Network Working Group
Internet-Draft
Intended status: Experimental
Expires: July 25, 2014

Y. Nir
Check Point

January 21, 2014

ChaCha20 and Poly1305 and their use in IPsec
draft-nir-ipsecme-chacha20-poly1305-00

Abstract

This document describes the use of the ChaCha20 stream cipher in IPsec, as well as the use of the Poly1305 authenticator, both as stand-alone algorithms, and as a combined mode AEAD algorithm.

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 July 25, 2014.

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, including IPsec VPNs. 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, VPN 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 IPsec implementations to use 3DES. [standby-cipher] describes this issue and the need for a standby cipher in greater detail.

This document proposes such a standby cipher for IPsec. We use ChaCha20 ([chacha]) with or without the Poly1305 ([poly1305]) authenticator.

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. The Algorithms

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

2.1. The ChaCha20 block function

The ChaCha20 block function is part of the ChaCha20 stream cipher

(Section 2.2). It uses a 256-bit key. There are some variations defined, such as 128-bit keys, and 8- and 12-round variants, but this document defines only a 256-bit key and a 20-round ChaCha. ChaCha20 works on 512-bit blocks (64 bytes, or 16 32-bit words). In this section we will describe the ChaCha block function.

ChaCha maps an array of sixteen 32-bit input words to sixteen 32-bit output words. The input words are partitioned 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.
- Word 13 is called Sender ID, or SID. Note that in the original ChaCha20 word 13 is also part of the block count, in case the encrypted data exceeds 256 gigabytes. That is not necessary for IPsec.
- Words 14-15 are a nonce, which should not be repeated for the same combination of key and sender ID. The 14th word is the least significant 32 bits of the input nonce (nonce | 0xffffffff), while the 15th word is the most significant 32 bits (nonce >> 32).

ChaCha20 runs 20 rounds, alternating between "column" and "diagonal" rounds. At the end of 20 rounds, the original input words are added to the output words, and the result is serialized by serializing the numbers in order, with the most significant octet of every 32-bit integer preceding the others. See Appendix A.1 for an example of this serialization.

The inputs to ChaCha20 are:

- A 256-bit key
- A 32-bit sender ID
- A 64-bit nonce, treated as a number
- A 32-bit block count parameter

The output is 64 random-looking bytes.

2.2. 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. There are some variations defined, such as 128-bit keys, and 8- and 12-round variants, but this document defines only a 256-bit key and a 20-round ChaCha.

ChaCha20 successively calls the ChaCha20 block function, with the same key, sender ID and nonce, and with successively increasing block counter parameters. Concatenating the results 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.

The inputs to ChaCha20 are:
- A 256-bit key
- A 32-bit sender ID

- A 32-bit initial counter
- A 64-bit nonce
- an arbitrary-length plaintext

The output is an encrypted message of the same length.

2.3. 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 key MUST NOT be re-used.

The inputs to Poly1305 are:
- A 256-bit one-time key
- An arbitrary length message

The output is a 128-bit tag.

2.4. 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 CHACHA20-POLY1305-ESP.

The inputs to CHACHA20-POLY1305-ESP are:

- A 256-bit key
- A 32-bit sender ID
- A 64-bit nonce
- An arbitrary length plaintext
- 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:

- First the ChaCha20 block function with the key, sender ID, nonce, and zero for the block counter. From the 64-byte output, the first 32 bytes are kept as the Poly1305 256-bit key.
- The ChaCha20 encryption function is called to encrypt the plaintext, using the same key, sender ID, nonce and plaintext, 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 additional data

  * The length of the additional data in octets (as a 32-bit network order integer)
  * The ciphertext
  * The length of the ciphertext in octets (as a 32-bit network order integer)

Decryption is pretty much the same, but will be expanded later on.

The output from the AEAD is twofold:

- A ciphertext of the same length as the plaintext.
- A 128-bit tag (the result of the Poly1305 function.

3. Algorithms for ESP & AH

This document defines three algorithms for use with the Encapsulated Security Protocol (ESP - [RFC4303]) and Authentication Header (AH - [RFC4302]):

- ChaCha20 for use as an encryption algorithms for ESP.
- Poly1305-MAC for use as a message authentication algorithm for ESP and AH.
- ChaCha20-Poly1305-ESP as an AEAD algorithm for ESP.

3.1. ENCR_ChaCha20 for ESP

For ChaCha20 used in ESP, the following parameters are used:

- The IV is 64-bit. Since this is used as the nonce for the ChaCha20 function, this IV MUST NOT repeat. The most natural way to implement this is with a counter, but anything that guarantees uniqueness can be used, such as a linear feedback shift register (LFSR). Note that the encrypter can use any IV generation method that meets the uniqueness requirement, without coordinating with the decrypter.
- The Internet Key Exchange protocol (IKE - [RFC5996]) generates a bitstring called KEYMAT that is generated from a PRF. That KEYMAT is divided into keys for encryption, message authentication and whatever else is needed. For the ChaCha20 algorithm, 256 bits are used for the key.
- The ChaCha20 encryption algorithm is called with the 256-bit key, one (1) for the initial counter, the packet IV as a nonce, and the plaintext. For regular IPsec, the Sender ID is set to zero. For multi-sender SAs, such as described in [RFC6054], there sender ID can be set to a different value for each sender. The 64-bit IV is treated as a 64-bit network order integer, and passed to the ChaCha20 function as a number. The reason that one (1) is used for the initial counter rather than zero is that one is used for encryption in the AEAD algorithm (because zero is reserved for generating the one-time Poly1305 key), so one was used here for

consistency.

- As ChaCha20 is not a block cipher, no padding should be necessary. However, in keeping with the specification in RFC 4303, the ESP does have padding, so as to align the buffer to an integral multiple of 4 octets.

The encryption algorithm transform ID for negotiating this algorithm in IKE is TBA by IANA.

3.2. AUTH_Poly1305 for ESP and AH

You cannot use a part of the keying material directly, because Poly1305 requires a different key for each authenticated message. So the Poly1305 key for each message is generated as follows:

- The key for AUTH_Poly1305 is 256-bits.
- To determine the per-packet Poly1305 key, the ChaCha20 block function is called with the following parameters:
  - The AUTH_Poly1305 256-bit key is used as the key.
  - The sender ID is set to zero.
  - The packet replay counter is used as the nonce. If Extended Sequence Numbers (ESN) are employed, then the full 64-bit is used for the nonce.
  - Zero is used for the block counter.
- The 512-bit result is then truncated. The top 256 bits are used as the one-time key for Poly1305 to calculate the message authentication code (MAC).
- The 128-bit output serves as the MAC for the packet. All 16 bytes are included in the packet.

The integrity algorithm transform ID for negotiating this algorithm in IKE is TBA by IANA.

3.3. ESP_ChaCha20-Poly1305 for ESP

ESP_ChaCha20-Poly1305 is a combined mode algorithm, or AEAD. The construction follows the AEAD construction in Section 2.4:

- As in Section 3.1, the IV is 64-bit, and is used as a nonce.
- Also as in Section 3.1, the encryption key is 256-bit.
- As in Section 3.2, the nonce, along with a block counter of zero is passed to the ChaCha20 block function, and the top part of the result used as the Poly1305 key.
- The ChaCha20 encryption function is then called with the nonce, the key, and an initial counter of zero.
- Finally, the Poly1305 function is run on the data to be authenticated, which is, as specified in Section 2.4 a concatenation of the following:

  - The Authenticated Additional Data (AAD) - see Section 3.3.1.

    * The AAD length in bytes as a 32-bit network order quantity.
    * The ciphertext
    * The length of the ciphertext as a 32-bit network order quantity.

  o The 128-bit output of Poly1305 is used as the tag. All 16 bytes are included in the packet.

The encryption algorithm transform ID for negotiating this algorithm in IKE is TBA by IANA.

3.3.1. AAD Construction

The construction of the Additional Authenticated Data (AAD) is similar to the one in [RFC4106]. For security associations (SAs) with 32-bit sequence numbers the AAD is 8 bytes: 4-byte SPI followed by 4-byte sequence number ordered exactly as it is in the packet. For SAs with ESN the AAD is 12 bytes: 4-byte SPI followed by an 8-byte sequence number as a 64-bit network order integer.

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].

The most important security consideration in implementing this draft is the uniqueness of the nonce used in ChaCha20. This is trivial in AUTH_Poly1305, because the packet sequence number is used as a nonce, but is required for ChaCha20 as well. As said in Section 3.1, the nonce should be selected uniquely for a particular key. 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 same kind of collision risk as in the above paragraph also exists in the selection of the Poly1305 key in both the AEAD construction (Section 2.4) and in the AUTH_Poly1305 algorithm (Section 3.2). ChaCha20 is used to generate a 512-bit value, which is unique to each packet, but this uniqueness is not guaranteed for its 256-bit truncation, which serves as the actual key for Poly1305. While uniqueness is not guaranteed, it is still very likely. Birthday

paradox calculations show that generating Poly1305 keys needs to happen over 2^101 times (way more than the 2^64 allowed by IPsec) to drive the chances of collision up to 0.00000000000001%. This is why we may safely ignore the possibility of collision.

5. IANA Considerations

IANA is requested to assign two values from the IKEv2 "Transform Type 1 - Encryption Algorithm Transform IDs" registry, as follows:

- ENCR_ChaCha20
- ESP_ChaCha20-Poly1305

IANA is also requested to assign one value from the IKEv2 "Transform Type 3 - Integrity Algorithm Transform IDs" registry with name "AUTH_Poly1305".

6. Acknowledgements

Much of the text in this document was inspired by Adam Langley's draft for TLS, and by "inspired" I mean "shamelessly copied". The author would also like to thank Adam, Watson Ladd and Dave McGrew for allaying his fears about writing a document describing crypto.

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.

[RFC4302] Kent, S., "IP Authentication Header", RFC 4302, December 2005.

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, December 2005.

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

[RFC6054] McGrew, D. and B. Weis, "Using Counter Modes with Encapsulating Security Payload (ESP) and Authentication Header (AH) to Protect Group Traffic", RFC 6054,

November 2010.

[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, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

[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>.

[RFC4106] Viega, J. and D. McGrew, "The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP)", RFC 4106, June 2005.

[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.

[standby-cipher] McGrew, D., Grieco, A., and Y. Sheffer, "Selection of Future Cryptographic Standards", draft-mcgrew-standby-cipher (work in progress).

Appendix A. Examples of the algorithms

A.1. Example of the block function

In this example, we show a single run of the ChaCha20 block. For our example, we choose

- Key:
  000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f

- Sender ID: 0
- Nonce: fedcba9876543210
- Block Count: 5

The block is set up as follows:

```
61707865 3320646e 79622d32 6b206574
03020100 07060504 0b0a0908 0f0e0d0c
13121110 17161514 1b1a1918 1f1e1d1c
00000005 00000000 76543210 fedcba98
```

After running the block function, the result is as follows:

```
a435cd80 a16b9b89 9112c0a4 487b4230
0d98f8cf b5b59396 4cce17a6 d17db31a
6dd35e45 658dff64 52b615fb b3c17e96
31d2725f ecd5246f 739bdab7 07cf7593
```

The resulting octet string looks like this:

```
a435cd80a16b9b899112c0a4487b42300d98f8cfb5b593964cce17a6d17db31a
6dd35e45658dff6452b615fbb3c17e9631d2725fecd5246f739bdab707cf7593
```

A.2. Example of ChaCha20 Encryption in IPsec

In this example, we show a run of the ChaCha20 encryption function. For our example, we chose some arbitrary data. Also,

- Key:
  000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
- Sender ID: 0
- IV: 0x000000000000004a (74)
- Message Length: 80 octets

o The message itself:

35:6f:16:49:d0:2a:7f:19:30:f1:8f:af:e6:d1:69:d3:22:12:1f:76:78:be:6c
af:90:17:f3:fb:8e:12:d3:f1:f7:dc:7c:31:02:70:36:54:0c:5a:37:aa:31:fd
8a:e2:31:73:8a:ff:cc:77:8c:ba:c4:7b:6b:d1:17:6c:ce:cc:90:56:ae:65:b9
c6:af:40:e6:9d:f4:37:8f:4a:a9:f3

First, we need to apply padding. We will use an ESP-style padding, rounding up the length of the packet to 84 octets, with two padding bytes, a pad length, and a Next Header field set to 0x04:

35:6f:16:49:d0:2a:7f:19:30:f1:8f:af:e6:d1:69:d3:22:12:1f:76:78:be:6c
af:90:17:f3:fb:8e:12:d3:f1:f7:dc:7c:31:02:70:36:54:0c:5a:37:aa:31:fd
8a:e2:31:73:8a:ff:cc:77:8c:ba:c4:7b:6b:d1:17:6c:ce:cc:90:56:ae:65:b9
c6:af:40:e6:9d:f4:37:8f:4a:a9:f3:01:02:02:04

For 84 octets, we need two blocks. We call the block function twice, once with the block counter set to 1, and then again with the block counter set to two. The results are as follows:

Block 1 before rounds:
```
61707865 3320646e 79622d32 6b206574
03020100 07060504 0b0a0908 0f0e0d0c
13121110 17161514 1b1a1918 1f1e1d1c
00000001 00000000 0000004a 00000000
```

Block 1 after rounds:
```
9944f9d8 0f69b9cc 9002b600 559b9586
04579a7b a9a146c0 a601d9dd 8a141b06
d08a69bb e737527d 0ee7970d ff8512d8
d959240f 0f09eb18 eb0f2a3a ee0fbcf2
```

Block 2 before rounds:
```
61707865 3320646e 79622d32 6b206574
03020100 07060504 0b0a0908 0f0e0d0c
13121110 17161514 1b1a1918 1f1e1d1c
00000002 00000000 0000004a 00000000
```

Block 2 after rounds:
```
213bebc0 720ce566 26195c71 84e2ee1d
511f64c2 b8ed11da ede4e571 d9b57c34
```

```
   0aa05765 dd073b1c 22d7dc84 86c91bc0
   a9b89717 dbf8c56f 5822cafd deec62ff

84 bytes of key stream:
99:44:f9:d8:0f:69:b9:cc:90:02:b6:00:55:9b:95:86:04:57:9a:7b:a9:a1:46
c0:a6:01:d9:dd:8a:14:1b:06:d0:8a:69:bb:e7:37:52:7d:0e:e7:97:0d:ff:85
12:d8:d9:59:24:0f:0f:09:eb:18:eb:0f:2a:3a:ee:0f:bc:f2:21:3b:eb:c0:72
0c:e5:66:26:19:5c:71:84:e2:ee:1d:51:1f:64:c2

XOR'd with the plaintext:
ac:2b:ef:91:df:43:c6:d5:a0:f3:39:af:b3:4a:fc:55:26:45:85:0d:d1:1f:2a
6f:36:16:2a:26:04:06:c8:f7:27:56:15:8a:e5:47:64:29:02:bd:a0:a7:ce:78
98:3a:e8:2a:ae:f0:c3:7e:67:a2:2f:74:41:eb:f9:63:72:3e:b1:6d:45:a5:cb
ca:4a:26:c0:84:a8:46:0b:a8:47:ee:50:1d:66:c6
```

A.3. Example of the AUTH_Poly1305

In this example, we show a run of the AUTH_Poly1305 message authentication function. For our example, we choose some arbitrary data - the same data as in the ENCH_ChaCha20 example above. Also,

- Key:
  000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f

- Replay Counter: 0x0000004a (74)
- Message Length: 80 octets
- The message itself:

```
35:6f:16:49:d0:2a:7f:19:30:f1:8f:af:e6:d1:69:d3:22:12:1f:76:78:be:6c
af:90:17:f3:fb:8e:12:d3:f1:f7:dc:7c:31:02:70:36:54:0c:5a:37:aa:31:fd
8a:e2:31:73:8a:ff:cc:77:8c:ba:c4:7b:6b:d1:17:6c:ce:cc:90:56:ae:65:b9
c6:af:40:e6:9d:f4:37:8f:4a:a9:f3
```

First, we run the ChaCha20 algorithm to generate the one-time Poly1305 key:

```
Key block before the rounds:
   61707865 3320646e 79622d32 6b206574
   03020100 07060504 0b0a0908 0f0e0d0c
   13121110 17161514 1b1a1918 1f1e1d1c
   00000000 00000000 0000004a 00000000
```

Key block after rounds:
   64345099 66639817 2113da66 371489fa
   091f5ff3 c681552b b0f0a4b5 d0d7e6dd
   5401d1e1 b6ff30e5 25a5a57c b1af2f06
   8a064ae1 39b28c25 9698451b c688b187

256-bit one-time Poly1305 key:
643450996663981721 13da66371489fa091f5ff3c681552bb0f0a4b5d0d7e6dd

Tag: 43dc3ef779160bb54943a2279570d679

A.4. Example of ESP_ChaCha20-Poly1305 in IPsec

In this example, we encrypt the same data as in Appendix A.2, and also authenticate it. For the purposes of this example, we will assume that ESN is not used.

- Key:
  000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
- Sender ID: 0
- SPI: 0x7890abcd
- IV: 0x000000000000004a (74)
- Message Length: 80 octets
- The message itself:

35:6f:16:49:d0:2a:7f:19:30:f1:8f:af:e6:d1:69:d3:22:12:1f:76:78:be:6c
af:90:17:f3:fb:8e:12:d3:f1:f7:dc:7c:31:02:70:36:54:0c:5a:37:aa:31:fd
8a:e2:31:73:8a:ff:cc:77:8c:ba:c4:7b:6b:d1:17:6c:ce:cc:90:56:ae:65:b9
c6:af:40:e6:9d:f4:37:8f:4a:a9:f3

First, we repeat the calculations in Appendix A.3 to get the one-time

Poly1305 key:

Key block before the rounds:
   61707865 3320646e 79622d32 6b206574
   03020100 07060504 0b0a0908 0f0e0d0c
   13121110 17161514 1b1a1918 1f1e1d1c
   00000000 00000000 0000004a 00000000

Key block after rounds:
   64345099 66639817 2113da66 371489fa
   091f5ff3 c681552b b0f0a4b5 d0d7e6dd

```
   5401d1e1 b6ff30e5 25a5a57c b1af2f06
   8a064ae1 39b28c25 9698451b c688b187
```

256-bit one-time Poly1305 key:
6434509966639817211 3da66371489fa091f5ff3c681552bb0f0a4b5d0d7e6dd

Next, we need to apply padding. We will use an ESP-style padding, rounding up the length of the packet to 84 octets, with two padding bytes, a pad length, and a Next Header field set to 0x04:

35:6f:16:49:d0:2a:7f:19:30:f1:8f:af:e6:d1:69:d3:22:12:1f:76:78:be:6c
af:90:17:f3:fb:8e:12:d3:f1:f7:dc:7c:31:02:70:36:54:0c:5a:37:aa:31:fd
8a:e2:31:73:8a:ff:cc:77:8c:ba:c4:7b:6b:d1:17:6c:ce:cc:90:56:ae:65:b9
c6:af:40:e6:9d:f4:37:8f:4a:a9:f3:01:02:02:04

Step 3, we run the ChaCha20 block twice to create sufficient keystream to XOR with all the (padded) plaintext.

Block 1 before rounds:

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

```
   00000001 00000000 0000004a 00000000

Block 1 after rounds:
   9944f9d8 0f69b9cc 9002b600 559b9586
   04579a7b a9a146c0 a601d9dd 8a141b06
   d08a69bb e737527d 0ee7970d ff8512d8
   d959240f 0f09eb18 eb0f2a3a ee0fbcf2

Block 2 before rounds:
   61707865 3320646e 79622d32 6b206574
   03020100 07060504 0b0a0908 0f0e0d0c
   13121110 17161514 1b1a1918 1f1e1d1c
   00000002 00000000 0000004a 00000000

Block 2 after rounds:
   213bebc0 720ce566 26195c71 84e2ee1d
   511f64c2 b8ed11da ede4e571 d9b57c34
   0aa05765 dd073b1c 22d7dc84 86c91bc0
   a9b89717 dbf8c56f 5822cafd deec62ff

84 bytes of key stream:
99:44:f9:d8:0f:69:b9:cc:90:02:b6:00:55:9b:95:86:04:57:9a:7b:a9:a1:46
c0:a6:01:d9:dd:8a:14:1b:06:d0:8a:69:bb:e7:37:52:7d:0e:e7:97:0d:ff:85
12:d8:d9:59:24:0f:0f:09:eb:18:eb:0f:2a:3a:ee:0f:bc:f2:21:3b:eb:c0:72
0c:e5:66:26:19:5c:71:84:e2:ee:1d:51:1f:64:c2

XOR'd with the plaintext:
ac:2b:ef:91:df:43:c6:d5:a0:f3:39:af:b3:4a:fc:55:26:45:85:0d:d1:1f:2a
6f:36:16:2a:26:04:06:c8:f7:27:56:15:8a:e5:47:64:29:02:bd:a0:a7:ce:78
98:3a:e8:2a:ae:f0:c3:7e:67:a2:2f:74:41:eb:f9:63:72:3e:b1:6d:45:a5:cb
ca:4a:26:c0:84:a8:46:0b:a8:47:ee:50:1d:66:c6
```

next, we construct the AEAD. As described in Section 3.3.1, the AAD is a concatenation of the 4-byte SPI, followed by a 4-byte sequence number. In this case: 78:90:ab:cd:00:00:00:4a. The Poly1305 function is run over a concatenation of the AAD, the length of the AAD as a 32-bit integer (8), The ciphertext, and the length of the ciphertext (84):

Buffer to be authenticated:
78:90:ab:cd:00:00:00:4a:00:00:00:08:ac:2b:ef:91:df:43:c6:d5:a0:f3:39
af:b3:4a:fc:55:26:45:85:0d:d1:1f:2a:6f:36:16:2a:26:04:06:c8:f7:27:56
15:8a:e5:47:64:29:02:bd:a0:a7:ce:78:98:3a:e8:2a:ae:f0:c3:7e:67:a2:2f
74:41:eb:f9:63:72:3e:b1:6d:45:a5:cb:ca:4a:26:c0:84:a8:46:0b:a8:47:ee
50:1d:66:c6:00:00:00:54

Tag: 7af839e8f4961cd99f0600dc7b8ad8d6

Author's Address

Yoav Nir
Check Point Software Technologies Ltd.
5 Hasolelim st.
Tel Aviv 6789735
Israel

Email: ynir@checkpoint.com