Network Working Group
Internet-Draft
Intended status: Informational
Expires: October 5, 2014

Y. Nir
Check Point
A. Langley
Google Inc

April 3, 2014

ChaCha20 and Poly1305 for IETF protocols
draft-nir-cfrg-chacha20-poly1305-02

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.

Status of this Memo

Copyright Notice

Table of Contents

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 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 and secure. They are fast even if software-only C-language implementations, allowing for much quicker deployment when compared with algorithms such as AES that are significantly accelerated by hardware implementations.

These 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.

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 martices 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):

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

Where "+" denotes integer addition without carry, "^" 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:

```
o  b = 0x01020304
o  a = 0x11111111
o  d = 0x01234567
o  a = a + b = 0x11111111 + 0x01020304 = 0x12131415
o  d = d ^ a = 0x01234567 ^ 0x12131415 = 0x13305172
o  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.

```
o  a = 0x11111111
o  b = 0x01020304
o  c = 0x9b8d6f43
o  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:
- o A 256-bit key, treated as a concatenation of 8 32-bit little-endian integers.
- o A 96-bit nonce, treated as a concatenation of 3 32-bit little-endian integers.
- o 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 as follows:
- o The first 4 words (0-3) are constants: 0x61707865, 0x3320646e, 0x79622d32, 0x6b206574.
- o 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.
- o 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. Rounds 1-4 are part of the "column" round, while 5-8 are part of the "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, the original input words are added to the output words, and the result is serialized by sequencing the words one-by-one in little-endian order.

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:

- o 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.
- o Nonce = (00:00:00:09:00:00:00:4a:00:00:00:00)
- o 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
```

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. The resulting state is then serialized by writing the numbers in little-endian order. Concatenating the results from the successive blocks forms a key stream, which is then XOR-ed with the plaintext. 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:

- o A 256-bit key
- o 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.
- o A 96-bit nonce. In some protocols, this is known as the Initialization Vector.
- o an arbitrary-length plaintext

The output is an encrypted message of the same length.

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:

- o 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.
- o Nonce = (00:00:00:00:00:00:00:4a:00:00:00:00).
- o 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):

- o r[3], r[7], r[11], and r[15] are required to have their top four bits clear (be smaller than 16)
- o 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:

- o A 256-bit one-time key
- o 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: 3fffffffffffffffffffffffffffffffb. Also set a variable "accumulator" to zero.

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

- o Read the block as a little-endian number.
- o 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.
- o 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.
- o Add this number to the accumulator.
- o Multiply by "r"
- o 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:

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

- o The 256-bit session integrity key is used as the ChaCha20 key.
- o The block counter is set to zero.
- o 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 sender, 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. AEAD Construction

Note: Much of the content of this document, including this AEAD construction is taken from Adam Langley's draft ([agl-draft]) for the use of these algorithms in TLS. The AEAD construction described here is called AEAD_CHACHA20-POLY1305.

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

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

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

- o 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.
- o Next, a Poly1305 one-time key is generated from the 256-bit key and nonce using the procedure described in Section 2.6.
- o The ChaCha20 encryption function is called to encrypt the plaintext, using the same key and nonce, and with the initial counter set to 1.
- o The Poly1305 function is called with the Poly1305 key calculated above, and a message constructed as a concatenation of the following:
  - * The additional data
  - * The length of the additional data in octets (as a 64-bit little-endian integer). TBD: bit count rather than octets? network order?

   * The ciphertext
   * The length of the ciphertext in octets (as a 64-bit little-endian integer). TBD: bit count rather than octets? network order?

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, for 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).

2.7.1. 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 0c 00 00 00|PQRS............
016  00 00 00 00 d3 1a 8d 34 64 8e 60 db 7b 86 af bc|.......4d.`.{...
032  53 ef 7e c2 a4 ad ed 51 29 6e 08 fe a9 e2 b5 a7|S.~....Q)n......
048  36 ee 62 d6 3d be a4 5e 8c a9 67 12 82 fa fb 69|6.b.=..^..g....i
064  da 92 72 8b 1a 71 de 0a 9e 06 0b 29 05 d6 a5 b6|..r..q.....)....
080  7e cd 3b 36 92 dd bd 7f 2d 77 8b 8c 98 03 ae e3|~.;6...-w......
096  28 09 1b 58 fa b3 24 e4 fa d6 75 94 55 85 80 8b|(..X..$...u.U...
112  48 31 d7 bc 3f f4 de f0 8e 4b 7a 9d e5 76 d2 65|H1..?....Kz..v.e
128  86 ce c6 4b 61 16 72 00 00 00 00 00 00 00      |...Ka.r.......


Tag:
18:fb:11:a5:03:1a:d1:3a:7e:3b:03:d4:6e:e3:a6:a7
```

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, it is copied aside only to be added in later. This would be correct, but it saves a few operations to 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

None of the algorithms here are my own. ChaCha20 and Poly1305 were invented by Daniel J. Bernstein, and the AEAD construction was invented by Adam Langley.

Thanks to Robert Ransom and Ilari Liusvaara for their helpful comments and explanations.

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>.

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

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

[agl-draft] Langley, A. and W. Chang, "ChaCha20 and Poly1305 based Cipher Suites for TLS", draft-agl-tls-chacha20poly1305-04 (work in progress), November 2013.

[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).

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