Internet Engineering Task Force (IETF)
Request for Comments: 7914
Category: Informational
ISSN: 2070-1721

C. Percival
Tarsnap
S. Josefsson
SJD AB

August 2016

The scrypt Password-Based Key Derivation Function

Abstract

This document specifies the password-based key derivation function scrypt. The function derives one or more secret keys from a secret string. It is based on memory-hard functions, which offer added protection against attacks using custom hardware. The document also provides an ASN.1 schema.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7914.

Copyright Notice

Table of Contents

1. Introduction

Password-based key derivation functions are used in cryptography and security protocols for deriving one or more secret keys from a secret value. Over the years, several password-based key derivation functions have been used, including the original DES-based UNIX Crypt-function, FreeBSD MD5 crypt, Public-Key Cryptography Standards#5 (PKCS#5) PBKDF2 [RFC2898] (typically used with SHA-1), GNU SHA-256/512 crypt [SHA2CRYPT], Windows NT LAN Manager (NTLM) [NTLM] hash, and the Blowfish-based bcrypt [BCRYPT]. These algorithms are all based on a cryptographic primitive combined with salting and/or iteration. The iteration count is used to slow down the computation, and the salt is used to make pre-computation costlier.

All password-based key derivation functions mentioned above share the same weakness against powerful attackers. Provided that the number of iterations used is increased as computer systems get faster, this allows legitimate users to spend a constant amount of time on key derivation without losing ground to attackers' ever-increasing computing power -- as long as attackers are limited to the same software implementations as legitimate users. While parallelized hardware implementations may not change the number of operations performed compared to software implementations, this does not prevent them from dramatically changing the asymptotic cost, since in many contexts -- including the embarrassingly parallel task of performing a brute-force search for a passphrase -- dollar-seconds are the most appropriate units for measuring the cost of a computation. As semiconductor technology develops, circuits do not merely become faster; they also become smaller, allowing for a larger amount of parallelism at the same cost.

Consequently, with existing key derivation algorithms, even when the iteration count is increased so that the time taken to verify a password remains constant, the cost of finding a password by using a brute-force attack implemented in hardware drops each year.

The scrypt function aims to reduce the advantage that attackers can gain by using custom-designed parallel circuits for breaking password-based key derivation functions.

This document does not introduce scrypt for the first time. The original scrypt paper [SCRYPT] was published as a peer-reviewed scientific paper and contains further background and discussions.

The purpose of this document is to serve as a stable reference for documents making use of scrypt. The rest of this document is divided into sections that each describe parameter choices and algorithm steps needed for the final "scrypt" algorithm.

2. scrypt Parameters

The scrypt function takes several parameters. The passphrase P is typically a human-chosen password. The salt is normally uniquely and randomly generated [RFC4086]. The parameter r ("blockSize") specifies the block size. The CPU/Memory cost parameter N ("costParameter") must be larger than 1, a power of 2, and less than 2^(128 * r / 8). The parallelization parameter p ("parallelizationParameter") is a positive integer less than or equal to ((2^32-1) * 32) / (128 * r). The intended output length dkLen is the length in octets of the key to be derived ("keyLength"); it is a positive integer less than or equal to (2^32 - 1) * 32.

Users of scrypt can tune the parameters N, r, and p according to the amount of memory and computing power available, the latency-bandwidth product of the memory subsystem, and the amount of parallelism desired. At the current time, r=8 and p=1 appears to yield good results, but as memory latency and CPU parallelism increase, it is likely that the optimum values for both r and p will increase. Note also that since the computations of SMix are independent, a large value of p can be used to increase the computational cost of scrypt without increasing the memory usage; so we can expect scrypt to remain useful even if the growth rates of CPU power and memory capacity diverge.

3. The Salsa20/8 Core Function

Salsa20/8 Core is a round-reduced variant of the Salsa20 Core. It is a hash function from 64-octet strings to 64-octet strings. Note that Salsa20/8 Core is not a cryptographic hash function since it is not collision resistant. See Section 8 of [SALSA20SPEC] for its specification and [SALSA20CORE] for more information. The algorithm description, in C language, is included below as a stable reference, without endianness conversion and alignment.

```
#define R(a,b) (((a) << (b)) | ((a) >> (32 - (b))))
void salsa20_word_specification(uint32 out[16],uint32 in[16])
{
  int i;
  uint32 x[16];
  for (i = 0;i < 16;++i) x[i] = in[i];
  for (i = 8;i > 0;i -= 2) {
    x[ 4] ^= R(x[ 0]+x[12], 7);  x[ 8] ^= R(x[ 4]+x[ 0], 9);
    x[12] ^= R(x[ 8]+x[ 4],13);  x[ 0] ^= R(x[12]+x[ 8],18);
    x[ 9] ^= R(x[ 5]+x[ 1], 7);  x[13] ^= R(x[ 9]+x[ 5], 9);
    x[ 1] ^= R(x[13]+x[ 9],13);  x[ 5] ^= R(x[ 1]+x[13],18);
    x[14] ^= R(x[10]+x[ 6], 7);  x[ 2] ^= R(x[14]+x[10], 9);
    x[ 6] ^= R(x[ 2]+x[14],13);  x[10] ^= R(x[ 6]+x[ 2],18);
    x[ 3] ^= R(x[15]+x[11], 7);  x[ 7] ^= R(x[ 3]+x[15], 9);
    x[11] ^= R(x[ 7]+x[ 3],13);  x[15] ^= R(x[11]+x[ 7],18);
    x[ 1] ^= R(x[ 0]+x[ 3], 7);  x[ 2] ^= R(x[ 1]+x[ 0], 9);
    x[ 3] ^= R(x[ 2]+x[ 1],13);  x[ 0] ^= R(x[ 3]+x[ 2],18);
    x[ 6] ^= R(x[ 5]+x[ 4], 7);  x[ 7] ^= R(x[ 6]+x[ 5], 9);
    x[ 4] ^= R(x[ 7]+x[ 6],13);  x[ 5] ^= R(x[ 4]+x[ 7],18);
    x[11] ^= R(x[10]+x[ 9], 7);  x[ 8] ^= R(x[11]+x[10], 9);
    x[ 9] ^= R(x[ 8]+x[11],13);  x[10] ^= R(x[ 9]+x[ 8],18);
    x[12] ^= R(x[15]+x[14], 7);  x[13] ^= R(x[12]+x[15], 9);
    x[14] ^= R(x[13]+x[12],13);  x[15] ^= R(x[14]+x[13],18);
  }
  for (i = 0;i < 16;++i) out[i] = x[i] + in[i];
}
```

4. The scryptBlockMix Algorithm

The scryptBlockMix algorithm is the same as the BlockMix algorithm described in [SCRYPT] but with Salsa20/8 Core used as the hash function H. Below, Salsa(T) corresponds to the Salsa20/8 Core function applied to the octet vector T.

```
Algorithm scryptBlockMix

Parameters:
         r       Block size parameter.

Input:
         B[0] || B[1] || ... || B[2 * r - 1]
                Input octet string (of size 128 * r octets),
                treated as 2 * r 64-octet blocks,
                where each element in B is a 64-octet block.

Output:
         B'[0] || B'[1] || ... || B'[2 * r - 1]
                Output octet string.

Steps:

  1. X = B[2 * r - 1]

  2. for i = 0 to 2 * r - 1 do
       T = X xor B[i]
       X = Salsa (T)
       Y[i] = X
     end for

  3. B' = (Y[0], Y[2], ..., Y[2 * r - 2],
           Y[1], Y[3], ..., Y[2 * r - 1])
```

5. The scryptROMix Algorithm

The scryptROMix algorithm is the same as the ROMix algorithm described in [SCRYPT] but with scryptBlockMix used as the hash function H and the Integerify function explained inline.

```
Algorithm scryptROMix

Input:
         r       Block size parameter.
         B       Input octet vector of length 128 * r octets.
         N       CPU/Memory cost parameter, must be larger than 1,
                 a power of 2, and less than 2^(128 * r / 8).

Output:
         B'      Output octet vector of length 128 * r octets.

Steps:

  1. X = B

  2. for i = 0 to N - 1 do
       V[i] = X
       X = scryptBlockMix (X)
     end for

  3. for i = 0 to N - 1 do
       j = Integerify (X) mod N
              where Integerify (B[0] ... B[2 * r - 1]) is defined
              as the result of interpreting B[2 * r - 1] as a
              little-endian integer.
       T = X xor V[j]
       X = scryptBlockMix (T)
     end for

  4. B' = X
```

6. The scrypt Algorithm

The PBKDF2-HMAC-SHA-256 function used below denotes the PBKDF2 algorithm [RFC2898] used with HMAC-SHA-256 [RFC6234] as the Pseudorandom Function (PRF). The HMAC-SHA-256 function generates 32-octet outputs.

```
Algorithm scrypt

Input:
         P       Passphrase, an octet string.
         S       Salt, an octet string.
         N       CPU/Memory cost parameter, must be larger than 1,
                 a power of 2, and less than 2^(128 * r / 8).
         r       Block size parameter.
         p       Parallelization parameter, a positive integer
                 less than or equal to ((2^32-1) * hLen) / MFLen
                 where hLen is 32 and MFlen is 128 * r.
         dkLen   Intended output length in octets of the derived
                 key; a positive integer less than or equal to
                 (2^32 - 1) * hLen where hLen is 32.

Output:
         DK      Derived key, of length dkLen octets.

Steps:

 1. Initialize an array B consisting of p blocks of 128 * r octets
    each:
     B[0] || B[1] || ... || B[p - 1] =
       PBKDF2-HMAC-SHA256 (P, S, 1, p * 128 * r)

 2. for i = 0 to p - 1 do
       B[i] = scryptROMix (r, B[i], N)
     end for

 3. DK = PBKDF2-HMAC-SHA256 (P, B[0] || B[1] || ... || B[p - 1],
                              1, dkLen)
```

7. ASN.1 Syntax

This section defines ASN.1 syntax for the scrypt key derivation function (KDF). This is intended to operate on the same abstraction level as PKCS#5's PBKDF2. The OID id-scrypt below can be used where id-PBKDF2 is used, with scrypt-params corresponding to PBKDF2-params. The intended application of these definitions includes PKCS #8 and other syntax for key management.

The object identifier id-scrypt identifies the scrypt key derivation function.

```
id-scrypt OBJECT IDENTIFIER ::= {1 3 6 1 4 1 11591 4 11}
```

The parameters field associated with this OID in an AlgorithmIdentifier shall have type scrypt-params:

```
scrypt-params ::= SEQUENCE {
       salt OCTET STRING,
       costParameter INTEGER (1..MAX),
       blockSize INTEGER (1..MAX),
       parallelizationParameter INTEGER (1..MAX),
       keyLength INTEGER (1..MAX) OPTIONAL }
```

The fields of type scrypt-params have the following meanings:

- salt specifies the salt value. It shall be an octet string.

- costParameter specifies the CPU/Memory cost parameter N.

- blockSize specifies the block size parameter r.

- parallelizationParameter specifies the parallelization parameter.

- keyLength, an optional field, is the length in octets of the derived key. The maximum key length allowed depends on the implementation; it is expected that implementation profiles may further constrain the bounds. This field only provides convenience; the key length is not cryptographically protected.

To be usable in PKCS#8 [RFC5208] and Asymmetric Key Packages [RFC5958], the following extension of the PBES2-KDFs type is needed:

```
PBES2-KDFs ALGORITHM-IDENTIFIER ::=
    { {scrypt-params IDENTIFIED BY id-scrypt}, ... }
```

7.1. ASN.1 Module

For reference purposes, the ASN.1 syntax is presented as an ASN.1 module here.

```
-- scrypt ASN.1 Module

scrypt-0 {1 3 6 1 4 1 11591 4 10}

DEFINITIONS ::= BEGIN

id-scrypt OBJECT IDENTIFIER ::= {1 3 6 1 4 1 11591 4 11}

scrypt-params ::= SEQUENCE {
    salt OCTET STRING,
    costParameter INTEGER (1..MAX),
    blockSize INTEGER (1..MAX),
    parallelizationParameter INTEGER (1..MAX),
    keyLength INTEGER (1..MAX) OPTIONAL
}

PBES2-KDFs ALGORITHM-IDENTIFIER ::=
       { {scrypt-params IDENTIFIED BY id-scrypt}, ... }

END
```

8. Test Vectors for Salsa20/8 Core

Below is a sequence of octets that illustrate input and output values for the Salsa20/8 Core. The octets are hex encoded and whitespace is inserted for readability. The value corresponds to the first input and output pair generated by the first scrypt test vector below.

```
INPUT:
7e 87 9a 21 4f 3e c9 86 7c a9 40 e6 41 71 8f 26
ba ee 55 5b 8c 61 c1 b5 0d f8 46 11 6d cd 3b 1d
ee 24 f3 19 df 9b 3d 85 14 12 1e 4b 5a c5 aa 32
76 02 1d 29 09 c7 48 29 ed eb c6 8d b8 b8 c2 5e

OUTPUT:
a4 1f 85 9c 66 08 cc 99 3b 81 ca cb 02 0c ef 05
04 4b 21 81 a2 fd 33 7d fd 7b 1c 63 96 68 2f 29
b4 39 31 68 e3 c9 e6 bc fe 6b c5 b7 a0 6d 96 ba
e4 24 cc 10 2c 91 74 5c 24 ad 67 3d c7 61 8f 81
```

9. Test Vectors for scryptBlockMix

Below is a sequence of octets that illustrate input and output values for scryptBlockMix. The test vector uses an r value of 1. The octets are hex encoded and whitespace is inserted for readability. The value corresponds to the first input and output pair generated by the first scrypt test vector below.

```
INPUT
B[0] =  f7 ce 0b 65 3d 2d 72 a4 10 8c f5 ab e9 12 ff dd
        77 76 16 db bb 27 a7 0e 82 04 f3 ae 2d 0f 6f ad
        89 f6 8f 48 11 d1 e8 7b cc 3b d7 40 0a 9f fd 29
        09 4f 01 84 63 95 74 f3 9a e5 a1 31 52 17 bc d7

B[1] =  89 49 91 44 72 13 bb 22 6c 25 b5 4d a8 63 70 fb
        cd 98 43 80 37 46 66 bb 8f fc b5 bf 40 c2 54 b0
        67 d2 7c 51 ce 4a d5 fe d8 29 c9 0b 50 5a 57 1b
        7f 4d 1c ad 6a 52 3c da 77 0e 67 bc ea af 7e 89

OUTPUT
B'[0] = a4 1f 85 9c 66 08 cc 99 3b 81 ca cb 02 0c ef 05
        04 4b 21 81 a2 fd 33 7d fd 7b 1c 63 96 68 2f 29
        b4 39 31 68 e3 c9 e6 bc fe 6b c5 b7 a0 6d 96 ba
        e4 24 cc 10 2c 91 74 5c 24 ad 67 3d c7 61 8f 81

B'[1] = 20 ed c9 75 32 38 81 a8 05 40 f6 4c 16 2d cd 3c
        21 07 7c fe 5f 8d 5f e2 b1 a4 16 8f 95 36 78 b7
        7d 3b 3d 80 3b 60 e4 ab 92 09 96 e5 9b 4d 53 b6
        5d 2a 22 58 77 d5 ed f5 84 2c b9 f1 4e ef e4 25
```

10. Test Vectors for scryptROMix

Below is a sequence of octets that illustrate input and output values for scryptROMix. The test vector uses an r value of 1 and an N value of 16. The octets are hex encoded and whitespace is inserted for readability. The value corresponds to the first input and output pair generated by the first scrypt test vector below.

```
INPUT:
B = f7 ce 0b 65 3d 2d 72 a4 10 8c f5 ab e9 12 ff dd
    77 76 16 db bb 27 a7 0e 82 04 f3 ae 2d 0f 6f ad
    89 f6 8f 48 11 d1 e8 7b cc 3b d7 40 0a 9f fd 29
    09 4f 01 84 63 95 74 f3 9a e5 a1 31 52 17 bc d7
    89 49 91 44 72 13 bb 22 6c 25 b5 4d a8 63 70 fb
    cd 98 43 80 37 46 66 bb 8f fc b5 bf 40 c2 54 b0
    67 d2 7c 51 ce 4a d5 fe d8 29 c9 0b 50 5a 57 1b
    7f 4d 1c ad 6a 52 3c da 77 0e 67 bc ea af 7e 89

OUTPUT:
B = 79 cc c1 93 62 9d eb ca 04 7f 0b 70 60 4b f6 b6
    2c e3 dd 4a 96 26 e3 55 fa fc 61 98 e6 ea 2b 46
    d5 84 13 67 3b 99 b0 29 d6 65 c3 57 60 1f b4 26
    a0 b2 f4 bb a2 00 ee 9f 0a 43 d1 9b 57 1a 9c 71
    ef 11 42 e6 5d 5a 26 6f dd ca 83 2c e5 9f aa 7c
    ac 0b 9c f1 be 2b ff ca 30 0d 01 ee 38 76 19 c4
    ae 12 fd 44 38 f2 03 a0 e4 e1 c4 7e c3 14 86 1f
    4e 90 87 cb 33 39 6a 68 73 e8 f9 d2 53 9a 4b 8e
```

11. Test Vectors for PBKDF2 with HMAC-SHA-256

Below is a sequence of octets that illustrate input and output values for PBKDF2-HMAC-SHA-256. The octets are hex encoded and whitespace is inserted for readability. The test vectors below can be used to verify the PBKDF2-HMAC-SHA-256 [RFC2898] function. The password and salt strings are passed as sequences of ASCII [RFC20] octets.

```
PBKDF2-HMAC-SHA-256 (P="passwd", S="salt",
                    c=1, dkLen=64) =
55 ac 04 6e 56 e3 08 9f ec 16 91 c2 25 44 b6 05
f9 41 85 21 6d de 04 65 e6 8b 9d 57 c2 0d ac bc
49 ca 9c cc f1 79 b6 45 99 16 64 b3 9d 77 ef 31
7c 71 b8 45 b1 e3 0b d5 09 11 20 41 d3 a1 97 83

PBKDF2-HMAC-SHA-256 (P="Password", S="NaCl",
                     c=80000, dkLen=64) =
4d dc d8 f6 0b 98 be 21 83 0c ee 5e f2 27 01 f9
64 1a 44 18 d0 4c 04 14 ae ff 08 87 6b 34 ab 56
a1 d4 25 a1 22 58 33 54 9a db 84 1b 51 c9 b3 17
6a 27 2b de bb a1 d0 78 47 8f 62 b3 97 f3 3c 8d
```

12. Test Vectors for scrypt

For reference purposes, we provide the following test vectors for scrypt, where the password and salt strings are passed as sequences of ASCII [RFC20] octets.

The parameters to the scrypt function below are, in order, the password P (octet string), the salt S (octet string), the CPU/Memory cost parameter N, the block size parameter r, the parallelization parameter p, and the output size dkLen. The output is hex encoded and whitespace is inserted for readability.

```
scrypt (P="", S="",
        N=16, r=1, p=1, dklen=64) =
77 d6 57 62 38 65 7b 20 3b 19 ca 42 c1 8a 04 97
f1 6b 48 44 e3 07 4a e8 df df fa 3f ed e2 14 42
fc d0 06 9d ed 09 48 f8 32 6a 75 3a 0f c8 1f 17
e8 d3 e0 fb 2e 0d 36 28 cf 35 e2 0c 38 d1 89 06

scrypt (P="password", S="NaCl",
        N=1024, r=8, p=16, dkLen=64) =
fd ba be 1c 9d 34 72 00 78 56 e7 19 0d 01 e9 fe
7c 6a d7 cb c8 23 78 30 e7 73 76 63 4b 37 31 62
2e af 30 d9 2e 22 a3 88 6f f1 09 27 9d 98 30 da
c7 27 af b9 4a 83 ee 6d 83 60 cb df a2 cc 06 40

scrypt (P="pleaseletmein", S="SodiumChloride",
        N=16384, r=8, p=1, dkLen=64) =
70 23 bd cb 3a fd 73 48 46 1c 06 cd 81 fd 38 eb
fd a8 fb ba 90 4f 8e 3e a9 b5 43 f6 54 5d a1 f2
d5 43 29 55 61 3f 0f cf 62 d4 97 05 24 2a 9a f9
e6 1e 85 dc 0d 65 1e 40 df cf 01 7b 45 57 58 87

scrypt (P="pleaseletmein", S="SodiumChloride",
        N=1048576, r=8, p=1, dkLen=64) =
21 01 cb 9b 6a 51 1a ae ad db be 09 cf 70 f8 81
ec 56 8d 57 4a 2f fd 4d ab e5 ee 98 20 ad aa 47
8e 56 fd 8f 4b a5 d0 9f fa 1c 6d 92 7c 40 f4 c3
37 30 40 49 e8 a9 52 fb cb f4 5c 6f a7 7a 41 a4
```

13. Test Vectors for PKCS#8

PKCS#8 [RFC5208] and Asymmetric Key Packages [RFC5958] encode encrypted private-keys. Using PBES2 with scrypt as the KDF, the following illustrates an example of a PKCS#8-encoded private-key. The password is "Rabbit" (without the quotes) with N=1048576, r=8, and p=1. The salt is "Mouse" and the encryption algorithm used is aes256-CBC. The derived key is: E2 77 EA 2C AC B2 3E DA-FC 03 9D 22 9B 79 DC 13 EC ED B6 01 D9 9B 18 2A-9F ED BA 1E 2B FB 4F 58.


```
-----BEGIN ENCRYPTED PRIVATE KEY-----
MIHiME0GCSqGSIb3DQEFDTBAMB8GCSsGAQQB2kcECzASBAVNb3VzZQIDEAAAAgEI
AgEBMB0GCWCGSAFlAwQBKgQQyYmguHMsOwzGMPoyObk/JgSBkJb47EWd5iAqJlyy
+ni5ftd6gZgOPaLQClL7mEZc2KQay0VhjZm/7MbBUNbqOAXNM6OGebXxVp6sHUAL
iBGY/Dls7B1TsWeGObE0sS1MXEpuREuloZjcsNVcNXWPlLdZtkSH6uwWzR0PyG/Z
+ZXfNodZtd/voKlvLOw5B3opGIFaLkbtLZQwMiGtl42AS89lZg==
-----END ENCRYPTED PRIVATE KEY-----
```


14. Security Considerations

This document specifies a cryptographic algorithm, and there is always a risk that someone will find a weakness in it. By following the cryptographic research area, you may learn of publications relevant to scrypt.

ROMix has been proven sequential memory-hard under the random oracle model for the hash function. The security of scrypt relies on the assumption that BlockMix with Salsa20/8 Core does not exhibit any "shortcuts" that would allow it to be iterated more easily than a random oracle. For other claims about the security properties, see [SCRYPT].

Passwords and other sensitive data, such as intermediate values, may continue to be stored in memory, core dumps, swap areas, etc., for a long time after the implementation has processed them. This makes attacks on the implementation easier. Thus, implementation should consider storing sensitive data in protected memory areas. How to achieve this is system dependent.

By nature and depending on parameters, running the scrypt algorithm may require large amounts of memory. Systems should protect against a denial-of-service attack resulting from attackers presenting unreasonably large parameters.

Poor parameter choices can be harmful for security; for example, if you tune the parameters so that memory use is reduced to small amounts that will affect the properties of the algorithm.

15. References

15.1. Normative References

[RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, DOI 10.17487/RFC2898, September 2000, <http://www.rfc-editor.org/info/rfc2898>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <http://www.rfc-editor.org/info/rfc6234>.

15.2. Informative References

[BCRYPT] Provos, N. and D. Mazieres, "A Future-Adaptable Password Scheme", USENIX 1999, June 1999, <https://www.usenix.org/legacy/event/usenix99/provos/provos.pdf>.

[NTLM] Microsoft, "[MS-NLMP]: NT LAN Manager (NTLM) Authentication Protocol", 2015, <https://msdn.microsoft.com/en-us/library/cc236621.aspx>.

[RFC20] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <http://www.rfc-editor.org/info/rfc20>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <http://www.rfc-editor.org/info/rfc4086>.

[RFC5208] Kaliski, B., "Public-Key Cryptography Standards (PKCS) #8: Private-Key Information Syntax Specification Version 1.2", RFC 5208, DOI 10.17487/RFC5208, May 2008, <http://www.rfc-editor.org/info/rfc5208>.

[RFC5958] Turner, S., "Asymmetric Key Packages", RFC 5958, DOI 10.17487/RFC5958, August 2010, <http://www.rfc-editor.org/info/rfc5958>.

[SALSA20CORE] Bernstein, D., "The Salsa20 Core", March 2005, <http://cr.yp.to/salsa20.html>.

[SALSA20SPEC]
Bernstein, D., "Salsa20 specification", April 2005, <http://cr.yp.to/snuffle/spec.pdf>.

[SCRYPT] Percival, C., "STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS", BSDCan'09, May 2009, <http://www.tarsnap.com/scrypt/scrypt.pdf>.

[SHA2CRYPT]
Drepper, U., "Unix crypt using SHA-256 and SHA-512", April 2008, <http://www.akkadia.org/drepper/SHA-crypt.txt>.

Acknowledgements

Text in this document was borrowed from [SCRYPT] and [RFC2898]. The PKCS#8 test vector was provided by Stephen N. Henson.

Feedback on this document was received from Dmitry Chestnykh, Alexander Klink, Rob Kendrick, Royce Williams, Ted Rolle, Jr., Eitan Adler, Stephen Farrel, Nikos Mavrogiannopoulos, and Paul Kyzivat.

Authors' Addresses

Colin Percival
Tarsnap

Email: cperciva@tarsnap.com

Simon Josefsson
SJD AB

Email: simon@josefsson.org
URI: http://josefsson.org/