Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 4, 2015

B. Black
Microsoft
J. Bos
NXP Semiconductors
C. Costello
P. Longa
M. Naehrig
Microsoft Research

July 3, 2014

Elliptic Curve Cryptography (ECC) Nothing Up My Sleeve (NUMS) Curves and Curve Generation
draft-black-numscurves-01

Abstract

This memo describes a family of deterministically generated Nothing Up My Sleeve (NUMS) elliptic curves over prime fields offering high practical security in cryptographic applications, including Transport Layer Security (TLS) and X.509 certificates. The domain parameters are defined for both classical Weierstrass curves, for compatibility with existing applications, and modern twisted Edwards curves, allowing further efficiency improvements for a given security level.

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 http://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 January 4, 2015.

Copyright Notice

Table of Contents

1. Introduction 2
1.1. Requirements Language 3
2. Scope and Relation to Other Specifications 3
3. Requirements 4
3.1. Technical Requirements 4
3.2. Security Requirements 4
4. Notation 5
5. Curve Parameters 5
5.1. Parameters for 256-bit Curves 5
5.2. Parameters for 384-bit Curves 6
5.3. Parameters for 512-bit Curves 7
6. Object Identifiers and ASN.1 Syntax for X.509 Certificates . 8
6.1. Object Identifiers 8
6.2. ASN.1 Syntax for X.509 Certificates 8
7. Acknowledgements 9
8. Security Considerations 9
9. Intellectual Property Rights 9
10. IANA Considerations 10
11. References 10
11.1. Normative References 10
11.2. Informative References 10
Appendix A. Parameter Generation 12
A.1. Prime Generation 12
A.2. Deterministic Curve Parameter Generation 12
A.2.1. Weierstrass Curves 12
A.2.2. Twisted Edwards Curves 13
Appendix B. Generators 13
Authors' Addresses 14

1. Introduction

Since the initial standardization of elliptic curve cryptography (ECC) in [SEC1] there has been significant progress related to both efficiency and security of curves and implementations. Notable examples are algorithms protected against certain side-channel attacks, different 'special' prime shapes which allow faster modular

arithmetic, and a larger set of curve models from which to choose. There is also concern in the community regarding the generation and potential weaknesses of the curves defined in [NIST].

This memo describes a set of elliptic curves for cryptography, defined in [MSR] which have been specifically chosen to support constant-time, exception-free scalar multiplications that are resistant to a wide range of side-channel attacks including timing and cache attacks, thereby offering high practical security in cryptographic applications. These curves are deterministically generated based on algorithms defined in this document and without any hidden parameters or reliance on randomness, hence they are called Nothing Up My Sleeve (NUMS) curves. The domain parameters are defined for both classical Weierstrass curves, for compatibility with existing applications while delivering better performance and stronger security, and modern twisted Edwards curves, allowing even further efficiency improvements for a given security level.

1.1. Requirements Language

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].

2. Scope and Relation to Other Specifications

This RFC specifies elliptic curve domain parameters over prime fields GF(p) with p having a length of 256, 384, and 512 bits, in both Weierstrass and twisted Edwards form. These parameters were generated in a transparent and deterministic way and have been shown to resist current cryptanalytic approaches. Furthermore, this document identifies the security and implementation requirements for the parameters, and describes the methods used for the deterministic generation of the parameters.

This document also describes use of the specified parameters in X.509 certificates, in accordance with [RFC3279] and [RFC5480]. It does not address the cryptographic algorithms to be used with the specified parameters nor their application in other standards. However, it is consistent with the following RFCs that specify the usage of ECC in protocols and applications:

- [RFC4050] for XML signatures
- [RFC4492] for TLS
- [RFC4754] for IKE

- o [RFC5753] for cryptographic message syntax (CMS)

3. Requirements

3.1. Technical Requirements

1. Applicability to multiple cryptographic algorithms without transformation, in particular key exchange, e.g. Elliptic Curve Diffie-Hellman (ECDH), and digital signature algorithms, e.g., (ECDSA), Schnorr.

2. Multiple security levels using the same curve generation algorithm with only a security parameter change. The curve generation algorithm must be extensible to any security level.

3. Ability to use pre-computation for increased performance. In particular, speed-up in key generation is important when a curve is used with ephemeral key exchange algorithm, such as ECDHE.

4. The bit length of prime and order of curves for a given security level MUST be divisible by 8. Specifically, constructions such as NIST P-521 are to be avoided as they introduce interoperability and implementation problems.

3.2. Security Requirements

For each curve type (twisted Edwards or Weierstrass) at a specific specific security level:

1. The domain parameters SHALL be generated in a simple, deterministic manner, without any secret or random inputs. The derivation of the curve parameters is defined in Appendix A.

2. The curve SHALL NOT restrict the scalars to a small subset. Using full-set scalars prevents implementation pitfalls that might otherwise go unnoticed.

3. The curve selection SHALL include prime order curves with cofactor 1 only. Composite order curves require changes in protocols and in implementations. Additionally, implementations for composite order curves must thwart subgroup attacks.

4. The trace of Frobenius MUST NOT be in {0, 1} in order to rule out the attacks described in [Smart], [AS], and [S], as in [EBP].

5. MOV Degree: the embedding degree k MUST be greater than (r - 1) / 100, as in [EBP].

6. CM Discriminant: discriminant D MUST be greater than 2^100, as in [SC].

4. Notation

Throughout this document, the following notation is used:

- s: Denotes the bit length, here s in {256,384,512}.
- p: Denotes the prime number defining the base field.
- c: A positive integer used in the representation of the prime p = 2^s - c.
- GF(p): The finite field with p elements.
- b: An element in the finite field GF(p), different from -2,2.
- Eb: The elliptic curve Eb/GF(p): y^2 = x^3 - 3x + b in short Weierstrass form, defined over GF(p) by the parameter b.
- rb: The order rb = #Eb(GF(p)) of the group of GF(p)-rational points on Eb.
- tb: The trace of Frobenius tb = p + 1 - rb of Eb.
- rb': The order rb' = #E'b(GF(p)) = p + 1 + tb of the group of GF(p)-rational points on the quadratic twist Eb': y^2 = x^3 - 3x - b.
- d: An element in the finite field GF(p), different from -1,0.
- Ed: The elliptic curve Ed/GF(p): -x^2 + y^2 = 1 + dx^2y^2 in twisted Edwards form, defined over GF(p) by the parameter d.
- rd: The subgroup order such that 4 * rd = #Ed(GF(p)) is the order of the group of GF(p)-rational points on Ed.
- td: The trace of Frobenius td = p + 1 - 4 * rd of Ed.
- rd': The subgroup order such that 4 * rd' = #Ed'(GF(p)) = p + 1 + tb is the order of the group of GF(p)-rational points on the quadratic twist Ed': -x^2 = y^2 = 1 + (1 / d) * x^2 * y^2.
- P: A generator point defined over GF(p) either of prime order rb in the Weierstrass curve Eb, or of prime order rd on the twisted Edwards curve Ed.
- X(P): The x-coordinate of the elliptic curve point P.
- Y(P): The y-coordinate of the elliptic curve point P.

5. Curve Parameters

5.1. Parameters for 256-bit Curves

```
   p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
       FFFFF43
   a = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
       FFFFF40
   b = 0x25581
   r = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE43C8275EA265C60E43C8275E
       A265C60
X(P) = 0x01
Y(P) = 0x696F1853C1E466D7FC82C96CCEEEDD6BD02C2F9375894EC10BF46306C
       2B56C77
   h = 0x01
```

Curve-Id: numsp256d1

```
   p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
       FFFFF43
   a = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
       FFFFF42
   d = 0x3BEE
   r = 0x3FFFFFFFFFFFFFFFFFFFFFFFFFFFFFBE6AA55AD0A6BC64E5B84E6F1
       122B4AD
X(P) = 0x0D
Y(P) = 0x7D0AB41E2A1276DBA3D330B39FA046BFBE2A6D63824D303F707F6FB53
       31CADBA
   h = 0x04
```

Curve-Id: numsp256t1

5.2. Parameters for 384-bit Curves

```
   p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
       FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEC3
   a = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
       FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEC0
   b = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
       FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF77BB
   r = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD61EAF1EE
       B5D6881BEDA9D3D4C37E27A604D81F67B0E61B9
X(P) = 0x02
Y(P) = 0x3C9F82CB4B87B4DC71E763E0663E5DBD8034ED422F04F82673330DC58
       D15FFA2B4A3D0BAD5D30F865BCBBF503EA66F43
   h = 0x01
```

Curve-Id: numsp384d1

```
   p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
       FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEC3
   a = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC
       FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEC2
   d = 0x5158A
   r = 0x3FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFECD7D11ED
       5A259A25A13A0458E39F4E451D6D71F70426E25
X(P) = 0x08
Y(P) = 0x749CDABA136CE9B65BD4471794AA619DAA5C7B4C930BFF8EBD798A8AE
       753C6D72F003860FEBABAD534A4ACF5FA7F5BEE
   h = 0x04
```

Curve-Id: numsp384t1

5.3. Parameters for 512-bit Curves

```
   p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
       FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
       FFFFFFFFFFDC7
   a = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
       FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
       FFFFFFFFFFDC4
   b = 0x1D99B
   r = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
       FFFFFFF5B3CA4FB94E7831B4FC258ED97D0BDC63B568B36607CD243CE
       153F390433555D
X(P) = 0x02
Y(P) = 0x1C282EB23327F9711952C250EA61AD53FCC13031CF6DD336E0B932843
       3AFBDD8CC5A1C1F0C716FDC724DDE537C2B0ADB00BB3D08DC83755B20
       5CC30D7F83CF28
   h = 0x01
```

Curve-Id: numsp512d1

```
      p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
            FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
            FFFFFFFFFFDC7
      a = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
            FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
            FFFFFFFFFFDC6
      d = 0x9BAA8
      r = 0x3FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
            FFFFFFFA7E50809EFDABBB9A624784F449545F0DCEA5FF0CB800F894E
            78D1CB0B5F0189
   X(P) = 0x20
   Y(P) = 0x7D67E841DC4C467B605091D80869212F9CEB124BF726973F9FF048779
            E1D614E62AE2ECE5057B5DAD96B7A897C1D72799261134638750F4F0C
            B91027543B1C5E
      h = 0x04
```

Curve-Id: numsp512t1

6. Object Identifiers and ASN.1 Syntax for X.509 Certificates

6.1. Object Identifiers

The root of the tree for the object identifiers defined in this specification is given by:

```
[TBDOID]
```

The following object identifiers represent the domain parameters for the curves defined in this draft:

```
numsp256d1 OBJECT IDENTIFIER ::= {versionOne 1}

numsp256t1 OBJECT IDENTIFIER ::= {versionOne 2}

numsp384d1 OBJECT IDENTIFIER ::= {versionOne 3}

numsp384t1 OBJECT IDENTIFIER ::= {versionOne 4}

numsp512d1 OBJECT IDENTIFIER ::= {versionOne 5}

numsp512t1 OBJECT IDENTIFIER ::= {versionOne 6}
```

6.2. ASN.1 Syntax for X.509 Certificates

The domain parameters for the curves specified in this RFC SHALL be used with X.509 certificates according to [RFC5480]. Specifically, the algorithm field of subjectPublicKeyInfo MUST be one of:

- o id-ecPublicKey to indicate that the algorithms that can be used with the subject public key are unrestricted, as required for ECDSA, or

- o id-ecDH to indicate that the algorithm that can be used with the subject public key is restricted to the ECDH key agreement algorithm, or

- o id-ecMQV indicates that the algorithm that can be used with the subject public key is restricted to the Elliptic Curve Menezes-Qu-Vanstone (ECMQV) key agreement algorithm, and

The field algorithm.parameter of subjectPublicKeyInfo MUST be of type namedCurve. No other values for this field are acceptable.

7. Acknowledgements

The authors would like to thank Brian Lamacchia and Tolga Acar for their help in the development of this draft.

8. Security Considerations

In addition to the discussion in the requirements, [MSR], [SC], and the other reference documents on EC security, users SHOULD match curves with cryptographic functions of similar strength. Specific recommendations for algorithms, per [RFC5480] are as follows:

Minimum Bits of Security	EC Key Size	Message Digest Algorithm	Curves
128	256	SHA-256	numsp256d1/t1
192	384	SHA-384	numsp384d1/t1
256	512	SHA-512	numsp512d1/t1

Table 1

9. Intellectual Property Rights

The authors have no knowledge about any intellectual property rights that cover the usage of the domain parameters defined herein. However, readers should be aware that implementations based on these domain parameters may require use of inventions covered by patent rights.

10. IANA Considerations

IANA is requested to allocate an object identifier for elliptic curves under the PKIX root declared in [RFC5480]:

```
PKIX1Algorithms2008 { iso(1) identified-organization(3) dod(6)
  internet(1) security(5) mechanisms(5) pkix(7) id-mod(0) 45 }
```

IANA is further requested to allocate object identifiers under this new elliptic curve root for the named curves in Section 6.1.

11. References

11.1. Normative References

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

11.2. Informative References

[AS] Satoh, T. and K. Araki, "Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves", 1998.

[EBP] ECC Brainpool, "ECC Brainpool Standard Curves and Curve Generation", October 2005, <http://www.ecc-brainpool.org/download/Domain-parameters.pdf>.

[ECCP] Bos, J., Halderman, J., Heninger, N., Moore, J., Naehrig, M., and E. Wustrow, "Elliptic Curve Cryptography in Practice", December 2013, <https://eprint.iacr.org/2013/734>.

[FPPR] Faugere, J., Perret, L., Petit, C., and G. Renault, 2012, <http://dx.doi.org/10.1007/978-3-642-29011-4_4>.

[MSR] Bos, J., Costello, C., Longa, P., and M. Naehrig, "Selecting Elliptic Curves for Cryptography: An Efficiency and Security Analysis", February 2014, <http://eprint.iacr.org/2014/130.pdf>.

[NIST] National Institute of Standards, "Recommended Elliptic Curves for Federal Government Use", July 1999, <http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf>.

[RFC3279] Bassham, L., Polk, W., and R. Housley, "Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3279, April 2002.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, July 2003.

[RFC4050] Blake-Wilson, S., Karlinger, G., Kobayashi, T., and Y. Wang, "Using the Elliptic Curve Signature Algorithm (ECDSA) for XML Digital Signatures", RFC 4050, April 2005.

[RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, May 2006.

[RFC4754] Fu, D. and J. Solinas, "IKE and IKEv2 Authentication Using the Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 4754, January 2007.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, March 2009.

[RFC5753] Turner, S. and D. Brown, "Use of Elliptic Curve Cryptography (ECC) Algorithms in Cryptographic Message Syntax (CMS)", RFC 5753, January 2010.

[S] Semaev, I., "Evaluation of discrete logarithms on some elliptic curves", 1998.

[SC] Bernstein, D. and T. Lange, "SafeCurves: choosing safe curves for elliptic-curve cryptography", June 2014, <http://safecurves.cr.yp.to/>.

[SEC1] Certicom Research, "SEC 1: Elliptic Curve Cryptography", September 2000, <http://www.secg.org/collateral/sec1_final.pdf>.

[Smart] Smart, N., "The discrete logarithm problem on elliptic curves of trace one", 1999.

Appendix A. Parameter Generation

This section describes the generation of the curve parameters, namely the base field prime p, the curve parameters b and d for the Weierstrass and twisted Edwards curves, respectively, and a generator point P of the prime order subgroup of the elliptic curve.

A.1. Prime Generation

For a given bitlength s in {256, 384, 512}, a prime p is selected as a pseudo-Mersenne prime of the form p = 2^s - c for a positive integer c. Each prime is determined by the smallest positive integer c such that p = 2^s - c is prime and p = 3 mod 4.

```
Input: a bit length s in {256, 384, 512}
Output: a prime p = 2^s - c with p = 3 mod 4
  1. Set c = 1
  2. while (p = 2^s - c is not prime) do
       c = c + 4
     end while
  3. Output p
```

GenerateP

A.2. Deterministic Curve Parameter Generation

A.2.1. Weierstrass Curves

For a given bitlength s in {256, 384, 512} and a corresponding prime p = 2^s - c selected according to Section A.1, the elliptic curve Eb in short Weierstrass form is determined by the element b from GF(p), different from -2,2 with smallest absolute value (when represented as an integer in the interval [-(p - 1) / 2, (p - 1) / 2]) such that both group orders rb and rb’ are prime, and the group order rb < p, i.e. tb > 1. In addition, care must be taken to ensure the MOV degree and CM discriminant requirements from Section 3.2 are met.

```
Input: a prime p = 2^s - c with p = 3 mod 4
Output: the parameter b defining the curve Eb
  1. Set b = 1
  2. while (rb is not prime or rb' is not prime) do
       b = b + 1
     end while
  3. if p + 1 < rb then
       b = -b
     end if
  4. Output b
```

GenerateCurveWeierstrass

A.2.2. Twisted Edwards Curves

For a given bitlength s in {256, 384, 512} and a corresponding prime p = 2^s - c selected according to Section A.1, the elliptic curve Ed in twisted Edwards form is determined by the element d from GF(p), different from -1,0 with smallest value (when represented as a positive integer) such that both subgroup orders rd and rd' are prime, and the group order 4 * rd < p, i.e. td > 1. In addition, care must be taken to ensure the MOV degree and CM discriminant requirements from Section 3.2 are met.

```
Input: a prime p = 2^s - c with p = 3 mod 4
Output: the parameter d defining the curve Ed
  1. Set d = 1
  2. while (rd is not prime or rd' is not prime or 4*rd > p) do
       d = d + 1;
     end while
  3. Output d
```

GenerateCurveTEdwards

Appendix B. Generators

The generator points on all six curves are selected as the points of order rb and rd, respectively, with the smallest value for x(P) when represented as a positive integer.

```
   Input: a prime p, and a Weierstrass curve parameter b
   Output: a generator point P = (x(P), y(P)) of order rb
   1. Set x = 1
   2. while ((x^3 - 3 * x + b) is not a quadratic residue modulo p) do
        x = x + 1
      end while
   3. Compute an integer s, 0 < s < p, such that
           s^2 = x^3 - 3 * x + b mod p
   4. Set y = min(s, p - s)
   5. Output P = (x, y)
```

GenerateGenWeierstrass

```
   Input: a prime p and a twisted Edwards curve parameter d
   Output: a generator point P = (x(P), y(P)) of order rd
   1. Set x = 1
   2. while ((d * x^2 = 1 mod p)
             or ((1 + x^2) * (1 - d * x^2) is not a quadratic residue
             modulo p)) do x = x + 1
      end while
   3. Compute an integer s, 0 < s < p, such that
           s^2 * (1 - d * x^2) = 1 + x^2 mod p
   4. Set y = min(s, p - s)
   5. Output P = (x, y)
```

GenerateGenTEdwards

Authors' Addresses

Benjamin Black
Microsoft
One Microsoft Way
Redmond, WA 98115
US

Email: benblack@microsoft.com

Joppe W. Bos
NXP Semiconductors
Interleuvenlaan 80
3001 Leuven
Belgium

Email: joppe.bos@nxp.com

Craig Costello
Microsoft Research
One Microsoft Way
Redmond, WA 98115
US

Email: craigco@microsoft.com

Patrick Longa
Microsoft Research
One Microsoft Way
Redmond, WA 98115
US

Email: plonga@microsoft.com

Michael Naehrig
Microsoft Research
One Microsoft Way
Redmond, WA 98115
US

Email: mnaehrig@microsoft.com

Network Working Group
Internet-Draft
Obsoletes: 6090 (if approved)
Intended status: Informational
Expires: December 31, 2015

D. McGrew
Cisco Systems
K. Igoe
M. Salter
National Security Agency
P. Hoffman
VPN Consortium

June 29, 2015

Fundamental Elliptic Curve Cryptography Algorithms
draft-hoffman-rfc6090bis-02

Abstract

This note describes the fundamental algorithms of Elliptic Curve Cryptography (ECC) as they were defined in some seminal references from 1994 and earlier. These descriptions may be useful for implementing the fundamental algorithms without using any of the specialized methods that were developed in following years. Only elliptic curves defined over fields of characteristic greater than three are in scope; these curves are those used in Suite B.

This version of the note incorporates errata that were reported on RFC 6090 [RFC6090].

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 http://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 December 31, 2015.

Copyright Notice

Table of Contents

1. Introduction 3
1.1. Conventions Used in This Document 4
2. Mathematical Background 4
2.1. Modular Arithmetic 4
2.2. Group Operations 5
2.3. The Finite Field Fp 6
3. Elliptic Curve Groups 7
3.1. Homogeneous Coordinates 8
3.2. Other Coordinates 9
3.3. ECC Parameters 9
3.3.1. Discriminant 10
3.3.2. Security 10
4. Elliptic Curve Diffie-Hellman (ECDH) 10
4.1. Data Types 11
4.2. Compact Representation 11
5. Elliptic Curve ElGamal Signatures 11
5.1. Background 11
5.2. Hash Functions 12
5.3. KT-IV Signatures 12
5.3.1. Keypair Generation 13
5.3.2. Signature Creation 13
5.3.3. Signature Verification 13
5.4. KT-I Signatures 14
5.4.1. Keypair Generation 14
5.4.2. Signature Creation 14
5.4.3. Signature Verification 14
5.5. Converting KT-IV Signatures to KT-I Signatures 15
5.6. Rationale 15
6. Converting between Integers and Octet Strings 16
6.1. Octet-String-to-Integer Conversion 17
6.2. Integer-to-Octet-String Conversion 17
7. Interoperability 17
7.1. ECDH 17
7.2. KT-I and ECDSA 18
8. Validating an Implementation 18
8.1. ECDH 19

8.2. KT-I 20
9. Intellectual Property 20
9.1. Disclaimer 21
10. Security Considerations 21
10.1. Subgroups 22
10.2. Diffie-Hellman 22
10.3. Group Representation and Security 22
10.4. Signatures 23
11. Acknowledgements 24
12. References 24
12.1. Normative References 24
12.2. Informative References 25
Appendix A. Key Words 29
Appendix B. Random Integer Generation 29
Appendix C. Why Compact Representation Works 30
Appendix D. Example ECC Parameter Set 31
Appendix E. Additive and Multiplicative Notation 31
Appendix F. Algorithms 32
F.1. Affine Coordinates 32
F.2. Homogeneous Coordinates 33
Authors' Addresses 34

1. Introduction

ECC is a public-key technology that offers performance advantages at higher security levels. It includes an elliptic curve version of the Diffie-Hellman key exchange protocol [DH1976] and elliptic curve versions of the ElGamal Signature Algorithm [E1985]. The adoption of ECC has been slower than had been anticipated, perhaps due to the lack of freely available normative documents and uncertainty over intellectual property rights.

This note contains a description of the fundamental algorithms of ECC over finite fields with characteristic greater than three, based directly on original references. Its intent is to provide the Internet community with a summary of the basic algorithms that predate any specialized or optimized algorithms. The summary is detailed enough for use as a normative reference. The original descriptions and notations were followed as closely as possible.

This version of the note incorporates verified errata that were reported against RFC 6090. Paragraphs or artwork that has errata applied are marked with "%%%". Thise markings will be removed when this document is published as an RFC.

There are several standards that specify or incorporate ECC algorithms, including the Internet Key Exchange (IKE), ANSI X9.62, and IEEE P1363. The algorithms in this note can interoperate with

some of the algorithms in these standards, with a suitable choice of parameters and options. The specifics are itemized in Section 7.

The rest of the note is organized as follows. Sections 2.1, 2.2, and 2.3 furnish the necessary terminology and notation from modular arithmetic, group theory, and the theory of finite fields, respectively. Section 3 defines the groups based on elliptic curves over finite fields of characteristic greater than three. Section 4 presents the fundamental Elliptic Curve Diffie-Hellman (ECDH) algorithm. Section 5 presents elliptic curve versions of the ElGamal signature method. The representation of integers as octet strings is specified in Section 6. Sections 2 through 6, inclusive, contain all of the normative text (the text that defines the norm for implementations conforming to this specification), and all of the following sections are purely informative. Interoperability is discussed in Section 7. Validation testing is described in Section 8. Section 9 reviews intellectual property issues. Section 10 summarizes security considerations. Appendix B describes random number generation, and other appendices provide clarifying details.

1.1. Conventions Used in This Document

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 Appendix A.

2. Mathematical Background

This section reviews mathematical preliminaries and establishes terminology and notation that are used below.

2.1. Modular Arithmetic

This section reviews modular arithmetic. Two integers x and y are said to be congruent modulo n if x - y is an integer multiple of n.

Two integers x and y are coprime when their greatest common divisor is 1; in this case, there is no third number z > 1 such that z divides x and z divides y.

The set Zq = { 0, 1, 2, ..., q-1 } is closed under the operations of modular addition, modular subtraction, modular multiplication, and modular inverse. These operations are as follows.

For each pair of integers a and b in Zq, a + b mod q is equal to a + b if a + b < q, and is equal to a + b - q otherwise.

For each pair of integers a and b in Zq, a - b mod q is equal to a - b if a - b >= 0, and is equal to a - b + q otherwise.

For each pair of integers a and b in Zq, a * b mod q is equal to the remainder of a * b divided by q.

For each integer x in Zq that is coprime with q, the inverse of x modulo q is denoted as 1/x mod q, and can be computed using the extended Euclidean algorithm (see Section 4.5.2 of [K1981v2], for example).

Algorithms for these operations are well known; for instance, see Chapter 4 of [K1981v2].

2.2. Group Operations

This section establishes some terminology and notation for mathematical groups, which are needed later on. Background references abound; see [D1966], for example.

A group is a set of elements G together with an operation that combines any two elements in G and returns a third element in G. The operation is denoted as * and its application is denoted as a * b, for any two elements a and b in G. The operation is associative, that is, for all a, b, and c in G, a * (b * c) is identical to (a * b) * c. Repeated application of the group operation N-1 times to the element a is denoted as a^N, for any element a in G and any positive integer N. That is, a^2 = a * a, a^3 = a * a * a, and so on. The associativity of the group operation ensures that the computation of a^n is unambiguous; any grouping of the terms gives the same result.

%%% The above definition of a group operation uses multiplicative notation. Sometimes an alternative called additive notation is used, in which a * b is denoted as a + b, and a^N is denoted as Na. In multiplicative notation, a^N is called exponentiation, while the equivalent operation in additive notation is called scalar multiplication. In this document, multiplicative notation is used throughout for consistency. Appendix E elucidates the correspondence between the two notations.

%%% Every group has a special element called the identity element, which we denote as e. For each element a in G, e * a = a * e = a. By convention, a^0 is equal to the identity element and a^1 is equal to a itself for any a in G.

Every group element a has a unique inverse element b such that

a * b = b * a = e. The inverse of a is denoted as a^-1 in multiplicative notation. (In additive notation, the inverse of a is denoted as -a.)

For any positive integer X, a^(-X) is defined to be (a^-1)^(X). Using this convention, exponentiation behaves as one would expect, namely for any integers X and Y:

a^(X+Y) = (a^X)*(a^Y)

(a^X)^Y = a^(XY) = (a^Y)^X.

In cryptographic applications, one typically deals with finite groups (groups with a finite number of elements), and for such groups, the number of elements of the group is also called the order of the group. A group element a is said to have finite order if a^X = e for some positive integer X, and the order of a is the smallest such X. If no such X exists, a is said to have infinite order. All elements of a finite group have a finite order, and the order of an element is always a divisor of the group order.

If a group element a has order R, then for any integers X and Y,

a^X = a^(X mod R),

a^X = a^Y if and only if X is congruent to Y mod R,

the set H = { a, a^2, a^3, ... , a^R=e } forms a subgroup of G, called the cyclic subgroup generated by a, and a is said to be a generator of H.

%%% Typically, there are several group elements that generate H. Any group element of the form a^M, with M relatively prime to R, also generates H. Note that a^M is equal to a^(M mod R) for any non-negative integer M.

Given the element a of order R, and an integer i between 1 and R-1, inclusive, the element a^i can be computed by the "square and multiply" method outlined in Section 2.1 of [M1983] (see also Knuth, Vol. 2, Section 4.6.3), or other methods.

2.3. The Finite Field Fp

This section establishes terminology and notation for finite fields with prime characteristic.

When p is a prime number, then the set Zp, with the addition, subtraction, multiplication, and division operations, is a finite

field with characteristic p. Each nonzero element x in Zp has an inverse 1/x. There is a one-to-one correspondence between the integers between zero and p-1, inclusive, and the elements of the field. The field Zp is sometimes denoted as Fp or GF(p).

Equations involving field elements do not explicitly denote the "mod p" operation, but it is understood to be implicit. For example, the statement that x, y, and z are in Fp and

z = x + y

is equivalent to the statement that x, y, and z are in the set { 0, 1, ..., p-1 } and

z = x + y mod p.

3. Elliptic Curve Groups

This note only covers elliptic curves over fields with characteristic greater than three; these are the curves used in Suite B [SuiteB]. For other fields, the definition of the elliptic curve group would be different.

An elliptic curve over a field Fp is defined by the curve equation

y^2 = x^3 + a*x + b,

where x, y, a, and b are elements of the field Fp [M1985], and the discriminant is nonzero (as described in Section 3.3.1). A point on an elliptic curve is a pair (x,y) of values in Fp that satisfies the curve equation, or it is a special point (@,@) that represents the identity element (which is called the "point at infinity"). The order of an elliptic curve group is the number of distinct points.

Two elliptic curve points (x1,y1) and (x2,y2) are equal whenever x1=x2 and y1=y2, or when both points are the point at infinity. The inverse of the point (x1,y1) is the point (x1,-y1). The point at infinity is its own inverse.

The group operation associated with the elliptic curve group is as follows [BC1989]. To an arbitrary pair of points P and Q specified by their coordinates (x1,y1) and (x2,y2), respectively, the group operation assigns a third point P*Q with the coordinates (x3,y3). These coordinates are computed as follows:

(x3,y3) = (@,@) when P is not equal to Q and x1 is equal to x2.

x3 = ((y2-y1)/(x2-x1))^2 - x1 - x2 and

y3 = (x1-x3)*(y2-y1)/(x2-x1) - y1 when P is not equal to Q and x1 is not equal to x2.

(x3,y3) = (@,@) when P is equal to Q and y1 is equal to 0.

x3 = ((3*x1^2 + a)/(2*y1))^2 - 2*x1 and
y3 = (x1-x3)*(3*x1^2 + a)/(2*y1) - y1 if P is equal to Q and y1 is not equal to 0.

In the above equations, a, x1, x2, x3, y1, y2, and y3 are elements of the field Fp; thus, computation of x3 and y3 in practice must reduce the right-hand-side modulo p. Pseudocode for the group operation is provided in Appendix F.1.

The representation of elliptic curve points as a pair of integers in Zp is known as the affine coordinate representation. This representation is suitable as an external data representation for communicating or storing group elements, though the point at infinity must be treated as a special case.

Some pairs of integers are not valid elliptic curve points. A valid pair will satisfy the curve equation, while an invalid pair will not.

3.1. Homogeneous Coordinates

An alternative way to implement the group operation is to use homogeneous coordinates [K1987] (see also [KMOV1991]). This method is typically more efficient because it does not require a modular inversion operation.

An elliptic curve point (x,y) (other than the point at infinity (@,@)) is equivalent to a point (X,Y,Z) in homogeneous coordinates whenever x=X/Z mod p and y=Y/Z mod p.

Let P1=(X1,Y1,Z1) and P2=(X2,Y2,Z2) be points on an elliptic curve, and suppose that the points P1 and P2 are not equal to (@,@), P1 is not equal to P2, and P1 is not equal to P2^-1. Then the product P3=(X3,Y3,Z3) = P1 * P2 is given by

X3 = v * (Z2 * (Z1 * u^2 - 2 * X1 * v^2) - v^3) mod p

Y3 = Z2 * (3 * X1 * u * v^2 - Y1 * v^3 - Z1 * u^3) + u * v^3 mod p

Z3 = v^3 * Z1 * Z2 mod p

where u = Y2 * Z1 - Y1 * Z2 mod p and v = X2 * Z1 - X1 * Z2 mod p.

When the points P1 and P2 are equal, then (X1/Z1, Y1/Z1) is equal to (X2/Z2, Y2/Z2), which is true if and only if u and v are both equal to zero.

The product P3=(X3,Y3,Z3) = P1 * P1 is given by

$$X3 = 2 * Y1 * Z1 * (w^2 - 8 * X1 * Y1^2 * Z1) \bmod p$$

$$Y3 = 4 * Y1^2 * Z1 * (3 * w * X1 - 2 * Y1^2 * Z1) - w^3 \bmod p$$

$$Z3 = 8 * (Y1 * Z1)^3 \bmod p$$

where $w = 3 * X1^2 + a * Z1^2 \bmod p$. In the above equations, a, u, v, w, X1, X2, X3, Y1, Y2, Y3, Z1, Z2, and Z3 are integers in the set Fp. Pseudocode for the group operation in homogeneous coordinates is provided in Appendix F.2.

When converting from affine coordinates to homogeneous coordinates, it is convenient to set Z to 1. When converting from homogeneous coordinates to affine coordinates, it is necessary to perform a modular inverse to find 1/Z mod p.

3.2. Other Coordinates

Some other coordinate systems have been described; several are documented in [CC1986], including Jacobi coordinates.

3.3. ECC Parameters

In cryptographic contexts, an elliptic curve parameter set consists of a cyclic subgroup of an elliptic curve together with a preferred generator of that subgroup. When working over a prime order finite field with characteristic greater than three, an elliptic curve group is completely specified by the following parameters:

The prime number p that indicates the order of the field Fp.

The value a used in the curve equation.

The value b used in the curve equation.

The generator g of the subgroup.

The order n of the subgroup generated by g.

An example of an ECC parameter set is provided in Appendix D.

Parameter generation is out of scope for this note.

Each elliptic curve point is associated with a particular parameter set. The elliptic curve group operation is only defined between two points in the same group. It is an error to apply the group operation to two elements that are from different groups, or to apply the group operation to a pair of coordinates that is not a valid point. (A pair (x,y) of coordinates in Fp is a valid point only when it satisfies the curve equation.) See Section 10.3 for further information.

3.3.1. Discriminant

For each elliptic curve group, the discriminant -16*(4*a^3 + 27*b^2) must be nonzero modulo p [S1986]; this requires that

$$4*a^3 + 27*b^2 \neq 0 \bmod p.$$

3.3.2. Security

Security is highly dependent on the choice of these parameters. This section gives normative guidance on acceptable choices. See also Section 10 for informative guidance.

The order of the group generated by g MUST be divisible by a large prime, in order to preclude easy solutions of the discrete logarithm problem [K1987].

With some parameter choices, the discrete log problem is significantly easier to solve. This includes parameter sets in which b = 0 and p = 3 (mod 4), and parameter sets in which a = 0 and p = 2 (mod 3) [MOV1993]. These parameter choices are inferior for cryptographic purposes and SHOULD NOT be used.

4. Elliptic Curve Diffie-Hellman (ECDH)

The Diffie-Hellman (DH) key exchange protocol [DH1976] allows two parties communicating over an insecure channel to agree on a secret key. It was originally defined in terms of operations in the multiplicative group of a field with a large prime characteristic. Massey [M1983] observed that it can be easily generalized so that it is defined in terms of an arbitrary cyclic group. Miller [M1985] and Koblitz [K1987] analyzed the DH protocol over an elliptic curve group. We describe DH following the former reference.

Let G be a group, and g be a generator for that group, and let t denote the order of G. The DH protocol runs as follows. Party A chooses an exponent j between 1 and t-1, inclusive, uniformly at random, computes g^j, and sends that element to B. Party B chooses an exponent k between 1 and t-1, inclusive, uniformly at random,

computes g^k, and sends that element to A. Each party can compute g^(j*k); party A computes (g^k)^j, and party B computes (g^j)^k.

See Appendix B regarding generation of random integers.

4.1. Data Types

Each run of the ECDH protocol is associated with a particular parameter set (as defined in Section 3.3), and the public keys g^j and g^k and the shared secret g^(j*k) are elements of the cyclic subgroup associated with the parameter set.

An ECDH private key z is an integer in Zt, where t is the order of the subgroup.

4.2. Compact Representation

As described in the final paragraph of [M1985], the x-coordinate of the shared secret value g^(j*k) is a suitable representative for the entire point whenever exponentiation is used as a one-way function. In the ECDH key exchange protocol, after the element g^(j*k) has been computed, the x-coordinate of that value can be used as the shared secret. We call this compact output.

Following [M1985] again, when compact output is used in ECDH, only the x-coordinate of an elliptic curve point needs to be transmitted, instead of both coordinates as in the typical affine coordinate representation. We call this the compact representation. Its mathematical background is explained in Appendix C.

ECDH can be used with or without compact output. Both parties in a particular run of the ECDH protocol MUST use the same method. ECDH can be used with or without compact representation. If compact representation is used in a particular run of the ECDH protocol, then compact output MUST be used as well.

5. Elliptic Curve ElGamal Signatures

5.1. Background

The ElGamal signature algorithm was introduced in 1984 [E1984a] [E1984b] [E1985]. It is based on the discrete logarithm problem, and was originally defined for the multiplicative group of the integers modulo a large prime number. It is straightforward to extend it to use other finite groups, such as the multiplicative group of the finite field GF(2^w) [AMV1990] or an elliptic curve group [A1992].

An ElGamal signature consists of a pair of components. There are many possible generalizations of ElGamal signature methods that have been obtained by different rearrangements of the equation for the second component; see [HMP1994], [HP1994], [NR1994], [A1992], and [AMV1990]. These generalizations are independent of the mathematical group used, and have been described for the multiplicative group modulo a prime number, the multiplicative group of GF(2^w), and elliptic curve groups [HMP1994] [NR1994] [AMV1990] [A1992].

The Digital Signature Algorithm (DSA) [FIPS186] is an important ElGamal signature variant.

5.2. Hash Functions

ElGamal signatures must use a collision-resistant hash function, so that it can sign messages of arbitrary length and can avoid existential forgery attacks; see Section 10.4. (This is true for all ElGamal variants [HMP1994].) We denote the hash function as h(). Its input is a bit string of arbitrary length, and its output is a non-negative integer.

Let H() denote a hash function whose output is a fixed-length bit string. To use H in an ElGamal signature method, we define the mapping between that output and the non-negative integers; this realizes the function h() described above. Given a bit string m, the function h(m) is computed as follows:

1. H(m) is evaluated; the result is a fixed-length bit string.

2. Convert the resulting bit string to an integer i by treating its leftmost (initial) bit as the most significant bit of i, and treating its rightmost (final) bit as the least significant bit of i.

5.3. KT-IV Signatures

Koyama and Tsuruoka described a signature method based on Elliptic Curve ElGamal, in which the first signature component is the x-coordinate of an elliptic curve point reduced modulo q [KT1994]. In this section, we recall that method, which we refer to as KT-IV.

The algorithm uses an elliptic curve group, as described in Section 3.3, with prime field order p and curve equation parameters a and b. We denote the generator as alpha, and the order of the generator as q. We follow [FIPS186] in checking for exceptional cases.

5.3.1. Keypair Generation

The private key z is an integer between 1 and q-1, inclusive, generated uniformly at random. (See Appendix B regarding random integers.) The public key is the group element Y = alpha^z. Each public key is associated with a particular parameter set as per Section 3.3.

5.3.2. Signature Creation

To compute a KT-IV signature for a message m using the private key z:

1. Choose an integer k uniformly at random from the set of all integers between 1 and q-1, inclusive. (See Appendix B regarding random integers.)

2. Calculate R = (r_x, r_y) = alpha^k.

3. Calculate s1 = r_x mod q.

4. Check if h(m) + z * s1 = 0 mod q; if so, a new value of k MUST be generated and the signature MUST be recalculated. As an option, one MAY check if s1 = 0; if so, a new value of k SHOULD be generated and the signature SHOULD be recalculated. (It is extremely unlikely that s1 = 0 or h(m) + z * s1 = 0 mod q if signatures are generated properly.)

5. Calculate s2 = k/(h(m) + z*s1) mod q.

The signature is the ordered pair (s1, s2). Both signature components are non-negative integers.

5.3.3. Signature Verification

Given the message m, the generator g, the group order q, the public key Y, and the signature (s1, s2), verification is as follows:

1. Check to see that 0 < s1 < q and 0 < s2 < q; if either condition is violated, the signature SHALL be rejected.

2. Compute the non-negative integers u1 and u2, where

   u1 = h(m) * s2 mod q, and

   u2 = s1 * s2 mod q.

3. Compute the elliptic curve point R’ = alpha^u1 * Y^u2.

4. If the x-coordinate of R' mod q is equal to s1, then the signature and message pass the verification; otherwise, they fail.

5.4. KT-I Signatures

Horster, Michels, and Petersen categorized many different ElGamal signature methods, demonstrated their equivalence, and showed how to convert signatures of one type to another type [HMP1994]. In their terminology, the signature method of Section 5.3 and [KT1994] is a Type IV method, which is why it is denoted as KT-IV.

A Type I KT signature method has a second component that is computed in the same manner as that of the Digital Signature Algorithm. In this section, we describe this method, which we refer to as KT-I.

5.4.1. Keypair Generation

Keypairs and keypair generation are exactly as in Section 5.3.1.

5.4.2. Signature Creation

To compute a KT-I signature for a message m using the private key z:

1. Choose an integer k uniformly at random from the set of all integers between 1 and q-1, inclusive. (See Appendix B regarding random integers.)

2. Calculate R = (r_x, r_y) = alpha^k.

3. Calculate s1 = r_x mod q.

4. Calculate s2 = (h(m) + z*s1)/k mod q.

5. As an option, one MAY check if s1 = 0 or s2 = 0. If either s1 = 0 or s2 = 0, a new value of k SHOULD be generated and the signature SHOULD be recalculated. (It is extremely unlikely that s1 = 0 or s2 = 0 if signatures are generated properly.)

The signature is the ordered pair (s1, s2). Both signature components are non-negative integers.

5.4.3. Signature Verification

Given the message m, the public key Y, and the signature (s1, s2), verification is as follows:

1. Check to see that 0 < s1 < q and 0 < s2 < q; if either condition is violated, the signature SHALL be rejected.

2. Compute s2_inv = 1/s2 mod q.

3. Compute the non-negative integers u1 and u2, where

   u1 = h(m) * s2_inv mod q, and

   u2 = s1 * s2_inv mod q.

4. Compute the elliptic curve point R' = alpha^u1 * Y^u2.

5. If the x-coordinate of R' mod q is equal to s1, then the signature and message pass the verification; otherwise, they fail.

5.5. Converting KT-IV Signatures to KT-I Signatures

A KT-IV signature for a message m and a public key Y can easily be converted into a KT-I signature for the same message and public key. If (s1, s2) is a KT-IV signature for a message m, then (s1, 1/s2 mod q) is a KT-I signature for the same message [HMP1994].

The conversion operation uses only public information, and it can be performed by the creator of the pre-conversion KT-IV signature, the verifier of the post-conversion KT-I signature, or by any other entity.

An implementation MAY use this method to compute KT-I signatures.

5.6. Rationale

This subsection is not normative for this specification and is provided only as background information.

[HMP1994] presents many generalizations of ElGamal signatures. Equation (5) of that reference shows the general signature equation

A = x_A * B + k * C (mod q)

where x_A is the private key, k is the secret value, and A, B, and C are determined by the Type of the equation, as shown in Table 1 of [HMP1994]. DSA [FIPS186] is an EG-I.1 signature method (as is KT-I), with A = m, B = -r, and C = s. (Here we use the notation of [HMP1994] in which the first signature component is r and the second signature component is s; in KT-I and KT-IV these components are

denoted as s1 and s2, respectively. The private key x_A corresponds to the private key z.) Its signature equation is

$$m = -r * z + s * k \pmod{q}.$$

The signature method of [KT1994] and Section 5.3 is an EG-IV.1 method, with A = m * s, B = -r * s, C = 1. Its signature equation is

$$m * s = -r * s * z + k \pmod{q}$$

The functions f and g mentioned in Table 1 of [HMP1994] are merely multiplication, as described under the heading "Fifth generalization".

In the above equations, we rely on the implicit conversion of the message m from a bit string to an integer. No hash function is shown in these equations, but, as described in Section 10.4, a hash function should be applied to the message prior to signing in order to prevent existential forgery attacks.

Nyberg and Rueppel [NR1994] studied many different ElGamal signature methods and defined "strong equivalence" as follows:

> Two signature methods are called strongly equivalent if the signature of the first scheme can be transformed efficiently into signatures of the second scheme and vice versa, without knowledge of the private key.

KT-I and KT-IV signatures are obviously strongly equivalent.

A valid signature with s2=0 leaks the secret key, since in that case z = -h(m) / s1 mod q. We follow [FIPS186] in checking for this exceptional case and the case that s1=0. The s2=0 check was suggested by Rivest [R1992] and is discussed in [BS1992].

[KT1994] uses "a positive integer q' that does not exceed q" when computing the signature component s1 from the x-coordinate r_x of the elliptic curve point R = (r_x, r_y). The value q' is also used during signature validation when comparing the x-coordinate of a computed elliptic curve point to the value to s1. In this note, we use the simplifying convention that q' = q.

6. Converting between Integers and Octet Strings

A method for the conversion between integers and octet strings is specified in this section, following the established conventions of public key cryptography [R1993]. This method allows integers to be represented as octet strings that are suitable for transmission or

storage. This method SHOULD be used when representing an elliptic curve point or an elliptic curve coordinate as they are defined in this note.

6.1. Octet-String-to-Integer Conversion

The octet string S shall be converted to an integer x as follows. Let S1, ..., Sk be the octets of S from first to last. Then the integer x shall satisfy

$$x = \sum_{i=1}^{k} 2^{(8(k-i))} Si \ .$$

In other words, the first octet of S has the most significance in the integer and the last octet of S has the least significance.

Note: the integer x satisfies $0 <= x < 2^{(8*k)}$.

6.2. Integer-to-Octet-String Conversion

%%% The integer x shall be converted to an octet string S of length k as follows. The string S shall satisfy

$$y = \sum_{i=1}^{k} 2^{(8(k-i))} Si \ ,$$

where S1, ..., Sk are the octets of S from first to last. Note that the conversion fails if $y >= 2^{(8*k)}$.

In other words, the first octet of S has the most significance in the integer, and the last octet of S has the least significance.

7. Interoperability

The algorithms in this note can be used to interoperate with some other ECC specifications. This section provides details for each algorithm.

7.1. ECDH

Section 4 can be used with the Internet Key Exchange (IKE) versions one [RFC2409] or two [RFC5996]. These algorithms are compatible with the ECP groups defined by [RFC5903], [RFC5114], [RFC2409], and [RFC2412]. The group definition in this protocol uses an affine coordinate representation of the public key. [RFC5903] uses the compact output of Section 4.2, while [RFC4753] (which was obsoleted

by RFC 5903) does not. Neither of those RFCs use compact representation. Note that some groups indicate that the curve parameter "a" is negative; these values are to be interpreted modulo the order of the field. For example, a parameter of a = -3 is equal to p - 3, where p is the order of the field. The test cases in Section 8 of [RFC5903] can be used to test an implementation; these cases use the multiplicative notation, as does this note. The KEi and KEr payloads are equal to g^j and g^k, respectively, with 64 bits of encoding data prepended to them.

The algorithms in Section 4 can be used to interoperate with the IEEE [P1363] and ANSI [X9.62] standards for ECDH based on fields of characteristic greater than three. IEEE P1363 ECDH can be used in a manner that will interoperate with this note, with the following options and parameter choices from that specification:

prime curves with a cofactor of 1,

the ECSVDP-DH (Elliptic Curve Secret Value Derivation Primitive, Diffie-Hellman version),

the Key Derivation Function (KDF) must be the "identity" function (equivalently, the KDF step should be omitted and the shared secret value should be output directly).

7.2. KT-I and ECDSA

The Digital Signature Algorithm (DSA) is based on the discrete logarithm problem over the multiplicative subgroup of the finite field with large prime order [DSA1991] [FIPS186]. The Elliptic Curve Digital Signature Algorithm (ECDSA) [P1363] [X9.62] is an elliptic curve version of DSA.

%%% For many hash functions KT-I is mathematically and functionally equivalent to ECDSA, and can interoperate with the IEEE [P1363] and ANSI [X9.62] standards for Elliptic Curve DSA (ECDSA) based on fields of characteristic greater than three. KT-I signatures can be verified using the ECDSA verification algorithm, and ECDSA signatures can be verified using the KT-I verification algorithm (refer to Section 10.4).

8. Validating an Implementation

It is essential to validate the implementation of a cryptographic algorithm. This section outlines tests that should be performed on the algorithms defined in this note.

A known answer test, or KAT, uses a fixed set of inputs to test an algorithm; the output of the algorithm is compared with the expected output, which is also a fixed value. KATs for ECDH and KT-I are set out in the following subsections.

A consistency test generates inputs for one algorithm being tested using a second algorithm that is also being tested, then checks the output of the first algorithm. A signature creation algorithm can be tested for consistency against a signature verification algorithm. Implementations of KT-I should be tested in this way. Their signature generation processes are non-deterministic, and thus cannot be tested using a KAT. Signature verification algorithms, on the other hand, are deterministic and should be tested via a KAT. This combination of tests provides coverage for all of the operations, including keypair generation. Consistency testing should also be applied to ECDH.

8.1. ECDH

An ECDH implementation can be validated using the known answer test cases from [RFC5903] or [RFC5114]. The correspondence between the notation in RFC 5903 and the notation in this note is summarized in the following table. (Refer to Sections 3.3 and 4; the generator g is expressed in affine coordinate representation as (gx, gy)).

ECDH	RFC 5903
order p of field Fp	p
curve coefficient a	-3
curve coefficient b	b
generator g	g=(gx, gy)
private keys j and k	i and r
public keys g^j, g^k	g^i = (gix, giy) and g^r = (grx, gry)

The correspondence between the notation in RFC 5114 and the notation in this note is summarized in the following table.

ECDH	RFC 5114
order p of field Fp	p
curve coefficient a	a
curve coefficient b	b
generator g	g=(gx, gy)
group order n	n
private keys j and k	dA and dB
public keys g^j, g^k	g^(dA) = (x_qA, y_qA) and g^(dB) = (x_qB, y_qB)
shared secret g^(j*k)	g^(dA*dB) = (x_Z, y_Z)

8.2. KT-I

A KT-I implementation can be validated using the known answer test cases from [RFC4754]. The correspondence between the notation in that RFC and the notation in this note is summarized in the following table.

KT-I	RFC 4754
order p of field Fp	p
curve coefficient a	-3
curve coefficient b	b
generator alpha	g
group order q	q
private key z	w
public key Y	g^w = (gwx,gwy)
random k	ephem priv k
s1	r
s2	s
s2_inv	sinv
u1	u = h*sinv mod q
u2	v = r*sinv mod q

9. Intellectual Property

Concerns about intellectual property have slowed the adoption of ECC because a number of optimizations and specialized algorithms have been patented in recent years.

All of the normative references for ECDH (as defined in Section 4) were published during or before 1989, and those for KT-I were

published during or before May 1994. All of the normative text for these algorithms is based solely on their respective references.

9.1. Disclaimer

This document is not intended as legal advice. Readers are advised to consult their own legal advisers if they would like a legal interpretation of their rights.

The IETF policies and processes regarding intellectual property and patents are outlined in [RFC3979] and [RFC4879] and at https://datatracker.ietf.org/ipr/about/.

10. Security Considerations

The security level of an elliptic curve cryptosystem is determined by the cryptanalytic algorithm that is the least expensive for an attacker to implement. There are several algorithms to consider.

The Pohlig-Hellman method is a divide-and-conquer technique [PH1978]. If the group order n can be factored as

$$n = q1 * q2 * ... * qz,$$

then the discrete log problem over the group can be solved by independently solving a discrete log problem in groups of order q1, q2, ..., qz, then combining the results using the Chinese remainder theorem. The overall computational cost is dominated by that of the discrete log problem in the subgroup with the largest order.

Shanks' algorithm [K1981v3] computes a discrete logarithm in a group of order n using O(sqrt(n)) operations and O(sqrt(n)) storage. The Pollard rho algorithm [P1978] computes a discrete logarithm in a group of order n using O(sqrt(n)) operations, with a negligible amount of storage, and can be efficiently parallelized [VW1994].

The Pollard lambda algorithm [P1978] can solve the discrete logarithm problem using O(sqrt(w)) operations and O(log(w)) storage, when the exponent is known to lie in an interval of width w.

The algorithms described above work in any group. There are specialized algorithms that specifically target elliptic curve groups. There are no known subexponential algorithms against general elliptic curve groups, though there are methods that target certain special elliptic curve groups; see [MOV1993] and [FR1994].

10.1. Subgroups

A group consisting of a nonempty set of elements S with associated group operation * is a subgroup of the group with the set of elements G, if the latter group uses the same group operation and S is a subset of G. For each elliptic curve equation, there is an elliptic curve group whose group order is equal to the order of the elliptic curve; that is, there is a group that contains every point on the curve.

The order m of the elliptic curve is divisible by the order n of the group associated with the generator; that is, for each elliptic curve group, m = n * c for some number c. The number c is called the "cofactor" [P1363]. Each ECC parameter set as in Section 3.3 is associated with a particular cofactor.

It is possible and desirable to use a cofactor equal to 1.

10.2. Diffie-Hellman

Note that the key exchange protocol as defined in Section 4 does not protect against active attacks; Party A must use some method to ensure that (g^k) originated with the intended communicant B, rather than an attacker, and Party B must do the same with (g^j).

It is not sufficient to authenticate the shared secret g^(j*k), since this leaves the protocol open to attacks that manipulate the public keys. Instead, the values of the public keys g^x and g^y that are exchanged should be directly authenticated. This is the strategy used by protocols that build on Diffie-Hellman and that use end-entity authentication to protect against active attacks, such as OAKLEY [RFC2412] and the Internet Key Exchange [RFC2409] [RFC4306] [RFC5996].

When the cofactor of a group is not equal to 1, there are a number of attacks that are possible against ECDH. See [VW1996], [AV1996], and [LL1997].

10.3. Group Representation and Security

The elliptic curve group operation does not explicitly incorporate the parameter b from the curve equation. This opens the possibility that a malicious attacker could learn information about an ECDH private key by submitting a bogus public key [BMM2000]. An attacker can craft an elliptic curve group G’ that has identical parameters to a group G that is being used in an ECDH protocol, except that b is different. An attacker can submit a point on G’ into a run of the ECDH protocol that is using group G, and gain information from the

fact that the group operations using the private key of the device under attack are effectively taking place in G' instead of G.

This attack can gain useful information about an ECDH private key that is associated with a static public key, i.e., a public key that is used in more than one run of the protocol. However, it does not gain any useful information against ephemeral keys.

This sort of attack is thwarted if an ECDH implementation does not assume that each pair of coordinates in Zp is actually a point on the appropriate elliptic curve.

These considerations also apply when ECDH is used with compact representation (see Appendix C).

10.4. Signatures

Elliptic curve parameters should only be used if they come from a trusted source; otherwise, some attacks are possible [AV1996] [V1996].

If no hash function is used in an ElGamal signature system, then the system is vulnerable to existential forgeries, in which an attacker who does not know a private key can generate valid signatures for the associated public key, but cannot generate a signature for a message of its own choosing. (See [E1985] for instance.) The use of a collision-resistant hash function eliminates this vulnerability.

In principle, any collision-resistant hash function is suitable for use in KT signatures. To facilitate interoperability, we recognize the following hashes as suitable for use as the function H defined in Section 5.2:

SHA-256, which has a 256-bit output.

SHA-384, which has a 384-bit output.

SHA-512, which has a 512-bit output.

All of these hash functions are defined in [FIPS180-2].

The number of bits in the output of the hash used in KT signatures should be equal or close to the number of bits needed to represent the group order.

11. Acknowledgements

This update to RFC 6090 includes errata that were reported against that RFC. The authors gratefully acknowledge Annie Yousar and Watson Ladd for those reports.

The authors also expresses their thanks to the originators of elliptic curve cryptography, whose work made this note possible, and all of the reviewers, who provided valuable constructive feedback. Thanks for RFC 6090 are especially due to Howard Pinder, Andrey Jivsov, Alfred Hoenes (who contributed the algorithms in Appendix F), Dan Harkins, and Tina Tsou.

12. References

12.1. Normative References

[AMV1990] Agnew, G., Mullin, R., and S. Vanstone, "Improved Digital Signature Scheme based on Discrete Exponentiation", Electronics Letters Vol. 26, No. 14, July, 1990.

[BC1989] Bender, A. and G. Castagnoli, "On the Implementation of Elliptic Curve Cryptosystems", Advances in Cryptology - CRYPTO ’89 Proceedings, Springer Lecture Notes in Computer Science (LNCS), volume 435, 1989.

[CC1986] Chudnovsky, D. and G. Chudnovsky, "Sequences of numbers generated by addition in formal groups and new primality and factorization tests", Advances in Applied Mathematics, Volume 7, Issue 4, December 1986.

[D1966] Deskins, W., "Abstract Algebra", MacMillan Company New York, 1966.

[DH1976] Diffie, W. and M. Hellman, "New Directions in Cryptography", IEEE Transactions in Information Theory IT-22, pp. 644-654, 1976.

[FR1994] Frey, G. and H. Ruck, "A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves.", Mathematics of Computation Vol. 62, No. 206, pp. 865-874, 1994.

[HMP1994] Horster, P., Michels, M., and H. Petersen, "Meta-ElGamal signature schemes", University of Technology Chemnitz-Zwickau Department of Computer Science, Technical Report TR-94-5, May 1994.

[K1981v2] Knuth, D., "The Art of Computer Programming, Vol. 2: Seminumerical Algorithms", Addison Wesley , 1981.

[K1987] Koblitz, N., "Elliptic Curve Cryptosystems", Mathematics of Computation, Vol. 48, 1987, pp. 203-209, 1987.

[KT1994] Koyama, K. and Y. Tsuruoka, "Digital signature system based on elliptic curve and signer device and verifier device for said system", Japanese Unexamined Patent Application Publication H6-43809, February 18, 1994.

[M1983] Massey, J., "Logarithms in finite cyclic groups - cryptographic issues", Proceedings of the 4th Symposium on Information Theory, 1983.

[M1985] Miller, V., "Use of elliptic curves in cryptography", Advances in Cryptology - CRYPTO ’85 Proceedings, Springer Lecture Notes in Computer Science (LNCS), volume 218, 1985.

[MOV1993] Menezes, A., Vanstone, S., and T. Okamoto, "Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field", IEEE Transactions on Information Theory Vol. 39, No. 5, pp. 1639-1646, September, 1993.

[R1993] RSA Laboratories, , "PKCS#1: RSA Encryption Standard", Technical Note version 1.5, 1993.

[S1986] Silverman, J., "The Arithmetic of Elliptic Curves", Springer-Verlag, New York, 1986.

12.2. Informative References

[A1992] Anderson, J., "Response to the proposed DSS", Communications of the ACM, v. 35, n. 7, p. 50-52, July 1992.

[AV1996] Anderson, R. and S. Vaudenay, "Minding Your P’s and Q’s", Advances in Cryptology - ASIACRYPT ’96 Proceedings, Springer Lecture Notes in Computer Science (LNCS), volume 1163, 1996.

[BMM2000] Biehl, I., Meyer, B., and V. Muller, "Differential fault analysis on elliptic curve cryptosystems", Advances in Cryptology - CRYPTO 2000 Proceedings, Springer Lecture Notes in Computer Science (LNCS), volume 1880, 2000.

[BS1992] Branstad, D. and M. Smid, "Response to Comments on the NIST Proposed Digital Signature Standard", Advances in Cryptology - CRYPTO '92 Proceedings, Springer Lecture Notes in Computer Science (LNCS), volume 740, August 1992.

[DSA1991] U.S. National Institute of Standards and Technology, , "DIGITAL SIGNATURE STANDARD", Federal Register, Vol. 56, August 1991.

[E1984a] ElGamal, T., "Cryptography and logarithms over finite fields", Stanford University, UMI Order No. DA 8420519, 1984.

[E1984b] ElGamal, T., "Cryptography and logarithms over finite fields", Advances in Cryptology - CRYPTO '84 Proceedings, Springer Lecture Notes in Computer Science (LNCS), volume 196, 1984.

[E1985] ElGamal, T., "A public key cryptosystem and a signature scheme based on discrete logarithms", IEEE Transactions on Information Theory, Vol. 30, No. 4, pp. 469-472, 1985.

[FIPS180-2] U.S. National Institute of Standards and Technology, , "SECURE HASH STANDARD", Federal Information Processing Standard (FIPS) 180-2, August 2002.

[FIPS186] U.S. National Institute of Standards and Technology, , "DIGITAL SIGNATURE STANDARD", Federal Information Processing Standard FIPS-186, May 1994.

[HP1994] Horster, P. and H. Petersen, "Verallgemeinerte ElGamal-Signaturen", Proceedings der Fachtagung SIS '94, Verlag der Fachvereine, Zurich, 1994.

[K1981v3] Knuth, D., "The Art of Computer Programming, Vol. 3: Sorting and Searching", Addison Wesley, 1981.

[KMOV1991] Koyama, K., Maurer, U., Vanstone, S., and T. Okamoto, "New Public-Key Schemes Based on Elliptic Curves over the Ring Zn", Advances in Cryptology - CRYPTO '91 Proceedings, Springer Lecture Notes in Computer Science (LNCS), volume 576, 1991.

[L1969] Lehmer, D., "Computer technology applied to the theory of numbers", M.A.A. Studies in Mathematics, 180-2, 1969.

[LL1997] Lim, C. and P. Lee, "A Key Recovery Attack on Discrete Log-based Schemes Using a Prime Order Subgroup", Advances in Cryptology - CRYPTO '97 Proceedings, Springer Lecture Notes in Computer Science (LNCS), volume 1294, 1997.

[NR1994] Nyberg, K. and R. Rueppel, "Message Recovery for Signature Schemes Based on the Discrete Logarithm Problem", Advances in Cryptology - EUROCRYPT '94 Proceedings, Springer Lecture Notes in Computer Science (LNCS), volume 950, May 1994.

[P1363] "Standard Specifications for Public Key Cryptography", Institute of Electric and Electronic Engineers (IEEE), P1363, 2000.

[P1978] Pollard, J., "Monte Carlo methods for index computation mod p", Mathematics of Computation, Vol. 32, 1978.

[PH1978] Pohlig, S. and M. Hellman, "An Improved Algorithm for Computing Logarithms over GF(p) and its Cryptographic Significance", IEEE Transactions on Information Theory, Vol. 24, pp. 106-110, 1978.

[R1988] Rose, H., "A Course in Number Theory", Oxford University Press, 1988.

[R1992] Rivest, R., "Response to the proposed DSS", Communications of the ACM, v. 35, n. 7, p. 41-47, July 1992.

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

[RFC2409] Harkins, D. and D. Carrel, "The Internet Key Exchange (IKE)", RFC 2409, November 1998.

[RFC2412] Orman, H., "The OAKLEY Key Determination Protocol", RFC 2412, November 1998.

[RFC3979] Bradner, S., "Intellectual Property Rights in IETF Technology", BCP 79, RFC 3979, March 2005.

[RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[RFC4306] Kaufman, C., "Internet Key Exchange (IKEv2) Protocol", RFC 4306, December 2005.

[RFC4753] Fu, D. and J. Solinas, "ECP Groups For IKE and IKEv2", RFC 4753, January 2007.

[RFC4754] Fu, D. and J. Solinas, "IKE and IKEv2 Authentication Using the Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 4754, January 2007.

[RFC4879] Narten, T., "Clarification of the Third Party Disclosure Procedure in RFC 3979", BCP 79, RFC 4879, April 2007.

[RFC5114] Lepinski, M. and S. Kent, "Additional Diffie-Hellman Groups for Use with IETF Standards", RFC 5114, January 2008.

[RFC5903] Fu, D. and J. Solinas, "Elliptic Curve Groups modulo a Prime (ECP Groups) for IKE and IKEv2", RFC 5903, June 2010.

[RFC5996] Kaufman, C., Hoffman, P., Nir, Y., and P. Eronen, "Internet Key Exchange Protocol Version 2 (IKEv2)", RFC 5996, September 2010.

[RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, February 2011.

[SuiteB] U. S. National Security Agency (NSA), , "NSA Suite B Cryptography", 2014, <http://www.nsa.gov/ia/programs/suiteb_cryptography/index.shtml>.

[V1996] Vaudenay, S., "Hidden Collisions on DSS", Advances in Cryptology - CRYPTO ’96 Proceedings, Springer Lecture Notes in Computer Science (LNCS), volume 1109, 1996.

[VW1994] van Oorschot, P. and M. Wiener, "Parallel Collision Search with Application to Hash Functions and Discrete Logarithms", Proceedings of the 2nd ACM Conference on Computer and communications security, pp. 210-218, 1994.

[VW1996] van Oorschot, P. and M. Wiener, "On Diffie-Hellman key agreement with short exponents", Advances in Cryptology - EUROCRYPT ’96 Proceedings, Springer Lecture Notes in Computer Science (LNCS), volume 1070, 1996.

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

Appendix A. Key Words

The definitions of these key words are quoted from [RFC2119] and are commonly used in Internet standards. They are reproduced in this note in order to avoid a normative reference from after 1994.

1. MUST - This word, or the terms "REQUIRED" or "SHALL", means that the definition is an absolute requirement of the specification.

2. MUST NOT - This phrase, or the phrase "SHALL NOT", means that the definition is an absolute prohibition of the specification.

3. SHOULD - This word, or the adjective "RECOMMENDED", means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

4. SHOULD NOT - This phrase, or the phrase "NOT RECOMMENDED", means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

5. MAY - This word, or the adjective "OPTIONAL", means that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides.)

Appendix B. Random Integer Generation

It is easy to generate an integer uniformly at random between zero and (2^t)-1, inclusive, for some positive integer t. Generate a random bit string that contains exactly t bits, and then convert the bit string to a non-negative integer by treating the bits as the coefficients in a base-2 expansion of an integer.

It is sometimes necessary to generate an integer r uniformly at random so that r satisfies a certain property P, for example, lying

within a certain interval. A simple way to do this is with the rejection method:

1. Generate a candidate number c uniformly at random from a set that includes all numbers that satisfy property P (plus some other numbers, preferably not too many)

2. If c satisfies property P, then return c. Otherwise, return to Step 1.

For example, to generate a number between 1 and n-1, inclusive, repeatedly generate integers between zero and (2^t)-1, inclusive, stopping at the first integer that falls within that interval.

Recommendations on how to generate random bit strings are provided in [RFC4086].

Appendix C. Why Compact Representation Works

In the affine representation, the x-coordinate of the point P^i does not depend on the y-coordinate of the point P, for any non-negative exponent i and any point P. This fact can be seen as follows. When given only the x-coordinate of a point P, it is not possible to determine exactly what the y-coordinate is, but the y value will be a solution to the curve equation

$$y^2 = x^3 + a*x + b \pmod{p}.$$

There are at most two distinct solutions y = w and y = -w mod p, and the point P must be either Q=(x,w) or Q^-1=(x,-w). Thus P^n is equal to either Q^n or (Q^-1)^n = (Q^n)^-1. These values have the same x-coordinate. Thus, the x-coordinate of a point P^i can be computed from the x-coordinate of a point P by computing one of the possible values of the y coordinate of P, then computing the ith power of P, and then ignoring the y-coordinate of that result.

In general, it is possible to compute a square root modulo p by using Shanks' method [K1981v2]; simple methods exist for some values of p. When p = 3 (mod 4), the square roots of z mod p are w and -w mod p, where

$$w = z \wedge ((p+1)/4) \pmod{p};$$

this observation is due to Lehmer [L1969]. When p satisfies this property, y can be computed from the curve equation, and either y = w or y = -w mod p, where

$$w = (x^3 + a*x + b)^((p+1)/4) \pmod{p}.$$

Square roots modulo p only exist for a quadratic residue modulo p, [R1988]; if z is not a quadratic residue, then there is no number w such that w^2 = z (mod p). A simple way to verify that z is a quadratic residue after computing w is to verify that w * w = z (mod p). If this relation does not hold for the above equation, then the value x is not a valid x-coordinate for a valid elliptic curve point. This is an important consideration when ECDH is used with compact output; see Section 10.3.

The primes used in the P-256, P-384, and P-521 curves described in [RFC5903] all have the property that p = 3 (mod 4).

Appendix D. Example ECC Parameter Set

For concreteness, we recall an elliptic curve defined by Solinas and Fu in [RFC5903] and referred to as P-256, which is believed to provide a 128-bit security level. We use the notation of Section 3.3, and express the generator in the affine coordinate representation g=(gx,gy), where the values gx and gy are in Fp.

p: FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF

a: - 3

b: 5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B

n: FFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551

gx: 6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296

gy: 4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5

Note that p can also be expressed as

$$p = 2^{256}-2^{224}+2^{192}+2^{96}-1.$$

Appendix E. Additive and Multiplicative Notation

The early publications on elliptic curve cryptography used multiplicative notation, but most modern publications use additive notation. This section includes a table mapping between those two conventions. In this section, a and b are elements of an elliptic curve group, and N is an integer.

Multiplicative Notation	Additive Notation
multiplication	addition
a * b	a + b
squaring	doubling
a * a = a^2	a + a = 2a
exponentiation	scalar multiplication
a^N = a * a * ... * a	Na = a + a + ... + a
inverse	inverse
a^-1	-a

Appendix F. Algorithms

This section contains a pseudocode description of the elliptic curve group operation. Text that follows the symbol "//" is to be interpreted as comments rather than instructions.

F.1. Affine Coordinates

To an arbitrary pair of elliptic curve points P and Q specified by their affine coordinates P=(x1,y1) and Q=(x2,y2), the group operation assigns a third point R = P*Q with the coordinates (x3,y3). These coordinates are computed as follows:

```
if P is (@,@),
   R = Q
else if Q is (@,@),
   R = P
else if P is not equal to Q and x1 is equal to x2,
   R = (@,@)
else if P is not equal to Q and x1 is not equal to x2,
   x3 = ((y2-y1)/(x2-x1))^2 - x1 - x2 mod p and
   y3 = (x1-x3)*(y2-y1)/(x2-x1) - y1 mod p
else if P is equal to Q and y1 is equal to 0,
   R = (@,@)
else    // P is equal to Q and y1 is not equal to 0
   x3 = ((3*x1^2 + a)/(2*y1))^2 - 2*x1 mod p and
   y3 = (x1-x3)*(3*x1^2 + a)/(2*y1) - y mod p.
```

From the first and second case, it follows that the point at infinity is the neutral element of this operation, which is its own inverse.

From the curve equation, it follows that for a given curve point P = (x,y) distinct from the point at infinity, (x,-y) also is a curve point, and from the third and the fifth case it follows that this is the inverse of P, P^-1.

Note: The fifth and sixth case are known as "point squaring".

F.2. Homogeneous Coordinates

An elliptic curve point (x,y) (other than the point at infinity (@,@)) is equivalent to a point (X,Y,Z) in homogeneous coordinates (with X, Y, and Z in Fp and not all three being zero at once) whenever x=X/Z and y=Y/Z. "Homogenous coordinates" means that two triples (X,Y,Z) and (X',Y',Z') are regarded as "equal" (i.e., representing the same point) if there is some nonzero s in Fp such that X'=s*X, Y'=s*Y, and Z'=s*Z. The point at infinity (@,@) is regarded as equivalent to the homogenous coordinates (0,1,0), i.e., it can be represented by any triple (0,Y,0) with nonzero Y in Fp.

Let P1=(X1,Y1,Z1) and P2=(X2,Y2,Z2) be points on the elliptic curve, and let u = Y2 * Z1 - Y1 * Z2 and v = X2 * Z1 - X1 * Z2.

We observe that the points P1 and P2 are equal if and only if u and v are both equal to zero. Otherwise, if either P1 or P2 are equal to the point at infinity, v is zero and u is nonzero (but the converse implication does not hold).

Then, the product P3=(X3,Y3,Z3) = P1 * P2 is given by:

```
  if P1 is the point at infinity,
     P3 = P2
  else if P2 is the point at infinity,
     P3 = P1

  %%%
  else if P1=-P2 as projective points
     P3 = (0,1,0)
  else if P1 does not equal P2
     X3 = v * (Z2 * (Z1 * u^2 - 2 * X1 * v^2) - v^3)
     Y3 = Z2 * (3 * X1 * u * v^2 - Y1 * v^3 - Z1 * u^3) + u * v^3
     Z3 = v^3 * Z1 * Z2
  else    // P2 equals P1, P3 = P1 * P1
      w = 3 * X1^2 + a * Z1^2
     X3 = 2 * Y1 * Z1 * (w^2 - 8 * X1 * Y1^2 * Z1)
     Y3 = 4 * Y1^2 * Z1 * (3 * w * X1 - 2 * Y1^2 * Z1) - w^3
     Z3 = 8 * (Y1 * Z1)^3
```

It thus turns out that the point at infinity is the identity element and for P1=(X,Y,Z) not equal to this point at infinity, P2=(X,-Y,Z) represents P1^-1.

Authors' Addresses

David A. McGrew
Cisco Systems
510 McCarthy Blvd.
Milpitas, CA 95035
USA

Phone: (408) 525 8651
Email: mcgrew@cisco.com
URI: http://www.mindspring.com/~dmcgrew/dam.htm

Kevin M. Igoe
National Security Agency
Commercial Solutions Center
United States of America

Email: kmigoe@nsa.gov

Margaret Salter
National Security Agency
9800 Savage Rd.
Fort Meade, MD 20755-6709
USA

Email: misalte@nsa.gov

Paul Hoffman
VPN Consortium

Email: paul.hoffman@vpnc.org

Crypto Forum Research Group
Internet-Draft
Intended status: Informational
Expires: January 5, 2015

D. McGrew
M. Curcio
Cisco Systems

July 4, 2014

Hash-Based Signatures
draft-mcgrew-hash-sigs-02

Abstract

This note describes a digital signature system based on cryptographic hash functions, following the seminal work in this area. It specifies a one-time signature scheme based on the work of Lamport, Diffie, Winternitz, and Merkle (LDWM), and a general signature scheme, Merkle Tree Signatures (MTS). These systems provide asymmetric authentication without using large integer mathematics and can achieve a high security level. They are suitable for compact implementations, are relatively simple to implement, and naturally resist side-channel attacks. Unlike most other signature systems, hash-based signatures would still be secure even if it proves feasible for an attacker to build a quantum computer.

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 http://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 January 5, 2015.

Copyright Notice

Table of Contents

1. Introduction 5
1.1. Conventions Used In This Document 5

2. Notation 6
2.1. Data Types 6
2.1.1. Operators 6
2.1.2. Strings of w-bit elements 6
2.2. Functions 7

3. LDWM One-Time Signatures 9
3.1. Parameters 9
3.2. Hashing Functions 9
3.3. Signature Methods 10
3.4. Private Key 10
3.5. Public Key 11
3.6. Checksum 11
3.7. Signature Generation 12
3.8. Signature Verification 13
3.9. Notes 13
3.10. Formats 13

4. Merkle Tree Signatures 17
4.1. Private Key 17
4.2. MTS Public Key 17
4.3. MTS Signature 18
4.3.1. MTS Signature Generation 19
4.4. MTS Signature Verification 19
4.5. MTS Formats 20

5. Rationale 23

6. History 24

7. IANA Considerations 25

8. Security Considerations 28
8.1. Security of LDWM Checksum 29
8.2. Security Conjectures 29
8.3. Post-Quantum Security 30

9. Acknowledgements 31

10. References 32
10.1. Normative References 32
10.2. Informative References 32

Appendix A. LDWM Parameter Options 33

Appendix B. Example Data for Testing 35
B.1. Parameters 35
B.2. Key Generation 35
B.3. Signature Generation 41
B.4. Signature Verification 45
B.5. Intermediate Calculation Values 45

Authors' Addresses 51

1. Introduction

One-time signature systems, and general purpose signature systems built out of one-time signature systems, have been known since 1979 [Merkle79], were well studied in the 1990s, and have benefited from renewed development in the last decade. The characteristics of these signature systems are small private and public keys and fast signature generation and verification, but large signatures and relatively slow key generation. In recent years there has been interest in these systems because of their post-quantum security (see Section 8.3) and their suitability for compact implementations.

This note describes the original Lamport-Diffie-Winternitz-Merkle (LDWM) one-time signature system (following Merkle 1979 but also using a technique from Merkle's later work [C:Merkle87][C: Merkle89a][C:Merkle89b]) and Merkle tree signature system (following Merkle 1979) with enough specificity to ensure interoperability between implementations.

A signature system provides asymmetric message authentication. The key generation algorithm produces a public/private key pair. A message is signed by a private key, producing a signature, and a message/signature pair can be verified by a public key. A One-Time Signature (OTS) system can be used to sign exactly one message securely. A general signature system can be used to sign multiple messages. The Merkle Tree Signatures (MTS) is a general signature system that uses an OTS system as a component. In principle the MTS can be used with any OTS system, but in this note we describe its use with the LDWM system.

This note is structured as follows. Notation is introduced in Section 2. The LDWM signature system is described in Section 3, and the Merkle tree signature system is described in Section 4. Sufficient detail is provided to ensure interoperability. Appendix B describes test considerations and contains test cases that can be used to validate an implementation. The IANA registry for these signature systems is described in Section 7. Security considerations are presented in Section 8.

1.1. Conventions Used In This Document

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. Notation

2.1. Data Types

Bytes and byte strings are the fundamental data types. A single byte is denoted as a pair of hexadecimal digits with a leading "0x". A byte string is an ordered sequence of zero or more bytes and is denoted as an ordered sequence of hexadecimal characters with a leading "0x". For example, 0xe534f0 is a byte string with a length of three. An array of byte strings is an ordered set, indexed starting at zero, in which all strings have the same length.

2.1.1. Operators

When a and b are real numbers, mathematical operators are defined as follows:

^ : a ^ b denotes the result of a raised to the power of b

* : a * b denotes the product of a multiplied by b

/ : a / b denotes the quotient of a divided by b

% : a % b denotes the remainder of the integer division of a by b

+ : a + b denotes the sum of a and b

- : a - b denotes the difference of a and b

The standard order of operations is used when evaluating arithmetic expressions.

If A and B are bytes, then A AND B denotes the bitwise logical and operation.

When B is a byte and i is an integer, then B >> i denotes the logical right-shift operation. Similarly, B << i denotes the logical left-shift operation.

If S and T are byte strings, then S || T denotes the concatenation of S and T.

The i^th byte string in an array A is denoted as A[i].

2.1.2. Strings of w-bit elements

If S is a byte string, then byte(S, i) denotes its i^th byte, where byte(S, 0) is the leftmost byte. In addition, bytes(S, i, j) denotes

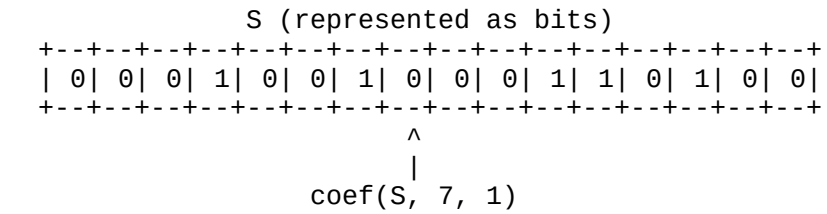
the range of bytes from the i^th to the j^th byte, inclusive. For example, if S = 0x02040608, then byte(S, 0) is 0x02 and bytes(S, 1, 2) is 0x0406.

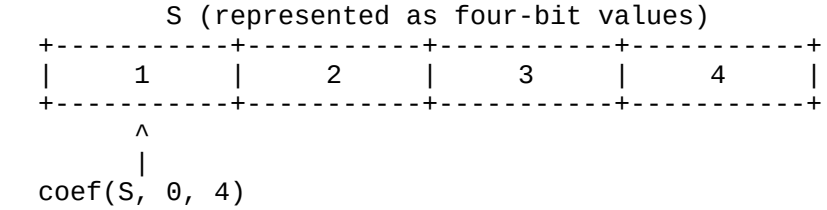
A byte string can be considered to be a string of w-bit unsigned integers; the correspondence is defined by the function coef(S, i, w) as follows:

If S is a string, i is a positive integer, and w is a member of the set { 1, 2, 4, 8 }, then coef(S, i, w) is the i^th, w-bit value, if S is interpreted as a sequence of w-bit values. That is,

```
coef(S, i, w) = (2^w - 1) AND
                ( byte(S, floor(i * w / 8)) >>
                  (8 - (w * (i % (8 / w)) + w)) )
```

For example, if S is the string 0x1234, then coef(S, 7, 1) is 0 and coef(S, 0, 4) is 1.





The return value of coef is an unsigned integer. If i is larger than the number of w-bit values in S, then coef(S, i, w) is undefined, and an attempt to compute that value should raise an error.

2.2. Functions

If r is a non-negative real number, then we define the following functions:

ceil(r) : returns the smallest integer larger than r

floor(r) : returns the largest integer smaller than r

lg(r) : returns the base-2 logarithm of r

When F is a function that takes r-byte strings as input and returns r-byte strings as output, we denote the repeated applications of F with itself a non-negative, integral number of times i as F^i.

Thus for any m-byte string x ,

$$F^i(x) = \begin{cases} F(F^{(i-1)}(x)) & \text{for } i > 0 \\ x & \text{for } i = 0. \end{cases}$$

For example, F^2(x) = F(F(x)).

3. LDWM One-Time Signatures

This section defines LDWM signatures. The signature is used to validate the authenticity of a message by associating a secret private key with a shared public key. These are one-time signatures; each private key MUST be used only one time to sign any given message.

As part of the signing process, a digest of the original message is computed using the collision-resistant hash function H (see Section 3.2), and the resulting digest is signed.

3.1. Parameters

The signature system uses the parameters m, n, and w; they are all positive integers. The algorithm description also uses the values p and ls. These parameters are summarized as follows:

m : the length in bytes of each element of an LDWM signature

n : the length in bytes of the result of the hash function

w : the Winternitz parameter; it is a member of the set { 1, 2, 4, 8 }

p : the number of m-byte string elements that make up the LDWM signature

ls : the number of left-shift bits used in the checksum function C (defined in Section 3.6).

The values of m and n are determined by the functions selected for use as part of the LDWM algorithm. They are chosen to ensure an appropriate level of security. The parameter w can be chosen to set the number of bytes in the signature; it has little effect on security. Note however, that there is a larger computational cost to generate and verify a shorter signature. The values of p and ls are dependent on the choices of the parameters n and w, as described in Appendix A. A table illustrating various combinations of n, w, p, and ls is provided in Table 4.

3.2. Hashing Functions

The LDWM algorithm uses a collision-resistant hash function H and a one way (preimage resistant) function F. H accepts byte strings of any length, and returns an n-byte string. F has m-byte inputs and m-byte outputs.

3.3. Signature Methods

To fully describe a LDWM signature method, the parameters m, n, and w, as well as the functions H and F MUST be specified. This section defines several LDWM signature systems, each of which is identified by a name. Values for p and ls are provided as a convenience.

Name	H	F	m	n	w	p	ls
LDWM_SHA512_M64_W1	SHA512	SHA512	32	32	1	265	7
LDWM_SHA512_M64_W2	SHA512	SHA512	32	32	2	133	6
LDWM_SHA512_M64_W4	SHA512	SHA512	32	32	4	67	4
LDWM_SHA512_M64_W8	SHA512	SHA512	32	32	8	34	0
LDWM_SHA256_M32_W1	SHA256	SHA256	32	32	1	265	7
LDWM_SHA256_M32_W2	SHA256	SHA256	32	32	2	133	6
LDWM_SHA256_M32_W4	SHA256	SHA256	32	32	4	67	4
LDWM_SHA256_M32_W8	SHA256	SHA256	32	32	8	34	0
LDWM_SHA256_M20_W1	SHA256	SHA256-20	20	32	1	265	7
LDWM_SHA256_M20_W2	SHA256	SHA256-20	20	32	2	133	6
LDWM_SHA256_M20_W4	SHA256	SHA256-20	20	32	4	67	4
LDWM_SHA256_M20_W8	SHA256	SHA256-20	20	32	8	34	0

Table 1

Here SHA512 and SHA256 denotes the NIST standard hash functions [FIPS180]. SHA256-20 denotes the SHA256 hash function with its final output truncated to return the leftmost 20 bytes.

3.4. Private Key

The LDWM private key is an array of size p containing m-byte strings. Let x denote the private key. This private key must be used to sign one and only one message. It must therefore be unique from all other private keys. The following algorithm shows pseudocode for generating x.

Algorithm 0: Generating a Private Key

```
for ( i = 0; i < p; i = i + 1 ) {
  set x[i] to a uniformly random m-byte string
}
return x
```

An implementation MAY use a pseudorandom method to compute x[i], as suggested in [Merkle79], page 46. The details of the pseudorandom method do not affect interoperability, but the cryptographic strength MUST match that of the LDWM algorithm.

3.5. Public Key

The LDWM public key is generated from the private key by applying the function F^(2^w - 1) to each individual element of x, then hashing all of the resulting values. The following algorithm shows pseudocode for generating the public key, where the array x is the private key.

Algorithm 1: Generating a Public Key From a Private Key

```
e = 2^w - 1
for ( i = 0; i < p; i = i + 1 ) {
  y[i] = F^e(x[i])
}
return H(y[0] || y[1] || ... || y[p-1])
```

3.6. Checksum

A checksum is used to ensure that any forgery attempt that manipulates the elements of an existing signature will be detected. The security property that it provides is detailed in Section 8.

The checksum value is calculated using a non-negative integer, sum, whose width is sized an integer number of w-bit fields such that it is capable of holding the difference of the total possible number of applications of the function F as defined in the signing algorithm of Section 3.7 and the total actual number. In the worst case (i.e. the actual number of times F is iteratively applied is 0), the sum is (2^w - 1) * ceil(8*n/w). Thus for the purposes of this document, which describes signature methods based on H = SHA256 (n = 32 bytes) and w = { 1, 2, 4, 8 }, let sum be a 16-bit non-negative integer for all combinations of n and w. The calculation uses the parameter ls defined in Section 3.1 and calculated in Appendix A, which indicates the number of bits used in the left-shift operation. The checksum function C is defined as follows, where S denotes the byte string that is input to that function.

Algorithm 2: Checksum Calculation

```
sum = 0
for ( i = 0; i < u; i = i + 1 ) {
  sum = sum + (2^w - 1) - coef(S, i, w)
}
return (sum << ls)
```

Because of the left-shift operation, the rightmost bits of the result of C will often be zeros. Due to the value of p, these bits will not be used during signature generation or verification.

Implementation Note: Based on the previous fact, an implementation MAY choose to optimize the width of sum to (v * w) bits and set ls to 0. The rationale for this is given that (2^w - 1) * ceil(8*n/w) is the maximum value of sum and the value of (2^w - 1) is represented by w bits, the result of adding u w-bit numbers, where u = ceil(8*n/w), requires at most (ceil(lg(u)) + w) bits. Dividing by w and taking the next largest integer gives the total required number of w-bit fields and gives (ceil(lg(u)) / w) + 1, or v. Thus sum requires a minimum width of (v * w) bits and no left-shift operation is performed.

3.7. Signature Generation

The LDWM signature is generated by using H to compute the hash of the message, concatenating the checksum of the hash to the hash itself, then considering the resulting value as a sequence of w-bit values, and using using each of the the w-bit values to determine the number of times to apply the function F to the corresponding element of the private key. The outputs of the function F are concatenated together and returned as the signature. The pseudocode for this procedure is shown below.

Algorithm 3: Generating a Signature From a Private Key and a Message

```
V = ( H(message) || C(H(message)) )
for ( i = 0; i < p; i = i + 1 ) {
  a = coef(V, i, w)
  y[i] = F^a(x[i])
}
return (y[0] || y[1] || ... || y[p-1])
```

Note that this algorithm results in a signature whose elements are intermediate values of the elements computed by the public key algorithm in Section 3.5.

The signature should be provided by the signer to the verifier, along

with the message and the public key.

3.8. Signature Verification

In order to verify a message with its signature (an array of m-byte strings, denoted as y), the receiver must "complete" the series of applications of F, using the w-bit values of the message hash and its checksum. This computation should result in a value that matches the provided public key.

Algorithm 4: Verifying a Signature and Message Using a Public Key

```
V = ( H(message) || C(H(message)) )
for ( i = 0; i < p; i = i + 1 ) {
  a = (2^w - 1) - coef(V, i, w)
  z[i] = F^a(y'[i])
}
if public key is equal to H(z[0] || z[1] || ... || z[p-1])
  return 1 (message signature is valid)
else
  return 0 (message signature is invalid)
```

3.9. Notes

A future version of this specification may define a method for computing the signature of a very short message in which the hash is not applied to the message during the signature computation. That would allow the signatures to have reduced size.

3.10. Formats

The signature and public key formats are formally defined using XDR [RFC4506] in order to provide an unambiguous, machine readable definition. For clarity, we also include a private key format as well, though consistency is not needed for interoperability and an implementation MAY use any private key format. Though XDR is used, these formats are simple and easy to parse without any special tools. To avoid the need to convert to and from network / host byte order, the enumeration values are all palindromes. The definitions are as follows:

```
/*
 * ots_algorithm_type identifies a particular signature algorithm
 */
enum ots_algorithm_type {
  ots_reserved        = 0,
  ldwm_sha256_m20_w1  = 0x01000001,
```

```
     ldwm_sha256_m20_w2  = 0x02000002,
     ldwm_sha256_m20_w4  = 0x03000003,
     ldwm_sha256_m20_w8  = 0x04000004,
     ldwm_sha256_m32_w1  = 0x05000005,
     ldwm_sha256_m32_w2  = 0x06000006,
     ldwm_sha256_m32_w4  = 0x07000007,
     ldwm_sha256_m32_w8  = 0x08000008,
     ldwm_sha512_m64_w1  = 0x09000009,
     ldwm_sha512_m64_w2  = 0x0a00000a,
     ldwm_sha512_m64_w4  = 0x0b00000b,
     ldwm_sha512_m64_w8  = 0x0c00000c
   };

   /*
    * byte string
    */
   typedef opaque bytestring20[20];
   typedef opaque bytestring32[32];
   typedef opaque bytestring64[64];

   union ots_signature switch (ots_algorithm_type type) {
    case ldwm_sha256_m20_w1:
         bytestring20 y_m20_p265[265];
    case ldwm_sha256_m20_w2:
         bytestring20 y_m20_p133[133];
    case ldwm_sha256_m20_w4:
         bytestring20 y_m20_p67[67];
    case ldwm_sha256_m20_w8:
         bytestring20 y_m20_p34[34];
    case ldwm_sha256_m32_w1:
         bytestring32 y_m32_p265[265];
    case ldwm_sha256_m32_w2:
         bytestring32 y_m3_p133[133];
    case ldwm_sha256_m32_w4:
         bytestring32 y_m32_y_p67[67];
    case ldwm_sha256_m32_w8:
         bytestring32 y_m32_p34[34];
    case ldwm_sha512_m64_w1:
         bytestring64 y_m64_p265[265];
    case ldwm_sha512_m64_w2:
         bytestring64 y_m64_p133[133];
    case ldwm_sha512_m64_w4:
         bytestring64 y_m64_y_p67[67];
    case ldwm_sha512_m64_w8:
         bytestring64 y_m64_p34[34];
    default:
      void;   /* error condition */
   };
```

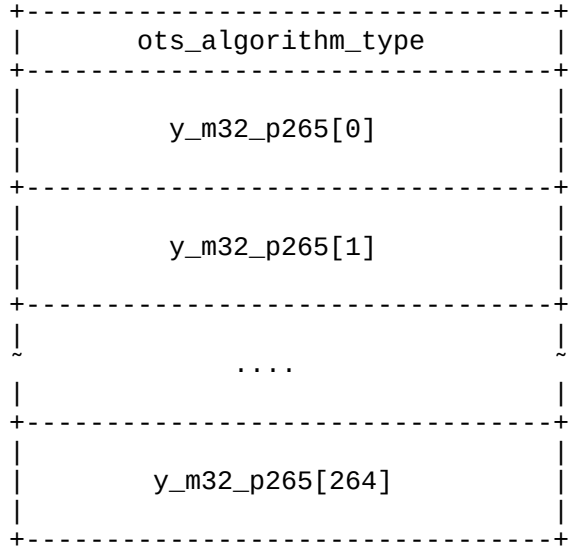
```
union ots_public_key switch (ots_algorithm_type type) {
 case ldwm_sha256_m20_w1:
 case ldwm_sha256_m20_w2:
 case ldwm_sha256_m20_w4:
 case ldwm_sha256_m20_w8:
 case ldwm_sha256_m32_w1:
 case ldwm_sha256_m32_w2:
 case ldwm_sha256_m32_w4:
 case ldwm_sha256_m32_w8:
      bytestring32 y32;
 case ldwm_sha512_m64_w1:
 case ldwm_sha512_m64_w2:
 case ldwm_sha512_m64_w4:
 case ldwm_sha512_m64_w8:
      bytestring64 y64;
 default:
   void;   /* error condition */
 };

union ots_private_key switch (ots_algorithm_type type) {
 case ldwm_sha256_m20_w1:
 case ldwm_sha256_m20_w2:
 case ldwm_sha256_m20_w4:
 case ldwm_sha256_m20_w8:
      bytestring20 x20;
 case ldwm_sha256_m32_w1:
 case ldwm_sha256_m32_w2:
 case ldwm_sha256_m32_w4:
 case ldwm_sha256_m32_w8:
      bytestring32 x32;
 case ldwm_sha512_m64_w1:
 case ldwm_sha512_m64_w2:
 case ldwm_sha512_m64_w4:
 case ldwm_sha512_m64_w8:
      bytestring64 y64;
 default:
   void;   /* error condition */
 };
```

Though the data formats are formally defined by XDR, we diagram the format as well as a convenience to the reader. An example of the format of an ldwm_signature is illustrated below, for ldwm_sha256_m32_w1. An ots_signature consists of a 32-bit unsigned integer that indicates the ots_algorithm_type, followed by other data, whose format depends only on the ots_algorithm_type. In the case of LDWM, the data is an array of equal-length byte strings. The number of bytes in each byte string, and the number of elements in the array, are determined by the ots_algorithm_type field. In the

case of ldwm_sha256_m32_w1, the array has 265 elements, each of which is a 32-byte string. The XDR array y_m32_p265 denotes the array y as used in the algorithm descriptions above, using the parameters of m=32 and p=265 for ldwm_sha256_m32_w1.

A verifier MUST check the ots_algorithm_type field, and a verification operation on a signature with an unknown ldwm_algorithm_type MUST return FAIL.



4. Merkle Tree Signatures

Merkle Tree Signatures (MTS) are a method for signing a potentially large but fixed number of messages. An MTS system uses two cryptographic components: a one-time signature method and a collision-resistant hash function. Each MTS public/private key pair is associated with a perfect k-ary tree, each node of which contains an n-byte value. Each leaf of the tree contains the value of the public key of an LDWM public/private key pair. The value contained by the root of the tree is the MTS public key. Each interior node is computed by applying the hash function to the concatenation of the values of its children nodes.

An MTS system has the following parameters:

k : the number of children nodes of an interior node,

h : the height (number of levels - 1) in the tree, and

n : the number of bytes associated with each node.

There are k^h leaves in the tree.

4.1. Private Key

An MTS private key consists of k^h one-time signature private keys and the leaf number of the next LDWM private key that has not yet been used. The leaf number is initialized to zero when the MTS private key is created.

An MTS private key MAY be generated pseudorandomly from a secret value, in which case the secret value MUST be at least n bytes long, be uniformly random, and MUST NOT be used for any other purpose than the generation of the MTS private key. The details of how this process is done do not affect interoperability; that is, the public key verification operation is independent of these details.

4.2. MTS Public Key

An MTS public key is defined as follows, where we denote the public key associated with the i^th LDWM private key as ldwm_public_key(i).

The MTS public key can be computed using the following algorithm or any equivalent method. The algorithm uses a stack of hashes for data and a separate stack of integers to keep track of the level of the Merkle tree.

Algorithm 5: Generating an MTS Public Key From an MTS Private Key

```
for ( i = 0; i < num_ldwm_keys; i = i + k ) {
  level = 0;
  for ( j = 0; j < k; j = j + 1 ) {
    push ldwm_public_key(i+j) onto the data stack
    push level onto the integer stack
  }
  while ( height of the integer stack >= k ) {
    if level of the top k elements on the integer stack are equal {
      hash_init()
      siblings = ""
      repeat ( k ) {
        siblings = (pop(data stack) || siblings)
        level = pop(integer stack)
      }
      hash_update(siblings)
      push hash_final() onto the data stack
      push (level + 1) onto the integer stack
    }
  }
}
public_key = pop(data stack)
```

Note that this pseudocode expects, as was defined earlier, the Merkle Tree to be perfect. That is, all h^k leaves of the tree have equal depth. Also, neither stack ever contains more than h*(k-1)+1 elements. For typical parameters, it will hold roughly 20 32-byte values.

4.3. MTS Signature

An MTS signature consists of

an LDWM signature,

a node number that identifies the leaf node associated with the signature, and

an array of values that is associated with the path through the tree from the leaf associated with the LDWM signature to the root.

The array of values contains contains the siblings of the nodes on the path from the leaf to the root but does not contain the nodes on the path itself. The array for a tree with branching number k and height h will have (k-1)*h values. The first (k-1) values are the siblings of the leaf, the next (k-1) values are the siblings of the parent of the leaf, and so on.

4.3.1. MTS Signature Generation

To compute the MTS signature of a message with an MTS private key, the signer first computes the LDWM signature of the message using the leaf number of the next unused LDWM private key. Before releasing the signature, the leaf number in the MTS private key MUST be incremented to prevent the LDWM private key from being used again. The node number in the signature is set to the leaf number of the MTS private key that was used in the signature.

The array of node values MAY be computed in any way. There are many potential time/storage tradeoffs. The fastest alternative is to store all of the nodes of the tree and set the array in the signature by copying them. The least storage intensive alternative is to recompute all of the nodes for each signature. Note that the details of this procedure are not important for interoperability; it is not necessary to know any of these details in order to perform the signature verification operation.

4.4. MTS Signature Verification

An MTS signature is verified by first using the LDWM signature verification algorithm to compute the LDWM public key from the LDWM signature and the message. The value of the leaf associated with the LDWM signature is assigned to the public key. Then the root of the tree is computed from the leaf value and the node array (path[]) as described below. If the root value matches the public key, then the signature is valid; otherwise, the signature fails.

Algorithm 6: Computing the MTS Root Value

```
n = node number
v = leaf
step = 0
for ( i = 0; i < h; i = i + 1 ) {
  position = n % k
  hash_init()
  for ( j = 0; j < position; j = j + 1 ) {
    hash_update(path[step + j])
  }
  hash_update(v)
  for ( j = position; j < (k-1); j = j + 1 ) {
    hash_update(path[step + j])
  }
  v = hash_final()
  n = floor(n/k)
  step = step + (k-1)
}
```

Upon completion, v contains the value of the root of the Merkle Tree for comparison.

This algorithm uses the typical init/update/final interface to hash functions; the result of the invocations hash_init(), hash_update(N[1]), hash_update(N[2]), ... , hash_update(N[n]), v = hash_final(), in that order, is identical to that of the invocation of H(N[1] || N[2] || ... || N[n]).

This algorithm works because the leaves of the MTS tree are numbered starting at zero. Therefore leaf n is in the position (n % k) in the highest level of the tree.

The verifier MAY cache interior node values that have been computed during a successful signature verification for use in subsequent signature verifications. However, any implementation that does so MUST make sure any nodes that are cached during a signature verification process are deleted if that process does not result in a successful match between the root of the tree and the MTS public key.

A full test example that combines the LDWM OTS and MTS algorithms is given in Appendix B.

4.5. MTS Formats

MTS signatures and public keys are defined using XDR syntax as follows:

```
  enum mts_algorithm_type {
    mts_reserved      = 0x00000000,
    mts_sha256_k2_h20 = 0x01000001,
    mts_sha256_k4_h10 = 0x02000002,
    mts_sha256_k8_h7  = 0x03000003,
    mts_sha256_k16_h5 = 0x04000004,
    mts_sha512_k2_h20 = 0x05000005,
    mts_sha512_k4_h10 = 0x06000006,
    mts_sha512_k8_h7  = 0x07000007,
    mts_sha512_k16_h5 = 0x08000008
  };

  union mts_path switch (mts_algorithm_type type) {
   case mts_sha256_k2_h20:
     bytestring32 path_n32_t20[20];
   case mts_sha256_k4_h10:
     bytestring32 path_n32_t30[30];
   case mts_sha256_k8_h7:
     bytestring32 path_n32_t49[49];
   case mts_sha256_k16_h5:
     bytestring32 path_n32_t75[75];
   case mts_sha512_k2_h20:
     bytestring64 path_n64_t20[20];
   case mts_sha512_k4_h10:
     bytestring64 path_n64_t30[30];
   case mts_sha512_k8_h7:
     bytestring64 path_n64_t49[49];
   case mts_sha512_k16_h5:
     bytestring64 path_n64_t75[75];
   default:
     void;     /* error condition */
  };

  struct mts_signature {
    ots_signature ots_sig;
    unsigned int signature_leaf_number;
    mts_path nodes;
  };

  struct mts_public_key_n32 {
    ots_algorithm_type ots_alg_type;
    opaque value[32];                   /* public key */
  };

  struct mts_public_key_n64 {
    ots_algorithm_type ots_alg_type;
    opaque value[64];                   /* public key */
  };
```

```
  union mts_public_key switch (mts_algorithm_type type) {
   case mts_sha256_k2_h20:
   case mts_sha256_k4_h10:
   case mts_sha256_k8_h7:
   case mts_sha256_k16_h5:
        mts_public_key_n32 z_n32;
   case mts_sha512_k2_h20:
   case mts_sha512_k4_h10:
   case mts_sha512_k8_h7:
   case mts_sha512_k16_h5:
        mts_public_key_n64 z_n64;
    default:
     void;     /* error condition */
  };

  struct mts_private_key_n32 {
    ots_algorithm_type ots_alg_type;
    unsigned int next_ldwm_leaf_number;  /* leaf # for next signature */
    opaque value[32];                    /* private key */
  };

  struct mts_private_key_n64 {
    ots_algorithm_type ots_alg_type;
    unsigned int next_ldwm_leaf_number;  /* leaf # for next signature */
    opaque value[64];                    /* private key */
  };

  union mts_private_key switch (mts_algorithm_type mts_alg_type) {
   case mts_sha256_k2_h20:
   case mts_sha256_k4_h10:
   case mts_sha256_k8_h7:
   case mts_sha256_k16_h5:
      mts_private_key_n32 body_n32;
   case mts_sha512_k2_h20:
   case mts_sha512_k4_h10:
   case mts_sha512_k8_h7:
   case mts_sha512_k16_h5:
      mts_private_key_n64 body_n64;
   default:
     void;     /* error condition */
  };
```

5. Rationale

The goal of this note is to describe the LDWM and MTS algorithms following the original references and present the modern security analysis of those algorithms. Other signature methods are out of scope and may be interesting follow-on work.

The signature and public key formats are designed so that they are easy to parse. Each format starts with a 32-bit enumeration value that indicates all of the details of the signature algorithm and hence defines all of the information that is needed in order to parse the format.

The enumeration values used in this note are palindromes, which have the same byte representation in either host order or network order. This fact allows an implementation to omit the conversion between byte order for those enumerations. Note however that the leaf number field used in the MTS signature and keys must be properly converted to and from network byte order; this is the only field that requires such conversion. There are 2^32 XDR enumeration values, 2^16 of which are palindromes, which is more than enough for the foreseeable future. If there is a need for more assignments, non-palindromes can be assigned.

6. History

This is the initial version of this draft.

This section is to be removed by the RFC editor upon publication.

7. IANA Considerations

The Internet Assigned Numbers Authority (IANA) is requested to create two registries: one for OTS signatures, which includes all of the LDWM signatures as defined in Section 3, and one for Merkle Tree Signatures, as defined in Section 4. Additions to these registries require that a specification be documented in an RFC or another permanent and readily available reference in sufficient detail that interoperability between independent implementations is possible. Each entry in the registry contains the following elements:

a short name, such as "MTS_SHA256_K16_H5",

a positive number, and

a reference to a specification that completely defines the signature method test cases that can be used to verify the correctness of an implementation.

Requests to add an entry to the registry MUST include the name and the reference. The number is assigned by IANA. These number assignments SHOULD use the smallest available palindromic number. Submitters SHOULD have their requests reviewed by the IRTF Crypto Forum Research Group (CFRG) at cfrg@ietf.org. Interested applicants that are unfamiliar with IANA processes should visit http://www.iana.org.

The numbers between 0xDDDDDDDD (decimal 3,722,304,989) and 0xFFFFFFFF (decimal 4,294,967,295) inclusive, will not be assigned by IANA, and are reserved for private use; no attempt will be made to prevent multiple sites from using the same value in different (and incompatible) ways [RFC2434].

The LDWM registry is as follows.

Name	Reference	Numeric Identifier
LDWM_SHA256_M20_W1	Section 3	0x01000001
LDWM_SHA256_M20_W2	Section 3	0x02000002
LDWM_SHA256_M20_W4	Section 3	0x03000003
LDWM_SHA256_M20_W8	Section 3	0x04000004
LDWM_SHA256_M32_W1	Section 3	0x05000005
LDWM_SHA256_M32_W2	Section 3	0x06000006
LDWM_SHA256_M32_W4	Section 3	0x07000007
LDWM_SHA256_M32_W8	Section 3	0x08000008
LDWM_SHA512_M64_W1	Section 3	0x09000009
LDWM_SHA512_M64_W2	Section 3	0x0a00000a
LDWM_SHA512_M64_W4	Section 3	0x0b00000b
LDWM_SHA512_M64_W8	Section 3	0x0c00000c

Table 2

The MTS registry is as follows.

Name	Reference	Numeric Identifier
MTS_SHA256_K2_H20	Section 4	0x01000001
MTS_SHA256_K4_H10	Section 4	0x02000002
MTS_SHA256_K8_H7	Section 4	0x03000003
MTS_SHA256_K16_H5	Section 4	0x04000004
MTS_SHA512_K2_H20	Section 4	0x05000005
MTS_SHA512_K4_H10	Section 4	0x06000006
MTS_SHA512_K8_H7	Section 4	0x07000007
MTS_SHA512_K16_H5	Section 4	0x08000008

Table 3

An IANA registration of a signature system does not constitute an endorsement of that system or its security.

8. Security Considerations

The security goal of a signature system is to prevent forgeries. A successful forgery occurs when an attacker who does not know the private key associated with a public key can find a message and signature that are valid with that public key (that is, the Signature Verification algorithm applied to that signature and message and public key will return "valid"). Such an attacker, in the strongest case, may have the ability to forge valid signatures for an arbitrary number of other messages.

The security of the algorithms defined in this note can be roughly described as follows. For a security level of roughly 128 bits, assuming that there are no quantum computers, use the LDWM algorithms with m=32 and MTS with n=32. For a security level of roughly 128 bits, assuming that there are quantum computers, use the LDWM algorithms with m=64 and the MTS algorithms with n=64. For the smallest possible signatures that provide a currently adequate security level, use the LDWM algorithms with m=20 and MTS algorithms with n=32. We emphasize that this is a rough estimate, and not a security proof.

LDWM signatures rely on the fact that, given an m-byte string y, it is prohibitively expensive to compute a value x such that F^i(x) = y for any i. Informally, F is said to be a "one-way" function, or a preimage-resistant function. Both LDWM and MTS signatures rely on the fact that H is collision-resistant, that is, it is prohibitively expensive for an attacker to find two byte strings a and b such that H(a) = H(b).

There are several formal security proofs for one time signatures and Merkle tree signatures in the cryptographic literature. Several of these analyze variants of those algorithms, and are not directly applicable to the original algorithms; thus caution is needed when applying these analyses. The MTS scheme has been shown to provide roughly b bits of security when used with a hash function with an output size of 2*b bits [BDM08]. (A cryptographic scheme has b bits of security when an attacker must perform O(2^b) computations to defeat it.) More precisely, that analysis shows that MTS is existentially unforgeable under an adaptive chosen message attack. However, the analysis assumes that the hash function is chosen uniformly at random from a family of hash functions, and thus is not completely applicable. Similarly, LDWM with w=1 has been shown to be existentially unforgeable under an adaptive chosen message attack, when F is a one-way function [BDM08], when F is chosen uniformly at random from a family of one-way functions; when F has c-bit inputs and outputs, it provides roughly b bits of security. LDWM signatures, as specified in this note, have been shown to be secure

based on the collision resistance of F [C:Dods05]; that analysis provides a lower bound on security (and it appears to be pessimistic, especially in the case of the m=20 signatures).

It may be desirable to adapt this specification in a way that better aligns with the security proofs. In particular, a random "salt" value could be generated along with the key, used as an additional input to F and H, and then provided as part of the public key. This change appears to make the analysis of [BDM08] applicable, and it would improve the resistance of these signature schemes against key collision attacks, that is, scenarios in which an attacker concurrently attacks many signatures made with many private keys.

8.1. Security of LDWM Checksum

To show the security of LDWM checksum, we consider the signature y of a message with a private key x and let h = H(message) and c = C(H(message)) (see Section 3.7). To attempt a forgery, an attacker may try to change the values of h and c. Let h' and c' denote the values used in the forgery attempt. If for some integer j in the range 0 to (u-1), inclusive,

a' = coef(h', j, w),

a = coef(h, j, w), and

a' > a

then the attacker can compute F^a'(x[j]) from F^a(x[j]) = y[j] by iteratively applying function F to the j^th term of the signature an additional (a' - a) times. However, as a result of the increased number of hashing iterations, the checksum value c' will decrease from its original value of c. Thus a valid signature's checksum will have, for some number k in the range u to (p-1), inclusive,

b' = coef(c', k, w),

b = coef(c, k, w), and

b' < b

Due to the one-way property of F, the attacker cannot easily compute F^b'(x[k]) from F^b(x[k]) = y[k].

8.2. Security Conjectures

LDWM and MTS signatures rely on a minimum of security conjectures. In particular, their security does not rely on the computational

difficulty of factoring composites with large prime factors (as does RSA) or the difficulty of computing the discrete logarithm in a finite field (as does DSA) or an elliptic curve group (as does ECDSA). All of these signature schemes also rely on the security of the hash function that they use, but with LDWM and MTS, the security of the hash function is sufficient.

8.3. Post-Quantum Security

A post-quantum cryptosystem is a system that is secure against quantum computers that have more than a trivial number of quantum bits. It is open to conjecture whether or not it is feasible to build such a machine.

The LDWM and Merkle signature systems are post-quantum secure if they are used with an appropriate underlying hash function, in which the size of m and n are double what they would be otherwise, in order to protect against quantum square root attacks due to Grover’s algorithm. In contrast, the signature systems in wide use (RSA, DSA, and ECDSA) are not post-quantum secure.

9. Acknowledgements

Thanks are due to Chirag Shroff for constructive feedback, and to Andreas Hulsing, Burt Kaliski, Eric Osterweil, Ahmed Kosba, and Russ Housley for valuable detailed review.

10. References

10.1. Normative References

[FIPS180] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS 180-4, March 2012.

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

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 2434, October 1998.

[RFC4506] Eisler, M., "XDR: External Data Representation Standard", STD 67, RFC 4506, May 2006.

10.2. Informative References

[BDM08] Buchmann, J., Dahmen, E., and M. Szydlo, "Hash-based Digital Signature Schemes", Technische Universitat Darmstadt Technical Report https:// www.cdc.informatik.tu-darmstadt.de/~dahmen/papers/ hashbasedcrypto.pdf, 2008.

[C:Dods05] Dods, C., Smart, N., and M. Stam, "Hash Based Digital Signature Schemes", Lecture Notes in Computer Science vol. 3796 Cryptography and Coding, 2005.

[C:Merkle87] Merkle, R., "A Digital Signature Based on a Conventional Encryption Function", Lecture Notes in Computer Science crypto87vol, 1988.

[C:Merkle89a] Merkle, R., "A Certified Digital Signature", Lecture Notes in Computer Science crypto89vol, 1990.

[C:Merkle89b] Merkle, R., "One Way Hash Functions and DES", Lecture Notes in Computer Science crypto89vol, 1990.

[Merkle79] Merkle, R., "Secrecy, Authentication, and Public Key Systems", Stanford University Information Systems Laboratory Technical Report 1979-1, 1979.

Appendix A. LDWM Parameter Options

A table illustrating various combinations of n and w with the associated values of u, v, ls, and p is provided in Table 4.

The parameters u, v, ls, and p are computed as follows:

```
u = ceil(8*n/w)
v = ceil((floor(lg((2^w - 1) * u)) + 1) / w)
ls = (number of bits in sum) - (v * w)
p = u + v
```

Here u and v represent the number of w-bit fields required to contain the hash of the message and the checksum byte strings, respectively. The "number of bits in sum" is defined according to Section 3.6. And as the value of p is the number of w-bit elements of (H(message) || C(H(message))), it is also equivalently the number of byte strings that form the private key and the number of byte strings in the signature.

Hash Length in Bytes (n)	Winternitz Parameter (w)	w-bit Elements in Hash (u)	w-bit Elements in Checksum (v)	Left Shift (ls)	Total Number of w-bit Elements (p)
20	1	160	8	8	168
20	2	80	4	8	84
20	4	40	3	4	43
20	8	20	2	0	22
32	1	256	9	7	265
32	2	128	5	6	133
32	4	64	3	4	67
32	8	32	2	0	34
48	1	384	9	7	393
48	2	192	5	6	197
48	4	96	3	4	99
48	8	48	2	0	50
64	1	512	10	6	522
64	2	256	5	6	261
64	4	128	3	4	131
64	8	64	2	0	66

Table 4

Appendix B. Example Data for Testing

As with all cryptosystems, implementations of LDWM signatures and Merkle signatures need to be tested before they are used. This section contains sample data generated from the signing and verification operations of software that implements the algorithms described in this document.

B.1. Parameters

The example contained in this section demonstrates the calculations of LDWM_SHA256_M20_W4 using a Merkle Tree Signature of degree 4 and height 2. This corresponds to the following parameter values:

m	n	w	p	ls	k	h
20	32	4	67	4	4	2

Table 5

The non-standard size of the Merkle tree (h = 2) has been selected specifically for this example to reduce the amount of data presented.

B.2. Key Generation

The LDWM algorithm does not define a required method of key generation. This is left to the implementer. The selected method, however, must satisfy the requirement that the private keys of the one-time signatures are uniformly random, independent, and unpredicable. In addition, all LDWM key pairs must be generated in advance in order to calculate the value of the Merkle public key.

For the test data presented here, a summary of the key generation method is as follows:

1. MTS Private Key - Set mts_private_key to a pseudorandomly generated n-byte value.

2. OTS Private Keys - Use the mts_private_key as a key derivation key input to some key derivation function, thereby producing n^k derived keys. Then use each derived key as an input to the same function again to further derive p elements of n-bytes each. This accomplishes the result of Algorithm 0 of Section 3.4 for each leaf of the Merkle tree.

3. OTS Public Keys - For each OTS private key, calculate the corresponding OTS public key as in Algorithm 1 of Section 3.5.

4. MTS Public Key - Each OTS public key is the value of a leaf on the Merkle tree. Calculate the MTS public key using the pseudocode algorithm of Section 4.2 or some equivalent implementation.

The above steps result in the following data values associated with the first leaf of the Merkle tree, leaf 0.

MTS Private Key
0x0f677ff1b4cbf10baec89959f051f203 3371492da02f62dd61d6fbd1cee1bd14

Table 6

Key Element Index (i)	OTS Private Key 0 Element (x[i])
0	0xbfb757383fb08d324629115a84daf00b 188d5695303c83c184e1ec7a501c431f
1	0x7ce628fb82003a2829aab708432787d0 fc735a29d671c7d790068b453dc8c913
2	0x8174929461329d15068a4645a34412bd 446d4c9e757463a7d5164efd50e05c93
3	0xf283f3480df668de4daa74bb0e4c5531 5bc00f7d008bb6311e59a5bbca910fd7
4	0xe62708eaf9c13801622563780302a068 0ba9d39c078daa5ebc3160e1d80a1ea7
5	0x1f002efad2bfb4275e376af7138129e3 3e88cf7512ec1dcdc7df8d5270bc0fd7
6	0x8ed5a703e9200658d18bc4c05dd0ca8a 356448a26f3f4fe4e0418b52bd6750a2
7	0xc74e56d61450c5387e86ddad5a8121c8 8b1bc463e64f248a1f1d91d950957726

8	0x629f18b6a2a4ea65fff4cf758b57333f e1d34af05b1cd7763696899c9869595f	
9	0x1741c31fdbb4864712f6b17fadc05d45 926c831c7a755b7d7af57ac316ba6c2a	
10	0xe59a7b81490c5d1333a9cdd48b9cb364 56821517a3a13cb7a8ed381d4d5f3545	
11	0x3ba97fe8b2967dd74c8b10f31fc5f527 a23b89c1266202a4d7c281e1f41fa020	
12	0xa262a9287cc979aaa59225d75df51b82 57b92e780d1ab14c4ac3ecdac58f1280	
13	0x9dfe0af1a3d9064338d96cb8eae88baa 6a69265538873b4c17265fa9d573bcff	
14	0xde9c5c6a5c6a274eabe90ed2a8e6148c 720196d237a839aaf5868af8da4d0829	
15	0x5de81ec17090a82cb722f616362d3808 30f04841191e44f1f81b9880164b14cd	
16	0xc0d047000604105bad657d9fa2f9ef10 1cfd9490f4668b700d738f2fa9e1d11a	
17	0xf45297ef310941e1e855f97968129bb1 73379193919f7b0fee9c037ae507c2d2	
18	0x46ef43a877f023e5e66bbcd4f06b839f 3bfb2b64de25cd67d1946b0711989129	
19	0x46e2a599861bd9e8722ad1b55b8f0139 305fcf8b6077d545d4488c4bcb652f29	
20	0xe1ad4d2d296971e4b0b7a57de305779e 82319587b58d3ef4daeb08f630bd5684	
21	0x7a07fa7aed97cb54ae420a0e6a58a153 38110f7743cab8353371f8ca710a4409	
22	0x40601f6c4b35362dd4948d5687b5cb6b 5ec8b2ec59c2f06fd50f8919ebeaae92	
23	0xa061b0ba9f493c4991be5cd3a9d15360 a9eb94f6f7adc28dddf174074f3df3c4	

24	0xcf1546a814ff16099cebf1fe0db1ace5 1c272fda9846fbb535815924b0077fa4	
25	0xcbb06f13155ce4e56c85a32661c90142 8b630a4c37ea5c7062156f07f6b3efff	
26	0x1181ee7fc03342415094e36191eb450a 11cdea9c6f6cdc34de79cee0ba5bf230	
27	0xe9f1d429b343bb897881d2a19ef363cd 1ab4117cbaad54dc292b74b8af9f5cf2	
28	0x87f34b2551ef542f579fa65535c5036f 80eb83be4c898266ffc531da2e1a9122	
29	0x9b4b467852fe33a03a872572707342fd ddeae64841225186babf353fa2a0cd09	
30	0x19d58cd240ab5c80be6ddf5f60d18159 2dca2be40118c1fdd46e0f14dffbcc7d	
31	0x5c9ad386547ba82939e49c9c74a8eccf 1cea60aa327b5d2d0a66b1ca48912d6d	
32	0xf49083e502400ffae9273c6de92a301e 7bda1537cab085e5adfa9eb746e8eca9	
33	0x4074e1812d69543ce3c1ce706f6e0b45 f5f26f4ef39b34caa709335fd71e8fc0	
34	0x1256612b0ca8398e97b247ae564b74b1 3839b3b1cf0a0dd8ba629a2c58355f84	
35	0xbab3989f00fd2c327bbfb35a218cc3ce 49d6b34cbf8b6e8919e90c4eff400ca9	
36	0x96b52a5d395a5615b73dae65586ac5c8 7f9dd3b9b3f82dbf509b5881f0643fa8	
37	0x5d05ca4c644e1c41ccdaedbd2415d4f0 9b4a1b940b51fe823dff7617b8ee8304	
38	0xd96aab95ef6248e235d91d0f23b64727 a6675adfc64efea72f6f8b4a47996c0d	
39	0xfd9c384d52d3ac27c4f4898fcc15e83a c182f97ea63f7d489283e2cc7e6ed180	

40	0xc86eaed6a9e3fbe5b262c1fa1f099f7c 35ece71d9e467fab7a371dbcf400b544	
41	0xf462b3719a2ed8778155638ff814dbf4 2b107bb5246ee3dd82abf97787e6a69e	
42	0x014670912e3eb74936ebb64168b447e4 2522b57c2540ac4b49b9ae356c01eca6	
43	0x2b411096e0ca16587830d3acd673e858 863fedc4cea046587cba0556d2bf9884	
44	0xa73917c74730582e8e1815b8a07b1896 2ac05e500e045676be3f1495fcfa18ca	
45	0xa4ab61e6962fe39a255dbf8a46d25110 0d127fab08db59512653607bda24302c	
46	0x9b910ca516413f376b9eba4b0d571b22 253c2a9646131ac9a2af5f615f7322b8	
47	0xfc1b4ce627c77ad35a21ea9ded2cce91 b3758a758224e35cf2918153a513d64c	
48	0xc1902d8e8c02d9442581d7e053a2798a a84d77a74b6e7f2cc5096d50646c890f	
49	0xb3f47e2e8e2dcdd890ea00934b9d8234 830dbc4a30ac996b144f12b3e463c77f	
50	0x8188d1ecfc6ae6118911f2b9b3a6c7a1 e5f909aa8b5c0aab8c69f1a7d436c307	
51	0xca42d985974c7b870bc76494604eff49 2676c942c6cb7c75d4938805885dd054	
52	0xbe58851ebe566057e1ee16b8c604a473 4c373af622660b2a82357ac6effb4566	
53	0xc22d493f7a5642fceba2404dbefa8f95 6323fac87fac425f6de8d23c9e8b20ca	
54	0x1a76c1ffa467906173fd0245b0cd6639 e6013ca79c4ed92426ee69ff5beeac0b	
55	0xbc6c0cb7808f379af1b7b7327436ad65 c05458f2d0a6923c333e5129c4c99671	

56	0xfbb04488c3c088dc5e63d13e6a701036 6109ca4c5f4b0a8d37780187e2e9930e
57	0xaec10811569d4d72e3a1baf71a886b75 eba6dc07ed027af0b2beffa71f9b43c8
58	0xf5529be3b7a19212e8baa970d2420bf4 123f678267f96c1c3ef26ab610cb0061
59	0x172ba1ba0b701eeafe00692d1eb90181 8ccaefaeb8f799395da81711766d1f43
60	0xfe1f8c15825208f3a21346b894b3d94e 4f3aa29cbc194a7b2c8a810c4c509042
61	0x2e81c66cc914ea1b0fa5942fe9780d54 8c0b330e3bf73f0cb0bda4bc9c9e6ff4
62	0xfc3453aec5cc19a6a4bda4bc25931604 704bf4386cd65780c6e73214c1da85ba
63	0x4e8000c587dc917888e7e3d817672c0a ef812788cc8579afa7e9b2e566309003
64	0xba667ca0e44a8601a0fde825d4d2cf1b b9cf467041e04af84c9d0cd9fd8dc784
65	0x4965db75f81c8a596680753ce70a94c6 156253bb426947de1d7662dd7e05e9a8
66	0x2c23cc3e5ca37dec279c506101a3d8d9 f1e4f99b2a33741b59f8bddba7455419

Table 7

Using the value of the OTS private key above, the corresponding public key is given below. Intermediate values of the SHA256-20 function F^(2^w - 1)(x[i]) are provided in Table 13.

OTS Public Key 0
0x2db55a72075fcfab5aedbef77bf6b371 dfb489d6e61ad2884a248345e6910618

Table 8

Following the creation of all OTS public/private key pairs, the OTS public keys in Table 14 are used to determine the MTS public key below. Intermediate values of the interior nodes of the Merkle tree are provided in Table 15.

MTS Public Key
0x6610803d9a3546fb0a7895f6a4a0cfed 3a07d45e51d096e204b018e677453235

Table 9

B.3. Signature Generation

In order to test signature generation, a text file containing the content "Hello world!\n", where '\n' represents the ASCII line feed character, was created and signed. A raw hex dump of the file contents is shown in the table below.

Hexadecimal Byte Values	ASCII Representation ('.' is substituted for non-printing characters)
0x48 0x65 0x6c 0x6c 0x6f 0x20 0x77 0x6f 0x72 0x6c 0x64 0x21 0x0a	Hello world!.

Table 10

The SHA256 hash of the text file is provided below.

SHA256 Hash of Signed File (H("Hello world!\n"))
0x0ba904eae8773b70c75333db4de2f3ac 45a8ad4ddba1b242f0b3cfc199391dd8

Table 11

This value was subsequently used in Algorithm 3 of Section 3.7 to create the one-time signature of the message. Algorithm 2 of Section 3.6 was applied to calculate a checksum of 0x1cc. The resulting signature is shown in the following table.

OTS Element Index (i)	Function Iteration Count (a = coef(H(msg) || C(H(msg)), i, w))	OTS Element (y[i] = F^a(x[i]))
0	0	0xbfb757383fb08d324629115a84daf00b188d5695
1	11	0x4af079e885ddfd3245f29778d265e868a3bfeaa4
2	10	0xfbad1928bfc57b22bcd949192452293d07d6b9ad
3	9	0xb98063e184b4cb949a51e1bb76d99d4249c0b448
4	0	0xe62708eaf9c13801622563780302a0680ba9d39c
5	4	0x39343cba3ffa6d75074ce89831b3f3436108318c
6	14	0xfe08aa73607aec5664188a9dacdc34a295588c9a
7	10	0xd3346382119552d1ceb92a78597a00c956372bf0
8	14	0xf1dd245ec587c0a7a1b754cc327b27c839a6e46a
9	8	0xa5f158adc1decaf0c1edc1a3a5d8958d726627b5
10	7	0x06d2990f62f22f0c943a418473678e3ffdbff482
11	7	0xf3390b8d6e5229ae9c5d4c3f45e10455d8241a49
12	3	0x22dd5f9d3c89180caa0f695203d8cf90f3c359be
13	11	0x67999c4043f95de5f07d82b741347a3eb6ac0c25
14	7	0xc4ffe472d48adeb37c7360da70711462013b7a4e
15	0	0x5de81ec17090a82cb722f616362d380830f04841
16	12	0x2f892c824af65cc749f912a36dfa8ade2e4c3fd1
17	7	0xb644393e8030924403b594fb5cacd8b2d28862e2
18	5	0x31b8d2908911dbbf5ba1f479a854808945d9e948
19	3	0xa9a02269d24eb8fed6fb86101cbd0d8977219fb1

	20	3	0xe4aae6e6a9fe1b0d5099513f170c111dee95714d
	21	3	0xd79c16e7f2d4dd790e28bab0d562298c864e31e9
	22	13	0xc29678f0bb4744597e04156f532646c98a0b42e8
	23	11	0x57b31d75743ff0f9bcf2db39d9b6224110b8d27b
	24	4	0x0a336d93aac081a2d849c612368b8cbb2fa9563a
	25	13	0x917be0c94770a7bb12713a4bae801fb3c1c43002
	26	14	0x91586feaadcf691b6cb07c16c8a2ed0884666e84
	27	2	0xdd4e4b720fb2517c4bc6f91ccb8725118e5770c6
	28	15	0x491f6ec665f54c4b3cffaa02ec594d31e6e26c0e
	29	3	0x4f5a082c9d9c9714701de0bf426e9f893484618c
	30	10	0x11f7017313f0c9549c5d415a8abc25243028514d
	31	12	0x6839a994fccb9cb76241d809146906a3d13f89f1
	32	4	0x71cd1d9163d7cd563936837c61d97bb1a5337cc0
	33	5	0x77c9034ffc0f9219841aa8e1edbfb62017ef9fd1
	34	10	0xad9f6034017d35c338ac35778dd6c4c1abe4472a
	35	8	0x4a1c396b22e4f5cc2428045b36d13737c4007515
	36	10	0x98cb57b779c5fd3f361cd5debc243303ae5baefd
	37	13	0x29857298f274d6bf595eadc89e5464ccf9608a6c
	38	4	0x95e35a26815a3ae9ad84a24464b174a29364da18
	39	13	0x4afeb3b95b5b333759c0acdd96ce3f26314bb22b
	40	13	0x325a37ee5e349b22b13b54b24be5145344e7b8f3
	41	11	0x4f772c93f56fd6958ce135f02847996c67e1f2ef
	42	10	0xd4f6d91c577594060be328b013c9e9b0e8a2e5d8
	43	1	0x717e1a81c325cdccacb6e9fd9e92dd3e1bb84ae8

44	11	0x1dd363724ec66c090a1228dfa1cd3d9cc806f346
45	2	0x64b4110476dd0beea78714c5ab71278818792cfa
46	4	0xe22290e740056a144af50f0b10962b5bcc18fc82
47	2	0x34fd87046a183f4732a52bb7805ce207eebdafc5
48	15	0xbd2fdc5e4e8d0ed7c48c1bad9c2f7793fc2c9303
49	0	0xb3f47e2e8e2dcdd890ea00934b9d8234830dbc4a
50	11	0xcd29719c56cdb507030e6132132179e5807e1d3b
51	3	0xf9edb9b301916217de0d746a0542316bebe9e806
52	12	0x7a3801cbfe0cafed863d81210c1ec721eede49e5
53	15	0x5caba3ec960efa210f5f3e1c22c567ca475ef3ec
54	12	0xf911b5d148e1b03fe6983c53411f76ea78772379
55	1	0x06da2baa75c6ef752bf59f3812fa042ff8181209
56	9	0x2b29f5aa2f34af51a78a5fac586004f749c6e6dc
57	9	0x55e033ababac0845cc9142e24f9ef0a641c51cbe
58	3	0xb62d207bb700071fba8a68312ca204ce4d994c33
59	9	0x551d5c00fad905bdb99c4f70ec7590a10d3ff8ca
60	1	0x0d03b1845b5f8838d735142f185f9cf8f8d2db6c
61	13	0x3b5d9e49e7ede41cd9aa5a09f72a0384fd4ff511
62	13	0xa766b0278d14a9b7d32bf0307c0737a8ecf82ab1
63	8	0xca85296f354e6e3d2a96ab497c01e5ccd4530cf1
64	1	0x7bb29db7dd8aaaf1cd11487cea0d13730edb1df3
65	12	0x547ef341b3cf3208753bb1b62d85a4e3fc2cffe0
66	12	0xb890e1a99da4b2e0a9dde42f82f92d0946327cee

Table 12

Finally, based on the fact that the message is the first to be signed by the Merkle tree (i.e. using leaf node 0), the values of the leaf and interior nodes that compose the authentication path from leaf to root are determined as described in Section 4.3. These values are marked with an asterisk ('*') in Table 14 and Table 15.

B.4. Signature Verification

The signature verification step was provided the following items:

1. OTS = (y[0] || y[1] || ... || y[p-1]) - from Table 12.

2. Authentication Path = concatenation of (k-1)*h Merkle tree node values - from Table 14 and Table 15.

3. Message Number = leaf number of Merkle tree.

4. Merkle Public Key = root of Merkle tree - from Table 9.

Using Algorithm 4 of Section 3.8 as a start, the potential OTS public key was calculated from the value of the OTS. Since the actual OTS public key was not provided to the verifier, the calculated key was checked for validity using the pseudocode algorithm of Section 4.4 and the provided values of the Authentication Path and Message Number. Since the message was valid, the calculated value of the root matched the Merkle public key. Otherwise, verification would have failed.

B.5. Intermediate Calculation Values

Key Element Index (i)	SHA256-20 Result for w = 4 (F^15(x[i]))
0	0x6eff4b0c224874ecc4e4f4500da53dbe2a030e45
1	0x58ac2c6c451c7779d67efefdb12e5c3d85475a94
2	0xb1f3e42e29c710d69268eed1bbdb7f5a500b7937
3	0x51d28e573aac2b84d659abb961c32c465e911b55
4	0xa0ed62bccac5888f5000ca6a01e5ffefd442a1c6
5	0x44da9e145666322422c1e2b5e21627e05aeb4367
6	0x04e7ff9213c2655f28364f659c35d3086d7414e1

7	0x414cdb3215408b9722a02577eeb71f9e016e4251
8	0xa3ab06b90a2b20f631175daa9454365a4f408e9e
9	0xe38acfd3c0a03faa82a0f4aeac1a7c04983fad25
10	0xd95a289094ccce8ad9ff1d5f9e38297f9bb306ff
11	0x593d148b22e33c32f18b66340bdaffceb3ad1a55
12	0x16b53fbea11dc7ab70c8336ec3c23881ae5d51bf
13	0xa639ca0cf871188cadd0020832c4f06e6ebd5f98
14	0xe3ab3e0c5ad79d6c8c2a7e9a79856d4380941fe0
15	0x8368c2933dabcde69c373867a9bf2dc78df97bea
16	0xe3609fca11545da156a7779ae565b1e3c87902c0
17	0xab029e62c7011772dc0589d79fad01aacf8d2177
18	0xa8310f1c27c1aa481192de07d4397b8c4716e25f
19	0xdbdbb14dbd9a5f03c1849af24b69b9e3f80faca2
20	0x1a17399d555dec07d3d4f6d54b2b87d2bcaa398b
21	0xf81c66cc522bfb203232e44d0003ed65d2462867
22	0x202a625b8c5f22de6ea081af6da077cf5c63202f
23	0x2e080f3591f5ff3d5de39c2698846cc107a09816
24	0xa1d9c78c22f9810e3b7db2d59ad9f5fdd259f4d4
25	0x658eeb85ebe0f4542c4d32dced2d7226929266b2
26	0x67fae1a784f919577afc091504d82d31b4ba9fc7
27	0xfc39fb43677fb2d433a6292f19c6e7320279655a
28	0x491f6ec665f54c4b3cffaa02ec594d31e6e26c0e
29	0x17cec813a5781409b11d2e4a85f62301c2fd8873
30	0xc578eb105454d900c053eb55833db607aa5757e0

31	0xaed094323290a41fd4b546919620e2f6b23916c8
32	0x192b5a87b5124dc287e06cdd4ec7c0c11f67dda6
33	0x4e9e2bdc1b0204d1ceeb68fb4159e752c40b9608
34	0xf34c57ad9ce45d67fd32dc2737e6263bcc5cc61f
35	0xf73bd27d376186310f83cc66e72060aeaccde371
36	0xeea482511acd8be783e9be42b48799653b222db4
37	0xa2e53196fec8676065b8f32b3e8498e66a4af3cf
38	0x670c98185157e1b28d38f7dafb00796b434c8316
39	0x441afbb265b93595389aaa66325de792f343f209
40	0x7b6c50d20b5edc0bc90eb4b289770514cbc8d547
41	0xfde6e862a7ba3534893a3e630e209a24be590b1e
42	0xc59611200c20b2e73dfb24c84cedf4792d6daf10
43	0x66e3527bee88373d18f91b230b53b569361f0a15
44	0xd0fd79c7116198e689275fec9b4c46f4aac73293
45	0x65f07406ad4241e7cf4174c5f284267292cdbc32
46	0x7b1b5535d45f46542e2b876245b66ea83cde3d8f
47	0x7a11620934eb0eb17e10e4a8bbd52aa4b020da0e
48	0xbd2fdc5e4e8d0ed7c48c1bad9c2f7793fc2c9303
49	0x00432602437252a0622a30676dbaaef3023328b9
50	0x09a9c4b25034466a5acd7ff681af1c27e8f97577
51	0x4b31481d52aa5e1a261064bbd87ea46479a6be23
52	0xaca2ad4aa1264618ab633bf11cbca3cc8fa43091
53	0x5caba3ec960efa210f5f3e1c22c567ca475ef3ec
54	0x353e3ffcedfd9500141921cf2aebc2e111364dad

55	0xe1c498c32169c869174ccf2f1e71e7202f45fba7
56	0x5b8519a40d4305813936c7c00a96f5b4ceb603f1
57	0x3b942ae6a6bd328d08804ade771a0775bb3ff8f8
58	0x6f3be60ee1c34372599b8d634be72e168453bf10
59	0xf700c70bac24db0aab1257940661f5b57da6e817
60	0x85ccf60624b13663a290fa808c6bbecaf89523cd
61	0xd049be55ab703c44f42167d5d9e939c830df960f
62	0xd27a178ccc3b364c7e03d2266093a0d1dfdd9d51
63	0xd73c53fdddbe196b9ab56fcc5c9a4a57ad868cd1
64	0xb59a70a7372f0c121fa71727baaf6588eccec400
65	0x9b5bf379f989f9a499799c12a3202db58b084eed
66	0xccabf40f3c1dacf114b5e5f98a73103b4c1f9b55

Table 13

MTS Leaf (Level 3) Node Number	OTS Public Key (H(x[0] || x[1] || ... || x[p-1]))	Member of Authentication Path of Message 0
0	0x2db55a72075fcfab5aedbef77bf6b371 dfb489d6e61ad2884a248345e6910618	
1	0x8c6c6a1215bfe7fda10b7754e73cd984 a64823b1ab9d5f50feda6b151c0fee6d	*
2	0xc1fb91de68b3059c273e53596108ec7c f39923757597fe86439e91ce1c25fc84	*
3	0x1b511189baee50251335695b74d67c40 5a04eddaa79158a9090cc7c3eb204cbf	*
4	0xf3bcf088ccf9d00338b6c87e8f822da6 8ec471f88d1561193b3c017d20b3c971	
5	0x40584c059e6cc72fb61f7bd1b9c28e73 c689551e6e7de6b0b9b730fab9237531	
6	0x1b1d09de1ca16ca890036e018d7e73de b39b07de80c19dcc5e55a699f021d880	
7	0x83a82632acaac5418716f4f357f5007f 719d604525dbe1831c09a2ead9400a52	
8	0xccb8b2a1d60f731b5f51910eb427e211 96090d5cd2a077f33968b425301e3fbd	
9	0x616767ebf3c1f3ec662d8c57c630c6ae b31853fd40a18c3d831f5490610c1f16	
10	0x5a4b3e157b66327c75d7f01304d188e2 cecd1b6168240c11a01775d581b01fb6	
11	0xf25744b8a1c2184ba38521801bf4727c 407b85eb5aef8884d8fbb1c12e2f6108	
12	0xaf8189f51874999162890f72e0ef25e6 f76b4ab94dc53569bdd66507f5ab0d8e	
13	0x96251e396756686645f35cd059da329f 7083838d56c9ccacebbaf8486af18844	

14	0x773d5206e40065d3553c3c2ed2500122 e3ee6fd2c91f35a57f084dc839aab1fc	
15	0xcda7fae67ce2c3ed29ce426fdcd3f2a8 eb699e47a67a52f1c94e89726ffe97fa	

Table 14

MTS Interior (Level 2) Node Number	Node Value (H(child_0 || child_1 || ... || child_k-1))	Member of Authentication Path of Message 0
0	0xb6a310deb55ed48004133ece2aebb25e d74defb77ebd8d63c79a42b5b4191b0c	
1	0x71a0c8b767ade2c97ebac069383e4dfb a1c06d5fd3f69a775711ea6470747664	*
2	0x91109fa97662dc88ae63037391ac2650 f6c664ac2448b54800a1df748953af31	*
3	0xd277fb8c89689525f90de567068d6c93 565df3588b97223276ef8e9495468996	*

Table 15

Authors' Addresses

David McGrew
Cisco Systems
13600 Dulles Technology Drive
Herndon, VA 20171
USA

Email: mcgrew@cisco.com

Michael Curcio
Cisco Systems
7025-2 Kit Creek Road
Research Triangle Park, NC 27709-4987
USA

Email: micurcio@cisco.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 5, 2015

Y. Nir
Check Point
A. Langley
Google Inc
July 4, 2014

ChaCha20 and Poly1305 for IETF protocols
draft-nir-cfrg-chacha20-poly1305-06

Abstract

This document defines the ChaCha20 stream cipher, as well as the use of the Poly1305 authenticator, both as stand-alone algorithms, and as a "combined mode", or Authenticated Encryption with Additional Data (AEAD) algorithm.

This document does not introduce any new crypto, but is meant to serve as a stable reference and an implementation guide.

Table of Contents

1. Introduction 3
1.1. Conventions Used in This Document 3
2. The Algorithms 4
2.1. The ChaCha Quarter Round 4
2.1.1. Test Vector for the ChaCha Quarter Round 4
2.2. A Quarter Round on the ChaCha State 5
2.2.1. Test Vector for the Quarter Round on the ChaCha state 5
2.3. The ChaCha20 block Function 6
2.3.1. Test Vector for the ChaCha20 Block Function 7
2.4. The ChaCha20 encryption algorithm 8
2.4.1. Example and Test Vector for the ChaCha20 Cipher . . . 9
2.5. The Poly1305 algorithm 11
2.5.1. Poly1305 Example and Test Vector 13
2.6. Generating the Poly1305 key using ChaCha20 14
2.6.1. Poly1305 Key Generation Test Vector 15
2.7. A Pseudo-Random Function for ChaCha/Poly-1305 based Crypto Suites 16
2.8. AEAD Construction 16
2.8.1. Example and Test Vector for AEAD_CHACHA20-POLY1305 . . 17
3. Implementation Advice 19
4. Security Considerations 20
5. IANA Considerations 21
6. Acknowledgements 21
7. References 21
7.1. Normative References 21
7.2. Informative References 22
Appendix A. Additional Test Vectors 23
A.1. The ChaCha20 Block Functions 23
A.2. ChaCha20 Encryption 26
A.3. Poly1305 Message Authentication Code 28
A.4. Poly1305 Key Generation Using ChaCha20 34
A.5. ChaCha20-Poly1305 AEAD Decryption 35
Authors' Addresses 38

1. Introduction

The Advanced Encryption Standard (AES - [FIPS-197]) has become the gold standard in encryption. Its efficient design, wide implementation, and hardware support allow for high performance in many areas. On most modern platforms, AES is anywhere from 4x to 10x as fast as the previous most-used cipher, 3-key Data Encryption Standard (3DES - [FIPS-46]), which makes it not only the best choice, but the only practical choice.

The problem is that if future advances in cryptanalysis reveal a weakness in AES, users will be in an unenviable position. With the only other widely supported cipher being the much slower 3DES, it is not feasible to re-configure implementations to use 3DES. [standby-cipher] describes this issue and the need for a standby cipher in greater detail.

This document defines such a standby cipher. We use ChaCha20 ([chacha]) with or without the Poly1305 ([poly1305]) authenticator. These algorithms are not just fast. They are fast even in software-only C-language implementations, allowing for much quicker deployment when compared with algorithms such as AES that are significantly accelerated by hardware implementations.

This document does not introduce these new algorithms. They have been defined in scientific papers by D. J. Bernstein, which are referenced by this document. The purpose of this document is to serve as a stable reference for IETF documents making use of these algorithms.

These algorithms have undergone rigorous analysis. Several papers discuss the security of Salsa and ChaCha ([LatinDances], [Zhenqing2012]).

1.1. Conventions Used in This Document

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].

The description of the ChaCha algorithm will at various time refer to the ChaCha state as a "vector" or as a "matrix". This follows the use of these terms in DJB's paper. The matrix notation is more visually convenient, and gives a better notion as to why some rounds are called "column rounds" while others are called "diagonal rounds". Here's a diagram of how to matrices relate to vectors (using the C language convention of zero being the index origin).

```
   0   1   2   3
   4   5   6   7
   8   9  10  11
  12  13  14  15
```

The elements in this vector or matrix are 32-bit unsigned integers.

The algorithm name is "ChaCha". "ChaCha20" is a specific instance where 20 "rounds" (or 80 quarter rounds - see Section 2.1) are used. Other variations are defined, with 8 or 12 rounds, but in this document we only describe the 20-round ChaCha, so the names "ChaCha" and "ChaCha20" will be used interchangeably.

2. The Algorithms

The subsections below describe the algorithms used and the AEAD construction.

2.1. The ChaCha Quarter Round

The basic operation of the ChaCha algorithm is the quarter round. It operates on four 32-bit unsigned integers, denoted a, b, c, and d. The operation is as follows (in C-like notation):

- a += b; d ^= a; d <<<= 16;
- c += d; b ^= c; b <<<= 12;
- a += b; d ^= a; d <<<= 8;
- c += d; b ^= c; b <<<= 7;

Where "+" denotes integer addition modulo 2^32, "^" denotes a bitwise XOR, and "<<< n" denotes an n-bit left rotation (towards the high bits).

For example, let’s see the add, XOR and roll operations from the first line with sample numbers:

- b = 0x01020304
- a = 0x11111111
- d = 0x01234567
- a = a + b = 0x11111111 + 0x01020304 = 0x12131415
- d = d ^ a = 0x01234567 ^ 0x12131415 = 0x13305172
- d = d<<<16 = 0x51721330

2.1.1. Test Vector for the ChaCha Quarter Round

For a test vector, we will use the same numbers as in the example, adding something random for c.

- a = 0x11111111

- b = 0x01020304
- c = 0x9b8d6f43
- d = 0x01234567

After running a Quarter Round on these 4 numbers, we get these:

- a = 0xea2a92f4
- b = 0xcb1cf8ce
- c = 0x4581472e
- d = 0x5881c4bb

2.2. A Quarter Round on the ChaCha State

The ChaCha state does not have 4 integer numbers, but 16. So the quarter round operation works on only 4 of them - hence the name. Each quarter round operates on 4 pre-determined numbers in the ChaCha state. We will denote by QUATERROUND(x,y,z,w) a quarter-round operation on the numbers at indexes x, y, z, and w of the ChaCha state when viewed as a vector. For example, if we apply QUARTERROUND(1,5,9,13) to a state, this means running the quarter round operation on the elements marked with an asterisk, while leaving the others alone:

```
   0  *a   2   3
   4  *b   6   7
   8  *c  10  11
  12  *d  14  15
```

Note that this run of quarter round is part of what is called a "column round".

2.2.1. Test Vector for the Quarter Round on the ChaCha state

For a test vector, we will use a ChaCha state that was generated randomly:

Sample ChaCha State

```
    879531e0  c5ecf37d  516461b1  c9a62f8a
    44c20ef3  3390af7f  d9fc690b  2a5f714c
    53372767  b00a5631  974c541a  359e9963
    5c971061  3d631689  2098d9d6  91dbd320
```

We will apply the QUARTERROUND(2,7,8,13) operation to this state. For obvious reasons, this one is part of what is called a "diagonal round":

After applying QUARTERROUND(2,7,8,13)

```
    879531e0  c5ecf37d  bdb886dc  c9a62f8a
    44c20ef3  3390af7f  d9fc690b  cfacafd2
    e46bea80  b00a5631  974c541a  359e9963
    5c971061  ccc07c79  2098d9d6  91dbd320
```

Note that only the numbers in positions 2, 7, 8, and 13 changed.

2.3. The ChaCha20 block Function

The ChaCha block function transforms a ChaCha state by running multiple quarter rounds.

The inputs to ChaCha20 are:

- A 256-bit key, treated as a concatenation of 8 32-bit little-endian integers.
- A 96-bit nonce, treated as a concatenation of 3 32-bit little-endian integers.
- A 32-bit block count parameter, treated as a 32-bit little-endian integer.

The output is 64 random-looking bytes.

The ChaCha algorithm described here uses a 256-bit key. The original algorithm also specified 128-bit keys and 8- and 12-round variants, but these are out of scope for this document. In this section we describe the ChaCha block function.

Note also that the original ChaCha had a 64-bit nonce and 64-bit block count. We have modified this here to be more consistent with recommendations in section 3.2 of [RFC5116]. This limits the use of a single (key,nonce) combination to 2^32 blocks, or 256 GB, but that is enough for most uses. In cases where a single key is used by multiple senders, it is important to make sure that they don’t use the same nonces. This can be assured by partitioning the nonce space so that the first 32 bits are unique per sender, while the other 64 bits come from a counter.

The ChaCha20 state is initialized as follows:

- The first 4 words (0-3) are constants: 0x61707865, 0x3320646e, 0x79622d32, 0x6b206574.
- The next 8 words (4-11) are taken from the 256-bit key by reading the bytes in little-endian order, in 4-byte chunks.
- Word 12 is a block counter. Since each block is 64-byte, a 32-bit word is enough for 256 Gigabytes of data.

o Words 13-15 are a nonce, which should not be repeated for the same key. The 13th word is the first 32 bits of the input nonce taken as a little-endian integer, while the 15th word is the last 32 bits.

```
 cccccccc  cccccccc  cccccccc  cccccccc
 kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
 kkkkkkkk  kkkkkkkk  kkkkkkkk  kkkkkkkk
 bbbbbbbb  nnnnnnnn  nnnnnnnn  nnnnnnnn
```

c=constant k=key b=blockcount n=nonce

ChaCha20 runs 20 rounds, alternating between "column" and "diagonal" rounds. Each round is 4 quarter-rounds, and they are run as follows. Quarter-rounds 1-4 are part of a "column" round, while 5-8 are part of a "diagonal" round:

1. QUARTERROUND (0, 4, 8,12)
2. QUARTERROUND (1, 5, 9,13)
3. QUARTERROUND (2, 6,10,14)
4. QUARTERROUND (3, 7,11,15)
5. QUARTERROUND (0, 5,10,15)
6. QUARTERROUND (1, 6,11,12)
7. QUARTERROUND (2, 7, 8,13)
8. QUARTERROUND (3, 4, 9,14)

At the end of 20 rounds, we add the original input words to the output words, and serialize the result by sequencing the words one-by-one in little-endian order.

Note: "addition" in the above paragraph is done modulo 2^32. In some machine languages this is called carryless addition on a 32-bit word.

2.3.1. Test Vector for the ChaCha20 Block Function

For a test vector, we will use the following inputs to the ChaCha20 block function:

- Key = 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. The key is a sequence of octets with no particular structure before we copy it into the ChaCha state.
- Nonce = (00:00:00:09:00:00:00:4a:00:00:00:00)
- Block Count = 1.

After setting up the ChaCha state, it looks like this:

ChaCha State with the key set up.

```
61707865  3320646e  79622d32  6b206574
03020100  07060504  0b0a0908  0f0e0d0c
13121110  17161514  1b1a1918  1f1e1d1c
00000001  09000000  4a000000  00000000
```

After running 20 rounds (10 column rounds interleaved with 10 diagonal rounds), the ChaCha state looks like this:

ChaCha State after 20 rounds

```
837778ab  e238d763  a67ae21e  5950bb2f
c4f2d0c7  fc62bb2f  8fa018fc  3f5ec7b7
335271c2  f29489f3  eabda8fc  82e46ebd
d19c12b4  b04e16de  9e83d0cb  4e3c50a2
```

Finally we add the original state to the result (simple vector or matrix addition), giving this:

ChaCha State at the end of the ChaCha20 operation

```
e4e7f110  15593bd1  1fdd0f50  c47120a3
c7f4d1c7  0368c033  9aaa2204  4e6cd4c3
466482d2  09aa9f07  05d7c214  a2028bd9
d19c12b5  b94e16de  e883d0cb  4e3c50a2
```

After we serialize the state, we get this:

```
Serialized Block:
000  10 f1 e7 e4 d1 3b 59 15 50 0f dd 1f a3 20 71 c4  .....;Y.P.... q.
016  c7 d1 f4 c7 33 c0 68 03 04 22 aa 9a c3 d4 6c 4e  ....3.h.."....lN
032  d2 82 64 46 07 9f aa 09 14 c2 d7 05 d9 8b 02 a2  ..dF............
048  b5 12 9c d1 de 16 4e b9 cb d0 83 e8 a2 50 3c 4e  ......N......P<N
```

2.4. The ChaCha20 encryption algorithm

ChaCha20 is a stream cipher designed by D. J. Bernstein. It is a refinement of the Salsa20 algorithm, and uses a 256-bit key.

ChaCha20 successively calls the ChaCha20 block function, with the same key and nonce, and with successively increasing block counter parameters. ChaCha20 then serializes the resulting state by writing the numbers in little-endian order, creating a key-stream block. Concatenating the key-stream blocks from the successive blocks forms a key stream, which is then XOR-ed with the plaintext. Alternatively, each key-stream block can be XOR-ed with a plaintext block before proceeding to create the next block, saving some memory.

There is no requirement for the plaintext to be an integral multiple of 512-bits. If there is extra keystream from the last block, it is discarded. Specific protocols MAY require that the plaintext and ciphertext have certain length. Such protocols need to specify how the plaintext is padded, and how much padding it receives.

The inputs to ChaCha20 are:

- A 256-bit key
- A 32-bit initial counter. This can be set to any number, but will usually be zero or one. It makes sense to use 1 if we use the zero block for something else, such as generating a one-time authenticator key as part of an AEAD algorithm.
- A 96-bit nonce. In some protocols, this is known as the Initialization Vector.
- An arbitrary-length plaintext

The output is an encrypted message of the same length.

Decryption is done in the same way. The ChaCha20 block function is used to expand the key into a key stream, which is XOR-ed with the ciphertext giving back the plaintext.

2.4.1. Example and Test Vector for the ChaCha20 Cipher

For a test vector, we will use the following inputs to the ChaCha20 block function:

- Key = 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.
- Nonce = (00:00:00:00:00:00:00:4a:00:00:00:00).
- Initial Counter = 1.

We use the following for the plaintext. It was chosen to be long enough to require more than one block, but not so long that it would make this example cumbersome (so, less than 3 blocks):

```
Plaintext Sunscreen:
000  4c 61 64 69 65 73 20 61 6e 64 20 47 65 6e 74 6c  Ladies and Gentl
016  65 6d 65 6e 20 6f 66 20 74 68 65 20 63 6c 61 73  emen of the clas
032  73 20 6f 66 20 27 39 39 3a 20 49 66 20 49 20 63  s of '99: If I c
048  6f 75 6c 64 20 6f 66 66 65 72 20 79 6f 75 20 6f  ould offer you o
064  6e 6c 79 20 6f 6e 65 20 74 69 70 20 66 6f 72 20  nly one tip for
080  74 68 65 20 66 75 74 75 72 65 2c 20 73 75 6e 73  the future, suns
096  63 72 65 65 6e 20 77 6f 75 6c 64 20 62 65 20 69  creen would be i
112  74 2e                                            t.
```

The following figure shows 4 ChaCha state matrices:

1. First block as it is set up.
2. Second block as it is set up. Note that these blocks are only two bits apart - only the counter in position 12 is different.
3. Third block is the first block after the ChaCha20 block operation.
4. Final block is the second block after the ChaCha20 block operation was applied.

After that, we show the keystream.

First block setup:

```
61707865  3320646e  79622d32  6b206574
03020100  07060504  0b0a0908  0f0e0d0c
13121110  17161514  1b1a1918  1f1e1d1c
00000001  00000000  4a000000  00000000
```

Second block setup:

```
61707865  3320646e  79622d32  6b206574
03020100  07060504  0b0a0908  0f0e0d0c
13121110  17161514  1b1a1918  1f1e1d1c
00000002  00000000  4a000000  00000000
```

First block after block operation:

```
f3514f22  e1d91b40  6f27de2f  ed1d63b8
821f138c  e2062c3d  ecca4f7e  78cff39e
a30a3b8a  920a6072  cd7479b5  34932bed
40ba4c79  cd343ec6  4c2c21ea  b7417df0
```

Second block after block operation:

```
9f74a669  410f633f  28feca22  7ec44dec
6d34d426  738cb970  3ac5e9f3  45590cc4
da6e8b39  892c831a  cdea67c1  2b7e1d90
037463f3  a11a2073  e8bcfb88  edc49139
```

Keystream:

```
22:4f:51:f3:40:1b:d9:e1:2f:de:27:6f:b8:63:1d:ed:8c:13:1f:82:3d:2c:06
e2:7e:4f:ca:ec:9e:f3:cf:78:8a:3b:0a:a3:72:60:0a:92:b5:79:74:cd:ed:2b
93:34:79:4c:ba:40:c6:3e:34:cd:ea:21:2c:4c:f0:7d:41:b7:69:a6:74:9f:3f
63:0f:41:22:ca:fe:28:ec:4d:c4:7e:26:d4:34:6d:70:b9:8c:73:f3:e9:c5:3a
c4:0c:59:45:39:8b:6e:da:1a:83:2c:89:c1:67:ea:cd:90:1d:7e:2b:f3:63
```

Finally, we XOR the Keystream with the plaintext, yielding the Ciphertext:

```
  Ciphertext Sunscreen:
  000  6e 2e 35 9a 25 68 f9 80 41 ba 07 28 dd 0d 69 81  n.5.%h..A..(..i.
  016  e9 7e 7a ec 1d 43 60 c2 0a 27 af cc fd 9f ae 0b  .~z..C`..'......
  032  f9 1b 65 c5 52 47 33 ab 8f 59 3d ab cd 62 b3 57  ..e.RG3..Y=..b.W
  048  16 39 d6 24 e6 51 52 ab 8f 53 0c 35 9f 08 61 d8  .9.$.QR..S.5..a.
  064  07 ca 0d bf 50 0d 6a 61 56 a3 8e 08 8a 22 b6 5e  ....P.jaV...."."^
  080  52 bc 51 4d 16 cc f8 06 81 8c e9 1a b7 79 37 36  R.QM.........y76
  096  5a f9 0b bf 74 a3 5b e6 b4 0b 8e ed f2 78 5e 42  Z...t.[......x^B
  112  87 4d                                           .M
```

2.5. The Poly1305 algorithm

Poly1305 is a one-time authenticator designed by D. J. Bernstein. Poly1305 takes a 32-byte one-time key and a message and produces a 16-byte tag.

The original article ([poly1305]) is entitled "The Poly1305-AES message-authentication code", and the MAC function there requires a 128-bit AES key, a 128-bit "additional key", and a 128-bit (non-secret) nonce. AES is used there for encrypting the nonce, so as to get a unique (and secret) 128-bit string, but as the paper states, "There is nothing special about AES here. One can replace AES with an arbitrary keyed function from an arbitrary set of nonces to 16-byte strings."

Regardless of how the key is generated, the key is partitioned into two parts, called "r" and "s". The pair (r,s) should be unique, and MUST be unpredictable for each invocation (that is why it was originally obtained by encrypting a nonce), while "r" MAY be constant, but needs to be modified as follows before being used: ("r" is treated as a 16-octet little-endian number):

- r[3], r[7], r[11], and r[15] are required to have their top four bits clear (be smaller than 16)
- r[4], r[8], and r[12] are required to have their bottom two bits clear (be divisible by 4)

The following sample code clamps "r" to be appropriate:

```
/*
Adapted from poly1305aes_test_clamp.c version 20050207
D. J. Bernstein
Public domain.
*/

#include "poly1305aes_test.h"

void poly1305aes_test_clamp(unsigned char r[16])
{
  r[3] &= 15;
  r[7] &= 15;
  r[11] &= 15;
  r[15] &= 15;
  r[4] &= 252;
  r[8] &= 252;
  r[12] &= 252;
}
```

The "s" should be unpredictable, but it is perfectly acceptable to generate both "r" and "s" uniquely each time. Because each of them is 128-bit, pseudo-randomly generating them (see Section 2.6) is also acceptable.

The inputs to Poly1305 are:

- A 256-bit one-time key
- An arbitrary length message

The output is a 128-bit tag.

First, the "r" value should be clamped.

Next, set the constant prime "P" be 2^130-5: 3ffffffffffffffffffffffffffffffb. Also set a variable "accumulator" to zero.

Next, divide the message into 16-byte blocks. The last one might be shorter:

- Read the block as a little-endian number.
- Add one bit beyond the number of octets. For a 16-byte block this is equivalent to adding 2^128 to the number. For the shorter block it can be 2^120, 2^112, or any power of two that is evenly divisible by 8, all the way down to 2^8.
- If the block is not 17 bytes long (the last block), pad it with zeros. This is meaningless if you’re treating it them as numbers.
- Add this number to the accumulator.

- Multiply by "r"
- Set the accumulator to the result modulo p. To summarize: Acc = ((Acc+block)*r) % p.

Finally, the value of the secret key "s" is added to the accumulator, and the 128 least significant bits are serialized in little-endian order to form the tag.

2.5.1. Poly1305 Example and Test Vector

For our example, we will dispense with generating the one-time key using AES, and assume that we got the following keying material:

- Key Material: 85:d6:be:78:57:55:6d:33:7f:44:52:fe:42:d5:06:a8:01:03:80:8a:fb:0d:b2:fd:4a:bf:f6:af:41:49:f5:1b
- s as an octet string: 01:03:80:8a:fb:0d:b2:fd:4a:bf:f6:af:41:49:f5:1b
- s as a 128-bit number: 1bf54941aff6bf4afdb20dfb8a800301
- r before clamping: 85:d6:be:78:57:55:6d:33:7f:44:52:fe:42:d5:06:a8
- Clamped r as a number: 806d5400e52447c036d555408bed685.

For our message, we'll use a short text:

```
Message to be Authenticated:
000  43 72 79 70 74 6f 67 72 61 70 68 69 63 20 46 6f  Cryptographic Fo
016  72 75 6d 20 52 65 73 65 61 72 63 68 20 47 72 6f  rum Research Gro
032  75 70                                            up
```

Since Poly1305 works in 16-byte chunks, the 34-byte message divides into 3 blocks. In the following calculation, "Acc" denotes the accumulator and "Block" the current block:

Block #1

```
Acc = 00
Block = 6f4620636968706172676f7470797243
Block with 0x01 byte = 016f4620636968706172676f7470797243
Acc + block = 016f4620636968706172676f7470797243
(Acc+Block) * r =
     b83fe991ca66800489155dcd69e8426ba2779453994ac90ed284034da565ecf
Acc = ((Acc+Block)*r) % P = 2c88c77849d64ae9147ddeb88e69c83fc
```

Block #2

Acc = 2c88c77849d64ae9147ddeb88e69c83fc
Block = 6f7247206863726165736552206d7572
Block with 0x01 byte = 016f7247206863726165736552206d7572
Acc + block = 437febea505c820f2ad5150db0709f96e
(Acc+Block) * r =
21dcc992d0c659ba4036f65bb7f88562ae59b32c2b3b8f7efc8b00f78e548a26
Acc = ((Acc+Block)*r) % P = 2d8adaf23b0337fa7cccfb4ea344b30de

Last Block

Acc = 2d8adaf23b0337fa7cccfb4ea344b30de
Block = 7075
Block with 0x01 byte = 017075
Acc + block = 2d8adaf23b0337fa7cccfb4ea344ca153
(Acc + Block) * r =
16d8e08a0f3fe1de4fe4a15486aca7a270a29f1e6c849221e4a6798b8e45321f
((Acc + Block) * r) % P = 28d31b7caff946c77c8844335369d03a7

Adding s we get this number, and serialize if to get the tag:

Acc + s = 2a927010caf8b2bc2c6365130c11d06a8

Tag: a8:06:1d:c1:30:51:36:c6:c2:2b:8b:af:0c:01:27:a9

2.6. Generating the Poly1305 key using ChaCha20

As said in Section 2.5, it is acceptable to generate the one-time Poly1305 pseudo-randomly. This section proposes such a method.

To generate such a key pair (r,s), we will use the ChaCha20 block function described in Section 2.3. This assumes that we have a 256-bit session key for the MAC function, such as SK_ai and SK_ar in IKEv2 ([RFC5996]), the integrity key in ESP and AH, or the client_write_MAC_key and server_write_MAC_key in TLS. Any document that specifies the use of Poly1305 as a MAC algorithm for some protocol must specify that 256 bits are allocated for the integrity key.

The method is to call the block function with the following parameters:

- The 256-bit session integrity key is used as the ChaCha20 key.
- The block counter is set to zero.
- The protocol will specify a 96-bit or 64-bit nonce. This MUST be unique per invocation with the same key, so it MUST NOT be randomly generated. A counter is a good way to implement this, but other methods, such as an LFSR are also acceptable. ChaCha20

as specified here requires a 96-bit nonce. So if the provided nonce is only 64-bit, then the first 32 bits of the nonce will be set to a constant number. This will usually be zero, but for protocols with multiple senders it may be different for each sender, but should be the same for all invocations of the function with the same key by a particular sender.

After running the block function, we have a 512-bit state. We take the first 256 bits or the serialized state, and use those as the one-time Poly1305 key: The first 128 bits are clamped, and form "r", while the next 128 bits become "s". The other 256 bits are discarded.

Note that while many protocols have provisions for a nonce for encryption algorithms (often called Initialization Vectors, or IVs), they usually don't have such a provision for the MAC function. In that case the per-invocation nonce will have to come from somewhere else, such as a message counter.

2.6.1. Poly1305 Key Generation Test Vector

For this example, we'll set:

Key:
```
000  80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f  ................
016  90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f  ................
```

Nonce:
```
000  00 00 00 00 00 01 02 03 04 05 06 07              ............
```

The ChaCha state set up with key, nonce, and block counter zero:
```
      61707865  3320646e  79622d32  6b206574
      83828180  87868584  8b8a8988  8f8e8d8c
      93929190  97969594  9b9a9998  9f9e9d9c
      00000000  00000000  03020100  07060504
```

The ChaCha state after 20 rounds:
```
      8ba0d58a  cc815f90  27405081  7194b24a
      37b633a8  a50dfde3  e2b8db08  46a6d1fd
      7da03782  9183a233  148ad271  b46773d1
      3cc1875a  8607def1  ca5c3086  7085eb87
```

Output bytes:
```
000  8a d5 a0 8b 90 5f 81 cc 81 50 40 27 4a b2 94 71  ....._...P@'J..q
```

```
016  a8 33 b6 37 e3 fd 0d a5 08 db b8 e2 fd d1 a6 46  .3.7...........F
```

And that output is also the 32-byte one-time key used for Poly1305.

2.7. A Pseudo-Random Function for ChaCha/Poly-1305 based Crypto Suites

Some protocols such as IKEv2([RFC5996]) require a Pseudo-Random Function (PRF), mostly for key derivation. In the IKEv2 definition, a PRF is a function that accepts a variable-length key and a variable-length input, and returns a fixed-length output. This section does not specify such a function.

Poly-1305 is an obvious choice, because MAC functions are often used as PRFs. However, Poly-1305 prohibits using the same key twice, whereas the PRF in IKEv2 is used multiple times with the same key. Adding a nonce or a counter to Poly-1305 can solve this issue, much as we do when using this function as a MAC, but that would require changing the interface for the PRF function.

Chacha20 could be used as a key-derivation function, by generating an arbitrarily long keystream. However, that is not what protocols such as IKEv2 require.

For this reason, this document does not specify a PRF, and recommends that crypto suites use some other PRF such as PRF_HMAC_SHA2_256 (section 2.1.2 of [RFC4868])

2.8. AEAD Construction

AEAD_CHACHA20-POLY1305 is an authenticated encryption with additional data algorithm. The inputs to AEAD_CHACHA20-POLY1305 are:

- A 256-bit key
- A 96-bit nonce - different for each invocation with the same key.
- An arbitrary length plaintext
- Arbitrary length additional authenticated data (AAD)

The ChaCha20 and Poly1305 primitives are combined into an AEAD that takes a 256-bit key and 64-bit IV as follows:

- First the 96-bit nonce is constructed by prepending a 32-bit constant value to the IV. This could be set to zero, or could be derived from keying material, or could be assigned to a sender. It is up to the specific protocol to define the source for that 32-bit value.
- Next, a Poly1305 one-time key is generated from the 256-bit key and nonce using the procedure described in Section 2.6.
- The ChaCha20 encryption function is called to encrypt the plaintext, using the same key and nonce, and with the initial counter set to 1.

- The Poly1305 function is called with the Poly1305 key calculated above, and a message constructed as a concatenation of the following:
  - The AAD
  - padding1 - the padding is up to 15 zero bytes, and it brings the total length so far to an integral multiple of 16. If the length of the AAD was already an integral multiple of 16 bytes, this field is zero-length.
  - The ciphertext
  - padding2 - the padding is up to 15 zero bytes, and it brings the total length so far to an integral multiple of 16. If the length of the ciphertext was already an integral multiple of 16 bytes, this field is zero-length.
  - The length of the additional data in octets (as a 64-bit little-endian integer).
  - The length of the ciphertext in octets (as a 64-bit little-endian integer).

Decryption is pretty much the same thing.

The output from the AEAD is twofold:
- A ciphertext of the same length as the plaintext.
- A 128-bit tag, which is the output of the Poly1305 function.

A few notes about this design:
1. The amount of encrypted data possible in a single invocation is 2^32-1 blocks of 64 bytes each, because of the size of the block counter field in the ChaCha20 block function. This gives a total of 247,877,906,880 bytes, or nearly 256 GB. This should be enough for traffic protocols such as IPsec and TLS, but may be too small for file and/or disk encryption. For such uses, we can return to the original design, reduce the nonce to 64 bits, and use the integer at position 13 as the top 32 bits of a 64-bit block counter, increasing the total message size to over a million petabytes (1,180,591,620,717,411,303,360 bytes to be exact).
2. Despite the previous item, the ciphertext length field in the construction of the buffer on which Poly1305 runs limits the ciphertext (and hence, the plaintext) size to 2^64 bytes, or sixteen thousand petabytes (18,446,744,073,709,551,616 bytes to be exact).

2.8.1. Example and Test Vector for AEAD_CHACHA20-POLY1305

For a test vector, we will use the following inputs to the AEAD_CHACHA20-POLY1305 function:

```
  Plaintext:
  000  4c 61 64 69 65 73 20 61 6e 64 20 47 65 6e 74 6c  Ladies and Gentl
  016  65 6d 65 6e 20 6f 66 20 74 68 65 20 63 6c 61 73  emen of the clas
  032  73 20 6f 66 20 27 39 39 3a 20 49 66 20 49 20 63  s of '99: If I c
  048  6f 75 6c 64 20 6f 66 66 65 72 20 79 6f 75 20 6f  ould offer you o
  064  6e 6c 79 20 6f 6e 65 20 74 69 70 20 66 6f 72 20  nly one tip for
  080  74 68 65 20 66 75 74 75 72 65 2c 20 73 75 6e 73  the future, suns
  096  63 72 65 65 6e 20 77 6f 75 6c 64 20 62 65 20 69  creen would be i
  112  74 2e                                            t.


   AAD:
   000  50 51 52 53 c0 c1 c2 c3 c4 c5 c6 c7              PQRS........


  Key:
  000  80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f  ................
  016  90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f  ................


   IV:
   000  40 41 42 43 44 45 46 47                        @ABCDEFG


   32-bit fixed-common part:
   000  07 00 00 00                                     ....


   Set up for generating poly1305 one-time key (sender id=7):
       61707865  3320646e  79622d32  6b206574
       83828180  87868584  8b8a8988  8f8e8d8c
       93929190  97969594  9b9a9998  9f9e9d9c
       00000000  00000007  43424140  47464544


   After generating Poly1305 one-time key:
       252bac7b  af47b42d  557ab609  8455e9a4
       73d6e10a  ebd97510  7875932a  ff53d53e
       decc7ea2  b44ddbad  e49c17d1  d8430bc9
       8c94b7bc  8b7d4b4b  3927f67d  1669a432


  Poly1305 Key:
  000  7b ac 2b 25 2d b4 47 af 09 b6 7a 55 a4 e9 55 84  {.+%-.G...zU..U.
  016  0a e1 d6 73 10 75 d9 eb 2a 93 75 78 3e d5 53 ff  ...s.u..*.ux>.S.

  Poly1305 r =  455e9a4057ab6080f47b42c052bac7b
  Poly1305 s = ff53d53e7875932aebd9751073d6e10a
```

Keystream bytes:
9f:7b:e9:5d:01:fd:40:ba:15:e2:8f:fb:36:81:0a:ae:
c1:c0:88:3f:09:01:6e:de:dd:8a:d0:87:55:82:03:a5:
4e:9e:cb:38:ac:8e:5e:2b:b8:da:b2:0f:fa:db:52:e8:
75:04:b2:6e:be:69:6d:4f:60:a4:85:cf:11:b8:1b:59:
fc:b1:c4:5f:42:19:ee:ac:ec:6a:de:c3:4e:66:69:78:
8e:db:41:c4:9c:a3:01:e1:27:e0:ac:ab:3b:44:b9:cf:
5c:86:bb:95:e0:6b:0d:f2:90:1a:b6:45:e4:ab:e6:22:
15:38

Ciphertext:

```
000  d3 1a 8d 34 64 8e 60 db 7b 86 af bc 53 ef 7e c2  ...4d.‘.{...S.˜.
016  a4 ad ed 51 29 6e 08 fe a9 e2 b5 a7 36 ee 62 d6  ...Q)n......6.b.
032  3d be a4 5e 8c a9 67 12 82 fa fb 69 da 92 72 8b  =..^..g....i..r.
048  1a 71 de 0a 9e 06 0b 29 05 d6 a5 b6 7e cd 3b 36  .q.....).....˜.;6
064  92 dd bd 7f 2d 77 8b 8c 98 03 ae e3 28 09 1b 58  ....-w......(..X
080  fa b3 24 e4 fa d6 75 94 55 85 80 8b 48 31 d7 bc  ..$...u.U...H1..
096  3f f4 de f0 8e 4b 7a 9d e5 76 d2 65 86 ce c6 4b  ?....Kz..v.e...K
112  61 16                                           a.
```

AEAD Construction for Poly1305:

```
000  50 51 52 53 c0 c1 c2 c3 c4 c5 c6 c7 00 00 00 00  PQRS............
016  d3 1a 8d 34 64 8e 60 db 7b 86 af bc 53 ef 7e c2  ...4d.‘.{...S.˜.
032  a4 ad ed 51 29 6e 08 fe a9 e2 b5 a7 36 ee 62 d6  ...Q)n......6.b.
048  3d be a4 5e 8c a9 67 12 82 fa fb 69 da 92 72 8b  =..^..g....i..r.
064  1a 71 de 0a 9e 06 0b 29 05 d6 a5 b6 7e cd 3b 36  .q.....).....˜.;6
080  92 dd bd 7f 2d 77 8b 8c 98 03 ae e3 28 09 1b 58  ....-w......(..X
096  fa b3 24 e4 fa d6 75 94 55 85 80 8b 48 31 d7 bc  ..$...u.U...H1..
112  3f f4 de f0 8e 4b 7a 9d e5 76 d2 65 86 ce c6 4b  ?....Kz..v.e...K
128  61 16 00 00 00 00 00 00 00 00 00 00 00 00 00 00  a...............
144  0c 00 00 00 00 00 00 00 72 00 00 00 00 00 00 00  ........r.......
```

Note the 4 zero bytes in line 000 and the 14 zero bytes in line 128

Tag:
1a:e1:0b:59:4f:09:e2:6a:7e:90:2e:cb:d0:60:06:91

3. Implementation Advice

Each block of ChaCha20 involves 16 move operations and one increment operation for loading the state, 80 each of XOR, addition and Roll operations for the rounds, 16 more add operations and 16 XOR operations for protecting the plaintext. Section 2.3 describes the ChaCha block function as "adding the original input words". This implies that before starting the rounds on the ChaCha state, we copy

it aside, only to add it in later. This is correct, but we can save a few operations if we instead copy the state and do the work on the copy. This way, for the next block you don't need to recreate the state, but only to increment the block counter. This saves approximately 5.5% of the cycles.

It is not recommended to use a generic big number library such as the one in OpenSSL for the arithmetic operations in Poly1305. Such libraries use dynamic allocation to be able to handle any-sized integer, but that flexibility comes at the expense of performance as well as side-channel security. More efficient implementations that run in constant time are available, one of them in DJB's own library, NaCl ([NaCl]). A constant-time but not optimal approach would be to naively implement the arithmetic operations for a 288-bit integers, because even a naive implementation will not exceed 2^288 in the multiplication of (acc+block) and r. An efficient constant-time implementation can be found in the public domain library poly1305-donna ([poly1305_donna]).

4. Security Considerations

The ChaCha20 cipher is designed to provide 256-bit security.

The Poly1305 authenticator is designed to ensure that forged messages are rejected with a probability of 1-(n/(2^102)) for a 16n-byte message, even after sending 2^64 legitimate messages, so it is SUF-CMA in the terminology of [AE].

Proving the security of either of these is beyond the scope of this document. Such proofs are available in the referenced academic papers.

The most important security consideration in implementing this draft is the uniqueness of the nonce used in ChaCha20. Counters and LFSRs are both acceptable ways of generating unique nonces, as is encrypting a counter using a 64-bit cipher such as DES. Note that it is not acceptable to use a truncation of a counter encrypted with a 128-bit or 256-bit cipher, because such a truncation may repeat after a short time.

The Poly1305 key MUST be unpredictable to an attacker. Randomly generating the key would fulfill this requirement, except that Poly1305 is often used in communications protocols, so the receiver should know the key. Pseudo-random number generation such as by encrypting a counter is acceptable. Using ChaCha with a secret key and a nonce is also acceptable.

The algorithms presented here were designed to be easy to implement in constant time to avoid side-channel vulnerabilities. The operations used in ChaCha20 are all additions, XORs, and fixed rotations. All of these can and should be implemented in constant time. Access to offsets into the ChaCha state and the number of operations do not depend on any property of the key, eliminating the chance of information about the key leaking through the timing of cache misses.

For Poly1305, the operations are addition, multiplication and modulus, all on >128-bit numbers. This can be done in constant time, but a naive implementation (such as using some generic big number library) will not be constant time. For example, if the multiplication is performed as a separate operation from the modulus, the result will some times be under 2^256 and some times be above 2^256. Implementers should be careful about timing side-channels for Poly1305 by using the appropriate implementation of these operations.

5. IANA Considerations

There are no IANA considerations for this document.

6. Acknowledgements

ChaCha20 and Poly1305 were invented by Daniel J. Bernstein. The AEAD construction and the method of creating the one-time poly1305 key were invented by Adam Langley.

Thanks to Robert Ransom, Watson Ladd, Stefan Buhler, and kenny patterson for their helpful comments and explanations. Thanks to Niels Moeller for suggesting the more efficient AEAD construction in this document. Special thanks to Ilari Liusvaara for providing extra test vectors, helpful comments, and for being the first to attempt an implementation from this draft.

7. References

7.1. Normative References

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

[chacha] Bernstein, D., "ChaCha, a variant of Salsa20", Jan 2008.

[poly1305]

Bernstein, D., "The Poly1305-AES message-authentication code", Mar 2005.

7.2. Informative References

[AE] Bellare, M. and C. Namprempre, "Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm", <http://cseweb.ucsd.edu/~mihir/papers/oem.html>.

[FIPS-197] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", FIPS PUB 197, November 2001.

[FIPS-46] National Institute of Standards and Technology, "Data Encryption Standard", FIPS PUB 46-2, December 1993, <http://www.itl.nist.gov/fipspubs/fip46-2.htm>.

[LatinDances] Aumasson, J., Fischer, S., Khazaei, S., Meier, W., and C. Rechberger, "New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba", Dec 2007.

[NaCl] Bernstein, D., Lange, T., and P. Schwabe, "NaCl: Networking and Cryptography library", <http://nacl.cace-project.eu/index.html>.

[RFC4868] Kelly, S. and S. Frankel, "Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec", RFC 4868, May 2007.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, January 2008.

[RFC5996] Kaufman, C., Hoffman, P., Nir, Y., and P. Eronen, "Internet Key Exchange Protocol Version 2 (IKEv2)", RFC 5996, September 2010.

[Zhenqing2012] Zhenqing, S., Bin, Z., Dengguo, F., and W. Wenling, "Improved key recovery attacks on reduced-round salsa20 and chacha", 2012.

[poly1305_donna] Floodyberry, A., "Poly1305-donna", <https://github.com/floodyberry/poly1305-donna>.

[standby-cipher] McGrew, D., Grieco, A., and Y. Sheffer, "Selection of

Future Cryptographic Standards",
draft-mcgrew-standby-cipher (work in progress).

Appendix A. Additional Test Vectors

The sub-sections of this appendix contain more test vectors for the algorithms in the sub-sections of Section 2.

A.1. The ChaCha20 Block Functions

Test Vector #1:
==============

Key:
```
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Nonce:
```
000  00 00 00 00 00 00 00 00 00 00 00 00              ............
```

Block Counter = 0

ChaCha State at the end
```
    ade0b876  903df1a0  e56a5d40  28bd8653
    b819d2bd  1aed8da0  ccef36a8  c70d778b
    7c5941da  8d485751  3fe02477  374ad8b8
    f4b8436a  1ca11815  69b687c3  8665eeb2
```

Keystream:
```
000  76 b8 e0 ad a0 f1 3d 90 40 5d 6a e5 53 86 bd 28  v.....=.@]j.S..(
016  bd d2 19 b8 a0 8d ed 1a a8 36 ef cc 8b 77 0d c7  .........6...w..
032  da 41 59 7c 51 57 48 8d 77 24 e0 3f b8 d8 4a 37  .AY|QWH.w$.?..J7
048  6a 43 b8 f4 15 18 a1 1c c3 87 b6 69 b2 ee 65 86  jC.........i..e.
```

Test Vector #2:
==============

Key:
```
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Nonce:
```
000  00 00 00 00 00 00 00 00 00 00 00 00              ............
```

Block Counter = 1

ChaCha State at the end
```
    bee7079f  7a385155  7c97ba98  0d082d73
    a0290fcb  6965e348  3e53c612  ed7aee32
    7621b729  434ee69c  b03371d5  d539d874
    281fed31  45fb0a51  1f0ae1ac  6f4d794b
```

Keystream:
```
000  9f 07 e7 be 55 51 38 7a 98 ba 97 7c 73 2d 08 0d  ....UQ8z...|s-..
016  cb 0f 29 a0 48 e3 65 69 12 c6 53 3e 32 ee 7a ed  ..).H.ei..S>2.z.
032  29 b7 21 76 9c e6 4e 43 d5 71 33 b0 74 d8 39 d5  ).!v..NC.q3.t.9.
048  31 ed 1f 28 51 0a fb 45 ac e1 0a 1f 4b 79 4d 6f  1..(Q..E....KyMo
```

Test Vector #3:
==============

Key:
```
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01  ................
```

Nonce:
```
000  00 00 00 00 00 00 00 00 00 00 00 00              ............
```

Block Counter = 1

ChaCha State at the end
```
    2452eb3a  9249f8ec  8d829d9b  ddd4ceb1
    e8252083  60818b01  f38422b8  5aaa49c9
    bb00ca8e  da3ba7b4  c4b592d1  fdf2732f
    4436274e  2561b3c8  ebdd4aa6  a0136c00
```

Keystream:
```
000  3a eb 52 24 ec f8 49 92 9b 9d 82 8d b1 ce d4 dd  :.R$..I.........
016  83 20 25 e8 01 8b 81 60 b8 22 84 f3 c9 49 aa 5a  . %....‘."...I.Z
032  8e ca 00 bb b4 a7 3b da d1 92 b5 c4 2f 73 f2 fd  ......;...../s..
048  4e 27 36 44 c8 b3 61 25 a6 4a dd eb 00 6c 13 a0  N’6D..a%.J...l..
```

Test Vector #4:
==============

Key:
```
000  00 ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Nonce:
```
000  00 00 00 00 00 00 00 00 00 00 00 00              ............
```

Block Counter = 2

ChaCha State at the end
```
    fb4dd572  4bc42ef1  df922636  327f1394
    a78dea8f  5e269039  a1bebbc1  caf09aae
    a25ab213  48a6b46c  1b9d9bcb  092c5be6
    546ca624  1bec45d5  87f47473  96f0992e
```

Keystream:
```
000  72 d5 4d fb f1 2e c4 4b 36 26 92 df 94 13 7f 32  r.M....K6&.....2
016  8f ea 8d a7 39 90 26 5e c1 bb be a1 ae 9a f0 ca  ....9.&^........
032  13 b2 5a a2 6c b4 a6 48 cb 9b 9d 1b e6 5b 2c 09  ..Z.l..H.....[,.
048  24 a6 6c 54 d5 45 ec 1b 73 74 f4 87 2e 99 f0 96  $.lT.E..st......
```

Test Vector #5:
==============

Key:
```
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Nonce:
```
000  00 00 00 00 00 00 00 00 00 00 00 02              ............
```

Block Counter = 0

ChaCha State at the end
```
    374dc6c2  3736d58c  b904e24a  cd3f93ef
    88228b1a  96a4dfb3  5b76ab72  c727ee54
    0e0e978a  f3145c95  1b748ea8  f786c297
    99c28f5f  628314e8  398a19fa  6ded1b53
```

Keystream:
```
000  c2 c6 4d 37 8c d5 36 37 4a e2 04 b9 ef 93 3f cd  ..M7..67J.....?.
016  1a 8b 22 88 b3 df a4 96 72 ab 76 5b 54 ee 27 c7  .."".....r.v[T.'.
032  8a 97 0e 0e 95 5c 14 f3 a8 8e 74 1b 97 c2 86 f7  .....\....t.....
048  5f 8f c2 99 e8 14 83 62 fa 19 8a 39 53 1b ed 6d  _.......b...9S..m
```

A.2. ChaCha20 Encryption

```
  Test Vector #1:
  ==============

  Key:
  000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

  Nonce:
  000  00 00 00 00 00 00 00 00 00 00 00 00              ............

  Initial Block Counter = 0

  Plaintext:
  000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  032  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  048  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

  Ciphertext:
  000  76 b8 e0 ad a0 f1 3d 90 40 5d 6a e5 53 86 bd 28  v.....=.@]j.S..(
  016  bd d2 19 b8 a0 8d ed 1a a8 36 ef cc 8b 77 0d c7  .........6...w..
  032  da 41 59 7c 51 57 48 8d 77 24 e0 3f b8 d8 4a 37  .AY|QWH.w$.?..J7
  048  6a 43 b8 f4 15 18 a1 1c c3 87 b6 69 b2 ee 65 86  jC.........i..e.


  Test Vector #2:
  ==============

  Key:
  000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01  ................

  Nonce:
  000  00 00 00 00 00 00 00 00 00 00 00 02              ............

  Initial Block Counter = 1

  Plaintext:
  000  41 6e 79 20 73 75 62 6d 69 73 73 69 6f 6e 20 74  Any submission t
  016  6f 20 74 68 65 20 49 45 54 46 20 69 6e 74 65 6e  o the IETF inten
  032  64 65 64 20 62 79 20 74 68 65 20 43 6f 6e 74 72  ded by the Contr
  048  69 62 75 74 6f 72 20 66 6f 72 20 70 75 62 6c 69  ibutor for publi
  064  63 61 74 69 6f 6e 20 61 73 20 61 6c 6c 20 6f 72  cation as all or
  080  20 70 61 72 74 20 6f 66 20 61 6e 20 49 45 54 46   part of an IETF
  096  20 49 6e 74 65 72 6e 65 74 2d 44 72 61 66 74 20   Internet-Draft
```

```
112  6f 72 20 52 46 43 20 61 6e 64 20 61 6e 79 20 73  or RFC and any s
128  74 61 74 65 6d 65 6e 74 20 6d 61 64 65 20 77 69  tatement made wi
144  74 68 69 6e 20 74 68 65 20 63 6f 6e 74 65 78 74  thin the context
160  20 6f 66 20 61 6e 20 49 45 54 46 20 61 63 74 69   of an IETF acti
176  76 69 74 79 20 69 73 20 63 6f 6e 73 69 64 65 72  vity is consider
192  65 64 20 61 6e 20 22 49 45 54 46 20 43 6f 6e 74  ed an "IETF Cont
208  72 69 62 75 74 69 6f 6e 22 2e 20 53 75 63 68 20  ribution". Such
224  73 74 61 74 65 6d 65 6e 74 73 20 69 6e 63 6c 75  statements inclu
240  64 65 20 6f 72 61 6c 20 73 74 61 74 65 6d 65 6e  de oral statemen
256  74 73 20 69 6e 20 49 45 54 46 20 73 65 73 73 69  ts in IETF sessi
272  6f 6e 73 2c 20 61 73 20 77 65 6c 6c 20 61 73 20  ons, as well as
288  77 72 69 74 74 65 6e 20 61 6e 64 20 65 6c 65 63  written and elec
304  74 72 6f 6e 69 63 20 63 6f 6d 6d 75 6e 69 63 61  tronic communica
320  74 69 6f 6e 73 20 6d 61 64 65 20 61 74 20 61 6e  tions made at an
336  79 20 74 69 6d 65 20 6f 72 20 70 6c 61 63 65 2c  y time or place,
352  20 77 68 69 63 68 20 61 72 65 20 61 64 64 72 65   which are addre
368  73 73 65 64 20 74 6f                             ssed to
```

Ciphertext:

```
000  a3 fb f0 7d f3 fa 2f de 4f 37 6c a2 3e 82 73 70  ...}../.O7l.>.sp
016  41 60 5d 9f 4f 4f 57 bd 8c ff 2c 1d 4b 79 55 ec  A‘].OOW...,.KyU.
032  2a 97 94 8b d3 72 29 15 c8 f3 d3 37 f7 d3 70 05  *....r)....7..p.
048  0e 9e 96 d6 47 b7 c3 9f 56 e0 31 ca 5e b6 25 0d  ....G...V.1.^.%.
064  40 42 e0 27 85 ec ec fa 4b 4b b5 e8 ea d0 44 0e  @B.’....KK....D.
080  20 b6 e8 db 09 d8 81 a7 c6 13 2f 42 0e 52 79 50   ........./B.RyP
096  42 bd fa 77 73 d8 a9 05 14 47 b3 29 1c e1 41 1c  B..ws....G.)..A.
112  68 04 65 55 2a a6 c4 05 b7 76 4d 5e 87 be a8 5a  h.eU*....vM^...Z
128  d0 0f 84 49 ed 8f 72 d0 d6 62 ab 05 26 91 ca 66  ...I..r..b..&..f
144  42 4b c8 6d 2d f8 0e a4 1f 43 ab f9 37 d3 25 9d  BK.m-....C..7.%.
160  c4 b2 d0 df b4 8a 6c 91 39 dd d7 f7 69 66 e9 28  ......l.9...if.(
176  e6 35 55 3b a7 6c 5c 87 9d 7b 35 d4 9e b2 e6 2b  .5U;.l\..{5....+
192  08 71 cd ac 63 89 39 e2 5e 8a 1e 0e f9 d5 28 0f  .q..c.9.^.....(.
208  a8 ca 32 8b 35 1c 3c 76 59 89 cb cf 3d aa 8b 6c  ..2.5.<vY...=..l
224  cc 3a af 9f 39 79 c9 2b 37 20 fc 88 dc 95 ed 84  .:..9y.+7 ......
240  a1 be 05 9c 64 99 b9 fd a2 36 e7 e8 18 b0 4b 0b  ....d....6....K.
256  c3 9c 1e 87 6b 19 3b fe 55 69 75 3f 88 12 8c c0  ....k.;.Uiu?....
272  8a aa 9b 63 d1 a1 6f 80 ef 25 54 d7 18 9c 41 1f  ...c..o..%T...A.
288  58 69 ca 52 c5 b8 3f a3 6f f2 16 b9 c1 d3 00 62  Xi.R..?.o......b
304  be bc fd 2d c5 bc e0 91 19 34 fd a7 9a 86 f6 e6  ...-.....4......
320  98 ce d7 59 c3 ff 9b 64 77 33 8f 3d a4 f9 cd 85  ...Y...dw3.=....
336  14 ea 99 82 cc af b3 41 b2 38 4d d9 02 f3 d1 ab  .......A.8M.....
352  7a c6 1d d2 9c 6f 21 ba 5b 86 2f 37 30 e3 7c fd  z....o!.[./70.|.
368  c4 fd 80 6c 22 f2 21                             ...l".!
```

Test Vector #3:
==============

Key:
```
000  1c 92 40 a5 eb 55 d3 8a f3 33 88 86 04 f6 b5 f0  ..@..U...3......
016  47 39 17 c1 40 2b 80 09 9d ca 5c bc 20 70 75 c0  G9..@+....\. pu.
```

Nonce:
```
000  00 00 00 00 00 00 00 00 00 00 00 02              ............
```

Initial Block Counter = 42

Plaintext:
```
000  27 54 77 61 73 20 62 72 69 6c 6c 69 67 2c 20 61  'Twas brillig, a
016  6e 64 20 74 68 65 20 73 6c 69 74 68 79 20 74 6f  nd the slithy to
032  76 65 73 0a 44 69 64 20 67 79 72 65 20 61 6e 64  ves.Did gyre and
048  20 67 69 6d 62 6c 65 20 69 6e 20 74 68 65 20 77   gimble in the w
064  61 62 65 3a 0a 41 6c 6c 20 6d 69 6d 73 79 20 77  abe:.All mimsy w
080  65 72 65 20 74 68 65 20 62 6f 72 6f 67 6f 76 65  ere the borogove
096  73 2c 0a 41 6e 64 20 74 68 65 20 6d 6f 6d 65 20  s,.And the mome
112  72 61 74 68 73 20 6f 75 74 67 72 61 62 65 2e     raths outgrabe.
```

Ciphertext:
```
000  62 e6 34 7f 95 ed 87 a4 5f fa e7 42 6f 27 a1 df  b.4....._..Bo'..
016  5f b6 91 10 04 4c 0d 73 11 8e ff a9 5b 01 e5 cf  _....L.s....[...
032  16 6d 3d f2 d7 21 ca f9 b2 1e 5f b1 4c 61 68 71  .m=..!...._.Lahq
048  fd 84 c5 4f 9d 65 b2 83 19 6c 7f e4 f6 05 53 eb  ...O.e...l....S.
064  f3 9c 64 02 c4 22 34 e3 2a 35 6b 3e 76 43 12 a6  ..d.."4.*5k>vC..
080  1a 55 32 05 57 16 ea d6 96 25 68 f8 7d 3f 3f 77  .U2.W....%h.}??w
096  04 c6 a8 d1 bc d1 bf 4d 50 d6 15 4b 6d a7 31 b1  .......MP..Km.1.
112  87 b5 8d fd 72 8a fa 36 75 7a 79 7a c1 88 d1     ....r..6uzyz...
```

A.3. Poly1305 Message Authentication Code

Notice how in test vector #2 r is equal to zero. The part of the Poly1305 algorithm where the accumulator is multiplied by r means that with r equal zero, the tag will be equal to s regardless of the content of the Text. Fortunately, all the proposed methods of generating r are such that getting this particular weak key is very unlikely.

Test Vector #1:
==============

One-time Poly1305 Key:
```
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Text to MAC:
```
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
032  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
048  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Tag:
```
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Test Vector #2:
==============

One-time Poly1305 Key:

```
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  36 e5 f6 b5 c5 e0 60 70 f0 ef ca 96 22 7a 86 3e  6.....‘p...."z.>
```

Text to MAC:

```
000  41 6e 79 20 73 75 62 6d 69 73 73 69 6f 6e 20 74  Any submission t
016  6f 20 74 68 65 20 49 45 54 46 20 69 6e 74 65 6e  o the IETF inten
032  64 65 64 20 62 79 20 74 68 65 20 43 6f 6e 74 72  ded by the Contr
048  69 62 75 74 6f 72 20 66 6f 72 20 70 75 62 6c 69  ibutor for publi
064  63 61 74 69 6f 6e 20 61 73 20 61 6c 6c 20 6f 72  cation as all or
080  20 70 61 72 74 20 6f 66 20 61 6e 20 49 45 54 46   part of an IETF
096  20 49 6e 74 65 72 6e 65 74 2d 44 72 61 66 74 20   Internet-Draft
112  6f 72 20 52 46 43 20 61 6e 64 20 61 6e 79 20 73  or RFC and any s
128  74 61 74 65 6d 65 6e 74 20 6d 61 64 65 20 77 69  tatement made wi
144  74 68 69 6e 20 74 68 65 20 63 6f 6e 74 65 78 74  thin the context
160  20 6f 66 20 61 6e 20 49 45 54 46 20 61 63 74 69   of an IETF acti
176  76 69 74 79 20 69 73 20 63 6f 6e 73 69 64 65 72  vity is consider
192  65 64 20 61 6e 20 22 49 45 54 46 20 43 6f 6e 74  ed an "IETF Cont
208  72 69 62 75 74 69 6f 6e 22 2e 20 53 75 63 68 20  ribution". Such
224  73 74 61 74 65 6d 65 6e 74 73 20 69 6e 63 6c 75  statements inclu
240  64 65 20 6f 72 61 6c 20 73 74 61 74 65 6d 65 6e  de oral statemen
256  74 73 20 69 6e 20 49 45 54 46 20 73 65 73 73 69  ts in IETF sessi
272  6f 6e 73 2c 20 61 73 20 77 65 6c 6c 20 61 73 20  ons, as well as
288  77 72 69 74 74 65 6e 20 61 6e 64 20 65 6c 65 63  written and elec
304  74 72 6f 6e 69 63 20 63 6f 6d 6d 75 6e 69 63 61  tronic communica
320  74 69 6f 6e 73 20 6d 61 64 65 20 61 74 20 61 6e  tions made at an
336  79 20 74 69 6d 65 20 6f 72 20 70 6c 61 63 65 2c  y time or place,
352  20 77 68 69 63 68 20 61 72 65 20 61 64 64 72 65   which are addre
368  73 73 65 64 20 74 6f                             ssed to
```

Tag:

```
000  36 e5 f6 b5 c5 e0 60 70 f0 ef ca 96 22 7a 86 3e  6.....‘p...."z.>
```

Test Vector #3:
==============

One-time Poly1305 Key:

```
000  36 e5 f6 b5 c5 e0 60 70 f0 ef ca 96 22 7a 86 3e  6.....‘p...."z.>
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Text to MAC:

```
000  41 6e 79 20 73 75 62 6d 69 73 73 69 6f 6e 20 74  Any submission t
016  6f 20 74 68 65 20 49 45 54 46 20 69 6e 74 65 6e  o the IETF inten
032  64 65 64 20 62 79 20 74 68 65 20 43 6f 6e 74 72  ded by the Contr
048  69 62 75 74 6f 72 20 66 6f 72 20 70 75 62 6c 69  ibutor for publi
064  63 61 74 69 6f 6e 20 61 73 20 61 6c 6c 20 6f 72  cation as all or
080  20 70 61 72 74 20 6f 66 20 61 6e 20 49 45 54 46   part of an IETF
096  20 49 6e 74 65 72 6e 65 74 2d 44 72 61 66 74 20   Internet-Draft
112  6f 72 20 52 46 43 20 61 6e 64 20 61 6e 79 20 73  or RFC and any s
128  74 61 74 65 6d 65 6e 74 20 6d 61 64 65 20 77 69  tatement made wi
144  74 68 69 6e 20 74 68 65 20 63 6f 6e 74 65 78 74  thin the context
160  20 6f 66 20 61 6e 20 49 45 54 46 20 61 63 74 69   of an IETF acti
176  76 69 74 79 20 69 73 20 63 6f 6e 73 69 64 65 72  vity is consider
192  65 64 20 61 6e 20 22 49 45 54 46 20 43 6f 6e 74  ed an "IETF Cont
208  72 69 62 75 74 69 6f 6e 22 2e 20 53 75 63 68 20  ribution". Such
224  73 74 61 74 65 6d 65 6e 74 73 20 69 6e 63 6c 75  statements inclu
240  64 65 20 6f 72 61 6c 20 73 74 61 74 65 6d 65 6e  de oral statemen
256  74 73 20 69 6e 20 49 45 54 46 20 73 65 73 73 69  ts in IETF sessi
272  6f 6e 73 2c 20 61 73 20 77 65 6c 6c 20 61 73 20  ons, as well as
288  77 72 69 74 74 65 6e 20 61 6e 64 20 65 6c 65 63  written and elec
304  74 72 6f 6e 69 63 20 63 6f 6d 6d 75 6e 69 63 61  tronic communica
320  74 69 6f 6e 73 20 6d 61 64 65 20 61 74 20 61 6e  tions made at an
336  79 20 74 69 6d 65 20 6f 72 20 70 6c 61 63 65 2c  y time or place,
352  20 77 68 69 63 68 20 61 72 65 20 61 64 64 72 65   which are addre
368  73 73 65 64 20 74 6f                             ssed to
```

Tag:

```
000  f3 47 7e 7c d9 54 17 af 89 a6 b8 79 4c 31 0c f0  .G~|.T.....yL1..
```

Test Vector #4:
==============

One-time Poly1305 Key:
```
000  1c 92 40 a5 eb 55 d3 8a f3 33 88 86 04 f6 b5 f0  ..@..U...3......
016  47 39 17 c1 40 2b 80 09 9d ca 5c bc 20 70 75 c0  G9..@+....\. pu.
```

Text to MAC:
```
000  27 54 77 61 73 20 62 72 69 6c 6c 69 67 2c 20 61  'Twas brillig, a
016  6e 64 20 74 68 65 20 73 6c 69 74 68 79 20 74 6f  nd the slithy to
032  76 65 73 0a 44 69 64 20 67 79 72 65 20 61 6e 64  ves.Did gyre and
048  20 67 69 6d 62 6c 65 20 69 6e 20 74 68 65 20 77   gimble in the w
064  61 62 65 3a 0a 41 6c 6c 20 6d 69 6d 73 79 20 77  abe:.All mimsy w
080  65 72 65 20 74 68 65 20 62 6f 72 6f 67 6f 76 65  ere the borogove
096  73 2c 0a 41 6e 64 20 74 68 65 20 6d 6f 6d 65 20  s,.And the mome
112  72 61 74 68 73 20 6f 75 74 67 72 61 62 65 2e     raths outgrabe.
```

Tag:
```
000  45 41 66 9a 7e aa ee 61 e7 08 dc 7c bc c5 eb 62  EAf.~..a...|...b
```

Test Vector #5: If one uses 130-bit partial reduction, does the code handle the case where partially reduced final result is not fully reduced?

```
R:
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
S:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data:
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
tag:
03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Test Vector #6: What happens if addition of s overflows modulo 2^128?

```
R:
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
S:
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
data:
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
tag:
03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Test Vector #7: What happens if data limb is all ones and there is carry from lower limb?

```
R:
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
S:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data:
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
F0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
11 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
tag:
05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Test Vector #8: What happens if final result from polynomial part is exactly 2^130-5?

```
R:
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
S:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data:
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FB FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
tag:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Test Vector #9: What happens if final result from polynomial part is exactly 2^130-6?

```
R:
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
S:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data:
FD FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
tag:
FA FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

Test Vector #10: What happens if 5*H+L-type reduction produces 131-bit intermediate result?

```
R:
01 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00
S:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data:
E3 35 94 D7 50 5E 43 B9 00 00 00 00 00 00 00 00
33 94 D7 50 5E 43 79 CD 01 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
tag:
14 00 00 00 00 00 00 00 55 00 00 00 00 00 00 00
```

Test Vector #11: What happens if 5*H+L-type reduction produces 131-bit final result?

```
R:
01 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00
S:
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data:
E3 35 94 D7 50 5E 43 B9 00 00 00 00 00 00 00 00
33 94 D7 50 5E 43 79 CD 01 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
tag:
13 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

A.4. Poly1305 Key Generation Using ChaCha20

Test Vector #1:
==============

The key:

```
000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

The nonce:

```
000  00 00 00 00 00 00 00 00 00 00 00 00              ............
```

Poly1305 one-time key:

```
000  76 b8 e0 ad a0 f1 3d 90 40 5d 6a e5 53 86 bd 28  v.....=.@]j.S..(
016  bd d2 19 b8 a0 8d ed 1a a8 36 ef cc 8b 77 0d c7  .........6...w..
```

```
  Test Vector #2:
  ==============

  The key:
  000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  016  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01  ................

  The nonce:
  000  00 00 00 00 00 00 00 00 00 00 00 02              ............

  Poly1305 one-time key:
  000  ec fa 25 4f 84 5f 64 74 73 d3 cb 14 0d a9 e8 76  ..%O._dts......v
  016  06 cb 33 06 6c 44 7b 87 bc 26 66 dd e3 fb b7 39  ..3.lD{..&f....9


  Test Vector #3:
  ==============

  The key:
  000  1c 92 40 a5 eb 55 d3 8a f3 33 88 86 04 f6 b5 f0  ..@..U...3......
  016  47 39 17 c1 40 2b 80 09 9d ca 5c bc 20 70 75 c0  G9..@+....\. pu.

  The nonce:
  000  00 00 00 00 00 00 00 00 00 00 00 02              ............

  Poly1305 one-time key:
  000  96 5e 3b c6 f9 ec 7e d9 56 08 08 f4 d2 29 f9 4b  .^;...~.V....).K
  016  13 7f f2 75 ca 9b 3f cb dd 59 de aa d2 33 10 ae  ...u..?..Y...3..
```

A.5. ChaCha20-Poly1305 AEAD Decryption

Below we'll see decrypting a message. We receive a ciphertext, a nonce, and a tag. We know the key. We will check the tag, and then (assuming that it validates) decrypt the ciphertext. In this particular protocol, we'll assume that there is no padding of the plaintext.

```
  The key:
  000  1c 92 40 a5 eb 55 d3 8a f3 33 88 86 04 f6 b5 f0  ..@..U...3......
  016  47 39 17 c1 40 2b 80 09 9d ca 5c bc 20 70 75 c0  G9..@+....\. pu.

  Ciphertext:
  000  64 a0 86 15 75 86 1a f4 60 f0 62 c7 9b e6 43 bd  d...u...‘.b...C.
  016  5e 80 5c fd 34 5c f3 89 f1 08 67 0a c7 6c 8c b2  ^.\.4\....g..l..
  032  4c 6c fc 18 75 5d 43 ee a0 9e e9 4e 38 2d 26 b0  Ll..u]C....N8-&.
  048  bd b7 b7 3c 32 1b 01 00 d4 f0 3b 7f 35 58 94 cf  ...<2.....;.5X..
  064  33 2f 83 0e 71 0b 97 ce 98 c8 a8 4a bd 0b 94 81  3/..q......J....
  080  14 ad 17 6e 00 8d 33 bd 60 f9 82 b1 ff 37 c8 55  ...n..3.‘....7.U
  096  97 97 a0 6e f4 f0 ef 61 c1 86 32 4e 2b 35 06 38  ...n...a..2N+5.8
  112  36 06 90 7b 6a 7c 02 b0 f9 f6 15 7b 53 c8 67 e4  6..{j|.....{S.g.
  128  b9 16 6c 76 7b 80 4d 46 a5 9b 52 16 cd e7 a4 e9  ..lv{.MF..R.....
  144  90 40 c5 a4 04 33 22 5e e2 82 a1 b0 a0 6c 52 3e  .@...3"^.....lR>
  160  af 45 34 d7 f8 3f a1 15 5b 00 47 71 8c bc 54 6a  .E4..?..[.Gq..Tj
  176  0d 07 2b 04 b3 56 4e ea 1b 42 22 73 f5 48 27 1a  ..+..VN..B"s.H’.
  192  0b b2 31 60 53 fa 76 99 19 55 eb d6 31 59 43 4e  ..1‘S.v..U..1YCN
  208  ce bb 4e 46 6d ae 5a 10 73 a6 72 76 27 09 7a 10  ..NFm.Z.s.rv’.z.
  224  49 e6 17 d9 1d 36 10 94 fa 68 f0 ff 77 98 71 30  I....6...h..w.q0
  240  30 5b ea ba 2e da 04 df 99 7b 71 4d 6c 6f 2c 29  0[.......{qMlo,)
  256  a6 ad 5c b4 02 2b 02 70 9b                       ..\..+.p.

  The nonce:
  000  00 00 00 00 01 02 03 04 05 06 07 08              ............

  The AAD:
  000  f3 33 88 86 00 00 00 00 00 00 4e 91              .3........N.

  Received Tag:
  000  ee ad 9d 67 89 0c bb 22 39 23 36 fe a1 85 1f 38  ...g..."9#6....8
```

First, we calculate the one-time Poly1305 key

```
@@@  ChaCha state with key set up
      61707865  3320646e  79622d32  6b206574
      a540921c  8ad355eb  868833f3  f0b5f604
      c1173947  09802b40  bc5cca9d  c0757020
      00000000  00000000  04030201  08070605

@@@  ChaCha state after 20 rounds
      a94af0bd  89dee45c  b64bb195  afec8fa1
      508f4726  63f554c0  1ea2c0db  aa721526
      11b1e514  a0bacc0f  828a6015  d7825481
      e8a4a850  d9dcbbd6  4c2de33a  f8ccd912

@@@ out bytes:
bd:f0:4a:a9:5c:e4:de:89:95:b1:4b:b6:a1:8f:ec:af:
26:47:8f:50:c0:54:f5:63:db:c0:a2:1e:26:15:72:aa

Poly1305 one-time key:
000  bd f0 4a a9 5c e4 de 89 95 b1 4b b6 a1 8f ec af  ..J.\.....K.....
016  26 47 8f 50 c0 54 f5 63 db c0 a2 1e 26 15 72 aa  &G.P.T.c....&.r.
```

Next, we construct the AEAD buffer

```
Poly1305 Input:
000  f3 33 88 86 00 00 00 00 00 00 4e 91 00 00 00 00  .3........N.....
016  64 a0 86 15 75 86 1a f4 60 f0 62 c7 9b e6 43 bd  d...u...‘.b...C.
032  5e 80 5c fd 34 5c f3 89 f1 08 67 0a c7 6c 8c b2  ^.\.4\....g..l..
048  4c 6c fc 18 75 5d 43 ee a0 9e e9 4e 38 2d 26 b0  Ll..u]C....N8-&.
064  bd b7 b7 3c 32 1b 01 00 d4 f0 3b 7f 35 58 94 cf  ...<2.....;.5X..
080  33 2f 83 0e 71 0b 97 ce 98 c8 a8 4a bd 0b 94 81  3/..q......J....
096  14 ad 17 6e 00 8d 33 bd 60 f9 82 b1 ff 37 c8 55  ...n..3.‘....7.U
112  97 97 a0 6e f4 f0 ef 61 c1 86 32 4e 2b 35 06 38  ...n...a..2N+5.8
128  36 06 90 7b 6a 7c 02 b0 f9 f6 15 7b 53 c8 67 e4  6..{j|.....{S.g.
144  b9 16 6c 76 7b 80 4d 46 a5 9b 52 16 cd e7 a4 e9  ..lv{.MF..R.....
160  90 40 c5 a4 04 33 22 5e e2 82 a1 b0 a0 6c 52 3e  .@...3"^.....lR>
176  af 45 34 d7 f8 3f a1 15 5b 00 47 71 8c bc 54 6a  .E4..?..[.Gq..Tj
192  0d 07 2b 04 b3 56 4e ea 1b 42 22 73 f5 48 27 1a  ..+..VN..B"s.H’.
208  0b b2 31 60 53 fa 76 99 19 55 eb d6 31 59 43 4e  ..1‘S.v..U..1YCN
224  ce bb 4e 46 6d ae 5a 10 73 a6 72 76 27 09 7a 10  ..NFm.Z.s.rv’.z.
240  49 e6 17 d9 1d 36 10 94 fa 68 f0 ff 77 98 71 30  I....6...h..w.q0
256  30 5b ea ba 2e da 04 df 99 7b 71 4d 6c 6f 2c 29  0[.......{qMlo,)
272  a6 ad 5c b4 02 2b 02 70 9b 00 00 00 00 00 00 00  ..\..+.p........
288  0c 00 00 00 00 00 00 00 09 01 00 00 00 00 00 00  ................
```

We calculate the Poly1305 tag and find that it matches

```
Calculated Tag:
000  ee ad 9d 67 89 0c bb 22 39 23 36 fe a1 85 1f 38  ...g..."9#6....8
```

Finally, we decrypt the ciphertext

```
Plaintext::
000  49 6e 74 65 72 6e 65 74 2d 44 72 61 66 74 73 20  Internet-Drafts
016  61 72 65 20 64 72 61 66 74 20 64 6f 63 75 6d 65  are draft docume
032  6e 74 73 20 76 61 6c 69 64 20 66 6f 72 20 61 20  nts valid for a
048  6d 61 78 69 6d 75 6d 20 6f 66 20 73 69 78 20 6d  maximum of six m
064  6f 6e 74 68 73 20 61 6e 64 20 6d 61 79 20 62 65  onths and may be
080  20 75 70 64 61 74 65 64 2c 20 72 65 70 6c 61 63   updated, replac
096  65 64 2c 20 6f 72 20 6f 62 73 6f 6c 65 74 65 64  ed, or obsoleted
112  20 62 79 20 6f 74 68 65 72 20 64 6f 63 75 6d 65   by other docume
128  6e 74 73 20 61 74 20 61 6e 79 20 74 69 6d 65 2e  nts at any time.
144  20 49 74 20 69 73 20 69 6e 61 70 70 72 6f 70 72   It is inappropr
160  69 61 74 65 20 74 6f 20 75 73 65 20 49 6e 74 65  iate to use Inte
176  72 6e 65 74 2d 44 72 61 66 74 73 20 61 73 20 72  rnet-Drafts as r
192  65 66 65 72 65 6e 63 65 20 6d 61 74 65 72 69 61  eference materia
208  6c 20 6f 72 20 74 6f 20 63 69 74 65 20 74 68 65  l or to cite the
224  6d 20 6f 74 68 65 72 20 74 68 61 6e 20 61 73 20  m other than as
240  2f e2 80 9c 77 6f 72 6b 20 69 6e 20 70 72 6f 67  /...work in prog
256  72 65 73 73 2e 2f e2 80 9d                       ress./...
```

Authors' Addresses

Yoav Nir
Check Point Software Technologies Ltd.
5 Hasolelim st.
Tel Aviv 6789735
Israel

Email: ynir.ietf@gmail.com

Adam Langley
Google Inc

Email: agl@google.com