Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 2, 2018

C. Cremers
L. Garratt
University of Oxford
S. Smyshlyaev
CryptoPro
N. Sullivan
Cloudflare
C. Wood
Apple Inc.
March 01, 2018

Randomness Improvements for Security Protocols
draft-cremers-cfrg-randomness-improvements-00

Abstract

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. This document describes a way for security protocol participants to mix their long-term private key into the entropy pool(s) from which random values are derived. This augments and improves randomness from broken or otherwise subverted CSPRNGs.

Table of Contents

1. Introduction

Randomness is a crucial ingredient for TLS and related transport security protocols. TLS in particular uses Random Number Generators (RNGs) to generate several values: session IDs, ephemeral key shares, and ClientHello and ServerHello random values. RNG failures such as the Debian bug described in [DebianBug] can lead to insecure TLS connections. RNGs may also be intentionally weakened to cause harm [DualEC]. In such cases where RNGs 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 entropy 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 entropy output 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 raw entropy output 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.

2. Randomness Wrapper

Let x be the raw entropy output of a CSPRNG. When properly instantiated, x should be indistinguishable from a random string of length |x|. 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 artificially injects randomness into a value that may be lacking entropy.

Let PRF(k, m) be a cryptographic pseudorandom function, e.g., HMAC [RFC2104], that takes as input a key k of length L and message m and produces an output of length M. For example, when using HMAC with SHA256, L and M are 256 bits. Let Sig(sk, m) be a function that computes a signature of message m given private key sk. Let G be an algorithm that generates random numbers from raw entropy, i.e., the output of a CSPRNG. Let tag be a fixed, context-dependent string. Let KDF be a key derivation function, e.g., HKDF-Extract [RFC5869] (with first argument set to nil), that extracts a key of length L suitable for cryptographic use. Lastly, let H be a cryptographic hash function that produces output of length M.

The construction works as follows: instead of using x when randomness is needed, use:

PRF(KDF(G(x) || H(Sig(sk, tag1))), tag2)

Functionally, this computes the PRF of a string (tag2) with 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. The PRF behaves in a manner that is indistinguishable from a truly random function from {0, 1}^L to {0, 1}^M assuming the key is selected at random. Thus, the security of this construction depends upon the secrecy of H(Sig(sk, tag1)) and G(x). If the signature is leaked, then security reduces to the scenario wherein the PRF provides only a wrapper to G(x).

In systems where signature computations are not cheap, these values may be precomputed in anticipation of future randomness requests. This is possible since the construction depends solely upon the CSPRNG output and private key.

Sig(sk, tag1) MUST NOT be used or exposed beyond its role in this computation. Moreover, Sig MUST be a deterministic signature function, e.g., deterministic ECDSA [RFC6979].

3. Tag Generation

Both tags SHOULD be generated such that they never collide with another accessor 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 Sig(sk, tag1). Device specific information may include, for example, a MAC address. 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 randomness were to repeat.

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" PRNG, i.e., the PRNG that generates private values, such as key shares, with:

HMAC(HKDF-Extract(nil, G(x) || Sig(sk, tag1)), tag2)

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. IANA Considerations

This document makes no request to IANA.

6. Security Considerations

A security analysis was performed by two authors of this document. Generally speaking, security depends on keeping the private key secret. If this secret is compromised, the scheme reduces to the scenario wherein the PRF provides only an outer wrapper on usual CSPRNG generation.

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.

The signature in the construction as well as in the protocol itself MUST be deterministic: if the signatures are probabilistic, then with 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 analysing protocols specifically. However, this construction is generic so the analyses of many protocols will still hold even if this proposed construction is incorporated.

7. Normative References

[DebianBug] Yilek, Scott, et al, ., "When private keys are public - Results from the 2008 Debian OpenSSL vulnerability", n.d., <https://pdfs.semanticscholar.org/fcf9/fe0946c20e936b507c023bbf89160cc995b9.pdf>.

[DualEC] Bernstein, Daniel et al, ., "Dual EC - A standardized back door", n.d., <https://projectbullrun.org/dual-ec/documents/dual-ec-20150731.pdf>.

[NAXOS] LaMacchia, Brian et al, ., "Stronger Security of Authenticated Key Exchange", n.d., <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/strongake-submitted.pdf>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <https://www.rfc-editor.org/info/rfc6979>.

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

Authors' Addresses

Cas Cremers
University of Oxford
Wolfson Building, Parks Road
Oxford
England

Email: cas.cremers@cs.ox.ac.uk

Luke Garratt
University of Oxford
Wolfson Building, Parks Road
Oxford
England

Email: luke.garratt@cs.ox.ac.uk

Stanislav Smyshlyaev
CryptoPro
18, Suschevsky val
Moscow
Russian Federation

Email: svs@cryptopro.ru

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

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 6, 2018

N. Sullivan
Cloudflare
C. Wood
Apple Inc.

March 05, 2018

Hashing to Elliptic Curves
draft-sullivan-cfrg-hash-to-curve-00

Abstract

This document specifies a number of algorithms that may be used to hash arbitrary strings to 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 6, 2018.

Table of Contents

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], and Boneh-Lynn-Shacham signatures [BLS01].

Let E be an elliptic curve over base field GF(p). In practice, efficient (polynomial-time) functions that hash arbitrary input to E can be constructed by composing a cryptographically secure hash function F1 : {0,1}^* ->GF(p) and an injection F2 : GF(p) -> E, i.e., Hash(m) = F2(F1(m)). Probabilistic constructions of Hash, e.g., the MapToGroup function described by Boneh et al. [BLS01]. Their algorithm fails with probability 2^I, where I is a tunable parameter that one can control. Another variant, dubbed the "Try and Increment" approach, was described by Boneh et al. [BLS01]. This function works by hashing input m using a standard hash function, e.g., SHA256, and then checking to see if the resulting point E(m, f(m)), for curve function f, belongs on E. This algorithm is expected to find a valid curve point after approximately two

attempts, i.e., when ctr=1, on average. (See Appendix Appendix A for a more detailed description of this algorithm.) Since the running time of the 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. Shallue and Woestijne [SWU] first introduced a deterministic algorithm that maps elements in F_{q} to an EC 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} 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].

This document specifies several algorithms for deterministically hashing onto a curve with varying properties: Icart, SWU, Simplified SWU, and Elligator2. Each algorithm conforms to a common interface, i.e., it maps an element from a base field F to a curve E. For each variant, we describe the requirements for F and 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. Algorithm Recommendations

The following table lists recommended algorithms to use for specific curves.

Curve	Algorithm
P-256	SWU Section 4.3
P-384	Icart Section 4.1
Curve25519	Elligator2 Section 4.4
Curve448	Elligator2 Section 4.4

The SWU variant from Section Section 4.2 applies to any curve. As such, this algorithm SHOULD be used if no other better alternative is known. More efficient variants and their curve requirements are shown in the table below. These MAY be used if the target curve meets the listed criteria.

Algorithm	Requirement
Icart Section 4.1	p = 2 mod 3
SWU Section 4.2	None
Simplified SWU Section 4.3	p = 3 mod 4
Elligator2 Section 4.4	p is large and there is a point of order two and j-invariant != 1728

3. Generic Interface

The generic interface for hashing to elliptic curves is as follows:

hash_to_curve(alpha)

where alpha is a message to hash onto a curve.

3.1. Utility Functions

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

- HashToBase(x): H(x)[0:log2(p) + 1], i.e., hash-truncate-reduce, where H is a cryptographic hash function, such as SHA256, and p is the prime order of base field Fp.

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

Note: We assume that HashToBase maps its input to the base field uniformly. In practice, there may be inherent biases in p, e.g., p = 2^k - 1 will have non-negligible bias in higher bits.

((TODO: expand on this problem))

4. Hashing Variants

4.1. Icart Method

The following hash_to_curve_icart(alpha) implements the Icart method from [Icart09]. This algorithm works for any curve over F_{p^n}, where p^n = 2 mod 3 (or p = 2 mod 3 and for odd n), including:

- P384
- Curve1174
- Curve448

Unsupported curves include: P224, P256, P521, and Curve25519 since, for each, p = 1 mod 3.

Mathematically, given input alpha, and A and B from E, the Icart method works as follows:

```
u = HashToBase(alpha)
x = (v^2 - b - (u^6 / 27))^(1/3) + (u^2 / 3)
y = ux + v
```

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

The following procedure implements this algorithm in a straight-line fashion. It requires knowledge of A and B, the constants from the curve Weierstrass form. It outputs a point with affine coordinates.

```
  hash_to_curve_icart(alpha)

  Input:

    alpha - value to be hashed, an octet string

  Output:

    (x, y) - a point in E

  Steps:

  1.   u = HashToBase(alpha)   // {0,1}^* -> Fp
  2.  u2 = u^2 (mod p)         // u^2
  3.  t2 = u2^2 (mod p)        // u^4
  4.  v1 = 3 * A (mod p)       // 3A
  5.  v1 = v1 - t2 (mod p)     // 3A - u^4
  6.  t1 = 6 * u (mod p)       // 6u
  7.  t3 = t1 ^ (-1) (mod p)   // modular inverse
  8.   v = v1 * t3 (mod p)     // (3A - u^4)/(6u)
  9.   x = v^2 (mod p)         // v^2
  10.  x = x - B (mod p)       // v^2 - b
  11. t1 = 27 ^ (-1) (mod p)   // 1/27
  12. t1 = t1 * u2 (mod p)     // u^4 / 27
  13. t1 = t1 * t2 (mod p)     // u^6 / 27
  14.  x = x - t1 (mod p)      // v^2 - b - u^6/27
  15. t1 = (2 * p) - 1 (mod p) // 2p - 1
  16. t1 = t1 / 3 (mod p)      // (2p - 1)/3
  17.  x = x^t1 (mod p)        // (v^2 - b - u^6/27) ^ (1/3)
  18. t2 = u2 / 3 (mod p)      // u^2 / 3
  19.  x = x + t2 (mod p)      // (v^2 - b - u^6/27) ^ (1/3) + (u^2 / 3)
  20.  y = u * x (mod p)       // ux
  21.  y = y + v (mod p)       // ux + v
  22. Output (x, y)
```

4.2. Shallue-Woestijne-Ulas Method

((TODO: write this section))

4.3. Simplified SWU Method

The following hash_to_curve_simple_swu(alpha) implements the simplfied Shallue-Woestijne-Ulas algorithm from [SimpleSWU]. This algorithm works for any curve over F_{p^n}, where p = 3 mod 4, including:

- P256

- ...

Given curve equation g(x) = x^3 + Ax + B, this algorithm works as follows:

1. t = HashToBase(alpha)
2. alpha = (-b / a) * (1 + (1 / (t^4 + t^2)))
3. beta = -t^2 * alpha
4. z = t^3 * g(alpha)
5. Output (-g * alpha) * (g * beta)

The following procedure implements this algorithm. It outputs a point with affine coordinates.

```
hash_to_curve_simple_swu(alpha)

Input:

  alpha - value to be hashed, an octet string

Output:

  (x, y) - a point in E

Steps:

1.     t = HashToBase(alpha)
2. alpha = t^2 (mod p)
3. alpha = alpha * -1 (mod p)
4. right = alpha^2 + alpha (mod p)
5. right = right^(-1) (mod p)
6. right = right + 1 (mod p)
7.  left = B * -1 (mod p)
8.  left = left / A (mod p)
9.    x2 = left * right (mod p)
10.   x3 = alpha * x2 (mod p)
11.   h2 = x2 ^ 3 (mod p)
12.   i2 = x2 * A (mod p)
13.   i2 = i2 + B (mod p)
14.   h2 = h2 + i2 (mod p)
15.   h3 = x3 ^ 3 (mod p)
16.   i3 = x3 * A (mod p)
17.   i3 = i3 + B (mod p)
18.   h3 = h3 + i3 (mod p)
19.   y1 = h2 ^ ((p + 1) // 4) (mod p)
20.   y2 = h3 ^ ((p + 1) // 4) (mod p)
21.    e = (y1 ^ 2 == h2)
22.    x = CMOV(x2, x3, e)    // If e = 1, choose x2, else choose x3
23.    y = CMOV(y1, y2, e)    // If e = 1, choose y1, else choose y2
24. Output (x, y)
```

4.4. Elligator2 Method

The following hash_to_curve_elligator2(alpha) implements the Elligator2 method from [Elligator2]. This algorithm works for any curve with a point of order 2 and j-invariant != 1728. Given curve equation f(x) = y^2 = x(x^2 + Ax + B), i.e., a Montgomery form with the point of order 2 at (0,0), this algorithm works as shown below. (Note that any curve with a point of order 2 is isomorphic to this representation.)

1. r = HashToBase(alpha)
2. If f(-A/(1+ur^2)) is square, then output f(-A/(1+ur^2))^(1/2)
3. Else, output f(-Aur^2/(1+ur^2))^(1/2)

Another way to express this algorithm is as follows:

1. r = HashToBase(alpha)
2. d = -A / (1 + ur^2)
3. e = f(d)^((p-1)/2)
4. u = ed - (1 - e)A/u

Here, e is the Legendre symbol of y = (d^3 + Ad^2 + d), which will be 1 if y is a quadratic residue (square) mod p, and -1 otherwise. (Note that raising y to ((p -1) / 2) is a common way to compute the Legendre symbol.)

The following procedure implements this algorithm.

```
hash_to_curve_elligator2(alpha)

Input:

  alpha - value to be hashed, an octet string

  u - fixed non-square value in Fp.
  f() - Curve function

Output:

  (x, y) - a point in E

Steps:

1.   r = HashToBase(alpha)
2.   r = r^2 (mod p)
3.  nu = r * u (mod p)
4.   r = nu
5.   r = r + 1 (mod p)
6.   r = r^(-1) (mod p)
7.   v = A * r (mod p)
8.   v = v * -1 (mod p)   // -A / (1 + ur^2)
9.  v2 = v^2 (mod p)
10. v3 = v * v2 (mod p)
11.  e = v3 * v (mod p)
12. v2 = v2 * A (mod p)
13.  e = v2 * e (mod p)
14.  e = e^((p - 1) / 2)  // Legendre symbol
15. nv = v * -1 (mod p)
16.  v = CMOV(v, nv, e)   // If e = 1, choose v, else choose nv
17. v2 = CMOV(0, A, e)    // If e = 1, choose 0, else choose A
18.  u = v - v2 (mod p)
19. Output (u, f(u))
```

Elligator2 can be simplified with projective coordinates.

((TODO: write this variant))

5. Curve Transformations

((TODO: write this section))

6. Cost Comparison

The following table summarizes the cost of each hash_to_curve variant. We express this cost in terms of additions (A), multiplications (M), squares (SQ), and square roots (SR).

((TODO: finish this section))

Algorithm	Cost (Operations)
hash_to_curve_icart	TODO
hash_to_curve_swu	TODO
hash_to_curve_simple_swu	TODO
hash_to_curve_elligator2	TODO

7. IANA Considerations

This document has no IANA actions.

8. Security Considerations

Each hash 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. Some variants variants are not 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.

9. 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.

10. Contributors

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

11. Normative References

[BLS01] "Short signatures from the Weil pairing", n.d., <https://iacr.org/archive/asiacrypt2001/22480516.pdf>.

[BMP00] "Provably secure password-authenticated key exchange using diffie-hellman", n.d..

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

[Elligator2] "Elligator -- Elliptic-curve points indistinguishable from uniform random strings", n.d., <https://dl.acm.org/ft_gateway.cfm?id=2516734&type=pdf>.

[ElligatorAGL] "Implementing Elligator for Curve25519", n.d., <https://www.imperialviolet.org/2013/12/25/elligator.html>.

[Icart09] "How to Hash into Elliptic Curves", n.d., <https://eprint.iacr.org/2009/226.pdf>.

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

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[SECG1] "SEC 1 -- Elliptic Curve Cryptography", n.d., <http://www.secg.org/sec1-v2.pdf>.

[SimpleSWU] "Efficient Indifferentiable Hashing into Ordinary Elliptic Curves", n.d..

[SWU] "Rational points on certain hyperelliptic curves over finite fields", n.d., <https://arxiv.org/pdf/0706.1448>.

Appendix A. 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 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 E(m, f(m)), for curve function f, belongs on E. This is detailed below.

1. ctr = 0
3. h = "INVALID"
4. While h is "INVALID" or h is EC point at infinity:
   A. CTR = I2OSP(ctr, 4)
   B. ctr = ctr + 1
   C. attempted_hash = Hash(m || CTR)
   D. h = RS2ECP(attempted_hash)
   E. 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 is a function that converts of a random 2n-octet string to an EC point as specified in Section 5.1.3 of [RFC8032].

Appendix B. Sample Code

B.1. Icart Method

The following Sage program implements hash_to_curve_icart(alpha) for P-384.

```
p = 39402006196394479212279040100143613805079739270465446667948293404 \
45721771496870329047266088258938001861606973112319
F = GF(p)
A = p - 3
B = 0xb3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875a \
c656398d8a2ed19d2a85c8edd3ec2aef
q = 39402006196394479212279040100143613805079739270465446667946905279 \
27659399113263569398956308152294913554433653942643
E = EllipticCurve([F(A), F(B)])
g = E(0xaa87ca22be8b05378eb1c71ef320ad746e1d3b628ba79b9859f741e082542a \
385502f25dbf55296c3a545e3872760ab7, \
    0x3617de4a96262c6f5d9e98bf9292dc29f8f41dbd289a147ce9da3113b5f0b8c0 \
0a60b1ce1d7e819d7a431d7c90ea0e5f)
E.set_order(q)

def icart(u):
  u = F(u)
```

```
  v = (3*A - u^4)//(6*u)
  x = (v^2 - B - u^6/27)^((2*p-1)//3) + u^2/3
  y = u*x + v
  return E(x, y)

def icart_straight(u):
    u = F(u)
    u2 = u ^ 2
    t2 = u2 ^ 2
    assert t2 == u^4

    v1 = 3 * A
    v1 = v1 - t2
    t1 = 6 * u
    t3 = t1 ^ (-1)
    v = v1 * t3
    assert v == (3 * A - u^4) // (6 * u)

    x = v ^ 2
    x = x - B
    assert x == (v^2 - B)

    t1 = F(27) ^ (-1)
    t1 = t1 * u2
    t1 = t1 * t2
    assert t1 == ((u^6) / 27)

    x = x - t1
    t1 = (2 * p) - 1
    t1 = t1 / 3
    assert t1 == ((2*p) - 1) / 3

    x = x ^ t1

    t2 = u2 / 3
    x = x + t2
    y = u * x
    y = y + v
    return E(x, y)
```

B.2. Shallue-Woestijne-Ulas Method

((TODO: write this section))

B.3. Simplified SWU Method

The following Sage program implements hash_to_curve_swu(alpha) for P-256.

```
p = 115792089210356248762697446949407573530086143415290314195533631308 \
867097853951
F = GF(p)
A = F(p - 3)
B = F(ZZ("5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2 \
604b", 16))
E = EllipticCurve([A, B])

def simple_swu(alpha):
    t = F(alpha)

    alpha = -(t^2)
    frac = (1 / (alpha^2 + alpha))
    x2 = (-B / A) * (1 + frac)

    x3 = alpha * x2
    h2 = x2^3 + A * x2 + B
    h3 = x3^3 + A * x3 + B

    if is_square(h2):
        return E(x2, h2^((p + 1) // 4))
    else:
        return E(x3, h3^((p + 1) // 4))

def simple_swu_straight(alpha):
    t = F(alpha)

    alpha = t^2
    alpha = alpha * -1

    right = alpha^2 + alpha
    right = right^(-1)
    right = right + 1

    left = B * -1
    left = left / A

    x2 = left * right
    x3 = alpha * x2

    h2 = x2 ^ 3
    i2 = x2 * A
    i2 = i2 + B
```

```
    h2 = h2 + i2

    h3 = x3 ^ 3
    i3 = x3 * A
    i3 = i3 + B
    h3 = h3 + i3

    y1 = h2^((p + 1) // 4)
    y2 = h3^((p + 1) // 4)

    # Is it square?
    e = y1^2 == h2

    x = x2
    if e != 1:
        x = x3

    y = y1
    if e != 1:
        y = y2

    return E(x, y)
```

B.4. Elligator2 Method

The following Sage program implements hash_to_curve_elligator2(alpha) for Curve25519.

```
p = 2**255 - 19
F = GF(p)
A = 486662
B = 1
E = EllipticCurve(F, [0, A, 0, 1, 0])

def curve25519(x):
    return x^3 + (A * x^2) + x

def elligator2(alpha):

    r = F(alpha)

    # u is a fixed nonsquare value, eg -1 if p==3 mod 4.
    u = F(2) # F(2)
    assert(not u.is_square())

    # If f(-A/(1+ur^2)) is square, return its square root.
    # Else, return the square root of f(-Aur^2/(1+ur^2)).
    x = -A / (1 + (u * r^2))
```

```
        y = curve25519(x)
        if y.is_square(): # is this point square?
            y = y.square_root()
        else:
            x = (-A * u * r^2) / (1 + (u * r^2))
            y = curve25519(x).square_root()

        return (x, curve25519(x))

    def elligator2_straight(alpha):
        r = F(alpha)

        r = r^2
        r = r * 2
        r = r + 1
        r = r^(-1)
        v = A * r
        v = v * -1 # d

        v2 = v^2
        v3 = v * v2
        e = v3 + v
        v2 = v2 * A
        e = v2 + e

        # Legendre symbol
        e = e^((p - 1) / 2)

        nv = v * -1
        if e != 1:
            v = nv

        v2 = 0
        if e != 1:
            v2 = A

        u = v - v2

        return (u, curve25519(u))
```

Authors' Addresses

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

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 12, 2019

A. Davidson
N. Sullivan
Cloudflare
C. Wood
Apple Inc.

March 11, 2019

Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups
draft-sullivan-cfrg-voprf-03

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.

Table of Contents

1. Introduction

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 [SJKS17]; 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. [JKKX16].

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 Section 2: Describe background, related work, and use cases of OPRF/VOPRF protocols.
- Section Section 3: Discuss security properties of OPRFs/VOPRFs.
- Section 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 Section 5: Specify the NIZK discrete logarithm equality (DLEQ) construction used for constructing the VOPRF protocol.
- Section Section 6: Specifies how the DLEQ proof mechanism can be batched for multiple VOPRF invocations, and how this changes the protocol execution.
- Section Section 7: Considers explicit instantiations of the protocol in the elliptic curve setting.
- Section Section 8: Discusses the security considerations for the OPRF and VOPRF protocol.
- Section Section 9: Discusses some existing applications of OPRF and VOPRF protocols.
- Section 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 (mod N). Send m' to the P.
2. P uses m' to compute s' = (m')^d (mod N), and sends s' to the V.
3. V removes the blinding factor r to obtain the original signature as s = (s')^(r^-1) (mod 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 = 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' != 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.

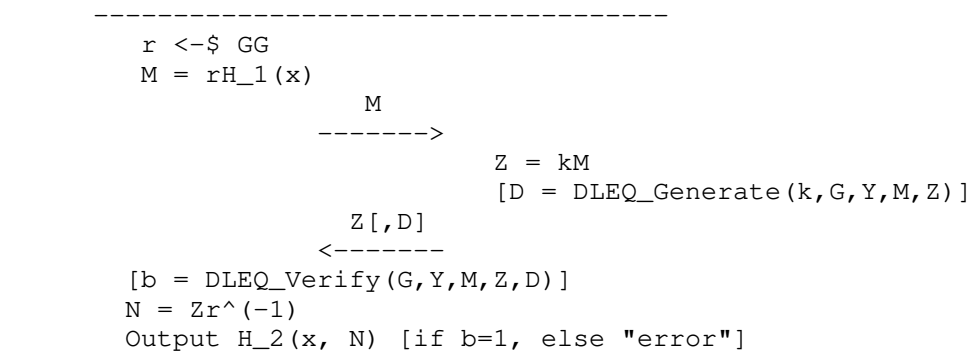
In practice, the notion of verifiability requires that P commits to the key k before the actual protocol execution takes place. Then V verifies that P has used k in the protocol using this commitment.

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.



Steps that are enclosed in square brackets (DLEQ_Generate and DLEQ_Verify) are optional for achieving verifiability. These are described in Section 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 am 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 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 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 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 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 <- {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 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:

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.

Steps:

1. Sample k_bin <-$ {0,1}^l
2. Output k <- bin2scalar(k_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 <-$ 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 <- 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 <- {0,1}^l for sufficient length l, and interprets it as an integer.

2. P commits to k by computing (G,Y) for 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 GF(p), and computes M = rX.

4. V sends M to P.

5. P computes Z = kM = rkX, and D = DLEQ_Generate(k,G,Y,M,Z).

6. P sends (Z, D) to V.

7. V ensures that 1 = 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 <- 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 <-$ 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 <- vec(s) (converts s to a column vector of dimension l)
2. p2Vec <- (2^0, 2^1, ..., 2^{l-1}) (row vector of dimension l)
3. z <- p2Vec * sVec (mod 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 <-$ 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 <-$ GF(p)
2. A := rG and B := rM.
3. c <- H_3(G,Y,M,Z,A,B)
4. s := (r - ck) (mod 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' <- 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 Zi 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 (Mi,Zi). 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 x ZZ -> ({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.
[Mi]: An array of points in GG of length n.
[Zi]: 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 <- H_4(G,Y,[Mi,Zi]))
2. d1,...dn <- PRNG(seed,salt,info,n)
3. c1,...,cn := (int)d1,...,(int)dn
4. M := c1M1 + ... + cnMn
5. Z := c1Z1 + ... + cnZn
6. Output D <- DLEQ_Generate(k,G,Y,M,Z)

6.1.2. Batched_DLEQ_Verify

Input:

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

Output:

True if log_G(Y) == log_(Mi)(Zi) for each i in 1...n, False otherwise.

Steps:

1. seed <- H_4(G,Y,[Mi,Zi]))
2. d1,...dn <- PRNG(seed,salt,info,n)
3. c1,...,cn := (int)d1,...,(int)dn
4. M := c1M1 + ... + cnMn
5. Z := c1Z1 + ... + cnZn
6. Output DLEQ_Verify(G,Y,M,Z,D)

6.2. Modified protocol execution

The VOPRF protocol from Section Section 4 changes to allow specifying multiple blinded PRF inputs [Mi] for i in 1...n. Then P computes the array [Zi] 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 d1,...,dn 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:

- GG: 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., <http://sceweb.sce.uhcl.edu/yang/teaching/csci5234WebSecurityFall2011/Chaum-blind-signatures.PDF>.

[ChaumPedersen]
"Wallet Databases with Observers", n.d., <https://chaum.com/publications/Wallet_Databases.pdf>.

[DECAF] "Decaf, Eliminating cofactors through point compression", n.d., <https://www.shiftleft.org/papers/decaf/decaf.pdf>.

[DGSTV18] "Privacy Pass, Bypassing Internet Challenges Anonymously", n.d., <https://www.degruyter.com/view/j/popets.2018.2018.issue-3/popets-2018-0026/popets-2018-0026.xml>.

[I-D.irtf-cfrg-hash-to-curve] Scott, S., Sullivan, N., and C. Wood, "Hashing to Elliptic Curves", draft-irtf-cfrg-hash-to-curve-02 (work in progress), October 2018.

[JKK14] "Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only model", n.d., <https://eprint.iacr.org/2014/650.pdf>.

[JKKX16] "Highly-Efficient and Composable Password-Protected Secret Sharing (Or, How to Protect Your Bitcoin Wallet Online)", n.d., <https://eprint.iacr.org/2016/144>.

[NIST] "Keylength - NIST Report on Cryptographic Key Length and Cryptoperiod (2016)", n.d., <https://www.keylength.com/en/4/>.

[OPAQUE] "The OPAQUE Asymmetric PAKE Protocol", n.d., <https://tools.ietf.org/html/draft-krawczyk-cfrg-opaque-01>.

[PrivacyPass] "Privacy Pass", n.d., <https://github.com/privacypass/challenge-bypass-server>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RISTRETTO] "The ristretto255 Group", n.d., <https://tools.ietf.org/html/draft-hdevalence-cfrg-ristretto-00>.

[SEC2] Standards for Efficient Cryptography Group (SECG), ., "SEC 2: Recommended Elliptic Curve Domain Parameters", n.d., <http://www.secg.org/sec2-v2.pdf>.

[SHAKE] "SHA-3 Standard, Permutation-Based Hash and Extendable-Output Functions", n.d., <https://www.nist.gov/publications/sha-3-standard-permutation-based-hash-and-extendable-output-functions?pub_id=919061>.

[SJKS17] "SPHINX, A Password Store that Perfectly Hides from Itself", n.d., <http://webee.technion.ac.il/%7Ehugo/sphinx.pdf>.

Appendix A. Test Vectors

This section includes test vectors for the ECVOPRF-P256-HKDF-SHA256 VOPRF ciphersuite, including batched DLEQ output.

```
P-256
X: 04b14b08f954f5b6ab1d014b1398f03881d70842acdf06194eb96a6d08186f8cb985c1c5521 \
    f4ee19e290745331f7eb89a4053de0673dc8ef14cfe9bf8226c6b31
r: b72265c85b1ba42cfed7caaf00d2ccac0b1a99259ba0dbb5a1fc2941526a6849
M: 046025a41f81a160c648cfe8fdcaa42e5f7da7a71055f8e23f1dc7e4204ab84b705043ba5c7 \
    000123e1fd058150a4d3797008f57a8b2537766d9419c7396ba5279
k: f84e197c8b712cdf452d2cff52dec1bd96220ed7b9a6f66ed28c67503ae62133
Z: 043ab5ccb690d844dcb780b2d9e59126d62bc853ba01b2c339ba1c1b78c03e4b6adc5402f77 \
    9fc29f639edc138012f0e61960e1784973b37f864e4dc8abbc68e0b
N: 04e8aa6792d859075821e2fba28500d6974ba776fe230ba47ef7e42be1d967654ce776f889e \
    e1f374ffa0bce904408aaa4ed8a19c6cc7801022b7848031f4e442a
D: { s: faddfaf6b5d6b4b6357adf856fc1e0044614ebf9dafdb4c6541c1c9e61243c5b,
     c: 8b403e170b56c915cc18864b3ab3c2502bd8f5ca25301bc03ab5138343040c7b }

P-256
X: 047e8d567e854e6bdc95727d48b40cbb5569299e0a4e339b6d707b2da3508eb6c238d3d4cb4 \
    68afc6ffc82fccbda8051478d1d2c9b21ffdfd628506c873ebb1249
r: f222dfe530fdbfcb02eb851867bfa8a6da1664dfc7cee4a51eb6ff83c901e15e
M: 04e2efdc73747e15e38b7a1bb90fe5e4ef964b3b8dccfda428f85a431420c84efca02f0f09c \
    83a8241b44572a059ab49c080a39d0bce2d5d0b44ff5d012b5184e7
k: fb164de0a87e601fd4435c0d7441ff822b5fa5975d0c68035beac05a82c41118
Z: 049d01e1c555bd3324e8ce93a13946b98bdcc765298e6d60808f93c00bdfba2ebf48eef8f28 \
    d8c91c903ad6bea3d840f3b9631424a6cc543a0a0e1f2d487192d5b
N: 04723880e480b60b4415ca627585d1715ab5965570d30c94391a8b023f8854ac26f76c1d6ab \
    bb38688a5affbcadad50ecbf7c93ef33ddfd735003b5a4b1a21ba14
D: { s: dfdf6ae40d141b61d5b2d72cf39c4a6c88db6ac5b12044a70c212e2bf80255b4,
     c: 271979a6b51d5f71719127102621fe250e3235867cfcf8dea749c3e253b81997 }

Batched DLEQ (P256)
M_0: 046025a41f81a160c648cfe8fdcaa42e5f7da7a71055f8e23f1dc7e4204ab84b705043ba5c\
    7000123e1fd058150a4d3797008f57a8b2537766d9419c7396ba5279
M_1: 04e2efdc73747e15e38b7a1bb90fe5e4ef964b3b8dccfda428f85a431420c84efca02f0f09\
    c83a8241b44572a059ab49c080a39d0bce2d5d0b44ff5d012b5184e7
Z_0: 043ab5ccb690d844dcb780b2d9e59126d62bc853ba01b2c339ba1c1b78c03e4b6adc5402f7\
    79fc29f639edc138012f0e61960e1784973b37f864e4dc8abbc68e0b
Z_1: 04647e1ab7946b10c1c1c92dd333e2fc9e93e85fdef5939bf2f376ae859248513e0cd91115\
    e48c6852d8dd173956aec7a81401c3f63a133934898d177f2a237eeb
k: f84e197c8b712cdf452d2cff52dec1bd96220ed7b9a6f66ed28c67503ae62133
PRNG: HKDF-SHA256
salt: "DLEQ_PROOF"
info: an iterator i for invoking the PRNG on M_i and Z_i
D: { s: b2123044e633d4721894d573decebc9366869fe3c6b4b79a00311ecfa46c9e34,
     c: 3506df9008e60130fcddf86fdb02cbfe4ceb88ff73f66953b1606f6603309862 }
```

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

Internet Research Task Force (IRTF)
Internet-Draft
Intended status: Informational
Expires: June 17, 2018

B. Viguier
Radboud University

December 14, 2017

KangarooTwelve
draft-viguier-kangarootwelve-01

Abstract

This document defines the KangarooTwelve eXtendable Output Function (XOF), a hash function with arbitrary output length. It provides an efficient and secure hashing primitive, which is able to exploit the parallelism of the implementation in a scalable way. It uses tree hashing over a round-reduced version of SHAKE128 as underlying primitive.

This document builds up on the definitions of the permutations and of the sponge construction in [FIPS 202], and is meant to serve as a stable reference and an implementation guide.

Table of Contents

1. Introduction

This document defines the KangarooTwelve eXtendable Output Function (XOF) [K12], i.e. a generalization of a hash function that can return arbitrary output length. KangarooTwelve is based on a Keccak-p permutation specified in [FIPS202] and aims at higher speed than SHAKE and SHA-3.

The SHA-3 functions process data in a serial manner and are unable to optimally exploit parallelism available in modern CPU architectures. KangarooTwelve splits the input message in fragments and applies an inner hash function F on each of them separately. It then applies F again on the concatenation of the digests. It makes use of Sakura coding for ensuring soundness of the tree hashing mode [SAKURA]. The inner hash function F is a sponge function and uses a round-reduced version of the permutation Keccak-f used in SHA-3. Its security builds up on the scrutiny that Keccak has received since its publication [KECCAK_CRYPTANALYSIS].

1.1. Conventions

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 RFC 2119 [RFC2119].

The following notations are used throughout the document:

‘...‘ denotes a string of bytes given in hexadecimal. For example, ‘0B 80‘.

|s| denotes the length of a byte string ‘s‘. For example, |‘FF FF‘| = 2.

‘00‘^b denotes a byte string consisting of the concatenation of b bytes ‘00‘. For example, ‘00‘^7 = ‘00 00 00 00 00 00 00‘.

‘00‘^0 denotes the empty byte-string.

a||b denotes the concatenation of two strings a and b. For example, ‘10‘||‘F1‘ = ‘10 F1‘

s[n:m] denotes the selection of bytes from n to m exclusive of a string s. For example, for s = ‘A5 C6 D7‘, s[0:1] = ‘A5‘ and s[1:3] = ‘C6 D7‘.

s[n:] denotes the selection of bytes from n to the end of a string s. For example, for s = ‘A5 C6 D7‘, s[0:] = ‘A5 C6 D7‘ and s[2:] = ‘D7‘.

In the following, x and y are byte strings of equal length:

x^=y denotes x takes the value x XOR y.

x & y denotes x AND y.

In the following, x and y are integers:

x+=y denotes x takes the value x + y.

x-=y denotes x takes the value x - y.

x**y denotes x multiplied by itself y times.

2. Specifications

KangarooTwelve is an eXtendable Output Function (XOF). It takes as an input a couple of byte-strings (M, C) and a positive integer L where

M byte-string, is the Message and

C byte-string, is a OPTIONAL Customization string and

L positive integer, the number of output bytes requested.

The Customization string MAY serves as domain separation. It is typically a short string such as a name or an identifier (e.g. URI, ODI...)

By default, the Customization string is the empty string. For an API does that not support a customization string input, C MUST be the empty string.

2.1. Inner function F

The inner function F makes use of the permutation Keccak-p[1600,n_r=12], i.e., a version of the permutation Keccak-f[1600] used in SHAKE and SHA-3 instances reduced to its last n_r=12 rounds and specified in FIPS 202, sections 3.3 and 3.4 [FIPS202]. KP denotes this permutation.

F is a sponge function calling this permutation KP with a rate of 168 bytes or 1344 bits. It follows that F has a capacity of 1600 - 1344 = 256 bits or 32 bytes.

The sponge function F takes:

input byte-string, the input bytes and

outputByteLen positive integer, the Length of the output in bytes

First the message is padded with zeroes to the closest multiple of 168 bytes. Then a byte ‘80‘ is XORed to the last byte of the padded message. and the resulting string is split into a sequence of 168-byte blocks.

As defined by the sponge construction, the process operates on a state and consists of two phases.

In the absorbing phase the state is initialized to all-zero. The message blocks are XORed into the first 168 bytes of the state. Each block absorbed is followed with an application of KP to the state.

In the squeezing phase output is formed by taking the first 168 bytes of the state, repeated as many times as necessary until outputByteLen bytes are obtained, interleaved with the application of KP to the state.

This definition of the sponge construction assumes a at least one-byte-long input where the last byte is in the ‘01‘-‘7F‘ range. This is the case in KangarooTwelve.

A pseudo-code version is available as follows:

```
F(input, outputByteLen):
  offset = 0
  state = ‘00‘^200

  # === Absorb complete blocks ===
  while offset < |input| - 168
      state ^= inputBytes[offset : offset + 168] || ‘00‘^32
      state = KP(state)
      offset += 168

  # === Absorb last block and treatment of padding ===
  LastBlockLength = |input| - offset
  state ^= inputBytes[offset:] || ‘00‘^(200-LastBlockLength)
  state ^= ‘00‘^167 || ‘80‘ || ‘00‘^32
  state = KP(state)

  # === Squeeze ===
  output = ‘00‘^0
  while outputByteLen > 168
      output = output || state[0:168]
      outputByteLen -= 168
      state = KP(state)

  output = output || state[0:outputByteLen]

  return output
  end
```

2.2. Tree hashing over F

On top of the sponge function F, KangarooTwelve uses a Sakura-compatible tree hash mode [SAKURA]. First, merge M and the OPTIONAL

C to a single input string S in a reversible way. length_encode(|C|) gives the length in bytes of C as a byte-string. See Section 2.3.

```
S = M || C || length_encode( |C| )
```

Then, split S into n chunks of 8192 bytes.

```
S = S_0 || .. || S_n-1
  |S_0| = .. = |S_n-2| = 8192 bytes
  |S_n-1| <= 8192 bytes
```

From S_1 .. S_n-1, compute the 32-bytes Chaining Values CV_1 .. CV_n-1. This computation SHOULD exploit the parallelism available on the platform in order to be optimally efficient.

```
CV_i    = F( S_i||‘0B‘, 32 )
```

Compute the final node: FinalNode.

- If |S| <= 8192 bytes, FinalNode = S
- Otherwise compute FinalNode as follow:

```
FinalNode = S_0 || ‘03 00 00 00 00 00 00 00‘
FinalNode = FinalNode || CV_1
      ..
FinalNode = FinalNode || CV_n-1
FinalNode = FinalNode || length_encode(n-1)
FinalNode = FinalNode || ‘FF FF‘
```

Finally, KangarooTwelve output is retrieved:

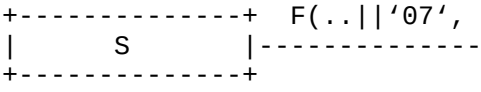
- If |S| <= 8192 bytes, from F(FinalNode||‘07‘, L)

```
KangarooTwelve( M, C, L ) = F( FinalNode||‘07‘, L )
```

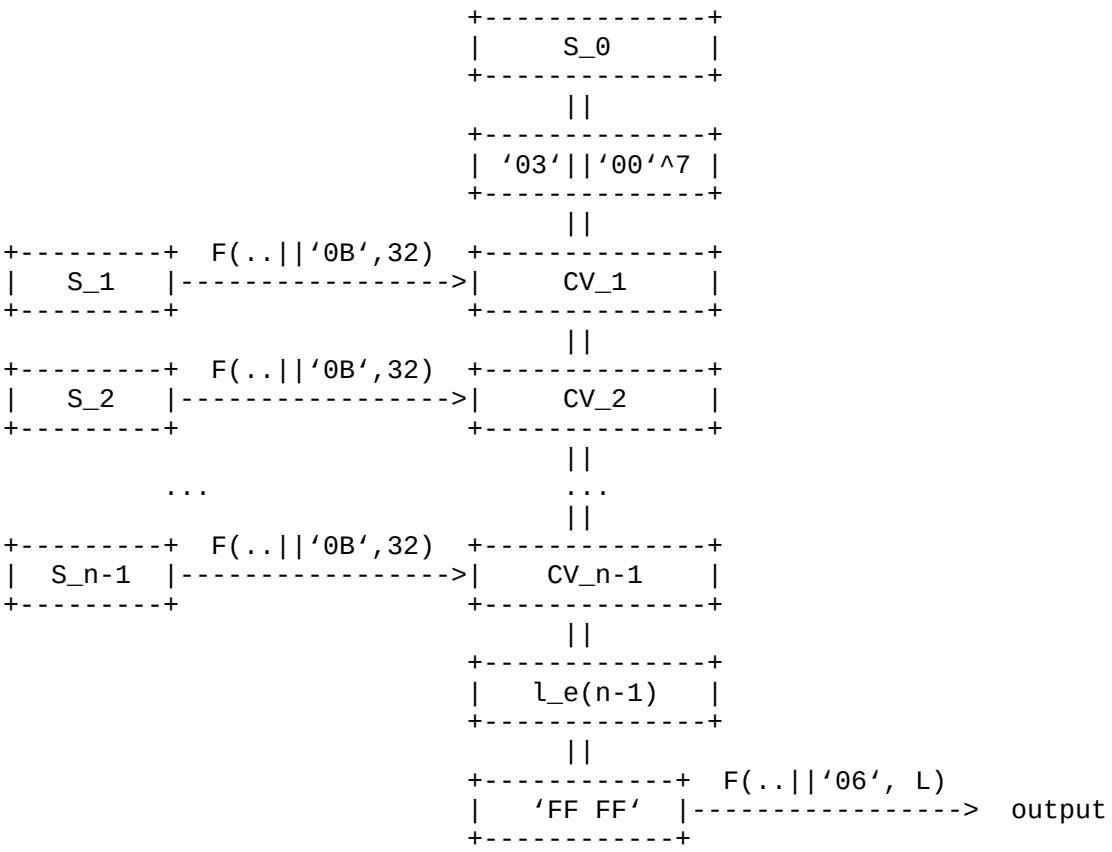
- Otherwise from F(FinalNode||‘06‘, L)

```
KangarooTwelve( M, C, L ) = F( FinalNode||‘06‘, L )
```

The following figure illustrates the computation flow of KangarooTwelve for |S| <= 8192 bytes:



The following figure illustrates the computation flow of KangarooTwelve for |S| > 8192 bytes:



We provide a pseudo code version in Appendix A.2.

In the table below are gathered the values of the domain separation bytes used by the tree hash mode:

Type	Byte
SingleNode	‘07‘
IntermediateNode	‘0B‘
FinalNode	‘06‘

2.3. length_encode(x)

The function length_encode takes as inputs a non negative integer x < 256**255 and outputs a string of bytes x_n-1 || .. || x_0 || n where

x = sum from i=0..n-1 of 256**i * x_i

and where n is the smallest non-negative integer such that x < 256**n. n is also the length of x_n-1 || .. || x_0.

As example, length_encode(0) = ‘00‘, length_encode(12) = ‘0C 01‘ and length_encode(65538) = ‘01 00 02 03‘

A pseudo code version is as follow.

```
length_encode(x):
  S = ‘00‘^0

  while x > 0
      S = x mod 256 || S
      x = x / 256

  S = S || length(S)

  return S
  end
```

3. Test vectors

Test vectors are based on the repetition of the pattern ‘00 01 .. FA‘ with a specific length. ptn(n) defines a string by repeating the pattern ‘00 01 .. FA‘ as many times as necessary and truncated to n bytes e.g.

```
Pattern for a length of 17 bytes:
ptn(17) =
  ‘00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10‘
```

```
  Pattern for a length of 17**2 bytes:
  ptn(17**2) =
    ‘00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
     10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
     20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
     30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
     40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
     50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
     60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
     70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
     80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
     90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
     A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
     B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
     C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
     D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
     E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
     F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA
     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
     10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
     20 21 22 23 24 25‘

KangarooTwelve(M=‘00‘^0, C=‘00‘^0, 32):
  ‘1A C2 D4 50 FC 3B 42 05 D1 9D A7 BF CA 1B 37 51
   3C 08 03 57 7A C7 16 7F 06 FE 2C E1 F0 EF 39 E5‘

KangarooTwelve(M=‘00‘^0, C=‘00‘^0, 64):
  ‘1A C2 D4 50 FC 3B 42 05 D1 9D A7 BF CA 1B 37 51
   3C 08 03 57 7A C7 16 7F 06 FE 2C E1 F0 EF 39 E5
   42 69 C0 56 B8 C8 2E 48 27 60 38 B6 D2 92 96 6C
   C0 7A 3D 46 45 27 2E 31 FF 38 50 81 39 EB 0A 71‘

KangarooTwelve(M=‘00‘^0, C=‘00‘^0, 10032), last 32 bytes:
  ‘E8 DC 56 36 42 F7 22 8C 84 68 4C 89 84 05 D3 A8
   34 79 91 58 C0 79 B1 28 80 27 7A 1D 28 E2 FF 6D‘

KangarooTwelve(M=ptn(1 bytes), C=‘00‘^0, 32):
  ‘2B DA 92 45 0E 8B 14 7F 8A 7C B6 29 E7 84 A0 58
   EF CA 7C F7 D8 21 8E 02 D3 45 DF AA 65 24 4A 1F‘

KangarooTwelve(M=ptn(17 bytes), C=‘00‘^0, 32):
  ‘6B F7 5F A2 23 91 98 DB 47 72 E3 64 78 F8 E1 9B
   0F 37 12 05 F6 A9 A9 3A 27 3F 51 DF 37 12 28 88‘

KangarooTwelve(M=ptn(17**2 bytes), C=‘00‘^0, 32):
  ‘0C 31 5E BC DE DB F6 14 26 DE 7D CF 8F B7 25 D1
   E7 46 75 D7 F5 32 7A 50 67 F3 67 B1 08 EC B6 7C‘
```

```
KangarooTwelve(M=ptn(17**3 bytes), C=‘00‘^0, 32):
  ‘CB 55 2E 2E C7 7D 99 10 70 1D 57 8B 45 7D DF 77
   2C 12 E3 22 E4 EE 7F E4 17 F9 2C 75 8F 0D 59 D0‘

KangarooTwelve(M=ptn(17**4 bytes), C=‘00‘^0, 32):
  ‘87 01 04 5E 22 20 53 45 FF 4D DA 05 55 5C BB 5C
   3A F1 A7 71 C2 B8 9B AE F3 7D B4 3D 99 98 B9 FE‘

KangarooTwelve(M=ptn(17**5 bytes), C=‘00‘^0, 32):
  ‘84 4D 61 09 33 B1 B9 96 3C BD EB 5A E3 B6 B0 5C
   C7 CB D6 7C EE DF 88 3E B6 78 A0 A8 E0 37 16 82‘

KangarooTwelve(M=ptn(17**6 bytes), C=‘00‘^0, 32):
  ‘3C 39 07 82 A8 A4 E8 9F A6 36 7F 72 FE AA F1 32
   55 C8 D9 58 78 48 1D 3C D8 CE 85 F5 8E 88 0A F8‘

KangarooTwelve(M=‘00‘^0, C=ptn(1 bytes), 32):
  ‘FA B6 58 DB 63 E9 4A 24 61 88 BF 7A F6 9A 13 30
   45 F4 6E E9 84 C5 6E 3C 33 28 CA AF 1A A1 A5 83‘

KangarooTwelve(M=‘FF‘, C=ptn(41 bytes), 32):
  ‘D8 48 C5 06 8C ED 73 6F 44 62 15 9B 98 67 FD 4C
   20 B8 08 AC C3 D5 BC 48 E0 B0 6B A0 A3 76 2E C4‘

KangarooTwelve(M=‘FF FF FF‘, C=ptn(41**2), 32):
  ‘C3 89 E5 00 9A E5 71 20 85 4C 2E 8C 64 67 0A C0
   13 58 CF 4C 1B AF 89 44 7A 72 42 34 DC 7C ED 74‘

KangarooTwelve(M=‘FF FF FF FF FF FF FF‘, C=ptn(41**3 bytes), 32):
  ‘75 D2 F8 6A 2E 64 45 66 72 6B 4F BC FC 56 57 B9
   DB CF 07 0C 7B 0D CA 06 45 0A B2 91 D7 44 3B CF‘
```

4. IANA Considerations

None.

5. Security Considerations

This document is meant to serve as a stable reference and an implementation guide for the KangarooTwelve eXtendable Output Function. It makes no assertion to its security and relies on the cryptanalysis of Keccak [KECCAK_CRYPTANALYSIS].

6. References

6.1. Normative References

[FIPS202] National Institute of Standards and Technology, "FIPS PUB 202 - SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", WWW http://dx.doi.org/10.6028/NIST.FIPS.202, August 2015.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

6.2. Informative References

[K12] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., and R. Van Keer, "KangarooTwelve: fast hashing based on Keccak-p", WWW http://eprint.iacr.org/2016/770.pdf, August 2016.

[KCP] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., and R. Van Keer, "Keccak Code Package", WWW https://github.com/KeccakTeam/KeccakCodePackage, December 2017.

[KECCAK_CRYPTANALYSIS] Keccak Team, "Summary of Third-party cryptanalysis of Keccak", WWW https://www.keccak.team/third_party.html, 2017.

[SAKURA] Bertoni, G., Daemen, J., Peeters, M., and G. Van Assche, "Sakura: a flexible coding for tree hashing", WWW http://eprint.iacr.org/2013/231.pdf, April 2013.

Appendix A. Pseudo code

The sub-sections of this appendix contain pseudo code definitions of KangarooTwelve. A standalone Python version is also available in the Keccak Code Package [KCP] and in [K12]

A.1. Keccak-p[1600,n_r=12]

```
KP(state):
  RC[0]  = ‘8B 80 00 80 00 00 00 00‘
  RC[1]  = ‘8B 00 00 00 00 00 00 80‘
  RC[2]  = ‘89 80 00 00 00 00 00 80‘
  RC[3]  = ‘03 80 00 00 00 00 00 80‘
  RC[4]  = ‘02 80 00 00 00 00 00 80‘
  RC[5]  = ‘80 00 00 00 00 00 00 80‘
  RC[6]  = ‘0A 80 00 00 00 00 00 00‘
  RC[7]  = ‘0A 00 00 80 00 00 00 80‘
  RC[8]  = ‘81 80 00 80 00 00 00 80‘
  RC[9]  = ‘80 80 00 00 00 00 00 80‘
  RC[10] = ‘01 00 00 80 00 00 00 00‘
  RC[11] = ‘08 80 00 80 00 00 00 80‘

  for x from 0 to 4
    for y from 0 to 4
      lanes[x][y] = state[8*(x+5*y):8*(x+5*y)+8]

  for round from 0 to 11
    # theta
    for x from 0 to 4
      C[x] = lanes[x][0]
      C[x] ^= lanes[x][1]
      C[x] ^= lanes[x][2]
      C[x] ^= lanes[x][3]
      C[x] ^= lanes[x][4]
    for x from 0 to 4
      D[x] = C[(x+4) mod 5] ^ ROL64(C[(x+1) mod 5], 1)
    for y from 0 to 4
      for x from 0 to 4
        lanes[x][y] = lanes[x][y]^D[x]

    # rho and pi
    (x, y) = (1, 0)
    current = lanes[x][y]
    for t from 0 to 23
      (x, y) = (y, (2*x+3*y) mod 5)
      (current, lanes[x][y]) =
          (lanes[x][y], ROL64(current, (t+1)*(t+2)/2))
```

```
      # chi
      for y from 0 to 4
        for x from 0 to 4
          T[x] = lanes[x][y]
        for x from 0 to 4
          lanes[x][y] = T[x] ^((not T[(x+1) mod 5]) & T[(x+2) mod 5])

      # iota
      lanes[0][0] ^= RC[round]

    state = '00'^0
    for x from 0 to 4
      for y from 0 to 4
        state = state || lanes[x][y]

    return state
    end
```

where ROL64(x, y) is a rotation of the 'x' 64-bit word toward the bits with higher indexes by 'y' positions. The 8-bytes byte-string x is interpreted as a 64-bit word in little-endian format.

A.2. KangarooTwelve

```
KangarooTwelve(inputMessage, customString, outputByteLen):
  S = inputMessage || customString
  S = S || length_encode( |customString| )

  if |S| <= 8192
      return F(S || '07', outputByteLen)
  else
      # === Kangaroo hopping ===
      FinalNode = S[0:8192] || '03' || '00'^7
      offset = 8192
      numBlock = 0
      while offset < |S|
          blockSize = min( |S| - offset, 8192)
          CV = F(S[offset : offset + blockSize] || '0B', 32)
          FinalNode = FinalNode || CV
          numBlock += 1
          offset   += blockSize

      FinalNode = FinalNode || length_encode( numBlock ) || 'FF FF'

      return F(FinalNode || '06', outputByteLen)
  end
```

Author’s Address

Benoit Viguier
Radboud University
Toernooiveld 212
Nijmegen
The Netherlands

EMail: b.viguier@cs.ru.nl