0232f040f8a82934f462e1ccedac76ad8549ae581f17c821a5944244f \\
  g1 &= 8a78bbf9c2156533fa0d9d37533752508a061b90108675ad7050097adabff9cb \\
  x2 &= 498c0e2faee29adf4e6aed9120eb8c69cd3bb7206bcd47a746fb5ed4ed5529a5 \\
  g2 &= 63adfce3aaa4d56b70cc3e8e7475154b5963855e275fffc26858cbf2456ea5f52 \\
  y1 &= 3b81976ce93e79d2ba13394a6b5deb34602d6829f4625d987fc98ca79d5d5c98
\end{align*}
\]

Output:

\[
\begin{align*}
  x &= 6ae899e0232f040f8a82934f462e1ccedac76ad8549ae581f17c821a5944244f \\
  y &= 3b81976ce93e79d2ba13394a6b5deb34602d6829f4625d987fc98ca79d5d5c98
\end{align*}
\]
Input:

alpha = ff

Intermediate values:

\[
\begin{align*}
u &= 86855e4bc3905ae04f6b284820856db6809633c5046ed92816a4e9976e994818 \\
n1 &= 5ec1cf436c1a2e84b53674bcf2470a0aeeda9550c474b06da4bda83bda56f2e3 \\
x1 &= 04e73147d10de271f7d77a9a3d6dd761d5b892ab39224b9dab93a250139b124a \\
gx1 &= 9d26bd1c5a97ccf9a7963a099e3c0b98070525b7ed08e8f32f44aef918b15f \\
x2 &= 28566b4d673bf59f00d42771968bd69b1a54e8b557857ba231cbddfeb18b38b5 \\
gx2 &= 3b7edb432f00509ed44a4e6a2cbdbc69321215097953dac5bab8a901ad0998 \\
y2 &= 6644bab658f2915f2129791db0eb29eaeb34036db1bced721b161e06caae008
\end{align*}
\]

Output:

\[
\begin{align*}
x &= 28566b4d673bf59f00d42771968bd69b1a54e8b557857ba231cbddfeb18b38b5 \\
y &= 6644bab658f2915f2129791db0eb29eaeb34036db1bced721b161e06caae008
\end{align*}
\]
Input:

\[
\alpha = \text{ff00112233441122334411223344112233445566778855667788556677885566778855667788556677885566778855667788556677885566778855667788}
\]

Intermediate values:

\[
u = \text{34a8fc904e2d40dabb826b914917a6f6f6f8716b26f8f4b7aaf}
\]

\[
n_1 = \text{3b14efe9953378860e667b9051f9e412811e71b489ad8b72a8856f}
\]

\[
x_1 = \text{8ac342ff43931be5b1a9de4f602994853fa9ec943eacc5e39760df73fb4d9799}
\]

\[
g_{x1} = \text{b45e916f64f943e1ba89e559c38f95457f2cadc1aa8d54b0cac9507ebc6ba}
\]

\[
x_2 = \text{df9e15f7507632859104da82a28882021608b2c4f2fcf3b1a82e432841284ec7}
\]

\[
g_{x2} = \text{194033ff4cd98e41cc5e863eb355168b5d794af03ca374244c7ac9ac5e2f7b0}
\]

\[
y_2 = \text{180369d261ec6086346e6b2d36990a3aaa803558f1398b6816c3c618d41ff73e}
\]

Output:

\[
x = \text{f9e15f7507632859104da82a28882021608b2c4f2fcf3b1a82e432841284ec7}
\]

\[
y = \text{180369d261ec6086346e6b2d36990a3aaa803558f1398b6816c3c618d41ff73e}
\]

D.5. Boneh-Franklin to P-503

The P-503 curve is a supersingular curve defined as "\(y^2 = x^3 + 1\)" over "GF(p)", where \(p = 2^{250} \times 3^{159} - 1\).
Input:
alpha =

Intermediate values:

\[ u = 198008fe3da9ee741c2ff07b9d4732df88a3cb98e8227b2cf49d55 \]
\[ 57aec1e61d1d29f460c6e4572b2baa21d2444d6459cdd2c0dfa2 \]
\[ 0144dfab7e92a83e00 \]
\[ t0 = 1f6bb1854a1ff7db82b43c235727d998fe28889152ec4efa533994 \]
\[ fc6d0e77c9f3dddb8c46226de8e5de75f70537094b809fe0ca092 \]
\[ 8587addb9c54ae1a05 \]
\[ t1 = 1f6bb1854a1ff7db82b43c235727d998fe28889152ec4efa533994 \]
\[ fc6d0e77c9f3dddb8c46226de8e5de75f70537094b809fe0ca092 \]
\[ 8587addb9c54ae1a04 \]
\[ x = 04671bf3e7f9f7905848cd4c0fc652bd22200e6df29ef8e13cccb \]
\[ 5536e4a1db4366d2f346070d63c994bf0a4b1a4e555d6b3d021a \]
\[ eba340b641ada82054 \]

Output:

\[ x = 04671bf3e7f9f7905848cd4c0fc652bd22200e6df29ef8e13cccb \]
\[ 5536e4a1db4366d2f346070d63c994bf0a4b1a4e555d6b3d021a \]
\[ eba340b641ada82054 \]
\[ y = 198008fe3da9ee741c2ff07b9d4732df88a3cb98e8227b2cf49d55 \]
\[ 57aec1e61d1d29f460c6e4572b2baa21d2444d6459cdd2c0dfa2 \]
\[ 0144dfab7e92a83e00 \]
Input:

alpha = 00

Intermediate values:

\[ u = 30e30a56d82cdca830f08d729ce909fc1ffec68df49ba75f9a1af7 \\
2ca242e92742f34b474a299bb452c6a71b69bdc9ee2403eaac7c84 \\
120a160737d667e29e \]

\[ t0 = 0a64d9f288a0881bb6addebc0db89f146b282b05570efa3419f5d3 \\
f1lec7bb449alda8b33817642f01db039f838ad0bd459ec03e76d \\
8ee3a1e79d6c63f79 \]

\[ t1 = 0a64d9f288a0881bb6addebc0db89f146b282b05570efa3419f5d3 \\
f1lec7bb449alda8b33817642f01db039f838ad0bd459ec03e76d \\
8ee3a1e79d6c63f78 \]

\[ x = 0970ff4bb9237704cc30f5b0d80a9d97001064ab4cddbf8de74f8d7 \\
283b922726406393c07ad01de0499e46ebc0ed1cd116112cf8965f \\
b8f918205adb13d3a \]

Output:

\[ x = 0970ff4bb9237704cc30f5b0d80a9d97001064ab4cddbf8de74f8d7 \\
283b922726406393c07ad01de0499e46ebc0ed1cd116112cf8965f \\
b8f918205adb13d3a \]

\[ y = 30e30a56d82cdca830f08d729ce909fc1ffec68df49ba75f9a1af7 \\
2ca242e92742f34b474a299bb452c6a71b69bdc9ee2403eaac7c84 \\
120a160737d667e29e \]
Input:

\[ \alpha = \text{ff} \]

Intermediate values:

\[
\begin{align*}
u &= 3808ae24b17af9147bd16077e3e83aff5c579784c8a1443d90e5ff \\
e2451bfabacba73ee8b8f652b991290f5c64b34b1a4c9a498e21d4 \\
3d000dae7f8860200a \\
t_0 &= 2282d37dce4761dad69df0e12c8580ba4e23158a0621fb3f51813 \\
10e7275e95573c89a8f0cda7ad98ca9e0a9e04ef94a1a79685d069 \\
6ac6ad423a0de96b7d \\
t_1 &= 2282d37dce4761dad69df0e12c8580ba4e23158a0621fb3f51813 \\
10e7275e95573c89a8f0cda7ad98ca9e0a9e04ef94a1a79685d069 \\
6ac6ad423a0de96b7c \\
x &= 173dc6d853d9024f367e24a2837668e11ce559473e788f3c0ed0281 \\
6b48043fc6e100d4935b3f6197799bfb4fd94b365656252f12b \\
27fa46602c76ae1370
\end{align*}
\]

Output:

\[
\begin{align*}
x &= 173dc6d853d9024f367e24a2837668e11ce559473e788f3c0ed0281 \\
6b48043fc6e100d4935b3f6197799bfb4fd94b365656252f12b \\
27fa46602c76ae1370 \\
y &= 3808ae24b17af9147bd16077e3e83aff5c579784c8a1443d90e5ff \\
e2451bfabacba73ee8b8f652b991290f5c64b34b1a4c9a498e21d4 \\
3d000dae7f8860200a
\end{align*}
\]
Input:

\alpha = \text{ff0011234112334412233444556677885566778855}
66778855667788

Intermediate values:

\begin{align*}
  u &= \text{3ebdfccb07dcd61d9f81be2b9f5a7a8733581f1a8d531d78229d7b}
      \text{0be50f30887f085ef393422ef96e06ff1df4b608b05c53320a9012}
      \text{09b8df48b68ab338ec} \\
  t0 &= \text{27958e69b08a9fd2d1765ce3e8dbaf8645c28e5ce033b9d0a7875c}
      \text{e7e73d6583e62ff3a06a2b55de1cb8c26819d0cd4aed2dc7cb65fa}
      \text{d5eb3c149db9e8381b} \\
  t1 &= \text{27958e69b08a9fd2d1765ce3e8dbaf8645c28e5ce033b9d0a7875c}
      \text{e7e73d6583e62ff3a06a2b55de1cb8c26819d0cd4aed2dc7cb65fa}
      \text{d5eb3c149db9e8381a} \\
  x &= \text{3fe94cd4d2be061834da5020ca181562fdb7e9787f71965ca55cd}
      \text{dbf069b68dd5e2b05a5696a061723093914e69b0540402bba0db3}
      \text{fddc517df4211daea1}
\end{align*}

Output:

\begin{align*}
  x &= \text{3fe94cd4d2be061834da5020ca181562fdb7e9787f71965ca55cd}
      \text{dbf069b68dd5e2b05a5696a061723093914e69b0540402bba0db3}
      \text{fddc517df4211daea1} \\
  y &= \text{3ebdfccb07dcd61d9f81be2b9f5a7a8733581f1a8d531d78229d7b}
      \text{0be50f30887f085ef393422ef96e06ff1df4b608b05c53320a9012}
      \text{09b8df48b68ab338ec}
\end{align*}

D.6. Fouque-Tibouchi to BN256

An instance of a BN curve is defined as "BN256: \(y^2=x^3+1\)" over "GF(\(p(t)\))" such that

\begin{align*}
  t &= -(2^{62} + 2^{55} + 1), \\
  p &= 0x2523648240000001ba344d800000000861210000000000013a70000000000013
\end{align*}
Input:

\[ \alpha = \]

Intermediate values:

\[ u = 1f6f2aceae3d9323ea64e9be00566f863cc1583385eaff6b01aed7a762b11122 \]
\[ t0 = 1e9c884ab8d2015985a3e3d2764798b183ff5982b0fd9034f274560f19d06ed0 \]
\[ x1 = 0843eb0f5ed559e940a453f257b2a2e297895ecc2375a070168117b5127ec2ae \]
\[ x2 = 1cdf7972e12aa618798ff98da84d5d25c997a133dc8a5fa3907ee84aed813d64 \]
\[ x3 = 042f756fe42e2ed4c58990da3b2567a7b16252c0e17b2da55b8f68be71ebd432 \]
\[ e = 2523648240000001ba344d800000008612100000000013a7000000000012 \]
\[ fx1 = 0a8442855e93541a104052273e2bbba930338d392d71f70efe83c77ae95471a4e \]
\[ y1 = 135a017a32abc542796e55d0b68840546c3b2498963773635e27c25aa3737199 \]

Output:

\[ x = 0843eb0f5ed559e940a453f257b2a2e297895ecc2375a070168117b5127ec2ae \]
\[ y = 135a017a32abc542796e55d0b68840546c3b2498963773635e27c25aa3737199 \]
Input:

alpha = 00

Intermediate values:

\[ u = 053c7251b0e5e5c9acde43c6abd44ffebeb13109f61ec27ba0a8191f1165435065 \]

\[ t0 = 0377baf027b80854661187280a98ae1320d7fd8cb0a65fd70772706c38cb71d8 \]

\[ x1 = 0f5173cd2eb84352497a9cb56ebf40b623d9dabb7dcc3f626b1f389e49b9356 \]

\[ x2 = 15d1f0b511472bcc959ca3b4a9140bfcfee3625448233c1d804e0c761b646cbc \]

\[ x3 = 100fb33ce2b98b99ca5a279e1b4e5b0cf6927ded3cb729a822483809e486dc7 \]

\[ e = 252364824000001ba344d800000008612100000000000013a700000000000012 \]

\[ fx1 = 044c88525cbf81408b9bac1c83bdc49e3f31ec5a7b68495b5d03e518448af09 \]

\[ y1 = 18e4bd91f687e110fb5f57411fccc34b4b1d16d3978a75d988c38d338522d7c \]

Output:

\[ x = 0f5173cd2eb84352497a9cb56ebf40b623d9dabb7dcc3f626b1f389e49b9356 \]

\[ y = 18e4bd91f687e110fb5f57411fccc34b4b1d16d3978a75d988c38d338522d7c \]
Input:

\[ \alpha = ff \]

Intermediate values:

\[
\begin{align*}
u & = \text{077033c69096f00eb76446a64be88c7a}51f1921b977381a6f2e9a8336191e783 \\
t0 & = \text{1716fb7790dd8e2e5a3ef94d63ca31682dd8b92ce13b93e0977943} \\
x1 & = \text{187ca1d0f0dec664467d49b4a}a661602faac5453fbd4ad9e3f15d \\
    & \quad \text{a35627459e} \\
x2 & = \text{0ca6c2b14f21399d73b703cb5b599ea831763abac042b539c30ea2} \\
    & \quad \text{5ca9d8ba74} \\
x3 & = \text{0f694914de2533b1fba6495b1de12cde6965bba0b505b527c1cb0} \\
    & \quad \text{69a5ffdf03} \\
e & = \text{00000000000000000000000000000000000000000000000000000000000000001} \\
fx1 & = \text{067a294268373f0123d95357d7d46c730277e67e68daf3a2c605bf} \\
    & \quad \text{035f680a7b} \\
y1 & = \text{0de5f5d8ecfc19580a882c53c08b47791edf4499965df86263c525} \\
    & \quad \text{afdf4fe0769} \\
\end{align*}
\]

Output:

\[
\begin{align*}
x & = \text{187ca1d0f0dec664467d49b4a}a661602faac5453fbd4ad9e3f15d \\
    & \quad \text{a35627459e} \\
y & = \text{0de5f5d8ecfc19580a882c53c08b47791edf4499965df86263c525} \\
    & \quad \text{afdf4fe0769} \\
\end{align*}
\]
Input:

\[ \text{alpha} = \text{ff00112233441122334411223344556677885566778855667788} \]

Intermediate values:

\[ u = 1\text{dd9ec37d5abeed0f289dadd685d45a395a90f2730a9adead62bf} \]
\[ t0 = 23\text{d0adbb23709a3732948019e038c13f498b33812149fe503b68da} \]
\[ x1 = 0\text{e2d073931bc2f38a069df42afbf9e6f04155e52cf6211be3d40} \]
\[ x2 = 24\text{40940eace43d0e302daf8b5040369f21ceaa1ad309e01e8c2bf} \]
\[ x3 = 0\text{9c1ba4259e59a54221b5761cf9438a60e6cd644996e7c8a11be96} \]
\[ e = 25\text{23648240000001ba344d8000000086121000000000013a7000} \]
\[ fx1 = 0\text{80e2ae1644070ac0f9d6563db6805684572eb33f457d9d75ed5c} \]
\[ fx2 = 0\text{c2937174e6a4a89c1574ed4fa96d83a64fb9670c49a8b492321a} \]
\[ fx3 = 1\text{18bc595ca0eac3ae6e56595267670caf75d34386dadc99284bf8} \]
\[ y3 = 1\text{90e8d47070240ff3c78a03d07123334e67b207fe555c31d0900fe} \]

Output:

\[ x = 0\text{9c1ba4259e59a54221b5761cf9438a60e6cd644996e7c8a11be96} \]
\[ y = 1\text{90e8d47070240ff3c78a03d07123334e67b207fe555c31d0900fe} \]

D.7. Sample hash2base

```
hash2base("H2C-Curve25519-SHA256-Elligator-Clear", 1234)
= 1e10b542835e7b227c727bd0a7b2790f39ca1e09fc8538b3c70e736cb1c298f
hash2base("H2C-P256-SHA512-SWU-", 1234)
= 4fabef095423c97566bd28b70ee70fb4dd95acfeec076862f4e40981a6c9dd85
hash2base("H2C-P256-SHA512-SSWU-", 1234)
= d6f68507d692e24ae13ab154684ae46c5311b78a704c6e11b2f44f4db4c6e47
```
Authors’ Addresses

Sam Scott
Cornell Tech
2 West Loop Rd
New York, New York 10044
United States of America

Email: sam.scott@cornell.edu

Nick Sullivan
Cloudflare
101 Townsend St
San Francisco
United States of America

Email: nick@cloudflare.com

Christopher A. Wood
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: cawood@apple.com
Randomness is a crucial ingredient for TLS and related security protocols. Weak or predictable "cryptographically-strong" pseudorandom number generators (CSPRNGs) can be abused or exploited for malicious purposes. The Dual EC random number backdoor and Debian bugs are relevant examples of this problem. An initial entropy source that seeds a CSPRNG might be weak or broken as well, which can also lead to critical and systemic security problems. This document describes a way for security protocol participants to augment their CSPRNGs using long-term private keys. This improves randomness from broken or otherwise subverted CSPRNGs.
1. Introduction

Randomness is a crucial ingredient for TLS and related transport security protocols. TLS in particular uses random number generators (generally speaking, CSPRNGs) to generate several values: session IDs, ephemeral key shares, and ClientHello and ServerHello random values. CSPRNG failures such as the Debian bug described in [DebianBug] can lead to insecure TLS connections. CSPRNGs may also be intentionally weakened to cause harm [DualEC]. Initial entropy sources can also be weak or broken, and that would lead to insecurity of all CSPRNG instances seeded with them. In such cases where CSPRNGs are poorly implemented or insecure, an adversary may be able to predict its output and recover secret Diffie-Hellman key shares that protect the connection.

This document proposes an improvement to randomness generation in security protocols inspired by the "NAXOS trick" [NAXOS]. Specifically, instead of using raw randomness where needed, e.g., in generating ephemeral key shares, a party’s long-term private key is mixed into the entropy pool. In the NAXOS key exchange protocol, raw...
random value \( x \) is replaced by \( H(x, sk) \), where \( sk \) is the sender’s private key. Unfortunately, as private keys are often isolated in HSMs, direct access to compute \( H(x, sk) \) is impossible. An alternate yet functionally equivalent construction is needed.

The approach described herein replaces the NAXOS hash with a keyed hash, or pseudorandom function (PRF), where the key is derived from a raw random value and a private key signature. Implementations SHOULD apply this technique when indirect access to a private key is available and CSPRNG randomness guarantees are dubious, or to provide stronger guarantees about possible future issues with the randomness. Roughly, the security properties provided by the proposed construction are as follows:

1. If the CSPRNG works fine, that is, in a certain adversary model the CSPRNG output is indistinguishable from a truly random sequence, then the output of the proposed construction is also indistinguishable from a truly random sequence in that adversary model.

2. An adversary \( \text{Adv} \) with full control of a (potentially broken) CSPRNG and able to observe all outputs of the proposed construction, does not obtain any non-negligible advantage in leaking the private key, modulo side channel attacks.

3. If the CSPRNG is broken or controlled by adversary \( \text{Adv} \), the output of the proposed construction remains indistinguishable from random provided the private key remains unknown to \( \text{Adv} \).

2. Randomness Wrapper

Let \( x \) be the output of a CSPRNG. When properly instantiated, \( x \) should be indistinguishable from a random string of \( x \) bytes. However, as previously discussed, this is not always true. To mitigate this problem, we propose an approach for wrapping the CSPRNG output with a construction that mixes secret data into a value that may be lacking randomness.

Let \( G(n) \) be an algorithm that generates \( n \) random bytes, i.e., the output of a CSPRNG. Define an augmented CSPRNG \( G' \) as follows. Let \( \text{Sig}(sk, m) \) be a function that computes a signature of message \( m \) given private key \( sk \). Let \( H \) be a cryptographic hash function that produces output of length \( M \). Let \( \text{Extract}(\text{salt}, \text{IKM}) \) be a randomness extraction function, e.g., HKDF-Extract [RFC5869], which accepts a salt and input keying material (IKM) parameter and produces a pseudorandom key of length \( L \) suitable for cryptographic use. Let \( \text{Expand}(k, \text{info}, n) \) be a variable-length output PRF, e.g., HKDF-Expand [RFC5869], that takes as input a pseudorandom key \( k \) of length \( L \), info
string, and output length n, and produces output of n bytes. Finally, let tag1 be a fixed, context-dependent string, and let tag2 be a dynamically changing string.

The construction works as follows. Instead of using G(n) when randomness is needed, use G'(n), where

\[ G'(n) = \text{Expand}(\text{Extract}(H(Sig(sk, tag1)), G(L)), tag2, n) \]

Functionally, this expands n random bytes from a key derived from the CSPRNG output and signature over a fixed string (tag1). See Section 3 for details about how "tag1" and "tag2" should be generated and used per invocation of the randomness wrapper. Expand() generates a string that is computationally indistinguishable from a truly random string of n bytes. Thus, the security of this construction depends upon the secrecy of H(Sig(sk, tag1)) and G(L). If the signature is leaked, then security of G'(n) reduces to the scenario wherein randomness is expanded directly from G(L).

If a private key sk is stored and used inside an HSM, then the signature calculation is implemented inside it, while all other operations (including calculation of a hash function, Extract and Expand functions) can be implemented either inside or outside the HSM.

Sig(sk, tag1) should only be computed once for the lifetime of the randomness wrapper, and MUST NOT be used or exposed beyond its role in this computation. To achieve this, tag1 may have the format that is not supported (or explicitly forbidden) by other applications using sk.

Sig MUST be a deterministic signature function, e.g., deterministic ECDSA [RFC6979], or use an independent (and completely reliable) entropy source, e.g., if Sig is implemented in an HSM with its own internal trusted entropy source for signature generation.

In systems where signature computations are expensive, Sig(sk, tag1) may be cached. In that case the relative cost of using G'(n) instead of G(n) tends to be negligible with respect to cryptographic operations in protocols such as TLS. A description of the performance experiments and their results can be found in the appendix of [SecAnalysis].

Moreover, the values of G'(n) may be precomputed and pooled. This is possible since the construction depends solely upon the CSPRNG output and private key.
3. Tag Generation

Both tags SHOULD be generated such that they never collide with another contender or owner of the private key. This can happen if, for example, one HSM with a private key is used from several servers, or if virtual machines are cloned.

To mitigate collisions, tag strings SHOULD be constructed as follows:

- **tag1**: Constant string bound to a specific device and protocol in use. This allows caching of $\text{Sig}(sk, \text{tag1})$. Device specific information may include, for example, a MAC address. To provide security in the cases of usage of CSPRNGs in virtual environments, it is RECOMMENDED to incorporate all available information specific to the process that would ensure the uniqueness of each tag1 value among different instances of virtual machines (including ones that were cloned or recovered from snapshots). It is needed to address the problem of CSPRNG state cloning (see [RY2010]). See Section 4 for example protocol information that can be used in the context of TLS 1.3.

- **tag2**: Non-constant string that includes a timestamp or counter. This ensures change over time even if outputs of $G(L)$ were to repeat. It MUST be implemented such that its values never repeat. This means, in particular, that timestamp is guaranteed to change between two requests to CSPRNG (otherwise counters should be used).

4. Application to TLS

The PRF randomness wrapper can be applied to any protocol wherein a party has a long-term private key and also generates randomness. This is true of most TLS servers. Thus, to apply this construction to TLS, one simply replaces the "private" CSPRNG $G(n)$, i.e., the CSPRNG that generates private values, such as key shares, with:

$$G'(n) = \text{HKDF-Expand}(\text{HKDF-Extract}(H(\text{Sig}(sk, \text{tag1})), G(L)), \text{tag2}, n)$$

Moreover, we fix tag1 to protocol-specific information such as "TLS 1.3 Additional Entropy" for TLS 1.3. Older variants use similarly constructed strings.

5. Acknowledgements

We thank Liliya Akhmetzyanova for her deep involvement in the security assessment in [SecAnalysis].
6. IANA Considerations

This document makes no request to IANA.

7. Security Considerations

A security analysis was performed in [SecAnalysis]. Generally speaking, the following security theorem has been proven: if the adversary learns only one of the signature or the usual randomness generated on one particular instance, then under the security assumptions on our primitives, the wrapper construction should output randomness that is indistinguishable from a random string.

The main reason one might expect the signature to be exposed is via a side-channel attack. It is therefore prudent when implementing this construction to take into consideration the extra long-term key operation if equipment is used in a hostile environment when such considerations are necessary. Hence, it is recommended to generate a key specifically for the purposes of the defined construction and not to use it another way.

The signature in the construction as well as in the protocol itself MUST NOT use randomness from entropy sources with dubious security guarantees. Thus, the signature scheme MUST either use a reliable entropy source (independent from the CSPRNG that is being improved with the proposed construction) or be deterministic: if the signatures are probabilistic and use weak entropy, our construction does not help and the signatures are still vulnerable due to repeat randomness attacks. In such an attack, the adversary might be able to recover the long-term key used in the signature.

Under these conditions, applying this construction should never yield worse security guarantees than not applying it assuming that applying the PRF does not reduce entropy. We believe there is always merit in analyzing protocols specifically. However, this construction is generic so the analyses of many protocols will still hold even if this proposed construction is incorporated.

The proposed construction cannot provide any guarantees of security if the CSPRNG state is cloned due to the virtual machine snapshots or process forking (see [MAFS2017]). Thus tag1 SHOULD incorporate all available information about the environment, such as process attributes, virtual machine user information, etc.
8. Comparison to RFC 6979

The construction proposed herein has similarities with that of RFC 6979 [RFC6979]: both of them use private keys to seed a DRBG. Section 3.3 of RFC 6979 recommends deterministically instantiating an instance of the HMAC DRBG pseudorandom number generator, described in [SP80090A] and Annex D of [X962], using the private key sk as the entropy_input parameter and H(m) as the nonce. The construction G'(n) provided herein is similar, with such difference that a key derived from G(n) and H(Sig(sk, tag1)) is used as the entropy input and tag2 is the nonce.

However, the semantics and the security properties obtained by using these two constructions are different. The proposed construction aims to improve CSPRNG usage such that certain trusted randomness would remain even if the CSPRNG is completely broken. Using a signature scheme which requires entropy sources according to RFC 6979 is intended for different purposes and does not assume possession of any entropy source - even an unstable one. For example, if in a certain system all private key operations are performed within an HSM, then the differences will manifest as follows: the HMAC DRBG construction of RFC 6979 may be implemented inside the HSM for the sake of signature generation, while the proposed construction would assume calling the signature implemented in the HSM.

9. Normative References


Authors' Addresses
Abstract

An Oblivious Pseudorandom Function (OPRF) is a two-party protocol for computing the output of a PRF. One party (the server) holds the PRF secret key, and the other (the client) holds the PRF input. The 'obliviousness' property ensures that the server does not learn anything about the client’s input during the evaluation. The client should also not learn anything about the server’s secret PRF key. Optionally, OPRFs can also satisfy a notion 'verifiability' (VOPRF). In this setting, the client can verify that the server’s output is indeed the result of evaluating the underlying PRF with just a public key. This document specifies OPRF and VOPRF constructions instantiated within prime-order groups, including elliptic curves.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

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."

This Internet-Draft will expire on September 12, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.
This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction ........................................... 3
   1.1. Terminology ....................................... 4
   1.2. Requirements ..................................... 5
2. Background ............................................. 5
3. Security Properties .................................... 6
4. OPRF Protocol .......................................... 6
   4.1. Protocol correctness ............................... 8
   4.2. Instantiations of GG ............................... 8
   4.3. OPRF algorithms .................................. 9
       4.3.1. OPRF_Setup ................................... 9
       4.3.2. OPRF_Blind .................................. 10
       4.3.3. OPRF_Sign ................................... 10
       4.3.4. OPRF_Unblind ................................ 11
       4.3.5. OPRF_Finalize ................................ 11
   4.4. VOPRF algorithms ................................ 11
       4.4.1. VOPRF_Setup .................................. 12
       4.4.2. VOPRF_Blind .................................. 12
       4.4.3. VOPRF_Sign .................................. 13
       4.4.4. VOPRF_Unblind ................................ 13
       4.4.5. VOPRF_Finalize ................................ 13
   4.5. Utility algorithms ................................ 14
       4.5.1. bin2scalar .................................... 14
   4.6. Efficiency gains with pre-processing and additive blinding .................................. 14
       4.6.1. OPRF_Preprocess ................................ 15
       4.6.2. OPRF_Blind .................................... 15
       4.6.3. OPRF_Unblind .................................. 16
5. NIZK Discrete Logarithm Equality Proof ...................... 16
   5.1. DLEQ_Generate ..................................... 16
   5.2. DLEQ_Verify ....................................... 17
6. Batched VOPRF evaluation ................................ 17
   6.1. Batched DLEQ algorithms ............................ 18
       6.1.1. Batched_DLEQ_Generate .......................... 18
       6.1.2. Batched_DLEQ_Verify ............................ 19
   6.2. Modified protocol execution ........................ 20
   6.3. PRNG and resampling ............................... 20
A pseudorandom function (PRF) $F(k, x)$ is an efficiently computable function with secret key $k$ on input $x$. Roughly, $F$ is pseudorandom if the output $y = F(k, x)$ is indistinguishable from uniformly sampling any element in $F$’s range for random choice of $k$. An oblivious PRF (OPRF) is a two-party protocol between a prover $P$ and verifier $V$ where $P$ holds a PRF key $k$ and $V$ holds some input $x$. The protocol allows both parties to cooperate in computing $F(k, x)$ with $P$’s secret key $k$ and $V$’s input $x$ such that: $V$ learns $F(k, x)$ without learning anything about $k$; and $P$ does not learn anything about $x$. A Verifiable OPRF (VOPRF) is an OPRF wherein $P$ can prove to $V$ that $F(k, x)$ was computed using key $k$, which is bound to a trusted public key $Y = kG$. Informally, this is done by presenting a non-interactive zero-knowledge (NIZK) proof of equality between $(G, Y)$ and $(Z, M)$, where $Z = kM$ for some point $M$.

OPRFs have been shown to be useful for constructing: password-protected secret sharing schemes [JKK14]; privacy-preserving password stores [SJRS17]; and password-authenticated key exchange or PAKE [OPAQUE]. VOPRFs are useful for producing tokens that are verifiable by $V$. This may be needed, for example, if $V$ wants assurance that $P$ did not use a unique key in its computation, i.e., if $V$ wants key consistency from $P$. This property is necessary in some applications, e.g., the Privacy Pass protocol [PrivacyPass], wherein this VOPRF is used to generate one-time authentication tokens to bypass CAPTCHA challenges. VOPRFs have also been used for password-protected secret sharing schemes e.g. [JHKX16].

This document introduces an OPRF protocol built in prime-order groups, applying to finite fields of prime-order and also elliptic curve (EC) settings. The protocol has the option of being extended...
to a VOPRF with the addition of a NIZK proof for proving discrete log equality relations. This proof demonstrates correctness of the computation using a known public key that serves as a commitment to the server's secret key. In the EC setting, we will refer to the protocol as ECOPRF (or ECVOPRF if verifiability is concerned). The document describes the protocol, its security properties, and provides preliminary test vectors for experimentation. The rest of the document is structured as follows:

- Section 2: Describe background, related work, and use cases of OPRF/VOPRF protocols.
- Section 3: Discuss security properties of OPRFs/VOPRFs.
- Section 4: Specify an authentication protocol from OPRF functionality, based in prime-order groups (with an optional verifiable mode). Algorithms are stated formally for OPRFs in Section 4.3 and for VOPRFs in Section 4.4.
- Section 5: Specify the NIZK discrete logarithm equality (DLEQ) construction used for constructing the VOPRF protocol.
- Section 6: Specifies how the DLEQ proof mechanism can be batched for multiple VOPRF invocations, and how this changes the protocol execution.
- Section 7: Considers explicit instantiations of the protocol in the elliptic curve setting.
- Section 8: Discusses the security considerations for the OPRF and VOPRF protocol.
- Section 9: Discusses some existing applications of OPRF and VOPRF protocols.
- Appendix A: Specifies test vectors for implementations in the elliptic curve setting.

1.1. Terminology

The following terms are used throughout this document.

- PRF: Pseudorandom Function.
- OPRF: Oblivious PRF.
- VOPRF: Verifiable Oblivious Pseudorandom Function.
ECVOPRF: A VOPRF built on Elliptic Curves.

Verifier (V): Protocol initiator when computing F(k, x).

Prover (P): Holder of secret key k.

NIZK: Non-interactive zero knowledge.

DLEQ: Discrete Logarithm Equality.

1.2. Requirements

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 [RFC2119].

2. Background

OPRFs are functionally related to RSA-based blind signature schemes, e.g., [ChaumBlindSignature]. Briefly, a blind signature scheme works as follows. Let m be a message to be signed by a server. It is assumed to be a member of the RSA group. Also, let N be the RSA modulus, and e and d be the public and private keys, respectively. A prover P and verifier V engage in the following protocol given input m.

1. V generates a random blinding element r from the RSA group, and compute $m' = m^r \pmod{N}$. Send $m'$ to the P.

2. P uses $m'$ to compute $s' = (m')^d \pmod{N}$, and sends $s'$ to the V.

3. V removes the blinding factor r to obtain the original signature as $s = (s')^{r^{-1}} \pmod{N}$.

By the properties of RSA, s is clearly a valid signature for m. OPRF protocols can be used to provide a symmetric equivalent to blind signatures. Essentially the client learns $y = \text{PRF}(k,x)$ for some input $x$ of their choice, from a server that holds $k$. Since the security of an OPRF means that $x$ is hidden in the interaction, then the client can later reveal $x$ to the server along with $y$.

The server can verify that $y$ is computed correctly by recomputing the PRF on $x$ using $k$. In doing so, the client provides knowledge of a ‘signature’ $y$ for their value $x$. However, the verification procedure is symmetric since it requires knowledge of $k$. This is discussed more in the following section.
3. Security Properties

The security properties of an OPRF protocol with functionality \( y = F(k, x) \) include those of a standard PRF. Specifically:

- Given value \( x \), it is infeasible to compute \( y = F(k, x) \) without knowledge of \( k \).
- The output distribution of \( y = F(k, x) \) is indistinguishable from the uniform distribution in the domain of the function \( F \).

Additionally, we require the following additional properties:

- Non-malleable: Given \( (x, y = F(k, x)) \), \( V \) must not be able to generate \( (x', y') \) where \( x' \neq x \) and \( y' = F(k, x') \).
- Oblivious: \( P \) must learn nothing about \( V \)'s input, and \( V \) must learn nothing about \( P \)'s private key.
- Unlinkable: If \( V \) reveals \( x \) to \( P \), \( P \) cannot link \( x \) to the protocol instance in which \( y = F(k, x) \) was computed.

Optionally, for any protocol that satisfies the above properties, there is an additional security property:

- Verifiable: \( V \) must only complete execution of the protocol if it can successfully assert that \( P \) used its secret key \( k \).

In practice, the notion of verifiability requires that \( P \) commits to the key \( k \) before the actual protocol execution takes place. Then \( V \) verifies that \( P \) has used \( k \) in the protocol using this commitment.

4. OPRF Protocol

In this section we describe the OPRF protocol. Let \( GG \) be a prime-order additive subgroup, with two distinct hash functions \( H_1 \) and \( H_2 \), where \( H_1 \) maps arbitrary input onto \( GG \) and \( H_2 \) maps arbitrary input to a fixed-length output, e.g., SHA256. All hash functions in the protocol are modelled as random oracles. Let \( L \) be the security parameter. Let \( k \) be the prover's (\( P \)) secret key, and \( Y = kG \) be its corresponding 'public key' for some generator \( G \) taken from the group \( GG \). This public key is also referred to as a commitment to the key \( k \). Let \( x \) be the verifier's (\( V \)) input to the OPRF protocol. (Commonly, it is a random \( L \)-bit string, though this is not required.)

The OPRF protocol begins with \( V \) blinding its input for the signer such that it appears uniformly distributed \( GG \). The latter then applies its secret key to the blinded value and returns the result.
To finish the computation, V then removes its blind and hashes the result using $H_2$ to yield an output. This flow is illustrated below.

Verifier  Prover
------------------------------------
  r <-$ GG
  M = rH_1(x)  

--------->
  Z = kM  

[D = DLEQ_Generate(k,G,Y,M,Z)]

Z[,D]  <--------

[b = DLEQ_Verify(G,Y,M,Z,D)]

N = Zr^(-1)

Output $H_2(x, N)$ [if b=1, else "error"]

Steps that are enclosed in square brackets (DLEQ_Generate and DLEQ_Verify) are optional for achieving verifiability. These are described in Section 5. In the verifiable mode, we assume that P has previously committed to their choice of key k with some values (G,Y=kG) and these are publicly known by V. Notice that revealing (G,Y) does not reveal k by the well-known hardness of the discrete log problem.

Strictly speaking, the actual PRF function that is computed is:

$$F(k, x) = N = kH_1(x)$$

It is clear that this is a PRF $H_1(x)$ maps x to a random element in GG, and GG is cyclic. This output is computed when the client computes $Zr^(-1)$ by the commutativity of the multiplication. The client finishes the computation by outputting $H_2(x,N)$. Note that the output from P is not the PRF value because the actual input x is blinded by r.

This protocol may be decomposed into a series of steps, as described below:

- **OPRF_Setup(l):** Generate an integer k of sufficient bit-length l and output k.
- **OPRF_Blind(x):** Compute and return a blind, r, and blinded representation of x in GG, denoted M.
- **OPRF_Sign(k,M,h):** Sign input M using secret key k to produce Z, the input h is optional and equal to the cofactor of an elliptic curve. If h is not provided then it defaults to 1.
OPRF Unblind(r, Z): Unblind blinded signature Z with blind r, yielding N and output N.

OPRF_Finalize(x, N): Finalize N to produce the output \( H_2(x, N) \).

For verifiability we modify the algorithms of VOPRF_Setup, VOPRF_Sign and VOPRF_Unblind to be the following:

VOPRF_Setup(l): Generate an integer \( k \) of sufficient bit-length \( l \) and output \( (k, (G, Y)) \) where \( Y = kG \) for some generator \( G \) in GG.

VOPRF_Sign(k, (G, Y), M, h): Sign input M using secret key k to produce Z. Generate a NIZK proof \( D = DLEQ_{Generate}(k, G, Y, M, Z) \), and output \( (Z, D) \). The optional cofactor \( h \) can also be provided as in OPRF_Sign.

VOPRF_Unblind(r, G, Y, M, (Z, D)): Unblind blinded signature Z with blind r, yielding N. Output N if \( 1 = DLEQ\_{Verify}(G, Y, M, Z, D) \). Otherwise, output "error".

We leave the rest of the OPRF algorithms unmodified. When referring explicitly to VOPRF execution, we replace ‘OPRF’ in all method names with ‘VOPRF’.

4.1. Protocol correctness

Protocol correctness requires that, for any key \( k \), input \( x \), and \( (r, M) = OPRF\_Blind(x) \), it must be true that:

\[ OPRF\_Finalize(x, OPRF\_Unblind(r, M, OPRF\_Sign(k, M))) = H_2(x, F(k, x)) \]

with overwhelming probability. Likewise, in the verifiable setting, we require that:

\[ VOPRF\_Finalize(x, VOPRF\_Unblind(r, (G, Y), M, (VOPRF\_Sign(k, (G, Y), M)))) = H_2(x, F(k, x)) \]

with overwhelming probability, where \( (r, M) = VOPRF\_Blind(x) \).

4.2. Instantiations of GG

As we remarked above, GG is a subgroup with associated prime-order \( p \). While we choose to write operations in the setting where GG comes equipped with an additive operation, we could also define the operations in the multiplicative setting. In the multiplicative setting we can choose GG to be a prime-order subgroup of a finite field \( FF_p \). For example, let \( p \) be some large prime (e.g. > 2048 bits) where \( p = 2q + 1 \) for some other prime \( q \). Then the subgroup of squares of \( FF_p \) (elements \( u^2 \) where \( u \) is an element of \( FF_p \)) is
cyclic, and we can pick a generator of this subgroup by picking \( g \) from \( \text{FF}_p \) (ignoring the identity element).

For practicality of the protocol, it is preferable to focus on the cases where \( GG \) is an additive subgroup so that we can instantiate the OPRF in the elliptic curve setting. This amounts to choosing \( GG \) to be a prime-order subgroup of an elliptic curve over base field \( \text{GF}(p) \) for prime \( p \). There are also other settings where \( GG \) is a prime-order subgroup of an elliptic curve over a base field of non-prime order, these include the work of Ristretto [RISTRETTO] and Decaf [DECAF].

We will use \( p > 0 \) generally for constructing the base field \( \text{GF}(p) \), not just those where \( p \) is prime. To reiterate, we focus only on the additive case, and so we focus only on the cases where \( \text{GF}(p) \) is indeed the base field.

### 4.3. OPRF algorithms

This section provides algorithms for each step in the OPRF protocol. We describe the VOPRF analogues in Section 4.4. We provide generic utility algorithms in Section 4.5.

1. \( P \) samples a uniformly random key \( k \leftarrow \{0,1\}^l \) for sufficient length \( l \), and interprets it as an integer.

2. \( V \) computes \( X = H_1(x) \) and a random element \( r \) (blinding factor) from \( \text{GF}(p) \), and computes \( M = rX \).

3. \( V \) sends \( M \) to \( P \).

4. \( P \) computes \( Z = kM = rkX \).

5. In the elliptic curve setting, \( P \) multiplies \( Z \) by the cofactor (denoted \( h \)) of the elliptic curve.

6. \( P \) sends \( Z \) to \( V \).

7. \( V \) unblinds \( Z \) to compute \( N = r^{-(1)}Z = kX \).

8. \( V \) outputs the pair \( H_2(x, N) \).

#### 4.3.1. OPRF_Setup
Input:

1: Some suitable choice of key-length (e.g. as described in {{NIST}}).

Output:

k: A key chosen from \(\{0,1\}^l\) and interpreted as an integer value.

Steps:

1. Sample \(k_{\text{bin}} \leftarrow \{0,1\}^l\)
2. Output \(k \leftarrow \text{bin2scalar}(k_{\text{bin}}, l)\)

4.3.2. OPRF_Blind

Input:

x: V’s PRF input.

Output:

r: Random scalar in \([1, p - 1]\).
M: Blinded representation of x using blind r, an element in GG.

Steps:

1. \(r \leftarrow \text{GF}(p)\)
2. \(M := rH_1(x)\)
3. Output \((r, M)\)

4.3.3. OPRF_Sign

Input:

k: Signer secret key.
M: An element in GG.
h: optional cofactor (defaults to 1).

Output:

Z: Scalar multiplication of the point M by k, element in GG.

Steps:

1. \(Z := kM\)
2. \(Z \leftarrow hZ\)
3. Output Z
4.3.4. OPRF_Unblind

Input:

r: Random scalar in \([1, p - 1]\).
Z: An element in GG.

Output:

N: Unblinded signature, element in GG.

Steps:

1. \(N := (1/r)Z\)
2. Output N

4.3.5. OPRF_Finalize

Input:

x: PRF input string.
N: An element in GG.

Output:

y: Random element in \(\{0,1\}^L\).

Steps:

1. \(y := H_2(x, N)\)
2. Output y

4.4. VOPRF algorithms

The steps in the VOPRF setting are written as:

1. P samples a uniformly random key \(k \leftarrow \{0,1\}^l\) for sufficient length \(l\), and interprets it as an integer.

2. P commits to \(k\) by computing \((G, Y) = Y=kG\) and where \(G\) is a generator of GG. P makes \((G, Y)\) publicly available.

3. V computes \(X = H_1(x)\) and a random element \(r\) (blinding factor) from \(\text{GF}(p)\), and computes \(M = rX\).

4. V sends \(M\) to P.

5. P computes \(Z = kM = rkX\), and \(D = \text{DLEQ}_\text{Generate}(k,G,Y,M,Z)\).
6. P sends \((Z, D)\) to V.

7. V ensures that \(1 = \text{DLEQ\_Verify}(G,Y,M,Z,D)\). If not, V outputs an error.

8. V unblinds \(Z\) to compute \(N = r^{(-1)Z} = kX\).

9. V outputs the pair \(H_2(x, N)\).

4.4.1. VOPRF\_Setup

Input:

\(G\): Public generator of GG.

\(l\): Some suitable choice of key-length (e.g. as described in \{{NIST}\}).

Output:

\(k\): A key chosen from \(\{0,1\}^l\) and interpreted as an integer value.

\((G,Y)\): A pair of curve points, where \(Y = kG\).

Steps:

1. \(k \leftarrow \text{OPRF\_Setup}(l)\)
2. \(Y := kG\)
3. Output \((k, (G,Y))\)

4.4.2. VOPRF\_Blind

Input:

\(x\): V’s PRF input.

Output:

\(r\): Random scalar in \([1, p - 1]\).

\(M\): Blinded representation of \(x\) using blind \(r\), an element in GG.

Steps:

1. \(r \leftarrow \$ \text{GF}(p)\)
2. \(M := rH_1(x)\)
3. Output \((r, M)\)
4.4.3. VOPRF_Sign

Input:

k: Signer secret key.
G: Public generator of group GG.
Y: Signer public key (= kG).
M: An element in GG.
h: optional cofactor (defaults to 1).

Output:

Z: Scalar multiplication of the point M by k, element in GG.
D: DLEQ proof that log_G(Y) == log_M(Z).

Steps:

1. Z := kM
2. Z <- hZ
3. D = DLEQ_Generate(k,G,Y,M,Z)
4. Output (Z, D)

4.4.4. VOPRF_Unblind

Input:

r: Random scalar in [1, p - 1].
G: Public generator of group GG.
Y: Signer public key.
M: Blinded representation of x using blind r, an element in GG.
Z: An element in GG.
D: D = DLEQ_Generate(k,G,Y,M,Z).

Output:

N: Unblinded signature, element in GG.

Steps:

1. N := (1/r)Z
2. If 1 = DLEQ_Verify(G,Y,M,Z,D), output N
3. Output "error"

4.4.5. VOPRF_Finalize
Input:

x: PRF input string.
N: An element in GG, or "error".

Output:

y: Random element in \(\{0,1\}^L\), or "error"

Steps:

1. If N == "error", output "error".
2. \(y := H_2(x, N)\)
3. Output y

4.5. Utility algorithms

4.5.1. bin2scalar

This algorithm converts a binary string to an integer modulo p.

Input:

s: binary string (little-endian)
l: length of binary string
p: modulus

Output:

z: An integer modulo p

Steps:

1. \(sVec \leftarrow \text{vec}(s)\) (converts s to a column vector of dimension l)
2. \(p2Vec \leftarrow (2^0, 2^1, \ldots, 2^{(l-1)})\) (row vector of dimension l)
3. \(z \leftarrow p2Vec \times sVec \pmod{p}\)
4. Output z

4.6. Efficiency gains with pre-processing and additive blinding

In the [OPAQUE] draft, it is noted that it may be more efficient to use additive blinding rather than multiplicative if the client can preprocess some values. For example, computing \(rH_1(x)\) is an example of multiplicative blinding. A valid way of computing additive blinding would be to instead compute \(H_1(x) + rG\), where G is the common generator for the group.
If the client preprocesses values of the form \( rG \), then computing \( H_1(x) + rG \) is more efficient than computing \( rH_1(x) \) (one addition against \( \log_2(r) \)). Therefore, it may be advantageous to define the OPRF and VOPRF protocols using additive blinding rather than multiplicative blinding. In fact the only algorithms that need to change are OPRF_Blind and OPRF_Unblind (and similarly for the VOPRF variants).

We define the additive blinding variants of the above algorithms below along with a new algorithm OPRF_Preprocess that defines how preprocessing is carried out. The equivalent algorithms for VOPRF are almost identical and so we do not redefine them here. Notice that the only computation that changes is for \( V \), the necessary computation of \( P \) does not change.

4.6.1. OPRF_Preprocess

Input:

\( G \): Public generator of \( GG \)

Output:

\( r \): Random scalar in \([1, p-1]\)
\( rG \): An element in \( GG \).
\( rY \): An element in \( GG \).

Steps:

1. \( r \leftarrow \text{GF}(p) \)
2. Output \((r, rG, rY)\)

4.6.2. OPRF_Blind

Input:

\( x \): \( V \)'s PRF input.
\( rG \): Preprocessed element of \( GG \).

Output:

\( M \): Blinded representation of \( x \) using blind \( r \), an element in \( GG \).

Steps:

1. \( M := H_1(x) + rG \)
2. Output \( M \)
4.6.3. OPRF_Unblind

Input:

rY: Preprocessed element of GG.
M: Blinded representation of x using rG, an element in GG.
Z: An element in GG.

Output:

N: Unblinded signature, element in GG.

Steps:

1. \( N := Z - rY \)
2. Output N

Notice that OPRF_Unblind computes \((Z - rY) = k(H_1(x) + rG) - rkG = kH_1(x)\) by the commutativity of scalar multiplication in GG. This is the same output as in the original OPRF_Unblind algorithm.

5. NIZK Discrete Logarithm Equality Proof

For the VOPRF protocol we require that V is able to verify that P has used its private key \( k \) to evaluate the PRF. We can do this by showing that the original commitment \((G, Y)\) output by VOPRF_Setup(l) satisfies \( \log_G(Y) = \log_M(Z) \) where \( Z \) is the output of VOPRF_Sign\((k, (G, Y), M)\).

This may be used, for example, to ensure that P uses the same private key for computing the VOPRF output and does not attempt to "tag" individual verifiers with select keys. This proof must not reveal the P’s long-term private key to V.

Consequently, this allows extending the OPRF protocol with a (non-interactive) discrete logarithm equality (DLEQ) algorithm built on a Chaum-Pedersen \([ChaumPedersen]\) proof. This proof is divided into two procedures: DLEQ_Generate and DLEQ_Verify. These are specified below.

5.1. DLEQ_Generate
Input:

k: Signer secret key.
G: Public generator of GG.
Y: Signer public key (= kG).
M: An element in GG.
Z: An element in GG.
H_3: A hash function from GG to \{0,1\}^L, modelled as a random oracle.

Output:

D: DLEQ proof \( (c, s) \).

Steps:

1. \( r \leftarrow GF(p) \)
2. \( A := rG \) and \( B := rM \).
3. \( c \leftarrow H_3(G,Y,M,Z,A,B) \)
4. \( s := (r - ck) \pmod p \)
5. Output \( D := (c, s) \)

5.2. DLEQ_Verify

Input:

G: Public generator of GG.
Y: Signer public key.
M: An element in GG.
Z: An element in GG.
D: DLEQ proof \( (c, s) \).

Output:

True if \( \log_G(Y) = \log_M(Z) \), False otherwise.

Steps:

1. \( A' := (sG + cY) \)
2. \( B' := (sM + cZ) \)
3. \( c' \leftarrow H_3(G,Y,M,Z,A',B') \)
4. Output \( c = c' \)

6. Batched VOPRF evaluation

Common applications (e.g. [PrivacyPass]) require V to obtain multiple PRF evaluations from P. In the VOPRF case, this would also require generation and verification of a DLEQ proof for each \( Z_i \) received by V. This is costly, both in terms of computation and
communication. To get around this, applications use a ‘batching’ procedure for generating and verifying DLEQ proofs for a finite number of PRF evaluation pairs \((M_i, Z_i)\). For \(n\) PRF evaluations:

- Proof generation is slightly more expensive from \(2n\) modular exponentiations to \(2n+2\).
- Proof verification is much more efficient, from \(4m\) modular exponentiations to \(2n+4\).
- Communications falls from \(2n\) to \(2\) group elements.

Therefore, since \(P\) is usually a powerful server, we can tolerate a slight increase in proof generation complexity for much more efficient communication and proof verification.

In this section, we describe algorithms for batching the DLEQ generation and verification procedure. For these algorithms we require a pseudorandom generator PRNG: \((0,1)^a \times \mathbb{Z}_p \rightarrow (0,1)^b)^n\) that takes a seed of length \(a\) and an integer \(n\) as input, and outputs \(n\) elements in \(0,1)^b\).

6.1. Batched DLEQ algorithms

6.1.1. Batched_DLEQ_Generate
Input:

- **k**: Signer secret key.
- **G**: Public generator of group \( GG \).
- **Y**: Signer public key \( (= kG) \).
- **n**: Number of PRF evaluations.
- \([M_i]\): An array of points in \( GG \) of length \( n \).
- \([Z_i]\): An array of points in \( GG \) of length \( n \).
- **PRNG**: A pseudorandom generator of the form above.
- **salt**: An integer salt value for each PRNG invocation
- **info**: A string value for splitting the domain of the PRNG
- **H_4**: A hash function from \( GG^{(2n+2)} \) to \( \{0,1\}^a \), modelled as a random oracle.

Output:

- **D**: DLEQ proof \((c, s)\).

Steps:

1. \( seed \leftarrow H_4(G, Y, [M_i, Z_i]) \)
2. \( d_1, \ldots, d_n \leftarrow \text{PRNG}(seed, salt, info, n) \)
3. \( c_1, \ldots, c_n := \text{(int)}d_1, \ldots, \text{(int)}d_n \)
4. \( M := c_1M_1 + \ldots + c_nM_n \)
5. \( Z := c_1Z_1 + \ldots + c_nZ_n \)
6. Output \( D \leftarrow \text{DLEQ\_Generate}(k, G, Y, M, Z) \)

6.1.2. **Batched\_DLEQ\_Verify**

Input:

- **G**: Public generator of group \( GG \).
- **Y**: Signer public key.
- \([M_i]\): An array of points in \( GG \) of length \( n \).
- \([Z_i]\): An array of points in \( GG \) of length \( n \).
- **D**: DLEQ proof \((c, s)\).

Output:

True if \( \log_G(Y) = \log_{(M_i)}(Z_i) \) for each \( i \) in \( 1 \ldots n \), False otherwise.

Steps:

1. \( seed \leftarrow H_4(G, Y, [M_i, Z_i]) \)
2. \( d_1, \ldots, d_n \leftarrow \text{PRNG}(seed, salt, info, n) \)
3. \( c_1, \ldots, c_n := \text{(int)}d_1, \ldots, \text{(int)}d_n \)
4. \( M := c_1M_1 + \ldots + c_nM_n \)
5. \( Z := c_1Z_1 + \ldots + c_nZ_n \)
6. Output \( \text{DLEQ\_Verify}(G, Y, M, Z, D) \)
6.2. Modified protocol execution

The VOPRF protocol from Section 4 changes to allow specifying multiple blinded PRF inputs \([M_i]\) for \(i\) in \(1\ldots n\). Then \(P\) computes the array \([Z_i]\) and replaces DLEQ_Generate with Batched_DLEQ_Generate over these arrays. The same applies to the algorithm VOPRF_Sign. The same applies for replacing DLEQ_Verify with Batched_DLEQ_Verify when \(V\) verifies the response from \(P\) and during the algorithm VOPRF_Verify.

6.3. PRNG and resampling

Any function that satisfies the security properties of a pseudorandom number generator can be used for computing the batched DLEQ proof. For example, SHAKE-256 [SHAKE] or HKDF-SHA256 [RFC5869] would be reasonable choices for groups that have an order of 256 bits.

We note that the PRNG outputs \(d_1,\ldots,d_n\) must be smaller than the order of the group/curve that is being used. Resampling can be achieved by increasing the value of the iterator that is used in the info field of the PRNG input.

7. Supported ciphersuites

This section specifies supported ECVOPRF group and hash function instantiations. We only provide ciphersuites in the EC setting as these provide the most efficient way of instantiating the OPRF. Our instantiation includes considerations for providing the DLEQ proofs that make the instantiation a VOPRF. Supporting OPRF operations (ECOPRF) alone can be allowed by simply dropping the relevant components. In addition, we currently only support ciphersuites demonstrating 128 bits of security.

7.1. ECVOPRF-P256-HKDF-SHA256-SSWU:

- \(G_1:\) SECP256K1 curve [SEC2]
- \(H_1:\) H2C-P256-SHA256-SSWU- [I-D.irtf-cfrg-hash-to-curve]
  * label: voprf_h2c
- \(H_2:\) SHA256
- \(H_3:\) SHA256
- \(H_4:\) SHA256
- PRNG: HKDF-SHA256
7.2. ECVOPRF-RISTRETTO-HKDF-SHA512-Elligator2:

- GG: Ristretto [RISTRETTO]
- \( H_1 \): H2C-Curve25519-SHA512-Elligator2-Clear [I-D.irtf-cfrg-hash-to-curve]
  * label: voprf_h2c
- \( H_2 \): SHA512
- \( H_3 \): SHA512
- \( H_4 \): SHA512
- PRNG: HKDF-SHA512

In the case of Ristretto, internal point representations are represented by Ed25519 [RFC7748] points. As a result, we can use the same hash-to-curve encoding as we would use for Ed25519 [I-D.irtf-cfrg-hash-to-curve]. We remark that the 'label' field is necessary for domain separation of the hash-to-curve functionality.

8. Security Considerations

Security of the protocol depends on P’s secrecy of k. Best practices recommend P regularly rotate k so as to keep its window of compromise small. Moreover, it each key should be generated from a source of safe, cryptographic randomness.

Another critical aspect of this protocol is reliance on [I-D.irtf-cfrg-hash-to-curve] for mapping arbitrary inputs \( x \) to points on a curve. Security requires this mapping be pre-image and collision resistant.

8.1. Timing Leaks

To ensure no information is leaked during protocol execution, all operations that use secret data MUST be constant time. Operations that SHOULD be constant time include: \( H_1() \) (hashing arbitrary strings to curves) and DLEQ_Generate(). [I-D.irtf-cfrg-hash-to-curve] describes various algorithms for constant-time implementations of \( H_1 \).
8.2. Hashing to curves

We choose different encodings in relation to the elliptic curve that is used, all methods are illuminated precisely in [I-D.irtf-cfrg-hash-to-curve]. In summary, we use the simplified Shallue-Woestijne-Ulas algorithm for hashing binary strings to the P-256 curve; the Icart algorithm for hashing binary strings to P384; the Elligator2 algorithm for hashing binary strings to CURVE25519 and CURVE448.

8.3. Verifiability (key consistency)

DLEQ proofs are essential to the protocol to allow V to check that P’s designated private key was used in the computation. A side effect of this property is that it prevents P from using a unique key for select verifiers as a way of "tagging" them. If all verifiers expect use of a certain private key, e.g., by locating P’s public key published from a trusted registry, then P cannot present unique keys to an individual verifier.

For this side effect to hold, P must also be prevented from using other techniques to manipulate their public key within the trusted registry to reduce client anonymity. For example, if P’s public key is rotated too frequently then this may stratify the user base into small anonymity groups (those with VOPRF_Sign outputs taken from a given key epoch). In this case, it may become practical to link VOPRF sessions for a given user and thus compromises their privacy.

Similarly, if P can publish N public keys to a trusted registry then P may be able to control presentation of these keys in such a way that V is retroactively identified by V’s key choice across multiple requests.

9. Applications

This section describes various applications of the VOPRF protocol.

9.1. Privacy Pass

This VOPRF protocol is used by Privacy Pass system to help Tor users bypass CAPTCHA challenges. Their system works as follows. Client C connects - through Tor - to an edge server E serving content. Upon receipt, E serves a CAPTCHA to C, who then solves the CAPTCHA and supplies, in response, n blinded points. E verifies the CAPTCHA response and, if valid, signs (at most) n blinded points, which are then returned to C along with a batched DLEQ proof. C stores the tokens if the batched proof verifies correctly. When C attempts to connect to E again and is prompted with a CAPTCHA, C uses one of the
unblinded and signed points, or tokens, to derive a shared symmetric key sk used to MAC the CAPTCHA challenge. C sends the CAPTCHA, MAC, and token input x to E, who can use x to derive sk and verify the CAPTCHA MAC. Thus, each token is used at most once by the system.

The Privacy Pass implementation uses the P-256 instantiation of the VOPRF protocol. For more details, see [DGSTV18].

9.2. Private Password Checker

In this application, let D be a collection of plaintext passwords obtained by prover P. For each password p in D, P computes VOPRF_Sign on H_1(p), where H_1 is as described above, and stores the result in a separate collection D'. P then publishes D' with Y, its public key. If a client C wishes to query D' for a password p', it runs the VOPRF protocol using p as input x to obtain output y. By construction, y will be the signature of p hashed onto the curve. C can then search D' for y to determine if there is a match.

Examples of such password checkers already exist, for example: [JKKX16], [JKK14] and [SJKS17].

9.2.1. Parameter Commitments

For some applications, it may be desirable for P to bind tokens to certain parameters, e.g., protocol versions, ciphersuites, etc. To accomplish this, P should use a distinct scalar for each parameter combination. Upon redemption of a token T from V, P can later verify that T was generated using the scalar associated with the corresponding parameters.

10. Acknowledgements

This document resulted from the work of the Privacy Pass team [PrivacyPass]. The authors would also like to acknowledge the helpful conversations with Hugo Krawczyk. Eli-Shaoul Khedouri provided additional review and comments on key consistency.

11. Normative References

[ChaumBlindSignature]
"Blind Signatures for Untraceable Payments", n.d.,

[ChaumPedersen]
"Wallet Databases with Observers", n.d.,

[DECAF] "Decaf, Eliminating cofactors through point compression",

[DGSTV18] "Privacy Pass, Bypassing Internet Challenges Anonymously",
popets-2018-0026.xml>.

draft-irtf-cfrg-hash-to-curve-02 (work in progress), October 2018.


[JKXX16] "Highly-Efficient and Composable Password-Protected Secret Sharing (Or, How to Protect Your Bitcoin Wallet Online)",


[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119,
DOI 10.17487/RFC2119, March 1997,

RFC 5869, DOI 10.17487/RFC5869, May 2010,

Appendix A. Test Vectors

This section includes test vectors for the ECVOPRF-P256-HKDF-SHA256 VOPRF ciphersuite, including batched DLEQ output.
P-256
X: 04b14b08f954f5b6ab1d014b1398f03881d70842acdf06194eb96a6d08186f8cb985c1c521 \ f4ee1e9290745331f7ebe89a4053de0673dc8ef14ce9f8f8226cb31
r: b72265c85b1ba42cfed7caaf00d2ccac0b1a99259b0adb5a5cf2945156a849
M: 046025a4f1f81a106c648cfe8fdcaaa2e5f7fda7a11055f8e23f1dc7e4204ab84b705043ba5c7 \ 000123e1fd058150a4d3797008f57a8b2537766d9419c7396ba5279
k: f84e197c8b712cdfe5452d2ccf52dec1bd96220ed7b9a6f6ed28c67503ae62133
Z: 043ab5cbb69oj844dc7b802b9e59126d62bc853aba1b2c339ba1cb78c034b6adc540f2f77 \ 9fc29f639edc138012f0e61960e1784973b37f864e4dc8a9b9eb80e
N: 04e86a87672d859075382e2f2ba28500d6974ba776fe230ba47e74e2be1d967654ce77f6899e \ elf374f6a0bce9044e84ad9a9cc7801022b7848031f4e442a
D: { s: faddfafa6b5d64b6357afdf856fc1e0044614ebf9dafdb4c6541c1c9e61243c5b, c: 8b403e170b56915cc18864b3ab3c2502bd8f5ca235013bc0b5a138340407b }

P-256
X: 047e8d567e854e6bdc95727d48404cb5569299e04e339b6707b2da3508eb6c23bd3d4cb4 \ 68af6ff682f9ccbd80514781d2cb21ffded628506c873eb1249
r: f222dfe530fdbfcb02eb851867bfa86da16644dfc7ce4a51e6ff83c901e15e
M: 04e2efdc73747e15e38b7a1bb90fe5e4ef964b3b8dc2cda4228f85a431420c84efca02f0f9c \ 83a8241b4572a590a949c0390db2c2d5d044ff5012051b84e7
k: fbe14d60a87e061f4435c0d7441ff8f822b5f5975760c6535eac058a24241118
Z: 049d01e1c55b032d848c93a13946b9b5dcc76752986e6d60808f93c00bdfba2ebf48eef8fe8 \ d8c91c903ad6bea3d8403b9631424a6cc543a001e1fd248192d5
N: 047238804806b04a415ca627585d1715ab5965570b30c94391a8b023f8854ac26f76c1d6ab \ bb3868a5af4bbdcd3a50ecbf69c03f33d4d7d5300b5a4b1a21ba1
D: { s: dfdf6a40d411b615d4b272cf39c4a6c8db6ac5b12044a7c0212e2bf80255b4, c: 271979a6b51d5f71719127102621fe250e3253876cfcf8dea749a3e25381997 }

Batched DLEQ (P256)
M_0: 046025a4f1f81a106c648cfe8fdcaaa2e5f7fda7a171055f8e23f1dc7e4204ab84b705043ba5c \ 7000123e1fd058150a4d3797008f57a8b2537766d9419c7396ba5279
M_1: 04e2efdc73747e15e38b7a1bb90fe5e4ef964b3b8dc2cda4228f85a431420c84efca02f0f9c \ cs8a241b4572a590a949c0390db2c2d5d044ff5012051b84e7
Z_0: 043ab5cbb69o844dc7b802b9e59126d62bc853aba1b2c339ba1cb78c034b6adc540f2f77 \ 79fc29f639edc138012f0e61960e1784973b37f864e4dc8a9b9eb80e
Z_1: 04647e1ab7946b10c1c1192dd33e2f9c9e93e85fd59399bf2f376ae859248513e0c91115 \ e48c6852d8dd173956aee7a81401c3f63a133934898d177f2a37eeb
k: f84e197c8b712cdfe5452d2ccf52dec1bd96220ed7b9a6f6ed28c67503ae62133
PRNG: HKDF-SHA256
salt: "DLEQ_PROOF"
info: an iterator i for invoking the PRNG on M_i and Z_i
D: { s: b2123044e63d47218945d73decebc9366869f3e6db4b79a0031ecf4a6c9e34, c: 3506df9008e60130fcd8f86f0b2cbe4ceb88f73f66953b1606f66039862 }
Authors’ Addresses

Alex Davidson
Cloudflare
County Hall
London, SE1 7GP
United Kingdom

Email: adavidson@cloudflare.com

Nick Sullivan
Cloudflare
101 Townsend St
San Francisco
United States of America

Email: nick@cloudflare.com

Christopher A. Wood
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: cawood@apple.com