Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 24 January 2026

D. Miller
OpenSSH
S.G. Tatham
PuTTY
S. Josefsson
Independent

23 July 2025

Secure Shell (SSH) authenticated encryption cipher: chacha20-poly1305
draft-ietf-sshm-chacha20-poly1305-02

Abstract

This document describes the Secure Shell (SSH) chacha20-poly1305 authenticated encryption cipher.

Status of This Memo

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

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

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 24 January 2026.

Copyright Notice

Table of Contents

1. Introduction

ChaCha20 is a stream cipher designed by Daniel Bernstein and described in [ChaCha], with some details inherited from [Salsa20]. It operates by permuting 128 fixed bits, 128 or 256 bits of key, a 64 bit nonce and a 64 bit counter into 64 bytes of output. This output is used as a keystream, with any unused bytes simply discarded.

Poly1305 [Poly1305], also by Daniel Bernstein, is a one-time Carter-Wegman MAC that computes a 128 bit integrity tag given a message and a single-use 256 bit secret key.

The "chacha20-poly1305" cipher combines these two primitives into an authenticated encryption mode. The construction used is based on that proposed for TLS by Adam Langley in [I-D.agl-tls-chacha20poly1305], but differs in the layout of data passed to the MAC and in the addition of encryption of the packet lengths. In particular, the key generation for Poly1305 is based on ChaCha20, instead of AES as described in [Poly1305].

This document specifies and registers "chacha20-poly1305" to be identical to the already widely deployed "chacha20-poly1305@openssh.com".

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. ChaCha20

ChaCha20 is defined by two long documents, but the actual cipher definition is simple enough to restate the full specification here.

ChaCha20 operates on an array of sixteen 32-bit integers, usually written in the form of a square matrix, with x[0], ..., x[3] on the top row, ..., x[12], ..., x[15] on the bottom row.

The ChaCha20 permutation is a fixed, keyless transformation which turns an input matrix into an output one. It is described in full by the following pseudocode, in which addition is modulo 2^32, and ROL describes bitwise left rotation of 32-bit integers:

```
one_16th_of_a_round(matrix, p, q, r, rotation):
    matrix[p] <- matrix[p] + matrix[q]
    matrix[r] <- (matrix[r] XOR matrix[p]) ROL rotation

one_quarter_of_a_round(matrix, a, b, c, d):
    one_16th_of_a_round(matrix, a, b, d, 16)
    one_16th_of_a_round(matrix, c, d, b, 12)
    one_16th_of_a_round(matrix, a, b, d, 8)
    one_16th_of_a_round(matrix, c, d, b, 7)

even_numbered_round(matrix):
    one_quarter_of_a_round(matrix, 0, 4,  8, 12)
    one_quarter_of_a_round(matrix, 1, 5,  9, 13)
    one_quarter_of_a_round(matrix, 2, 6, 10, 14)
    one_quarter_of_a_round(matrix, 3, 7, 11, 15)

odd_numbered_round(matrix):
    one_quarter_of_a_round(matrix, 0, 5, 10, 15)
    one_quarter_of_a_round(matrix, 1, 6, 11, 12)
    one_quarter_of_a_round(matrix, 2, 7,  8, 13)
    one_quarter_of_a_round(matrix, 3, 4,  9, 14)

chacha20(matrix):
    orig <- copy(matrix)
    repeat 10 times:
        even_numbered_round(matrix)
        odd_numbered_round(matrix)
    matrix = matrix + orig   // element-wise addition of two matrices
```

Figure 1: Pseudocode of the ChaCha20 transform

To generate a ChaCha20 cipher block, the first four words x[0],...,x[3] of the input matrix are initialized to fixed values obtained by encoding the string "expand 32-byte k" in ASCII and interpreting those 16 bytes as the little-endian encoding of four 32-bit integers. The next 8 words x[4],...,x[11] are initialized from the 32-byte cipher key in the same fashion. The next 2 words x[12],x[13] are a 64-bit counter which is incremented for each output data block, with x[12] being the low-order 32 bits of the counter and x[13] the high-order bits. The final 2 words x[14],x[15] are initialized from an 8-byte "nonce" which must be a unique identifier for each separate message encrypted with the same key.

To create a full ChaCha20 cipher stream, a succession of input matrices is constructed as above, incrementing the 64-bit block counter by 1 in each matrix. Each input matrix is independently transformed via the permutation described above, and the output matrix is converted into 64 bytes of cipher stream by encoding each

32-bit integer little-endian. Generate as many blocks of the cipher stream as necessary; XOR the cipher stream byte-wise into the message to be encrypted; discard any remaining part of the final block.

4. Poly1305

The Poly1305 authenticator used in this specification differs from the description in [Poly1305] in how it generates the key material, so it is convenient to specify that in full here as well.

Poly1305 as described here requires two 16-byte key inputs. Each one is interpreted as the little-endian encoding of a 128-bit integer. One of these integers is then bitwise-ANDed with the integer 0x0FFFFFFC0FFFFFFC0FFFFFFC0FFFFFFF, and the result is denoted as 'r'. The other integer is denoted as 'n', and requires no modification.

The input message to be authenticated is divided into blocks of 16 bytes, with a short block at the end if its length is not a multiple of 16. Each block is converted into an integer of size up to 2^129, by appending the byte 0x01 and then interpreting the extended block as the little-endian encoding of an integer. (So all the resulting integers have bit 128 set, except that if there is a final short block of s < 16 bytes, then that integer will have bit (8*s) set.)

Denote the message integers as C[1], C[2], ..., C[m]. Then the MAC is computed as follows.

Perform a polynomial evaluation in which the variable is the key integer 'r'; the constant coefficient is zero; the coefficient of r^1 is c[m], the coefficient of r^2 is c[m-1], ..., and the coefficient of r^m is c[1]. This can be achieved by initializing an accumulator A to zero, and then for each coefficient c[i], in turn, setting A <- (A + c[i]) * r.

The result of this evaluation is reduced modulo 2^130-5 to give the least non-negative residue. (It follows that the intermediate results of the polynomial evaluation can be reduced in the same way, without affecting the output.) That residue is then reduced again modulo 2^128, and added modulo 2^128 to the other key integer 'n'. This final 128-bit integer is encoded little-endian and forms the output authentication tag.

In the original specification [Poly1305], the key provided by the application consists of 16 bytes of data used to create 'r' as described above, and another 16 bytes used as an AES key to encrypt a per-message nonce, with the ciphertext being used to create 'n'. So 'r' is reused between messages, with only 'n' changing. However, in this specification, both 'r' and 'n' are created from parts of the output of the same ChaCha20 transformation, as described below. So both values change in every message.

5. Negotiation

The "chacha20-poly1305" offers both encryption and authentication. As such, no separate MAC is required. If the "chacha20-poly1305" cipher is selected in key exchange, the offered MAC algorithms are ignored and no MAC is required to be negotiated.

6. Detailed Construction

The "chacha20-poly1305" cipher requires 512 bits of key material as output from the SSH key exchange. This forms two 256 bit keys (K_1 and K_2), used by two separate instances of chacha20.

The instance keyed by the second half of the key material, K_2, is a stream cipher that is used only to encrypt the 4 byte packet length field. The instance keyed by the first half K_1 is used to encrypt the remainder of the packet. The concatenated ciphertext from both of these operations is then authenticated using Poly1305 with a key derived from the K_1 cipher stream.

The 8-byte nonce required by ChaCha20 is obtained by encoding the SSH packet sequence number as a 64-bit integer, in the usual SSH big-endian encoding. Since the SSH packet sequence number is defined to be a 32-bit integer, this means that the first four bytes of the ChaCha20 nonce, creating the input matrix entry x[14], are always zero. The second four bytes are the big-endian encoding of the SSH sequence number, which means that the matrix entry x[15] is the sequence number with its byte order reversed, because that results from decoding the same four bytes little-endian.

For the K_2 cipher instance that encrypts the length, the block counter is always 0, because only one block is needed (and only the first four bytes of that block are used). For the K_1 cipher instance used for the packet payload, the block counter value starts at 1 and increments to generate the cipher stream that encrypts the packet.

To generate keying information for the Poly1305 MAC, the K_1 cipher instance is run with the same key and nonce but a block counter value of 0. The first 16 bytes of the output are converted into the Poly1305 key integer 'r' at which the polynomial is evaluated, by decoding them as a 128-bit little-endian integer and clearing certain bits of that integer as described in [Poly1305]. The second 16 bytes are converted into the mask integer 'n' added at the end of the Poly1305 algorithm.

7. Packet Handling

When receiving a packet, the length must be decrypted first. When 4 bytes of ciphertext length have been received, they may be decrypted using the K_2 key, a nonce consisting of the packet sequence number encoded as a uint64 under the usual SSH wire encoding and a zero block counter to obtain the plaintext length.

Once the entire packet has been received, the MAC MUST be checked before decryption. A per-packet Poly1305 key is generated as described above and the MAC tag calculated using Poly1305 with this key over the ciphertext of the packet length and the payload together. The calculated MAC is then compared in constant time with the one appended to the packet and the packet decrypted using ChaCha20 as described above (with K_1, the packet sequence number as nonce and a starting block counter of 1).

To send a packet, first encode the 4 byte length and encrypt it using K_2. Encrypt the packet payload (using K_1) and append it to the encrypted length. Finally, calculate a MAC tag and append it.

8. Rekeying

ChaCha20 must never reuse a {key, nonce} for encryption nor may it be used to encrypt more than 2^70 bytes under the same {key, nonce}. The SSH Transport protocol [RFC4253] recommends a far more conservative rekeying every 1GB of data sent or received. If this recommendation is followed, then "chacha20-poly1305" requires no special handling in this area.

9. Acknowledgements

Markus Friedl helped on the design. Theo de Raadt supported to register the OpenSSH private-use cipher for generic use.

10. Security Considerations

The use of this cipher and MAC combination has a KNOWN SECURITY VULNERABILITY. This specification modifies the SSH binary packet protocol in a way that makes the encryption and authentication of every packet independent: instead of generating a single continuous cipher stream and dividing it at packet boundaries, a fresh cipher stream is started for each packet based on that packet's sequence number. This makes it possible for a MITM attacker to delete an entire packet from one direction of the SSH data stream, if they can manipulate the receiving end's idea of the packet sequence number. In the "Terrapin" attack [Terrapin] (CVE-2023-48795), this is achieved by combining deletion of the first encrypted packet with injection of an earlier SSH_MSG_IGNORE packet, in the cleartext segment of the SSH session before the initial key exchange completes. The injected and deleted packets have opposite effects on the sequence number and cancel one another out, so that the receiving side uses the correct sequence number to decrypt the first remaining packet in the cipher stream, and never detects the modification. This is a breach of the SSH transport protocol's security guarantee.

Implementations deploying the "chacha20-poly1305" cipher and MAC combination MUST support a means of mitigating this attack, and deploy the mitigation if the other end of the SSH connection is also capable of doing so.

As of 2025, a mitigation commonly deployed is called Strict KEX [I-D.ietf-sshm-strict-kex]. In this modification, the packet sequence number used for cryptographic purposes is reset to 0 immediately after each SSH_MSG_NEWKEYS, and extraneous packets such as SSH_MSG_IGNORE are prohibited in the initial cleartext segment of the session.

On the other hand, the use of two separate cipher instances is itself a mitigation for another family of attacks found in SSH (e.g. CVE-2008-5161), in which an attacker uses the fact that the receiver acts on the decrypted length before checking the MAC, to obtain an oracle for the packet payload cipher by modifying the packet length field and observing the receiver's response. In this protocol, the length is encrypted using an independently-keyed cipher, so any attempt to exploit the decryption of the length field only gains information about the length cipher, and not about the separate cipher instance that encrypts the packet payload.

In the original specification for Poly1305, the nonce value added to the result of the polynomial evaluation was generated by AES. In this specification, ChaCha20 is used instead, and there is an assumption this doesn't introduce new risks.

11. IANA Considerations

IANA is requested to add a new "Encryption Algorithm Name" of "chacha20-poly1305" the "Encryption Algorithm Names" registry for Secure Shell (SSH) Protocol Parameters [IANA-ENCALG] with a "Reference" field referring this RFC.

12. Normative References

[I-D.ietf-sshm-strict-kex]
Miller, D., "SSH Strict KEX extension", Work in Progress, Internet-Draft, draft-ietf-sshm-strict-kex-00, 21 July 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-sshm-strict-kex-00>.

[IANA-ENCALG]
IANA, "Secure Shell (SSH) Protocol Parameters: Encryption Algorithm Names", <https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml#ssh-parameters-17>.

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

[RFC4253] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, DOI 10.17487/RFC4253, January 2006, <https://www.rfc-editor.org/info/rfc4253>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

13. Informative References

[ChaCha] Bernstein, J., "ChaCha, a variant of Salsa20", January 2008, <http://cr.yp.to/chacha/chacha-20080128.pdf>.

[I-D.agl-tls-chacha20poly1305]
Langley, A. and W. Chang, "ChaCha20 and Poly1305 based Cipher Suites for TLS", Work in Progress, Internet-Draft, draft-agl-tls-chacha20poly1305-04, 22 November 2013, <https://datatracker.ietf.org/doc/html/draft-agl-tls-chacha20poly1305-04>.

[Poly1305] Bernstein, J., "The Poly1305-AES message-authentication code", March 2005, <http://cr.yp.to/mac/poly1305-20050329.pdf>.

[Salsa20] Bernstein, J., "The Salsa20 family of stream ciphers", December 2007, <http://cr.yp.to/snuffle/salsafamily-20071225.pdf>.

[Terrapin] B辰umer, F., Brinkmann, M., and J. Schwenk, "Terrapin Attack: Breaking SSH Channel Integrity By Sequence Number Manipulation", December 2023, <https://arxiv.org/abs/2312.12422>.

Appendix A. Worked Example

Since this specification modifies both Poly1305 (by generating its key data using a different cipher) and the SSH binary packet protocol, it is useful to present a full example of how an SSH packet is encrypted and authenticated to generate the exact data sent over the network.

The example input packet, described using the conventional SSH marshalling notation, is as follows:

```
byte      0x5e (SSH2_MSG_CHANNEL_DATA)
uint32    0 (recipient channel id)
string    "Lorem ipsum dolor sit amet, consectetur adipisicing elit"
```

Figure 2: Logical form of the input packet

After encoding in the binary packet protocol, this becomes the following data:

```
uint32    0x48 (packet length not including this field)
byte      0x06 (number of bytes of padding)
byte      0x5e (SSH2_MSG_CHANNEL_DATA)
uint32    0 (recipient channel id)
string    "Lorem ipsum dolor sit amet, consectetur adipisicing elit"
byte[6]   0x4e 0x43 0xe8 0x04 0xdc 0x6c (random padding bytes)
```

Figure 3: Full encoding of the input packet before encryption

which in turn is encoded into the following bytes:

```
00 00 00 48  06 5e 00 00  00 00 00 00  00 38 4c 6f
72 65 6d 20  69 70 73 75  6d 20 64 6f  6c 6f 72 20
73 69 74 20  61 6d 65 74  2c 20 63 6f  6e 73 65 63
74 65 74 75  72 20 61 64  69 70 69 73  69 63 69 6e
67 20 65 6c  69 74 4e 43  e8 04 dc 6c
```

Figure 4: Input packet encoded as hex bytes, before encryption

In this example connection, this packet was sent from client to server, with a sequence number of 7.

The encryption key for the client-to-server direction of the connection, derived as specified in RFC4253 section 7.2, is 64 bytes long and consists of the following bytes:

```
8b bf f6 85  5f c1 02 33  8c 37 3e 73  aa c0 c9 14
f0 76 a9 05  b2 44 4a 32  ee ca ff ea  e2 2b ec c5
e9 b7 a7 a5  82 5a 82 49  34 6e c1 c2  83 01 cf 39
45 43 fc 75  69 88 7d 76  e1 68 f3 75  62 ac 07 40
```

Figure 5: Key material output from the key exchange

To encrypt the first 4 bytes of the packet, consisting of the length, the following matrix of 32-bit integers is constructed as input to the ChaCha20 permutation, with the key material derived from the second half of the above block, and a block counter value of 0. (Note that the packet sequence number, in the bottom right corner, has been endianness-swapped as a result of encoding it big-endian in the SSH convention and then decoding it little-endian in the ChaCha20 convention.)

```
0x61707865  0x3320646e  0x79622d32  0x6b206574
0xa5a7b7e9  0x49825a82  0xc2c16e34  0x39cf0183
0x75fc4345  0x767d8869  0x75f368e1  0x4007ac62
0x00000000  0x00000000  0x00000000  0x07000000
```

Figure 6: ChaCha20 preimage matrix for encrypting the packet length

After the ChaCha20 transformation, the output matrix is as follows:

```
0xaccc3e2c  0xf62f4341  0x94f7b07b  0xe6f481fb
0x75306ddf  0x8b82a326  0x5b1aec13  0x119ff043
0x0ae8ba12  0x1cb72290  0xb1565876  0xf5fee756
0x8ea9bef4  0x6ae96e0d  0x32148e17  0xf9531ec9
```

Figure 7: ChaCha20 output matrix for encrypting the packet length

To encrypt the 32-bit length field, only 32 bits of this cipher stream are required. The first word, 0xaccc3e2c, is encoded little-endian into the four bytes 2c 3e cc ac, and these are XORed with the big-endian SSH length field 00 00 00 48 to produce the encrypted length field 2c 3e cc e4.

```
2c 3e cc e4
```

Figure 8: Encrypted bytes of the packet length field

The remaining 60 bytes of this output block are discarded. When encrypting the length field of the next packet, a fresh data block will be generated using that packet's sequence number.

To encrypt the remainder of the packet, two cipher blocks are required. These are generated from the first 32 bytes of the cipher key data, and have block counter values 1 and 2 (value 0 is used to generate the MAC input). The two matrices input to ChaCha20 are therefore as follows, differing only in the block counter at the start of the bottom row:

```
0x61707865  0x3320646e  0x79622d32  0x6b206574
0x85f6bf8b  0x3302c15f  0x733e378c  0x14c9c0aa
0x05a976f0  0x324a44b2  0xeaffcaee  0xc5ec2be2
0x00000001  0x00000000  0x00000000  0x07000000

0x61707865  0x3320646e  0x79622d32  0x6b206574
0x85f6bf8b  0x3302c15f  0x733e378c  0x14c9c0aa
0x05a976f0  0x324a44b2  0xeaffcaee  0xc5ec2be2
0x00000002  0x00000000  0x00000000  0x07000000
```

Figure 9: ChaCha20 preimage matrices for encrypting the main packet data

The ChaCha20 permutation transforms these two matrices into the following output blocks:

```
0x8905e2a3  0x7b7af05b  0xa9fa6ea9  0x5cc14cfa
0xc2f3c7ea  0x88a92e6d  0x25b5c029  0xee0caac8
0x57e5cf62  0x54d6a747  0x055c2b83  0x11c56757
0x2c0b08c4  0x02606ea4  0xfea28051  0xee4eabfc

0x69c3a5f4  0xb55efbdd  0x80a9caea  0xf270c0a0
0xa45d5d78  0x0b44d75b  0xf75247b7  0xe14c0e37
0x98039827  0xd8b8e8de  0xbf7035b6  0xc90bef5b
0x33e3c45a  0x51418cf6  0xcafeae62  0x6da70d32
```

Figure 10: ChaCha20 output matrices for encrypting the main packet data

These blocks are converted back into bytes in little-endian encoding, to create 128 bytes of data beginning a3 e2 05 89 ... and ending ... 32 0d a7 6d. The 72 bytes of packet data (apart from the separately encrypted length field) are XORed with the first 72 bytes of this data to produce the following encrypted output:

```
a5 bc 05 89  5b f0 7a 7b  a9 56 b6 c6  88 29 ac 7c
83 b7 80 b7  00 0e cd e7  45 af c7 05  bb c3 78 ce
03 a2 80 23  6b 87 b5 3b  ed 58 39 66  23 02 b1 64
b6 28 6a 48  cd 1e 09 71  38 e3 cb 90  9b 8b 2b 82
9d d1 8d 2a  35 ff 82 d9
```

Figure 11: Encrypted bytes of the packet, excluding the length field

in which, for example, the first byte 0xa5 is the bitwise XOR of the first cipher stream byte 0xa3 with the byte 0x06 from the cleartext packet (indicating the number of padding bytes).

The encrypted length field is prefixed to the above, to produce the complete 76-byte ciphertext:

```
2c 3e cc e4  a5 bc 05 89  5b f0 7a 7b  a9 56 b6 c6
88 29 ac 7c  83 b7 80 b7  00 0e cd e7  45 af c7 05
bb c3 78 ce  03 a2 80 23  6b 87 b5 3b  ed 58 39 66
23 02 b1 64  b6 28 6a 48  cd 1e 09 71  38 e3 cb 90
9b 8b 2b 82  9d d1 8d 2a  35 ff 82 d9
```

Figure 12: Encrypted bytes of the whole packet, including its length

This ciphertext is the input to the Poly1305 MAC, which is computed as follows.

The packet data is converted into the following sequence of polynomial coefficients, by repeatedly taking a maximum of 16 bytes from the start of the data, appending the byte 0x01, and decoding as a little-endian integer:

```
C[1] = 0x1c6b656a97b7af05b8905bca5e4cc3e2c
C[2] = 0x105c7af45e7cd0e00b780b7837cac2988
C[3] = 0x1663958ed3bb5876b2380a203ce78c3bb
C[4] = 0x190cbe33871091ecd486a28b664b10223
C[5] =         0x1d982ff352a8dd19d822b8b9b
```

Figure 13: Poly1305 coefficients derived from the encrypted packet

The MAC key is constructed by generating a block of ChaCha20 cipher stream, using the same cipher that encrypted most of the packet, with a block counter value of 0. So the input to the ChaCha20 transformation is the following matrix:

```
0x61707865  0x3320646e  0x79622d32  0x6b206574
0x85f6bf8b  0x3302c15f  0x733e378c  0x14c9c0aa
0x05a976f0  0x324a44b2  0xeaffcaee  0xc5ec2be2
0x00000000  0x00000000  0x00000000  0x07000000
```

Figure 14: ChaCha20 preimage matrix for keying the MAC

and the ChaCha20 permutation transforms it into this:

```
0xfba86ef6  0x046d187a  0xa6b4d75d  0xa4487348
0xebc13a8f  0xe0be63fa  0x65a5e1c1  0xdd5b9fd0
0x95f6ca26  0x509e2eec  0xb9f34fe2  0xc565e785
0xd52da783  0xc8f24915  0xb3907730  0x1da12e04
```

Figure 15: ChaCha20 output matrix for keying the MAC

The top row of this matrix is converted into a 128-bit integer by recombining the four words in little-endian order, to make 0xa4487348a6b4d75d046d187afba86ef6. This is transformed into the Poly1305 integer 'r' at which the polynomial is evaluated, by computing its bitwise AND with the integer 0x0FFFFFFC0FFFFFFC0FFFFFFC0FFFFFFF. The result is r = 0x0448734806b4d75c046d18780ba86ef6.

The second row of the matrix is similarly converted into a 128-bit integer 0xdd5b9fd065a5e1c1e0be63faebc13a8f, which is used as the mask value 'n' added at the end of the computation. The remaining two rows of the ChaCha20 output are discarded.

The Poly1305 MAC is constructed by calculating the least non-negative residue mod 2^130-5 of the following expression:

```
C[1] * r^5 + C[2] * r^4 + C[3] * r^3 + C[4] * r^[2] + C[5] * r
```

Figure 16: Polynomial expression evaluated while computing Poly1305

The output of this computation is the integer 0x2db8d08018cd015cc486e8c6099dcfa06. This is reduced mod 2^128 to obtain 0xdb8d08018cd015cc486e8c6099dcfa06, which is then added modulo 2^128 to the nonce value 0xdd5b9fd065a5e1c1e0be63faebc13a8f, giving 0xb8e8a7d1f275f78e292cf05b859e3495. This is encoded as a 16-bit little-endian integer to give the final authentication tag:

```
95 34 9e 85  5b f0 2c 29  8e f7 75 f2  d1 a7 e8 b8
```

Figure 17: Bytes of the final Poly1305 authentication tag

The full encrypted and authenticated data of this packet, sent over the TCP connection to the SSH server, is constructed by concatenating the encrypted length field, the rest of the encrypted packet data, and the authentication tag, and is as follows:

```
2c 3e cc e4  a5 bc 05 89  5b f0 7a 7b  a9 56 b6 c6
88 29 ac 7c  83 b7 80 b7  00 0e cd e7  45 af c7 05
bb c3 78 ce  03 a2 80 23  6b 87 b5 3b  ed 58 39 66
23 02 b1 64  b6 28 6a 48  cd 1e 09 71  38 e3 cb 90
9b 8b 2b 82  9d d1 8d 2a  35 ff 82 d9  95 34 9e 85
5b f0 2c 29  8e f7 75 f2  d1 a7 e8 b8
```

Figure 18: Full wire format of the packet, including the MAC

Authors' Addresses

Damien Miller
OpenSSH
Email: djm@mindrot.org

Simon Tatham
PuTTY
Email: anakin@pobox.com

Simon Josefsson
Independent
Email: simon@josefsson.org