Internet Research Task Force
Internet-Draft
Intended status: Informational
Expires: January 29, 2018

D. Harkins
HP Enterprise

July 28, 2017

Public Key Exchange
draft-harkins-pkex-04

Abstract

This memo describes a password-authenticated protocol to allow two devices to exchange "raw" (uncertified) public keys and establish trust that the keys belong to their respective identities.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 29, 2018.

Copyright Notice

Table of Contents

1. Introduction

Many authenticated key exchange protocols allow for authentication using uncertified, or "raw", public keys. Usually these specifications-- e.g. [RFC7250] for TLS and [RFC7670] for IKEv2-- assume keys are exchanged in some out-of-band mechanism.

[RFC7250] further states that "the main security challenge [to using ’raw’ public keys] is how to associate the public key with a specific entity. Without a secure binding between identifier and key, the protocol will be vulnerable to man-in-the- middle attacks."

The Public Key Exchange (PKEX) is designed to fill that gap: it establishs a secure binding between exchanged public keys and identifiers, it provides proof-of-possession of the exchanged public keys to each peer, and it enables the establishment of trust in

public keys that can subsequently be used to faccilitate authentication in other authentication and key exchange protocols.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

1.2. Notation

This memo describes a cryptographic exchange using sets of elements called groups. Groups can be either traditional finite field or can be based on elliptic curves. The public keys exchanged by PKEX are elements in a group. Elements in groups are denoted in upper-case and scalar values are denoted with lower-case. The generator of the group is G.

When both the initator and responder use a similar, but unique, datum it is denoted by appending an "i" for initiator or "r" for responder, e.g. if each side needs an element C then the initiator's is Ci and the responder's is Cr.

During the exchange, one side will generate data and the other side will attempt to reconstruct it. The reconstructed data is "primed". That is, if the initiator generates C then when responder tries to reconstruct it, the responder will refer to it as C'. Data that is directly sent and received is not primed.

The following notation is used in this memo:

C = A + B
: The "group operation" on two elements, A and B, that produces a third element, C. For finite field cryptography this is the modular multiplication, for elliptic curve cryptography this is point addition.

C = A - B
: The "group operation" on element A and the inverse of element B to produce a third element, C. Inversion is defined such that the group operation on an element and its inverse results in the identity element, the value one (1) for finite field cryptography and the "point at infinity" for elliptic curve cryptography.

C = a * B
: This denotes repeated application of the group operation to B-- i.e. B + B-- (a - 1) times.

a = H(b)
: A cryptographic hash function that takes data b of indeterminate length and returns a fixed sized digest a.

a = F(B)
: A mapping function that takes an element and returns a scalar. For elliptic curve cryptography, F() returns the x-coordinate of the point B. For finite field cryptography, F() is the identity function.

a = KDF-b(c, d)
: A key derivation function that derives an output key a of length b from an input key c and context d.

a = HMAC(b, c)
: A keyed MAC function that produces a digest a using key b and text c.

a | b
: Concatentation of data a with data b.

{a}b[c]
: Authenticated-encryption of data (a), with a key (b), and associated data (c) that is authenticated but not encrypted.

2. Properties

Subversion of PKEX involves an adversary being able to insert its own public key into the exchange without the exchange failing, resulting in one of the parties to the exchange believing the adversary's public key actually belongs to the protocol peer.

PKEX has the following properties:

- An adversary is unable to subvert the exchange without knowing the password.
- An adversary is unable to discover the password through passive attack.
- The only information exposed by an active attack is whether a single guess of the password is correct or not.
- Proof-of-possession of the private key is provided.
- At the end of the protocol, either trust is established in the peer's public key and the public key is bound to the peer's identity, or the exchange fails.

3. Assumptions

Due to the nature of the exchange, only DSA ([DSS]) and ECDSA ([X9.62]) keys can be exchanged with PKEX.

PKEX requires fixed elements that are unique to the particular role in the protocol, an initiator-specific element and a responder-specific element. They need not be secret. It is assumed that both parties know the role-specific elements for the particular group in which their key pairs were derived. Techniques to generate role-specific elements, and generated elements for popular groups, are listed in Appendix A.1 and Appendix A.2.

The authenticated-encryption algorithm provides deterministic "key wrapping". To achieve this the AE scheme used in PKEX is AES-SIV as defined in [RFC5297].

The KDF provides for the generation of a cryptographically strong secret key from an "imperfect" source of randomness. To achieve this the KDF used in PKEX is the unsalted version of [RFC5869].

The keyed MAC function is HMAC per [RFC2104].

The following assumptions are made on PKEX:

- Only the peers involved in the exchange know the password.
- The peers’ public keys are from the same group.
- The discrete logarithms of the public role-specific elements are unknown, and determining them is computationally infeasible.

4. Cryptographic Primitives

HKDF and HMAC require an underlying hash function and AES-SIV requires a key length. To provide for consistent security the hash algorithm and key length depend on the group chosen to use with PKEX.

For ECC, the hash algorithm and key length depends on the size of the prime defining the curve, p:

- SHA-256 and 256 bits: when len(p) <= 256
- SHA-384 and 384 bits: when 256 < len(p) <= 384
- SHA-512 and 512 bits: when 384 < len(p)

For FFC, the hash algorithm depends on the prime, p, defining the finite field:

- SHA-256 and 256 bits: when len(p) <= 2048
- SHA-384 and 384 bits: when 2048 < len(p) <= 3072
- SHA-512 and 512 bits: when 3072 < len(p)

5. Protocol Definition

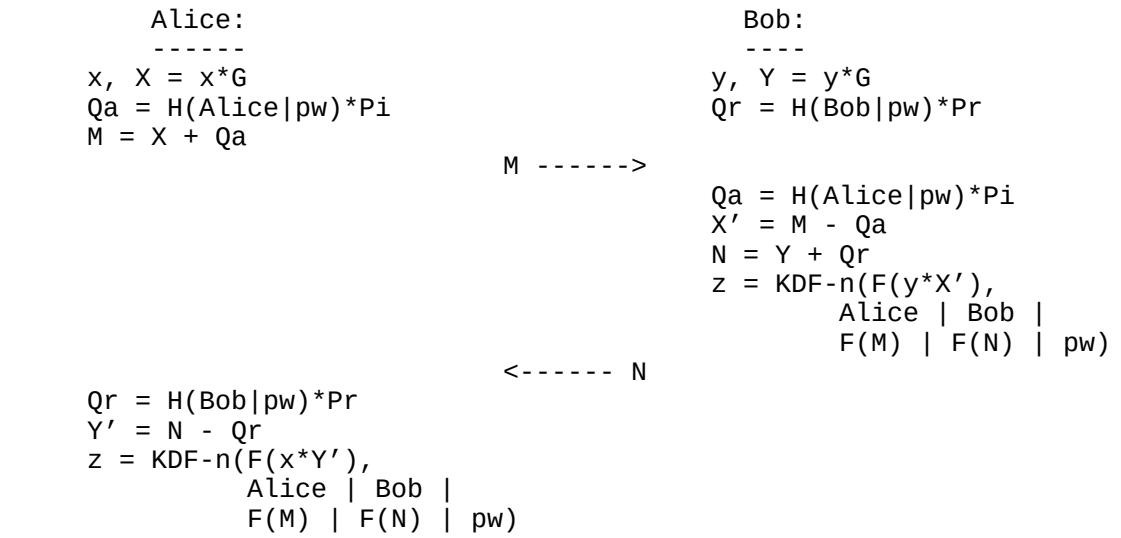
PKEX is a balanced PAKE. The identical version of the password is used by both parties.

PKEX consists of two phases: exchange and commit/reveal. It is described using the popular protocol participants, Alice (an initiator of PKEX), and Bob (a responder of PKEX).

We denote Alice’s role-specific element a Pi and Bob’s as Pr. The password is pw. For simplicity, Alice’s identity is "Alice" and Bob’s identity is "Bob". Alice’s public key she wants to share with Bob is A and her private key is a, while Bob’s public key he wants to share with Alice is B and his private key is b.

5.1. Exchange Phase

The Exchange phase is essentially the SPAKE2 key exchange. The peers derive ephemeral public keys, encrypt, and exchange them. Each party hashes a concatentation of his or her identity and the password and operates on the role-specific element to obtain a secret encrypting element. The group operation is then performed with the ephemeral key and the secret encrypting element to produce an encrypted ephmeral key.



Both M and N MUST be verified to be valid elements in the selected group. If either one is not valid the protocol fails.

At this point the peers have exchanged ephemeral elements that will be unknown except by someone with knowledge of the password. Given our assumptions that means only Alice and Bob can know the elements X and Y, and the secret key, z.

The secret encrypting elements Qa and Qb SHALL be irretrievably deleted at this point. The password MAY be irretrievably deleted at this time.

5.2. Commit/Reveal Phase

In the Commit/Reveal phase the peers commit to the particular public key they wish to exchange and reveal it to the peer. Proof-of-possession of the private key is accomplished by "signing" the public key, the identity to which the public key is bound, the receipient's ephemeral public key, and the sender's ephemeral public key.

The messages exchanged in the Commit/Reveal phase are encrypted and authenticated with AES-SIV using a key derived from the SPAKE2 key exchange in Section 5.1. Successful construction and validation of these messages authenticates the SPAKE2 exchange by proving possession of the SPAKE2 shared secret and therefore knowledge of the password. A single octet of the value zero (0) is used as associated data when encrypting Alice's message to Bob and a single octet of the value one (1) is used as associated data when constructing Bob's

response. The associated data is not transferred as part of the either message.

```
      Alice:                                Bob:
      ------                                ----
  u = HMAC(F(a*Y'), Alice | F(A) |
           F(Y') | F(X))

                  {A, u}z[0] ------>

                                  if (SIV-decrypt returns fail) fail
                                  if (A not valid element) fail
                                  u' = HMAC(F(y*A), Alice | F(A) |
                                            F(Y) | F(X'))
                                  if (u' != u) fail
                                  v = HMAC(F(b*X'), Bob | F(B) |
                                           F(X') | F(Y))

                         <------ {B, v}z[1]

  if (SIV-decrypt returns fail) fail
  if (B not valid element) fail
  v' = HMAC(F(x*B), Bob | F(B) |
            F(X) | F(Y))
  if (v'!= v) fail
```

where 0 and 1 are single octets of the value zero and one, respectively, n is the key length from Section 4, and both the KDF and HMAC use the hash algorithm from Section 4.

If the parties didn't fail they have each other's public key, knowledge that the peer possesses the corresponding private key, and trust that the public key belongs to the peer's identity that was authenticated in the Exchange Phase.

All ephemeral state created during the PKEX exchange SHALL be irretrievably deleted at this point.

6. IANA Considerations

This memo could create a registry of the fixed public elements for a nice cross section of popular groups. Or not. Once published this document will be a stable reference and a registry might not be needed.

7. Security Considerations

The encrypted shares exchanged in the Exchange phase MUST be ephemeral. Reuse of these keys, even with a different password, voids the security of the exchange.

If fixed elements other than those in Appendix A.1 and Appendix A.2 are used, their discrete logarithm MUST not be known. Knowledge of of the discrete logarithm of either of the fixed elements voids the security of the exchange.

The public keys exchanged in PKEX are never disclosed to an attacker, either passive or active. While they are, as the name implies, public, PKEX provides for secrecy of the exchanged keys for any protocol that might need such a capability.

PKEX has forward secrecy in the sense that exposure of the password used in a previous run of the protocol will not affect the security of that run. This also means that once PKEX has finished, the password can be exposed to a third party with out loss of security-- the public keys exchanged are still trusted and still bound to the entities that performed the exchange originally.

The Exchange Phase of PKEX is SPAKE2. The SPAKE2 security proof guarantees that if both sides bind the same password to each other’s identity they will derive the same secret. This means that the public key sent in the Commit/Reveal phase is guaranteed to be sent by the identified peer-- it is sent in a message that is integrity protected and encrypted by a key, z, derived from the SPAKE2 shared secret. This binds the peer’s public key to its authenticated identity. Proof-of-possession of the private key is provided by also sending a digest keyed by the result of a function of the private key and the peer’s ephemeral share from the Exchange Phase. Since the sender is not able to predict what random ephemeral share will be received in the Exchange Phase, it is unable to generate a keyed digest without knowing the private analog to the public key it is sending.

There is no proof of security of PKEX at this time.

8. References

8.1. Normative References

[DSS] U.S. Department of Commerce/National Institute of Standards and Technology, "Digital Signature Standard (DSS)", Federal Information Processing Standards FIPS PUB 186-4, July 2013.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <http://www.rfc-editor.org/info/rfc2104>.

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

[RFC3526] Kivinen, T. and M. Kojo, "More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)", RFC 3526, DOI 10.17487/RFC3526, May 2003, <http://www.rfc-editor.org/info/rfc3526>.

[RFC5297] Harkins, D., "Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)", RFC 5297, DOI 10.17487/RFC5297, October 2008, <http://www.rfc-editor.org/info/rfc5297>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <http://www.rfc-editor.org/info/rfc5869>.

[X9.62] American National Standards Institute, "X9.62-2005", Public Key Cryptography for the Financial Services Industry (ECDSA), 2005.

8.2. Informative References

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <http://www.rfc-editor.org/info/rfc7250>.

[RFC7670] Kivinen, T., Wouters, P., and H. Tschofenig, "Generic Raw Public-Key Support for IKEv2", RFC 7670, DOI 10.17487/RFC7670, January 2016, <http://www.rfc-editor.org/info/rfc7670>.

Appendix A. Role-specific Elements

Role-specific elements for six popular elliptic curves and four popular modp groups from [RFC3526] have been generated using the following technique.

A loop is performed to generate role-specific elements by generating a candidate, testing the candidate, and exiting the loop once the

test succeeds. A single octet counter is incremented each time through the loop (first time through the loop, the counter is one).

To find a candidate, a hash of an identifier (the concatenation of the ASN.1 of the OID of the curve or the name of the FFC group), a constant string, and the counter is produced. If the length of the hash's digest is less than the desired bits, the digest is pre-pended to the inputs and the result is fed back into the hash (this time it is a hash of a concatentation of the old digest, asn.1, constant string, counter) to produce the next length-of-digest bits. This is repeated until the number of bits has been produced. Excess octets are stripped off. The resulting string is interpreted as an integer with the first octet of (the first) hash being the high-order octet of the integer. For curves whose prime is not an integral number of octets, the bitstring is right-shifted, pre-pending with zero bits, in order to make a big-endian bitstring of the appropriate length. If that resulting number is larger than the prime defining the group the counter is incremented and the loop continues. Once an candidate has been produced it is checked to see whether it is a valid element in the group (for ECC it is treated as an x-coordinate and checked whether it produces a valid point on the curve, for FFC it is checked by seeing whether the exponentiation of the candidate to the power of the prime minus one divided by the order, modulo the prime, is one). If it is not, the counter is incremented and the whole loop is performed again. This process is repeated until an element is found. The hash algorithm used to generate candidates is determined by Section 4.

The loop is performed twice for each elliptic curve and FFC group to produce initiator- and responder-specific elements. The string passed for the initiator-specific element is "PKEX Initiator", the string passed for the responder-specific element is "PKEX Responder".

For FFC groups, the identifier is "group X" (including the space character and excluding the quotation marks) where X is the id assigned to the group, e.g. the 2048-bit group is named "group 14". For ECC groups, the identifier is the DER-encoded ASN.1 representation of the OID of the curve.

A.1. ECC Role-specific Elements

A.1.1. Role-specific Elements for NIST p256

```
unsigned char nist_p256_initiator_x_coord[32] = {
0x56, 0x26, 0x12, 0xcf, 0x36, 0x48, 0xfe, 0x0b,
0x07, 0x04, 0xbb, 0x12, 0x22, 0x50, 0xb2, 0x54,
0xb1, 0x94, 0x64, 0x7e, 0x54, 0xce, 0x08, 0x07,
0x2e, 0xec, 0xca, 0x74, 0x5b, 0x61, 0x2d, 0x25
};
unsigned char nist_p256_initiator_y_coord[32] = {
0x3e, 0x44, 0xc7, 0xc9, 0x8c, 0x1c, 0xa1, 0x0b,
0x20, 0x09, 0x93, 0xb2, 0xfd, 0xe5, 0x69, 0xdc,
0x75, 0xbc, 0xad, 0x33, 0xc1, 0xe7, 0xc6, 0x45,
0x4d, 0x10, 0x1e, 0x6a, 0x3d, 0x84, 0x3c, 0xa4
};

unsigned char nist_p256_responder_x_coord[32] = {
0x1e, 0xa4, 0x8a, 0xb1, 0xa4, 0xe8, 0x42, 0x39,
0xad, 0x73, 0x07, 0xf2, 0x34, 0xdf, 0x57, 0x4f,
0xc0, 0x9d, 0x54, 0xbe, 0x36, 0x1b, 0x31, 0x0f,
0x59, 0x91, 0x52, 0x33, 0xac, 0x19, 0x9d, 0x76
};
unsigned char nist_p256_responder_y_coord[32] = {
0x26, 0x04, 0x09, 0x45, 0x0a, 0x05, 0x20, 0xe7,
0xa7, 0x27, 0xc1, 0x36, 0x76, 0x85, 0xca, 0x3e,
0x42, 0x16, 0xf4, 0x89, 0x85, 0x34, 0x6e, 0xd5,
0x17, 0xde, 0xc0, 0xb8, 0xad, 0xfd, 0xb2, 0x98
};
```

A.1.2. Role-specific Elements for NIST p384

```
unsigned char nist_p384_initiator_x_coord[48] = {
0x95, 0x3f, 0x42, 0x9e, 0x50, 0x7f, 0xf9, 0xaa,
0xac, 0x1a, 0xf2, 0x85, 0x2e, 0x64, 0x91, 0x68,
0x64, 0xc4, 0x3c, 0xb7, 0x5c, 0xf8, 0xc9, 0x53,
0x6e, 0x58, 0x4c, 0x7f, 0xc4, 0x64, 0x61, 0xac,
0x51, 0x8a, 0x6f, 0xfe, 0xab, 0x74, 0xe6, 0x12,
0x81, 0xac, 0x38, 0x5d, 0x41, 0xe6, 0xb9, 0xa3
};
unsigned char nist_p384_initiator_y_coord[48] = {
0x89, 0xd0, 0x97, 0x7b, 0x59, 0x4f, 0xa6, 0xd6,
0x7c, 0x5d, 0x93, 0x5b, 0x93, 0xc4, 0x07, 0xa9,
0x89, 0xee, 0xd5, 0xcd, 0x6f, 0x42, 0xf8, 0x38,
0xc8, 0xc6, 0x62, 0x24, 0x69, 0x0c, 0xd4, 0x48,
0xd8, 0x44, 0xd6, 0xc2, 0xe8, 0xcc, 0x62, 0x6b,
0x3c, 0x25, 0x53, 0xba, 0x4f, 0x71, 0xf8, 0xe7
};

unsigned char nist_p384_responder_x_coord[48] = {
0xad, 0xbe, 0xd7, 0x1d, 0x3a, 0x71, 0x64, 0x98,
0x5f, 0xb4, 0xd6, 0x4b, 0x50, 0xd0, 0x84, 0x97,
0x4b, 0x7e, 0x57, 0x70, 0xd2, 0xd9, 0xf4, 0x92,
0x2a, 0x3f, 0xce, 0x99, 0xc5, 0x77, 0x33, 0x44,
0x14, 0x56, 0x92, 0xcb, 0xae, 0x46, 0x64, 0xdf,
0xe0, 0xbb, 0xd7, 0xb1, 0x29, 0x20, 0x72, 0xdf
};
unsigned char nist_p384_responder_y_coord[48] = {
0x54, 0x58, 0x20, 0xad, 0x55, 0x1d, 0xca, 0xf3,
0x1c, 0x8a, 0xcd, 0x19, 0x40, 0xf9, 0x37, 0x83,
0xc7, 0xd6, 0xb3, 0x13, 0x7d, 0x53, 0x28, 0x5c,
0xf6, 0x2d, 0xf1, 0xdd, 0xa5, 0x8b, 0xad, 0x5d,
0x81, 0xab, 0xb1, 0x00, 0x39, 0xd6, 0xcc, 0x9c,
0xea, 0x1e, 0x84, 0x1d, 0xbf, 0xe3, 0x35, 0xf9
};
```

A.1.3. Role-specific Elements for NIST p521

```
unsigned char nist_p521_initiator_x_coord[66] = {
0x00, 0x16, 0x20, 0x45, 0x19, 0x50, 0x95, 0x23,
0x0d, 0x24, 0xbe, 0x00, 0x87, 0xdc, 0xfa, 0xf0,
0x58, 0x9a, 0x01, 0x60, 0x07, 0x7a, 0xca, 0x76,
0x01, 0xab, 0x2d, 0x5a, 0x46, 0xcd, 0x2c, 0xb5,
0x11, 0x9a, 0xff, 0xaa, 0x48, 0x04, 0x91, 0x38,
0xcf, 0x86, 0xfc, 0xa4, 0xa5, 0x0f, 0x47, 0x01,
0x80, 0x1b, 0x30, 0xa3, 0xae, 0xe8, 0x1c, 0x2e,
0xea, 0xcc, 0xf0, 0x03, 0x9f, 0x77, 0x4c, 0x8d,
0x97, 0x76
};
unsigned char nist_p521_initiator_y_coord[66] = {
0x01, 0x4c, 0x71, 0xfd, 0x1b, 0xd5, 0x9c, 0xa6,
0xed, 0x39, 0xef, 0x45, 0xc5, 0x06, 0xfd, 0x66,
0xc0, 0xeb, 0x0f, 0xbf, 0x21, 0xa3, 0x36, 0x74,
0xfd, 0xaa, 0x05, 0x6e, 0x4e, 0x33, 0x95, 0x42,
0x1a, 0x9d, 0x3f, 0x3a, 0x1c, 0x5e, 0xa8, 0x60,
0xf7, 0xe5, 0x59, 0x1d, 0x07, 0xaa, 0x6f, 0x40,
0x0a, 0x59, 0x3c, 0x27, 0xad, 0xe0, 0x48, 0xfd,
0xd1, 0x83, 0x37, 0x4c, 0xdf, 0xe1, 0x86, 0x72,
0xfc, 0x57
};

unsigned char nist_p521_responder_x_coord[66] = {
0x00, 0x79, 0xe4, 0x4d, 0x6b, 0x5e, 0x12, 0x0a,
0x18, 0x2c, 0xb3, 0x05, 0x77, 0x0f, 0xc3, 0x44,
0x1a, 0xcd, 0x78, 0x46, 0x14, 0xee, 0x46, 0x3f,
0xab, 0xc9, 0x59, 0x7c, 0x85, 0xa0, 0xc2, 0xfb,
0x02, 0x32, 0x99, 0xde, 0x5d, 0xe1, 0x0d, 0x48,
0x2d, 0x71, 0x7d, 0x8d, 0x3f, 0x61, 0x67, 0x9e,
0x2b, 0x8b, 0x12, 0xde, 0x10, 0x21, 0x55, 0x0a,
0x5b, 0x2d, 0xe8, 0x05, 0x09, 0xf6, 0x20, 0x97,
0x84, 0xb4
};
unsigned char nist_p521_responder_y_coord[66] = {
0x01, 0xb9, 0x9c, 0xc6, 0x41, 0x32, 0x5b, 0xd2,
0x35, 0xd8, 0x8b, 0x2b, 0xe4, 0x6e, 0xcc, 0xdf,
0x7c, 0x38, 0xc4, 0x5b, 0xf6, 0x74, 0x71, 0x5c,
0x77, 0x16, 0x8a, 0x80, 0xa9, 0x84, 0xc7, 0x7b,
0x9d, 0xfd, 0x83, 0x6f, 0xae, 0xf8, 0x24, 0x16,
0x2f, 0x21, 0x25, 0x65, 0xa2, 0x1a, 0x6b, 0x2d,
0x30, 0x62, 0xb3, 0xcc, 0x6e, 0x59, 0x3c, 0x7f,
0x58, 0x91, 0x81, 0x72, 0x07, 0x8c, 0x91, 0xac,
0x31, 0x1e
};
```

A.1.4. Role-specific Elements for brainpool p256r1

```
unsigned char brainpool_p256r1_initiator_x_coord[32] = {
0x46, 0x98, 0x18, 0x6c, 0x27, 0xcd, 0x4b, 0x10,
0x7d, 0x55, 0xa3, 0xdd, 0x89, 0x1f, 0x9f, 0xca,
0xc7, 0x42, 0x5b, 0x8a, 0x23, 0xed, 0xf8, 0x75,
0xac, 0xc7, 0xe9, 0x8d, 0xc2, 0x6f, 0xec, 0xd8
};
unsigned char brainpool_p256r1_initiator_y_coord[32] = {
0x16, 0x30, 0x68, 0x32, 0x3b, 0xb0, 0x21, 0xee,
0xeb, 0xf7, 0xb6, 0x7c, 0xae, 0x52, 0x26, 0x42,
0x59, 0x28, 0x58, 0xb6, 0x14, 0x90, 0xed, 0x69,
0xd0, 0x67, 0xea, 0x25, 0x60, 0x0f, 0xa9, 0x6c
};

unsigned char brainpool_p256r1_responder_x_coord[32] = {
0x90, 0x18, 0x84, 0xc9, 0xdc, 0xcc, 0xb5, 0x2f,
0x4a, 0x3f, 0x4f, 0x18, 0x0a, 0x22, 0x56, 0x6a,
0xa9, 0xef, 0xd4, 0xe6, 0xc3, 0x53, 0xc2, 0x1a,
0x23, 0x54, 0xdd, 0x08, 0x7e, 0x10, 0xd8, 0xe3
};
unsigned char brainpool_p256r1_responder_y_coord[32] = {
0x2a, 0xfa, 0x98, 0x9b, 0xe3, 0xda, 0x30, 0xfd,
0x32, 0x28, 0xcb, 0x66, 0xfb, 0x40, 0x7f, 0xf2,
0xb2, 0x25, 0x80, 0x82, 0x44, 0x85, 0x13, 0x7e,
0x4b, 0xb5, 0x06, 0xc0, 0x03, 0x69, 0x23, 0x64
};
```

A.1.5. Role-specific Elements for brainpool p384r1

```
unsigned char brainpool_p384r1_initiator_x_coord[48] = {
0x0a, 0x2c, 0xeb, 0x49, 0x5e, 0xb7, 0x23, 0xbd,
0x20, 0x5b, 0xe0, 0x49, 0xdf, 0xcf, 0xcf, 0x19,
0x37, 0x36, 0xe1, 0x2f, 0x59, 0xdb, 0x07, 0x06,
0xb5, 0xeb, 0x2d, 0xae, 0xc2, 0xb2, 0x38, 0x62,
0xa6, 0x73, 0x09, 0xa0, 0x6c, 0x0a, 0xa2, 0x30,
0x99, 0xeb, 0xf7, 0x1e, 0x47, 0xb9, 0x5e, 0xbe
};
unsigned char brainpool_p384r1_initiator_y_coord[48] = {
0x54, 0x76, 0x61, 0x65, 0x75, 0x5a, 0x2f, 0x99,
0x39, 0x73, 0xca, 0x6c, 0xf9, 0xf7, 0x12, 0x86,
0x54, 0xd5, 0xd4, 0xad, 0x45, 0x7b, 0xbf, 0x32,
0xee, 0x62, 0x8b, 0x9f, 0x52, 0xe8, 0xa0, 0xc9,
0xb7, 0x9d, 0xd1, 0x09, 0xb4, 0x79, 0x1c, 0x3e,
0x1a, 0xbf, 0x21, 0x45, 0x66, 0x6b, 0x02, 0x52
};

unsigned char brainpool_p384r1_responder_x_coord[48] = {
0x03, 0xa2, 0x57, 0xef, 0xe8, 0x51, 0x21, 0xa0,
0xc8, 0x9e, 0x21, 0x02, 0xb5, 0x9a, 0x36, 0x25,
0x74, 0x22, 0xd1, 0xf2, 0x1b, 0xa8, 0x9a, 0x9b,
0x97, 0xbc, 0x5a, 0xeb, 0x26, 0x15, 0x09, 0x71,
0x77, 0x59, 0xec, 0x8b, 0xb7, 0xe1, 0xe8, 0xce,
0x65, 0xb8, 0xaf, 0xf8, 0x80, 0xae, 0x74, 0x6c
};
unsigned char brainpool_p384r1_responder_y_coord[48] = {
0x2f, 0xd9, 0x6a, 0xc7, 0x3e, 0xec, 0x76, 0x65,
0x2d, 0x38, 0x7f, 0xec, 0x63, 0x26, 0x3f, 0x04,
0xd8, 0x4e, 0xff, 0xe1, 0x0a, 0x51, 0x74, 0x70,
0xe5, 0x46, 0x63, 0x7f, 0x5c, 0xc0, 0xd1, 0x7c,
0xfb, 0x2f, 0xea, 0xe2, 0xd8, 0x0f, 0x84, 0xcb,
0xe9, 0x39, 0x5c, 0x64, 0xfe, 0xcb, 0x2f, 0xf1
};
```

A.1.6. Role-specific Elements for brainpool p512r1

```
unsigned char brainpool_p512r1_initiator_x_coord[64] = {
0x4c, 0xe9, 0xb6, 0x1c, 0xe2, 0x00, 0x3c, 0x9c,
0xa9, 0xc8, 0x56, 0x52, 0xaf, 0x87, 0x3e, 0x51,
0x9c, 0xbb, 0x15, 0x31, 0x1e, 0xc1, 0x05, 0xfc,
0x7c, 0x77, 0xd7, 0x37, 0x61, 0x27, 0xd0, 0x95,
0x98, 0xee, 0x5d, 0xa4, 0x3d, 0x09, 0xdb, 0x3d,
0xfa, 0x89, 0x9e, 0x7f, 0xa6, 0xa6, 0x9c, 0xff,
0x83, 0x5c, 0x21, 0x6c, 0x3e, 0xf2, 0xfe, 0xdc,
0x63, 0xe4, 0xd1, 0x0e, 0x75, 0x45, 0x69, 0x0f
};
unsigned char brainpool_p512r1_initiator_y_coord[64] = {
0x5a, 0x28, 0x01, 0xbe, 0x96, 0x82, 0x4e, 0xf6,
0xfa, 0xed, 0x7d, 0xfd, 0x48, 0x8b, 0x48, 0x4e,
0xd1, 0x97, 0x87, 0xc4, 0x05, 0x5d, 0x15, 0x2a,
0xf4, 0x91, 0x4b, 0x75, 0x90, 0xd9, 0x34, 0x2c,
0x3c, 0x12, 0xf2, 0xf5, 0x25, 0x94, 0x24, 0x34,
0xa7, 0x6d, 0x66, 0xbc, 0x27, 0xa4, 0xa0, 0x8d,
0xd5, 0xe1, 0x54, 0xa3, 0x55, 0x26, 0xd4, 0x14,
0x17, 0x0f, 0xc1, 0xc7, 0x3d, 0x68, 0x7f, 0x5a
};

unsigned char brainpool_p512r1_responder_x_coord[64] = {
0x2a, 0x60, 0x32, 0x27, 0xa1, 0xe6, 0x94, 0x72,
0x1c, 0x48, 0xbe, 0xc5, 0x77, 0x14, 0x30, 0x76,
0xe4, 0xbf, 0xf7, 0x7b, 0xc5, 0xfd, 0xdf, 0x19,
0x1e, 0x0f, 0xdf, 0x1c, 0x40, 0xfa, 0x34, 0x9e,
0x1f, 0x42, 0x24, 0xa3, 0x2c, 0xd5, 0xc7, 0xc9,
0x7b, 0x47, 0x78, 0x96, 0xf1, 0x37, 0x0e, 0x88,
0xcb, 0xa6, 0x52, 0x29, 0xd7, 0xa8, 0x38, 0x29,
0x8e, 0x6e, 0x23, 0x47, 0xd4, 0x4b, 0x70, 0x3e
};
unsigned char brainpool_p512r1_responder_y_coord[64] = {
0x2a, 0xbe, 0x59, 0xe6, 0xc4, 0xb3, 0xd8, 0x09,
0x66, 0x89, 0x0a, 0x2d, 0x19, 0xf0, 0x9c, 0x9f,
0xb4, 0xab, 0x8f, 0x50, 0x68, 0x3c, 0x74, 0x64,
0x4e, 0x19, 0x55, 0x81, 0x9b, 0x48, 0x5c, 0xf4,
0x12, 0x8d, 0xb9, 0xd8, 0x02, 0x5b, 0xe1, 0x26,
0x7e, 0x19, 0x5c, 0xfd, 0x70, 0xf7, 0x4b, 0xdc,
0xb5, 0x5d, 0xc1, 0x7a, 0xe9, 0xd1, 0x05, 0x2e,
0xd1, 0xfd, 0x2f, 0xce, 0x63, 0x77, 0x48, 0x2c
};
```

A.2. FFC Role-specific Elements

A.2.1. Role-specific Elements for 2048-bit FFC group

```
unsigned char group14_initiator[256] = {
0x97, 0x15, 0x52, 0x05, 0x89, 0xdf, 0xeb, 0x3d,
0xd6, 0x50, 0xe1, 0x96, 0xd4, 0x30, 0x81, 0x04,
0x3a, 0x4d, 0x6c, 0xae, 0xe5, 0x81, 0xb6, 0x1d,
0x53, 0xca, 0x65, 0xfa, 0x19, 0x59, 0xfd, 0xe4,
0xa4, 0xf0, 0x7b, 0xc5, 0xc7, 0xc4, 0xa9, 0xd6,
0xdf, 0x79, 0x54, 0x04, 0x5d, 0x64, 0xdc, 0x3c,
0xec, 0x0a, 0xa0, 0xd3, 0x2f, 0xef, 0xf3, 0xf5,
0x2c, 0x51, 0xe6, 0x6d, 0x1c, 0xdc, 0xac, 0x09,
0x3c, 0x00, 0x62, 0x41, 0xe7, 0x0b, 0x0d, 0xba,
0x1b, 0xf2, 0xb9, 0x22, 0xe9, 0x11, 0xea, 0xc7,
0xb1, 0xb2, 0x15, 0xc5, 0x19, 0x46, 0x2e, 0x15,
0x88, 0x41, 0xe0, 0x24, 0x16, 0x13, 0xc1, 0x0e,
0x27, 0xd3, 0x5f, 0x71, 0x12, 0xc6, 0x6f, 0x75,
0x1e, 0xe8, 0xa4, 0xaa, 0x57, 0xa3, 0x22, 0x32,
0xdc, 0xd4, 0xe3, 0xb5, 0xa3, 0xd0, 0x87, 0xb1,
0x3d, 0x1c, 0x2a, 0xcf, 0xe5, 0x87, 0x1f, 0xea,
0x98, 0xdd, 0xd6, 0x8f, 0xce, 0x0b, 0xdd, 0xba,
0xae, 0x85, 0x81, 0xd6, 0x89, 0x27, 0x71, 0xc9,
0x9f, 0xf9, 0xea, 0x5a, 0x89, 0xa6, 0xaa, 0x0a,
0xc2, 0x76, 0xd6, 0x6d, 0x89, 0xd3, 0xd2, 0x4c,
0xc0, 0xad, 0xb0, 0xf6, 0x4d, 0x2b, 0x7c, 0xbf,
0xd2, 0x4e, 0xe3, 0x2b, 0x4f, 0xd7, 0xc5, 0x3a,
0x4b, 0x1c, 0xc0, 0x17, 0xbe, 0x1e, 0x7b, 0x81,
0x1c, 0x77, 0xd4, 0xc2, 0xf6, 0xca, 0xb8, 0x51,
0xa8, 0x94, 0xe9, 0xe5, 0xe9, 0xa5, 0x46, 0x60,
0xb5, 0x36, 0x10, 0xcf, 0xc8, 0x2a, 0xe7, 0x7f,
0x94, 0x49, 0x96, 0xeb, 0xff, 0x7b, 0x62, 0xa9,
0x95, 0x35, 0x5c, 0xad, 0xcd, 0x52, 0x06, 0xa1,
0x9c, 0xa1, 0xb7, 0xe0, 0xcd, 0xd3, 0x13, 0x43,
0xee, 0xe7, 0xf1, 0xb1, 0x4d, 0x76, 0x51, 0x53,
0x24, 0x4f, 0xb4, 0xbd, 0x8b, 0x0c, 0x73, 0xd1,
0x6f, 0xf5, 0x27, 0x3f, 0xb9, 0x60, 0xa6, 0x17
}

unsigned char group14_responder[256] = {
0x3c, 0x86, 0x96, 0x8a, 0xb0, 0x4b, 0x14, 0xdd,
0x27, 0xf0, 0x6f, 0x51, 0xa2, 0xb2, 0xbd, 0x7d,
0x27, 0x34, 0xf2, 0x21, 0x3c, 0x6a, 0x63, 0xcf,
0x15, 0xd8, 0xeb, 0x21, 0xc2, 0x2d, 0xe5, 0x75,
0xbb, 0x7e, 0x09, 0x2e, 0xba, 0xa2, 0xd7, 0x04,
0x7c, 0x81, 0x82, 0x07, 0x09, 0x15, 0x28, 0x35,
0xf7, 0x0a, 0xd6, 0xa9, 0xaf, 0x0d, 0xb1, 0x97,
0x71, 0x76, 0x74, 0x66, 0xa3, 0x54, 0x9e, 0xd9,
0xa9, 0x13, 0xa8, 0xcf, 0xec, 0xce, 0x60, 0xd8,
0xea, 0x18, 0x67, 0xb7, 0x15, 0x17, 0xf7, 0xe6,
```

```
0x23, 0xc8, 0x30, 0x02, 0xb9, 0x9e, 0x0e, 0xe6,
0x64, 0x3e, 0x1a, 0x61, 0xb9, 0xbf, 0xd8, 0x7a,
0x0e, 0xbe, 0x1e, 0x58, 0xdc, 0xba, 0x9e, 0x31,
0xab, 0xcc, 0x6b, 0x03, 0xcc, 0x08, 0xf8, 0xa2,
0xcb, 0x9d, 0xd5, 0x1a, 0xb8, 0x6a, 0x1f, 0x4b,
0xab, 0xe8, 0x2a, 0x0c, 0x44, 0xde, 0x2a, 0xfd,
0x0f, 0x6d, 0x2f, 0xe4, 0xc4, 0x41, 0x61, 0xed,
0x4a, 0x85, 0x2a, 0x06, 0x9d, 0x3a, 0x27, 0xf0,
0x30, 0x6e, 0xf8, 0xb1, 0xc8, 0xde, 0x1f, 0xe0,
0xfb, 0xb6, 0xd0, 0x51, 0xee, 0x7d, 0x3a, 0x05,
0x0c, 0xbb, 0xa0, 0x41, 0xc5, 0x5d, 0x25, 0xb4,
0x48, 0xd6, 0x4f, 0x08, 0x85, 0x96, 0xa3, 0xa6,
0xf5, 0x1b, 0xa1, 0xb3, 0x13, 0x50, 0x06, 0xb2,
0xef, 0xf8, 0x2f, 0xe8, 0x7b, 0xe0, 0x5a, 0xe1,
0x42, 0x16, 0xfc, 0xdf, 0xad, 0x53, 0x95, 0x43,
0xb0, 0x73, 0x33, 0xa9, 0x08, 0x29, 0xcd, 0x6c,
0x14, 0x98, 0x5e, 0x98, 0xe6, 0xca, 0x92, 0x55,
0xd9, 0x3e, 0xc3, 0x51, 0x83, 0xda, 0x1e, 0x6d,
0x16, 0x88, 0x3f, 0xd1, 0xa5, 0xd1, 0xc5, 0x43,
0xcf, 0x8a, 0xd7, 0x29, 0xaa, 0xa6, 0x4f, 0x6b,
0x4f, 0xee, 0x36, 0x65, 0xd6, 0x71, 0xef, 0x71,
0xc5, 0x5b, 0x7c, 0x6d, 0x90, 0x9d, 0xf5, 0x74
}
```

A.2.2. Role-specific Elements for 3072-bit FFC group

```
unsigned char group15_initiator[384] = {
0x12, 0xce, 0x47, 0xcf, 0xa9, 0xc4, 0xfc, 0x5e,
0x99, 0xf6, 0xd2, 0x43, 0x5c, 0x60, 0x39, 0xb0,
0x06, 0xe9, 0x4a, 0xec, 0x21, 0x60, 0x9f, 0x5a,
0x25, 0xb1, 0x22, 0xf2, 0x53, 0x2b, 0x44, 0xe3,
0x6c, 0xfb, 0x9c, 0x46, 0x3c, 0x8f, 0x88, 0xaa,
0x60, 0xfd, 0x3a, 0x51, 0xf1, 0x19, 0x8d, 0x88,
0xee, 0xa4, 0xc2, 0x21, 0x3c, 0xbb, 0xc5, 0x53,
0x12, 0x16, 0xfb, 0xd3, 0xa9, 0x4d, 0x85, 0x5d,
0x17, 0x9d, 0x92, 0x15, 0x30, 0xc7, 0x97, 0x0c,
0x68, 0x62, 0x91, 0xff, 0xce, 0x81, 0x97, 0x25,
0x54, 0x94, 0x0e, 0x3a, 0x14, 0x36, 0x4e, 0xc2,
0xda, 0xc2, 0xaa, 0xa3, 0x58, 0x49, 0xca, 0xa4,
0xa4, 0x0b, 0x2a, 0x26, 0x35, 0x0d, 0x72, 0x4f,
0x10, 0x3c, 0x5f, 0x4d, 0xbc, 0x7c, 0x09, 0xcb,
0xef, 0x99, 0xdd, 0x73, 0x1c, 0x23, 0x69, 0xa7,
0xc9, 0xc4, 0x1a, 0x4c, 0x7c, 0xf2, 0xca, 0x48,
0x15, 0xf4, 0xd6, 0x30, 0x25, 0x44, 0x9f, 0xcd,
0xc0, 0x23, 0x72, 0x4a, 0x4f, 0x83, 0x3c, 0xba,
0x88, 0x1c, 0x5a, 0xcc, 0x3f, 0xf6, 0x5e, 0x68,
0x5a, 0x38, 0x10, 0xa1, 0xd2, 0x99, 0x5f, 0x4d,
0x48, 0xec, 0xb1, 0x2f, 0x9a, 0x08, 0xcf, 0x59,
```

```
0x0f, 0xeb, 0x35, 0x0b, 0xf5, 0xab, 0x6e, 0xc9,
0x69, 0x44, 0xbd, 0x9a, 0x62, 0x3b, 0x53, 0x4e,
0x59, 0xc9, 0x38, 0x1d, 0x9e, 0x61, 0x5b, 0xdb,
0x72, 0x4e, 0xb9, 0x35, 0xb5, 0xc3, 0x9f, 0x47,
0x4f, 0x70, 0xfa, 0xff, 0x95, 0x45, 0xf9, 0x4a,
0xf4, 0xc3, 0xcc, 0x8e, 0xf4, 0x89, 0x0b, 0x73,
0x08, 0x97, 0x0d, 0x22, 0xe2, 0x97, 0xc8, 0xf8,
0x45, 0x83, 0x8f, 0xea, 0x68, 0x4b, 0xe0, 0xed,
0x71, 0xdb, 0x73, 0x62, 0x57, 0xab, 0x03, 0x69,
0x93, 0x66, 0x0e, 0xc1, 0x29, 0x2d, 0x01, 0x7c,
0x7d, 0x50, 0x14, 0x03, 0x9f, 0xdb, 0x5c, 0x4c,
0xf4, 0xdf, 0xa4, 0x9c, 0xda, 0x80, 0xd9, 0xbe,
0x0d, 0xdc, 0xb8, 0x0b, 0xc2, 0x19, 0x28, 0xaf,
0xf0, 0x98, 0x1a, 0xec, 0x26, 0xf0, 0x15, 0x1b,
0xa1, 0xa1, 0x11, 0x8f, 0x9f, 0x5a, 0x1e, 0x8a,
0x8e, 0x57, 0x84, 0x60, 0xc5, 0xda, 0xa2, 0x74,
0x3d, 0xe8, 0xc0, 0x08, 0x0f, 0x7e, 0xdd, 0x11,
0xd6, 0xbf, 0x5b, 0x2e, 0xde, 0x81, 0x1a, 0xfd,
0x33, 0x9c, 0x07, 0xcc, 0x1d, 0x0f, 0x63, 0xc8,
0x3e, 0x1d, 0xbb, 0x16, 0x5e, 0x70, 0x4c, 0x82,
0x8e, 0x72, 0xb6, 0x35, 0x69, 0xc6, 0xe4, 0xa7,
0xae, 0x6e, 0xa2, 0x23, 0xe8, 0x86, 0x99, 0x3a,
0x0b, 0x64, 0xec, 0xe2, 0xdb, 0xb2, 0xaa, 0xc4,
0x59, 0xe1, 0x23, 0x3d, 0xa5, 0x46, 0x92, 0x8f,
0x04, 0x34, 0x5f, 0x7a, 0x13, 0x55, 0x75, 0xd5,
0x6d, 0x0f, 0x5a, 0xc2, 0x0d, 0x16, 0xf9, 0xc0,
0xf3, 0xac, 0x0a, 0xa8, 0x62, 0x20, 0x09, 0x4e
}

unsigned char group15_responder[384] = {
0x9f, 0x17, 0xe0, 0xf9, 0x3d, 0x23, 0x36, 0x6e,
0x7d, 0xa6, 0x34, 0x75, 0xb7, 0xb4, 0x22, 0xb1,
0x87, 0x7a, 0x00, 0x4e, 0x02, 0x14, 0x4e, 0xe6,
0x96, 0xd4, 0x2c, 0x61, 0x00, 0x97, 0x7d, 0x99,
0xad, 0x18, 0x1c, 0xc4, 0x1b, 0xed, 0x6f, 0xd3,
0x9f, 0x85, 0xef, 0xfd, 0x1e, 0xcd, 0x13, 0xa0,
0x61, 0x2f, 0xf8, 0xa7, 0x11, 0xab, 0x83, 0xfc,
0xae, 0xad, 0xb6, 0xed, 0x6b, 0x7f, 0x34, 0x81,
0x30, 0xa2, 0x1c, 0x38, 0xb3, 0x31, 0x7b, 0x74,
0xc1, 0x0f, 0xf4, 0x29, 0x4b, 0xdd, 0x2b, 0x09,
0x32, 0xb7, 0x8f, 0x84, 0xab, 0x89, 0x9d, 0x64,
0xea, 0xec, 0x00, 0xa9, 0x0c, 0x82, 0x1c, 0x35,
0x75, 0x3b, 0x7f, 0x35, 0x28, 0xb8, 0xcc, 0xb1,
0x62, 0xc2, 0xd0, 0x74, 0x83, 0xb6, 0xf7, 0x76,
0x10, 0x2a, 0x7e, 0xce, 0xd0, 0x0a, 0x68, 0x46,
0xac, 0x78, 0x26, 0x90, 0xc7, 0x0e, 0xaa, 0x21,
0x59, 0xb1, 0x8d, 0x8e, 0xc8, 0xfb, 0x5c, 0x60,
0xec, 0x53, 0x9c, 0x40, 0xd9, 0x42, 0xab, 0xa3,
```

```
0xd8, 0x45, 0x81, 0x04, 0x6c, 0x13, 0x79, 0x66,
0x51, 0x1f, 0xa4, 0x81, 0x38, 0x0d, 0x48, 0x06,
0xef, 0x25, 0x89, 0x26, 0x5d, 0x09, 0x0e, 0xbe,
0xba, 0xe2, 0xca, 0x2c, 0xa6, 0x3e, 0x36, 0xd7,
0xef, 0x46, 0xe3, 0x8a, 0x1d, 0x85, 0x59, 0xc4,
0x89, 0x5e, 0x36, 0xea, 0xb2, 0x44, 0x79, 0xc8,
0x91, 0x80, 0x2c, 0x89, 0xfc, 0x55, 0x81, 0x62,
0x40, 0x82, 0x1a, 0x66, 0xf1, 0x1c, 0x10, 0xf0,
0x34, 0xdd, 0x52, 0x9f, 0xff, 0x63, 0x62, 0xe2,
0xec, 0x68, 0x75, 0xa0, 0x2e, 0x72, 0x44, 0xf3,
0x66, 0x33, 0x2a, 0x65, 0x61, 0x79, 0x91, 0x13,
0x4f, 0x73, 0x8f, 0x38, 0xef, 0xa6, 0x65, 0x88,
0xf9, 0x03, 0x35, 0x57, 0xed, 0xb9, 0x05, 0x7f,
0xe8, 0xfb, 0x90, 0xac, 0x19, 0x2e, 0xff, 0x9c,
0xaf, 0x76, 0x5f, 0x40, 0x74, 0x49, 0x11, 0xee,
0x18, 0xb5, 0x6e, 0xa4, 0x91, 0xbc, 0x42, 0x1c,
0x0e, 0x2e, 0x0d, 0x6f, 0xc3, 0x6a, 0x7b, 0x8b,
0xf4, 0x1a, 0x30, 0x52, 0x54, 0x99, 0xae, 0x8a,
0x8c, 0x33, 0x5e, 0x5c, 0xa8, 0xc2, 0x49, 0xf3,
0xbd, 0x0e, 0x85, 0x22, 0x9b, 0x5d, 0x92, 0xbc,
0x42, 0x8b, 0x60, 0x38, 0xd7, 0x14, 0x24, 0xaa,
0x36, 0xc7, 0x8f, 0xd7, 0xc2, 0x14, 0x20, 0x72,
0x0a, 0xba, 0x28, 0x95, 0x65, 0x53, 0x30, 0x0f,
0xc8, 0x17, 0xc2, 0x02, 0x58, 0x08, 0x7b, 0x93,
0x36, 0xcc, 0x80, 0x19, 0x9c, 0x1f, 0xad, 0x1f,
0x8e, 0x8e, 0x2e, 0x3a, 0xbf, 0x0f, 0xab, 0x76,
0x7a, 0xac, 0xce, 0x1a, 0x57, 0xe6, 0x7c, 0x64,
0x93, 0x5f, 0x92, 0x5d, 0xbe, 0xe2, 0x11, 0xf6,
0x58, 0x90, 0xd8, 0x87, 0xe4, 0x17, 0x8b, 0x61,
0xf6, 0x11, 0xe2, 0x0a, 0x99, 0xe8, 0x55, 0xcc
}
```

A.2.3. Role-specific Elements for 4096-bit FFC group

```
unsigned char group16_initiator[512] = {
0x79, 0x8f, 0xfe, 0xed, 0x53, 0x08, 0x41, 0x73,
0xba, 0x89, 0x66, 0x8d, 0xf0, 0x18, 0xee, 0xe0,
0x76, 0xda, 0x5f, 0xf8, 0x55, 0x36, 0x53, 0x71,
0xd3, 0xfd, 0xf5, 0x30, 0xd9, 0xa0, 0xd2, 0x30,
0x2c, 0x16, 0x38, 0x0a, 0x2b, 0x91, 0x6e, 0x02,
0xc9, 0x27, 0x52, 0xf2, 0x51, 0x0e, 0xe3, 0x1f,
0xbb, 0x2b, 0x8b, 0xad, 0xa8, 0xc3, 0xf4, 0xc0,
0xba, 0x45, 0xf5, 0xf7, 0x4c, 0x91, 0x6f, 0x86,
0x9a, 0xb8, 0xb1, 0xea, 0x7d, 0x89, 0x91, 0x39,
0xd0, 0xb0, 0x95, 0x98, 0xf1, 0xa9, 0x03, 0x8d,
0xc5, 0x7a, 0x75, 0x36, 0x7a, 0xf8, 0x0b, 0xf6,
0xbf, 0x5c, 0x33, 0xde, 0x7f, 0xed, 0xec, 0x1a,
0x1e, 0xbc, 0x54, 0xf1, 0x5f, 0x5c, 0xfa, 0x2f,
```

```
   0x98, 0x85, 0xe0, 0x6f, 0xb8, 0x2b, 0x16, 0xe9,
   0x48, 0x6e, 0xe4, 0xb3, 0x3f, 0x20, 0x66, 0x3b,
   0x3d, 0xcf, 0x62, 0xc6, 0xed, 0xe5, 0x2e, 0x7a,
   0xac, 0x0b, 0x15, 0x6d, 0x15, 0x4e, 0xcb, 0x23,
   0xcd, 0x5a, 0xed, 0x51, 0xf4, 0xbe, 0x52, 0x6b,
   0x55, 0x30, 0xe3, 0x57, 0x91, 0x1f, 0xf3, 0xbd,
   0x07, 0xd9, 0x8b, 0x32, 0x9d, 0xfb, 0x99, 0xbb,
   0x17, 0x81, 0x24, 0xd1, 0x82, 0x10, 0xce, 0x34,
   0x77, 0x4f, 0xbc, 0x4d, 0xe6, 0x23, 0xdf, 0x2c,
   0x24, 0x8d, 0xf5, 0xf5, 0xf9, 0x93, 0x3d, 0x08,
   0x55, 0x31, 0xc8, 0xe5, 0xf3, 0x5b, 0x4c, 0xe2,
   0x4a, 0xdf, 0x88, 0x83, 0xc7, 0x84, 0x1b, 0xfa,
   0x99, 0x72, 0x1b, 0x13, 0x9e, 0xf6, 0x76, 0xca,
   0xa9, 0xa2, 0x7f, 0xd8, 0xc0, 0x9a, 0x46, 0x15,
   0x0b, 0x20, 0x4e, 0x53, 0x09, 0xe5, 0x16, 0x67,
   0xaf, 0xe5, 0x07, 0xe9, 0x57, 0x2f, 0xdc, 0x38,
   0xde, 0x33, 0x19, 0x49, 0x08, 0x05, 0x6f, 0xb9,
   0xb0, 0xce, 0x97, 0xab, 0xb5, 0xac, 0x88, 0x4e,
   0x7a, 0xbe, 0xd0, 0xaa, 0x74, 0x67, 0x73, 0xe3,
   0xd5, 0x04, 0x18, 0x41, 0x51, 0xcd, 0xf5, 0x59,
   0x85, 0xb6, 0x4b, 0x63, 0x1f, 0x24, 0x27, 0x18,
   0x02, 0x1d, 0xb7, 0x95, 0x45, 0xae, 0x60, 0x3a,
   0xcc, 0xcb, 0xca, 0x6f, 0xb8, 0x6f, 0x1b, 0x95,
   0xba, 0x46, 0x76, 0x8d, 0xc3, 0x61, 0xd0, 0x86,
   0xc3, 0x6c, 0x9a, 0xd9, 0x33, 0xde, 0xce, 0x97,
   0x28, 0x69, 0xa3, 0xd4, 0xbc, 0x49, 0x6d, 0x10,
   0x8d, 0x31, 0x77, 0x53, 0x94, 0xe3, 0xc5, 0xf9,
   0xea, 0xcd, 0xb3, 0x5b, 0xbf, 0x51, 0x6f, 0x57,
   0x8b, 0x9f, 0x40, 0x8b, 0x47, 0xa8, 0x77, 0xbe,
   0x44, 0x3d, 0x8b, 0x54, 0x63, 0xc3, 0xbb, 0xdc,
   0xeb, 0x9e, 0xec, 0xf5, 0x7c, 0x31, 0x69, 0x45,
   0xf5, 0x48, 0x2c, 0x84, 0x57, 0x53, 0xe1, 0x20,
   0x25, 0x2b, 0x70, 0x3a, 0x4e, 0x9f, 0x36, 0xc3,
   0x16, 0xf8, 0x4f, 0xa3, 0x21, 0x66, 0x51, 0x49,
   0xc6, 0x6d, 0xac, 0xd0, 0x2a, 0x8e, 0xd4, 0x5d,
   0x55, 0x11, 0x1a, 0x31, 0x77, 0xdb, 0xa8, 0xf4,
   0x7a, 0xb9, 0x32, 0xe8, 0x54, 0xaf, 0xed, 0x55,
   0x6e, 0x87, 0xd0, 0x5c, 0x6d, 0xb9, 0x19, 0x0f,
   0xbf, 0x16, 0xa6, 0xc7, 0x7c, 0xa2, 0xd3, 0x95,
   0x34, 0xad, 0xfc, 0xc2, 0x0a, 0xfc, 0x23, 0x2f,
   0xc7, 0xe9, 0x98, 0x92, 0x90, 0x11, 0x1d, 0xd5,
   0xf5, 0xbc, 0xd2, 0x8f, 0x09, 0xb0, 0x43, 0x63,
   0x65, 0x51, 0x17, 0xfd, 0x11, 0xee, 0xd2, 0x23,
   0xff, 0x9f, 0x4e, 0x39, 0xda, 0xfe, 0x89, 0xc1,
   0x9d, 0xf3, 0xe0, 0x31, 0xdd, 0x74, 0x36, 0xb2,
   0x9f, 0xfa, 0xd2, 0xd0, 0xfc, 0xc4, 0x37, 0x83,
   0xf0, 0x13, 0x1f, 0x8d, 0x80, 0x47, 0x53, 0x93,
   0x0c, 0x84, 0x1f, 0x69, 0x08, 0x74, 0x33, 0x29,
```

```
   0x59, 0x5f, 0x62, 0x7d, 0xe2, 0x59, 0x1c, 0x56,
   0x52, 0x75, 0xc2, 0x83, 0xad, 0xc0, 0xd4, 0x66,
   0x71, 0x6a, 0x1a, 0x61, 0x94, 0xa4, 0xa5, 0x73
   }

   unsigned char group16_responder[512] = {
   0x06, 0x85, 0x8b, 0x2e, 0x0c, 0x05, 0xfd, 0x1b,
   0x1f, 0x93, 0xd3, 0xc2, 0xe6, 0x70, 0xc8, 0xe1,
   0x17, 0x39, 0xe6, 0x38, 0x75, 0xfd, 0xdd, 0xe6,
   0x4a, 0xfe, 0x95, 0x5e, 0xd6, 0x80, 0x17, 0x2c,
   0x1d, 0xbb, 0x8c, 0xf7, 0x2d, 0x9b, 0x17, 0x93,
   0x4d, 0x92, 0xd3, 0x57, 0xa0, 0xcd, 0x44, 0x37,
   0x1d, 0xdf, 0xd3, 0x80, 0x25, 0xa0, 0xa8, 0x51,
   0x13, 0xb9, 0x63, 0xec, 0xda, 0xa2, 0x8b, 0xdb,
   0x2e, 0x09, 0x9c, 0x93, 0x09, 0x02, 0x64, 0xb4,
   0xee, 0xa2, 0x3c, 0x75, 0x0e, 0xbb, 0x31, 0x44,
   0xff, 0xf0, 0x7e, 0x99, 0x86, 0x17, 0xe5, 0xc4,
   0xf9, 0x39, 0xe1, 0xec, 0xed, 0xd5, 0x13, 0xe9,
   0x97, 0xda, 0x2b, 0xb5, 0x1e, 0x23, 0x88, 0x1a,
   0xb5, 0x10, 0xda, 0x24, 0x05, 0xe7, 0xdf, 0xc3,
   0xc2, 0x24, 0xd2, 0xf4, 0x14, 0x6c, 0xfd, 0x2e,
   0x62, 0xa8, 0x00, 0x4e, 0xa9, 0x96, 0x3d, 0x48,
   0x4f, 0xcf, 0x62, 0xfe, 0x16, 0x06, 0x56, 0x81,
   0x1b, 0x58, 0xca, 0x84, 0xd2, 0x86, 0xad, 0xc6,
   0x66, 0xb4, 0x6f, 0x49, 0x91, 0x2a, 0xb5, 0x3b,
   0x39, 0xec, 0x88, 0xdf, 0x31, 0x24, 0x44, 0x04,
   0x30, 0x4e, 0x91, 0xdd, 0x1f, 0xf7, 0x62, 0x0c,
   0x8a, 0xf1, 0xcd, 0xf1, 0xcf, 0x56, 0x42, 0x1d,
   0x1b, 0xb2, 0x47, 0x7f, 0x4c, 0x82, 0x88, 0xbe,
   0x99, 0x31, 0x96, 0x6e, 0x5f, 0xa0, 0x6a, 0xa0,
   0x53, 0x63, 0xdb, 0xb9, 0xe0, 0xe4, 0x8f, 0xa9,
   0x44, 0x32, 0x2e, 0x05, 0x70, 0x3b, 0x6e, 0xc3,
   0x82, 0x36, 0x51, 0x4c, 0xbe, 0x38, 0x61, 0x54,
   0x66, 0x5c, 0x88, 0x42, 0x50, 0x84, 0x1a, 0x69,
   0xdf, 0xc5, 0x2b, 0x00, 0x3d, 0xdb, 0xe1, 0x92,
   0x69, 0xb4, 0xda, 0xfa, 0x87, 0x43, 0x9e, 0xdd,
   0x03, 0x29, 0xd4, 0x06, 0xef, 0x63, 0x60, 0xe3,
   0x83, 0xe3, 0x28, 0xd7, 0xa3, 0x47, 0xeb, 0xb7,
   0x0a, 0x20, 0x5a, 0x9e, 0x61, 0x68, 0xcc, 0x0b,
   0x39, 0xdc, 0x7b, 0x8c, 0x22, 0x0f, 0xc8, 0xd4,
   0x0c, 0x44, 0x9c, 0xa2, 0xb4, 0xd2, 0xf4, 0x71,
   0xeb, 0xc6, 0x75, 0xb8, 0x53, 0x8c, 0x93, 0x9a,
   0xf2, 0xd7, 0xba, 0x45, 0x40, 0xef, 0x56, 0xaf,
   0xdd, 0x1b, 0xcc, 0x0e, 0xe0, 0x3b, 0x2f, 0xd5,
   0xc5, 0xc7, 0x36, 0x37, 0xc0, 0x5e, 0xff, 0xb1,
   0x31, 0xce, 0xce, 0xc3, 0x28, 0xb7, 0x84, 0x88,
   0xe2, 0x7f, 0x11, 0x62, 0xb1, 0x14, 0x41, 0xe2,
   0x7e, 0xfb, 0x31, 0x0e, 0x5b, 0xba, 0x27, 0xf7,
```

```
0xa0, 0xce, 0xa0, 0xb8, 0xdd, 0xbe, 0xc4, 0x74,
0xe0, 0xc9, 0x50, 0x71, 0x16, 0x81, 0x42, 0x2a,
0xa9, 0xda, 0x2e, 0x3a, 0x85, 0x0b, 0x62, 0x5a,
0x55, 0x31, 0x31, 0xc4, 0xda, 0x4b, 0x36, 0x9a,
0xa6, 0x0b, 0x78, 0x51, 0x50, 0xea, 0x44, 0x07,
0x6d, 0xf7, 0x49, 0xb0, 0xea, 0x7e, 0x12, 0x92,
0x88, 0x5e, 0xb8, 0xee, 0x0b, 0xa9, 0xd8, 0x04,
0xbe, 0xd8, 0x5d, 0x8e, 0x0a, 0xea, 0x5c, 0xce,
0xf5, 0x2d, 0x80, 0xff, 0x57, 0x07, 0x0c, 0x06,
0x20, 0xaf, 0xb5, 0x32, 0x16, 0xb5, 0x79, 0x60,
0xce, 0x3b, 0xb8, 0x55, 0x4c, 0xf5, 0x58, 0x90,
0xeb, 0xae, 0x48, 0x04, 0x8b, 0x76, 0xfc, 0x66,
0x18, 0x70, 0x13, 0x5c, 0x85, 0xa4, 0x18, 0xf1,
0xbb, 0x06, 0x4e, 0x59, 0x19, 0x75, 0x4d, 0xaf,
0x97, 0x6b, 0x1c, 0x82, 0x2f, 0xbf, 0x91, 0xbb,
0xb7, 0x93, 0x21, 0xf2, 0xca, 0xf7, 0xec, 0x31,
0xf2, 0x05, 0x73, 0x1e, 0x69, 0x30, 0xd1, 0xd9,
0xef, 0x4f, 0x57, 0xc5, 0xf2, 0x46, 0xbf, 0xe8,
0x81, 0xb6, 0x24, 0xdb, 0xf3, 0x67, 0x27, 0x97,
0xb8, 0x04, 0x8c, 0xa3, 0xfe, 0xda, 0x6b, 0x9d,
0x51, 0xba, 0xbd, 0xb5, 0xca, 0xfe, 0x73, 0xce,
0x48, 0x9d, 0x28, 0x30, 0x7d, 0x53, 0x63, 0xc7
}
```

A.2.4. Role-specific Elements for 8192-bit FFC group

```
unsigned char group18_initiator[1024] = {
0x5b, 0x92, 0xc0, 0x76, 0x71, 0x48, 0x5e, 0x4d,
0x9d, 0xdf, 0xde, 0xc3, 0xcc, 0x32, 0x6e, 0xf1,
0x90, 0xef, 0x71, 0x86, 0xde, 0x55, 0x60, 0x3e,
0x24, 0x3e, 0xc2, 0x38, 0x09, 0x65, 0x56, 0xfb,
0x2b, 0x8b, 0x97, 0x07, 0x19, 0xac, 0xaf, 0x59,
0x32, 0xd7, 0x65, 0x19, 0xa2, 0x02, 0x83, 0xa3,
0x72, 0xc2, 0x2c, 0xe8, 0x15, 0x58, 0x0d, 0xb8,
0x1a, 0xad, 0xe3, 0xbc, 0xc0, 0x82, 0xab, 0x42,
0x9a, 0x7e, 0x0d, 0x39, 0x6a, 0x81, 0x3e, 0x72,
0xee, 0xba, 0x58, 0x01, 0xe6, 0x36, 0x9f, 0x82,
0x49, 0xa3, 0x6a, 0x88, 0x59, 0x0f, 0x77, 0x6b,
0x4d, 0x6a, 0x36, 0x25, 0xf0, 0xbc, 0x75, 0x53,
0x45, 0x1b, 0x02, 0x4c, 0x99, 0x5d, 0x51, 0x87,
0x41, 0x3c, 0xcc, 0x54, 0x6a, 0xdc, 0x6e, 0x22,
0xb5, 0x7d, 0xa9, 0x65, 0x90, 0xd3, 0x38, 0x4c,
0xa8, 0x26, 0x22, 0x57, 0xd8, 0x55, 0xc0, 0xc9,
0x9a, 0x62, 0x08, 0x71, 0x2b, 0x55, 0xde, 0x89,
0xd4, 0xf8, 0xab, 0x5f, 0x55, 0x42, 0xc0, 0x40,
0x60, 0x61, 0x1b, 0x68, 0x01, 0x80, 0x67, 0x27,
0x95, 0x8e, 0x6b, 0xcd, 0xc1, 0x04, 0x35, 0x96,
0x8e, 0x15, 0x1e, 0xd3, 0x01, 0x7d, 0x81, 0x38,
```

```
   0xe8, 0xbf, 0xd1, 0xa8, 0x31, 0xd5, 0x49, 0x11,
   0x0d, 0x78, 0x2f, 0x61, 0x31, 0xfc, 0xcc, 0x11,
   0x4f, 0x09, 0xa1, 0x13, 0x2b, 0x0e, 0x73, 0xbc,
   0x1f, 0xef, 0x01, 0x7d, 0xd3, 0x26, 0xe3, 0xda,
   0xa9, 0xaa, 0x15, 0xaf, 0x47, 0xe5, 0xc0, 0x39,
   0x29, 0xa3, 0x68, 0xfc, 0x03, 0xaa, 0x20, 0xc0,
   0xf8, 0xc5, 0xe2, 0xe3, 0x03, 0x0d, 0x3c, 0x20,
   0x7c, 0xbf, 0xa5, 0x1b, 0xd9, 0x92, 0x2d, 0x79,
   0x42, 0x42, 0xe8, 0x92, 0x25, 0x9a, 0x94, 0x54,
   0x40, 0xec, 0x8d, 0x55, 0x26, 0x71, 0xb3, 0x58,
   0x2c, 0x0b, 0x81, 0x4d, 0x53, 0xb8, 0x52, 0xf9,
   0x1b, 0xb1, 0x75, 0x60, 0xd4, 0x4b, 0x45, 0x72,
   0xa6, 0x61, 0x20, 0x96, 0xaa, 0x3b, 0xb9, 0x50,
   0x81, 0xe3, 0x93, 0xde, 0x4b, 0x80, 0xa2, 0xbd,
   0x20, 0x64, 0x63, 0xe2, 0x48, 0xc8, 0xec, 0x82,
   0x07, 0xa1, 0x7b, 0x45, 0x2a, 0xfb, 0xe9, 0x2f,
   0xa1, 0xf0, 0x69, 0x36, 0x2d, 0x4f, 0x1a, 0x85,
   0xf3, 0x58, 0x34, 0xe6, 0x0a, 0x9e, 0xe9, 0x9a,
   0x77, 0xe5, 0xf9, 0xa4, 0xc4, 0x14, 0xa2, 0x43,
   0xdd, 0xaa, 0x03, 0x17, 0x71, 0x55, 0x62, 0xf4,
   0xf5, 0x9c, 0x5f, 0x2f, 0xe7, 0x6f, 0xde, 0xa4,
   0x7a, 0xbb, 0x9d, 0xb5, 0x8d, 0xc3, 0x95, 0xf9,
   0x54, 0x06, 0xba, 0xd1, 0x31, 0xcf, 0x03, 0x6c,
   0x7a, 0x53, 0xd5, 0x76, 0x97, 0x4c, 0xbd, 0x23,
   0x59, 0xff, 0xfe, 0xea, 0xd3, 0xd1, 0x86, 0x10,
   0x2c, 0xf9, 0x9f, 0xc8, 0xd3, 0x45, 0x44, 0x2f,
   0x5a, 0xcf, 0x86, 0x8e, 0x1c, 0xc2, 0xb7, 0x04,
   0x75, 0x74, 0x78, 0xdf, 0x7a, 0x6f, 0xaf, 0x56,
   0x03, 0x93, 0x19, 0x4f, 0x4d, 0x73, 0x11, 0xc9,
   0x34, 0x90, 0x1a, 0x76, 0x18, 0x76, 0xa7, 0x19,
   0xe4, 0x5e, 0x66, 0x10, 0x2e, 0x0a, 0xbe, 0x7c,
   0x64, 0xd2, 0xd3, 0xbb, 0x18, 0x58, 0x86, 0xd9,
   0x54, 0x58, 0xf5, 0xeb, 0x86, 0xac, 0x61, 0x48,
   0xbc, 0x95, 0x1e, 0x13, 0xab, 0xef, 0x6e, 0xdf,
   0xbc, 0xa5, 0x78, 0x10, 0x87, 0x43, 0x9a, 0xd6,
   0xd6, 0x10, 0x30, 0xc0, 0xf5, 0x9b, 0x09, 0xc4,
   0x2c, 0xed, 0x8b, 0xeb, 0xc7, 0x3c, 0x12, 0xc5,
   0x1c, 0xf1, 0x88, 0xfd, 0x15, 0x45, 0xdb, 0xb3,
   0x35, 0x87, 0x40, 0xf8, 0x8a, 0xd1, 0x07, 0x32,
   0x2b, 0xf7, 0x4a, 0x77, 0xb5, 0x69, 0x4a, 0x20,
   0xdd, 0x69, 0x1e, 0x38, 0xac, 0x0b, 0x31, 0xda,
   0x43, 0x5d, 0xf0, 0x94, 0x22, 0x8d, 0x4a, 0x26,
   0xda, 0x91, 0xdf, 0xb7, 0xdd, 0xfb, 0x97, 0x88,
   0x7a, 0x43, 0x5e, 0xf3, 0x36, 0xbd, 0xef, 0xc0,
   0xe6, 0x7f, 0xd1, 0x81, 0x5b, 0xd6, 0x1b, 0x01,
   0x89, 0x19, 0x1d, 0x0e, 0xd0, 0x1a, 0x3a, 0x56,
   0x82, 0xf6, 0x2c, 0xdf, 0x6a, 0x42, 0xf5, 0x44,
   0x57, 0x61, 0x95, 0xdc, 0x9d, 0x4c, 0x15, 0xc0,
```

```
   0x29, 0x42, 0x55, 0x77, 0x28, 0xc8, 0x7c, 0xe2,
   0xc8, 0x44, 0xbd, 0xdd, 0x8e, 0xe1, 0xb8, 0xa3,
   0xa6, 0xb1, 0xa5, 0xfc, 0x9f, 0xed, 0x5f, 0xd7,
   0x58, 0xee, 0xe9, 0xa8, 0x1e, 0x11, 0x1a, 0x8c,
   0xf6, 0xea, 0x45, 0x8f, 0x41, 0x4d, 0xb9, 0x7f,
   0xe9, 0xd2, 0x90, 0x8c, 0xed, 0x0a, 0xd1, 0x12,
   0x48, 0x9b, 0x5e, 0x8a, 0x98, 0xc8, 0x0e, 0x71,
   0xff, 0x35, 0xa3, 0xc0, 0x17, 0x1c, 0x29, 0xe7,
   0x30, 0x47, 0x5f, 0x22, 0x62, 0x1b, 0xde, 0xb9,
   0xeb, 0x20, 0xcc, 0xeb, 0xfc, 0x7d, 0x38, 0x1e,
   0xce, 0x8a, 0x4e, 0xa8, 0xfe, 0xba, 0xa9, 0xfc,
   0x44, 0x8a, 0xcf, 0x0a, 0xe3, 0xf1, 0x91, 0x63,
   0xaf, 0xf2, 0x7d, 0x52, 0x2a, 0x6d, 0x38, 0xcf,
   0x10, 0xd5, 0xa3, 0xa1, 0xb0, 0xcc, 0x74, 0x08,
   0xe9, 0x97, 0xe1, 0x7e, 0xd8, 0xd1, 0x3f, 0x4e,
   0x8d, 0x6d, 0x4e, 0x3e, 0x33, 0xc7, 0xae, 0x28,
   0xb6, 0x6a, 0xd0, 0x15, 0xf3, 0xd6, 0xfd, 0x11,
   0x3e, 0xbc, 0x65, 0xe6, 0xf7, 0xb4, 0xfe, 0x55,
   0x03, 0x4d, 0x1f, 0x4f, 0x8b, 0xef, 0x8d, 0x11,
   0xa2, 0x9b, 0x42, 0x58, 0xf4, 0xdf, 0x0c, 0xf2,
   0x80, 0x0e, 0x02, 0xff, 0xe8, 0x46, 0x0d, 0xae,
   0x50, 0x41, 0x14, 0x37, 0x5d, 0x82, 0x26, 0x96,
   0x9f, 0x1d, 0xff, 0x9e, 0xe0, 0x01, 0x24, 0x19,
   0xe8, 0xca, 0xe7, 0x7b, 0xaa, 0xef, 0x05, 0x4a,
   0x8a, 0xdd, 0x3b, 0xe8, 0x93, 0xf0, 0x21, 0xab,
   0x7f, 0x77, 0xcd, 0xc1, 0x71, 0x9f, 0x6b, 0x2b,
   0x64, 0xfb, 0x43, 0x9e, 0x92, 0x33, 0x68, 0xe6,
   0x51, 0xc1, 0x16, 0x3d, 0xde, 0xf8, 0x85, 0x8a,
   0xb6, 0x6c, 0x96, 0x7a, 0x6b, 0x12, 0xd8, 0x18,
   0x30, 0x5e, 0x0c, 0x82, 0xff, 0xff, 0xd0, 0xf7,
   0x9c, 0x23, 0x30, 0x61, 0x80, 0x5b, 0xde, 0xd9,
   0x35, 0x16, 0xdd, 0x6a, 0x5b, 0xbe, 0x5a, 0x1d,
   0x77, 0x37, 0x2b, 0xee, 0x00, 0x61, 0xef, 0xfe,
   0xa8, 0x3f, 0x2e, 0xd6, 0x8e, 0x3a, 0x0f, 0x03,
   0xae, 0x46, 0x01, 0x0c, 0x75, 0x8c, 0x42, 0x9b,
   0x24, 0x02, 0x4b, 0xdb, 0xb8, 0x98, 0x31, 0xc7,
   0xd0, 0xd9, 0xb1, 0x89, 0x4a, 0x54, 0x74, 0xc0,
   0x68, 0x5d, 0xe7, 0x62, 0xe4, 0x44, 0x44, 0x5e,
   0x17, 0x75, 0x34, 0x96, 0xbe, 0xf1, 0x94, 0x49,
   0x14, 0xe7, 0x17, 0x79, 0xf6, 0xab, 0xe8, 0xf4,
   0x47, 0x77, 0x74, 0x10, 0x51, 0x3b, 0x30, 0x8e,
   0x8e, 0x00, 0x4f, 0x0e, 0x75, 0x03, 0xc7, 0x48,
   0xfb, 0xf8, 0x50, 0xc7, 0xe5, 0xfc, 0xe2, 0x7d,
   0x07, 0x90, 0xd5, 0x91, 0x6a, 0xce, 0x14, 0x12,
   0xab, 0xe6, 0x65, 0x64, 0xfb, 0x03, 0xce, 0xdf,
   0xf1, 0x0b, 0x11, 0x82, 0x5a, 0x11, 0xf6, 0xf9,
   0xd6, 0xf2, 0xfe, 0xd4, 0x72, 0x60, 0x80, 0xd4,
   0x53, 0x86, 0xe6, 0xfc, 0xb7, 0xc0, 0x03, 0x4d,
```

```
   0x3c, 0x32, 0xe9, 0xfd, 0x46, 0x9e, 0x81, 0x6a,
   0x72, 0xd6, 0x9c, 0x14, 0x70, 0x47, 0xbe, 0x35,
   0xef, 0xb2, 0xbb, 0x0a, 0xca, 0x84, 0xc9, 0x15,
   0xac, 0x83, 0x6b, 0x83, 0xe8, 0x36, 0x5d, 0x27,
   0xc0, 0x25, 0xc8, 0x92, 0x69, 0x6b, 0x51, 0x6a,
   0xc2, 0x8f, 0x9c, 0x8b, 0xf5, 0x35, 0x1e, 0x31,
   0x4b, 0xf7, 0xd6, 0x40, 0xa7, 0x1c, 0xe4, 0xa2,
   0x00, 0x3c, 0x78, 0x78, 0x8f, 0x27, 0x24, 0x7e,
   0x0b, 0x7f, 0xf6, 0xb0, 0x66, 0xf4, 0x79, 0x46,
   0x2e, 0x5b, 0x11, 0xca, 0x9b, 0x93, 0x4e, 0x99,
   0x2c, 0xd2, 0x2d, 0x88, 0x84, 0xf1, 0x1e, 0x0e
   }

   unsigned char group18_responder[1024] = {
   0xdf, 0x63, 0xdc, 0x3a, 0x74, 0xbd, 0x83, 0x8e,
   0x72, 0x90, 0x81, 0xf0, 0x85, 0x65, 0x86, 0x07,
   0x06, 0x05, 0xed, 0x93, 0x73, 0xa5, 0xbb, 0x88,
   0x42, 0x17, 0x8a, 0x10, 0x33, 0xc3, 0x6e, 0x9b,
   0x3b, 0x68, 0x33, 0x30, 0xa2, 0x3f, 0xb2, 0xa3,
   0x2d, 0x6b, 0xec, 0x34, 0x5a, 0xe7, 0x6a, 0xc1,
   0x11, 0xc8, 0xa5, 0x3d, 0x6c, 0xc5, 0xf7, 0x48,
   0xca, 0xac, 0xa6, 0x10, 0x0b, 0x7c, 0x93, 0xe9,
   0x45, 0x4d, 0xa7, 0x00, 0x30, 0xd2, 0xf5, 0xaf,
   0x93, 0xf1, 0xa5, 0x8a, 0x9f, 0x10, 0x14, 0xe6,
   0x6a, 0xe0, 0x5a, 0xe6, 0xea, 0x8b, 0x21, 0xbb,
   0x6a, 0x1f, 0x6c, 0x8c, 0x0b, 0x01, 0xda, 0xfd,
   0x4f, 0x0b, 0x7f, 0xe9, 0x46, 0x27, 0x8e, 0xaa,
   0x64, 0xd1, 0xd5, 0x40, 0xc9, 0xf7, 0x47, 0xef,
   0x9f, 0x7c, 0xce, 0x6c, 0x41, 0xd2, 0x9c, 0x47,
   0x09, 0x6e, 0xc7, 0xc2, 0xdc, 0x7d, 0x7e, 0xce,
   0x04, 0x6f, 0xf9, 0xc1, 0x86, 0x56, 0x1b, 0x88,
   0x7f, 0x62, 0x33, 0x3b, 0xca, 0xb9, 0xd4, 0x7d,
   0x24, 0xfa, 0x9f, 0xd8, 0xf8, 0x63, 0x91, 0x72,
   0x45, 0x82, 0x4d, 0x9f, 0xd7, 0x9d, 0xc8, 0x0b,
   0x4c, 0x6a, 0xc5, 0xf4, 0xec, 0x77, 0x3e, 0xfd,
   0xb7, 0x6b, 0xe0, 0x86, 0x32, 0x41, 0x25, 0xfe,
   0x43, 0x03, 0x0a, 0x07, 0x90, 0x75, 0xd5, 0xca,
   0x48, 0x23, 0xfb, 0x80, 0x5f, 0x22, 0xfe, 0x1c,
   0xba, 0x48, 0x2c, 0x28, 0x78, 0x5c, 0xc4, 0x98,
   0xad, 0xb7, 0xa6, 0x78, 0x5f, 0x84, 0x3a, 0xb6,
   0x96, 0xd5, 0xad, 0x88, 0x60, 0xb9, 0x09, 0x00,
   0xbb, 0x7d, 0x5c, 0xd6, 0x17, 0xfe, 0x18, 0x8e,
   0x07, 0x10, 0xc3, 0xe9, 0xd0, 0xb8, 0xe2, 0xfa,
   0x00, 0xae, 0xa1, 0xcd, 0x86, 0x33, 0xda, 0x4b,
   0x0c, 0x34, 0xa1, 0x6e, 0x19, 0x0b, 0xdb, 0xef,
   0xb0, 0xf6, 0x86, 0xc6, 0xe3, 0x8c, 0x9c, 0x53,
   0x15, 0x16, 0x04, 0xd9, 0xa8, 0xa7, 0x38, 0xc1,
   0x43, 0x9d, 0x7e, 0x33, 0x16, 0x1a, 0x8d, 0x33,
```

```
   0xe8, 0x2b, 0x47, 0xaf, 0x4c, 0xd0, 0x96, 0x87,
   0x5b, 0x57, 0x27, 0xc3, 0x1a, 0xf7, 0x12, 0xfd,
   0x8a, 0x64, 0xa0, 0xc9, 0x51, 0xc9, 0x95, 0xcb,
   0x7b, 0x9d, 0x97, 0xda, 0x3c, 0xae, 0x87, 0x1c,
   0x08, 0xa8, 0xb2, 0x3b, 0x92, 0xb7, 0x52, 0x97,
   0x99, 0x9c, 0x52, 0x92, 0xcc, 0xba, 0xbe, 0x16,
   0xf7, 0xac, 0x08, 0xb5, 0x1a, 0x99, 0x99, 0xee,
   0x33, 0x3a, 0x2a, 0xea, 0x70, 0xbe, 0xcc, 0xfd,
   0xc7, 0x6b, 0xf3, 0xa9, 0x9d, 0x66, 0x87, 0x1b,
   0x26, 0x0e, 0xf5, 0x04, 0x33, 0x82, 0x25, 0x75,
   0x74, 0x37, 0xbd, 0xfe, 0x78, 0xa2, 0x4a, 0x77,
   0xda, 0xef, 0x84, 0xfd, 0x5c, 0x1d, 0x09, 0xc7,
   0xd8, 0xc2, 0x97, 0xf8, 0xd0, 0x96, 0x6f, 0xf0,
   0x54, 0xea, 0x5e, 0x14, 0xb6, 0x9c, 0x16, 0x96,
   0xcd, 0x56, 0x3e, 0x49, 0xef, 0x8f, 0xbf, 0xc1,
   0x59, 0xd6, 0x0c, 0xa0, 0xd6, 0x6b, 0xff, 0x34,
   0xe5, 0x68, 0x70, 0x6b, 0x00, 0x87, 0xac, 0x4e,
   0x7e, 0x44, 0xa2, 0x04, 0xa2, 0x96, 0xfc, 0x73,
   0xe7, 0xe2, 0x7f, 0xc6, 0x0b, 0x17, 0xdb, 0xa6,
   0x45, 0xa8, 0x72, 0x61, 0x08, 0xc8, 0x4c, 0x19,
   0x2d, 0x27, 0x0b, 0x4f, 0xb9, 0xc0, 0x55, 0xd6,
   0x6d, 0x11, 0xd7, 0x15, 0x0d, 0xef, 0x97, 0x08,
   0xd3, 0x22, 0xd8, 0x03, 0x7f, 0x91, 0xe0, 0x0e,
   0xe4, 0x70, 0x75, 0x47, 0x33, 0xd9, 0x80, 0x08,
   0x5f, 0xc9, 0xea, 0x91, 0xd4, 0x4e, 0x80, 0x08,
   0xb3, 0x83, 0x37, 0xba, 0xd2, 0xe0, 0x6d, 0x44,
   0x83, 0x9a, 0xf1, 0xa6, 0x83, 0xc4, 0x5a, 0xd2,
   0x3f, 0x50, 0x7a, 0x19, 0xed, 0x82, 0xda, 0x02,
   0x6f, 0xbe, 0x27, 0x91, 0xd6, 0x7e, 0x11, 0xc1,
   0x95, 0x3a, 0x6a, 0x61, 0x80, 0x6c, 0x23, 0x1f,
   0xcd, 0x8d, 0xac, 0x5c, 0x4c, 0x14, 0xee, 0xde,
   0x08, 0x15, 0x87, 0xae, 0x2d, 0xf4, 0x77, 0x81,
   0xcb, 0x39, 0x9a, 0x51, 0xb1, 0x3e, 0xc6, 0xd5,
   0xcd, 0xa2, 0x3a, 0x15, 0x6a, 0x73, 0x2e, 0x78,
   0x24, 0x16, 0xb8, 0xd2, 0x3e, 0xb4, 0x0c, 0xc2,
   0x0e, 0x67, 0xea, 0xe9, 0x9c, 0xce, 0x5c, 0x6a,
   0x29, 0x27, 0xe1, 0x3f, 0xac, 0x9d, 0x31, 0xb8,
   0xda, 0x2f, 0x94, 0x6e, 0xfc, 0x33, 0xab, 0x54,
   0xf9, 0x8e, 0xe8, 0xbf, 0x8e, 0x33, 0x62, 0xab,
   0x1f, 0xca, 0x17, 0x68, 0x89, 0x27, 0x19, 0x52,
   0xb1, 0x4c, 0xd1, 0x04, 0xb8, 0x8d, 0xfe, 0xc2,
   0xa4, 0xbf, 0xb8, 0x4e, 0x7c, 0x24, 0x2f, 0xaa,
   0x8c, 0x57, 0x3c, 0x60, 0xe8, 0xd5, 0x1e, 0x7f,
   0x8b, 0x85, 0x0e, 0xfb, 0x0e, 0xbd, 0xed, 0x75,
   0x3f, 0x35, 0xeb, 0xb7, 0x35, 0x60, 0x53, 0x8f,
   0x7d, 0x86, 0xbc, 0x98, 0x8b, 0x2e, 0x22, 0x33,
   0x5e, 0x60, 0x49, 0x6d, 0xc3, 0xe0, 0xaa, 0xec,
   0x5e, 0x67, 0x87, 0xd8, 0x20, 0x92, 0x71, 0x34,
```

```
   0x46, 0xa4, 0xf4, 0x2d, 0x02, 0xaf, 0x64, 0x45,
   0xef, 0xea, 0xb3, 0x84, 0x39, 0xd6, 0x6b, 0xaf,
   0x7d, 0x63, 0xd3, 0x50, 0xf9, 0x95, 0xaf, 0xf7,
   0xf3, 0x8a, 0x6f, 0x59, 0xf1, 0x32, 0x37, 0x3d,
   0x6c, 0xd4, 0xaa, 0x17, 0xd7, 0x13, 0x19, 0xa7,
   0x51, 0x98, 0x21, 0x89, 0x26, 0x12, 0xe1, 0xed,
   0x78, 0x04, 0x33, 0xd7, 0xc6, 0xf4, 0xe2, 0x39,
   0x5c, 0x37, 0xdb, 0x16, 0x42, 0xf9, 0x0a, 0x3d,
   0xee, 0x4b, 0x96, 0xbd, 0x60, 0x1f, 0x36, 0xbb,
   0xe6, 0x15, 0x95, 0x47, 0x7f, 0x8f, 0x2e, 0xff,
   0x59, 0x9b, 0xfc, 0x99, 0x13, 0x87, 0xac, 0x85,
   0xd9, 0x84, 0x0c, 0x99, 0x95, 0xe8, 0x2b, 0xdf,
   0xae, 0x64, 0x7e, 0x24, 0x85, 0x67, 0x9c, 0x86,
   0x51, 0x8a, 0x61, 0x6c, 0x17, 0x24, 0x89, 0xba,
   0x2f, 0xfa, 0x9d, 0x3d, 0xa6, 0x51, 0xce, 0x85,
   0xf8, 0x95, 0x78, 0xeb, 0x00, 0x51, 0x06, 0xb4,
   0x8b, 0x02, 0x1b, 0x1c, 0xf7, 0x13, 0xcb, 0xee,
   0x83, 0x98, 0xdc, 0xab, 0xed, 0x57, 0x62, 0x78,
   0x1c, 0xc5, 0x5c, 0xac, 0xa6, 0x23, 0x68, 0xd0,
   0xa5, 0xda, 0x43, 0x2d, 0x61, 0x73, 0x66, 0x03,
   0xea, 0xc9, 0xad, 0x7e, 0xe3, 0x54, 0xa9, 0x53,
   0x3e, 0x23, 0x4c, 0x6a, 0x15, 0x70, 0xa5, 0x2c,
   0xee, 0xcd, 0x4d, 0x7e, 0x41, 0x6f, 0xa6, 0xc5,
   0x1c, 0x24, 0x37, 0x58, 0x00, 0x81, 0xd9, 0xb2,
   0xf7, 0x9a, 0x9c, 0xa3, 0xf5, 0xc6, 0x31, 0xc1,
   0xb2, 0x8b, 0x3d, 0xec, 0xbe, 0x21, 0xe7, 0x53,
   0x0f, 0xb8, 0x87, 0x76, 0xb5, 0x76, 0xcc, 0x50,
   0x03, 0x51, 0x8a, 0xa5, 0xb9, 0x50, 0xc7, 0x38,
   0xaf, 0x98, 0x01, 0xdf, 0x77, 0xb2, 0x9f, 0xe8,
   0xa5, 0x5b, 0x9d, 0x48, 0x3a, 0x82, 0xe1, 0x10,
   0xc8, 0x34, 0xc1, 0x07, 0x8f, 0x63, 0x60, 0x3e,
   0x25, 0xb6, 0x33, 0xbc, 0x15, 0xce, 0x99, 0x39,
   0x62, 0x83, 0x5b, 0xbc, 0x22, 0xb9, 0x0b, 0xd3,
   0x97, 0x2b, 0x87, 0xee, 0x85, 0xd6, 0x72, 0x01,
   0xb8, 0xdb, 0xe1, 0xdd, 0x5f, 0x61, 0x5f, 0x81,
   0x44, 0xcc, 0x65, 0x71, 0x44, 0xb7, 0xca, 0x48,
   0xd2, 0x33, 0x7b, 0x56, 0xe6, 0x07, 0xb8, 0xc5,
   0x6c, 0xb0, 0xf6, 0x72, 0x69, 0x75, 0xf7, 0xfc,
   0xd5, 0xab, 0xe0, 0xbb, 0x65, 0xcb, 0xd8, 0x4a,
   0xae, 0x99, 0x58, 0x5c, 0xe3, 0x61, 0x38, 0x07,
   0x97, 0xee, 0xa7, 0x67, 0x48, 0xb7, 0x9e, 0xc0,
   0xe9, 0xf5, 0x3c, 0x18, 0x3d, 0xb2, 0x06, 0x0f,
   0x75, 0x8d, 0xb8, 0x82, 0xfa, 0x6d, 0x30, 0x91,
   0x8b, 0x3b, 0xee, 0xf8, 0x27, 0x40, 0xec, 0x26,
   0x08, 0xd4, 0xca, 0x00, 0x8f, 0x28, 0xa2, 0x38,
   0xd9, 0xa0, 0x42, 0xc4, 0x51, 0x8b, 0x6c, 0xce
   }
```

Author's Address

Dan Harkins
HP Enterprise
1322 Crossman avenue
Sunnyvale, California 94089
USA

Phone: +1 415 997 9834
Email: dharkins@lounge.org

Internet Research Task Force
Internet-Draft
Intended status: Informational
Expires: February 7, 2019

D. Harkins
HP Enterprise

August 6, 2018

Public Key Exchange
draft-harkins-pkex-06

Abstract

This memo describes a password-authenticated protocol to allow two devices to exchange "raw" (uncertified) public keys and establish trust that the keys belong to their respective identities.

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 February 7, 2019.

Copyright Notice

Table of Contents

1. Introduction

Many authenticated key exchange protocols allow for authentication using uncertified, or "raw", public keys. Usually these specifications-- e.g. [RFC7250] for TLS and [RFC7670] for IKEv2-- assume keys are exchanged in some out-of-band mechanism.

[RFC7250] further states that "the main security challenge [to using ’raw’ public keys] is how to associate the public key with a specific entity. Without a secure binding between identifier and key, the protocol will be vulnerable to man-in-the- middle attacks."

The Public Key Exchange (PKEX) is designed to fill that gap: it establishs a secure binding between exchanged public keys and identifiers, it provides proof-of-possession of the exchanged public

keys to each peer, and it enables the establishment of trust in public keys that can subsequently be used to faccilitate authentication in other authentication and key exchange protocols. At the end of a successful run of PKEX the two peers will have trust in each others exchanged public keys and also share an authenticated symmetric key which may be discarded or used for another purpose.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

1.2. Notation

This memo describes a cryptographic exchange using sets of elements called groups. Groups can be based on elliptic curves (hereafter, ECC groups) or on the multiplicative group of the field of integers modulo an odd prime (hereafter FFC groups). The public keys exchanged by PKEX are elements in a group. Elements in groups are denoted in upper-case and scalar values are denoted with lower-case. There is a distinguised generator of the group, denoted G. The order of the sub-group formed by G is q which itself must be a large prime.

When both the initator and responder use a similar, but unique, datum it is denoted by appending an "i" for initiator or "r" for responder, e.g. if each side needs an element C then the initiator’s is Ci and the responder’s is Cr.

During the exchange, one side will generate data and the other side will attempt to reconstruct it. The reconstructed data is "primed". That is, if the initiator generates C then when responder tries to reconstruct it, the responder will refer to it as C’. Data that is directly sent and received is not primed.

The following notation is used in this memo:

C = A + B
: The "group operation" on two elements, A and B, that produces a third element, C. For finite field cryptography this is the modular multiplication, for elliptic curve cryptography this is point addition.

C = A - B
: The "group operation" on element A and the inverse of element B to produce a third element, C. Inversion is defined such that the group operation on an element and its inverse results in the

identity element, the value one (1) for finite field cryptography and the "point at infinity" for elliptic curve cryptography.

C = a * B
: This denotes repeated application of the group operation to B-- i.e. B + B-- (a - 1) times.

a = H(b)
: A cryptographic hash function that takes data b of indeterminate length and returns a fixed sized digest a.

a = F(B)
: A mapping function that takes an element and returns a scalar. For elliptic curve cryptography, F() returns the x-coordinate of the point B. For finite field cryptography, F() is the identity function.

a = KDF-b(c, d)
: A key derivation function that derives an output key a of length b from an input key c and context d.

a = HMAC(b, c)
: A keyed MAC function that produces a digest a using key b and text c.

a | b
: Concatentation of data a with data b.

{a}b[c]
: Authenticated-encryption of data (a), with a key (b), and associated data (c) that is authenticated but not encrypted. The result of this is a ciphertext that includes authentication information dependent on both a and c, whether c is actually transmitted or is somehow reconstructed. The receipent can decrypt a if it knows b and neither {a}b nor c have been altered during transmission, otherwise an error will be flagged.

2. Properties

Subversion of PKEX involves an adversary being able to insert its own public key into the exchange without the exchange failing, resulting in one of the parties to the exchange believing the adversary’s public key actually belongs to the protocol peer.

PKEX has the following properties:

- An adversary is unable to subvert the exchange without knowing the password.

- An adversary is unable to discover the password through passive attack.
- The only information exposed by an active attack is whether a single guess of the password is correct or not.
- Proof-of-possession of the private key is provided.
- At the end of the protocol, either trust is established in the peer's public key and the public key is bound to the peer's identity, or the exchange fails.

3. Assumptions

Due to the nature of the exchange, only DSA ([DSS]) and ECDSA ([X9.62]) keys can be exchanged with PKEX.

PKEX requires fixed elements that are unique to the particular role in the protocol, an initiator-specific element and a responder-specific element. They need not be secret. It is assumed that both parties know the role-specific elements for the particular group in which their key pairs were derived. Techniques to generate role-specific elements, and generated elements for popular groups, are listed in Appendix A.1 and Appendix A.2.

The generator used in PKEX SHALL be obtained from the domain parameter set defining the group, for example [DSS] for the NIST elliptic curves and [RFC5639] for brainpool curves.

The authenticated-encryption algorithm provides deterministic "key wrapping". To achieve this the AE scheme used in PKEX is AES-SIV as defined in [RFC5297].

The KDF provides for the generation of a cryptographically strong secret key from an "imperfect" source of randomness. To achieve this the KDF used in PKEX is the unsalted version of [RFC5869].

The keyed MAC function is HMAC per [RFC2104].

The following assumptions are made on PKEX:

- Only the peers involved in the exchange know the password.
- The peers' public keys are from the same group.
- The discrete logarithms of the public role-specific elements are unknown, and determining them is computationally infeasible.

4. Cryptographic Primitives

HKDF and HMAC require an underlying hash function and AES-SIV requires a key length. To provide for consistent security the hash algorithm and key length depend on the group chosen to use with PKEX.

For ECC, the hash algorithm and key length depends on the size of the prime defining the curve, p:

- SHA-256 and 256 bits: when len(p) <= 256
- SHA-384 and 384 bits: when 256 < len(p) <= 384
- SHA-512 and 512 bits: when 384 < len(p)

For FFC, the hash algorithm depends on the prime, p, defining the finite field:

- SHA-256 and 256 bits: when len(p) <= 2048
- SHA-384 and 384 bits: when 2048 < len(p) <= 3072
- SHA-512 and 512 bits: when 3072 < len(p)

5. Protocol Definition

PKEX is a balanced PAKE. The identical version of the password is used by both parties.

PKEX consists of two phases: authentication and reveal. It is described using the popular protocol participants, Alice (an initiator of PKEX), and Bob (a responder of PKEX).

We denote Alice’s role-specific element a Pi and Bob’s as Pr. The password is pw. For simplicity, Alice’s identity is "Alice" and Bob’s identity is "Bob". Alice’s public key she wants to share with Bob is A and her private key is a, while Bob’s public key he wants to share with Alice is B and his private key is b.

While both Alice and Bob expose their identities to passive eavesdroppers, the public keys they exchange (and ultimately gain trust in) are not. Once PKEX has finished, Alice and Bob can identifiy each other using their trusted public keys and thereby provide a level of anonymity to subsequent communications.

Implementations SHALL maintain a counter of unsuccessful exchanges for each password in order to defend against repeated active attacks to determine the password. This counter SHALL be set to zero when a

password is provisioned and incremented each time PKEX finishes unsuccessfully for that password. When the counter reaches a value of five (5) the password SHALL be irretrievably removed from the implementation.

5.1. Authentication Phase

The Authenticaiton phase is essentially the SPAKE2 key exchange. The peers derive ephemeral public keys, encrypt, and exchange them. Each party hashes the password and operates on the role-specific element to obtain a secret encrypting element. The group operation is then performed with the ephemeral key and the secret encrypting element to produce an encrypted ephmeral key. The ephemeral private keys MUST be generated with high quality (pseudo-)randomness and SHALL never be re-used.

```
     Alice:                                  Bob:
     ------                                  ----
  x, X = x*G                              y, Y = y*G
  Qa = H(pw)*Pi                           Qb = H(pw)*Pr
  M = X + Qa
                     Alice, M ------>
                                          Qa = H(pw)*Pi
                                          X' = M - Qa
                                          N = Y + Qb
                                          z = KDF-n(F(y*X'),
                                                    Alice | Bob |
                                                    F(M) | F(N) | pw)
                              <------ Bob, N
  Qb = H(pw)*Pr
  Y' = N - Qb
  z = KDF-n(F(x*Y'),
            Alice | Bob |
            F(M) | F(N) | pw)
```

where n is the key length and KDF uses the hash algorithm from Section 4.

Both M and N MUST be verified to be valid elements in the selected group. For ECC groups this means they MUST be valid points on the curve, for FFC groups they MUST be between one and the prime minus one, and the group operation on the element and the order of the sub-group, q, MUST equal one. If either element is not valid the protocol fails.

At this point the peers have exchanged ephemeral elements that will be unknown except by someone with knowledge of the password. Given

our assumptions that means only Alice and Bob can know the elements X and Y, and the secret key, z.

The secret encrypting elements Qa and Qb SHALL be irretrievably deleted at this point. The password MAY be irretrievably deleted at this time.

5.2. Reveal Phase

In the Reveal phase the peers commit to the particular public key they wish to exchange and reveal it to the peer. Proof-of-possession of the private key is accomplished by "signing" the public key, the identity to which the public key is bound, the receipient’s ephemeral public key, and the sender’s ephemeral public key.

The messages exchanged in the Reveal phase are encrypted and authenticated with AES-SIV using a key derived from the SPAKE2 key exchange in Section 5.1. Successful construction and validation of these messages authenticates the SPAKE2 exchange by proving possession of the SPAKE2 shared secret and therefore knowledge of the password. A single octet of the value zero (0) is used as associated data when encrypting Alice’s message to Bob and a single octet of the value one (1) is used as associated data when constructing Bob’s response. The associated data is not transferred as part of the either message.

The received public keys MUST be verified to be valid elements in the selected group using the same technique as above: for ECC groups they MUST be valid points on the curve, for FFC groups they MUST be between one and the prime minus one, and the group operation on the element and the order of the group, q, MUST equal one. If a received public key is not valid the protocol fails.

```
        Alice:                              Bob:
        ------                              ----
    u = HMAC(F(a*Y'), Alice | F(A) |
             F(Y') | F(X))

                    {A, u}z[0] ------>

                                        if (SIV-decrypt returns fail) fail
                                        if (A not valid element) fail
                                        u' = HMAC(F(y*A), Alice | F(A) |
                                                  F(Y) | F(X'))
                                        if (u' != u) fail
                                        v = HMAC(F(b*X'), Bob | F(B) |
                                                 F(X') | F(Y))

                              <------ {B, v}z[1]

    if (SIV-decrypt returns fail) fail
    if (B not valid element) fail
    v' = HMAC(F(x*B), Bob | F(B) |
              F(X) | F(Y))
    if (v'!= v) fail
```

where 0 and 1 are single octets of the value zero and one, respectively, and HMAC uses the hash algorithm from Section 4.

If the parties didn't fail they have each other's public key, knowledge that the peer possesses the corresponding private key, and trust that the public key belongs to the peer's identity that was authenticated in the Authentication Phase.

If the parties fail, the counter that protects against active attack (see Section 5) SHALL be incremented. If the value of the counter is five (5) the password SHALL be irretrievably deleted.

All ephemeral state created during the PKEX exchange SHALL be irretrievably deleted at this point. Once PKEX successfully completes the password MAY be deleted (or even exposed, with no loss of security). The authenticated and secret symmetric key, z, MAY be used for further key derivation with a different context but if not it SHOULD be irretrievably deleted.

6. IANA Considerations

This memo could create a registry of the fixed public elements for a nice cross section of popular groups. Or not. Once published this document will be a stable reference and a registry might not be needed.

7. Security Considerations

The encrypted shares exchanged in the Authentication phase MUST be ephemeral. Reuse of these keys, even with a different password, voids the security of the exchange.

If fixed elements other than those in Appendix A.1 and Appendix A.2 are used, their discrete logarithm MUST not be known. Knowledge of of the discrete logarithm of either of the fixed elements voids the security of the exchange.

The public keys exchanged in PKEX are never disclosed to an attacker, either passive or active. While they are, as the name implies, public, PKEX provides for secrecy of the exchanged keys for any protocol that might need such a capability.

PKEX has forward secrecy in the sense that exposure of the password used in a previous run of the protocol will not affect the security of that run. This also means that once PKEX has finished, the password can be exposed to a third party with out loss of security-- the public keys exchanged are still trusted and still bound to the entities that performed the exchange originally.

The Authentication Phase of PKEX is SPAKE2. The SPAKE2 security proof guarantees that if both sides bind the same password to each other’s identity they will derive the same secret. This means that the public key sent in the Reveal phase is guaranteed to be sent by the identified peer-- it is sent in a message that is integrity protected and encrypted by a key, z, derived from the SPAKE2 shared secret. This binds the peer’s public key to its authenticated identity. Proof-of-possession of the private key is provided by also sending a digest keyed by the result of a function of the private key and the peer’s ephemeral share from the Authentication Phase. Since the sender is not able to predict what random ephemeral share will be received in the Authentication Phase, it is unable to generate a keyed digest without knowing the private analog to the public key it is sending.

There is no proof of security of PKEX at this time.

8. Acknowledgements

The author wishes to thank Liliya Ruslanovna Ahmetzyanova, Stanislav Smyshlyaev, and Greg Rose for their detailed reviews, helpful comments, and patience in answering questions.

9. Changes/Author Notes

[RFC Editor: Please remove this section before publication]

00-04

Initial version recorded is -04

04-05

Accepted comments from Liliya Ruslanovna Ahmetzyanova and Stanislav Smyshlyaev.

Accepted comments from Greg Rose

Indicated G is from group definitions, added normative reference to brainpool curves.

Added a counter to deal with repeated active attack.

Mention in introduction that the result of PKEX is trust in the public key and an authenticated symmetric key

Make group description more formal and accurate.

Add some assumptions to the notational definition for authenticated encryption.

Send identities during auth phase instead of asssuming identities are somehow learned a priori. Add mention that public key is not exposed to passive attackers.

Note that ephemeral private keys must be generated with high quality randomness and never be reused.

Define what it means to verify M and N for both FFC and ECC.

Fix the technique used to generate the fixed elements for both FFC and ECC.

10. References

10.1. Normative References

[DSS] U.S. Department of Commerce/National Institute of Standards and Technology, "Digital Signature Standard (DSS)", Federal Information Processing Standards FIPS PUB 186-4, July 2013.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <http://www.rfc-editor.org/info/rfc2104>.

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

[RFC3526] Kivinen, T. and M. Kojo, "More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)", RFC 3526, DOI 10.17487/RFC3526, May 2003, <http://www.rfc-editor.org/info/rfc3526>.

[RFC5297] Harkins, D., "Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)", RFC 5297, DOI 10.17487/RFC5297, October 2008, <http://www.rfc-editor.org/info/rfc5297>.

[RFC5639] Lochter, M. and J. Merkle, "Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation", RFC 5639, DOI 10.17487/RFC5639, March 2010, <https://www.rfc-editor.org/info/rfc5639>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <http://www.rfc-editor.org/info/rfc5869>.

[X9.62] American National Standards Institute, "X9.62-2005", Public Key Cryptography for the Financial Services Industry (ECDSA), 2005.

10.2. Informative References

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <http://www.rfc-editor.org/info/rfc7250>.

[RFC7670] Kivinen, T., Wouters, P., and H. Tschofenig, "Generic Raw Public-Key Support for IKEv2", RFC 7670, DOI 10.17487/RFC7670, January 2016, <http://www.rfc-editor.org/info/rfc7670>.

Appendix A. Role-specific Elements

Role-specific elements for six popular elliptic curves and four popular modp groups from [RFC3526] have been generated using the following technique which guarantees that their discrete logarithm will be unknown.

A loop is performed to generate role-specific elements by generating a candidate, testing the candidate, and exiting the loop once the test succeeds. A single octet counter is incremented each time through the loop (first time through the loop, the counter is one).

To find a candidate, a hash of an identifier (the concatenation of the ASN.1 of the OID of the curve or the name of the FFC group), a constant string, and the counter is produced. If the length of the hash's digest is less than the desired bits, the digest is pre-pended to the inputs and the result is fed back into the hash (this time it is a hash of a concatentation of the old digest, identifier, constant string, counter) to produce the next length-of-digest bits. This is repeated until the number of bits has been produced. Excess octets are stripped off. The resulting string is interpreted as an integer with the first octet of the (first) hash being the high-order octet of the integer. If the prime defining the group (the modulo of all operations in an FFC group or the prime defining the curve for ECC groups) is not an integral number of octets, the bitstring is right-shifted, pre-pending with zero bits, in order to make a big-endian bitstring of the appropriate length. If that resulting big-endian number is larger than the prime defining group, the counter is incremented and the loop continues. Otherwise, the integer is checked to see whether it is in the correct sub-group. This process is different for ECC and FFC.

For ECC, the integer is treated as an x-coordinate and checked whether it produces a valid point on the curve. If a solution to the equation of the curve does not exist for that x-coordinate, the counter is incremented and a new candidate integer is calculated. If a solution to the equation of the curve exists for that x-coordinate, the polarity of the counter is used to select a y-coordinate-- if the counter is odd then use y-p, if the counter is even use y. This point is in the correct sub-group and looping terminates.

For FFC, the co-factor is calculated as (p-1)/q, where p is the prime modulus and q is the order of the sub-group. The integer is taken to the power of the co-factor modulo p. If the result is equal to one, the counter is increased and a new candidate integer is calculated. If the result is not equal to one then the result is in the correct sub-group and it becomes the element; looping terminates.

The hash algorithm used to generate candidates is determined using the criteria in Section 4.

The loop is performed twice for each elliptic curve and FFC group to produce initiator- and responder-specific elements. The string passed for the initiator-specific element is "PKEX Initiator", the string passed for the responder-specific element is "PKEX Responder".

For FFC groups, the identifier is "group X" (including the space character and excluding the quotation marks) where X is the id assigned to the group, e.g. the 2048-bit group is named "group 14". For ECC groups, the identifier is the DER-encoded ASN.1 representation of the OID of the curve.

A.1. ECC Role-specific Elements

A.1.1. Role-specific Elements for NIST p256

```
unsigned char nist_p256_initiator_x_coord[32] = {
0x56, 0x26, 0x12, 0xcf, 0x36, 0x48, 0xfe, 0x0b,
0x07, 0x04, 0xbb, 0x12, 0x22, 0x50, 0xb2, 0x54,
0xb1, 0x94, 0x64, 0x7e, 0x54, 0xce, 0x08, 0x07,
0x2e, 0xec, 0xca, 0x74, 0x5b, 0x61, 0x2d, 0x25
};
unsigned char nist_p256_initiator_y_coord[32] = {
0x3e, 0x44, 0xc7, 0xc9, 0x8c, 0x1c, 0xa1, 0x0b,
0x20, 0x09, 0x93, 0xb2, 0xfd, 0xe5, 0x69, 0xdc,
0x75, 0xbc, 0xad, 0x33, 0xc1, 0xe7, 0xc6, 0x45,
0x4d, 0x10, 0x1e, 0x6a, 0x3d, 0x84, 0x3c, 0xa4
};
unsigned char nist_p256_responder_x_coord[32] = {
0x1e, 0xa4, 0x8a, 0xb1, 0xa4, 0xe8, 0x42, 0x39,
0xad, 0x73, 0x07, 0xf2, 0x34, 0xdf, 0x57, 0x4f,
0xc0, 0x9d, 0x54, 0xbe, 0x36, 0x1b, 0x31, 0x0f,
0x59, 0x91, 0x52, 0x33, 0xac, 0x19, 0x9d, 0x76
};
unsigned char nist_p256_responder_y_coord[32] = {
0xd9, 0xfb, 0xf6, 0xb9, 0xf5, 0xfa, 0xdf, 0x19,
0x58, 0xd8, 0x3e, 0xc9, 0x89, 0x7a, 0x35, 0xc1,
0xbd, 0xe9, 0x0b, 0x77, 0x7a, 0xcb, 0x91, 0x2a,
0xe8, 0x21, 0x3f, 0x47, 0x52, 0x02, 0x4d, 0x67
};
```

A.1.2. Role-specific Elements for NIST p384

```
unsigned char nist_p384_initiator_x_coord[48] = {
0x95, 0x3f, 0x42, 0x9e, 0x50, 0x7f, 0xf9, 0xaa,
0xac, 0x1a, 0xf2, 0x85, 0x2e, 0x64, 0x91, 0x68,
0x64, 0xc4, 0x3c, 0xb7, 0x5c, 0xf8, 0xc9, 0x53,
0x6e, 0x58, 0x4c, 0x7f, 0xc4, 0x64, 0x61, 0xac,
0x51, 0x8a, 0x6f, 0xfe, 0xab, 0x74, 0xe6, 0x12,
0x81, 0xac, 0x38, 0x5d, 0x41, 0xe6, 0xb9, 0xa3
};
unsigned char nist_p384_initiator_y_coord[48] = {
0x76, 0x2f, 0x68, 0x84, 0xa6, 0xb0, 0x59, 0x29,
0x83, 0xa2, 0x6c, 0xa4, 0x6c, 0x3b, 0xf8, 0x56,
0x76, 0x11, 0x2a, 0x32, 0x90, 0xbd, 0x07, 0xc7,
0x37, 0x39, 0x9d, 0xdb, 0x96, 0xf3, 0x2b, 0xb6,
0x27, 0xbb, 0x29, 0x3c, 0x17, 0x33, 0x9d, 0x94,
0xc3, 0xda, 0xac, 0x46, 0xb0, 0x8e, 0x07, 0x18
};
unsigned char nist_p384_responder_x_coord[48] = {
0xad, 0xbe, 0xd7, 0x1d, 0x3a, 0x71, 0x64, 0x98,
0x5f, 0xb4, 0xd6, 0x4b, 0x50, 0xd0, 0x84, 0x97,
0x4b, 0x7e, 0x57, 0x70, 0xd2, 0xd9, 0xf4, 0x92,
0x2a, 0x3f, 0xce, 0x99, 0xc5, 0x77, 0x33, 0x44,
0x14, 0x56, 0x92, 0xcb, 0xae, 0x46, 0x64, 0xdf,
0xe0, 0xbb, 0xd7, 0xb1, 0x29, 0x20, 0x72, 0xdf
};
unsigned char nist_p384_responder_y_coord[48] = {
0xab, 0xa7, 0xdf, 0x52, 0xaa, 0xe2, 0x35, 0x0c,
0xe3, 0x75, 0x32, 0xe6, 0xbf, 0x06, 0xc8, 0x7c,
0x38, 0x29, 0x4c, 0xec, 0x82, 0xac, 0xd7, 0xa3,
0x09, 0xd2, 0x0e, 0x22, 0x5a, 0x74, 0x52, 0xa1,
0x7e, 0x54, 0x4e, 0xfe, 0xc6, 0x29, 0x33, 0x63,
0x15, 0xe1, 0x7b, 0xe3, 0x40, 0x1c, 0xca, 0x06
};
```

A.1.3. Role-specific Elements for NIST p521

```
unsigned char nist_p521_initiator_x_coord[66] = {
0x00, 0x16, 0x20, 0x45, 0x19, 0x50, 0x95, 0x23,
0x0d, 0x24, 0xbe, 0x00, 0x87, 0xdc, 0xfa, 0xf0,
0x58, 0x9a, 0x01, 0x60, 0x07, 0x7a, 0xca, 0x76,
0x01, 0xab, 0x2d, 0x5a, 0x46, 0xcd, 0x2c, 0xb5,
0x11, 0x9a, 0xff, 0xaa, 0x48, 0x04, 0x91, 0x38,
0xcf, 0x86, 0xfc, 0xa4, 0xa5, 0x0f, 0x47, 0x01,
0x80, 0x1b, 0x30, 0xa3, 0xae, 0xe8, 0x1c, 0x2e,
0xea, 0xcc, 0xf0, 0x03, 0x9f, 0x77, 0x4c, 0x8d,
0x97, 0x76
};
unsigned char nist_p521_initiator_y_coord[66] = {
0x00, 0xb3, 0x8e, 0x02, 0xe4, 0x2a, 0x63, 0x59,
0x12, 0xc6, 0x10, 0xba, 0x3a, 0xf9, 0x02, 0x99,
0x3f, 0x14, 0xf0, 0x40, 0xde, 0x5c, 0xc9, 0x8b,
0x02, 0x55, 0xfa, 0x91, 0xb1, 0xcc, 0x6a, 0xbd,
0xe5, 0x62, 0xc0, 0xc5, 0xe3, 0xa1, 0x57, 0x9f,
0x08, 0x1a, 0xa6, 0xe2, 0xf8, 0x55, 0x90, 0xbf,
0xf5, 0xa6, 0xc3, 0xd8, 0x52, 0x1f, 0xb7, 0x02,
0x2e, 0x7c, 0xc8, 0xb3, 0x20, 0x1e, 0x79, 0x8d,
0x03, 0xa8
};
unsigned char nist_p521_responder_x_coord[66] = {
0x00, 0x79, 0xe4, 0x4d, 0x6b, 0x5e, 0x12, 0x0a,
0x18, 0x2c, 0xb3, 0x05, 0x77, 0x0f, 0xc3, 0x44,
0x1a, 0xcd, 0x78, 0x46, 0x14, 0xee, 0x46, 0x3f,
0xab, 0xc9, 0x59, 0x7c, 0x85, 0xa0, 0xc2, 0xfb,
0x02, 0x32, 0x99, 0xde, 0x5d, 0xe1, 0x0d, 0x48,
0x2d, 0x71, 0x7d, 0x8d, 0x3f, 0x61, 0x67, 0x9e,
0x2b, 0x8b, 0x12, 0xde, 0x10, 0x21, 0x55, 0x0a,
0x5b, 0x2d, 0xe8, 0x05, 0x09, 0xf6, 0x20, 0x97,
0x84, 0xb4
};
unsigned char nist_p521_responder_y_coord[66] = {
0x00, 0x46, 0x63, 0x39, 0xbe, 0xcd, 0xa4, 0x2d,
0xca, 0x27, 0x74, 0xd4, 0x1b, 0x91, 0x33, 0x20,
0x83, 0xc7, 0x3b, 0xa4, 0x09, 0x8b, 0x8e, 0xa3,
0x88, 0xe9, 0x75, 0x7f, 0x56, 0x7b, 0x38, 0x84,
0x62, 0x02, 0x7c, 0x90, 0x51, 0x07, 0xdb, 0xe9,
0xd0, 0xde, 0xda, 0x9a, 0x5d, 0xe5, 0x94, 0xd2,
0xcf, 0x9d, 0x4c, 0x33, 0x91, 0xa6, 0xc3, 0x80,
0xa7, 0x6e, 0x7e, 0x8d, 0xf8, 0x73, 0x6e, 0x53,
0xce, 0xe1
};
```

A.1.4. Role-specific Elements for brainpool p256r1

```
unsigned char brainpool_p256_initiator_x_coord[32] = {
0x46, 0x98, 0x18, 0x6c, 0x27, 0xcd, 0x4b, 0x10,
0x7d, 0x55, 0xa3, 0xdd, 0x89, 0x1f, 0x9f, 0xca,
0xc7, 0x42, 0x5b, 0x8a, 0x23, 0xed, 0xf8, 0x75,
0xac, 0xc7, 0xe9, 0x8d, 0xc2, 0x6f, 0xec, 0xd8
};
unsigned char brainpool_p256_initiator_y_coord[32] = {
0x93, 0xca, 0xef, 0xa9, 0x66, 0x3e, 0x87, 0xcd,
0x52, 0x6e, 0x54, 0x13, 0xef, 0x31, 0x67, 0x30,
0x15, 0x13, 0x9d, 0x6d, 0xc0, 0x95, 0x32, 0xbe,
0x4f, 0xab, 0x5d, 0xf7, 0xbf, 0x5e, 0xaa, 0x0b
};
unsigned char brainpool_p256_responder_x_coord[32] = {
0x90, 0x18, 0x84, 0xc9, 0xdc, 0xcc, 0xb5, 0x2f,
0x4a, 0x3f, 0x4f, 0x18, 0x0a, 0x22, 0x56, 0x6a,
0xa9, 0xef, 0xd4, 0xe6, 0xc3, 0x53, 0xc2, 0x1a,
0x23, 0x54, 0xdd, 0x08, 0x7e, 0x10, 0xd8, 0xe3
};
unsigned char brainpool_p256_responder_y_coord[32] = {
0x2a, 0xfa, 0x98, 0x9b, 0xe3, 0xda, 0x30, 0xfd,
0x32, 0x28, 0xcb, 0x66, 0xfb, 0x40, 0x7f, 0xf2,
0xb2, 0x25, 0x80, 0x82, 0x44, 0x85, 0x13, 0x7e,
0x4b, 0xb5, 0x06, 0xc0, 0x03, 0x69, 0x23, 0x64
};
```

A.1.5. Role-specific Elements for brainpool p384r1

```
unsigned char brainpool_p384_initiator_x_coord[48] = {
0x0a, 0x2c, 0xeb, 0x49, 0x5e, 0xb7, 0x23, 0xbd,
0x20, 0x5b, 0xe0, 0x49, 0xdf, 0xcf, 0xcf, 0x19,
0x37, 0x36, 0xe1, 0x2f, 0x59, 0xdb, 0x07, 0x06,
0xb5, 0xeb, 0x2d, 0xae, 0xc2, 0xb2, 0x38, 0x62,
0xa6, 0x73, 0x09, 0xa0, 0x6c, 0x0a, 0xa2, 0x30,
0x99, 0xeb, 0xf7, 0x1e, 0x47, 0xb9, 0x5e, 0xbe
};
unsigned char brainpool_p384_initiator_y_coord[48] = {
0x54, 0x76, 0x61, 0x65, 0x75, 0x5a, 0x2f, 0x99,
0x39, 0x73, 0xca, 0x6c, 0xf9, 0xf7, 0x12, 0x86,
0x54, 0xd5, 0xd4, 0xad, 0x45, 0x7b, 0xbf, 0x32,
0xee, 0x62, 0x8b, 0x9f, 0x52, 0xe8, 0xa0, 0xc9,
0xb7, 0x9d, 0xd1, 0x09, 0xb4, 0x79, 0x1c, 0x3e,
0x1a, 0xbf, 0x21, 0x45, 0x66, 0x6b, 0x02, 0x52
};
unsigned char brainpool_p384_responder_x_coord[48] = {
0x03, 0xa2, 0x57, 0xef, 0xe8, 0x51, 0x21, 0xa0,
0xc8, 0x9e, 0x21, 0x02, 0xb5, 0x9a, 0x36, 0x25,
0x74, 0x22, 0xd1, 0xf2, 0x1b, 0xa8, 0x9a, 0x9b,
0x97, 0xbc, 0x5a, 0xeb, 0x26, 0x15, 0x09, 0x71,
0x77, 0x59, 0xec, 0x8b, 0xb7, 0xe1, 0xe8, 0xce,
0x65, 0xb8, 0xaf, 0xf8, 0x80, 0xae, 0x74, 0x6c
};
unsigned char brainpool_p384_responder_y_coord[48] = {
0x2f, 0xd9, 0x6a, 0xc7, 0x3e, 0xec, 0x76, 0x65,
0x2d, 0x38, 0x7f, 0xec, 0x63, 0x26, 0x3f, 0x04,
0xd8, 0x4e, 0xff, 0xe1, 0x0a, 0x51, 0x74, 0x70,
0xe5, 0x46, 0x63, 0x7f, 0x5c, 0xc0, 0xd1, 0x7c,
0xfb, 0x2f, 0xea, 0xe2, 0xd8, 0x0f, 0x84, 0xcb,
0xe9, 0x39, 0x5c, 0x64, 0xfe, 0xcb, 0x2f, 0xf1
};
```

A.1.6. Role-specific Elements for brainpool p512r1

```
   unsigned char brainpool_p512_initiator_x_coord[64] = {
   0x4c, 0xe9, 0xb6, 0x1c, 0xe2, 0x00, 0x3c, 0x9c,
   0xa9, 0xc8, 0x56, 0x52, 0xaf, 0x87, 0x3e, 0x51,
   0x9c, 0xbb, 0x15, 0x31, 0x1e, 0xc1, 0x05, 0xfc,
   0x7c, 0x77, 0xd7, 0x37, 0x61, 0x27, 0xd0, 0x95,
   0x98, 0xee, 0x5d, 0xa4, 0x3d, 0x09, 0xdb, 0x3d,
   0xfa, 0x89, 0x9e, 0x7f, 0xa6, 0xa6, 0x9c, 0xff,
   0x83, 0x5c, 0x21, 0x6c, 0x3e, 0xf2, 0xfe, 0xdc,
   0x63, 0xe4, 0xd1, 0x0e, 0x75, 0x45, 0x69, 0x0f
   };
   unsigned char brainpool_p512_initiator_y_coord[64] = {
   0x50, 0xb5, 0x9b, 0xfa, 0x45, 0x67, 0x75, 0x94,
   0x44, 0xe7, 0x68, 0xb0, 0xeb, 0x3e, 0xb3, 0xb8,
   0xf9, 0x99, 0x05, 0xef, 0xae, 0x6c, 0xbc, 0xe3,
   0xe1, 0xd2, 0x51, 0x54, 0xdf, 0x59, 0xd4, 0x45,
   0x41, 0x3a, 0xa8, 0x0b, 0x76, 0x32, 0x44, 0x0e,
   0x07, 0x60, 0x3a, 0x6e, 0xbe, 0xfe, 0xe0, 0x58,
   0x52, 0xa0, 0xaa, 0x8b, 0xd8, 0x5b, 0xf2, 0x71,
   0x11, 0x9a, 0x9e, 0x8f, 0x1a, 0xd1, 0xc9, 0x99
   };
   unsigned char brainpool_p512_responder_x_coord[64] = {
   0x2a, 0x60, 0x32, 0x27, 0xa1, 0xe6, 0x94, 0x72,
   0x1c, 0x48, 0xbe, 0xc5, 0x77, 0x14, 0x30, 0x76,
   0xe4, 0xbf, 0xf7, 0x7b, 0xc5, 0xfd, 0xdf, 0x19,
   0x1e, 0x0f, 0xdf, 0x1c, 0x40, 0xfa, 0x34, 0x9e,
   0x1f, 0x42, 0x24, 0xa3, 0x2c, 0xd5, 0xc7, 0xc9,
   0x7b, 0x47, 0x78, 0x96, 0xf1, 0x37, 0x0e, 0x88,
   0xcb, 0xa6, 0x52, 0x29, 0xd7, 0xa8, 0x38, 0x29,
   0x8e, 0x6e, 0x23, 0x47, 0xd4, 0x4b, 0x70, 0x3e
   };
   unsigned char brainpool_p512_responder_y_coord[64] = {
   0x80, 0x1f, 0x43, 0xd2, 0x17, 0x35, 0xec, 0x81,
   0xd9, 0x4b, 0xdc, 0x81, 0x19, 0xd9, 0x5f, 0x68,
   0x16, 0x84, 0xfe, 0x63, 0x4b, 0x8d, 0x5d, 0xaa,
   0x88, 0x4a, 0x47, 0x48, 0xd4, 0xea, 0xab, 0x7d,
   0x6a, 0xbf, 0xe1, 0x28, 0x99, 0x6a, 0x87, 0x1c,
   0x30, 0xb4, 0x44, 0x2d, 0x75, 0xac, 0x35, 0x09,
   0x73, 0x24, 0x3d, 0xb4, 0x43, 0xb1, 0xc1, 0x56,
   0x56, 0xad, 0x30, 0x87, 0xf4, 0xc3, 0x00, 0xc7
   };
```

A.2. FFC Role-specific Elements

A.2.1. Role-specific Elements for 2048-bit FFC group

```
   unsigned char group 14_initiator[256] = {
   0x01, 0x1f, 0x33, 0x72, 0x90, 0x86, 0x76, 0x68,
   0x9d, 0x29, 0x9c, 0x42, 0xd2, 0x43, 0x1b, 0xeb,
```

```
   0x99, 0x53, 0x3e, 0x5c, 0x3e, 0xe5, 0x15, 0xa1,
   0x06, 0x01, 0xb5, 0x8b, 0xac, 0x33, 0xd8, 0xc7,
   0x30, 0x4d, 0xec, 0x84, 0x0d, 0xb1, 0x13, 0xd0,
   0xb3, 0x44, 0xeb, 0xbe, 0x6f, 0x70, 0x21, 0x8b,
   0xd7, 0xe2, 0x86, 0x9f, 0xfc, 0x03, 0xc6, 0x34,
   0xd2, 0x08, 0xdb, 0x1d, 0x6e, 0x57, 0xe2, 0xe0,
   0xa8, 0x0c, 0xbb, 0xb8, 0x37, 0xa5, 0x73, 0x75,
   0x31, 0x48, 0x49, 0x43, 0x24, 0xdb, 0x96, 0x71,
   0x40, 0xc6, 0xfa, 0xe7, 0x12, 0x13, 0xb4, 0x20,
   0x89, 0x46, 0x63, 0xff, 0x38, 0xc3, 0x72, 0x82,
   0xf6, 0xa1, 0x23, 0xd2, 0x2c, 0x25, 0xf9, 0x46,
   0x80, 0x76, 0x82, 0xb7, 0xed, 0x6c, 0x21, 0x28,
   0x79, 0xdb, 0xaa, 0xdd, 0x69, 0x84, 0xd7, 0x09,
   0x20, 0xac, 0x5f, 0x94, 0xf4, 0x10, 0x86, 0x98,
   0x0a, 0x69, 0xc4, 0x62, 0xb7, 0x48, 0xea, 0xa5,
   0xdf, 0x41, 0xce, 0xfa, 0xb6, 0x00, 0x41, 0xbd,
   0x9e, 0x35, 0xfa, 0x15, 0x41, 0x3e, 0xa8, 0x7f,
   0x4f, 0x44, 0xae, 0x14, 0x48, 0x53, 0xf2, 0x7c,
   0x4d, 0x69, 0xe4, 0xa0, 0x44, 0x32, 0x78, 0xbf,
   0x7a, 0x59, 0xe3, 0xd9, 0x6a, 0x32, 0xb1, 0x5a,
   0x63, 0xd7, 0x7d, 0x47, 0xb6, 0xe6, 0x00, 0x00,
   0xea, 0x70, 0x91, 0x4b, 0xde, 0x0e, 0xf5, 0x76,
   0x0b, 0x45, 0x1b, 0xa8, 0xee, 0x99, 0xb0, 0xd2,
   0x34, 0x4e, 0x7a, 0x95, 0x46, 0xbb, 0xf6, 0x51,
   0xba, 0xfa, 0x15, 0x90, 0xf9, 0x88, 0xc0, 0x49,
   0x3f, 0x5d, 0x98, 0x4e, 0x36, 0xcb, 0x96, 0xa9,
   0xcd, 0x47, 0x7f, 0x21, 0xff, 0x32, 0xde, 0xb3,
   0x65, 0xc3, 0xe1, 0xe9, 0x88, 0x8e, 0xbd, 0x3e,
   0xc1, 0x84, 0x63, 0x77, 0x26, 0xf9, 0x90, 0x64,
   0x66, 0x3a, 0x5c, 0xfc, 0x44, 0xbd, 0x6f, 0xd0
   };
   unsigned char group 14_responder[256] = {
   0x7a, 0x9e, 0x5f, 0xa9, 0xcb, 0x6e, 0x36, 0xe1,
   0x66, 0x75, 0x95, 0x42, 0xe8, 0x86, 0x44, 0xf0,
   0xe5, 0xe5, 0x4e, 0x7f, 0xb0, 0x63, 0x5c, 0x38,
   0xd3, 0x25, 0x02, 0xd3, 0x2a, 0x72, 0x92, 0xfa,
   0x17, 0xa1, 0x93, 0xc2, 0x9a, 0x15, 0xf9, 0x81,
   0xa6, 0x16, 0xfc, 0x72, 0xaf, 0xfa, 0xe6, 0x71,
   0x08, 0x96, 0x26, 0x49, 0x7a, 0x4d, 0xc8, 0xc2,
   0xc1, 0xdb, 0x63, 0x9d, 0xc3, 0x22, 0x3c, 0x9f,
   0xb4, 0x00, 0x3e, 0xe7, 0x02, 0x89, 0x0c, 0xb1,
   0x65, 0x97, 0x55, 0x0a, 0x74, 0x83, 0x0d, 0xe9,
   0x77, 0x5f, 0xc4, 0x00, 0x1c, 0xaf, 0x24, 0xca,
   0xb1, 0xcc, 0x31, 0x1c, 0x2d, 0x53, 0x8a, 0x79,
   0x01, 0xe1, 0x00, 0x62, 0x61, 0x1c, 0xa8, 0xf7,
   0x76, 0x73, 0x24, 0xad, 0x8b, 0xb0, 0x6f, 0xd6,
   0x83, 0x3d, 0x06, 0x9f, 0x9d, 0xf0, 0x84, 0x64,
   0xb2, 0xba, 0x11, 0xec, 0x1e, 0xfb, 0x21, 0x96,
```

```
0x0a, 0xab, 0x4c, 0x70, 0x79, 0x47, 0x7b, 0x6e,
0xce, 0x22, 0xd5, 0x22, 0x82, 0x06, 0xa8, 0x81,
0x0d, 0x39, 0x03, 0xca, 0x5f, 0x54, 0x67, 0x79,
0x20, 0xa7, 0xde, 0xd6, 0xba, 0x1e, 0x33, 0xe8,
0x85, 0xa0, 0x39, 0x5f, 0x8d, 0x8a, 0x91, 0x28,
0xb2, 0x63, 0xe6, 0x9b, 0xd1, 0x68, 0xff, 0xd8,
0x57, 0x7d, 0x85, 0x43, 0x70, 0xe1, 0xab, 0x55,
0x13, 0xc7, 0x02, 0x23, 0xfa, 0x8f, 0xf7, 0x9c,
0x25, 0x8e, 0xc1, 0x0e, 0xd4, 0xab, 0xf4, 0x81,
0x38, 0x86, 0x22, 0x16, 0x24, 0x06, 0x7f, 0x37,
0xbb, 0x2d, 0x16, 0x2b, 0xc7, 0x82, 0xe4, 0x93,
0xf6, 0x6b, 0x8f, 0x1f, 0xb6, 0x6f, 0x63, 0x66,
0x4d, 0xa4, 0x39, 0xd5, 0x57, 0x3b, 0x73, 0x69,
0x22, 0xf1, 0x62, 0xb3, 0xf4, 0x8c, 0x5c, 0x3f,
0xc8, 0xb1, 0x94, 0x76, 0x2b, 0x7f, 0x6b, 0x8d,
0xc6, 0xa5, 0x5f, 0xc6, 0x06, 0x74, 0x36, 0xea
};
```

A.2.2. Role-specific Elements for 3072-bit FFC group

```
unsigned char group 15_initiator[384] = {
0x2a, 0xfc, 0x6e, 0xa4, 0x33, 0x54, 0xa1, 0xba,
0x34, 0x25, 0x84, 0x6c, 0xe3, 0x54, 0x2d, 0x52,
0xdd, 0x59, 0x9c, 0xef, 0xa6, 0x96, 0x2d, 0x1d,
0x53, 0xd4, 0xd4, 0x2e, 0xe9, 0x18, 0xb3, 0x2d,
0x75, 0x11, 0xeb, 0x3f, 0x1d, 0x3d, 0xac, 0x67,
0x62, 0x99, 0xa6, 0xe0, 0x22, 0xa1, 0xa5, 0xd6,
0x07, 0xfb, 0xe0, 0x76, 0x29, 0x8f, 0xf7, 0x3d,
0xa1, 0x99, 0x64, 0x44, 0xb5, 0xe4, 0xfa, 0x69,
0x00, 0x3c, 0x46, 0x56, 0x99, 0xf1, 0xb6, 0xc1,
0xa4, 0x2d, 0x54, 0xf4, 0x4e, 0x2c, 0xdc, 0x14,
0x27, 0xf5, 0xbb, 0x55, 0x61, 0xda, 0x36, 0x0d,
0x46, 0xa6, 0xd7, 0xe9, 0x9e, 0xcc, 0x7e, 0x35,
0x87, 0x32, 0xa1, 0xb9, 0x80, 0x07, 0x16, 0xaa,
0x74, 0xa5, 0x0f, 0xe0, 0x96, 0xb1, 0x25, 0x88,
0x6d, 0xda, 0x64, 0xc9, 0xa9, 0x5e, 0x6d, 0xb8,
0x7a, 0xf4, 0x42, 0xf3, 0xba, 0x37, 0xe8, 0xbd,
0x23, 0x36, 0x7b, 0xdc, 0x60, 0x93, 0x94, 0x5a,
0xb2, 0x99, 0x2a, 0x22, 0x1d, 0x50, 0xd6, 0x1d,
0xb7, 0xbc, 0xb9, 0xd1, 0x99, 0x3c, 0x06, 0x11,
0x79, 0x06, 0x21, 0x58, 0x60, 0x45, 0x3a, 0x00,
0xb6, 0x43, 0x0d, 0xcd, 0xa7, 0x60, 0x83, 0x3a,
0x7d, 0x9c, 0x35, 0x58, 0xc4, 0x0d, 0xcc, 0xef,
0x66, 0x55, 0xa9, 0xd2, 0xce, 0xe2, 0x80, 0x73,
0x26, 0xab, 0x7c, 0x8a, 0xf9, 0x1b, 0x3e, 0xf7,
0x75, 0x31, 0xea, 0x7f, 0x4a, 0x57, 0x15, 0x9a,
0x71, 0x92, 0xc3, 0x8f, 0xca, 0xab, 0x4b, 0x98,
0x11, 0xbe, 0x58, 0x8c, 0x20, 0x3d, 0x73, 0x4e,
```

```
0x39, 0xad, 0x17, 0x10, 0x99, 0x4f, 0x2e, 0x70,
0xae, 0xb6, 0xb8, 0x54, 0x2a, 0x37, 0x12, 0xf1,
0x85, 0x64, 0x9d, 0x97, 0x79, 0x8d, 0x69, 0x8c,
0x27, 0xd4, 0xf3, 0x65, 0x8a, 0xf3, 0x41, 0x42,
0x3e, 0x89, 0xf0, 0xa5, 0xbe, 0x71, 0x40, 0xb6,
0x56, 0x65, 0xb1, 0x62, 0x1f, 0x09, 0x76, 0xa3,
0xad, 0xb1, 0x16, 0x61, 0x87, 0x85, 0xfc, 0x1d,
0xa3, 0x1a, 0xf9, 0xa2, 0x4b, 0x25, 0x1c, 0x9f,
0x6d, 0x9b, 0xcd, 0x02, 0xc4, 0x0f, 0x64, 0x54,
0x97, 0x83, 0x2c, 0x41, 0xd6, 0x7b, 0x59, 0x0d,
0xcf, 0xdd, 0xa4, 0xd0, 0x75, 0xeb, 0xd9, 0x1c,
0xb8, 0xcb, 0x6c, 0x80, 0x00, 0x24, 0xf6, 0xf8,
0x62, 0x82, 0x97, 0x75, 0x0a, 0x4c, 0xfa, 0xbb,
0xbb, 0xe0, 0x87, 0x25, 0x86, 0x80, 0xc3, 0xb0,
0xc6, 0xb2, 0xfb, 0xe2, 0x8f, 0xb4, 0xd2, 0xc3,
0xbb, 0x78, 0xf4, 0xef, 0x9c, 0x1f, 0xd3, 0xa5,
0xab, 0xcf, 0xc2, 0xbd, 0x63, 0xc4, 0x5b, 0x2c,
0x9c, 0x3d, 0xa3, 0xed, 0xae, 0x97, 0xcc, 0x54,
0xdb, 0x3c, 0x04, 0x38, 0x1b, 0xaf, 0x22, 0x27,
0x53, 0xa4, 0xc1, 0xd6, 0x4a, 0x8f, 0xe9, 0x77,
0x13, 0x86, 0xf8, 0x0e, 0x1b, 0x2a, 0xdc, 0x6f
};
unsigned char group 15_responder[384] = {
0xbe, 0xfa, 0x77, 0xff, 0x9c, 0xa4, 0x21, 0x86,
0x6f, 0x22, 0x42, 0xf2, 0x86, 0x12, 0x70, 0x57,
0x7b, 0x1e, 0x00, 0x82, 0x0a, 0x10, 0xad, 0x84,
0x52, 0xe6, 0x3c, 0x39, 0x5e, 0x0d, 0xcc, 0x13,
0xfc, 0x82, 0x32, 0x58, 0x1d, 0x74, 0xab, 0x6e,
0xfa, 0xf1, 0xc2, 0x2f, 0x80, 0x55, 0xd0, 0x1e,
0x8a, 0x6d, 0x75, 0x8e, 0x80, 0x24, 0x64, 0x0e,
0x66, 0xc2, 0xf5, 0xbf, 0x89, 0x1c, 0x6b, 0xee,
0x35, 0x4c, 0x44, 0x16, 0x12, 0xe9, 0x26, 0x44,
0x74, 0xdd, 0x24, 0x84, 0x36, 0xfe, 0x5a, 0x66,
0x8a, 0xb6, 0x7c, 0xab, 0xf2, 0x8c, 0xc3, 0x98,
0xe7, 0xb0, 0xd1, 0x45, 0x22, 0xbf, 0x49, 0xa4,
0x09, 0x0e, 0xf0, 0xdf, 0xb5, 0xc4, 0xf7, 0xc9,
0x2d, 0x9e, 0x65, 0x93, 0x5d, 0x84, 0x1b, 0x93,
0xec, 0x5e, 0xdc, 0xb6, 0x8b, 0xee, 0x84, 0x3e,
0x0d, 0xf8, 0x81, 0x00, 0x60, 0x55, 0x8d, 0xab,
0x51, 0x31, 0x2c, 0xf4, 0x85, 0xbe, 0x4b, 0xe0,
0x61, 0xc2, 0x9a, 0xd1, 0xdb, 0xb2, 0x32, 0x11,
0x01, 0xca, 0xa3, 0x23, 0x28, 0xf8, 0x5a, 0x40,
0xe2, 0xaf, 0x65, 0xd5, 0xa1, 0x4f, 0xae, 0xa2,
0x1e, 0x3c, 0x23, 0xd3, 0x53, 0xb7, 0x59, 0xe6,
0x02, 0x5f, 0xb1, 0x81, 0xd9, 0xd9, 0x41, 0x02,
0x2d, 0xf3, 0x7f, 0xbe, 0x08, 0x9c, 0xa8, 0x58,
0x4f, 0x72, 0x7a, 0x71, 0xc8, 0x34, 0xb4, 0xbe,
0xd6, 0x46, 0x55, 0x47, 0x15, 0x29, 0x95, 0x01,
```

```
0x19, 0x1f, 0xbb, 0xfe, 0x0d, 0xce, 0xb6, 0x41,
0xf7, 0x22, 0x19, 0x8b, 0x57, 0x2a, 0x42, 0x05,
0xbc, 0xba, 0x08, 0xb5, 0xd3, 0x5b, 0xab, 0xc5,
0x34, 0xb5, 0xe4, 0x2e, 0xf4, 0x23, 0x69, 0x63,
0x0c, 0x0e, 0xfd, 0x9d, 0xf1, 0x3f, 0xee, 0x14,
0xad, 0x9b, 0x2c, 0x61, 0x09, 0xb0, 0xea, 0x46,
0x3c, 0x16, 0xca, 0xcd, 0x72, 0x53, 0xe9, 0xf7,
0x87, 0x96, 0x6f, 0xec, 0xbf, 0x93, 0x36, 0x43,
0x66, 0x60, 0x48, 0xfe, 0x3f, 0xb0, 0x47, 0x26,
0x87, 0x86, 0x07, 0x08, 0xb4, 0x7d, 0xab, 0x60,
0xad, 0xf1, 0x84, 0xd9, 0x5a, 0xeb, 0xbb, 0xdb,
0x15, 0x69, 0x42, 0x62, 0x2c, 0x82, 0x6a, 0x24,
0xcb, 0xce, 0x7d, 0xd9, 0xd3, 0xcb, 0x10, 0x55,
0x73, 0x36, 0x15, 0xe2, 0x05, 0x91, 0xc9, 0x68,
0x09, 0x76, 0xcb, 0xcf, 0x6c, 0xd2, 0x06, 0x34,
0xcd, 0xb5, 0x69, 0x44, 0x66, 0x33, 0x37, 0xec,
0x24, 0x17, 0x73, 0x79, 0x74, 0xdd, 0xba, 0x04,
0xad, 0xb9, 0xd6, 0xef, 0x60, 0xcb, 0x58, 0xfd,
0x71, 0xac, 0x6e, 0xb8, 0x78, 0xd7, 0x4d, 0x6e,
0x72, 0xa1, 0x78, 0x68, 0xbd, 0x9c, 0x56, 0x81,
0x94, 0x69, 0xc7, 0x63, 0xe3, 0x2b, 0xda, 0x76,
0xe5, 0x2f, 0xf8, 0xaa, 0xb2, 0x4b, 0xf6, 0xa1,
0xe5, 0xa7, 0xa2, 0xbc, 0xf9, 0x0a, 0xb9, 0x63
};
```

A.2.3. Role-specific Elements for 4096-bit FFC group

```
unsigned char group 16_initiator[512] = {
0x2e, 0xf4, 0x19, 0xbc, 0x45, 0x4b, 0x5a, 0x16,
0x05, 0x38, 0xc0, 0x82, 0x6e, 0xab, 0x66, 0xcc,
0xe5, 0xd1, 0xd8, 0x64, 0xdc, 0x5a, 0x8d, 0xae,
0x90, 0x00, 0x1a, 0x72, 0xb9, 0xd5, 0xbb, 0xfa,
0xc1, 0x91, 0xe3, 0xde, 0x50, 0xed, 0x31, 0x31,
0x4b, 0xf2, 0xb7, 0x2e, 0xbe, 0xa0, 0x31, 0x9b,
0xce, 0xbf, 0x35, 0xd8, 0xde, 0xb6, 0x38, 0xd3,
0x2c, 0xfc, 0xf5, 0x7b, 0x5f, 0x60, 0xef, 0x11,
0x08, 0x44, 0x0a, 0x68, 0x6c, 0x07, 0x40, 0x3b,
0xdc, 0xc8, 0x1d, 0xdd, 0xd0, 0xc3, 0x15, 0x19,
0xcf, 0x85, 0x43, 0xc0, 0xab, 0x65, 0x75, 0x48,
0x75, 0x54, 0x5d, 0x9d, 0x73, 0xd6, 0x57, 0x08,
0x78, 0x0c, 0x3b, 0xfd, 0x22, 0x90, 0xdf, 0x5b,
0x13, 0x90, 0x17, 0x61, 0xb3, 0x18, 0x67, 0x14,
0x1e, 0xaa, 0x81, 0xea, 0x9e, 0xd0, 0xe7, 0x4e,
0x8b, 0x69, 0xc8, 0xef, 0xe4, 0x58, 0x9e, 0xf5,
0x86, 0xd1, 0x3b, 0xd2, 0x94, 0x7d, 0x8a, 0x95,
0xca, 0xdc, 0x04, 0x80, 0x60, 0x66, 0x4f, 0x2c,
0xf5, 0x69, 0xb4, 0xd6, 0x9e, 0xe6, 0xf9, 0x88,
0x0a, 0x0b, 0x5e, 0x01, 0xc7, 0x50, 0xad, 0xe8,
```

```
   0x4f, 0x1d, 0x0c, 0xcd, 0x6c, 0x92, 0x46, 0x2e,
   0x06, 0x4f, 0x7d, 0x18, 0x1b, 0xb8, 0x03, 0xef,
   0xff, 0x85, 0x59, 0x16, 0x44, 0xd3, 0x28, 0x80,
   0x58, 0x91, 0xf0, 0x9c, 0x08, 0x83, 0x87, 0x63,
   0xf8, 0x6d, 0xc9, 0x3a, 0x17, 0x9d, 0xb0, 0x50,
   0x4f, 0x1f, 0xa5, 0x6a, 0x88, 0xda, 0xf2, 0xab,
   0x93, 0x36, 0x58, 0x9b, 0xf3, 0xe7, 0x93, 0xac,
   0x28, 0xd9, 0x62, 0xf3, 0xc5, 0xe0, 0x2c, 0xe1,
   0x23, 0x38, 0xb9, 0xd7, 0xfc, 0x54, 0x0e, 0x8e,
   0x28, 0xf3, 0x88, 0x32, 0x81, 0x8b, 0x45, 0x47,
   0xe9, 0x54, 0xff, 0x7f, 0x8b, 0x45, 0xc2, 0xc5,
   0xa1, 0xe3, 0x9a, 0x02, 0xd4, 0x8b, 0x91, 0x44,
   0x90, 0xfa, 0xb0, 0x86, 0x27, 0xac, 0x09, 0x7b,
   0x93, 0x75, 0x86, 0xfd, 0x46, 0x99, 0xbf, 0xd9,
   0xbd, 0xe2, 0xf2, 0x79, 0x24, 0x9a, 0x84, 0x5c,
   0x12, 0x67, 0xf8, 0xe1, 0xa8, 0xd6, 0x60, 0x31,
   0x0f, 0xd9, 0x7f, 0xb3, 0xba, 0x0c, 0x92, 0x55,
   0x6a, 0x5c, 0x8a, 0xca, 0x98, 0x78, 0xbc, 0x0d,
   0x9f, 0x6c, 0x26, 0xab, 0xfb, 0x80, 0xed, 0xa8,
   0xb3, 0x08, 0x15, 0xaa, 0x46, 0x09, 0x6a, 0x55,
   0x76, 0xd4, 0xbf, 0xc0, 0x84, 0xf6, 0xf0, 0x41,
   0xc0, 0xf8, 0xdb, 0xb7, 0x46, 0xe2, 0xa0, 0xf3,
   0xde, 0x6a, 0xdc, 0x44, 0x9c, 0x79, 0xb2, 0xff,
   0xc9, 0xaa, 0x42, 0x4d, 0x75, 0x53, 0x39, 0xaf,
   0x91, 0x84, 0x51, 0x9b, 0xc7, 0x5b, 0xbc, 0x36,
   0x5b, 0xbe, 0x47, 0xd4, 0x8b, 0x25, 0x4c, 0xa1,
   0xf6, 0x0f, 0x8a, 0x35, 0x45, 0x24, 0x23, 0x48,
   0x1a, 0xda, 0x84, 0xf9, 0x33, 0x67, 0x55, 0x24,
   0xf1, 0xff, 0xe0, 0x28, 0x5c, 0x8c, 0x85, 0x28,
   0xf1, 0xfc, 0x3d, 0x31, 0xb2, 0x38, 0x24, 0x79,
   0x1b, 0x80, 0x44, 0xca, 0xd9, 0x25, 0x87, 0xa1,
   0xba, 0x7f, 0xba, 0x40, 0x01, 0x1c, 0x7a, 0xc1,
   0x09, 0xe6, 0x37, 0xc0, 0xd3, 0x8d, 0xc5, 0xc4,
   0x81, 0xad, 0xc9, 0xa2, 0x86, 0x93, 0xb5, 0x50,
   0x29, 0xd8, 0x03, 0x8b, 0x76, 0xd7, 0x94, 0x65,
   0x7a, 0x8c, 0x85, 0xad, 0xd6, 0xf7, 0x83, 0x64,
   0x86, 0x5a, 0x53, 0xc4, 0xa8, 0x56, 0x87, 0xa1,
   0xb3, 0xd9, 0x8c, 0xed, 0xb8, 0x10, 0x91, 0xdf,
   0xbc, 0xb4, 0x64, 0xa8, 0x7c, 0x51, 0xf6, 0xaa,
   0x47, 0x62, 0xbe, 0x01, 0xa4, 0x10, 0x4d, 0x4a,
   0x9a, 0xf1, 0x0c, 0xb6, 0xd0, 0xde, 0xb2, 0x78,
   0x5b, 0xd8, 0x65, 0x6f, 0x6e, 0xf8, 0x12, 0x20,
   0xac, 0x3f, 0x1b, 0x6e, 0x3a, 0x0d, 0xed, 0x84,
   0xde, 0x5e, 0x23, 0x83, 0x9e, 0xd9, 0x6d, 0x05
   };
   unsigned char group 16_responder[512] = {
   0xe1, 0x1e, 0x9b, 0x32, 0x93, 0x44, 0xc0, 0xac,
   0xc2, 0x27, 0x6c, 0x08, 0xdc, 0x7f, 0xe7, 0x7b,
```

```
   0xa5, 0x21, 0xaa, 0x31, 0xc3, 0xd5, 0x45, 0xe2,
   0x8c, 0xd5, 0x01, 0x4f, 0x1c, 0x33, 0xba, 0x5e,
   0x28, 0x4e, 0x85, 0xd8, 0x2a, 0x88, 0x2f, 0x93,
   0x1d, 0x5e, 0x00, 0x2f, 0xc1, 0x4a, 0xc2, 0x17,
   0x76, 0x0f, 0xfb, 0xfd, 0x8b, 0xf7, 0xbc, 0x52,
   0x3a, 0x04, 0x32, 0x9e, 0xd7, 0xdf, 0xf2, 0x32,
   0x57, 0x15, 0xe1, 0xd4, 0x36, 0xaf, 0xd0, 0xb7,
   0xfc, 0xb3, 0x00, 0x11, 0x57, 0xf4, 0x85, 0xed,
   0x85, 0x47, 0xc9, 0xe8, 0xca, 0x89, 0x67, 0x95,
   0x44, 0x7b, 0x98, 0x38, 0xed, 0x6e, 0xb8, 0xc6,
   0x8b, 0xb0, 0xf5, 0x15, 0xde, 0xaf, 0x5b, 0x19,
   0xe2, 0x8f, 0xde, 0x85, 0x47, 0xdd, 0x36, 0xd2,
   0xf4, 0x49, 0xbd, 0x06, 0x75, 0xaa, 0x74, 0x7c,
   0xc9, 0x0d, 0xc2, 0x10, 0x3c, 0xf6, 0x0d, 0xe1,
   0x7a, 0x3f, 0x06, 0x8d, 0x98, 0x98, 0x9b, 0x21,
   0xdf, 0x30, 0xf6, 0xa6, 0x3d, 0x72, 0x59, 0x69,
   0x3b, 0x9f, 0xad, 0x82, 0x60, 0x8e, 0xf0, 0xa6,
   0x2c, 0xa6, 0x3c, 0x94, 0x1e, 0x1c, 0xe6, 0x8a,
   0xf2, 0xef, 0x66, 0xa9, 0x89, 0x80, 0x82, 0x5e,
   0x41, 0x51, 0xf4, 0x6e, 0xdd, 0xeb, 0x23, 0x67,
   0x42, 0x80, 0x28, 0xab, 0x8a, 0xf5, 0x04, 0xa1,
   0xae, 0x63, 0xdf, 0xa7, 0x8f, 0xdf, 0x91, 0x50,
   0x1d, 0x38, 0x52, 0xb3, 0x8b, 0x45, 0x9d, 0x91,
   0x91, 0xba, 0x07, 0x0b, 0xce, 0xd6, 0xb2, 0xa2,
   0xfc, 0xca, 0x16, 0x43, 0xae, 0x63, 0xf6, 0xc2,
   0xb3, 0xac, 0x44, 0x78, 0x88, 0xbe, 0xd1, 0x69,
   0xbb, 0x93, 0x40, 0x6f, 0x11, 0x83, 0xfa, 0x33,
   0xc4, 0xb4, 0x4b, 0x66, 0xda, 0xa3, 0x30, 0x1b,
   0x5d, 0x21, 0xc8, 0x3f, 0xde, 0xc5, 0xce, 0x2b,
   0x01, 0xd1, 0x4e, 0xcb, 0xa5, 0xe5, 0x42, 0x12,
   0xea, 0x48, 0xd1, 0x5c, 0x27, 0xf9, 0x94, 0x82,
   0x52, 0x8d, 0xe6, 0xbf, 0x67, 0x3e, 0xbd, 0xbb,
   0xea, 0xe7, 0x3c, 0x85, 0xf3, 0xcf, 0x8a, 0xd8,
   0x1f, 0x5c, 0x33, 0x90, 0x9b, 0x2c, 0x2a, 0xf1,
   0x29, 0x89, 0x1e, 0x42, 0x39, 0xef, 0xc0, 0xca,
   0x96, 0x3a, 0x8e, 0xc9, 0x73, 0xb2, 0xa8, 0x95,
   0xcb, 0x61, 0xc7, 0xa6, 0xac, 0x55, 0xb4, 0xef,
   0x71, 0x3c, 0x6e, 0xfd, 0x40, 0xe8, 0x19, 0x5b,
   0x2d, 0x66, 0x90, 0x8b, 0xa0, 0x8c, 0x56, 0xe4,
   0xaa, 0x10, 0xd5, 0xca, 0xed, 0x5e, 0x41, 0x19,
   0x57, 0x2b, 0xbd, 0x93, 0xf4, 0xc5, 0xaf, 0x47,
   0xf1, 0x61, 0xd1, 0xdd, 0xef, 0x3a, 0x73, 0xdd,
   0x28, 0xd0, 0xa9, 0xf1, 0x3b, 0x59, 0x85, 0x34,
   0x4a, 0xda, 0x1d, 0xa4, 0xf6, 0x57, 0x76, 0x04,
   0x88, 0x75, 0x68, 0xb2, 0x1b, 0xc4, 0xef, 0x1a,
   0x2c, 0x2d, 0x72, 0xa4, 0xbf, 0xfc, 0x62, 0x6e,
   0xe8, 0x3f, 0x07, 0x6f, 0x49, 0x62, 0x2d, 0x3b,
   0xca, 0x61, 0xeb, 0x9a, 0x85, 0xb0, 0x1f, 0x2b,
```

```
0x00, 0xb6, 0x59, 0x21, 0x9d, 0xc1, 0x91, 0xd8,
0x20, 0x0d, 0x83, 0x2c, 0xfa, 0x67, 0xd5, 0x5a,
0x1d, 0xa5, 0xdf, 0xc5, 0xae, 0x4b, 0x79, 0x41,
0x34, 0x18, 0x5b, 0xff, 0xa9, 0x07, 0x3c, 0x35,
0x92, 0x5a, 0x2b, 0x1a, 0x13, 0x5a, 0x2c, 0x88,
0x4c, 0x5b, 0x87, 0xee, 0x19, 0xc5, 0xca, 0xf9,
0x2b, 0x4c, 0x44, 0x59, 0x8b, 0x1f, 0x65, 0x48,
0x49, 0xbc, 0xb5, 0xf0, 0x02, 0x96, 0xb5, 0x18,
0xc5, 0x58, 0x01, 0x5c, 0xf3, 0x26, 0x39, 0x7b,
0x35, 0x74, 0x20, 0x0c, 0xcb, 0x86, 0x8d, 0x70,
0xfc, 0xce, 0xdf, 0x88, 0x7f, 0xe9, 0x6b, 0xc7,
0x08, 0x2d, 0x17, 0xea, 0x72, 0x6e, 0xbc, 0xdd,
0xc8, 0xfe, 0x62, 0x7c, 0x8f, 0x9a, 0x5e, 0xad,
0x47, 0x60, 0xb6, 0xa1, 0x82, 0x1a, 0xf9, 0xcc
};
```

A.2.4. Role-specific Elements for 8192-bit FFC group

```
unsigned char group 18_initiator[1024] = {
0x42, 0x5b, 0x57, 0x35, 0x57, 0xed, 0x1c, 0x14,
0xcd, 0x91, 0x34, 0x61, 0x75, 0x67, 0x88, 0x52,
0xf9, 0x10, 0x44, 0xad, 0x3c, 0xbf, 0x83, 0x2b,
0xd7, 0x94, 0x70, 0x7e, 0xe2, 0x79, 0x72, 0xfd,
0x44, 0xdb, 0xc2, 0xb5, 0x21, 0xae, 0x4a, 0x78,
0xad, 0x45, 0x09, 0xa6, 0x3c, 0x79, 0x07, 0x09,
0x65, 0x57, 0xf2, 0xac, 0x81, 0x90, 0xe9, 0x77,
0x3e, 0x7a, 0xd4, 0xbf, 0x56, 0x81, 0x35, 0x41,
0x31, 0x21, 0x8a, 0xfb, 0x03, 0xa2, 0xe0, 0x01,
0x27, 0x9b, 0x07, 0x45, 0x35, 0xcc, 0x84, 0x7e,
0xcc, 0x7b, 0x01, 0xb6, 0x80, 0xd9, 0x2e, 0x1e,
0xa3, 0x09, 0xf3, 0x15, 0x47, 0xf5, 0x37, 0x0d,
0xb0, 0x22, 0x39, 0x5a, 0xd1, 0xb3, 0xf5, 0x11,
0x5c, 0x63, 0x08, 0x8e, 0x80, 0xde, 0x08, 0xe2,
0xf5, 0xbc, 0xbb, 0xae, 0x21, 0xb5, 0xed, 0x2c,
0x7b, 0xa9, 0xdf, 0x54, 0xf3, 0x3a, 0xd5, 0x0e,
0x34, 0x33, 0x97, 0xae, 0x7f, 0x35, 0x67, 0x4e,
0x29, 0xca, 0x1d, 0xe6, 0xea, 0x04, 0x23, 0xad,
0x8f, 0x1e, 0xe3, 0xeb, 0xd2, 0x55, 0xc1, 0x02,
0x2e, 0x95, 0x4f, 0xd9, 0x97, 0x17, 0xd9, 0x7f,
0x31, 0xca, 0xf7, 0xa8, 0xa6, 0x59, 0x44, 0x44,
0xd2, 0x3f, 0xbe, 0x71, 0xb6, 0x87, 0xe8, 0x07,
0x84, 0x0d, 0x46, 0x7b, 0x24, 0x56, 0x74, 0x06,
0xcd, 0x46, 0x34, 0x6a, 0x73, 0x18, 0xbc, 0xbb,
0x57, 0x5e, 0xb1, 0x8d, 0xf5, 0xc7, 0xb6, 0x85,
0xdd, 0x14, 0x8a, 0x74, 0x15, 0xf1, 0x23, 0xda,
0xd3, 0xac, 0xfc, 0x59, 0x61, 0x0a, 0x78, 0x3b,
0x5a, 0x93, 0x8e, 0x10, 0x59, 0x74, 0x7c, 0xa8,
0x2c, 0x97, 0x50, 0xf0, 0x44, 0x73, 0x03, 0xad,
```

```
   0xb4, 0xb8, 0x1a, 0x2b, 0xa7, 0x54, 0xdb, 0x33,
   0xd3, 0x82, 0xda, 0x8b, 0x93, 0x39, 0x70, 0xe4,
   0x1a, 0x3a, 0x88, 0xc4, 0x9f, 0x62, 0x90, 0x3a,
   0xeb, 0x32, 0x07, 0x80, 0x64, 0x95, 0xf2, 0x9e,
   0xf1, 0xb5, 0xed, 0xcf, 0x78, 0x1a, 0x44, 0x96,
   0x48, 0xb5, 0x40, 0xc9, 0x0a, 0x46, 0xa6, 0xcb,
   0xb5, 0x85, 0xf2, 0x63, 0x5c, 0x0c, 0x4e, 0x77,
   0x06, 0xc1, 0x44, 0x5f, 0xd9, 0x0b, 0xeb, 0x14,
   0xa4, 0x74, 0x14, 0x57, 0x55, 0x5b, 0xad, 0xb2,
   0x92, 0x53, 0x0a, 0x10, 0x54, 0x61, 0xae, 0x95,
   0x2f, 0x54, 0x83, 0xfe, 0x22, 0x67, 0x3e, 0xe3,
   0x99, 0x06, 0x4e, 0x0c, 0x6a, 0x4d, 0xd0, 0xcb,
   0x82, 0xc1, 0x67, 0xfe, 0xb2, 0x9b, 0xde, 0xc4,
   0x0e, 0x19, 0x97, 0x59, 0xc8, 0xe3, 0x75, 0xc2,
   0xf1, 0xd6, 0xff, 0xf6, 0xca, 0x84, 0x4c, 0x40,
   0xa4, 0x41, 0xd0, 0xf6, 0x09, 0x99, 0x6d, 0x94,
   0x14, 0x80, 0xe2, 0x0a, 0x44, 0x5f, 0xf4, 0xce,
   0xe3, 0xc9, 0x3f, 0xff, 0x13, 0xdc, 0x90, 0xce,
   0x35, 0x21, 0xa9, 0x83, 0x59, 0xa3, 0x67, 0x3c,
   0xa4, 0x84, 0x3f, 0x82, 0x70, 0x19, 0xf1, 0x84,
   0x1a, 0x3f, 0xba, 0x71, 0xa3, 0x73, 0x0b, 0xa2,
   0x80, 0x1d, 0x45, 0xa9, 0xf0, 0xba, 0x4b, 0x72,
   0x7b, 0xc0, 0x16, 0x36, 0x37, 0x3e, 0x39, 0x2e,
   0xcb, 0x5e, 0x1b, 0x03, 0x6d, 0xab, 0xd1, 0xd0,
   0x24, 0x6f, 0x0f, 0x35, 0x83, 0x8f, 0xd1, 0x1b,
   0x0b, 0xb2, 0x69, 0xcf, 0x78, 0x50, 0xeb, 0x52,
   0xd6, 0x66, 0x01, 0xc9, 0x50, 0xa8, 0x11, 0x4d,
   0x2b, 0xf7, 0x95, 0x43, 0xe1, 0x44, 0x2c, 0x19,
   0xa3, 0x9e, 0xc6, 0x71, 0xdc, 0x76, 0x47, 0x1d,
   0xb8, 0x53, 0xab, 0xed, 0x28, 0x01, 0xdd, 0x6b,
   0x3b, 0xe2, 0x19, 0xce, 0xf9, 0x9b, 0xac, 0x9b,
   0xba, 0x50, 0x93, 0x6f, 0x90, 0xb3, 0x5a, 0x58,
   0xbf, 0x92, 0x2a, 0x30, 0x35, 0xe8, 0x0f, 0x23,
   0xc1, 0x8c, 0x86, 0x94, 0x89, 0xd2, 0x7c, 0x64,
   0x60, 0x32, 0xa6, 0x30, 0xdd, 0x50, 0xf3, 0x47,
   0xc2, 0x59, 0xb9, 0xa1, 0xf0, 0xa7, 0x7e, 0xb1,
   0x08, 0xea, 0xfb, 0x72, 0x7e, 0x24, 0xe0, 0x75,
   0x8e, 0x0e, 0xbf, 0xa0, 0x89, 0xa0, 0x73, 0x23,
   0x85, 0x37, 0xd6, 0xad, 0x67, 0x08, 0x8d, 0x4e,
   0x1c, 0x81, 0xdc, 0x3c, 0xd9, 0x69, 0x4a, 0x26,
   0x81, 0x4d, 0xb6, 0x4c, 0x70, 0x3b, 0xf9, 0x43,
   0xf9, 0x2e, 0xd7, 0xba, 0x24, 0x82, 0xc7, 0x67,
   0xac, 0xc4, 0xbe, 0x14, 0xf7, 0xdf, 0xd0, 0x6e,
   0xa0, 0x70, 0x0d, 0xff, 0x31, 0x59, 0xc7, 0xf6,
   0x43, 0x5f, 0x32, 0x94, 0xd1, 0xf5, 0x9c, 0x7c,
   0xff, 0x55, 0xc4, 0xf0, 0x43, 0x22, 0xe2, 0xb1,
   0x58, 0x83, 0xa7, 0x7e, 0x00, 0x15, 0xee, 0xe1,
   0xff, 0xe8, 0x81, 0xbc, 0xb1, 0xfc, 0x3d, 0xc6,
```

```
   0x5e, 0x12, 0xbe, 0xa2, 0x71, 0x82, 0x34, 0xbc,
   0xb9, 0x7a, 0xe5, 0x22, 0xc9, 0xe6, 0x13, 0x62,
   0x7a, 0xb3, 0x85, 0xec, 0xe3, 0x6d, 0xd0, 0xb6,
   0x44, 0x8c, 0x62, 0x3e, 0x06, 0x49, 0x77, 0x9d,
   0x90, 0x62, 0x19, 0x0f, 0x1e, 0xd3, 0x6a, 0x2b,
   0x9b, 0xe3, 0xf1, 0x9b, 0xc5, 0xc3, 0x81, 0x38,
   0x3f, 0x40, 0x26, 0x08, 0x0c, 0x4e, 0xfa, 0x0e,
   0x41, 0xb0, 0x4a, 0x6f, 0x85, 0x6f, 0x49, 0x94,
   0x01, 0x90, 0x8d, 0x02, 0x4b, 0x22, 0x54, 0x71,
   0xbb, 0x2b, 0xab, 0xb6, 0x95, 0xf7, 0x51, 0x53,
   0x27, 0x6c, 0x5a, 0x8e, 0x10, 0x03, 0x64, 0x63,
   0xf4, 0x2f, 0x40, 0x94, 0x66, 0xc7, 0x59, 0xa9,
   0xba, 0xd1, 0x4d, 0xa6, 0x8c, 0x55, 0x82, 0x25,
   0xa6, 0x3b, 0xb5, 0x92, 0xc1, 0x81, 0xf6, 0x2f,
   0x9d, 0x6d, 0xc0, 0x4c, 0x98, 0xd0, 0x82, 0x78,
   0xa5, 0xac, 0xba, 0xee, 0x33, 0x47, 0xa3, 0x49,
   0x00, 0xdd, 0x13, 0x09, 0x41, 0xaf, 0x52, 0xe3,
   0x5f, 0xe8, 0xc2, 0x66, 0x22, 0x53, 0x3c, 0xd9,
   0x17, 0xe6, 0x57, 0x0c, 0x49, 0xc0, 0xda, 0x45,
   0xc0, 0x61, 0x1c, 0x25, 0xd2, 0xa9, 0x90, 0x82,
   0x7e, 0x6b, 0x4a, 0xc1, 0xd2, 0xa3, 0x86, 0x0a,
   0x73, 0x8b, 0x42, 0x3f, 0xc9, 0x29, 0x70, 0xda,
   0xff, 0x29, 0x50, 0xa5, 0x25, 0x9f, 0xdd, 0xef,
   0x03, 0x2a, 0x79, 0x62, 0xf6, 0x55, 0xe8, 0x01,
   0xc0, 0x15, 0xb3, 0xb6, 0xb3, 0xad, 0x53, 0xd8,
   0x7e, 0xea, 0xef, 0xa5, 0x0e, 0xbd, 0x97, 0xfc,
   0xac, 0x15, 0xa5, 0x91, 0x4a, 0xc8, 0x9a, 0x4f,
   0x60, 0x0e, 0x64, 0xa4, 0x85, 0x8d, 0x85, 0x0a,
   0x6a, 0xd5, 0xe4, 0x67, 0x0a, 0x3a, 0x5b, 0x0e,
   0xe7, 0xc3, 0xf5, 0x76, 0x34, 0x8e, 0x47, 0x15,
   0x7b, 0x0f, 0xd0, 0x3b, 0xee, 0xd5, 0x9b, 0xa3,
   0x9a, 0x01, 0xb5, 0x90, 0x9d, 0xe1, 0xf2, 0xa2,
   0x35, 0xdd, 0x0b, 0xd8, 0x1d, 0xd7, 0xd6, 0x8f,
   0x34, 0x5d, 0x69, 0x9b, 0xc3, 0xae, 0x29, 0xcf,
   0x99, 0xf4, 0x94, 0x7f, 0x35, 0x92, 0x09, 0x21,
   0x93, 0x35, 0xba, 0x25, 0x97, 0x9a, 0xc5, 0x6c,
   0x64, 0xbe, 0xb1, 0x0a, 0x90, 0x4f, 0x3c, 0x16,
   0xe5, 0x59, 0x07, 0xb4, 0x6e, 0x4b, 0x50, 0x54,
   0x97, 0x53, 0xa5, 0x87, 0x9c, 0x4d, 0xb5, 0x96,
   0x76, 0xd1, 0x7e, 0x3d, 0xd1, 0x60, 0xb0, 0x14,
   0xc3, 0x43, 0xba, 0xdb, 0x2d, 0x16, 0x94, 0xdf,
   0xc0, 0x97, 0x84, 0x58, 0xb4, 0xdc, 0xd3, 0x02,
   0x83, 0xb8, 0x04, 0x94, 0xda, 0x66, 0x2f, 0x1e,
   0xf6, 0xe1, 0x82, 0x59, 0x1d, 0xfe, 0x93, 0xe6,
   0x92, 0xf7, 0x7d, 0xb9, 0x25, 0x97, 0xe2, 0x1c,
   0x2f, 0x3a, 0x42, 0xb6, 0xac, 0x46, 0x1a, 0x37,
   0xe8, 0xfd, 0x86, 0x51, 0xf2, 0x07, 0x46, 0xb3,
   0xb2, 0x71, 0xf6, 0xd0, 0x3d, 0x7e, 0x0a, 0x09,
```

```
   0x91, 0x96, 0xbb, 0xb6, 0x13, 0xa1, 0x04, 0xf4,
   0x5b, 0x82, 0xe9, 0x69, 0xf1, 0xfd, 0xab, 0x06,
   0x42, 0xda, 0xa9, 0x6a, 0xb7, 0x64, 0x30, 0x91
   };
   unsigned char group 18_responder[1024] = {
   0xdc, 0x33, 0x0f, 0xaf, 0x4a, 0x8f, 0x0d, 0x35,
   0xad, 0x20, 0x14, 0xfb, 0x37, 0x88, 0xee, 0xeb,
   0xdb, 0x71, 0x4d, 0x4b, 0x2a, 0x1d, 0xff, 0x5e,
   0xe7, 0x78, 0x2c, 0xa4, 0xc7, 0x6d, 0x4d, 0xb0,
   0x39, 0xb3, 0xbf, 0xc4, 0x2e, 0xe6, 0x30, 0x66,
   0x3d, 0x52, 0x2d, 0xf1, 0xce, 0x44, 0x6e, 0x1e,
   0x89, 0x85, 0x97, 0x4b, 0xdc, 0x50, 0x0c, 0xaf,
   0x59, 0xc8, 0xaf, 0x7f, 0xbb, 0x67, 0xb3, 0x68,
   0xf3, 0xf9, 0x10, 0x30, 0xde, 0x27, 0x2d, 0x83,
   0xc5, 0x9a, 0xa9, 0xc1, 0x06, 0x36, 0xb0, 0xdd,
   0xe5, 0x55, 0x1b, 0xbf, 0x7c, 0xbf, 0x2d, 0xca,
   0x8d, 0x84, 0x4d, 0x55, 0x44, 0x28, 0xe4, 0xcc,
   0xb4, 0xb8, 0x26, 0xd5, 0x11, 0x91, 0xdb, 0xf8,
   0x1b, 0x57, 0x17, 0xad, 0x3e, 0x2b, 0xee, 0x7a,
   0x7a, 0xdc, 0xc5, 0x46, 0xc9, 0x88, 0xf1, 0x39,
   0x9e, 0xf3, 0xca, 0x6d, 0x09, 0x11, 0xe4, 0xcc,
   0x31, 0x03, 0x80, 0xd1, 0x7d, 0xc8, 0xf7, 0xc5,
   0x10, 0x78, 0x5d, 0x15, 0xa8, 0xe3, 0x55, 0xd2,
   0x01, 0x54, 0x96, 0x99, 0xaf, 0x18, 0x33, 0x8c,
   0x71, 0xf9, 0x25, 0x7a, 0xc0, 0xa5, 0xfd, 0x61,
   0xf1, 0x04, 0xc5, 0x22, 0xfa, 0xa6, 0xdd, 0xc1,
   0x13, 0xf4, 0x2b, 0x39, 0xa8, 0x17, 0x8d, 0x68,
   0xb0, 0xe9, 0x70, 0x12, 0xda, 0x60, 0x47, 0x2b,
   0x17, 0xf4, 0x14, 0x53, 0xad, 0x29, 0x1c, 0xa5,
   0x07, 0x43, 0x91, 0xe9, 0xfb, 0xf4, 0x5d, 0x4c,
   0xe5, 0xb8, 0x07, 0x27, 0x37, 0x03, 0x39, 0xf8,
   0x28, 0x2d, 0xab, 0x2f, 0x5a, 0x1d, 0x41, 0xa3,
   0x38, 0x2e, 0x42, 0xe6, 0xe2, 0x32, 0xf9, 0x75,
   0xca, 0x19, 0x80, 0x0d, 0xd1, 0x15, 0x73, 0x45,
   0xda, 0x8a, 0x67, 0x7a, 0x3c, 0xfd, 0x6b, 0x2d,
   0x46, 0xa4, 0xd0, 0xd2, 0x8d, 0x12, 0x2d, 0x54,
   0x5d, 0x1d, 0xa7, 0xc3, 0x44, 0x98, 0x4f, 0x6d,
   0x83, 0xbf, 0x33, 0xf8, 0x51, 0xf2, 0x29, 0xa3,
   0x48, 0x26, 0x43, 0x26, 0xfa, 0x3a, 0x4a, 0x48,
   0x6a, 0xac, 0x0d, 0x2c, 0xb5, 0x89, 0xec, 0xff,
   0xc3, 0x6f, 0x28, 0x54, 0xe6, 0x54, 0x35, 0x5f,
   0x93, 0xb7, 0x9e, 0xfa, 0x04, 0x1b, 0x31, 0x5d,
   0xe0, 0x58, 0x4a, 0x8d, 0x54, 0xb9, 0x63, 0x72,
   0x4a, 0x68, 0x5f, 0x9d, 0x1b, 0xde, 0xdf, 0xbb,
   0xae, 0x9b, 0xb4, 0x65, 0x91, 0x93, 0x91, 0x9f,
   0xd9, 0xb5, 0xbc, 0x41, 0x32, 0xd3, 0x37, 0x95,
   0xb1, 0x0e, 0xec, 0xe5, 0x18, 0x38, 0xf9, 0xbe,
   0xc9, 0xf9, 0xc1, 0x5c, 0x18, 0x9e, 0xd0, 0x56,
```

```
   0x35, 0xe1, 0xf2, 0xd1, 0xeb, 0x09, 0x18, 0xc4,
   0xe3, 0x56, 0x10, 0x47, 0x3c, 0xb4, 0x8b, 0xcc,
   0xf0, 0xab, 0x4c, 0xcd, 0xdb, 0x6c, 0xa2, 0x39,
   0xb9, 0x32, 0xee, 0x57, 0x33, 0x9e, 0x9b, 0xc8,
   0x30, 0xa1, 0x60, 0x3f, 0xd0, 0xdb, 0xf3, 0xb5,
   0x14, 0x9f, 0xab, 0x9e, 0xaf, 0xd6, 0x88, 0x12,
   0x26, 0xaf, 0xf6, 0x3b, 0x4c, 0x20, 0xf9, 0xae,
   0xa4, 0x85, 0x7b, 0x07, 0x8a, 0x1e, 0x10, 0x90,
   0x29, 0x9e, 0xba, 0x63, 0x54, 0x3f, 0x0e, 0xbe,
   0x95, 0x39, 0x22, 0x11, 0xe8, 0xff, 0xda, 0x08,
   0xd6, 0x6e, 0xb3, 0xd5, 0xcc, 0xbd, 0x4f, 0x8a,
   0x1d, 0x37, 0x0f, 0x9b, 0x6d, 0xda, 0x9d, 0xd0,
   0x46, 0x4d, 0x0f, 0x57, 0x06, 0x9f, 0x15, 0x53,
   0x52, 0x1b, 0xfe, 0x2c, 0x52, 0xc4, 0xab, 0x80,
   0x29, 0xce, 0x64, 0x1a, 0x7a, 0xd2, 0x98, 0x56,
   0xcc, 0x90, 0xed, 0x6d, 0x1e, 0x9c, 0xe8, 0x5d,
   0xcf, 0x55, 0x89, 0x14, 0x3d, 0x38, 0x46, 0x68,
   0x67, 0xbe, 0x09, 0x9c, 0x9a, 0xdd, 0x74, 0xeb,
   0xde, 0xc8, 0x65, 0x71, 0x1a, 0xd7, 0x35, 0xa5,
   0xee, 0xf9, 0xbd, 0xe0, 0xf9, 0x04, 0xb5, 0x8e,
   0xc9, 0x42, 0xc4, 0xa4, 0x5e, 0x2c, 0xa6, 0x4e,
   0x19, 0x8d, 0x45, 0x24, 0x28, 0x02, 0xbe, 0x7f,
   0xea, 0xd7, 0xc1, 0x99, 0x04, 0x72, 0xf5, 0xb3,
   0x06, 0x2d, 0xaf, 0xf6, 0x37, 0x08, 0x22, 0x82,
   0xd7, 0xbb, 0x5a, 0xea, 0x72, 0x23, 0xce, 0xa8,
   0x2e, 0x73, 0x76, 0x9c, 0x3b, 0xa7, 0x23, 0xfa,
   0x82, 0x9f, 0xd8, 0x6b, 0x75, 0x57, 0xab, 0x5e,
   0x49, 0xcb, 0x29, 0x55, 0xe5, 0x55, 0xb1, 0x40,
   0x41, 0xa0, 0x36, 0xdc, 0xff, 0xc5, 0xd3, 0x8a,
   0xa8, 0x1d, 0xd2, 0x15, 0x9b, 0xc5, 0x84, 0xb7,
   0xea, 0x88, 0x85, 0xfa, 0x82, 0x75, 0x16, 0xbf,
   0x7f, 0xc9, 0x8f, 0xa0, 0x76, 0x90, 0x8c, 0x99,
   0x7e, 0xd9, 0x8d, 0xd6, 0x88, 0x08, 0x43, 0xd3,
   0x50, 0x98, 0x06, 0xda, 0x78, 0x9a, 0xdd, 0x71,
   0x7a, 0x61, 0xd2, 0x4c, 0xc8, 0xf0, 0xc1, 0x52,
   0x1c, 0x09, 0x7f, 0xe7, 0xba, 0x4e, 0x11, 0x39,
   0x06, 0xb8, 0xe9, 0xa1, 0xb0, 0x12, 0x9b, 0x6b,
   0xda, 0x90, 0x5e, 0x24, 0x54, 0x54, 0x87, 0xbb,
   0x69, 0x07, 0x5d, 0xf2, 0x65, 0xb3, 0xf8, 0x7e,
   0xea, 0xa5, 0xe5, 0x3c, 0xe9, 0x4b, 0x31, 0x47,
   0x79, 0x44, 0x74, 0x3b, 0x96, 0x1e, 0x1c, 0xbd,
   0x8a, 0xb2, 0x6f, 0x89, 0xd1, 0x60, 0xb8, 0x06,
   0xa7, 0x05, 0x1e, 0xe0, 0x6b, 0x26, 0xa1, 0xd2,
   0x85, 0xe9, 0x0a, 0x7d, 0x92, 0x26, 0xdd, 0xb0,
   0xa9, 0x51, 0x9b, 0x5d, 0xfa, 0x0a, 0x19, 0xe4,
   0x0d, 0xdb, 0xfb, 0x94, 0x60, 0x89, 0x3e, 0x49,
   0x10, 0x75, 0x5c, 0xa0, 0x1e, 0x52, 0xad, 0xf9,
   0x06, 0x04, 0x15, 0x91, 0xb0, 0x42, 0x67, 0x3a,
```

```
   0x1e, 0x43, 0x87, 0xdc, 0x06, 0x4c, 0x54, 0x37,
   0x41, 0xbf, 0x6f, 0x5b, 0xd5, 0xc1, 0xd9, 0x7b,
   0xdc, 0x25, 0x78, 0xa6, 0x6c, 0x56, 0x91, 0x0d,
   0x57, 0x4e, 0x6c, 0x1f, 0xc7, 0xab, 0xec, 0x53,
   0xbd, 0x4a, 0x1e, 0xee, 0x3e, 0x5e, 0x74, 0xe8,
   0x4c, 0x37, 0x46, 0xe3, 0xa7, 0x55, 0xce, 0x16,
   0x35, 0x6b, 0xbe, 0x43, 0x9d, 0xef, 0x7c, 0x6b,
   0x77, 0xb9, 0xc7, 0xda, 0x16, 0x6d, 0x7e, 0x4a,
   0x95, 0x09, 0x33, 0x70, 0x37, 0xe0, 0xe2, 0x44,
   0xad, 0x6b, 0xa2, 0x44, 0xe7, 0x96, 0x07, 0x18,
   0x19, 0xf1, 0x88, 0x2d, 0x47, 0x36, 0x6e, 0x57,
   0x48, 0xa3, 0x7a, 0x3b, 0x00, 0x70, 0x76, 0x52,
   0xe3, 0xd4, 0xa5, 0xac, 0x8d, 0x37, 0x2c, 0xd1,
   0xbe, 0x5c, 0x7c, 0x6c, 0xda, 0x86, 0x0b, 0x02,
   0x86, 0xcc, 0xbf, 0x44, 0x46, 0x69, 0xca, 0x86,
   0xa6, 0x9d, 0x98, 0xd5, 0xd7, 0x84, 0x57, 0xa5,
   0x90, 0x63, 0x49, 0x14, 0x48, 0x03, 0x25, 0x33,
   0x8f, 0xd5, 0xc2, 0x71, 0xfa, 0x6c, 0xf0, 0x9b,
   0x9e, 0xf0, 0x2a, 0xb2, 0x0c, 0x2d, 0x31, 0x5e,
   0xda, 0x33, 0x88, 0x72, 0xe7, 0x5a, 0x56, 0xbc,
   0xfd, 0x63, 0x70, 0xf6, 0xa3, 0xc2, 0x46, 0xec,
   0x57, 0x53, 0xfa, 0x09, 0x7d, 0x61, 0xc1, 0x56,
   0xa5, 0xee, 0x57, 0x47, 0x81, 0x9b, 0x7f, 0x7b,
   0x33, 0xdf, 0x20, 0xf6, 0x84, 0x24, 0x6e, 0xb6,
   0x29, 0xcc, 0xe5, 0x9c, 0x3e, 0x23, 0xd3, 0x0a,
   0x29, 0xf4, 0x46, 0x68, 0xb2, 0x88, 0xcc, 0x22,
   0x95, 0x70, 0xd3, 0x36, 0x0f, 0x60, 0xcd, 0xf1,
   0x0d, 0xfa, 0xcd, 0x22, 0x22, 0x8a, 0x30, 0x6b,
   0xc4, 0x44, 0x46, 0xe8, 0xf6, 0xad, 0x8f, 0x32,
   0x9a, 0x05, 0x30, 0x94, 0xbb, 0x47, 0x52, 0xd3,
   0x6f, 0x42, 0xe1, 0x6f, 0x50, 0xd1, 0x63, 0x0c,
   0x74, 0x6f, 0x75, 0x02, 0x40, 0x29, 0xab, 0x74,
   0xeb, 0x61, 0xb8, 0x09, 0xe2, 0x16, 0xf0, 0x48,
   0x8e, 0xd0, 0xf3, 0x9e, 0x11, 0x6c, 0x6e, 0xf1,
   0xe8, 0x73, 0x8f, 0x38, 0xba, 0x9c, 0x9b, 0x16,
   0xc0, 0xdf, 0x33, 0xb7, 0x29, 0x7b, 0xa6, 0x39,
   0xa5, 0x86, 0x7d, 0xda, 0xb7, 0xf9, 0x9e, 0x88
   };
```

Author’s Address

Dan Harkins
HP Enterprise
3333 Scott boulevard
Santa Clara, California 95054
United States of America

Phone: +1 415 997 9834
Email: dharkins@lounge.org

Network Working Group
Internet-Draft
Intended status: Informational
Expires: February 15, 2018

P. Hoffman
ICANN

August 14, 2017

The Transition from Classical to Post-Quantum Cryptography
draft-hoffman-c2pq-02

Abstract

Quantum computing is the study of computers that use quantum features in calculations. For over 20 years, it has been known that if very large, specialized quantum computers could be built, they could have a devastating effect on asymmetric classical cryptographic algorithms such as RSA and elliptic curve signatures and key exchange, as well as (but in smaller scale) on symmetric cryptographic algorithms such as block ciphers, MACs, and hash functions. There has already been a great deal of study on how to create algorithms that will resist large, specialized quantum computers, but so far, the properties of those algorithms make them onerous to adopt before they are needed.

Small quantum computers are being built today, but it is still far from clear when large, specialized quantum computers will be built that can recover private or secret keys in classical algorithms at the key sizes commonly used today. It is important to be able to predict when large, specialized quantum computers usable for cryptanalysis will be possible so that organization can change to post-quantum cryptographic algorithms well before they are needed.

This document describes quantum computing, how it might be used to attack classical cryptographic algorithms, and possibly how to predict when large, specialized quantum computers will become feasible.

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 February 15, 2018.

Table of Contents

1. Introduction

Early drafts of this document use "@@@@@" to indicate where the editor particularly want input from reviewers. The editor welcomes all types of review, but the areas marked with "@@@@@" are in the most noticeable need of new material. (The editor particularly appreciates new material that comes with references that can be included in this document as well.)

1.1. Disclaimer

**** This is an early version of this draft. **** As such, it has had little in-depth review in the cryptography community. Statements in this document might be wrong; given that the entire document is about cryptography, those wrong statements might have significant security problems associated with them.

Readers of this document should not rely on any statements in this version of this draft. As the draft gets more input from the cryptography community over time, this disclaimer will be softened and eventually eliminated.

1.2. Executive Summary

The development of quantum computers that can recover private or secret keys in classical algorithms at the key sizes commonly used today is at a very early stage. None of the published examples of such quantum computers is useful in recovering keys that are in use today. There is a great amount of interest in this development, and researchers expect large strides in this development in the coming decade.

There is active research in standardizing signing and key exchange algorithms that will withstand attacks from large, specialized quantum computers. However, all those algorithms to date have very large keys, very large signatures, or both. Thus, there is a large sustained cost in using those algorithms. Similarly, there is a large cost in being surprised about when quantum computers can cause damage to current cryptographic keys and signatures.

Because the world does not know when large, specialized quantum computers that can recover cryptographic keys will be available, organizations should be watching this area so that they have plenty of time to either change to larger key sizes for classical cryptography or to change to post-quantum algorithms. See Section 6 for a fuller discussion of determining how to predict when quantum computers that can harm current cryptography might become feasible.

1.3. Terminology

The term "classical cryptography" is used to indicate the cryptographic algorithms that are in common use today. In particular, signature and key exchange algorithms that are based on the difficulty of factoring numbers into two large prime numbers, or are based on the difficulty of determining the discrete log of a large composite number, are considered classical cryptography.

The term "post-quantum cryptography" refers to the invention and study of cryptographic mechanisms in which the security does not rely on computationally hard problems that can be efficiently solved on quantum computers. This excludes systems whose security relies on factoring numbers, or the difficulty of determining the discrete log of one group element with respect to another.

Note that these definitions apply to only one aspect of quantum computing as it relates to cryptography. It is expected that quantum computing will also be able to be used against symmetric key cryptography to make it possible to search for a secret symmetric key using far fewer operations than are needed using classical computers (see Section 5 for more detail). However, using longer keys to thwart that possibility is not normally called "post-quantum cryptography".

There are many terms that are only used in the field of quantum computing, such as "qubit", "quantum algorithm", and so on. Chapter 1 of [NielsenChuang] has good definitions of such terms.

Some papers discussing quantum computers and cryptanalysis say that large, specialized quantum computers "break" algorithms in classical cryptography. This paper does not use that terminology because the algorithms' strength will be reduced when large, specialized quantum computers exist, but not to the point where there is an immediate need to change algorithms.

The "^" symbol is used to indicate "the power of". The term "log" always means "logarithm base 2".

1.4. Not Covered: Post-Quantum Cryptographic Algorithms

This document discusses when an organization would want to consider using post-quantum cryptographic algorithms, but definitely does not delve into which of those algorithms would be best to use. Post-quantum cryptography is an active field of research; in fact, it is much more active than the study of when we might want to transition from classical to post-quantum cryptography.

Readers interested in post-quantum cryptographic algorithms will have no problem finding many articles proposing such algorithms, comparing the many current proposals, and so on. An excellent starting point is the web site <http://pqcrypto.org/>. The Open Quantum Safe (OQS) project <https://openquantumsafe.org/> is developing and prototyping quantum-resistant cryptography. Another is the article on post-quantum cryptography at Wikipedia: <https://en.wikipedia.org/wiki/Post-quantum_cryptography>.

Various organizations are working on standardizing the algorithms for post-quantum cryptography. For example, the US National Institute of Standards and Technology (commonly just called "NIST") is holding a competition to evaluate post-quantum cryptographic algorithms. NIST's description of that effort is currently at <http://csrc.nist.gov/groups/ST/post-quantum-crypto/>. Until recently, ETSI (the European Telecommunications Standards Institute) had a Quantum-Safe Cryptography (QSC) Industry Specification Group (ISG) that worked on specifying post-quantum algorithms; see <http://www.etsi.org/technologies-clusters/technologies/quantum-safe-cryptography> for results from this work.

1.5. Not Covered: Quantum Cryptography

Other than in this section, this document does not cover "quantum cryptography". The field of quantum cryptography uses quantum effects in order to secure communication between users. Quantum cryptography is not related to cryptanalysis. The best known and extensively studied example of quantum cryptography is a quantum key exchange, where users can share a secret key while preventing an eavesdropper from obtaining the key.

1.6. Where to Read More

There are many reasonably accessible articles on Wikipedia, notably the overview article at <https://en.wikipedia.org/wiki/Quantum_computing> and the timeline of quantum computing developments at <https://en.wikipedia.org/wiki/Timeline_of_quantum_computing>.

[NielsenChuang] is a well-regarded college textbook on quantum computers. Prerequisites for understanding the book include linear algebra and some quantum physics; however, even without those, a reader can probably get value from the introductory material in the book.

[Turing50Youtube] is a good overview of the near-term and longer-term prospects for designing and building quantum computers; it is a video of a panel discussion by quantum hardware and software experts given at the ACM's Turing 50 lecture.

@@@@@ Maybe add more references that might be useful to non-experts.

2. Brief Introduction to Quantum Computers

A quantum computer is a computer that uses quantum bits (qubits) in quantum circuits to perform calculations. Quantum computers also use classical bits and regular circuits: most calculations in a quantum computer are a mix of classical and quantum bits and circuits. For example, classical bits could be used for error correction or controlling the behavior of physical components of the quantum computer.

A basic principle that makes it possible to speed up calculations on qubits in quantum computers is quantum superposition. Informally, similarly to waves in classical physics, arbitrary number of quantum states can be added together and result will be another valid quantum state. That means that, for example, two qubits could be in any quantum superposition of four states, three qubits in quantum superposition of eight states, and so on. Generally n qubits can be in quantum superposition of 2^n states.

The main challenge for quantum computing is to create and maintain a significantly large number of superposed qubits while performing quantum computations. Physical components of quantum computers that are non-ideal results in the destruction of qubit state over time; this is the source of errors in quantum computation. See Section 3.1 for a description of how to overcome this problem.

A good description of different aspects of calculations on quantum computer could be found in [EstimatingPreimage].

A separate question is a measurement of a quantum state. Due to uncertainty of the state, the measurement process is stochastic. That means that in order to get the correct measurement one should run several consequent calculations and corresponding measurement in order to the expected value which is considered as a result of measurement.

@@@@@ Discuss measurements and how they have to be done with correlated qubits.

2.1. Quantum Computers that Recover Cryptographic Keys

Quantum computers are expected to be useful in the future for some problems that take up too many resources on a large classical computer. However, this document only discusses how they might recover cryptographic keys faster than classical computers. In order to recover cryptographic keys, a quantum computer needs to have a quantum circuit specifically designed for the type of key it is attempting to recover.

A quantum computer will need to have a circuit with thousands of qubits to be useful to recover the type and size keys that are in common use today. Smaller quantum computers (those with fewer qubits in superposition) are not useful for using Shor's algorithm (as discussed in Section 4.1) at all. That is, no one has devised a way to combine a bunch of smaller quantum computers to perform the same attacks on cryptographic keys via Shor's algorithm as a properly-sized quantum computer.

This is why this document uses the term "large, specialized quantum computer" when describing ones that can recover keys: there will certainly be small quantum computers built first, but those computers cannot recover the type and size keys that are in common use today. Further, there are already quantum computers that have many qubits but without the circuits needed to make those qubits useful for recovering cryptographic keys.

A straight-forward application of Shor's algorithm may not be the only way for large, specialized quantum computers to attack RSA keys. [LowResource] describes how to combine quantum computers with classical methods for recovering RSA keys at speeds faster than just using the classical methods.

3. Physical Designs for Quantum Computers

Quantum computers can be built using many different physical technologies. Deciding which physical technologies are best to pursue is an extremely active research topic. A few physical technologies (particularly trapped ions, super-conduction using Josephson junctions, and nuclear magnetic resonance) are currently getting the most press, but other technologies are also showing promise.

One factor that is important to quantum computers that can be used for cryptanalysis is the speed of the operations (transformations) on

qubits. Most of the estimates of speeds of these quantum computers assume that qubit operations will take about the same amount of time as operations in circuits that consist of classical gates and classical memory. Current quantum circuits are slower than classical circuits, but will certainly become faster as quantum computers are developed in the future.

Note that some current quantum computer research uses bits that are not fully entangled, and this will greatly affect their ability to make useful quantum calculations.

3.1. Qubits, Error Detection, and Error Correction

Researchers building small quantum computers have discovered that calculating the superposition of qubits often has a large rate of error, and that error rate increases rapidly over time. Performing quantum calculations such as those needed to recover cryptographic keys is not feasible with the current state of quantum computers.

In the future, actual quantum calculations will be performed on "logical qubits", that is, after the application of error correction codes on physical qubits. Thus, the number of physical qubits will be higher than the number of logical qubits, depending on the parameters of the error correction code, which in turn depends on the parameters of a technology used for a physical implementation of qubits. Currently, it is estimated that it takes hundreds or thousands of physical qubits to make a logical qubit. @@@@@ Need reference for this statement.

@@@@@ Lots more material should go here. We will need recent references for how many physical qubits are needed for each corrected qubit. It’s OK if this section has lots of references, but hopefully they don’t contradict each other.

3.2. Promising Physical Designs for Quantum Computers

@@@@@ It would be useful to have maybe two paragraphs about each physical design that is being actively pursued.

3.3. Challenges for Physical Designs

Different designs have different challenges to overcome before the physical technology can be scaled enough to build a useful large, specialized quantum computer. Some of those challenges include the following. (Note that some items on this list apply only to some of the physical technologies.)

Temperature: Getting stable operation without extreme cooling is difficult for many of the proposed technologies. The definition of "extreme" is different for different low-temperature technologies.

Stabilization: The length of time every qubit in a circuit holds is value

Quantum control: Coherence and reproducibility of qubits

Error detection and correction: Getting accurate results through simultaneous detection of bit-flip and phase-flip. See Section 3.1 for a longer description of this.

Substrate: The material on which the qubit circuits are built. This has a large effect on the stability of the qubits.

Particles: The atoms or sub-atomic particles used to make the qubits

Scalability: The ability to handle the number of physical qubits needed for the desired the circuit

Architecture: Ability to change quantum gates in a circuit

4. Quantum Computers and Public Key Cryptography

The area of quantum computing that has generated the most interest in the cryptographic community is the ability of quantum computers to find the private keys in encryption and signature algorithms based on discrete logarithms using exponentially fewer operations than classical computers would need to use.

As described in [RFC3766], it is widely believed that factoring large numbers and finding discrete logs using classical computers increases with the exponential size of the key. [RFC3766] describes in detail how classical computers can be used to determine keys; even though that RFC is over a decade old, no significant changes have been made to the process of classical attacks on RSA and Diffie-Hellman. @@@@@ CFRG: is that true? Does RFC 3766 need to be updated?

Shor’s algorithm shows that these problems can be solved on quantum computers in polynomial time, meaning that the speed of finding the keys is a polynomial function (with reasonable-sized coefficients) based on the size of the keys, which would require significantly fewer steps than a classical computer. The definitive paper on Shor’s algorithm is [Shor97].

4.1. Explanation of Shor's Algorithm

@@@@@ Pointers to understandable articles would be good here.

@@@@@ Describe period-finding and why it applies to finding prime factors and discrete logs.

@@@@@ Give the steps for applying Shor's algorithm to 2048-bit RSA. Describe how many rounds of the quantum subroutine would likely be needed. Describe how many rounds of the classical loop would likely be needed.

[ResourceElliptic] gives concrete estimates of the resources needed to build a quantum computer to compute elliptic curve discrete logarithms. It shows that for the common P-256 elliptic curve, 2330 logical qubits and over 10^11 Toffoli gates.

4.2. Properties of Large, Specialized Quantum Computers Needed for Recovering RSA Public Keys

Researchers have built small quantum computers that implement Shor's algorithm, factoring numbers with four or five bits. These are used to show that Shor's algorithm is possible to realize in actual hardware. (Note, however, that [PretendingFactor] indicates that these experiments may have taken shortcuts that prevent them from indicating real Shor designs.)

@@@@@ References are needed here. Did they implement all of Shor's algorithm, including the looping logic in the classical part and the looping logic in the quantum part?

@@@@@ Numbers and explanation is needed below:

A quantum computer that can determine the private keys for 2048-bit RSA would require SOME NUMBER GOES HERE correlated qubits and SOME NUMBER GOES HERE circuit elements. A quantum computer that can determine the private keys for 256-bt elliptic curves would require SOME NUMBER GOES HERE correlated qubits and SOME NUMBER GOES HERE circuit elements.

5. Quantum Computers and Symmetric Key Cryptography

Section 4 is about Shor's algorithm and compromises to public key cryptography. There is a second quantum computing algorithm, Grover's algorithm, that is often mentioned at the same time as Shor's algorithm. With respect to cryptanalysis, however, Grover's algorithm applies to tasks of finding a preimage, including tasks of finding a secret key of a symmetric algorithm such as AES if there is

knowledge of plaintext-ciphertext pairs. The definitive paper on Grover's algorithm is by Grover: [Grover96]. Grover later wrote a more accessible paper about the algorithm in [QuantumSearch].

Grover's algorithm gives a way to search for keys to symmetric algorithms in the square root of the time that a normal exhaustive search would take. Thus, a large, specialized quantum computer that implements Grover's algorithm could find a secret AES-128 key in about 2^64 steps instead of the 2^128 steps that would be required for a classical computer.

When it appears that it is feasible to build a large, specialized quantum computer that can defeat a particular symmetric algorithm at a particular key size, the proper response would be to use keys with twice as many bits. That is, if one is using the AES-128 algorithm and there is a concern that an adversary might be able to build a large, specialized quantum computer that is designed to attack AES-128 keys, move to an algorithm that has keys twice as long as AES-128, namely AES-256 (the block size used is not significant here).

It is currently expected that large, specialized quantum computers that implement Grover's algorithm are expected to be built long before ones that implement Shor's algorithm are. There are two primary reasons for this:

- Grover's algorithm is likely to be useful in areas other than cryptography. For example, a large, specialized quantum computer that implements Grover's algorithm might help create medicines by speeding up complex problems that involve how proteins fold. @@@@@ Add more likely examples and references here.

- A large, specialized quantum computer that can recover AES-128 keys will likely be much smaller (and thus easier to build) than one that implements Shor's algorithm for 256-bit elliptic curves or 2048-bit RSA/DSA keys.

5.1. Explanation of Grover's Algorithm

@@@@@ Give the steps for applying Grover's algorithm to AES-128.

5.2. Properties of Large, Specialized Quantum Computers Needed for Recovering Symmetric Keys

[ApplyingGrover] estimates that a quantum computer that can determine the secret keys for AES-128 would require 2953 correlated qubits and 2.74 * 2^86 gates.

5.3. Properties of Large, Specialized Quantum Computers for Computing Hash Collisions

@@@@@ More goes here. Also, discuss how Grover's algorithm does not appear to be useful for computing preimages (or say how it might be used.

6. Predicting When Useful Cryptographic Attacks Will Be Feasible

If quantum computers that perform useful cryptographic attacks can be built in the future, many organizations will want to start using post-quantum algorithms well before those computers can be built. However, given how few implementations of such quantum computers exist (even for tiny keys), it is impossible to predict with any accuracy when quantum computers that perform useful cryptographic attacks will be feasible.

The term "useful" above is relative to the value of the material being protected by the cryptographic algorithm to the attacker. For example, if the quantum computer attacking a particular key costs US$100 billion to build, costs US$1 billion a year to run, and can extract only one key a year, it is possibly useful to some governments, but probably not useful for attacking the TLS key used to protect a small mail server. On the other hand, if later a similar computer costs US$1 billion to build, costs US$10 million a year to run, and can extract ten keys a year, many more keys become vulnerable.

[BeReady] gives a simple way to approach the calculation of when one needs to deploy post-quantum algorithms. In short, if the sum of how long you need your keys to be secure plus how long it takes to deploy new algorithms is longer than the length of time it will take for an attacker to create a large, specialized quantum computer and use it against your keys, then you waited too long.

To date, few people have done systematic research that would give estimates for when useful quantum-based cryptographic attacks might be feasible, and at what cost. Without such research, it is easy to make wild guesses but those are not of much value to people having to decide when to start using post-quantum cryptography.

For example, in [NIST8105], NIST says "researchers working on building a quantum computer have estimated that it is likely that a quantum computer capable of recovering 2000-bit RSA in a matter of hours could be built by 2030 for a budget of about a billion dollars". However, the referenced link is to a YouTube video [MariantoniYoutube] where the researcher, Matteo Mariantoni, says "maybe you should not quote me on that". [NIST8105] gives no other

references for predictions on cost and availability of useful cryptographic attacks with quantum computers.

6.1. Proposal: Public Measurements of Various Quantum Technologies

In order to get a rough idea of when useful cryptographic attacks with quantum computers may be feasible, researchers creating such computers can demonstrate them when they can recover keys an eighth the size of those in common use. That is, given that 2048-bit RSA, 256-bit elliptic curve, and AES-128 are common today, when a research team has a computer than can recover 256-bit RSA, 32-bit elliptic curve, or AES-128 where only 16 bits are unknown, they should demonstrate it.

Such a demonstration could easily be made fair with trusted representatives from the cryptographic community using verifiable means to pick the keys to recover, and verifying the time that it takes to recover each key. It might be interesting to run the same tests in classical computers at the same time to give perspective.

These demonstrations will have many benefits to those who have to decide when post-quantum algorithms should be deployed in various environments.

- Demonstrations will likely use designs that are considered most efficient. This in turn will cause greater focus research on choosing good design candidates.

- The results of the demonstrations will help focus on issues important to cryptanalysis, namely the cost of building the systems and the speed of breaking a single key.

- Competing demonstrations will reveal where different research teams have made different optimizations from well-known designs.

- Public demonstrations could expose designs that work only in limited cases that are uncommon in normal cryptographic practice. (For example, [PretendingFactor] claims that all current factorization experiments have taken advantage of using a classical computer that already knows the answer to design the quantum circuits.)

Note that this proposal would only give an idea of how public progress is being made on quantum computers. Well-funded military agencies (and possibly even criminal enterprises) could be way ahead of the publicly-visible computers. No one should rely on just the public measurements when deciding how safe their keys are against quantum computers.

7. IANA Considerations

None, and thus this section can be removed at final publication.

8. Security Considerations

This entire document is about cryptography, and thus about security.

See Section 1.1 for an important disclaimer about this document and security.

This document is meant to help the reader predict when to transition from using classical cryptographic algorithms to post-quantum algorithms. That decision is ultimately up to the reader, and must be made not only based on predictions of how quantum computing is progressing but also the value of every key that the user handles. For example, a financial institution using TLS to protect its customers' transactions will probably consider its keys more valuable than a small online store, and will thus be likely to begin the transition earlier.

9. Acknowledgements

The list here is meant to acknowledge input to this document. The people listed here do not necessarily agree with ideas presented.

Many sections of text were contributed by Grigory Marshalko and Stanislav Smyshlyaev.

Some of the ideas in this document come from Denis Butin, Philip Lafrance, Hilarie Orman, and Tomofumi Okubo.

10. References

10.1. Normative References

[Grover96] Grover, L., "A fast quantum mechanical algorithm for database search", 1996, <https://arxiv.org/abs/quant-ph/9605043>.

[Shor97] Shor, P., "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", 1997, <http://epubs.siam.org/doi/pdf/10.1137/S0097539795293172>.

10.2. Informative References

[ApplyingGrover]
Grassl, M., Langenberg, B., Roetteler, M., and R. Steinwandt, "Applying Grover's algorithm to AES: quantum resource estimates", 2015, <https://arxiv.org/abs/1512.04965>.

[BeReady] Mosca, M., "Cybersecurity in an era with quantum computers: will we be ready?", 2015, <http://eprint.iacr.org/2015/1075>.

[EstimatingPreimage]
Amy, M., Di Matteo, O., Gheorghiu, V., Mosca, M., Parent, A., and J. Schanck, "Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3", 2016, <https://eprint.iacr.org/2016/992>.

[LowResource]
Bernstein, D., Fiassse, J., and M. Mosca, "A low-resource quantum factoring algorithm", 2017, <https://eprint.iacr.org/2017/352.pdf>.

[MariantoniYoutube]
Mariantoni, M., "Building a Superconducting Quantum Computer", 2014, <https://www.youtube.com/watch?v=wWHAs--HA1c>.

[NielsenChuang]
Nielsen, M. and I. Chuang, "Quantum Computation and Quantum Information, 10th Anniversary Edition", ISBN 97801-107-00217-3 , 2010.

[NIST8105]
Chen, L. and et. al, "Report on Post-Quantum Cryptography", 2016, <http://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf>.

[PretendingFactor]
Smolin, J., Vargo, A., and J. Smolin, "Pretending to factor large numbers on a quantum computer", 2013, <https://arxiv.org/abs/1301.7007>.

[QuantumSearch]
Grover, L., "From Schrodinger's Equation to the Quantum Search Algorithm", 2001, <https://arxiv.org/abs/quant-ph/0109116>.

[ResourceElliptic]
Roetteler, M., Naehrig, M., Svore, K., and K. Lauter, "Quantum Resource Estimates for Computing Elliptic Curve Discrete Logarithms", 2017, <https://eprint.iacr.org/2017/598>.

[RFC3766] Orman, H. and P. Hoffman, "Determining Strengths For Public Keys Used For Exchanging Symmetric Keys", BCP 86, RFC 3766, DOI 10.17487/RFC3766, April 2004, <http://www.rfc-editor.org/info/rfc3766>.

[Turing50Youtube]
Vazirani, U., Aharonov, D., Gambetta, J., Martinis, J., and A. Yao, "Quantum Computing: Far Away? Around the Corner?", 2017, <https://www.youtube.com/watch?v=SzfJRR5JrgQ>.

Author’s Address

Paul Hoffman
ICANN

Email: paul.hoffman@icann.org

Network Working Group
Internet-Draft
Intended status: Informational
Expires: November 22, 2019

P. Hoffman
ICANN

May 21, 2019

The Transition from Classical to Post-Quantum Cryptography
draft-hoffman-c2pq-05

Abstract

Quantum computing is the study of computers that use quantum features in calculations. For over 20 years, it has been known that if very large, specialized quantum computers could be built, they could have a devastating effect on asymmetric classical cryptographic algorithms such as RSA and elliptic curve signatures and key exchange, as well as (but in smaller scale) on symmetric cryptographic algorithms such as block ciphers, MACs, and hash functions. There has already been a great deal of study on how to create algorithms that will resist large, specialized quantum computers, but so far, the properties of those algorithms make them onerous to adopt before they are needed.

Small quantum computers are being built today, but it is still far from clear when large, specialized quantum computers will be built that can recover private or secret keys in classical algorithms at the key sizes commonly used today. It is important to be able to predict when large, specialized quantum computers usable for cryptanalysis will be possible so that organization can change to post-quantum cryptographic algorithms well before they are needed.

This document describes quantum computing, how it might be used to attack classical cryptographic algorithms, and possibly how to predict when large, specialized quantum computers will become feasible.

Status of This Memo

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 November 22, 2019.

Table of Contents

1. Introduction

Early drafts of this document use "@@@@@" to indicate where the editor particularly want input from reviewers. The editor welcomes all types of review, but the areas marked with "@@@@@" are in the most noticeable need of new material. (The editor particularly appreciates new material that comes with references that can be included in this document as well.)

1.1. Disclaimer

**** This is still an early version of this draft. **** As such, it has had only some review in the cryptography community. Statements in this document might be wrong; given that the entire document is about cryptography, those wrong statements might have significant security problems associated with them.

Readers of this document should not rely on any statements in this version of this draft. As the draft gets more input from the cryptography community over time, this disclaimer will be softened and eventually eliminated.

1.2. Executive Summary

The development of quantum computers that can recover private or secret keys in classical algorithms at the key sizes commonly used today is at a very early stage. None of the published examples of such quantum computers is useful in recovering keys that are in use today. There is a great amount of interest in this development, and researchers expect large strides in this development in the coming decade.

There is active research in standardizing signing and key exchange algorithms that will withstand attacks from large, specialized quantum computers. However, all those algorithms to date have very large keys, very large signatures, or both. Thus, there is a large sustained cost in using those algorithms. Similarly, there is a large cost in being surprised about when quantum computers can cause damage to current cryptographic keys and signatures.

Because the world does not know when large, specialized quantum computers that can recover cryptographic keys will be available, organizations should be watching this area so that they have plenty of time to either change to larger key sizes for classical cryptography or to change to post-quantum algorithms. See Section 6 for a fuller discussion of determining how to predict when quantum computers that can harm current cryptography might become feasible.

1.3. Terminology

The term "classical cryptography" is used to indicate the cryptographic algorithms that are in common use today. In particular, signature and key exchange algorithms that are based on the difficulty of factoring numbers into two large prime numbers, or are based on the difficulty of determining the discrete log of a large composite number, are considered classical cryptography.

The term "post-quantum cryptography" refers to the invention and study of cryptographic mechanisms in which the security does not rely on computationally hard problems that can be efficiently solved on quantum computers. This excludes systems whose security relies on factoring numbers, or the difficulty of determining the discrete log of one group element with respect to another.

Note that these definitions apply to only one aspect of quantum computing as it relates to cryptography. It is expected that quantum computing will also be able to be used against symmetric key cryptography to make it possible to search for a secret symmetric key using far fewer operations than are needed using classical computers (see Section 5 for more detail). However, using longer keys to thwart that possibility is not normally called "post-quantum cryptography".

There are many terms that are only used in the field of quantum computing, such as "qubit", "quantum algorithm", and so on. Chapter 1 of [NielsenChuang] has good definitions of such terms.

Some papers discussing quantum computers and cryptanalysis say that large, specialized quantum computers "break" algorithms in classical cryptography. This paper does not use that terminology because the algorithms' strength will be reduced when large, specialized quantum computers exist, but not to the point where there is an immediate need to change algorithms.

The "^" symbol is used to indicate "the power of". The term "log" always means "logarithm base 2".

1.4. Not Covered: Post-Quantum Cryptographic Algorithms

This document discusses when an organization would want to consider using post-quantum cryptographic algorithms, but definitely does not delve into which of those algorithms would be best to use. Post-quantum cryptography is an active field of research; in fact, it is much more active than the study of when we might want to transition from classical to post-quantum cryptography.

Readers interested in post-quantum cryptographic algorithms will have no problem finding many articles proposing such algorithms, comparing the many current proposals, and so on. An excellent starting point is the web site <http://pqcrypto.org/>. The Open Quantum Safe (OQS) project <https://openquantumsafe.org/> is developing and prototyping quantum-resistant cryptography. Another is the article on post-quantum cryptography at Wikipedia: <https://en.wikipedia.org/wiki/Post-quantum_cryptography>.

Various organizations are working on standardizing the algorithms for post-quantum cryptography. For example, the US National Institute of Standards and Technology (commonly just called "NIST") is holding a competition to evaluate post-quantum cryptographic algorithms. NIST's description of that effort is currently at <http://csrc.nist.gov/groups/ST/post-quantum-crypto/>. Until recently, ETSI (the European Telecommunications Standards Institute) had a Quantum-Safe Cryptography (QSC) Industry Specification Group (ISG) that worked on specifying post-quantum algorithms; see <http://www.etsi.org/technologies-clusters/technologies/quantum-safe-cryptography> for results from this work.

1.5. Not Covered: Quantum Cryptography

Other than in this section, this document does not cover "quantum cryptography". The field of quantum cryptography uses quantum effects in order to secure communication between users. Quantum cryptography is not related to cryptanalysis. The best known and extensively studied example of quantum cryptography is a quantum key exchange, where users can share a secret key while preventing an eavesdropper from obtaining the key.

1.6. Where to Read More

There are many reasonably accessible articles on Wikipedia, notably the overview article at <https://en.wikipedia.org/wiki/Quantum_computing> and the timeline of quantum computing developments at <https://en.wikipedia.org/wiki/Timeline_of_quantum_computing>.

[NielsenChuang] is a well-regarded college textbook on quantum computers. Prerequisites for understanding the book include linear algebra and some quantum physics; however, even without those, a reader can probably get value from the introductory material in the book.

A good overview of the current status of quantum computing in general is [ProgressProspects].

[QCPolicy] describes how the development of quantum computing affects encryption policies.

[Turing50Youtube] is a good overview of the near-term and longer-term prospects for designing and building quantum computers; it is a video of a panel discussion by quantum hardware and software experts given at the ACM's Turing 50 lecture.

@@@@@ Maybe add more references that might be useful to non-experts.

2. Brief Introduction to Quantum Computers

A quantum computer is a computer that uses quantum bits (qubits) in quantum circuits to perform calculations. Quantum computers also use classical bits and regular circuits: most calculations in a quantum computer are a mix of classical and quantum bits and circuits. For example, classical bits could be used for error correction or controlling the behavior of physical components of the quantum computer.

A basic principle that makes it possible to speed up calculations on qubits in quantum computers is quantum superposition. Informally, similarly to waves in classical physics, arbitrary number of quantum states can be added together and result will be another valid quantum state. That means that, for example, two qubits could be in any quantum superposition of four states, three qubits in quantum superposition of eight states, and so on. Generally n qubits can be in quantum superposition of 2^n states.

The main challenge for quantum computing is to create and maintain a significantly large number of superposed qubits while performing quantum computations. Physical components of quantum computers that are non-ideal results in the destruction of qubit state over time; this is the source of errors in quantum computation. See Section 3.1 for a description of how to overcome this problem.

A good description of different aspects of calculations on quantum computer could be found in [EstimatingPreimage].

A separate question is a measurement of a quantum state. Due to uncertainty of the state, the measurement process is stochastic. That means that in order to get the correct measurement one should run several consequent calculations and corresponding measurement in order to the expected value which is considered as a result of measurement.

@@@@@ Discuss measurements and how they have to be done with correlated qubits.

2.1. Quantum Computers that Recover Cryptographic Keys

Quantum computers are expected to be useful in the future for some problems that take up too many resources on a large classical computer. However, this document only discusses how they might recover cryptographic keys faster than classical computers. In order to recover cryptographic keys, a quantum computer needs to have a quantum circuit specifically designed for the type of key it is attempting to recover.

A quantum computer will need to have a circuit with thousands of qubits to be useful to recover the type and size keys that are in common use today. Smaller quantum computers (those with fewer qubits in superposition) are not useful for using Shor's algorithm (as discussed in Section 4.1) at all. That is, no one has devised a way to combine a bunch of smaller quantum computers to perform the same attacks on cryptographic keys via Shor's algorithm as a properly-sized quantum computer.

This is why this document uses the term "large, specialized quantum computer" when describing ones that can recover keys: there will certainly be small quantum computers built first, but those computers cannot recover the type and size keys that are in common use today. Further, there are already quantum computers that have many qubits but without the circuits needed to make those qubits useful for recovering cryptographic keys.

A straight-forward application of Shor's algorithm may not be the only way for large, specialized quantum computers to attack RSA keys. [LowResource] describes how to combine quantum computers with classical methods for recovering RSA keys at speeds faster than just using the classical methods.

3. Physical Designs for Quantum Computers

Quantum computers can be built using many different physical technologies. Deciding which physical technologies are best to pursue is an extremely active research topic. A few physical

technologies (particularly trapped ions and neutral atoms, super-conduction using Josephson junctions, and nuclear magnetic resonance) are currently getting the most press, but other technologies are also showing promise.

One factor that is important to quantum computers that can be used for cryptanalysis is the speed of the operations (transformations) on qubits. Most of the estimates of speeds of these quantum computers assume that qubit operations will take about the same amount of time as operations in circuits that consist of classical gates and classical memory. Current quantum circuits are currently slower than classical circuits, but will certainly become faster as quantum computers are developed in the future.

Note that some current quantum computer research uses bits that are not fully entangled, and this will greatly affect their ability to make useful quantum calculations.

3.1. Qubits, Error Detection, and Error Correction

Researchers building small quantum computers have discovered that calculating the superposition of qubits often has a large rate of error, and that error rate increases rapidly over time. Performing quantum calculations such as those needed to recover cryptographic keys is not feasible with the current state of quantum computers.

In the future, actual quantum calculations will be performed on "logical qubits", that is, after the application of error correction codes on physical qubits. Thus, the number of physical qubits will be higher than the number of logical qubits, depending on the parameters of the error correction code, which in turn depends on the parameters of a technology used for a physical implementation of qubits. Currently, it is estimated that it takes hundreds or thousands of physical qubits to make a logical qubit. @@@@@ Need reference for this statement.

@@@@@ Lots more material should go here. We will need recent references for how many physical qubits are needed for each corrected qubit. It’s OK if this section has lots of references, but hopefully they don’t contradict each other.

3.2. Promising Physical Designs for Quantum Computers

@@@@@ It would be useful to have maybe two paragraphs about each physical design that is being actively pursued.

3.3. Challenges for Physical Designs

Different designs have different challenges to overcome before the physical technology can be scaled enough to build a useful large, specialized quantum computer. Some of those challenges include the following. (Note that some items on this list apply only to some of the physical technologies.)

Temperature: Getting stable operation without extreme cooling is difficult for many of the proposed technologies. The definition of "extreme" is different for different low-temperature technologies.

Stabilization: The length of time every qubit in a circuit holds is value

Quantum control: Coherence and reproducibility of qubits

Error detection and correction: Getting accurate results through simultaneous detection of bit-flip and phase-flip. See Section 3.1 for a longer description of this.

Substrate: The material on which the qubit circuits are built. This has a large effect on the stability of the qubits.

Particles: The atoms or sub-atomic particles used to make the qubits

Scalability: The ability to handle the number of physical qubits needed for the desired the circuit

Architecture: Ability to change quantum gates in a circuit

4. Quantum Computers and Public Key Cryptography

The area of quantum computing that has generated the most interest in the cryptographic community is the ability of quantum computers to find the private keys in encryption and signature algorithms based on discrete logarithms using exponentially fewer operations than classical computers would need to use.

As described in [RFC3766], it is widely believed that factoring large numbers and finding discrete logs using classical computers increases with the exponential size of the key. [RFC3766] describes in detail how classical computers can be used to determine keys; even though that RFC is over a decade old, no significant changes have been made to the process of classical attacks on RSA and Diffie-Hellman. @@@@@ CFRG: is that true? Does RFC 3766 need to be updated?

Shor's algorithm shows that these problems can be solved on quantum computers in polynomial time, meaning that the speed of finding the keys is a polynomial function (with reasonable-sized coefficients) based on the size of the keys, which would require significantly fewer steps than a classical computer. The definitive paper on Shor's algorithm is [Shor97].

4.1. Explanation of Shor's Algorithm

@@@@@ Pointers to understandable articles would be good here.

@@@@@ Describe period-finding and why it applies to finding prime factors and discrete logs.

@@@@@ Give the steps for applying Shor's algorithm to 2048-bit RSA. Describe how many rounds of the quantum subroutine would likely be needed. Describe how many rounds of the classical loop would likely be needed.

[ResourceElliptic] gives concrete estimates of the resources needed to build a quantum computer to compute elliptic curve discrete logarithms. It shows that for the common P-256 elliptic curve, 2330 logical qubits and over 10^11 Toffoli gates.

[PrimeFactAnneal] describes a method of converting the integer factorization problem to one that can be executed on an adiabatic quantum computer. Adiabatic quantum computers are already available today, such as those from D-Wave Systems. Note that this method is not a way to run Shor's algorithm on an adiabatic quantum computer.

4.2. Properties of Large, Specialized Quantum Computers Needed for Recovering RSA Public Keys

Researchers have built small quantum computers that implement Shor's algorithm, factoring numbers with four or five bits. These are used to show that Shor's algorithm is possible to realize in actual hardware. (Note, however, that [PretendingFactor] indicates that these experiments may have taken shortcuts that prevent them from indicating real Shor designs.)

@@@@@ References are needed here. Did they implement all of Shor's algorithm, including the looping logic in the classical part and the looping logic in the quantum part?

@@@@@ Numbers and explanation is needed below:

A quantum computer that can determine the private keys for 2048-bit RSA would require SOME NUMBER GOES HERE correlated qubits and SOME

NUMBER GOES HERE circuit elements. A quantum computer that can determine the private keys for 256-bt elliptic curves would require SOME NUMBER GOES HERE correlated qubits and SOME NUMBER GOES HERE circuit elements.

5. Quantum Computers and Symmetric Key Cryptography

Section 4 is about Shor's algorithm and compromises to public key cryptography. There is a second quantum computing algorithm, Grover's algorithm, that is often mentioned at the same time as Shor's algorithm. With respect to cryptanalysis, however, Grover's algorithm applies to tasks of finding a preimage, including tasks of finding a secret key of a symmetric algorithm such as AES if there is knowledge of plaintext-ciphertext pairs. The definitive paper on Grover's algorithm is by Grover: [Grover96]. Grover later wrote a more accessible paper about the algorithm in [QuantumSearch].

Grover's algorithm gives a way to search for keys to symmetric algorithms in the square root of the time that a normal exhaustive search would take. Thus, a large, specialized quantum computer that implements Grover's algorithm could find a secret AES-128 key in about 2^64 steps instead of the 2^128 steps that would be required for a classical computer.

When it appears that it is feasible to build a large, specialized quantum computer that can defeat a particular symmetric algorithm at a particular key size, the proper response would be to use keys with twice as many bits. That is, if one is using the AES-128 algorithm and there is a concern that an adversary might be able to build a large, specialized quantum computer that is designed to attack AES-128 keys, move to an algorithm that has keys twice as long as AES-128, namely AES-256 (the block size used is not significant here).

It is currently expected that large, specialized quantum computers that implement Grover's algorithm may be built before ones that implement Shor's algorithm are. There are two primary reasons for this:

- Grover's algorithm is likely to be useful in areas other than cryptography. For example, a large, specialized quantum computer that implements Grover's algorithm might help create medicines by speeding up complex problems that involve how proteins fold. @@@@@ Add more likely examples and references here.

- A large, specialized quantum computer that can recover AES-128 keys will likely be much smaller (and thus easier to build) than

one that implements Shor's algorithm for 256-bit elliptic curves or 2048-bit RSA/DSA keys.

There are arguments against the likelihood of building computers using Grover's algorithm to break AES-128. As described in [FindCollisions]:

- Breaking AES-128 with Grover's method could be infeasible due to inherent inefficiencies in the algorithm. For example, the overhead of the quantum operations in the algorithm might be huge when compared to non-quantum operations.

- Grover's algorithm has not been parallelized in a quantum computer, so the 2^64 steps must be done serially. Unless the speed of quantum computations become as fast as current classical computers, this will make doing all the calculations needed to break an AES-128 key take so long as to be infeasible.

5.1. Properties of Large, Specialized Quantum Computers Needed for Recovering Symmetric Keys

[ApplyingGrover] estimates that a quantum computer that can determine the secret keys for AES-128 would require 2953 correlated qubits and 2.74 * 2^86 gates.

[GoverSDES] shows how to use Grover's algorithm to search for keys in SDES, a simplifed version of the DES encryption algorithm.

5.2. Properties of Large, Specialized Quantum Computers for Computing Hash Collisions

@@@@@ More goes here. Also, discuss how Grover's algorithm does not appear to be useful for computing preimages (or say how it might be used).

6. Predicting When Useful Cryptographic Attacks Will Be Feasible

If quantum computers that perform useful cryptographic attacks can be built in the future, many organizations will want to start using post-quantum algorithms well before those computers can be built. However, given how few implementations of such quantum computers exist (even for tiny keys), it is impossible to predict with any accuracy when quantum computers that perform useful cryptographic attacks will be feasible.

The term "useful" above is relative to the value of the material being protected by the cryptographic algorithm to the attacker. For example, if the quantum computer attacking a particular key costs

US$100 billion to build, costs US$1 billion a year to run, and can extract only one key a year, it is possibly useful to some governments, but probably not useful for attacking the TLS key used to protect a small mail server. On the other hand, if later a similar computer costs US$1 billion to build, costs US$10 million a year to run, and can extract ten keys a year, many more keys become vulnerable.

[BeReady] gives a simple way to approach the calculation of when one needs to deploy post-quantum algorithms. In short, if the sum of how long you need your keys to be secure plus how long it takes to deploy new algorithms is longer than the length of time it will take for an attacker to create a large, specialized quantum computer and use it against your keys, then you waited too long.

To date, few people have done systematic research that would give estimates for when useful quantum-based cryptographic attacks might be feasible, and at what cost. Without such research, it is easy to make wild guesses but those are not of much value to people having to decide when to start using post-quantum cryptography.

For example, in [NIST8105], NIST says "researchers working on building a quantum computer have estimated that it is likely that a quantum computer capable of recovering 2000-bit RSA in a matter of hours could be built by 2030 for a budget of about a billion dollars". However, the referenced link is to a YouTube video [MariantoniYoutube] where the researcher, Matteo Mariantoni, says "maybe you should not quote me on that". [NIST8105] gives no other references for predictions on cost and availability of useful cryptographic attacks with quantum computers.

6.1. Proposal: Public Measurements of Various Quantum Technologies

In order to get a rough idea of when useful cryptographic attacks with quantum computers may be feasible, researchers creating such computers can demonstrate them when they can recover keys an eighth the size of those in common use. That is, given that 2048-bit RSA, 256-bit elliptic curve, and AES-128 are common today, when a research team has a computer than can recover 256-bit RSA, 32-bit elliptic curve, or AES-128 where only 16 bits are unknown, they should demonstrate it.

Such a demonstration could easily be made fair with trusted representatives from the cryptographic community using verifiable means to pick the keys to recover, and verifying the time that it takes to recover each key. It might be interesting to run the same tests in classical computers at the same time to give perspective.

These demonstrations will have many benefits to those who have to decide when post-quantum algorithms should be deployed in various environments.

- Demonstrations will likely use designs that are considered most efficient. This in turn will cause greater focus research on choosing good design candidates.

- The results of the demonstrations will help focus on issues important to cryptanalysis, namely the cost of building the systems and the speed of breaking a single key.

- Competing demonstrations will reveal where different research teams have made different optimizations from well-known designs.

- Public demonstrations could expose designs that work only in limited cases that are uncommon in normal cryptographic practice. (For example, [PretendingFactor] claims that all current factorization experiments have taken advantage of using a classical computer that already knows the answer to design the quantum circuits.)

Note that this proposal would only give an idea of how public progress is being made on quantum computers. Well-funded military agencies (and possibly even criminal enterprises) could be way ahead of the publicly-visible computers. No one should rely on just the public measurements when deciding how safe their keys are against quantum computers.

7. IANA Considerations

None, and thus this section can be removed at final publication.

8. Security Considerations

This entire document is about cryptography, and thus about security.

See Section 1.1 for an important disclaimer about this document and security.

This document is meant to help the reader predict when to transition from using classical cryptographic algorithms to post-quantum algorithms. That decision is ultimately up to the reader, and must be made not only based on predictions of how quantum computing is progressing but also the value of every key that the user handles. For example, a financial institution using TLS to protect its customers' transactions will probably consider its keys more valuable

than a small online store, and will thus be likely to begin the transition earlier.

9. Acknowledgements

The list here is meant to acknowledge input to this document. The people listed here do not necessarily agree with ideas presented.

Many sections of text were contributed by Grigory Marshalko and Stanislav Smyshlyaev.

Some of the ideas in this document come from Denis Butin, Philip Lafrance, Hilarie Orman, and Tomofumi Okubo.

10. References

10.1. Normative References

[Grover96] Grover, L., "A fast quantum mechanical algorithm for database search", 1996, <https://arxiv.org/abs/quant-ph/9605043>.

[Shor97] Shor, P., "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", 1997, <http://epubs.siam.org/doi/pdf/10.1137/S0097539795293172>.

10.2. Informative References

[ApplyingGrover] Grassl, M., Langenberg, B., Roetteler, M., and R. Steinwandt, "Applying Grover's algorithm to AES: quantum resource estimates", 2015, <https://arxiv.org/abs/1512.04965>.

[BeReady] Mosca, M., "Cybersecurity in an era with quantum computers: will we be ready?", 2015, <http://eprint.iacr.org/2015/1075>.

[EstimatingPreimage] Amy, M., Di Matteo, O., Gheorghiu, V., Mosca, M., Parent, A., and J. Schanck, "Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3", 2016, <https://eprint.iacr.org/2016/992>.

[FindCollisions]
Bernstein, D., "Quantum algorithms to find collisions", 2017, <https://blog.cr.yp.to/20171017-collisions.html>.

[GoverSDES]
Denisenko, D. and M. Nikitenkova, "Application of Grover's Quantum Algorithm for SDES Key Searching", 2018.

[LowResource]
Bernstein, D., Fiassse, J., and M. Mosca, "A low-resource quantum factoring algorithm", 2017, <https://eprint.iacr.org/2017/352.pdf>.

[MariantoniYoutube]
Mariantoni, M., "Building a Superconducting Quantum Computer", 2014, <https://www.youtube.com/watch?v=wWHAs--HA1c>.

[NielsenChuang]
Nielsen, M. and I. Chuang, "Quantum Computation and Quantum Information, 10th Anniversary Edition", ISBN 97801-107-00217-3 , 2010.

[NIST8105]
Chen, L. and et. al, "Report on Post-Quantum Cryptography", 2016, <http://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf>.

[PretendingFactor]
Smolin, J., Vargo, A., and J. Smolin, "Pretending to factor large numbers on a quantum computer", 2013, <https://arxiv.org/abs/1301.7007>.

[PrimeFactAnneal]
Jiang, S., Britt, K., McCaskey, A., Humble, T., and S. Kais, "Quantum Annealing for Prime Factorization", 2018, <https://www.nature.com/articles/s41598-018-36058-z>.

[ProgressProspects]
The National Acadamies Sciences Engineering Medicine, "Quantum Computing: Progress and Prospects (2019)", n.d., <https://www.nap.edu/catalog/25196/quantum-computing-progress-and-prospects>.

[QCPolicy]
Princeton University Center for Information Technology Policy, "Implications of Quantum Computing for Encryption Policy", 2019, <https://carnegieendowment.org/2019/04/25/implications-of-quantum-computing-for-encryption-policy-pub-78985>.

[QuantumSearch]
Grover, L., "From Schrodinger's Equation to the Quantum Search Algorithm", 2001, <https://arxiv.org/abs/quant-ph/0109116>.

[ResourceElliptic]
Roetteler, M., Naehrig, M., Svore, K., and K. Lauter, "Quantum Resource Estimates for Computing Elliptic Curve Discrete Logarithms", 2017, <https://eprint.iacr.org/2017/598>.

[RFC3766] Orman, H. and P. Hoffman, "Determining Strengths For Public Keys Used For Exchanging Symmetric Keys", BCP 86, RFC 3766, DOI 10.17487/RFC3766, April 2004, <https://www.rfc-editor.org/info/rfc3766>.

[Turing50Youtube]
Vazirani, U., Aharonov, D., Gambetta, J., Martinis, J., and A. Yao, "Quantum Computing: Far Away? Around the Corner?", 2017, <https://www.youtube.com/watch?v=SzfJRR5JrgQ>.

Author's Address

Paul Hoffman
ICANN

Email: paul.hoffman@icann.org

CFRG
Internet-Draft
Intended status: Informational
Expires: April 12, 2018

S. Smyshlyaev, Ed.
CryptoPro

October 9, 2017

Re-keying Mechanisms for Symmetric Keys
draft-irtf-cfrg-re-keying-08

Abstract

A certain maximum amount of data can be safely encrypted when encryption is performed under a single key. This amount is called "key lifetime". This specification describes a variety of methods to increase the lifetime of symmetric keys. It provides external and internal re-keying mechanisms based on hash functions and on block ciphers, that can be used with modes of operations such as CTR, GCM, CBC, CFB and OMAC.

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 April 12, 2018.

Copyright Notice

Table of Contents

1. Introduction

A certain maximum amount of data can be safely encrypted when encryption is performed under a single key. This amount is called "key lifetime" and can be calculated from the following considerations:

1. Methods based on the combinatorial properties of the used block cipher mode of operation

   These methods do not depend on the underlying block cipher. Common modes restrictions derived from such methods are of order 2^{n/2}. [Sweet32] is an example of attack that is based on such methods.

2. Methods based on side-channel analysis issues

   In most cases these methods do not depend on the used encryption modes and weakly depend on the used block cipher features. Limitations resulting from these considerations are usually the most restrictive ones. [TEMPEST] is an example of attack that is based on such methods.

3. Methods based on the properties of the used block cipher

   The most common methods of this type are linear and differential cryptanalysis [LDC]. In most cases these methods do not depend on the used modes of operation. In case of secure block ciphers, bounds resulting from such methods are roughly the same as the natural bounds of 2^n, and are dominated by the other bounds above. Therefore, they can be excluded from the considerations here.

As a result, it is important to replace a key as soon as the total size of the processed plaintext under that key reaches the lifetime limitation. A specific value of the key lifetime should be determined in accordance with some safety margin for protocol security and the methods outlined above.

Suppose L is a key lifetime limitation in some protocol P. For simplicity, assume that all messages have the same length m. Hence, the number of messages q that can be processed with a single key K should be such that m * q <= L. This can be depicted graphically as a rectangle with sides m and q which is enclosed by area L (see Figure 1).

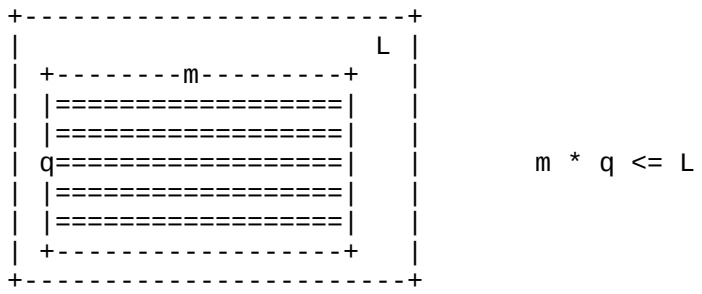


Figure 1: Graphic display of the key lifetime limitation

In practice, such amount of data that corresponds to limitation L may not be enough. The simplest and obvious way in this situation is a regular renegotiation of an initial key after processing this threshold amount of data L. However, this reduces the total performance, since it usually entails termination of application data transmission, additional service messages, the use of random number generator and many other additional calculations, including resource-intensive public key cryptography.

This specification presents two approaches to extend the lifetime of a key while avoiding renegotiation: external and internal re-keying.

External re-keying is performed by a protocol, and it is independent of the underlying block cipher and the mode of operation. External re-keying can use parallel and serial constructions. In the parallel case, data processing keys K^1, K^2, ... are generated directly from the initial key K independently of each other. In the serial case, every data processing key depends on the state that is updated after the generation of each new data processing key.

Internal re-keying is built into the mode, and it depends heavily on the properties of the mode of operation and the block size.

The re-keying approaches extend the key lifetime for a single initial key by providing the possibility to limit the leakages (via side channels) and by improving combinatorial properties of the used block cipher mode of operation.

In practical applications, re-keying can be useful for protocols that need to operate in hostile environments or under restricted resource conditions (e.g., that require lightweight cryptography, where ciphers have a small block size, that imposes strict combinatorial limitations). Moreover, mechanisms that use external and internal

re-keying may provide some properties of forward security and potentially some protection against future attacks (by limiting the number of plaintext-ciphertext pairs that an adversary can collect).

2. 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].

3. Basic Terms and Definitions

This document uses the following terms and definitions for the sets and operations on the elements of these sets:

V* the set of all bit strings of a finite length (hereinafter referred to as strings), including the empty string; substrings and string components are enumerated from right to left starting from one;

V_s the set of all bit strings of length s, where s is a non-negative integer;

|X| the bit length of the bit string X;

A | B concatenation of strings A and B both belonging to V*, i.e., a string in V_{|A|+|B|}, where the left substring in V_|A| is equal to A, and the right substring in V_|B| is equal to B;

(xor) exclusive-or of two bit strings of the same length;

Z_{2^n} ring of residues modulo 2^n;

Int_s: V_s -> Z_{2^s} the transformation that maps a string a = (a_s, ... , a_1), a in V_s, into the integer Int_s(a) = 2^{s-1} * a_s + ... + 2 * a_2 + a_1;

Vec_s: Z_{2^s} -> V_s the transformation inverse to the mapping Int_s;

MSB_i: V_s -> V_i the transformation that maps the string a = (a_s, ... , a_1) in V_s, into the string MSB_i(a) = (a_s, ... , a_{s-i+1}) in V_i;

LSB_i: V_s -> V_i the transformation that maps the string a = (a_s, ... , a_1) in V_s, into the string LSB_i(a) = (a_i, ... , a_1) in V_i;

Inc_c: V_s -> V_s the transformation that maps the string a = (a_s, ... , a_1) in V_s, into the string Inc_c(a) = MSB_{|a|-c}(a) | Vec_c(Int_c(LSB_c(a)) + 1(mod 2^c)) in V_s;

a^s denotes the string in V_s that consists of s 'a' bits;

E_{K}: V_n -> V_n the block cipher permutation under the key K in V_k;

ceil(x) the smallest integer that is greater than or equal to x;

floor(x) the biggest integer that is less than or equal to x;

k the bit-length of the K; k is assumed to be divisible by 8;

n the block size of the block cipher (in bits); n is assumed to be divisible by 8;

b the number of data blocks in the plaintext P (b = ceil(|P|/n));

N the section size (the number of bits that are processed with one section key before this key is transformed);

A plaintext message P and the corresponding ciphertext C are divided into b = ceil(|P|/n) blocks, denoted P = P_1 | P_2 | ... | P_b and C = C_1 | C_2 | ... | C_b, respectively. The first b-1 blocks P_i and C_i are in V_n, for i = 1, 2, ... , b-1. The b-th block P_b, C_b may be an incomplete block, i.e., in V_r, where r <= n if not otherwise specified.

4. Choosing Constructions and Security Parameters

External re-keying is an approach assuming that a key is transformed after encrypting a limited number of entire messages. External re-keying method is chosen at the protocol level, regardless of the underlying block cipher or the encryption mode. External re-keying is recommended for protocols that process relatively short messages or for protocols that have a way to divide a long message into manageable pieces. Through external re-keying the number of messages that can be securely processed with a single initial key K is substantially increased without loss in message length.

External re-keying has the following advantages:

1. it increases the lifetime of an initial key by increasing the number of messages processed with this key;

2. it has negligible affect on the performance, when the number of messages processed under one initial key is sufficiently large;

3. it provides forward and backward security of data processing keys.

However, the use of external re-keying has the following disadvantage: in case of restrictive key lifetime limitations the message sizes can become inconvenient due to impossibility of processing sufficiently large messages, so it could be necessary to perform additional fragmentation at the protocol level. E.g. if the key lifetime L is 1 GB and the message length m = 3 GB, then this message cannot be processed as a whole and it should be divided into three fragments that will be processed separately.

Internal re-keying is an approach assuming that a key is transformed during each separate message processing. Such procedures are integrated into the base modes of operations, so every internal re-keying mechanism is defined for the particular operation mode and the block size of the used cipher. Internal re-keying is recommended for protocols that process long messages: the size of each single message can be substantially increased without loss in number of messages that can be securely processed with a single initial key.

Internal re-keying has the following advantages:

1. it increases the lifetime of an initial key by increasing the size of the messages processed with one initial key;

2. it has minimal impact on performance;

3. internal re-keying mechanisms without a master key does not affect short messages transformation at all;

4. it is transparent (works like any mode of operation): does not require changes of IV’s and restarting MACing.

However, the use of internal re-keying has the following disadvantages:

1. a specific method must not be chosen independently of a mode of operation;

2. internal re-keying mechanisms without a master key do not provide backward security of data processing keys.

Any block cipher modes of operations with internal re-keying can be jointly used with any external re-keying mechanisms. Such joint

usage increases both the number of messages processed with one initial key and their maximum possible size.

If the adversary has access to the data processing interface the use of the same cryptographic primitives both for data processing and re-keying transformation decreases the code size but can lead to some possible vulnerabilities. This vulnerability can be eliminated by using different primitives for data processing and re-keying, however, in this case the security of the whole scheme cannot be reduced to standard notions like PRF or PRP, so security estimations become more difficult and unclear.

Summing up the above-mentioned issues briefly:

1. If a protocol assumes processing long records (e.g., [CMS]), internal re-keying should be used. If a protocol assumes processing a significant amount of ordered records, which can be considered as a single data stream (e.g., [TLS], [SSH]), internal re-keying may also be used.

2. For protocols which allow out-of-order delivery and lost records (e.g., [DTLS], [ESP]) external re-keying should be used. If at the same time records are long enough, internal re-keying should be additionally used during each separate message processing.

For external re-keying:

1. If it is desirable to separate transformations used for data processing and for key update, hash function based re-keying should be used.

2. If parallel data processing is required, then parallel external re-keying should be used.

3. In case of restrictive key lifetime limitations external tree-based re-keying should be used.

For internal re-keying:

1. If the property of forward and backward security is desirable for data processing keys and if additional key material can be easily obtained for the data processing stage, internal re-keying with a master key should be used.

5. External Re-keying Mechanisms

This section presents an approach to increase the initial key lifetime by using a transformation of a data processing key (frame key) after processing a limited number of entire messages (frame). It provides external parallel and serial re-keying mechanisms (see [AbBell]). These mechanisms use initial key K only for frame key generation and never use it directly for data processing. Such mechanisms operate outside of the base modes of operations and do not change them at all, therefore they are called "external re-keying" mechanisms in this document.

External re-keying mechanisms are recommended for usage in protocols that process quite small messages, since the maximum gain in increasing the initial key lifetime is achieved by increasing the number of messages.

External re-keying increases the initial key lifetime through the following approach. Suppose there is a protocol P with some mode of operation (base encryption or authentication mode). Let L1 be a key lifetime limitation induced by side-channel analysis methods (side-channel limitation), let L2 be a key lifetime limitation induced by methods based on the combinatorial properties of a used mode of operation (combinatorial limitation) and let q1, q2 be the total numbers of messages of length m, that can be safely processed with an initial key K according to these limitations.

Let L = min(L1, L2), q = min (q1, q2), q * m <= L. As L1 limitation is usually much stronger than L2 limitation (L1 < L2), the final key lifetime restriction is equal to the most restrictive limitation L1. Thus, as displayed in Figure 2, without re-keying only q1 (q1 * m <= L1) messages can be safely processed.

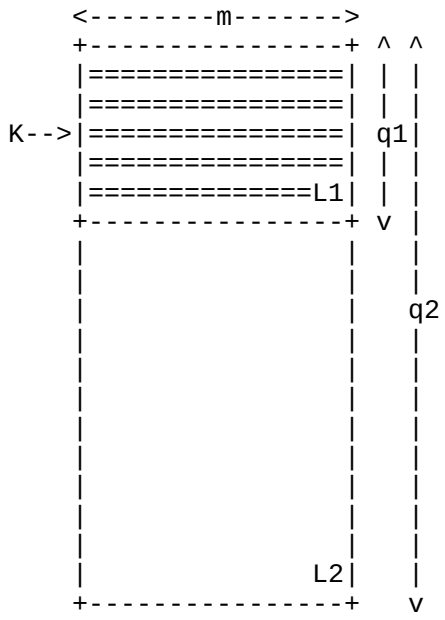


Figure 2: Basic principles of message processing without external re-keying

Suppose that the safety margin for the protocol P is fixed and the external re-keying approach is applied to the initial key K to generate the sequence of frame keys. The frame keys are generated in such a way that the leakage of a previous frame key does not have any impact on the following one, so the side channel limitation L1 goes off. Thus, the resulting key lifetime limitation of the initial key K can be calculated on the basis of a new combinatorial limitation L2'. It is proven (see [AbBell]) that the security of the mode of operation that uses external re-keying leads to an increase when compared to base mode without re-keying (thus, L2 < L2'). Hence, as displayed in Figure 3, the resulting key lifetime limitation in case of using external re-keying can be increased up to L2'.

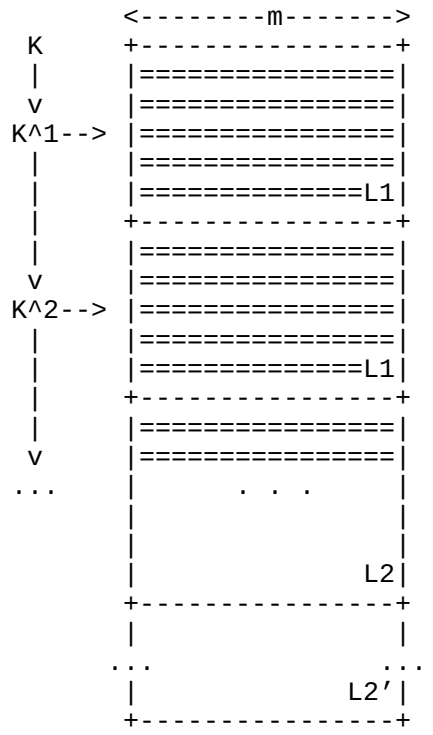


Figure 3: Basic principles of message processing with external re-keying

Note: the key transformation process is depicted in a simplified form. A specific approach (parallel and serial) is described below.

Consider an example. Let the message size in a protocol P be equal to 1 KB. Suppose L1 = 128 MB and L2 = 1 TB. Thus, if an external re-keying mechanism is not used, the initial key K must be renegotiated after processing 128 MB / 1 KB = 131072 messages.

If an external re-keying mechanism is used, the key lifetime limitation L1 goes off. Hence the resulting key lifetime limitation L2' can be set to more then 1 TB. Thus if an external re-keying mechanism is used, more then 1 TB / 1 KB = 2^30 messages can be processed before the initial key K is renegotiated. This is 8192 times greater than the number of messages that can be processed, when external re-keying mechanism is not used.

5.1. Methods of Key Lifetime Control

Suppose L is an amount of data that can be safely processed with one section key. For i in {1, 2, ... , t} the frame key K^i (see Figure 4 and Figure 5) should be transformed after processing q_i messages, where q_i can be calculated in accordance with one of the following two approaches:

- Explicit approach:
  q_i is such that |M^{i,1}| + ... + |M^{i,q_i}| <= L, |M^{i,1}| + ... + |M^{i,q_i+1}| > L.
  This approach allows to use the frame key K^i in almost optimal way but it can be applied only in case when messages cannot be lost or reordered (e.g., TLS records).

- Implicit approach:
  q_i = L / m_max, i = 1, ... , t.
  The amount of data processed with one frame key K^i is calculated under the assumption that every message has the maximum length m_max. Hence this amount can be considerably less than the key lifetime limitation L. On the other hand, this approach can be applied in case when messages may be lost or reordered (e.g., DTLS records).

5.2. Parallel Constructions

The main idea behind external re-keying with a parallel construction is presented in Figure 4:

Maximum message size = m_max.

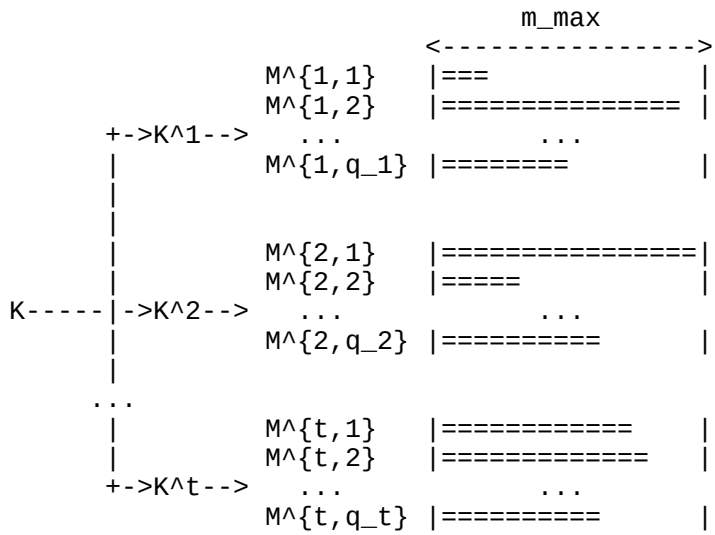


Figure 4: External parallel re-keying mechanisms

The frame key K^i, i = 1, ... , t-1, is updated after processing a certain amount of messages (see Section 5.1).

5.2.1. Parallel Construction Based on a KDF on a Block Cipher

ExtParallelC re-keying mechanism is based on the key derivation function on a block cipher and is used to generate t frame keys as follows:

K^1 | K^2 | ... | K^t = ExtParallelC(K, t * k) = MSB_{t * k}(E_{K}(Vec_n(0)) | E_{K}(Vec_n(1)) | ... | E_{K}(Vec_n(R - 1))),

where R = ceil(t * k/n).

5.2.2. Parallel Construction Based on HKDF

ExtParallelH re-keying mechanism is based on the key derivation function HKDF-Expand, described in [RFC5869], and is used to generate t frame keys as follows:

K^1 | K^2 | ... | K^t = ExtParallelH(K, t * k) = HKDF-Expand(K, label, t * k),

where label is a string (may be a zero-length string) that is defined by a specific protocol.

5.2.3. Tree-based Construction

The application of external tree-based mechanism leads to the construction of the key tree with the initial key K (root key) at the 0-level and the frame keys K^1, K^2, ... at the last level as described in Figure 6.

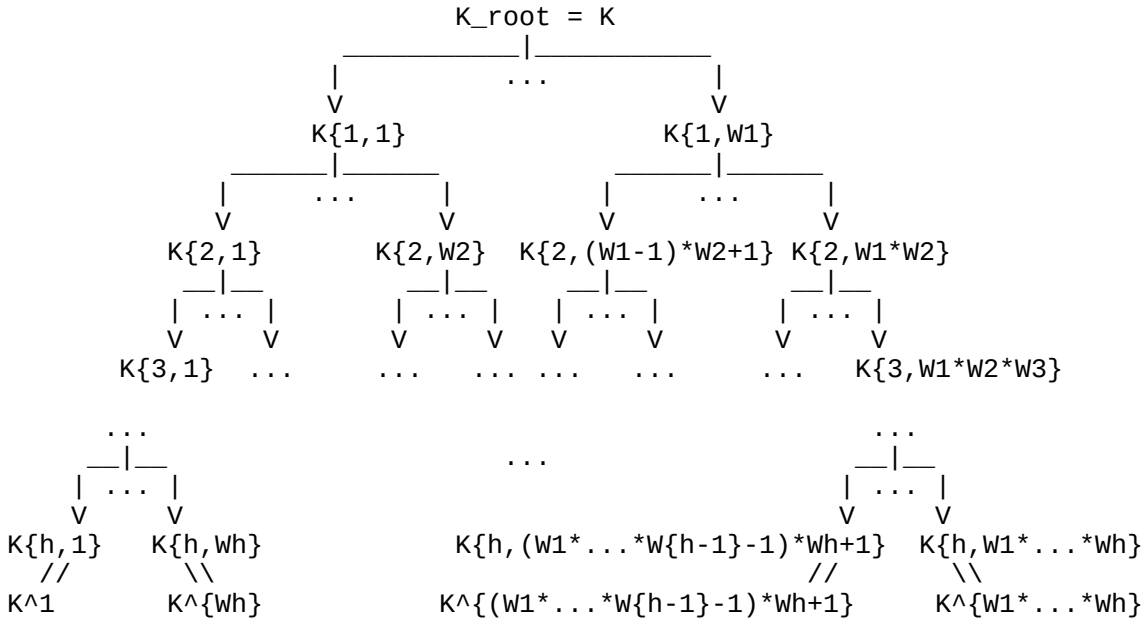


Figure 6: External Tree-based Mechanism

The height of tree h and the number of keys Wj, j in {1, ... , h}, which can be partitioned from "parent" key, are defined in accordance with a specific protocol and key lifetime limitations for a used derivation functions.

Each j-level key K{j,w}, where j in {1, ... , h}, w in {1, ... , W1 * ... * Wj}, is derived from the (j-1)-level "parent" key K{j-1,ceil(w/

Wi)} (and other appropriate input data) using j-th level derivation function that can be based on the block cipher function or on the hash function and that is defined in accordance with a specific protocol.

The i-th frame K^i, i in {1, 2, ... , W1*...*Wh}, can be calculated as follows:

```
K^i = ExtKeyTree(K, i) = KDF_h(KDF_{h-1}(... KDF_1(K, ceil(i / (W2
* ... * Wh)) ... , ceil(i / Wh)), i),
```

where KDF_j is a j-th level derivation function that takes two arguments (the parent key value and the integer in range from 1 to W1 * ... * Wj) and outputs the j-th level key value.

The frame key K^i is updated after processing a certain amount of messages (see Section 5.1).

In order to create an effective implementation, during frame key K^i generation the derivation functions KDF_j, j in {1, ... , h-1}, should be used only in case when ceil(i / (W{j+1} * ... * Wh)) != ceil((i - 1) / (W{j+1} * ... * Wh)); otherwise it is necessary to use previously generated value. This approach also makes it possible to take countermeasures against side channels attacks.

Consider an example. Suppose h = 3, W1 = W2 = W3 = W and KDF_1, KDF_2, KDF_3 are key derivation functions based on KDF_GOSTR3411_2012_256 (hereafter simply KDF) function described in [RFC7836]. The resulting ExtKeyTree function can be defined as follows:

```
ExtKeyTree(K, i) = KDF(KDF(KDF(K, "level1", ceil(i / W^2)),
"level2", ceil(i / W)), "level3", i).
```

where i in {1, 2, ... , W^3}.

The structure similar to external tree-based mechanism can be found in Section 6 of [NISTSP800-108].

5.3. Serial Constructions

The main idea behind external re-keying with a serial construction is presented in Figure 5:

Maximum message size = m_max.

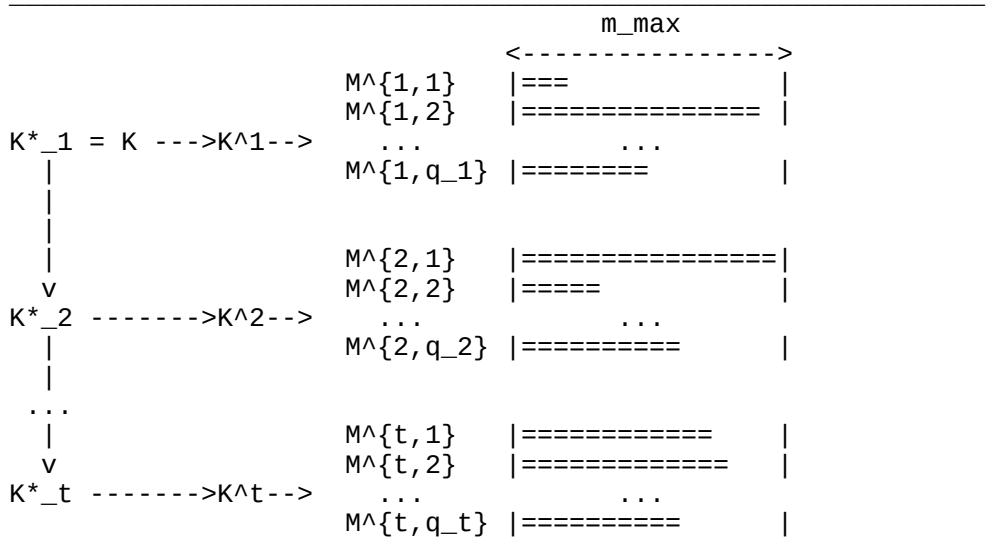


Figure 5: External serial re-keying mechanisms

The frame key K^i, i = 1, ... , t - 1, is updated after processing a certain amount of messages (see Section 5.1).

5.3.1. Serial Construction Based on a KDF on a Block Cipher

The frame key K^i is calculated using ExtSerialC transformation as follows:

```
K^i = ExtSerialC(K, i) =
MSB_k(E_{K*_i}(Vec_n(0)) |E_{K*_i}(Vec_n(1)) | ... |
E_{K*_i}(Vec_n(J - 1))),
```

where J = ceil(k / n), i = 1, ... , t, K*_i is calculated as follows:

```
K*_1 = K,

K*_{j+1} = MSB_k(E_{K*_j}(Vec_n(J)) | E_{K*_j}(Vec_n(J + 1)) |
... |
E_{K*_j}(Vec_n(2 * J - 1))),
```

where j = 1, ... , t - 1.

5.3.2. Serial Construction Based on HKDF

The frame key K^i is calculated using ExtSerialH transformation as follows:

K^i = ExtSerialH(K, i) = HKDF-Expand(K*_i, label1, k),

where i = 1, ... , t, HKDF-Expand is the HMAC-based key derivation function, described in [RFC5869], K*_i is calculated as follows:

K*_1 = K,

K*_{j+1} = HKDF-Expand(K*_j, label2, k), where j = 1, ... , t - 1,

where label1 and label2 are different strings from V* that are defined by a specific protocol (see, for example, TLS 1.3 updating traffic keys algorithm [TLSDraft]).

6. Internal Re-keying Mechanisms

This section presents an approach to increase the key lifetime by using a transformation of a data processing key (section key) during each separate message processing. Each message is processed starting with the same key (the first section key) and each section key is updated after processing N bits of message (section).

This section provides internal re-keying mechanisms called ACPKM (Advanced Cryptographic Prolongation of Key Material) and ACPKM-Master that do not use a master key and use a master key respectively. Such mechanisms are integrated into the base modes of operation and actually form new modes of operation, therefore they are called "internal re-keying" mechanisms in this document.

Internal re-keying mechanisms are recommended to be used in protocols that process large single messages (e.g., CMS messages), since the maximum gain in increasing the key lifetime is achieved by increasing the length of a message, while it provides almost no increase in the number of messages that can be processed with one initial key.

Internal re-keying increases the key lifetime through the following approach. Suppose protocol P uses some base mode of operation. Let L1 and L2 be a side channel and combinatorial limitations respectively and for some fixed amount of messages q let m1, m2 be the lengths of messages, that can be safely processed with a single initial key K according to these limitations.

Thus, by analogy with the Section 5 without re-keying the final key lifetime restriction, as displayed in Figure 7, is equal to L1 and only q messages of the length m1 can be safely processed.

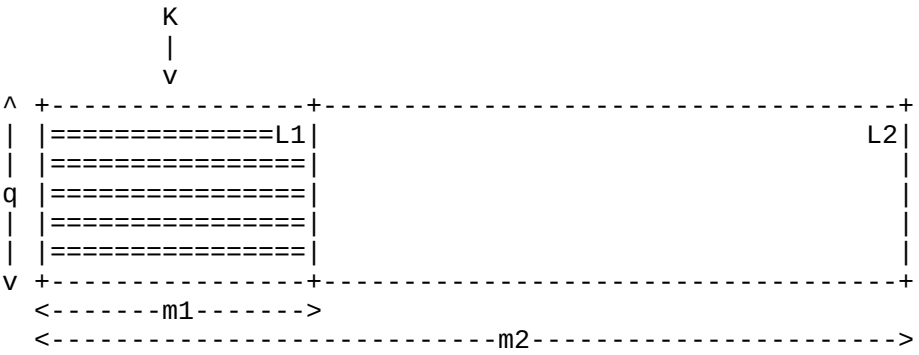


Figure 7: Basic principles of message processing without internal re-keying

Suppose that the safety margin for the protocol P is fixed and internal re-keying approach is applied to the base mode of operation. Suppose further that every message is processed with a section key, which is transformed after processing N bits of data, where N is a parameter. If q * N does not exceed L1 then the side channel limitation L1 goes off and the resulting key lifetime limitation of the initial key K can be calculated on the basis of a new combinatorial limitation L2'. The security of the mode of operation that uses internal re-keying increases when compared to base mode of operation without re-keying (thus, L2 < L2'). Hence, as displayed in Figure 8, the resulting key lifetime limitation in case of using internal re-keying can be increased up to L2'.

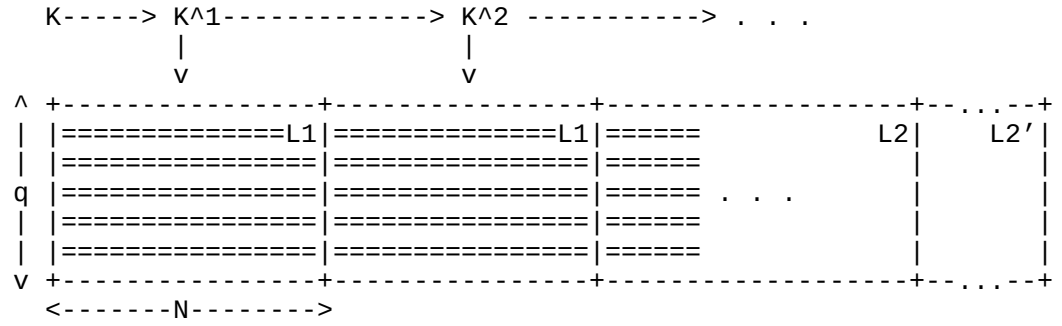


Figure 8: Basic principles of message processing with internal re-keying

Note: the key transformation process is depicted in a simplified form. A specific approach (ACPKM and ACPKM-Master re-keying mechanisms) is described below.

Since the performance of encryption can slightly decrease for rather small values of N, the parameter N should be selected for a particular protocol as maximum possible to provide necessary key lifetime for the security models that are considered.

Consider an example. Suppose L1 = 128 MB and L2 = 10 TB. Let the message size in the protocol be large/unlimited (may exhaust the whole key lifetime L2). The most restrictive resulting key lifetime limitation is equal to 128 MB.

Thus, there is a need to put a limit on the maximum message size m_max. For example, if m_max = 32 MB, it may happen that the renegotiation of initial key K would be required after processing only four messages.

If an internal re-keying mechanism with section size N = 1 MB is used, more than L1 / N = 128 MB / 1 MB = 128 messages can be processed before the renegotiation of initial key K (instead of 4 messages in case when an internal re-keying mechanism is not used). Note that only one section of each message is processed with the section key K^i, and, consequently, the key lifetime limitation L1 goes off. Hence the resulting key lifetime limitation L2' can be set to more then 10 TB (in the case when a single large message is processed using the initial key K).

6.1. Methods of Key Lifetime Control

Suppose L is an amount of data that can be safely processed with one section key, N is a section size (fixed parameter). Suppose M^{i}_1 is the first section of message M^{i}, i = 1, ... , q (see Figure 9 and Figure 10), then the parameter q can be calculated in accordance with one of the following two approaches:

- Explicit approach:
  q_i is such that |M^{1}_1| + ... + |M^{q}_1| <= L, |M^{1}_1| + ... + |M^{q+1}_1| > L
  This approach allows to use the section key K^i in an almost optimal way but it can be applied only in case when messages cannot be lost or reordered (e.g., TLS records).

- Implicit approach:
  q = L / N.
  The amount of data processed with one section key K^i is calculated under the assumption that the length of every message is equal or greater than section size N and so it can be considerably less than the key lifetime limitation L. On the other hand, this approach can be applied in case when messages may be lost or reordered (e.g., DTLS records).

6.2. Constructions that Do Not Require Master Key

This section describes the block cipher modes that use the ACPKM re-keying mechanism, which does not use a master key: an initial key is used directly for the encryption of the data.

6.2.1. ACPKM Re-keying Mechanisms

This section defines periodical key transformation without a master key, which is called ACPKM re-keying mechanism. This mechanism can be applied to one of the basic encryption modes (CTR and GCM block cipher modes) for getting an extension of this encryption mode that uses periodical key transformation without a master key. This extension can be considered as a new encryption mode.

An additional parameter that defines functioning of base encryption modes with the ACPKM re-keying mechanism is the section size N. The value of N is measured in bits and is fixed within a specific protocol based on the requirements of the system capacity and the key lifetime. The section size N MUST be divisible by the block size n.

The main idea behind internal re-keying without a master key is presented in Figure 9:

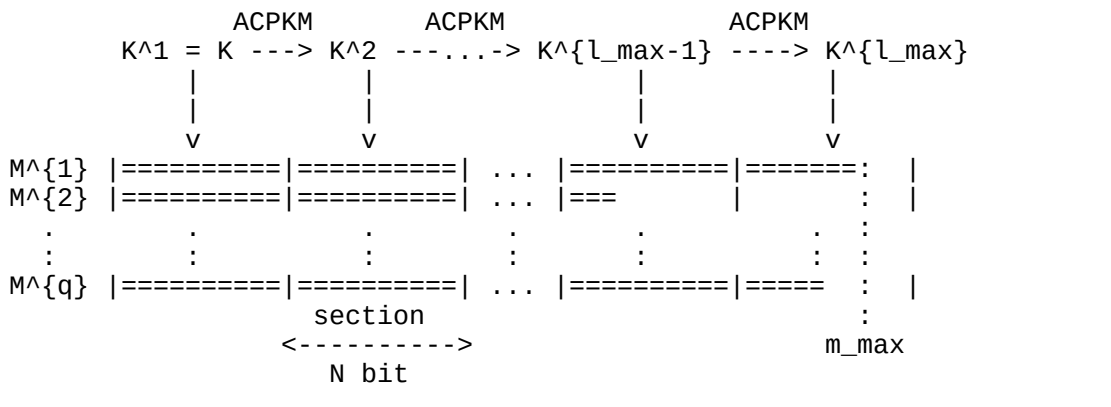


Figure 9: Internal re-keying without a master key

During the processing of the input message M with the length m in some encryption mode that uses ACPKM key transformation of the initial key K the message is divided into l = ceil(m / N) sections (denoted as M = M_1 | M_2 | ... | M_l, where M_i is in V_N for i in {1, 2, ... , l - 1} and M_l is in V_r, r <= N). The first section of each message is processed with the section key K^1 = K. To process the (i + 1)-th section of each message the section key K^{i+1} is calculated using ACPKM transformation as follows:

$$K^{i+1} = ACPKM(K^i) = MSB_k(E_{K^i}(D_1) \mid ... \mid E_{K^i}(D_J)),$$

where J = ceil(k/n) and D_1, D_2, ... , D_J are in V_n and are calculated as follows:

$$D_1 \mid D_2 \mid ... \mid D_J = MSB_{J * n}(D),$$

where D is the following constant in V_{1024}:

```
D = ( 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87
    | 88 | 89 | 8a | 8b | 8c | 8d | 8e | 8f
    | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97
    | 98 | 99 | 9a | 9b | 9c | 9d | 9e | 9f
    | a0 | a1 | a2 | a3 | a4 | a5 | a6 | a7
    | a8 | a9 | aa | ab | ac | ad | ae | af
    | b0 | b1 | b2 | b3 | b4 | b5 | b6 | b7
    | b8 | b9 | ba | bb | bc | bd | be | bf
    | c0 | c1 | c2 | c3 | c4 | c5 | c6 | c7
    | c8 | c9 | ca | cb | cc | cd | ce | cf
    | d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7
    | d8 | d9 | da | db | dc | dd | de | df
    | e0 | e1 | e2 | e3 | e4 | e5 | e6 | e7
    | e8 | e9 | ea | eb | ec | ed | ee | ef
    | f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7
    | f8 | f9 | fa | fb | fc | fd | fe | ff )
```

N o t e : The constant D is such that D_1, ... , D_J are pairwise different for any allowed n and k values.

N o t e : The constant D is such that the highest bit of its each octet is equal to 1. This condition is important, as in conjunction with message length limitation it allows to prevent collisions of block cipher permutation inputs in cases of key transformation and message processing.

6.2.2. CTR-ACPKM Encryption Mode

This section defines a CTR-ACPKM encryption mode that uses internal ACPKM re-keying mechanism for the periodical key transformation.

The CTR-ACPKM mode can be considered as the basic encryption mode CTR (see [MODES]) extended by the ACPKM re-keying mechanism.

The CTR-ACPKM encryption mode can be used with the following parameters:

- 64 <= n <= 512;
- 128 <= k <= 512;
- the number of bits c in a specific part of the block to be incremented is such that 32 <= c <= 3 / 4 n, c is a multiple of 8;
- the maximum message size m_max = n * 2^{c-1}.

The CTR-ACPKM mode encryption and decryption procedures are defined as follows:

```
+-------------------------------------------------------------------+
|  CTR-ACPKM-Encrypt(N, K, ICN, P)                                  |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - initial counter nonce ICN in V_{n-c},                          |
|  - plaintext P = P_1 | ... | P_b, |P| <= m_max.                   |
|  Output:                                                          |
|  - ciphertext C.                                                  |
|-------------------------------------------------------------------|
|  1. CTR_1 = ICN | 0^c                                             |
|  2. For j = 2, 3, ... , b do                                      |
|         CTR_{j} = Inc_c(CTR_{j-1})                                |
|  3. K^1 = K                                                       |
|  4. For i = 2, 3, ... , ceil(|P| / N)                             |
|         K^i = ACPKM(K^{i-1})                                      |
|  5. For j = 1, 2, ... , b do                                      |
|         i = ceil(j * n / N),                                      |
|         G_j = E_{K^i}(CTR_j)                                      |
|  6. C = P (xor) MSB_{|P|}(G_1 | ... | G_b)                        |
|  7. Return C                                                      |
+-------------------------------------------------------------------+

+-------------------------------------------------------------------+
|  CTR-ACPKM-Decrypt(N, K, ICN, C)                                  |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - initial counter nonce ICN in V_{n-c},                          |
|  - ciphertext C = C_1 | ... | C_b, |C| <= m_max.                  |
|  Output:                                                          |
|  - plaintext P.                                                   |
|-------------------------------------------------------------------|
|  1. P = CTR-ACPKM-Encrypt(N, K, ICN, C)                           |
|  2. Return P                                                      |
+-------------------------------------------------------------------+
```

The initial counter nonce ICN value for each message that is encrypted under the given initial key K must be chosen in a unique manner.

6.2.3. GCM-ACPKM Authenticated Encryption Mode

This section defines GCM-ACPKM authenticated encryption mode that uses internal ACPKM re-keying mechanism for the periodical key transformation.

The GCM-ACPKM mode can be considered as the basic authenticated encryption mode GCM (see [GCM]) extended by the ACPKM re-keying mechanism.

The GCM-ACPKM authenticated encryption mode can be used with the following parameters:

- n in {128, 256};
- 128 <= k <= 512;
- the number of bits c in a specific part of the block to be incremented is such that 1 / 4 n <= c <= 1 / 2 n, c is a multiple of 8;
- authentication tag length t;
- the maximum message size m_max = min{n * (2^{c-1} - 2), 2^{n/2} - 1}.

The GCM-ACPKM mode encryption and decryption procedures are defined as follows:

```
+-------------------------------------------------------------------+
|  GHASH(X, H)                                                      |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - bit string X = X_1 | ... | X_m, X_1, ... , X_m in V_n.         |
|  Output:                                                          |
|  - block GHASH(X, H) in V_n.                                      |
|-------------------------------------------------------------------|
|  1. Y_0 = 0^n                                                     |
|  2. For i = 1, ... , m do                                         |
|         Y_i = (Y_{i-1} (xor) X_i) * H                             |
|  3. Return Y_m                                                    |
+-------------------------------------------------------------------+

+-------------------------------------------------------------------+
|  GCTR(N, K, ICB, X)                                               |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - initial counter block ICB,                                     |
|  - X = X_1 | ... | X_b.                                           |
|  Output:                                                          |
|  - Y in V_{|X|}.                                                  |
```

```
|-------------------------------------------------------------------|
|  1. If X in V_0 then return Y, where Y in V_0                     |
|  2. GCTR_1 = ICB                                                  |
|  3. For i = 2, ... , b do                                         |
|         GCTR_i = Inc_c(GCTR_{i-1})                                |
|  4. K^1 = K                                                       |
|  5. For j = 2, ... , ceil(|X| / N)                                |
|         K^j = ACPKM(K^{j-1})                                      |
|  6. For i = 1, ... , b do                                         |
|         j = ceil(i * n / N),                                      |
|         G_i = E_{K_j}(GCTR_i)                                     |
|  7. Y = X (xor) MSB_{|X|}(G_1 | ... | G_b)                        |
|  8. Return Y                                                      |
+-------------------------------------------------------------------+

+-------------------------------------------------------------------+
|  GCM-ACPKM-Encrypt(N, K, ICN, P, A)                               |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - initial counter nonce ICN in V_{n-c},                          |
|  - plaintext P = P_1 | ... | P_b, |P| <= m_max,                   |
|  - additional authenticated data A.                               |
|  Output:                                                          |
|  - ciphertext C,                                                  |
|  - authentication tag T.                                          |
|-------------------------------------------------------------------|
|  1. H = E_{K}(0^n)                                                |
|  2. ICB_0 = ICN | 0^{c-1} | 1                                     |
|  3. C = GCTR(N, K, Inc_c(ICB_0), P)                               |
|  4. u = n * ceil(|C| / n) - |C|                                   |
|     v = n * ceil(|A| / n) - |A|                                   |
|  5. S = GHASH(A | 0^v | C | 0^u | Vec_{n/2}(|A|) |               |
|                | Vec_{n/2}(|C|), H)                               |
|  6. T = MSB_t(E_{K}(ICB_0) (xor) S)                               |
|  7. Return C | T                                                  |
+-------------------------------------------------------------------+

+-------------------------------------------------------------------+
|  GCM-ACPKM-Decrypt(N, K, ICN, A, C, T)                            |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - initial counter block ICN,                                     |
|  - additional authenticated data A,                               |
|  - ciphertext C = C_1 | ... | C_b, |C| <= m_max,                 |
```

```
|  - authentication tag T.                                              |
|  Output:                                                              |
|  - plaintext P or FAIL.                                               |
|-----------------------------------------------------------------------|
|  1. H = E_{K}(0^n)                                                    |
|  2. ICB_0 = ICN | 0^{c-1} | 1                                         |
|  3. P = GCTR(N, K, Inc_c(ICB_0), C)                                   |
|  4. u = n * ceil(|C| / n) - |C|                                       |
|     v = n * ceil(|A| / n) - |A|                                       |
|  5. S = GHASH(A | 0^v | C | 0^u | Vec_{n/2}(|A|) |                   |
|               | Vec_{n/2}(|C|), H)                                    |
|  6. T' = MSB_t(E_{K}(ICB_0) (xor) S)                                  |
|  7. If T = T' then return P; else return FAIL                         |
+-----------------------------------------------------------------------+
```

The * operation on (pairs of) the 2^n possible blocks corresponds to the multiplication operation for the binary Galois (finite) field of 2^n elements defined by the polynomial f as follows (by analogy with [GCM]):

n = 128: f = a^128 + a^7 + a^2 + a^1 + 1,

n = 256: f = a^256 + a^10 + a^5 + a^2 + 1.

The initial vector IV value for each message that is encrypted under the given initial key K must be chosen in a unique manner.

The key for computing values E_{K}(ICB_0) and H is not updated and is equal to the initial key K.

6.3. Constructions that Require Master Key

This section describes the block cipher modes that use the ACPKM-Master re-keying mechanism, which use the initial key K as a master key, so K is never used directly for data processing but is used for key derivation.

6.3.1. ACPKM-Master Key Derivation from the Master Key

This section defines periodical key transformation with a master key, which is called ACPKM-Master re-keying mechanism. This mechanism can be applied to one of the basic modes of operation (CTR, GCM, CBC, CFB, OMAC modes) for getting an extension that uses periodical key transformation with a master key. This extension can be considered as a new mode of operation.

Additional parameters that define the functioning of modes of operation that use the ACPKM-Master re-keying mechanism are the

section size N, the change frequency T* of the master keys K*_1, K*_2, ... (see Figure 10) and the size d of the section key material. The values of N and T* are measured in bits and are fixed within a specific protocol, based on the requirements of the system capacity and the key lifetime. The section size N MUST also be divisible by the block size n. The master key frequency T* MUST be divisible by d and by n.

The main idea behind internal re-keying with a master key is presented in Figure 10:

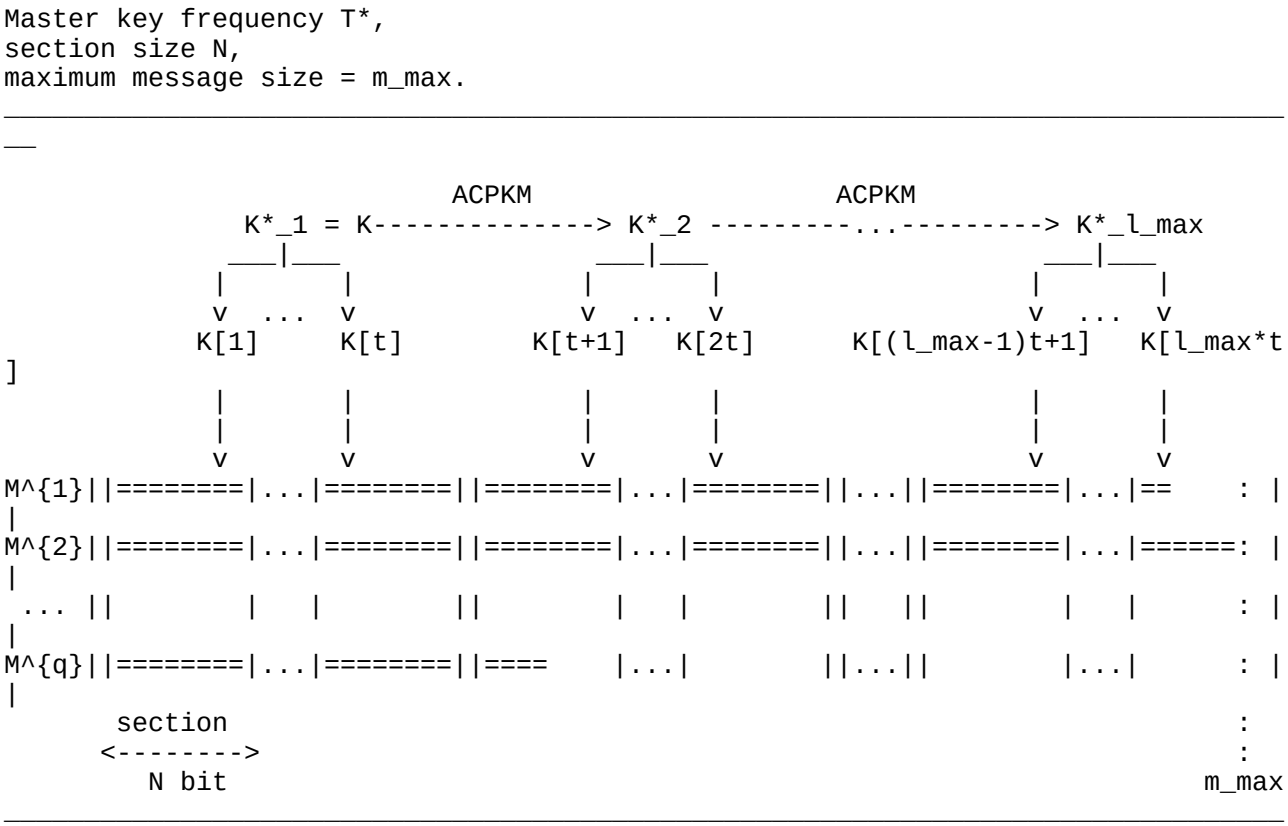


Figure 10: Internal re-keying with a master key

During the processing of the input message M with the length m in some mode of operation that uses ACPKM-Master key transformation with the initial key K and the master key frequency T* the message M is divided into l = ceil(m / N) sections (denoted as M = M_1 | M_2 | ... | M_l, where M_i is in V_N for i in {1, 2, ... , l - 1} and M_l is in V_r, r <= N). The j-th section of each message is processed

with the key material K[j], j in {1, ... , l}, |K[j]| = d, that is calculated with the ACPKM-Master algorithm as follows:

K[1] | ... | K[l] = ACPKM-Master(T*, K, d, l) = CTR-ACPKM-Encrypt (T*, K, 1^{n/2}, 0^{d*l}).

Note: the parameters d and l MUST be such that d * l <= n * 2^{n/2-1}.

6.3.2. CTR-ACPKM-Master Encryption Mode

This section defines a CTR-ACPKM-Master encryption mode that uses internal ACPKM-Master re-keying mechanism for the periodical key transformation.

The CTR-ACPKM-Master encryption mode can be considered as the basic encryption mode CTR (see [MODES]) extended by the ACPKM-Master re-keying mechanism.

The CTR-ACPKM-Master encryption mode can be used with the following parameters:

- 64 <= n <= 512;
- 128 <= k <= 512;
- the number of bits c in a specific part of the block to be incremented is such that 32 <= c <= 3 / 4 n, c is a multiple of 8;
- the maximum message size m_max = min{N * (n * 2^{n/2-1} / k), n * 2^c}.

The key material K[j] that is used for one section processing is equal to K^j, |K^j| = k bits.

The CTR-ACPKM-Master mode encryption and decryption procedures are defined as follows:

```
+-------------------------------------------------------------------+
|  CTR-ACPKM-Master-Encrypt(N, K, T*, ICN, P)                       |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - master key frequency T*,                                       |
|  - initial counter nonce ICN in V_{n-c},                          |
|  - plaintext P = P_1 | ... | P_b, |P| <= m_max.                   |
|  Output:                                                          |
|  - ciphertext C.                                                  |
|-------------------------------------------------------------------|
|  1. CTR_1 = ICN | 0^c                                             |
|  2. For j = 2, 3, ... , b do                                      |
|         CTR_{j} = Inc_c(CTR_{j-1})                                |
|  3. l = ceil(|P| / N)                                             |
|  4. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)                   |
|  5. For j = 1, 2, ... , b do                                      |
|         i = ceil(j * n / N),                                      |
|         G_j = E_{K^i}(CTR_j)                                      |
|  6. C = P (xor) MSB_{|P|}(G_1 | ... |G_b)                         |
|  7. Return C                                                      |
+-------------------------------------------------------------------+

+-------------------------------------------------------------------+
|  CTR-ACPKM-Master-Decrypt(N, K, T*, ICN, C)                       |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - master key frequency T*,                                       |
|  - initial counter nonce ICN in V_{n-c},                          |
|  - ciphertext C = C_1 | ... | C_b, |C| <= m_max.                  |
|  Output:                                                          |
|  - plaintext P.                                                   |
|-------------------------------------------------------------------|
|  1. P = CTR-ACPKM-Master-Encrypt(N, K, T*, ICN, C)                |
|  1. Return P                                                      |
+-------------------------------------------------------------------+
```

The initial counter nonce ICN value for each message that is encrypted under the given initial key must be chosen in a unique manner.

6.3.3. GCM-ACPKM-Master Authenticated Encryption Mode

This section defines a GCM-ACPKM-Master authenticated encryption mode that uses internal ACPKM-Master re-keying mechanism for the periodical key transformation.

The GCM-ACPKM-Master authenticated encryption mode can be considered as the basic authenticated encryption mode GCM (see [GCM]) extended by the ACPKM-Master re-keying mechanism.

The GCM-ACPKM-Master authenticated encryption mode can be used with the following parameters:

- n in {128, 256};
- 128 <= k <= 512;
- the number of bits c in a specific part of the block to be incremented is such that 1 / 4 n <= c <= 1 / 2 n, c is a multiple of 8;
- authentication tag length t;
- the maximum message size m_max = min{N * (n * 2^{n/2-1} / k), n * (2^c - 2), 2^{n/2} - 1}.

The key material K[j] that is used for the j-th section processing is equal to K^j, |K^j| = k bits.

The GCM-ACPKM-Master mode encryption and decryption procedures are defined as follows:

```
+-------------------------------------------------------------------+
|  GHASH(X, H)                                                      |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - bit string X = X_1 | ... | X_m, X_i in V_n for i in {1, ... ,m}|
|  Output:                                                          |
|  - block GHASH(X, H) in V_n                                       |
|-------------------------------------------------------------------|
|  1. Y_0 = 0^n                                                     |
|  2. For i = 1, ... , m do                                         |
|         Y_i = (Y_{i-1} (xor) X_i) * H                             |
|  3. Return Y_m                                                    |
+-------------------------------------------------------------------+

+-------------------------------------------------------------------+
```

```
|  GCTR(N, K, T*, ICB, X)                                              |
|----------------------------------------------------------------------|
|  Input:                                                              |
|  - section size N,                                                   |
|  - initial key K,                                                    |
|  - master key frequency T*,                                          |
|  - initial counter block ICB,                                        |
|  - X = X_1 | ... | X_b.                                              |
|  Output:                                                             |
|  - Y in V_{|X|}.                                                     |
|----------------------------------------------------------------------|
|  1. If X in V_0 then return Y, where Y in V_0                        |
|  2. GCTR_1 = ICB                                                     |
|  3. For i = 2, ... , b do                                            |
|         GCTR_i = Inc_c(GCTR_{i-1})                                   |
|  4. l = ceil(|X| / N)                                                |
|  5. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)                    |
|  6. For j = 1, ... , b do                                            |
|         i = ceil(j * n / N),                                         |
|         G_j = E_{K^i}(GCTR_j)                                        |
|  7. Y = X (xor) MSB_{|X|}(G_1 | ... | G_b)                           |
|  8. Return Y                                                         |
+----------------------------------------------------------------------+

+----------------------------------------------------------------------+
|  GCM-ACPKM-Master-Encrypt(N, K, T*, ICN, P, A)                       |
|----------------------------------------------------------------------|
|  Input:                                                              |
|  - section size N,                                                   |
|  - initial key K,                                                    |
|  - master key frequency T*,                                          |
|  - initial counter nonce ICN in V_{n-c},                             |
|  - plaintext P = P_1 | ... | P_b, |P| <= m_max.                     |
|  - additional authenticated data A.                                  |
|  Output:                                                             |
|  - ciphertext C,                                                     |
|  - authentication tag T.                                             |
|----------------------------------------------------------------------|
|  1. K^1 = ACPKM-Master(T*, K, k, 1)                                  |
|  2. H = E_{K^1}(0^n)                                                 |
|  3. ICB_0 = ICN | 0^{c-1} | 1                                        |
|  4. C = GCTR(N, K, T*, Inc_c(ICB_0), P)                              |
|  5. u = n * ceil(|C| / n) - |C|                                      |
|     v = n * ceil(|A| / n) - |A|                                      |
|  6. S = GHASH(A | 0^v | C | 0^u | Vec_{n/2}(|A|) |                 |
|                | Vec_{n/2}(|C|), H)                                  |
|  7. T = MSB_t(E_{K^1}(ICB_0) (xor) S)                                |
|  8. Return C | T                                                     |
```

```
+-------------------------------------------------------------------+

+-------------------------------------------------------------------+
|  GCM-ACPKM-Master-Decrypt(N, K, T*, ICN, A, C, T)                 |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - master key frequency T*,                                       |
|  - initial counter nonce ICN in V_{n-c},                          |
|  - additional authenticated data A.                               |
|  - ciphertext C = C_1 | ... | C_b, |C| <= m_max,                 |
|  - authentication tag T.                                          |
|  Output:                                                          |
|  - plaintext P or FAIL.                                           |
|-------------------------------------------------------------------|
|  1. K^1 = ACPKM-Master(T*, K, k, 1)                               |
|  2. H = E_{K^1}(0^n)                                              |
|  3. ICB_0 = ICN | 0^{c-1} | 1                                     |
|  4. P = GCTR(N, K, T*, Inc_c(ICB_0), C)                           |
|  5. u = n * ceil(|C| / n) - |C|                                   |
|     v = n * ceil(|A| / n) - |A|                                   |
|  6. S = GHASH(A | 0^v | C | 0^u | Vec_{n/2}(|A|) |               |
|               | Vec_{n/2}(|C|), H)                              |
|  7. T' = MSB_t(E_{K^1}(ICB_0) (xor) S)                            |
|  8. IF T = T' then return P; else return FAIL.                    |
+-------------------------------------------------------------------+
```

The * operation on (pairs of) the 2^n possible blocks corresponds to the multiplication operation for the binary Galois (finite) field of 2^n elements defined by the polynomial f as follows (by analogy with [GCM]):

n = 128: f = a^128 + a^7 + a^2 + a^1 + 1,

n = 256: f = a^256 + a^10 + a^5 + a^2 + 1.

The initial vector IV value for each message that is encrypted under the given initial key must be chosen in a unique manner.

6.3.4. CBC-ACPKM-Master Encryption Mode

This section defines a CBC-ACPKM-Master encryption mode that uses internal ACPKM-Master re-keying mechanism for the periodical key transformation.

The CBC-ACPKM-Master encryption mode can be considered as the basic encryption mode CBC (see [MODES]) extended by the ACPKM-Master re-keying mechanism.

The CBC-ACPKM-Master encryption mode can be used with the following parameters:

- 64 <= n <= 512;
- 128 <= k <= 512;
- the maximum message size m_max = N * (n * 2^{n/2-1} / k).

In the specification of the CBC-ACPKM-Master mode the plaintext and ciphertext must be a sequence of one or more complete data blocks. If the data string to be encrypted does not initially satisfy this property, then it MUST be padded to form complete data blocks. The padding methods are out of the scope of this document. An example of a padding method can be found in Appendix A of [MODES].

The key material K[j] that is used for the j-th section processing is equal to K^j, |K^j| = k bits.

We will denote by D_{K} the decryption function which is a permutation inverse to the E_{K}.

The CBC-ACPKM-Master mode encryption and decryption procedures are defined as follows:

```
+-------------------------------------------------------------------+
|  CBC-ACPKM-Master-Encrypt(N, K, T*, IV, P)                        |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - master key frequency T*,                                       |
|  - initialization vector IV in V_n,                               |
|  - plaintext P = P_1 | ... | P_b, |P_b| = n, |P| <= m_max.      |
|  Output:                                                          |
|  - ciphertext C.                                                  |
|-------------------------------------------------------------------|
|  1. l = ceil(|P| / N)                                             |
|  2. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)                 |
|  3. C_0 = IV                                                      |
|  4. For j = 1, 2, ... , b do                                      |
|         i = ceil(j * n / N),                                      |
|         C_j = E_{K^i}(P_j (xor) C_{j-1})                          |
|  5. Return C = C_1 | ... | C_b                                    |
|-------------------------------------------------------------------|
+-------------------------------------------------------------------+

+-------------------------------------------------------------------+
|  CBC-ACPKM-Master-Decrypt(N, K, T*, IV, C)                        |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - master key frequency T*,                                       |
|  - initialization vector IV in V_n,                               |
|  - ciphertext C = C_1 | ... | C_b, |C_b| = n, |C| <= m_max.     |
|  Output:                                                          |
|  - plaintext P.                                                   |
|-------------------------------------------------------------------|
|  1. l = ceil(|C| / N)                                             |
|  2. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)                 |
|  3. C_0 = IV                                                      |
|  4. For j = 1, 2, ... , b do                                      |
|         i = ceil(j * n / N)                                       |
|         P_j = D_{K^i}(C_j) (xor) C_{j-1}                          |
|  5. Return P = P_1 | ... | P_b                                    |
+-------------------------------------------------------------------+
```

The initialization vector IV for each message that is encrypted under the given initial key does not need to be secret, but must be unpredictable.

6.3.5. CFB-ACPKM-Master Encryption Mode

This section defines a CFB-ACPKM-Master encryption mode that uses internal ACPKM-Master re-keying mechanism for the periodical key transformation.

The CFB-ACPKM-Master encryption mode can be considered as the basic encryption mode CFB (see [MODES]) extended by the ACPKM-Master re-keying mechanism.

The CFB-ACPKM-Master encryption mode can be used with the following parameters:

- 64 <= n <= 512;
- 128 <= k <= 512;
- the maximum message size m_max = N * (n * 2^{n/2-1} / k).

The key material K[j] that is used for the j-th section processing is equal to K^j, |K^j| = k bits.

The CFB-ACPKM-Master mode encryption and decryption procedures are defined as follows:

```
+-------------------------------------------------------------+
|  CFB-ACPKM-Master-Encrypt(N, K, T*, IV, P)                  |
|-------------------------------------------------------------|
|  Input:                                                     |
|  - section size N,                                          |
|  - initial key K,                                           |
|  - master key frequency T*,                                 |
|  - initialization vector IV in V_n,                         |
|  - plaintext P = P_1 | ... | P_b, |P| <= m_max.            |
|  Output:                                                    |
|  - ciphertext C.                                            |
|-------------------------------------------------------------|
|  1. l = ceil(|P| / N)                                       |
|  2. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)           |
|  3. C_0 = IV                                                |
|  4. For j = 1, 2, ... , b - 1 do                            |
|         i = ceil(j * n / N),                                |
|         C_j = E_{K^i}(C_{j-1}) (xor) P_j                    |
|  5. C_b = MSB_{|P_b|}(E_{K^l}(C_{b-1})) (xor) P_b           |
|  6. Return C = C_1 | ... | C_b                              |
+-------------------------------------------------------------+

+-------------------------------------------------------------+
|  CFB-ACPKM-Master-Decrypt(N, K, T*, IV, C)                  |
|-------------------------------------------------------------|
|  Input:                                                     |
|  - section size N,                                          |
|  - initial key K,                                           |
|  - master key frequency T*,                                 |
|  - initialization vector IV in V_n,                         |
|  - ciphertext C = C_1 | ... | C_b, |C| <= m_max.           |
|  Output:                                                    |
|  - plaintext P.                                             |
|-------------------------------------------------------------|
|  1. l = ceil(|C| / N)                                       |
|  2. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)           |
|  3. C_0 = IV                                                |
|  4. For j = 1, 2, ... , b - 1 do                            |
|         i = ceil(j * n / N),                                |
|         P_j = E_{K^i}(C_{j-1}) (xor) C_j                    |
|  5. P_b = MSB_{|C_b|}(E_{K^l}(C_{b-1})) (xor) C_b           |
|  6. Return P = P_1 | ... | P_b                              |
+-------------------------------------------------------------+
```

The initialization vector IV for each message that is encrypted under the given initial key need not to be secret, but must be unpredictable.

6.3.6. OMAC-ACPKM-Master Authentication Mode

This section defines an OMAC-ACPKM-Master message authentication code calculation mode that uses internal ACPKM-Master re-keying mechanism for the periodical key transformation.

The OMAC-ACPKM-Master mode can be considered as the basic message authentication code calculation mode OMAC, which is also known as CMAC (see [RFC4493]), extended by the ACPKM-Master re-keying mechanism.

The OMAC-ACPKM-Master message authentication code calculation mode can be used with the following parameters:

- n in {64, 128, 256};
- 128 <= k <= 512;
- the maximum message size m_max = N * (n * 2^{n/2-1} / (k + n)).

The key material K[j] that is used for one section processing is equal to K^j | K^j_1, where |K^j| = k and |K^j_1| = n.

The following is a specification of the subkey generation process of OMAC:

```
+-------------------------------------------------------------------+
| Generate_Subkey(K1, r)                                            |
|-------------------------------------------------------------------|
| Input:                                                            |
|  - key K1.                                                        |
|  Output:                                                          |
|  - key SK.                                                        |
|-------------------------------------------------------------------|
|   1. If r = n then return K1                                      |
|   2. If r < n then                                                |
|          if MSB_1(K1) = 0                                         |
|              return K1 << 1                                       |
|          else                                                     |
|              return (K1 << 1) (xor) R_n                           |
|                                                                   |
+-------------------------------------------------------------------+
```

Where R_n takes the following values:

- n = 64: R_{64} = 0^{59} | 11011;

   o n = 128: R_{128} = 0^{120} | 10000111;

   o n = 256: R_{256} = 0^{145} | 10000100101.

   The OMAC-ACPKM-Master message authentication code calculation mode is defined as follows:

```
+-------------------------------------------------------------------------+
| OMAC-ACPKM-Master(K, N, T*, M)                                          |
|-------------------------------------------------------------------------|
| Input:                                                                  |
|  - section size N,                                                      |
|  - initial key K,                                                       |
|  - master key frequency T*,                                             |
|  - plaintext M = M_1 | ... | M_b, |M| <= m_max.                         |
|  Output:                                                                |
|  - message authentication code T.                                       |
|-------------------------------------------------------------------------|
| 1. C_0 = 0^n                                                            |
| 2. l = ceil(|M| / N)                                                    |
| 3. K^1 | K^1_1 | ... | K^l | K^l_1 = ACPKM-Master(T*, K, (k + n), l)  |
| 4. For j = 1, 2, ... , b - 1 do                                         |
|        i = ceil(j * n / N),                                             |
|        C_j = E_{K^i}(M_j (xor) C_{j-1})                                 |
| 5. SK = Generate_Subkey(K^l_1, |M_b|)                                  |
| 6. If |M_b| = n then M*_b = M_b                                         |
|                 else M*_b = M_b | 1 | 0^{n - 1 -|M_b|}                |
| 7. T = E_{K^l}(M*_b (xor) C_{b-1} (xor) SK)                             |
| 8. Return T                                                             |
+-------------------------------------------------------------------------+
```

7. Joint Usage of External and Internal Re-keying

   Any mechanism described in Section 5 can be used with any mechanism described in Section 6.

8. Security Considerations

   Re-keying should be used to increase "a priori" security properties of ciphers in hostile environments (e.g., with side-channel adversaries). If some efficient attacks are known for a cipher, it must not be used. So re-keying cannot be used as a patch for vulnerable ciphers. Base cipher properties must be well analyzed, because the security of re-keying mechanisms is based on the security of a block cipher as a pseudorandom function.

Re-keying is not intended to solve any post-quantum security issues for symmetric crypto, since the reduction of security caused by Grover's algorithm is not connected with a size of plaintext transformed by a cipher - only a negligible (sufficient for key uniqueness) material is needed; and the aim of re-keying is to limit a size of plaintext transformed on one initial key.

Re-keying can provide backward security only if previous traffic keys are securely deleted by all parties that have the keys.

9. References

9.1. Normative References

[CMS] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <http://www.rfc-editor.org/info/rfc5652>.

[DTLS] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <http://www.rfc-editor.org/info/rfc6347>.

[ESP] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <http://www.rfc-editor.org/info/rfc4303>.

[GCM] McGrew, D. and J. Viega, "The Galois/Counter Mode of Operation (GCM)", Submission to NIST http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec.pdf, January 2004.

[MODES] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", NIST Special Publication 800-38A, December 2001.

[NISTSP800-108] National Institute of Standards and Technology, "Recommendation for Key Derivation Using Pseudorandom Functions", NIST Special Publication 800-108, November 2008, <http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-108.pdf>.

[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>.

[RFC4493] Song, JH., Poovendran, R., Lee, J., and T. Iwata, "The AES-CMAC Algorithm", RFC 4493, DOI 10.17487/RFC4493, June 2006, <https://www.rfc-editor.org/info/rfc4493>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC7836] Smyshlyaev, S., Ed., Alekseev, E., Oshkin, I., Popov, V., Leontiev, S., Podobaev, V., and D. Belyavsky, "Guidelines on the Cryptographic Algorithms to Accompany the Usage of Standards GOST R 34.10-2012 and GOST R 34.11-2012", RFC 7836, DOI 10.17487/RFC7836, March 2016, <https://www.rfc-editor.org/info/rfc7836>.

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

[TLS] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

[TLSDraft] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", 2017, <https://tools.ietf.org/html/draft-ietf-tls-tls13-20>.

9.2. Informative References

[AbBell] Michel Abdalla and Mihir Bellare, "Increasing the Lifetime of a Key: A Comparative Analysis of the Security of Re-keying Techniques", ASIACRYPT2000, LNCS 1976, pp. 546-559, 2000.

[LDC] Howard M. Heys, "A Tutorial on Linear and Differential Cryptanalysis", 2017, <http://www.cs.bc.edu/~straubin/crypto2017/heys.pdf>.

[Sweet32] Karthikeyan Bhargavan, Gaetan Leurent, "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN", Cryptology ePrint Archive Report 2016/798, 2016, <https://sweet32.info/SWEET32_CCS16.pdf>.

[TEMPEST] By Craig Ramsay, Jasper Lohuis, "TEMPEST attacks against AES. Covertly stealing keys for 200 euro", 2017, <https://www.fox-it.com/en/wp-content/uploads/sites/11/Tempest_attacks_against_AES.pdf>.

Appendix A. Test examples

External re-keying with a parallel construction based on AES-256
****

```
k = 256
t = 128

Initial key:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

K^1:
51 16 8A B6 C8 A8 38 65 54 85 31 A5 D2 BA C3 86
64 7D 5C D5 1C 3D 62 98 BC 09 B1 D8 64 EC D9 B1

K^2:
6F ED F5 D3 77 57 48 75 35 2B 5F 4D B6 5B E0 15
B8 02 92 32 D8 D3 8D 73 FE DC DD C6 C8 36 78 BD

K^3:
B6 40 24 85 A4 24 BD 35 B4 26 43 13 76 26 70 B6
5B F3 30 3D 3B 20 EB 14 D1 3B B7 91 74 E3 DB EC

...

K^126:
2F 3F 15 1B 53 88 23 CD 7D 03 FC 3D FD B3 57 5E
23 E4 1C 4E 46 FF 6B 33 34 12 27 84 EF 5D 82 23

K^127:
8E 51 31 FB 0B 64 BB D0 BC D4 C5 7B 1C 66 EF FD
97 43 75 10 6C AF 5D 5E 41 E0 17 F4 05 63 05 ED

K^128:
77 4F BF B3 22 60 C5 3B A3 8E FE B1 96 46 76 41
94 49 AF 84 2D 84 65 A7 F4 F7 2C DC A4 9D 84 F9
```

External re-keying with a serial construction based on SHA-256
****

```
k = 256
```

```
t = 128

Initial key:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

label1:
SHA2label1

label2:
SHA2label2

K*_1:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

K^1:
2D A8 D1 37 6C FD 52 7F F7 36 A4 E2 81 C6 0A 9B
F3 8E 66 97 ED 70 4F B5 FB 10 33 CC EC EE D5 EC

K*_2:
14 65 5A D1 7C 19 86 24 9B D3 56 DF CC BE 73 6F
52 62 4A 9D E3 CC 40 6D A9 48 DA 5C D0 68 8A 04

K^2:
2F EA 8D 57 2B EF B8 89 42 54 1B 8C 1B 3F 8D B1
84 F9 56 C7 FE 01 11 99 1D FB 98 15 FE 65 85 CF

K*_3:
18 F0 B5 2A D2 45 E1 93 69 53 40 55 43 70 95 8D
70 F0 20 8C DF B0 5D 67 CD 1B BF 96 37 D3 E3 EB

K^3:
53 C7 4E 79 AE BC D1 C8 24 04 BF F6 D7 B1 AC BF
F9 C0 0E FB A8 B9 48 29 87 37 E1 BA E7 8F F7 92

...

K*_126:
A3 6D BF 02 AA 0B 42 4A F2 C0 46 52 68 8B C7 E6
5E F1 62 C3 B3 2F DD EF E4 92 79 5D BB 45 0B CA

K^126:
6C 4B D6 22 DC 40 48 0F 29 C3 90 B8 E5 D7 A7 34
23 4D 34 65 2C CE 4A 76 2C FE 2A 42 C8 5B FE 9A

K*_127:
84 5F 49 3D B8 13 1D 39 36 2B BE D3 74 8F 80 A1
```

```
   05 A7 07 37 BA 15 72 E0 73 49 C2 67 5D 0A 28 A1

   K^127:
   57 F0 BD 5A B8 2A F3 6B 87 33 CF F7 22 62 B4 D0
   F0 EE EF E1 50 74 E5 BA 13 C1 23 68 87 36 29 A2

   K*_128:
   52 F2 0F 56 5C 9C 56 84 AF 69 AD 45 EE B8 DA 4E
   7A A6 04 86 35 16 BA 98 E4 CB 46 D2 E8 9A C1 09

   K^128:
   9B DD 24 7D F3 25 4A 75 E0 22 68 25 68 DA 9D D5
   C1 6D 2D 2B 4F 3F 1F 2B 5E 99 82 7F 15 A1 4F A4



   CTR-ACPKM mode with AES-256
   ***************************
   k = 256
   n = 128
   c = 64
   N = 256

   Initial key K:
   88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77
   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF

   Plain text P:
   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88
   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A
   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00
   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11
   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22
   44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33
   55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33 44

   ICN:
   12 34 56 78 90 AB CE F0 A1 B2 C3 D4 E5 F0 01 12
   23 34 45 56 67 78 89 90 12 13 14 15 16 17 18 19

   D_1:
   80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F

   D_2:
   90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F

   ACPKM’s iteration 1
   Process block 1
```

```
   Input block (CTR):
   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 00

   Output block (G):
   FD 7E F8 9A D9 7E A4 B8 8D B8 B5 1C 1C 9D 6D D0

   Plain text block:
   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88

   Cipher text block:
   EC 5C CB DE 8C 18 D3 B8 72 56 68 D0 A7 37 F4 58

   Process block 2
   Input block (CTR):
   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 01

   Output block (G):
   19 98 C5 71 76 37 FB 17 11 E4 48 F0 0C 0D 60 B2

   Plain text block:
   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A

   Cipher text block:
   19 89 E7 42 32 62 9D 60 99 7D E2 4B C0 E3 9F B8

   Updated key:
   F6 80 D1 21 2F A4 3D F4 EC 3A 91 DE 2A B1 6F 1B
   36 B0 48 8A 4F C1 2E 09 98 D2 E4 A8 88 E8 4F 3D

   ACPKM's iteration 2
   Process block 1
   Input block (CTR):
   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 02

   Output block (G):
   E4 88 89 4F B6 02 87 DB 77 5A 07 D9 2C 89 46 EA

   Plain text block:
   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00

   Cipher text block:
   F5 AA BA 0B E3 64 F0 53 EE F0 BC 15 C2 76 4C EA

   Process block 2
   Input block (CTR):
   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 03

   Output block (G):
```

```
   BC 4F 87 23 DB F0 91 50 DD B4 06 C3 1D A9 7C A4

   Plain text block:
   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11

   Cipher text block:
   9E 7C C3 76 BD 87 19 C9 77 0F CA 2D E2 A3 7C B5

   Updated key:
   8E B9 7E 43 27 1A 42 F1 CA 8E E2 5F 5C C7 C8 3B
   1A CE 9E 5E D0 6A A5 3B 57 B9 6A CF 36 5D 24 B8

   ACPKM’s iteration 3
   Process block 1
   Input block (CTR):
   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 04

   Output block (G):
   68 6F 22 7D 8F B2 9C BD 05 C8 C3 7D 22 FE 3B B7

   Plain text block:
   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22

   Cipher text block:
   5B 2B 77 1B F8 3A 05 17 BE 04 2D 82 28 FE 2A 95

   Process block 2
   Input block (CTR):
   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 05

   Output block (G):
   C0 1B F9 7F 75 6E 12 2F 80 59 55 BD DE 2D 45 87

   Plain text block:
   44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33

   Cipher text block:
   84 4E 9F 08 FD F7 B8 94 4C B7 AA B7 DE 3C 67 B4

   Updated key:
   C5 71 6C C9 67 98 BC 2D 4A 17 87 B7 8A DF 94 AC
   E8 16 F8 0B DB BC AD 7D 60 78 12 9C 0C B4 02 F5

   ACPKM’s iteration 4
   Process block 1
   Input block (CTR):
   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 06
```

```
   Output block (G):
   03 DE 34 74 AB 9B 65 8A 3B 54 1E F8 BD 2B F4 7D

   Plain text block:
   55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33 44

   Cipher text block:
   56 B8 43 FC 32 31 DE 46 D5 AB 14 F8 AC 09 C7 39

   Cipher text C:
   EC 5C CB DE 8C 18 D3 B8 72 56 68 D0 A7 37 F4 58
   19 89 E7 42 32 62 9D 60 99 7D E2 4B C0 E3 9F B8
   F5 AA BA 0B E3 64 F0 53 EE F0 BC 15 C2 76 4C EA
   9E 7C C3 76 BD 87 19 C9 77 0F CA 2D E2 A3 7C B5
   5B 2B 77 1B F8 3A 05 17 BE 04 2D 82 28 FE 2A 95
   84 4E 9F 08 FD F7 B8 94 4C B7 AA B7 DE 3C 67 B4
   56 B8 43 FC 32 31 DE 46 D5 AB 14 F8 AC 09 C7 39



   OMAC-ACPKM-Master mode with AES-256
   ***********************************
   k = 256
   n = 128
   N = 256
   T* = 768

   Initial key K:
   88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77
   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF

   Plaintext M:
   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88
   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A
   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00
   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11
   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22

   D_1:
   80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F

   D_2:
   90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F

   K^1|K^1_1 K^2|K^2_1 K^3|K^3_1
   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60
   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
```

```
AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3
9D CC 66 42 0D FF 45 5B 21 F3 93 F0 D4 D6 6E 67
BB 1B 06 0B 87 66 6D 08 7A 9D A7 49 55 C3 5B 48
F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12
78 21 C7 C7 6C BD 79 63 56 AC F8 8E 69 6A 00 07

OMAC’s iteration 1
K^1:
9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60

K^1_1:
77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0

Block number 1
Plain text:
11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88

Input block:
11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88

Output block:
0B A5 89 BF 55 C1 15 42 53 08 89 76 A0 FE 24 3E

Block number 2
Plain text:
00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A

Input block:
0B B4 AB 8C 11 94 73 35 DB 91 23 CD 6C 10 DB 34

Output block:
1C 53 DD A3 6D DC E1 17 ED 1F 14 09 D8 6A F3 2C


OMAC’s iteration 2
K^2:
AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3
9D CC 66 42 0D FF 45 5B 21 F3 93 F0 D4 D6 6E 67

K^2_1:
BB 1B 06 0B 87 66 6D 08 7A 9D A7 49 55 C3 5B 48

Block number 3
Plain text:
11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00
```

```
   Input block:
   0D 71 EE E7 38 BA 96 9F 74 B5 AF C5 36 95 F9 2C

   Output block:
   4E D4 BC A6 CE 6D 6D 16 F8 63 85 13 E0 48 59 75

   Block number 4
   Plain text:
   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11

   Input block:
   6C E7 F8 F3 A8 1A E5 8F 52 D8 49 FD 1F 42 59 64

   Output block:
   B6 83 E3 96 FD 30 CD 46 79 C1 8B 24 03 82 1D 81


   OMAC’s iteration 3
   K^3:
   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12

   K^3_1:
   78 21 C7 C7 6C BD 79 63 56 AC F8 8E 69 6A 00 07

   MSB1(K1) == 0 -> K2 = K1 << 1

   Last block
   K1:
   78 21 C7 C7 6C BD 79 63 56 AC F8 8E 69 6A 00 07

   K2:
   F0 43 8F 8E D9 7A F2 C6 AD 59 F1 1C D2 D4 00 0E

   Block number 5
   Plain text:
   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22

   Using K1, src doesn’t require padding
   Input block:
   FD E6 71 37 E6 05 2D 8F 94 A1 9D 55 60 E8 0C A4

   Output block:
   B3 AD B8 92 18 32 05 4C 09 21 E7 B8 08 CF A0 B8

   Message authentication code T:
   B3 AD B8 92 18 32 05 4C 09 21 E7 B8 08 CF A0 B8
```

Appendix B. Contributors

- Russ Housley
  Vigil Security, LLC
  housley@vigilsec.com

- Evgeny Alekseev
  CryptoPro
  alekseev@cryptopro.ru

- Ekaterina Smyshlyaeva
  CryptoPro
  ess@cryptopro.ru

- Shay Gueron
  University of Haifa, Israel
  Intel Corporation, Israel Development Center, Israel
  shay.gueron@gmail.com

- Daniel Fox Franke
  Akamai Technologies
  dfoxfranke@gmail.com

- Lilia Ahmetzyanova
  CryptoPro
  lah@cryptopro.ru

Appendix C. Acknowledgments

We thank Mihir Bellare, Scott Fluhrer, Dorothy Cooley, Yoav Nir, Jim Schaad, Paul Hoffman and Dmitry Belyavsky for their useful comments.

Author’s Address

Stanislav Smyshlyaev (editor)
CryptoPro
18, Suschevsky val
Moscow 127018
Russian Federation

Phone: +7 (495) 995-48-20
Email: svs@cryptopro.ru

CFRG
Internet-Draft
Intended status: Informational
Expires: December 2, 2019

S. Smyshlyaev, Ed.
CryptoPro

May 31, 2019

Re-keying Mechanisms for Symmetric Keys
draft-irtf-cfrg-re-keying-17

Abstract

A certain maximum amount of data can be safely encrypted when encryption is performed under a single key. This amount is called "key lifetime". This specification describes a variety of methods to increase the lifetime of symmetric keys. It provides two types of re-keying mechanisms based on hash functions and on block ciphers, that can be used with modes of operations such as CTR, GCM, CBC, CFB and OMAC.

This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

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 December 2, 2019.

Table of Contents

1. Introduction

A certain maximum amount of data can be safely encrypted when encryption is performed under a single key. Hereinafter this amount will be referred to as "key lifetime". The need for such a limitation is dictated by the following methods of cryptanalysis:

1. Methods based on the combinatorial properties of the used block cipher mode of operation

   These methods do not depend on the underlying block cipher. Common modes restrictions derived from such methods are of order 2^{n/2}, where n is a block size defined in Section 3. [Sweet32] is an example of attack that is based on such methods.

2. Methods based on side-channel analysis issues

   In most cases these methods do not depend on the used encryption modes and weakly depend on the used block cipher features. Limitations resulting from these considerations are usually the most restrictive ones. [TEMPEST] is an example of attack that is based on such methods.

3. Methods based on the properties of the used block cipher

   The most common methods of this type are linear and differential cryptanalysis [LDC]. In most cases these methods do not depend on the used modes of operation. In case of secure block ciphers, bounds resulting from such methods are roughly the same as the natural bounds of 2^n, and are dominated by the other bounds above. Therefore, they can be excluded from the considerations here.

As a result, it is important to replace a key when the total size of the processed plaintext under that key approaches the lifetime limitation. A specific value of the key lifetime should be determined in accordance with some safety margin for protocol security and the methods outlined above.

Suppose L is a key lifetime limitation in some protocol P. For simplicity, assume that all messages have the same length m. Hence,

the number of messages q that can be processed with a single key K should be such that m * q <= L. This can be depicted graphically as a rectangle with sides m and q which is enclosed by area L (see Figure 1).

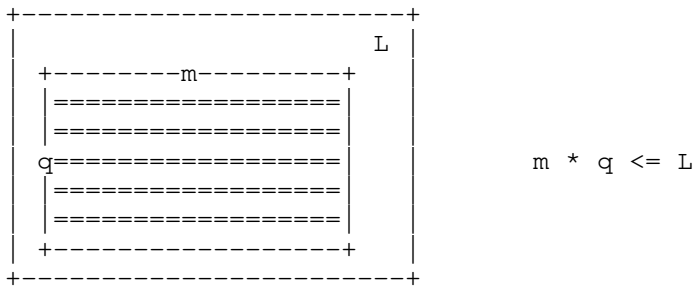


Figure 1: Graphic display of the key lifetime limitation

In practice, such amount of data that corresponds to limitation L may not be enough. The simplest and obvious way in this situation is a regular renegotiation of an initial key after processing this threshold amount of data L. However, this reduces the total performance, since it usually entails termination of application data transmission, additional service messages, the use of random number generator and many other additional calculations, including resource-intensive public key cryptography.

For the protocols based on block ciphers or stream ciphers a more efficient way to increasing the key lifetime is to use various re-keying mechanisms. This specification considers only the case of re-keying mechanisms for block ciphers, while re-keying mechanisms typical for stream ciphers (e.g., [Pietrzak2009], [FPS2012]) case go beyond the scope of this document.

Re-keying mechanisms can be applied on the different protocol levels: on the block cipher level (this approach is known as fresh re-keying and is described, for instance, in [FRESHREKEYING]), on the block cipher mode of operation level (see Section 6), on the protocol level above the block cipher mode of operation (see Section 5). The usage of the first approach is highly inefficient due to the key changing after processing each message block. Moreover, fresh re-keying mechanisms can change the block cipher internal structure, and, consequently, can require the additional security analysis for each particular block cipher. As a result, this approach depends on particular primitive properties and can not be applied to any

arbitrary block cipher without additional security analysis, therefore, fresh re-keying mechanisms go beyond the scope of this document.

Thus, this document contains the list of recommended re-keying mechanisms that can be used in the symmetric encryption schemes based on the block ciphers. These mechanisms are independent from the particular block cipher specification and their security properties rely only on the standard block cipher security assumption.

This specification presents two basic approaches to extend the lifetime of a key while avoiding renegotiation that were introduced in [AAOS2017]:

1. External re-keying

   External re-keying is performed by a protocol, and it is independent of the underlying block cipher and the mode of operation. External re-keying can use parallel and serial constructions. In the parallel case, data processing keys K^1, K^2, ... are generated directly from the initial key K independently of each other. In the serial case, every data processing key depends on the state that is updated after the generation of each new data processing key.

   As a generalization of external parallel re-keying an external tree-based mechanism can be considered. It is specified in the Section 5.2.3 and can be viewed as the [GGM] tree generalization. Similar constructions are used in the one-way tree mechanism ([OWT]) and [AESDUKPT] standard.

2. Internal re-keying

   Internal re-keying is built into the mode, and it depends heavily on the properties of the mode of operation and the block size.

The re-keying approaches extend the key lifetime for a single initial key by providing the possibility to limit the leakages (via side channels) and by improving combinatorial properties of the used block cipher mode of operation.

In practical applications, re-keying can be useful for protocols that need to operate in hostile environments or under restricted resource conditions (e.g., that require lightweight cryptography, where ciphers have a small block size, that imposes strict combinatorial limitations). Moreover, mechanisms that use external or internal re-keying may provide some protection against possible future attacks (by limiting the number of plaintext-ciphertext pairs that an

adversary can collect) and some properties of forward or backward security (meaning that past or future data processing keys remain secure even if the current key is compromised, see for more details [AbBell]). External or internal re-keying can be used in network protocols as well as in the systems for data-at-rest encryption.

Depending on the concrete protocol characteristics there might be situations in which both external and internal re-keying mechanisms (see Section 7) can be applied. For example, the similar approach was used in the Taha's tree construction (see [TAHA]).

Note that there are key updating (key regression) algorithms (e.g., [FKK2005] and [KMNT2003]) which are called "re-keying" as well, but they pursue the goal different from increasing key lifetime. Therefore, key regression algorithms are excluded from the considerations here.

This document represents the consensus of the Crypto Forum Research Group (CFRG).

2. 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].

3. Basic Terms and Definitions

This document uses the following terms and definitions for the sets and operations on the elements of these sets:

V* the set of all bit strings of a finite length (hereinafter referred to as strings), including the empty string;

V_s the set of all bit strings of length s, where s is a non-negative integer;

|X| the bit length of the bit string X;

A | B concatenation of strings A and B both belonging to V*, i.e., a string in V_{|A|+|B|}, where the left substring in V_|A| is equal to A, and the right substring in V_|B| is equal to B;

(xor) exclusive-or of two bit strings of the same length;

Z_{2^n} ring of residues modulo 2^n;

Int_s: V_s -> Z_{2^s} the transformation that maps a string a = (a_s, ... , a_1) in V_s into the integer Int_s(a) = 2^{s-1} * a_s + ... + 2 * a_2 + a_1 (the interpretation of the binary string as an integer);

Vec_s: Z_{2^s} -> V_s the transformation inverse to the mapping Int_s (the interpretation of an integer as a binary string);

MSB_i: V_s -> V_i the transformation that maps the string a = (a_s, ... , a_1) in V_s into the string MSB_i(a) = (a_s, ... , a_{s-i+1}) in V_i (most significant bits);

LSB_i: V_s -> V_i the transformation that maps the string a = (a_s, ... , a_1) in V_s into the string LSB_i(a) = (a_i, ... , a_1) in V_i (least significant bits);

Inc_c: V_s -> V_s the transformation that maps the string a = (a_s, ... , a_1) in V_s into the string Inc_c(a) = MSB_{|a|-c}(a) | Vec_c(Int_c(LSB_c(a)) + 1(mod 2^c)) in V_s (incrementing the least significant c bits of the bit string, regarded as the binary representation of an integer);

a^s the string in V_s that consists of s 'a' bits;

E_{K}: V_n -> V_n the block cipher permutation under the key K in V_k;

ceil(x) the smallest integer that is greater than or equal to x;

floor(x) the biggest integer that is less than or equal to x;

k the bit-length of the K; k is assumed to be divisible by 8;

n the block size of the block cipher (in bits); n is assumed to be divisible by 8;

b the number of data blocks in the plaintext P (b = ceil(|P|/n));

N the section size (the number of bits that are processed with one section key before this key is transformed).

A plaintext message P and the corresponding ciphertext C are divided into b = ceil(|P|/n) blocks, denoted P = P_1 | P_2 | ... | P_b and C = C_1 | C_2 | ... | C_b, respectively. The first b-1 blocks P_i and C_i are in V_n, for i = 1, 2, ... , b-1. The b-th blocks P_b, C_b may be an incomplete blocks, i.e., in V_r, where r <= n if not otherwise specified.

4. Choosing Constructions and Security Parameters

External re-keying is an approach assuming that a key is transformed after encrypting a limited number of entire messages. External re-keying method is chosen at the protocol level, regardless of the underlying block cipher or the encryption mode. External re-keying is recommended for protocols that process relatively short messages or for protocols that have a way to divide a long message into manageable pieces. Through external re-keying the number of messages that can be securely processed with a single initial key K is substantially increased without loss in message length.

External re-keying has the following advantages:

1. it increases the lifetime of an initial key by increasing the number of messages processed with this key;

2. it has minimal impact on performance, when the number of messages processed under one initial key is sufficiently large;

3. it provides forward and backward security of data processing keys.

However, the use of external re-keying has the following disadvantage: in case of restrictive key lifetime limitations the message sizes can become inconvenient due to impossibility of processing sufficiently large messages, so it could be necessary to perform additional fragmentation at the protocol level. E.g. if the key lifetime L is 1 GB and the message length m = 3 GB, then this message cannot be processed as a whole and it should be divided into three fragments that will be processed separately.

Internal re-keying is an approach assuming that a key is transformed during each separate message processing. Such procedures are integrated into the base modes of operations, so every internal re-keying mechanism is defined for the particular operation mode and the block size of the used cipher. Internal re-keying is recommended for protocols that process long messages: the size of each single message can be substantially increased without loss in number of messages that can be securely processed with a single initial key.

Internal re-keying has the following advantages:

1. it increases the lifetime of an initial key by increasing the size of the messages processed with one initial key;

2. it has minimal impact on performance;

3. internal re-keying mechanisms without a master key does not affect short messages transformation at all;

4. it is transparent (works like any mode of operation): does not require changes of IV's and restarting MACing.

However, the use of internal re-keying has the following disadvantages:

1. a specific method must not be chosen independently of a mode of operation;

2. internal re-keying mechanisms without a master key do not provide backward security of data processing keys.

Any block cipher modes of operations with internal re-keying can be jointly used with any external re-keying mechanisms. Such joint usage increases both the number of messages processed with one initial key and their maximum possible size.

If the adversary has access to the data processing interface the use of the same cryptographic primitives both for data processing and re-keying transformation decreases the code size but can lead to some possible vulnerabilities (the possibility of mounting a chosen-plaintext attack may lead to the compromise of the following keys). This vulnerability can be eliminated by using different primitives for data processing and re-keying, e.g., block cipher for data processing and hash for re-keying (see Section 5.2.2 and Section 5.3.2). However, in this case the security of the whole scheme cannot be reduced to standard notions like PRF or PRP, so security estimations become more difficult and unclear.

Summing up the above-mentioned issues briefly:

1. If a protocol assumes processing long records (e.g., [CMS]), internal re-keying should be used. If a protocol assumes processing a significant amount of ordered records, which can be considered as a single data stream (e.g., [TLS], [SSH]), internal re-keying may also be used.

2. For protocols which allow out-of-order delivery and lost records (e.g., [DTLS], [ESP]) external re-keying should be used as in this case records cannot be considered as a single data stream. If at the same time records are long enough, internal re-keying should be additionally used during each separate message processing.

For external re-keying:

1. If it is desirable to separate transformations used for data processing and for key update, hash function based re-keying should be used.

2. If parallel data processing is required, then parallel external re-keying should be used.

3. In case of restrictive key lifetime limitations external tree-based re-keying should be used.

For internal re-keying:

1. If the property of forward and backward security is desirable for data processing keys and if additional key material can be easily obtained for the data processing stage, internal re-keying with a master key should be used.

5. External Re-keying Mechanisms

This section presents an approach to increase the initial key lifetime by using a transformation of a data processing key (frame key) after processing a limited number of entire messages (frame). It provides external parallel and serial re-keying mechanisms (see [AbBell]). These mechanisms use initial key K only for frame keys generation and never use it directly for data processing. Such mechanisms operate outside of the base modes of operations and do not change them at all, therefore they are called "external re-keying" mechanisms in this document.

External re-keying mechanisms are recommended for usage in protocols that process quite small messages, since the maximum gain in increasing the initial key lifetime is achieved by increasing the number of messages.

External re-keying increases the initial key lifetime through the following approach. Suppose there is a protocol P with some mode of operation (base encryption or authentication mode). Let L1 be a key lifetime limitation induced by side-channel analysis methods (side-channel limitation), let L2 be a key lifetime limitation induced by methods based on the combinatorial properties of a used mode of operation (combinatorial limitation) and let q1, q2 be the total numbers of messages of length m, that can be safely processed with an initial key K according to these limitations.

Let L = min(L1, L2), q = min (q1, q2), q * m <= L. As L1 limitation is usually much stronger than L2 limitation (L1 < L2), the final key lifetime restriction is equal to the most restrictive limitation L1.

Thus, as displayed in Figure 2, without re-keying only q1 (q1 * m <= L1) messages can be safely processed.

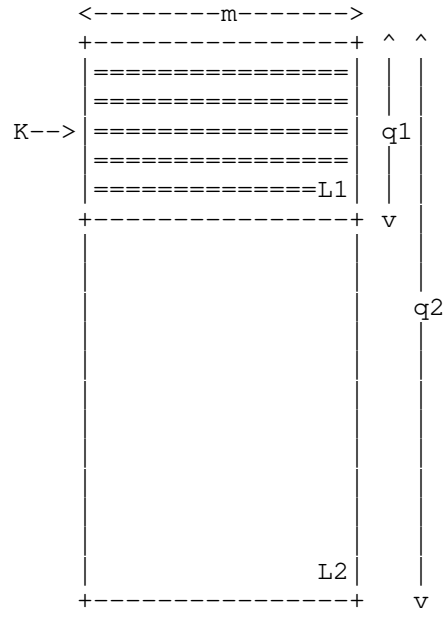


Figure 2: Basic principles of message processing without external re-keying

Suppose that the safety margin for the protocol P is fixed and the external re-keying approach is applied to the initial key K to generate the sequence of frame keys. The frame keys are generated in such a way that the leakage of a previous frame key does not have any impact on the following one, so the side channel limitation L1 goes off. Thus, the resulting key lifetime limitation of the initial key K can be calculated on the basis of a new combinatorial limitation L2'. It is proven (see [AbBell]) that the security of the mode of operation that uses external re-keying leads to an increase when compared to base mode without re-keying (thus, L2 < L2'). Hence, as displayed in Figure 3, the resulting key lifetime limitation in case of using external re-keying can be increased up to L2'.

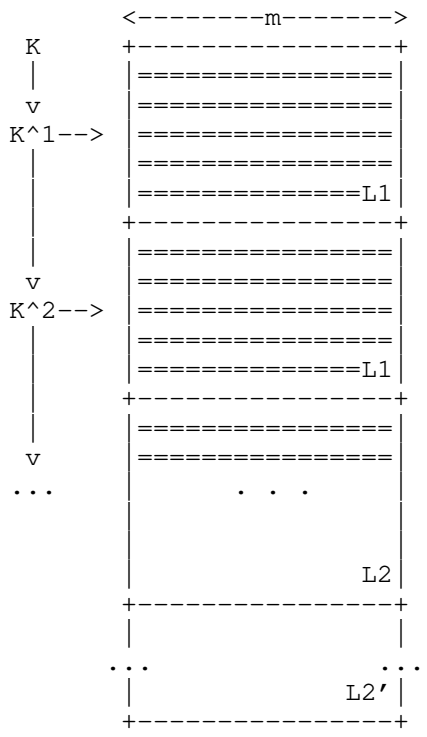


Figure 3: Basic principles of message processing
with external re-keying

Note: the key transformation process is depicted in a simplified form. A specific approach (parallel and serial) is described below.

Consider an example. Let the message size in a protocol P be equal to 1 KB. Suppose L1 = 128 MB and L2 = 1 TB. Thus, if an external re-keying mechanism is not used, the initial key K must be renegotiated after processing 128 MB / 1 KB = 131072 messages.

If an external re-keying mechanism is used, the key lifetime limitation L1 goes off. Hence the resulting key lifetime limitation L2' can be set to more then 1 TB. Thus if an external re-keying mechanism is used, more then 1 TB / 1 KB = 2^30 messages can be processed before the initial key K is renegotiated. This is 8192 times greater than the number of messages that can be processed, when external re-keying mechanism is not used.

5.1. Methods of Key Lifetime Control

Suppose L is an amount of data that can be safely processed with one frame key. For i in {1, 2, ... , t} the frame key K^i (see Figure 4 and Figure 6) should be transformed after processing q_i messages, where q_i can be calculated in accordance with one of the following approaches:

Explicit approach:

q_i is such that $|M^{i,1}| + ... + |M^{i,q_i}| <= L$, $|M^{i,1}| + ... + |M^{i,q_i+1}| > L$.
This approach allows to use the frame key K^i in almost optimal way but it can be applied only in case when messages cannot be lost or reordered (e.g., TLS records).

Implicit approach:

q_i = L / m_max, i = 1, ... , t.
The amount of data processed with one frame key K^i is calculated under the assumption that every message has the maximum length m_max. Hence this amount can be considerably less than the key lifetime limitation L. On the other hand, this approach can be applied in case when messages may be lost or reordered (e.g., DTLS records).

Dynamic key changes:

We can organize the key change using the Protected Point to Point ([P3]) solution by building a protected tunnel between the endpoints in which the information about frame key updating can be safely passed across. This can be useful, for example, when we wish the adversary not to detect the key change during the protocol evaluation.

5.2. Parallel Constructions

External parallel re-keying mechanisms generate frame keys K^1, K^2, ... directly from the initial key K independently of each other.

The main idea behind external re-keying with a parallel construction is presented in Figure 4:

Maximum message size = m_max.

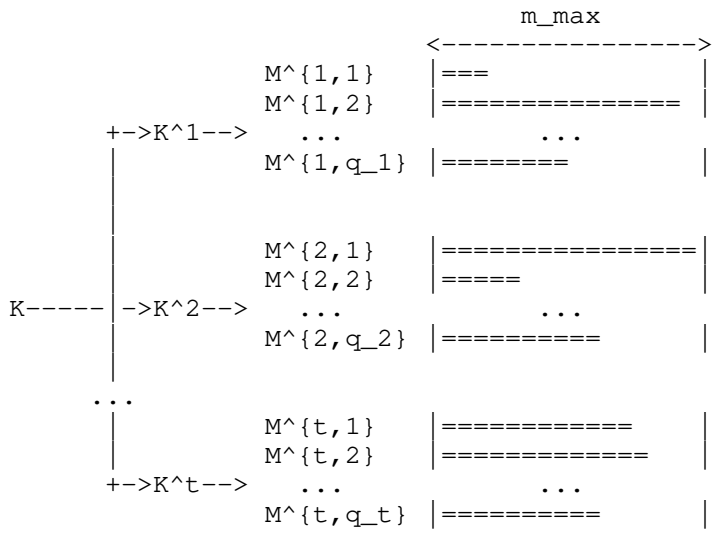


Figure 4: External parallel re-keying mechanisms

The frame key K^i, i = 1, ... , t-1, is updated after processing a certain amount of messages (see Section 5.1).

5.2.1. Parallel Construction Based on a KDF on a Block Cipher

ExtParallelC re-keying mechanism is based on the key derivation function on a block cipher and is used to generate t frame keys as follows:

```
K^1 | K^2 | ... | K^t = ExtParallelC(K, t * k) = MSB_{t *
k}(E_{K}(Vec_n(0)) |
E_{K}(Vec_n(1)) | ... | E_{K}(Vec_n(R - 1))),
```

where R = ceil(t * k/n).

5.2.2. Parallel Construction Based on a KDF on a Hash Function

ExtParallelH re-keying mechanism is based on the key derivation function HKDF-Expand, described in [RFC5869], and is used to generate t frame keys as follows:

K^1 | K^2 | ... | K^t = ExtParallelH(K, t * k) = HKDF-Expand(K, label, t * k),

where label is a string (may be a zero-length string) that is defined by a specific protocol.

5.2.3. Tree-based Construction

The application of external tree-based mechanism leads to the construction of the key tree with the initial key K (root key) at the 0-level and the frame keys K^1, K^2, ... at the last level as described in Figure 5.

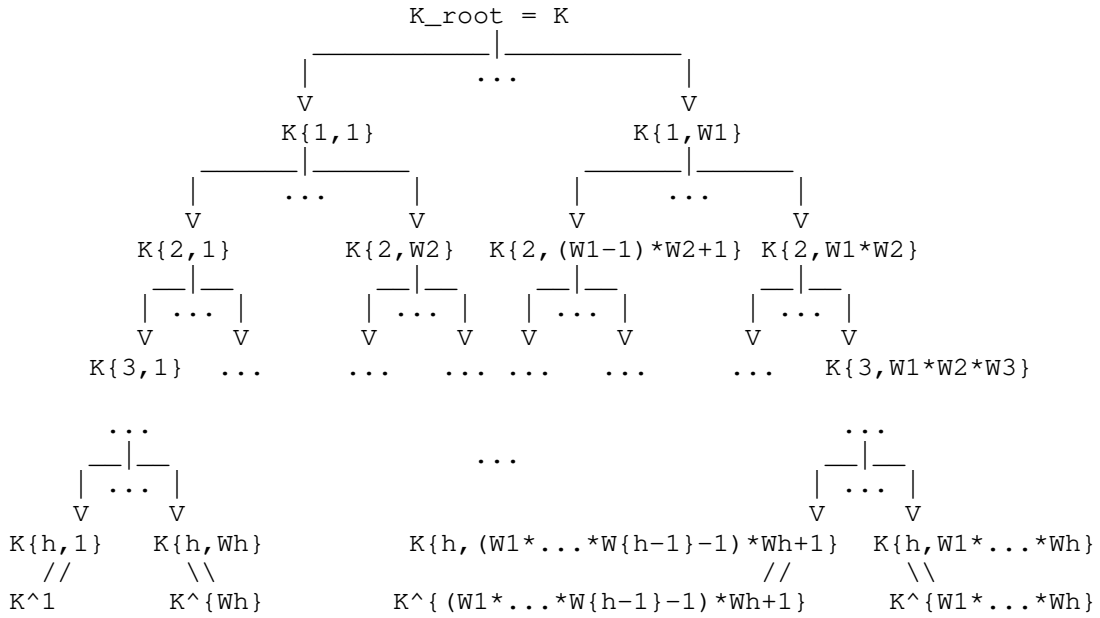


Figure 5: External Tree-based Mechanism

The tree height h and the number of keys Wj, j in {1, ... , h}, which can be partitioned from "parent" key, are defined in accordance with a specific protocol and key lifetime limitations for the used derivation functions.

Each j-level key K{j,w}, where j in {1, ... , h}, w in {1, ... , W1 * ... * Wj}, is derived from the (j-1)-level "parent" key K{j-1,ceil(w/Wi)} (and other appropriate input data) using the j-th level

derivation function that can be based on the block cipher function or on the hash function and that is defined in accordance with a specific protocol.

The i-th frame K^i, i in {1, 2, ... , W1*...*Wh}, can be calculated as follows:

```
K^i = ExtKeyTree(K, i) = KDF_h(KDF_{h-1}(... KDF_1(K, ceil(i / (W2
* ... * Wh)) ... , ceil(i / Wh)), i),
```

where KDF_j is the j-th level derivation function that takes two arguments (the parent key value and the integer in range from 1 to W1 * ... * Wj) and outputs the j-th level key value.

The frame key K^i is updated after processing a certain amount of messages (see Section 5.1).

In order to create an efficient implementation, during frame key K^i generation the derivation functions KDF_j, j in {1, ... , h-1}, should be used only in case when ceil(i / (W{j+1} * ... * Wh)) != ceil((i - 1) / (W{j+1} * ... * Wh)); otherwise it is necessary to use previously generated value. This approach also makes it possible to take countermeasures against side channels attacks.

Consider an example. Suppose h = 3, W1 = W2 = W3 = W and KDF_1, KDF_2, KDF_3 are key derivation functions based on the KDF_GOSTR3411_2012_256 (hereafter simply KDF) function described in [RFC7836]. The resulting ExtKeyTree function can be defined as follows:

```
ExtKeyTree(K, i) = KDF(KDF(KDF(K, "level1", ceil(i / W^2)),
"level2", ceil(i / W)), "level3", i).
```

where i in {1, 2, ... , W^3}.

The structure similar to external tree-based mechanism can be found in Section 6 of [NISTSP800-108].

5.3. Serial Constructions

External serial re-keying mechanisms generate frame keys, each of which depends on the secret state (K*_1, K*_2, ..., see Figure 6) that is updated after the generation of each new frame key. Similar approaches are used in the [SIGNAL] protocol, in the [TLS] updating traffic keys mechanism and were proposed for use in the [U2F] protocol.

External serial re-keying mechanisms have the obvious disadvantage of the impossibility to be implemented in parallel, but they can be preferred if additional forward secrecy is desirable: in case all keys are securely deleted after usage, compromise of a current secret state at some time does not lead to a compromise of all previous secret states and frame keys. In terms of [TLS], compromise of application_traffic_secret_N does not compromise all previous application_traffic_secret_i, i < N.

The main idea behind external re-keying with a serial construction is presented in Figure 6:

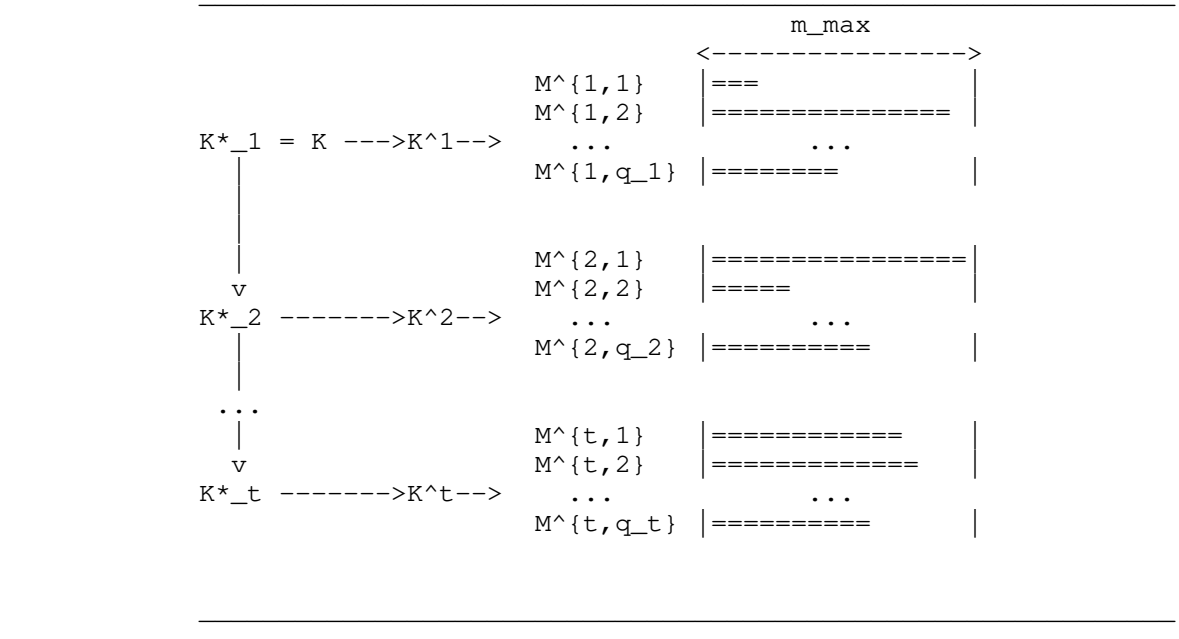


Figure 6: External serial re-keying mechanisms

The frame key K^i, i = 1, ... , t - 1, is updated after processing a certain amount of messages (see Section 5.1).

5.3.1. Serial Construction Based on a KDF on a Block Cipher

The frame key K^i is calculated using ExtSerialC transformation as follows:

```
K^i = ExtSerialC(K, i) =
MSB_k(E_{K*_i}(Vec_n(0)) |E_{K*_i}(Vec_n(1)) | ... |
E_{K*_i}(Vec_n(J - 1))),
```

where J = ceil(k / n), i = 1, ... , t, K*_i is calculated as follows:

```
K*_1 = K,

K*_{j+1} = MSB_k(E_{K*_j}(Vec_n(J)) | E_{K*_j}(Vec_n(J + 1)) |
... |
E_{K*_j}(Vec_n(2 * J - 1))),
```

where j = 1, ... , t - 1.

5.3.2. Serial Construction Based on a KDF on a Hash Function

The frame key K^i is calculated using ExtSerialH transformation as follows:

```
K^i = ExtSerialH(K, i) = HKDF-Expand(K*_i, label1, k),
```

where i = 1, ... , t, HKDF-Expand is the HMAC-based key derivation function, described in [RFC5869], K*_i is calculated as follows:

```
K*_1 = K,

K*_{j+1} = HKDF-Expand(K*_j, label2, k), where j = 1, ... , t - 1,
```

where label1 and label2 are different strings from V* that are defined by a specific protocol (see, for example, TLS 1.3 updating traffic keys algorithm [TLS]).

5.4. Using Additional Entropy during Re-keying

In many cases using additional entropy during re-keying won't increase security, but may give a false sense of that, therefore one can rely on additional entropy only after conducting a deep security analysis. For example, good PRF constructions do not require additional entropy for the quality of keys, so in most cases there is no need for using additional entropy with external re-keying mechanisms based on secure KDFs. However, in some situations mixed-in entropy can still increase security in the case of a time-limited but complete breach of the system, when an adversary can access the frame keys generation interface, but cannot reveal master keys (e.g., when master keys are stored in an HSM).

For example, an external parallel construction based on a KDF on a Hash function with a mixed-in entropy can be described as follows:

K^i = HKDF-Expand(K, label_i, k),

where label_i is additional entropy that must be sent to the recipient (e.g., be sent jointly with encrypted message). The entropy label_i and the corresponding key K^i must be generated directly before message processing.

6. Internal Re-keying Mechanisms

This section presents an approach to increase the key lifetime by using a transformation of a data processing key (section key) during each separate message processing. Each message is processed starting with the same key (the first section key) and each section key is updated after processing N bits of message (section).

This section provides internal re-keying mechanisms called ACPKM (Advanced Cryptographic Prolongation of Key Material) and ACPKM-Master that do not use a master key and use a master key respectively. Such mechanisms are integrated into the base modes of operation and actually form new modes of operation, therefore they are called "internal re-keying" mechanisms in this document.

Internal re-keying mechanisms are recommended to be used in protocols that process large single messages (e.g., CMS messages), since the maximum gain in increasing the key lifetime is achieved by increasing the length of a message, while it provides almost no increase in the number of messages that can be processed with one initial key.

Internal re-keying increases the key lifetime through the following approach. Suppose protocol P uses some base mode of operation. Let L1 and L2 be a side channel and combinatorial limitations respectively and for some fixed amount of messages q let m1, m2 be the lengths of messages, that can be safely processed with a single initial key K according to these limitations.

Thus, by analogy with the Section 5 without re-keying the final key lifetime restriction, as displayed in Figure 7, is equal to L1 and only q messages of the length m1 can be safely processed.

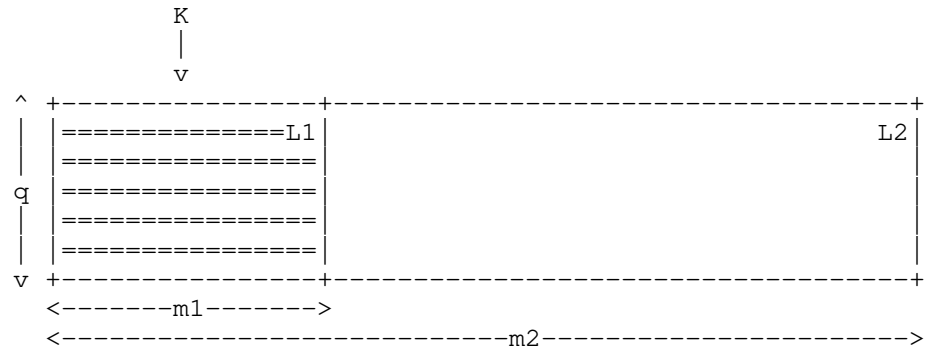


Figure 7: Basic principles of message processing
without internal re-keying

Suppose that the safety margin for the protocol P is fixed and internal re-keying approach is applied to the base mode of operation. Suppose further that every message is processed with a section key, which is transformed after processing N bits of data, where N is a parameter. If q * N does not exceed L1 then the side channel limitation L1 goes off and the resulting key lifetime limitation of the initial key K can be calculated on the basis of a new combinatorial limitation L2'. The security of the mode of operation that uses internal re-keying increases when compared to base mode of operation without re-keying (thus, L2 < L2'). Hence, as displayed in Figure 8, the resulting key lifetime limitation in case of using internal re-keying can be increased up to L2'.

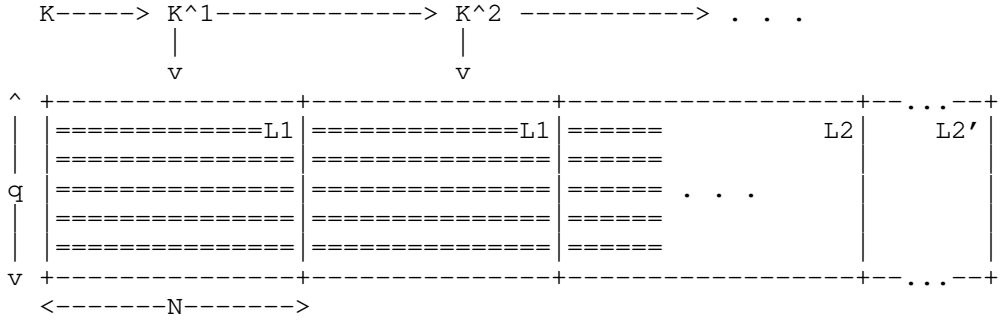


Figure 8: Basic principles of message processing
with internal re-keying

Note: the key transformation process is depicted in a simplified form. A specific approach (ACPKM and ACPKM-Master re-keying mechanisms) is described below.

Since the performance of encryption can slightly decrease for rather small values of N, the parameter N should be selected for a particular protocol as maximum possible to provide necessary key lifetime for the considered security models.

Consider an example. Suppose L1 = 128 MB and L2 = 10 TB. Let the message size in the protocol be large/unlimited (may exhaust the whole key lifetime L2). The most restrictive resulting key lifetime limitation is equal to 128 MB.

Thus, there is a need to put a limit on the maximum message size m_max. For example, if m_max = 32 MB, it may happen that the renegotiation of initial key K would be required after processing only four messages.

If an internal re-keying mechanism with section size N = 1 MB is used, more than L1 / N = 128 MB / 1 MB = 128 messages can be processed before the renegotiation of initial key K (instead of 4 messages in case when an internal re-keying mechanism is not used). Note that only one section of each message is processed with the section key K^i, and, consequently, the key lifetime limitation L1 goes off. Hence the resulting key lifetime limitation L2' can be set to more then 10 TB (in the case when a single large message is processed using the initial key K).

6.1. Methods of Key Lifetime Control

Suppose L is an amount of data that can be safely processed with one section key, N is a section size (fixed parameter). Suppose M^{i}_1 is the first section of message M^{i}, i = 1, ... , q (see Figure 9 and Figure 10), then the parameter q can be calculated in accordance with one of the following two approaches:

- Explicit approach:
  q_i is such that $|M^{1}_1| + ... + |M^{q}_1| <= L$, $|M^{1}_1| + ... + |M^{q+1}_1| > L$
  This approach allows to use the section key K^i in an almost optimal way but it can be applied only in case when messages cannot be lost or reordered (e.g., TLS records).

- Implicit approach:
  q = L / N.
  The amount of data processed with one section key K^i is calculated under the assumption that the length of every message

is equal or greater than section size N and so it can be considerably less than the key lifetime limitation L. On the other hand, this approach can be applied in case when messages may be lost or reordered (e.g., DTLS records).

6.2. Constructions that Do Not Require Master Key

This section describes the block cipher modes that use the ACPKM re-keying mechanism, which does not use a master key: an initial key is used directly for the data encryption.

6.2.1. ACPKM Re-keying Mechanisms

This section defines periodical key transformation without a master key, which is called ACPKM re-keying mechanism. This mechanism can be applied to one of the base encryption modes (CTR and GCM block cipher modes) for getting an extension of this encryption mode that uses periodical key transformation without a master key. This extension can be considered as a new encryption mode.

An additional parameter that defines functioning of base encryption modes with the ACPKM re-keying mechanism is the section size N. The value of N is measured in bits and is fixed within a specific protocol based on the requirements of the system capacity and the key lifetime. The section size N MUST be divisible by the block size n.

The main idea behind internal re-keying without a master key is presented in Figure 9:

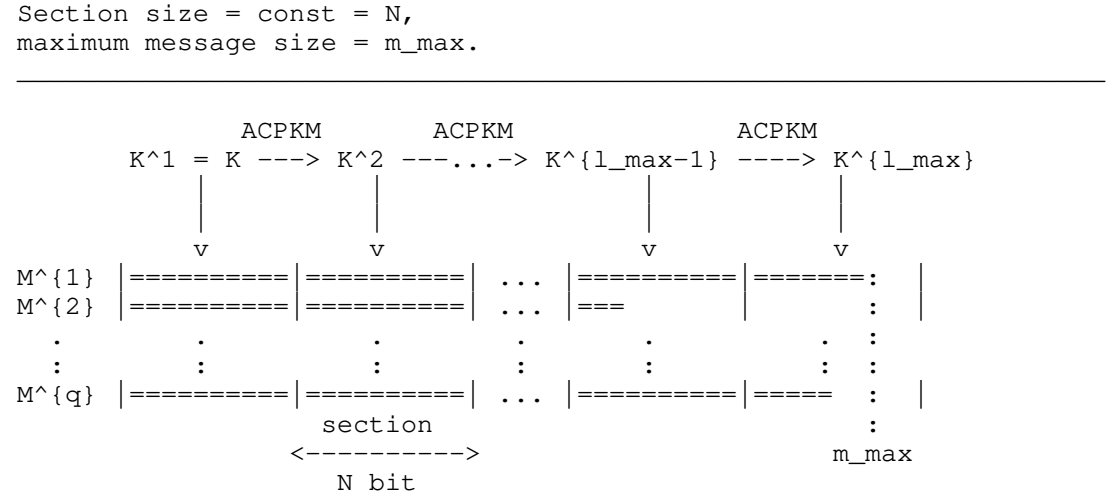


Figure 9: Internal re-keying without a master key

During the processing of the input message M with the length m in some encryption mode that uses ACPKM key transformation of the initial key K the message is divided into l = ceil(m / N) sections (denoted as M = M_1 | M_2 | ... | M_l, where M_i is in V_N for i in {1, 2, ... , l - 1} and M_l is in V_r, r <= N). The first section of each message is processed with the section key K^1 = K. To process the (i + 1)-th section of each message the section key K^{i+1} is calculated using ACPKM transformation as follows:

$$K^{i+1} = ACPKM(K^i) = MSB_k(E_{K^i}(D_1) \mid ... \mid E_{K^i}(D_J)),$$

where J = ceil(k/n) and D_1, D_2, ... , D_J are in V_n and are calculated as follows:

$$D_1 \mid D_2 \mid ... \mid D_J = MSB_{J * n}(D),$$

where D is the following constant in V_{1024}:

```
D = ( 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87
    | 88 | 89 | 8a | 8b | 8c | 8d | 8e | 8f
    | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97
    | 98 | 99 | 9a | 9b | 9c | 9d | 9e | 9f
    | a0 | a1 | a2 | a3 | a4 | a5 | a6 | a7
    | a8 | a9 | aa | ab | ac | ad | ae | af
    | b0 | b1 | b2 | b3 | b4 | b5 | b6 | b7
    | b8 | b9 | ba | bb | bc | bd | be | bf
    | c0 | c1 | c2 | c3 | c4 | c5 | c6 | c7
    | c8 | c9 | ca | cb | cc | cd | ce | cf
    | d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7
    | d8 | d9 | da | db | dc | dd | de | df
    | e0 | e1 | e2 | e3 | e4 | e5 | e6 | e7
    | e8 | e9 | ea | eb | ec | ed | ee | ef
    | f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7
    | f8 | f9 | fa | fb | fc | fd | fe | ff)
```

N o t e : The constant D is such that D_1, ... , D_J are pairwise different for any allowed n and k values.

N o t e : The highest bit of each octet of the constant D is equal to 1. This condition is important, as in conjunction with a certain mode message length limitation it allows to prevent collisions of block cipher permutation inputs in cases of key transformation and message processing (for more details see Section 4.4 of [AAOS2017]).

6.2.2. CTR-ACPKM Encryption Mode

This section defines a CTR-ACPKM encryption mode that uses the ACPKM internal re-keying mechanism for the periodical key transformation.

The CTR-ACPKM mode can be considered as the base encryption mode CTR (see [MODES]) extended by the ACPKM re-keying mechanism.

The CTR-ACPKM encryption mode can be used with the following parameters:

- 64 <= n <= 512;
- 128 <= k <= 512;
- the number c of bits in a specific part of the block to be incremented is such that 32 <= c <= 3 / 4 n, c is a multiple of 8;
- the maximum message size m_max = n * 2^{c-1}.

The CTR-ACPKM mode encryption and decryption procedures are defined as follows:

```
+-------------------------------------------------------------------+
|  CTR-ACPKM-Encrypt(N, K, ICN, P)                                  |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - initial counter nonce ICN in V_{n-c},                          |
|  - plaintext P = P_1 | ... | P_b, |P| <= m_max.                   |
|  Output:                                                          |
|  - ciphertext C.                                                  |
|-------------------------------------------------------------------|
|  1. CTR_1 = ICN | 0^c                                             |
|  2. For j = 2, 3, ... , b do                                      |
|         CTR_{j} = Inc_c(CTR_{j-1})                                |
|  3. K^1 = K                                                       |
|  4. For i = 2, 3, ... , ceil(|P| / N)                             |
|         K^i = ACPKM(K^{i-1})                                      |
|  5. For j = 1, 2, ... , b do                                      |
|         i = ceil(j * n / N),                                      |
|         G_j = E_{K^i}(CTR_j)                                      |
|  6. C = P (xor) MSB_{|P|}(G_1 | ... | G_b)                        |
|  7. Return C                                                      |
+-------------------------------------------------------------------+
```

```
+-------------------------------------------------------------------+
|  CTR-ACPKM-Decrypt(N, K, ICN, C)                                  |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - initial counter nonce ICN in V_{n-c},                          |
|  - ciphertext C = C_1 | ... | C_b, |C| <= m_max.                   |
|  Output:                                                          |
|  - plaintext P.                                                   |
|-------------------------------------------------------------------|
|  1. P = CTR-ACPKM-Encrypt(N, K, ICN, C)                           |
|  2. Return P                                                      |
+-------------------------------------------------------------------+
```

The initial counter nonce ICN value for each message that is encrypted under the given initial key K must be chosen in a unique manner.

6.2.3. GCM-ACPKM Authenticated Encryption Mode

This section defines GCM-ACPKM authenticated encryption mode that uses the ACPKM internal re-keying mechanism for the periodical key transformation.

The GCM-ACPKM mode can be considered as the base authenticated encryption mode GCM (see [GCM]) extended by the ACPKM re-keying mechanism.

The GCM-ACPKM authenticated encryption mode can be used with the following parameters:

- n in {128, 256};
- 128 <= k <= 512;
- the number c of bits in a specific part of the block to be incremented is such that 1 / 4 n <= c <= 1 / 2 n, c is a multiple of 8;
- authentication tag length t;
- the maximum message size m_max = min{n * (2^{c-1} - 2), 2^{n/2} - 1}.

The GCM-ACPKM mode encryption and decryption procedures are defined as follows:

```
+-------------------------------------------------------------------+
|  GHASH(X, H)                                                      |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - bit string X = X_1 | ... | X_m, X_1, ... , X_m in V_n.         |
|  Output:                                                          |
|  - block GHASH(X, H) in V_n.                                      |
|-------------------------------------------------------------------|
|  1. Y_0 = 0^n                                                     |
|  2. For i = 1, ... , m do                                         |
|         Y_i = (Y_{i-1} (xor) X_i) * H                             |
|  3. Return Y_m                                                    |
+-------------------------------------------------------------------+

+-------------------------------------------------------------------+
|  GCTR(N, K, ICB, X)                                               |
|-------------------------------------------------------------------|
|  Input:                                                           |
```

```
|  - section size N,                                                  |
|  - initial key K,                                                   |
|  - initial counter block ICB,                                       |
|  - X = X_1 | ... | X_b.                                             |
|  Output:                                                            |
|  - Y in V_{|X|}.                                                    |
|---------------------------------------------------------------------|
|  1. If X in V_0 then return Y, where Y in V_0                       |
|  2. GCTR_1 = ICB                                                    |
|  3. For i = 2, ... , b do                                           |
|        GCTR_i = Inc_c(GCTR_{i-1})                                   |
|  4. K^1 = K                                                         |
|  5. For j = 2, ... , ceil(|X| / N)                                  |
|        K^j = ACPKM(K^{j-1})                                         |
|  6. For i = 1, ... , b do                                           |
|        j = ceil(i * n / N),                                         |
|        G_i = E_{K_j}(GCTR_i)                                        |
|  7. Y = X (xor) MSB_{|X|}(G_1 | ... | G_b)                          |
|  8. Return Y                                                        |
+---------------------------------------------------------------------+

+---------------------------------------------------------------------+
|  GCM-ACPKM-Encrypt(N, K, ICN, P, A)                                 |
|---------------------------------------------------------------------|
|  Input:                                                             |
|  - section size N,                                                  |
|  - initial key K,                                                   |
|  - initial counter nonce ICN in V_{n-c},                            |
|  - plaintext P = P_1 | ... | P_b, |P| <= m_max,                     |
|  - additional authenticated data A.                                 |
|  Output:                                                            |
|  - ciphertext C,                                                    |
|  - authentication tag T.                                            |
|---------------------------------------------------------------------|
|  1. H = E_{K}(0^n)                                                  |
|  2. ICB_0 = ICN | 0^{c-1} | 1                                       |
|  3. C = GCTR(N, K, Inc_c(ICB_0), P)                                 |
|  4. u = n * ceil(|C| / n) - |C|                                     |
|     v = n * ceil(|A| / n) - |A|                                     |
|  5. S = GHASH(A | 0^v | C | 0^u | Vec_{n/2}(|A|) |                |
|                | Vec_{n/2}(|C|), H)                                 |
|  6. T = MSB_t(E_{K}(ICB_0) (xor) S)                                 |
|  7. Return C | T                                                    |
+---------------------------------------------------------------------+

+---------------------------------------------------------------------+
|  GCM-ACPKM-Decrypt(N, K, ICN, A, C, T)                              |
|---------------------------------------------------------------------|
```

```
|  Input:                                                              |
|  - section size N,                                                   |
|  - initial key K,                                                    |
|  - initial counter block ICN,                                        |
|  - additional authenticated data A,                                  |
|  - ciphertext C = C_1 | ... | C_b, |C| <= m_max,                    |
|  - authentication tag T.                                             |
|  Output:                                                             |
|  - plaintext P or FAIL.                                              |
|----------------------------------------------------------------------|
|  1. H = E_{K}(0^n)                                                   |
|  2. ICB_0 = ICN | 0^{c-1} | 1                                        |
|  3. P = GCTR(N, K, Inc_c(ICB_0), C)                                  |
|  4. u = n * ceil(|C| / n) - |C|                                      |
|     v = n * ceil(|A| / n) - |A|                                      |
|  5. S = GHASH(A | 0^v | C | 0^u | Vec_{n/2}(|A|) |                  |
|               | Vec_{n/2}(|C|), H)                                   |
|  6. T' = MSB_t(E_{K}(ICB_0) (xor) S)                                 |
|  7. If T = T' then return P; else return FAIL                        |
+----------------------------------------------------------------------+
```

The * operation on (pairs of) the 2^n possible blocks corresponds to the multiplication operation for the binary Galois (finite) field of 2^n elements defined by the polynomial f as follows (by analogy with [GCM]):

n = 128: f = a^128 + a^7 + a^2 + a^1 + 1,

n = 256: f = a^256 + a^10 + a^5 + a^2 + 1.

The initial counter nonce ICN value for each message that is encrypted under the given initial key K must be chosen in a unique manner.

The key for computing values E_{K}(ICB_0) and H is not updated and is equal to the initial key K.

6.3. Constructions that Require Master Key

This section describes the block cipher modes that use the ACPKM-Master re-keying mechanism, which use the initial key K as a master key, so K is never used directly for data processing but is used for key derivation.

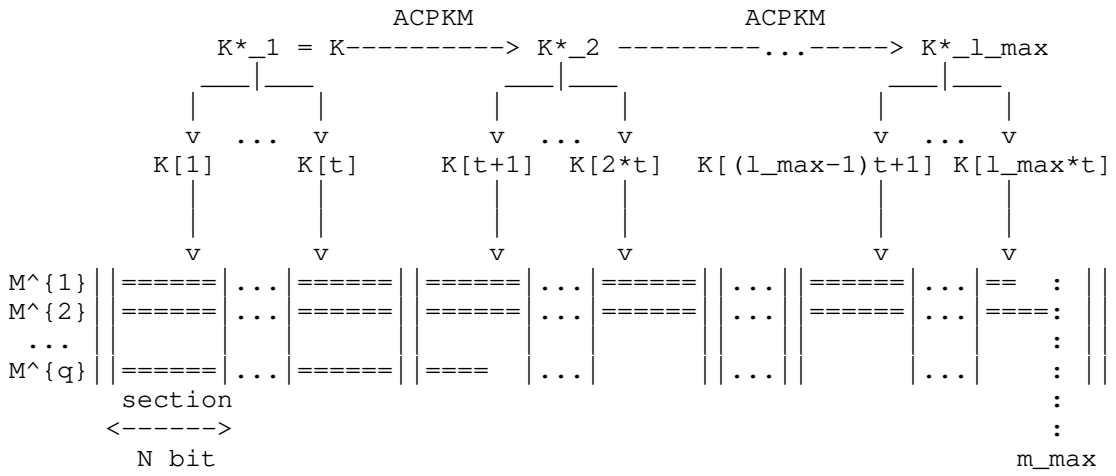
6.3.1. ACPKM-Master Key Derivation from the Master Key

This section defines periodical key transformation with a master key, which is called ACPKM-Master re-keying mechanism. This mechanism can be applied to one of the base modes of operation (CTR, GCM, CBC, CFB, OMAC modes) for getting an extension that uses periodical key transformation with a master key. This extension can be considered as a new mode of operation.

Additional parameters that define the functioning of modes of operation that use the ACPKM-Master re-keying mechanism are the section size N, the change frequency T* of the master keys K*_1, K*_2, ... (see Figure 10) and the size d of the section key material. The values of N and T* are measured in bits and are fixed within a specific protocol, based on the requirements of the system capacity and the key lifetime. The section size N MUST be divisible by the block size n. The master key frequency T* MUST be divisible by d and by n.

The main idea behind internal re-keying with a master key is presented in Figure 10:

Master key frequency T*,
section size N,
maximum message size = m_max.



|K[i]| = d,
t = T* / d,
l_max = ceil(m_max / (N * t)).

Figure 10: Internal re-keying with a master key

During the processing of the input message M with the length m in some mode of operation that uses ACPKM-Master key transformation with the initial key K and the master key frequency T* the message M is divided into l = ceil(m / N) sections (denoted as M = M_1 | M_2 | ... | M_l, where M_i is in V_N for i in {1, 2, ... , l - 1} and M_l is in V_r, r <= N). The j-th section of each message is processed with the key material K[j], j in {1, ... , l}, |K[j]| = d, that is calculated with the ACPKM-Master algorithm as follows:

K[1] | ... | K[l] = ACPKM-Master(T*, K, d, l) = CTR-ACPKM-Encrypt (T*, K, 1^{n/2}, 0^{d*l}).

Note: the parameters d and l MUST be such that d * l <= n * 2^{n/2-1}.

6.3.2. CTR-ACPKM-Master Encryption Mode

This section defines a CTR-ACPKM-Master encryption mode that uses the ACPKM-Master internal re-keying mechanism for the periodical key transformation.

The CTR-ACPKM-Master encryption mode can be considered as the base encryption mode CTR (see [MODES]) extended by the ACPKM-Master re-keying mechanism.

The CTR-ACPKM-Master encryption mode can be used with the following parameters:

- 64 <= n <= 512;
- 128 <= k <= 512;
- the number c of bits in a specific part of the block to be incremented is such that 32 <= c <= 3 / 4 n, c is a multiple of 8;
- the maximum message size m_max = min{N * (n * 2^{n/2-1} / k), n * 2^c}.

The key material K[j] that is used for one section processing is equal to K^j, |K^j| = k bits.

The CTR-ACPKM-Master mode encryption and decryption procedures are defined as follows:

```
+-------------------------------------------------------------------+
|  CTR-ACPKM-Master-Encrypt(N, K, T*, ICN, P)                       |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - master key frequency T*,                                       |
|  - initial counter nonce ICN in V_{n-c},                          |
|  - plaintext P = P_1 | ... | P_b, |P| <= m_max.                   |
|  Output:                                                          |
|  - ciphertext C.                                                  |
|-------------------------------------------------------------------|
|  1. CTR_1 = ICN | 0^c                                             |
|  2. For j = 2, 3, ... , b do                                      |
|        CTR_{j} = Inc_c(CTR_{j-1})                                 |
|  3. l = ceil(|P| / N)                                             |
|  4. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)                   |
|  5. For j = 1, 2, ... , b do                                      |
|        i = ceil(j * n / N),                                       |
|        G_j = E_{K^i}(CTR_j)                                       |
|  6. C = P (xor) MSB_{|P|}(G_1 | ... |G_b)                         |
|  7. Return C                                                      |
+-------------------------------------------------------------------+

+-------------------------------------------------------------------+
|  CTR-ACPKM-Master-Decrypt(N, K, T*, ICN, C)                       |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - master key frequency T*,                                       |
|  - initial counter nonce ICN in V_{n-c},                          |
|  - ciphertext C = C_1 | ... | C_b, |C| <= m_max.                   |
|  Output:                                                          |
|  - plaintext P.                                                   |
|-------------------------------------------------------------------|
|  1. P = CTR-ACPKM-Master-Encrypt(N, K, T*, ICN, C)                |
|  1. Return P                                                      |
+-------------------------------------------------------------------+
```

The initial counter nonce ICN value for each message that is encrypted under the given initial key must be chosen in a unique manner.

6.3.3. GCM-ACPKM-Master Authenticated Encryption Mode

This section defines a GCM-ACPKM-Master authenticated encryption mode that uses the ACPKM-Master internal re-keying mechanism for the periodical key transformation.

The GCM-ACPKM-Master authenticated encryption mode can be considered as the base authenticated encryption mode GCM (see [GCM]) extended by the ACPKM-Master re-keying mechanism.

The GCM-ACPKM-Master authenticated encryption mode can be used with the following parameters:

- n in {128, 256};
- 128 <= k <= 512;
- the number c of bits in a specific part of the block to be incremented is such that 1 / 4 n <= c <= 1 / 2 n, c is a multiple of 8;
- authentication tag length t;
- the maximum message size m_max = min{N * (n * 2^{n/2-1} / k), n * (2^c - 2), 2^{n/2} - 1}.

The key material K[j] that is used for the j-th section processing is equal to K^j, |K^j| = k bits.

The GCM-ACPKM-Master mode encryption and decryption procedures are defined as follows:

```
+-------------------------------------------------------------------+
|  GHASH(X, H)                                                      |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - bit string X = X_1 | ... | X_m, X_i in V_n for i in {1, ... ,m}|
|  Output:                                                          |
|  - block GHASH(X, H) in V_n                                       |
|-------------------------------------------------------------------|
|  1. Y_0 = 0^n                                                     |
|  2. For i = 1, ... , m do                                         |
|         Y_i = (Y_{i-1} (xor) X_i) * H                             |
|  3. Return Y_m                                                    |
+-------------------------------------------------------------------+
```

```
|  GCTR(N, K, T*, ICB, X)                                                |
|------------------------------------------------------------------------|
|  Input:                                                                |
|  - section size N,                                                     |
|  - initial key K,                                                      |
|  - master key frequency T*,                                            |
|  - initial counter block ICB,                                          |
|  - X = X_1 | ... | X_b.                                                |
|  Output:                                                               |
|  - Y in V_{|X|}.                                                       |
|------------------------------------------------------------------------|
|  1. If X in V_0 then return Y, where Y in V_0                          |
|  2. GCTR_1 = ICB                                                       |
|  3. For i = 2, ... , b do                                              |
|        GCTR_i = Inc_c(GCTR_{i-1})                                      |
|  4. l = ceil(|X| / N)                                                  |
|  5. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)                        |
|  6. For j = 1, ... , b do                                              |
|        i = ceil(j * n / N),                                            |
|        G_j = E_{K^i}(GCTR_j)                                           |
|  7. Y = X (xor) MSB_{|X|}(G_1 | ... | G_b)                             |
|  8. Return Y                                                           |
+------------------------------------------------------------------------+

+------------------------------------------------------------------------+
|  GCM-ACPKM-Master-Encrypt(N, K, T*, ICN, P, A)                         |
|------------------------------------------------------------------------|
|  Input:                                                                |
|  - section size N,                                                     |
|  - initial key K,                                                      |
|  - master key frequency T*,                                            |
|  - initial counter nonce ICN in V_{n-c},                               |
|  - plaintext P = P_1 | ... | P_b, |P| <= m_max.                       |
|  - additional authenticated data A.                                    |
|  Output:                                                               |
|  - ciphertext C,                                                       |
|  - authentication tag T.                                               |
|------------------------------------------------------------------------|
|  1. K^1 = ACPKM-Master(T*, K, k, 1)                                    |
|  2. H = E_{K^1}(0^n)                                                   |
|  3. ICB_0 = ICN | 0^{c-1} | 1                                          |
|  4. C = GCTR(N, K, T*, Inc_c(ICB_0), P)                                |
|  5. u = n * ceil(|C| / n) - |C|                                        |
|     v = n * ceil(|A| / n) - |A|                                        |
|  6. S = GHASH(A | 0^v | C | 0^u | Vec_{n/2}(|A|) |                   |
|                | Vec_{n/2}(|C|), H)                                    |
|  7. T = MSB_t(E_{K^1}(ICB_0) (xor) S)                                  |
|  8. Return C | T                                                       |
```

```
+------------------------------------------------------------------+

+------------------------------------------------------------------+
|  GCM-ACPKM-Master-Decrypt(N, K, T*, ICN, A, C, T)                |
|------------------------------------------------------------------|
|  Input:                                                          |
|  - section size N,                                               |
|  - initial key K,                                                |
|  - master key frequency T*,                                      |
|  - initial counter nonce ICN in V_{n-c},                         |
|  - additional authenticated data A.                              |
|  - ciphertext C = C_1 | ... | C_b, |C| <= m_max,                |
|  - authentication tag T.                                         |
|  Output:                                                         |
|  - plaintext P or FAIL.                                          |
|------------------------------------------------------------------|
|  1. K^1 = ACPKM-Master(T*, K, k, 1)                              |
|  2. H = E_{K^1}(0^n)                                             |
|  3. ICB_0 = ICN | 0^{c-1} | 1                                    |
|  4. P = GCTR(N, K, T*, Inc_c(ICB_0), C)                          |
|  5. u = n * ceil(|C| / n) - |C|                                  |
|     v = n * ceil(|A| / n) - |A|                                  |
|  6. S = GHASH(A | 0^v | C | 0^u | Vec_{n/2}(|A|) |              |
|                | Vec_{n/2}(|C|), H)                              |
|  7. T' = MSB_t(E_{K^1}(ICB_0) (xor) S)                           |
|  8. IF T = T' then return P; else return FAIL.                   |
+------------------------------------------------------------------+
```

The * operation on (pairs of) the 2^n possible blocks corresponds to the multiplication operation for the binary Galois (finite) field of 2^n elements defined by the polynomial f as follows (by analogy with [GCM]):

n = 128: f = a^128 + a^7 + a^2 + a^1 + 1,

n = 256: f = a^256 + a^10 + a^5 + a^2 + 1.

The initial counter nonce ICN value for each message that is encrypted under the given initial key must be chosen in a unique manner.

6.3.4. CBC-ACPKM-Master Encryption Mode

This section defines a CBC-ACPKM-Master encryption mode that uses the ACPKM-Master internal re-keying mechanism for the periodical key transformation.

The CBC-ACPKM-Master encryption mode can be considered as the base encryption mode CBC (see [MODES]) extended by the ACPKM-Master re-keying mechanism.

The CBC-ACPKM-Master encryption mode can be used with the following parameters:

- 64 <= n <= 512;
- 128 <= k <= 512;
- the maximum message size m_max = N * (n * 2^{n/2-1} / k).

In the specification of the CBC-ACPKM-Master mode the plaintext and ciphertext must be a sequence of one or more complete data blocks. If the data string to be encrypted does not initially satisfy this property, then it MUST be padded to form complete data blocks. The padding methods are out of the scope of this document. An example of a padding method can be found in Appendix A of [MODES].

The key material K[j] that is used for the j-th section processing is equal to K^j, |K^j| = k bits.

We will denote by D_{K} the decryption function which is a permutation inverse to E_{K}.

The CBC-ACPKM-Master mode encryption and decryption procedures are defined as follows:

```
+-------------------------------------------------------------------+
|  CBC-ACPKM-Master-Encrypt(N, K, T*, IV, P)                        |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - master key frequency T*,                                       |
|  - initialization vector IV in V_n,                               |
|  - plaintext P = P_1 | ... | P_b, |P_b| = n, |P| <= m_max.       |
|  Output:                                                          |
|  - ciphertext C.                                                  |
|-------------------------------------------------------------------|
|  1. l = ceil(|P| / N)                                             |
|  2. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)                   |
|  3. C_0 = IV                                                      |
|  4. For j = 1, 2, ... , b do                                      |
|         i = ceil(j * n / N),                                      |
|         C_j = E_{K^i}(P_j (xor) C_{j-1})                          |
|  5. Return C = C_1 | ... | C_b                                    |
+-------------------------------------------------------------------+

+-------------------------------------------------------------------+
|  CBC-ACPKM-Master-Decrypt(N, K, T*, IV, C)                        |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - master key frequency T*,                                       |
|  - initialization vector IV in V_n,                               |
|  - ciphertext C = C_1 | ... | C_b, |C_b| = n, |C| <= m_max.       |
|  Output:                                                          |
|  - plaintext P.                                                   |
|-------------------------------------------------------------------|
|  1. l = ceil(|C| / N)                                             |
|  2. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)                   |
|  3. C_0 = IV                                                      |
|  4. For j = 1, 2, ... , b do                                      |
|         i = ceil(j * n / N)                                       |
|         P_j = D_{K^i}(C_j) (xor) C_{j-1}                          |
|  5. Return P = P_1 | ... | P_b                                    |
+-------------------------------------------------------------------+
```

The initialization vector IV for any particular execution of the encryption process must be unpredictable.

6.3.5. CFB-ACPKM-Master Encryption Mode

This section defines a CFB-ACPKM-Master encryption mode that uses the ACPKM-Master internal re-keying mechanism for the periodical key transformation.

The CFB-ACPKM-Master encryption mode can be considered as the base encryption mode CFB (see [MODES]) extended by the ACPKM-Master re-keying mechanism.

The CFB-ACPKM-Master encryption mode can be used with the following parameters:

- 64 <= n <= 512;
- 128 <= k <= 512;
- the maximum message size m_max = N * (n * 2^{n/2-1} / k).

The key material K[j] that is used for the j-th section processing is equal to K^j, |K^j| = k bits.

The CFB-ACPKM-Master mode encryption and decryption procedures are defined as follows:

```
+-------------------------------------------------------------+
|  CFB-ACPKM-Master-Encrypt(N, K, T*, IV, P)                  |
|-------------------------------------------------------------|
|  Input:                                                     |
|  - section size N,                                          |
|  - initial key K,                                           |
|  - master key frequency T*,                                 |
|  - initialization vector IV in V_n,                         |
|  - plaintext P = P_1 | ... | P_b, |P| <= m_max.            |
|  Output:                                                    |
|  - ciphertext C.                                            |
|-------------------------------------------------------------|
|  1. l = ceil(|P| / N)                                       |
|  2. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)           |
|  3. C_0 = IV                                                |
|  4. For j = 1, 2, ... , b - 1 do                            |
|         i = ceil(j * n / N),                                |
|         C_j = E_{K^i}(C_{j-1}) (xor) P_j                    |
|  5. C_b = MSB_{|P_b|}(E_{K^l}(C_{b-1})) (xor) P_b           |
|  6. Return C = C_1 | ... | C_b                              |
+-------------------------------------------------------------+

+-------------------------------------------------------------+
|  CFB-ACPKM-Master-Decrypt(N, K, T*, IV, C)                  |
|-------------------------------------------------------------|
|  Input:                                                     |
|  - section size N,                                          |
|  - initial key K,                                           |
|  - master key frequency T*,                                 |
|  - initialization vector IV in V_n,                         |
|  - ciphertext C = C_1 | ... | C_b, |C| <= m_max.           |
|  Output:                                                    |
|  - plaintext P.                                             |
|-------------------------------------------------------------|
|  1. l = ceil(|C| / N)                                       |
|  2. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)           |
|  3. C_0 = IV                                                |
|  4. For j = 1, 2, ... , b - 1 do                            |
|         i = ceil(j * n / N),                                |
|         P_j = E_{K^i}(C_{j-1}) (xor) C_j                    |
|  5. P_b = MSB_{|C_b|}(E_{K^l}(C_{b-1})) (xor) C_b           |
|  6. Return P = P_1 | ... | P_b                              |
+-------------------------------------------------------------+
```

The initialization vector IV for any particular execution of the encryption process must be unpredictable.

6.3.6. OMAC-ACPKM-Master Authentication Mode

This section defines an OMAC-ACPKM-Master message authentication code calculation mode that uses the ACPKM-Master internal re-keying mechanism for the periodical key transformation.

The OMAC-ACPKM-Master mode can be considered as the base message authentication code calculation mode OMAC, which is also known as CMAC (see [RFC4493]), extended by the ACPKM-Master re-keying mechanism.

The OMAC-ACPKM-Master message authentication code calculation mode can be used with the following parameters:

- n in {64, 128, 256};
- 128 <= k <= 512;
- the maximum message size m_max = N * (n * 2^{n/2-1} / (k + n)).

The key material K[j] that is used for one section processing is equal to K^j | K^j_1, where |K^j| = k and |K^j_1| = n.

The following is a specification of the subkey generation process of OMAC:

```
+-------------------------------------------------------------------+
| Generate_Subkey(K1, r)                                            |
|-------------------------------------------------------------------|
| Input:                                                            |
|  - key K1.                                                        |
|  Output:                                                          |
|  - key SK.                                                        |
|-------------------------------------------------------------------|
|   1. If r = n then return K1                                      |
|   2. If r < n then                                                |
|          if MSB_1(K1) = 0                                         |
|              return K1 << 1                                       |
|          else                                                     |
|              return (K1 << 1) (xor) R_n                           |
|                                                                   |
+-------------------------------------------------------------------+
```

Here R_n takes the following values:

- n = 64: R_{64} = 0^{59} | 11011;

- n = 128: R_{128} = 0^{120} | 10000111;

- n = 256: R_{256} = 0^{145} | 10000100101.

The OMAC-ACPKM-Master message authentication code calculation mode is defined as follows:

```
+-------------------------------------------------------------------+
| OMAC-ACPKM-Master(K, N, T*, M)                                    |
|-------------------------------------------------------------------|
| Input:                                                            |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - master key frequency T*,                                       |
|  - plaintext M = M_1 | ... | M_b, |M| <= m_max.                   |
|  Output:                                                          |
|  - message authentication code T.                                 |
|-------------------------------------------------------------------|
| 1. C_0 = 0^n                                                      |
| 2. l = ceil(|M| / N)                                              |
| 3. K^1 | K^1_1 | ... | K^l | K^l_1 =                            |
|                  = ACPKM-Master(T*, K, (k + n), l)                |
| 4. For j = 1, 2, ... , b - 1 do                                   |
|        i = ceil(j * n / N),                                       |
|        C_j = E_{K^i}(M_j (xor) C_{j-1})                           |
| 5. SK = Generate_Subkey(K^l_1, |M_b|)                            |
| 6. If |M_b| = n then M*_b = M_b                                   |
|                  else M*_b = M_b | 1 | 0^{n - 1 -|M_b|}          |
| 7. T = E_{K^l}(M*_b (xor) C_{b-1} (xor) SK)                       |
| 8. Return T                                                       |
+-------------------------------------------------------------------+
```

7. Joint Usage of External and Internal Re-keying

Both external re-keying and internal re-keying have their own advantages and disadvantages discussed in Section 1. For instance, using external re-keying can essentially limit the message length, while in the case of internal re-keying the section size, which can be chosen as the maximal possible for operational properties, limits the amount of separate messages. Therefore, the choice of re-keying mechanism (either external or internal) depends on particular protocol features. However, some protocols may have features that require to take advantages provided by both external and internal re-keying mechanisms: for example, the protocol mainly transmits messages of small length, but it must additionally support very long messages processing. In such situations it is necessary to use

external and internal re-keying jointly, since these techniques negate each other's disadvantages.

For composition of external and internal re-keying techniques any mechanism described in Section 5 can be used with any mechanism described in Section 6.

For example, consider the GCM-ACPKM mode with external serial re-keying based on a KDF on a Hash function. Denote by a frame size the number of messages in each frame (in the case of implicit approach to the key lifetime control) for external re-keying.

Let L be a key lifetime limitation. The section size N for internal re-keying and the frame size q for external re-keying must be chosen in such a way that q * N must not exceed L.

Suppose that t messages (ICN_i, P_i, A_i), with initial counter nonce ICN_i, plaintext P_i and additional authenticated data A_i, will be processed before renegotiation.

For authenticated encryption of each message (ICN_i, P_i, A_i), i = 1, ..., t, the following algorithm can be applied:

1. j = ceil(i / q),
2. K^j = ExtSerialH(K, j),
3. C_i | T_i = GCM-ACPKM-Encrypt(N, K^j, ICN_i, P_i, A_i).

Note that nonces ICN_i, that are used under the same frame key, must be unique for each message.

8. Security Considerations

Re-keying should be used to increase "a priori" security properties of ciphers in hostile environments (e.g., with side-channel adversaries). If some efficient attacks are known for a cipher, it must not be used. So re-keying cannot be used as a patch for vulnerable ciphers. Base cipher properties must be well analyzed, because the security of re-keying mechanisms is based on the security of a block cipher as a pseudorandom function.

Re-keying is not intended to solve any post-quantum security issues for symmetric cryptography, since the reduction of security caused by Grover's algorithm is not connected with a size of plaintext transformed by a cipher - only a negligible (sufficient for key uniqueness) material is needed; and the aim of re-keying is to limit a size of plaintext transformed under one initial key.

Re-keying can provide backward security only if previous key material is securely deleted after usage by all parties.

9. IANA Considerations

This document does not require any IANA actions.

10. References

10.1. Normative References

[CMS] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <http://www.rfc-editor.org/info/rfc5652>.

[DTLS] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <http://www.rfc-editor.org/info/rfc6347>.

[ESP] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <http://www.rfc-editor.org/info/rfc4303>.

[GCM] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf, November 2007.

[MODES] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", NIST Special Publication 800-38A, December 2001.

[NISTSP800-108] National Institute of Standards and Technology, "Recommendation for Key Derivation Using Pseudorandom Functions", NIST Special Publication 800-108, November 2008, <http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-108.pdf>.

[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>.

[RFC4493] Song, JH., Poovendran, R., Lee, J., and T. Iwata, "The AES-CMAC Algorithm", RFC 4493, DOI 10.17487/RFC4493, June 2006, <https://www.rfc-editor.org/info/rfc4493>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC7836] Smyshlyaev, S., Ed., Alekseev, E., Oshkin, I., Popov, V., Leontiev, S., Podobaev, V., and D. Belyavsky, "Guidelines on the Cryptographic Algorithms to Accompany the Usage of Standards GOST R 34.10-2012 and GOST R 34.11-2012", RFC 7836, DOI 10.17487/RFC7836, March 2016, <https://www.rfc-editor.org/info/rfc7836>.

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

[TLS] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <http://www.rfc-editor.org/info/rfc8446>.

10.2. Informative References

[AAOS2017] Ahmetzyanova, L., Alekseev, E., Oshkin, I., and S. Smyshlyaev, "Increasing the Lifetime of Symmetric Keys for the GCM Mode by Internal Re-keying", Cryptology ePrint Archive Report 2017/697, 2017, <https://eprint.iacr.org/2017/697.pdf>.

[AbBell] Michel Abdalla and Mihir Bellare, "Increasing the Lifetime of a Key: A Comparative Analysis of the Security of Re-keying Techniques", ASIACRYPT2000, LNCS 1976, pp. 546-559, 2000.

[AESDUKPT] ANSI, "Retail Financial Services Symmetric Key Management - Part 3: Derived Unique Key Per Transaction", ANSI X9.24-3-2017, 2017.

[FKK2005] Fu, K., Kamara, S., and T. Kohno, "Key Regression: Enabling Efficient Key Distribution for Secure Distributed Storage", November 2005, <https://homes.cs.washington.edu/~yoshi/papers/KR/NDSS06.pdf>.

[FPS2012] Faust, S., Pietrzak, K., and j. Schipper, "Practical Leakage-Resilient Symmetric Cryptography", CHES2012 LNCS, vol. 7428, pp. 213-232,, 2012, <https://link.springer.com/content/pdf/10.1007%2F978-3-642-33027-8_13.pdf>.

[FRESHREKEYING] Dziembowski, S., Faust, S., Herold, G., Journault, A., Masny, D., and F. Standaert, "Towards Sound Fresh Re-Keying with Hard (Physical) Learning Problems", Cryptology ePrint Archive Report 2016/573, June 2016, <https://eprint.iacr.org/2016/573>.

[GGM] Goldreich, O., Goldwasser, S., and S. Micali, "How to Construct Random Functions", Journal of the Association for Computing Machinery Vol.33, No.4, pp. 792-807, October 1986, <http://www.wisdom.weizmann.ac.il/~/oded/X/ggm.pdf>.

[KMNT2003] Kim, Y., Maino, F., Narasimha, M., and G. Tsudik, "Secure Group Services for Storage Area Networks", IEEE Communication Magazine 41, pp. 92-99, 2003, <http://www.ics.uci.edu/~gts/paps/kmnt02.pdf>.

[LDC] Howard M. Heys, "A Tutorial on Linear and Differential Cryptanalysis", 2017, <http://www.cs.bc.edu/~straubin/crypto2017/heys.pdf>.

[OWT] Joye, M. and S. Yen, "One-Way Cross-Trees and Their Applications", DOI 10.1007/3-540-45664-3_25, February 2002, <https://link.springer.com/content/pdf/10.1007%2F3-540-45664-3_25.pdf>.

[P3] Peter Alexander, "Dynamic Key Changes on Encrypted Sessions", CFRG mail archive , December 2017, <https://www.ietf.org/mail-archive/web/cfrg/current/msg09401.html>.

[Pietrzak2009] Pietrzak, K., "A Leakage-Resilient Mode of Operation", EUROCRYPT2009 LNCS, vol. 5479, pp. 462-482,, 2009, <https://iacr.org/archive/eurocrypt2009/54790461/54790461.pdf>.

[SIGNAL] Perrin, T., Ed. and M. Marlinspike, "The Double Ratchet Algorithm", November 2016, <https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf>.

[Sweet32] Karthikeyan Bhargavan, Gaetan Leurent, "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN", Cryptology ePrint Archive Report 2016/798, 2016, <https://sweet32.info/SWEET32_CCS16.pdf>.

[TAHA] Taha, M. and P. Schaumont, "Key Updating for Leakage Resiliency With Application to AES Modes of Operation", DOI 10.1109/TIFS.2014.2383359, December 2014, <http://ieeexplore.ieee.org/document/6987331/>.

[TEMPEST] By Craig Ramsay, Jasper Lohuis, "TEMPEST attacks against AES. Covertly stealing keys for 200 euro", 2017, <https://www.fox-it.com/en/wp-content/uploads/sites/11/Tempest_attacks_against_AES.pdf>.

[U2F] Chang, D., Mishra, S., Sanadhya, S., and A. Singhl, "On Making U2F Protocol Leakage-Resilient via Re-keying.", Cryptology ePrint Archive Report 2017/721, August 2017, <https://eprint.iacr.org/2017/721.pdf>.

Appendix A. Test Examples

A.1. Test Examples for External Re-keying

A.1.1. External Re-keying with a Parallel Construction

External re-keying with a parallel construction based on AES-256
****
k = 256
t = 128

Initial key:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

K^1:
51 16 8A B6 C8 A8 38 65 54 85 31 A5 D2 BA C3 86
64 7D 5C D5 1C 3D 62 98 BC 09 B1 D8 64 EC D9 B1

K^2:
6F ED F5 D3 77 57 48 75 35 2B 5F 4D B6 5B E0 15
B8 02 92 32 D8 D3 8D 73 FE DC DD C6 C8 36 78 BD

K^3:
B6 40 24 85 A4 24 BD 35 B4 26 43 13 76 26 70 B6
5B F3 30 3D 3B 20 EB 14 D1 3B B7 91 74 E3 DB EC

...

K^126:
2F 3F 15 1B 53 88 23 CD 7D 03 FC 3D FD B3 57 5E
23 E4 1C 4E 46 FF 6B 33 34 12 27 84 EF 5D 82 23

K^127:
8E 51 31 FB 0B 64 BB D0 BC D4 C5 7B 1C 66 EF FD
97 43 75 10 6C AF 5D 5E 41 E0 17 F4 05 63 05 ED

K^128:
77 4F BF B3 22 60 C5 3B A3 8E FE B1 96 46 76 41
94 49 AF 84 2D 84 65 A7 F4 F7 2C DC A4 9D 84 F9

```
External re-keying with a parallel construction based on SHA-256
****************************************************************
k = 256
t = 128

label:
SHA2label

Initial key:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

K^1:
C1 A1 4C A0 30 29 BE 43 9F 35 3C 79 1A 51 48 57
26 7A CD 5A E8 7D E7 D1 B2 E2 C7 AF A4 29 BD 35

K^2:
03 68 BB 74 41 2A 98 ED C4 7B 94 CC DF 9C F4 9E
A9 B8 A9 5F 0E DC 3C 1E 3B D2 59 4D D1 75 82 D4

K^3:
2F D3 68 D3 A7 8F 91 E6 3B 68 DC 2B 41 1D AC 80
0A C3 14 1D 80 26 3E 61 C9 0D 24 45 2A BD B1 AE

...

K^126:
55 AC 2B 25 00 78 3E D4 34 2B 65 0E 75 E5 8B 76
C8 04 E9 D3 B6 08 7D C0 70 2A 99 A4 B5 85 F1 A1

K^127:
77 4D 15 88 B0 40 90 E5 8C 6A D7 5D 0F CF 0A 4A
6C 23 F1 B3 91 B1 EF DF E5 77 64 CD 09 F5 BC AF

K^128:
E5 81 FF FB 0C 90 88 CD E5 F4 A5 57 B6 AB D2 2E
94 C3 42 06 41 AB C1 72 66 CC 2F 59 74 9C 86 B3
```

A.1.2. External Re-keying with a Serial Construction

```
External re-keying with a serial construction based on AES-256
**************************************************************
AES 256 examples:
k = 256
t = 128
```

```
Initial key:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

K*_1:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

K^1:
66 B8 BD E5 90 6C EC DF FA 8A B2 FD 92 84 EB F0
51 16 8A B6 C8 A8 38 65 54 85 31 A5 D2 BA C3 86

K*_2:
64 7D 5C D5 1C 3D 62 98 BC 09 B1 D8 64 EC D9 B1
6F ED F5 D3 77 57 48 75 35 2B 5F 4D B6 5B E0 15

K^2:
66 B8 BD E5 90 6C EC DF FA 8A B2 FD 92 84 EB F0
51 16 8A B6 C8 A8 38 65 54 85 31 A5 D2 BA C3 86

K*_3:
64 7D 5C D5 1C 3D 62 98 BC 09 B1 D8 64 EC D9 B1
6F ED F5 D3 77 57 48 75 35 2B 5F 4D B6 5B E0 15

K^3:
66 B8 BD E5 90 6C EC DF FA 8A B2 FD 92 84 EB F0
51 16 8A B6 C8 A8 38 65 54 85 31 A5 D2 BA C3 86

...

K*_126:
64 7D 5C D5 1C 3D 62 98 BC 09 B1 D8 64 EC D9 B1
6F ED F5 D3 77 57 48 75 35 2B 5F 4D B6 5B E0 15

K^126:
66 B8 BD E5 90 6C EC DF FA 8A B2 FD 92 84 EB F0
51 16 8A B6 C8 A8 38 65 54 85 31 A5 D2 BA C3 86

K*_127:
64 7D 5C D5 1C 3D 62 98 BC 09 B1 D8 64 EC D9 B1
6F ED F5 D3 77 57 48 75 35 2B 5F 4D B6 5B E0 15

K^127:
66 B8 BD E5 90 6C EC DF FA 8A B2 FD 92 84 EB F0
51 16 8A B6 C8 A8 38 65 54 85 31 A5 D2 BA C3 86

K*_128:
64 7D 5C D5 1C 3D 62 98 BC 09 B1 D8 64 EC D9 B1
```

```
6F ED F5 D3 77 57 48 75 35 2B 5F 4D B6 5B E0 15

K^128:
66 B8 BD E5 90 6C EC DF FA 8A B2 FD 92 84 EB F0
51 16 8A B6 C8 A8 38 65 54 85 31 A5 D2 BA C3 86
```

External re-keying with a serial construction based on SHA-256

```
**************************************************************
k = 256
t = 128

Initial key:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

label1:
SHA2label1

label2:
SHA2label2

K*_1:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

K^1:
2D A8 D1 37 6C FD 52 7F F7 36 A4 E2 81 C6 0A 9B
F3 8E 66 97 ED 70 4F B5 FB 10 33 CC EC EE D5 EC

K*_2:
14 65 5A D1 7C 19 86 24 9B D3 56 DF CC BE 73 6F
52 62 4A 9D E3 CC 40 6D A9 48 DA 5C D0 68 8A 04

K^2:
2F EA 8D 57 2B EF B8 89 42 54 1B 8C 1B 3F 8D B1
84 F9 56 C7 FE 01 11 99 1D FB 98 15 FE 65 85 CF

K*_3:
18 F0 B5 2A D2 45 E1 93 69 53 40 55 43 70 95 8D
70 F0 20 8C DF B0 5D 67 CD 1B BF 96 37 D3 E3 EB

K^3:
53 C7 4E 79 AE BC D1 C8 24 04 BF F6 D7 B1 AC BF
F9 C0 0E FB A8 B9 48 29 87 37 E1 BA E7 8F F7 92
```

```
...

K*_126:
A3 6D BF 02 AA 0B 42 4A F2 C0 46 52 68 8B C7 E6
5E F1 62 C3 B3 2F DD EF E4 92 79 5D BB 45 0B CA

K^126:
6C 4B D6 22 DC 40 48 0F 29 C3 90 B8 E5 D7 A7 34
23 4D 34 65 2C CE 4A 76 2C FE 2A 42 C8 5B FE 9A

K*_127:
84 5F 49 3D B8 13 1D 39 36 2B BE D3 74 8F 80 A1
05 A7 07 37 BA 15 72 E0 73 49 C2 67 5D 0A 28 A1

K^127:
57 F0 BD 5A B8 2A F3 6B 87 33 CF F7 22 62 B4 D0
F0 EE EF E1 50 74 E5 BA 13 C1 23 68 87 36 29 A2

K*_128:
52 F2 0F 56 5C 9C 56 84 AF 69 AD 45 EE B8 DA 4E
7A A6 04 86 35 16 BA 98 E4 CB 46 D2 E8 9A C1 09

K^128:
9B DD 24 7D F3 25 4A 75 E0 22 68 25 68 DA 9D D5
C1 6D 2D 2B 4F 3F 1F 2B 5E 99 82 7F 15 A1 4F A4
```

A.2. Test Examples for Internal Re-keying

A.2.1. Internal Re-keying Mechanisms that Do Not Require Master Key

```
CTR-ACPKM mode with AES-256
***************************
k = 256
n = 128
c = 64
N = 256

Initial key K:
00000:   88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77
00010:   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF

Plain text P:
00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88
00010:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A
00020:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00
```

```
00030:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11
00040:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22
00050:   44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33
00060:   55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33 44

ICN:
12 34 56 78 90 AB CE F0 A1 B2 C3 D4 E5 F0 01 12
23 34 45 56 67 78 89 90 12 13 14 15 16 17 18 19

D_1:
00000:   80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F

D_2:
00000:   90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F

Section_1

Section key K^1:
00000:   88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77
00010:   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF

Input block CTR_1:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 00

Output block G_1:
00000:   FD 7E F8 9A D9 7E A4 B8 8D B8 B5 1C 1C 9D 6D D0

Input block CTR_2:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 01

Output block G_2:
00000:   19 98 C5 71 76 37 FB 17 11 E4 48 F0 0C 0D 60 B2

Section_2

Section key K^2:
00000:   F6 80 D1 21 2F A4 3D F4 EC 3A 91 DE 2A B1 6F 1B
00010:   36 B0 48 8A 4F C1 2E 09 98 D2 E4 A8 88 E8 4F 3D

Input block CTR_3:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 02

Output block G_3:
00000:   E4 88 89 4F B6 02 87 DB 77 5A 07 D9 2C 89 46 EA

Input block CTR_4:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 03
```

```
   Output block G_4:
   00000:   BC 4F 87 23 DB F0 91 50 DD B4 06 C3 1D A9 7C A4

   Section_3

   Section key K^3:
   00000:   8E B9 7E 43 27 1A 42 F1 CA 8E E2 5F 5C C7 C8 3B
   00010:   1A CE 9E 5E D0 6A A5 3B 57 B9 6A CF 36 5D 24 B8

   Input block CTR_5:
   00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 04

   Output block G_5:
   00000:   68 6F 22 7D 8F B2 9C BD 05 C8 C3 7D 22 FE 3B B7

   Input block CTR_6:
   00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 05

   Output block G_6:
   00000:   C0 1B F9 7F 75 6E 12 2F 80 59 55 BD DE 2D 45 87

   Section_4

   Section key K^4:
   00000:   C5 71 6C C9 67 98 BC 2D 4A 17 87 B7 8A DF 94 AC
   00010:   E8 16 F8 0B DB BC AD 7D 60 78 12 9C 0C B4 02 F5

   Block number 7:

   Input block CTR_7:
   00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 06

   Output block G_7:
   00000:   03 DE 34 74 AB 9B 65 8A 3B 54 1E F8 BD 2B F4 7D


   The result G = G_1 | G_2 | G_3 | G_4 | G_5 | G_6 | G_7:
   00000:   FD 7E F8 9A D9 7E A4 B8 8D B8 B5 1C 1C 9D 6D D0
   00010:   19 98 C5 71 76 37 FB 17 11 E4 48 F0 0C 0D 60 B2
   00020:   E4 88 89 4F B6 02 87 DB 77 5A 07 D9 2C 89 46 EA
   00030:   BC 4F 87 23 DB F0 91 50 DD B4 06 C3 1D A9 7C A4
   00040:   68 6F 22 7D 8F B2 9C BD 05 C8 C3 7D 22 FE 3B B7
   00050:   C0 1B F9 7F 75 6E 12 2F 80 59 55 BD DE 2D 45 87
   00060:   03 DE 34 74 AB 9B 65 8A 3B 54 1E F8 BD 2B F4 7D

   The result ciphertext C = P (xor) MSB_{|P|}(G):
   00000:   EC 5C CB DE 8C 18 D3 B8 72 56 68 D0 A7 37 F4 58
   00010:   19 89 E7 42 32 62 9D 60 99 7D E2 4B C0 E3 9F B8
```

```
00020:   F5 AA BA 0B E3 64 F0 53 EE F0 BC 15 C2 76 4C EA
00030:   9E 7C C3 76 BD 87 19 C9 77 0F CA 2D E2 A3 7C B5
00040:   5B 2B 77 1B F8 3A 05 17 BE 04 2D 82 28 FE 2A 95
00050:   84 4E 9F 08 FD F7 B8 94 4C B7 AA B7 DE 3C 67 B4
00060:   56 B8 43 FC 32 31 DE 46 D5 AB 14 F8 AC 09 C7 39
```

```
GCM-ACPKM mode with AES-128
***************************
k = 128
n = 128
c = 32
N = 256

Initilal Key K:
00000:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Additional data A:
00000:   11 22 33

Plaintext:
00000:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00020:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

ICN:
00000:   00 00 00 00 00 00 00 00 00 00 00 00

Number of sections: 2

Section key K^1:
00000:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Section key K^2:
00000:   15 1A 9F B0 B6 AC C5 97 6A FB 50 31 D1 DE C8 41

Encrypted GCTR_1 | GCTR_2 | GCTR_3:
00000:   03 88 DA CE 60 B6 A3 92 F3 28 C2 B9 71 B2 FE 78
00010:   F7 95 AA AB 49 4B 59 23 F7 FD 89 FF 94 8B C1 E0
00020:   D6 B3 12 46 E9 CE 9F F1 3A B3 42 7E E8 91 96 AD

Ciphertext C:
00000:   03 88 DA CE 60 B6 A3 92 F3 28 C2 B9 71 B2 FE 78
00010:   F7 95 AA AB 49 4B 59 23 F7 FD 89 FF 94 8B C1 E0
00020:   D6 B3 12 46 E9 CE 9F F1 3A B3 42 7E E8 91 96 AD

GHASH input:
```

```
00000:   11 22 33 00 00 00 00 00 00 00 00 00 00 00 00 00
00010:   03 88 DA CE 60 B6 A3 92 F3 28 C2 B9 71 B2 FE 78
00020:   F7 95 AA AB 49 4B 59 23 F7 FD 89 FF 94 8B C1 E0
00030:   D6 B3 12 46 E9 CE 9F F1 3A B3 42 7E E8 91 96 AD
00040:   00 00 00 00 00 00 00 18 00 00 00 00 00 00 01 80

GHASH output S:
00000:   E8 ED E9 94 9A DD 55 30 B0 F4 4E F5 00 FC 3E 3C

Authentication tag  T:
00000:   B0 0F 15 5A 60 A3 65 51 86 8B 53 A2 A4 1B 7B 66

The result C | T:
00000:   03 88 DA CE 60 B6 A3 92 F3 28 C2 B9 71 B2 FE 78
00010:   F7 95 AA AB 49 4B 59 23 F7 FD 89 FF 94 8B C1 E0
00020:   D6 B3 12 46 E9 CE 9F F1 3A B3 42 7E E8 91 96 AD
00030:   B0 0F 15 5A 60 A3 65 51 86 8B 53 A2 A4 1B 7B 66
```

A.2.2. Internal Re-keying Mechanisms with a Master Key

```
CTR-ACPKM-Master mode with AES-256
**********************************
k = 256
n = 128
c for CTR-ACPKM mode = 64
c for CTR-ACPKM-Master mode = 64
N = 256
T* = 512

Initial key K:
00000:   88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77
00010:   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF

Initial vector ICN:
00000:   12 34 56 78 90 AB CE F0 A1 B2 C3 D4 E5 F0 01 12

Plaintext P:
00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88
00010:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A
00020:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00
00030:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11
00040:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22
00050:   44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33
00060:   55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33 44
```

```
K^1 | K^2 | K^3 | K^4:
00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60
00020:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
00030:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3
00040:   E8 76 2B 30 8B 08 EB CE 3E 93 9A C2 C0 3E 76 D4
00050:   60 9A AB D9 15 33 13 D3 CF D3 94 E7 75 DF 3A 94
00060:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
00070:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12

Section_1

K^1:
00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60

Input block CTR_1:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 00

Output block G_1:
00000:   8C A2 B6 82 A7 50 65 3F 8E BF 08 E7 9F 99 4D 5C

Input block CTR_2:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 01

Output block G_2:
00000:   F6 A6 A5 BA 58 14 1E ED 23 DC 31 68 D2 35 89 A1


Section_2

K^2:
00000:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
00010:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3

Input block CTR_3:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 02

Output block G_3:
00000:   4A 07 5F 86 05 87 72 94 1D 8E 7D F8 32 F4 23 71

Input block CTR_4:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 03

Output block G_4:
00000:   23 35 66 AF 61 DD FE A7 B1 68 3F BA B0 52 4A D7
```

```
Section_3

K^3:
00000:   E8 76 2B 30 8B 08 EB CE 3E 93 9A C2 C0 3E 76 D4
00010:   60 9A AB D9 15 33 13 D3 CF D3 94 E7 75 DF 3A 94

Input block CTR_5:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 04

Output block G_5:
00000:   A8 09 6D BC E8 BB 52 FC DE 6E 03 70 C1 66 95 E8

Input block CTR_6:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 05

Output block G_6:
00000:   C6 E3 6E 8E 5B 82 AA C4 A6 6C 14 8D B1 F6 9B EF


Section_4

K^4:
00000:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
00010:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12

Input block CTR_7:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 06

Output block G_7:
00000:   82 2B E9 07 96 37 44 95 75 36 3F A7 07 F8 40 22


The result G = G_1 | G_2 | G_3 | G_4 | G_5 | G_6 | G_7:
00000:   8C A2 B6 82 A7 50 65 3F 8E BF 08 E7 9F 99 4D 5C
00010:   F6 A6 A5 BA 58 14 1E ED 23 DC 31 68 D2 35 89 A1
00020:   4A 07 5F 86 05 87 72 94 1D 8E 7D F8 32 F4 23 71
00030:   23 35 66 AF 61 DD FE A7 B1 68 3F BA B0 52 4A D7
00040:   A8 09 6D BC E8 BB 52 FC DE 6E 03 70 C1 66 95 E8
00050:   C6 E3 6E 8E 5B 82 AA C4 A6 6C 14 8D B1 F6 9B EF
00060:   82 2B E9 07 96 37 44 95 75 36 3F A7 07 F8 40 22

The result ciphertext C = P (xor) MSB_{|P|}(G):
00000:   9D 80 85 C6 F2 36 12 3F 71 51 D5 2B 24 33 D4 D4
00010:   F6 B7 87 89 1C 41 78 9A AB 45 9B D3 1E DB 76 AB
00020:   5B 25 6C C2 50 E1 05 1C 84 24 C6 34 DC 0B 29 71
00030:   01 06 22 FA 07 AA 76 3E 1B D3 F3 54 4F 58 4A C6
00040:   9B 4D 38 DA 9F 33 CB 56 65 A2 ED 8F CB 66 84 CA
00050:   82 B6 08 F9 D3 1B 00 7F 6A 82 EB 87 B1 E7 B9 DC
```

```
00060:   D7 4D 9E 8F 0F 9D FF 59 9B C9 35 A7 16 DA 73 66
```

```
GCM-ACPKM-Master mode with AES-256
**********************************
k = 192
n = 128
c for the CTR-ACPKM mode = 64
c for the GCM-ACPKM-Master mode = 32
T* = 384
N = 256

Initila Key K:
00000:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010:   00 00 00 00 00 00 00 00

Additional data A:
00000:   11 22 33

Plaintext:
00000:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00020:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00030:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00040:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

ICN:
00000:   00 00 00 00 00 00 00 00 00 00 00 00

Number of sections: 3

K^1 | K^2 | K^3:
00000:   93 BA AF FB 35 FB E7 39 C1 7C 6A C2 2E EC F1 8F
00010:   7B 89 F0 BF 8B 18 07 05 96 48 68 9F 36 A7 65 CC
00020:   CD 5D AC E2 0D 47 D9 18 D7 86 D0 41 A8 3B AB 99
00030:   F5 F8 B1 06 D2 71 78 B1 B0 08 C9 99 0B 72 E2 87
00040:   5A 2D 3C BE F1 6E 67 3C

Encrypted GCTR_1 | ... | GCTR_5
00000:   43 FA 71 81 64 B1 E3 D7 1E 7B 65 39 A7 02 1D 52
00010:   69 9B 9E 1B 43 24 B7 52 95 74 E7 90 F2 BE 60 E8
00020:   11 62 C9 90 2A 2B 77 7F D9 6A D6 1A 99 E0 C6 DE
00030:   4B 91 D4 29 E3 1A 8C 11 AF F0 BC 47 F6 80 AF 14
00040:   40 1C C1 18 14 63 8E 76 24 83 37 75 16 34 70 08

Ciphertext C:
```

```
00000:   43 FA 71 81 64 B1 E3 D7 1E 7B 65 39 A7 02 1D 52
00010:   69 9B 9E 1B 43 24 B7 52 95 74 E7 90 F2 BE 60 E8
00020:   11 62 C9 90 2A 2B 77 7F D9 6A D6 1A 99 E0 C6 DE
00030:   4B 91 D4 29 E3 1A 8C 11 AF F0 BC 47 F6 80 AF 14
00040:   40 1C C1 18 14 63 8E 76 24 83 37 75 16 34 70 08

GHASH input:
00000:   11 22 33 00 00 00 00 00 00 00 00 00 00 00 00 00
00010:   43 FA 71 81 64 B1 E3 D7 1E 7B 65 39 A7 02 1D 52
00020:   69 9B 9E 1B 43 24 B7 52 95 74 E7 90 F2 BE 60 E8
00030:   11 62 C9 90 2A 2B 77 7F D9 6A D6 1A 99 E0 C6 DE
00040:   4B 91 D4 29 E3 1A 8C 11 AF F0 BC 47 F6 80 AF 14
00050:   40 1C C1 18 14 63 8E 76 24 83 37 75 16 34 70 08
00060:   00 00 00 00 00 00 00 18 00 00 00 00 00 00 02 80

GHASH output S:
00000:   6E A3 4B D5 6A C5 40 B7 3E 55 D5 86 D1 CC 09 7D

Authentication tag  T:
00050:   CC 3A BA 11 8C E7 85 FD 77 78 94 D4 B5 20 69 F8

The result C | T:
00000:   43 FA 71 81 64 B1 E3 D7 1E 7B 65 39 A7 02 1D 52
00010:   69 9B 9E 1B 43 24 B7 52 95 74 E7 90 F2 BE 60 E8
00020:   11 62 C9 90 2A 2B 77 7F D9 6A D6 1A 99 E0 C6 DE
00030:   4B 91 D4 29 E3 1A 8C 11 AF F0 BC 47 F6 80 AF 14
00040:   40 1C C1 18 14 63 8E 76 24 83 37 75 16 34 70 08
00050:   CC 3A BA 11 8C E7 85 FD 77 78 94 D4 B5 20 69 F8
```

```
CBC-ACPKM-Master mode with AES-256
**********************************
k = 256
n = 128
c for the CTR-ACPKM mode = 64
N = 256
T* = 512

Initial key K:
00000:   88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77
00010:   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF

Initial vector IV:
00000:   12 34 56 78 90 AB CE F0 A1 B2 C3 D4 E5 F0 01 12

Plaintext P:
```

```
00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88
00010:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A
00020:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00
00030:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11
00040:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22
00050:   44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33
00060:   55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33 44

K^1 | K^2 | K^3 | K^4:
00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60
00020:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
00030:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3
00040:   E8 76 2B 30 8B 08 EB CE 3E 93 9A C2 C0 3E 76 D4
00050:   60 9A AB D9 15 33 13 D3 CF D3 94 E7 75 DF 3A 94
00060:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
00070:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12

Section_1

K^1:
00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60

Plaintext block P_1:
00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88

Input block P_1 (xor) C_0:
00000:   03 16 65 3C C5 CD B9 F0 5E 5C 1E 18 5E 5A 98 9A

Output block C_1:
00000:   59 CB 5B CA C2 69 2C 60 0D 46 03 A0 C7 40 C9 7C

Plaintext block P_2:
00000:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A

Input block P_2 (xor) C_1:
00000:   59 DA 79 F9 86 3C 4A 17 85 DF A9 1B 0B AE 36 76

Output block C_2:
00000:   80 B6 02 74 54 8B F7 C9 78 1F A1 05 8B F6 8B 42

Section_2

K^2:
00000:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
00010:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3
```

```
   Plaintext block P_3:
   00000:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00

   Input block P_3 (xor) C_2:
   00000:   91 94 31 30 01 ED 80 41 E1 B5 1A C9 65 09 81 42

   Output block C_3:
   00000:   8C 24 FB CF 68 15 B1 AF 65 FE 47 75 95 B4 97 59

   Plaintext block P_4:
   00000:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11

   Input block P_4 (xor) C_3:
   00000:   AE 17 BF 9A 0E 62 39 36 CF 45 8B 9B 6A BE 97 48

   Output block C_4:
   00000:   19 65 A5 00 58 0D 50 23 72 1B E9 90 E1 83 30 E9

   Section_3

   K^3:
   00000:   E8 76 2B 30 8B 08 EB CE 3E 93 9A C2 C0 3E 76 D4
   00010:   60 9A AB D9 15 33 13 D3 CF D3 94 E7 75 DF 3A 94

   Plaintext block P_5:
   00000:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22

   Input block P_5 (xor) C_4:
   00000:   2A 21 F0 66 2F 85 C9 89 C9 D7 07 6F EB 83 21 CB

   Output block C_5:
   00000:   56 D8 34 F4 6F 0F 4D E6 20 53 A9 5C B5 F6 3C 14

   Plaintext block P_6:
   00000:   44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33

   Input block P_6 (xor) C_5:
   00000:   12 8D 52 83 E7 96 E7 5D EC BD 56 56 B5 E7 1E 27

   Output block C_6:
   00000:   66 68 2B 8B DD 6E B2 7E DE C7 51 D6 2F 45 A5 45

   Section_4

   K^4:
   00000:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
   00010:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12
```

```
   Plaintext block P_7:
   00000:   55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33 44

   Input block P_7 (xor) C_6:
   00000:   33 0E 5C 03 44 C4 09 B2 30 38 5B D6 3E 67 96 01

   Output block C_7:
   00000:   7F 4D 87 F9 CA E9 56 09 79 C4 FA FE 34 0B 45 34

   Cipher text C:
   00000:   59 CB 5B CA C2 69 2C 60 0D 46 03 A0 C7 40 C9 7C
   00010:   80 B6 02 74 54 8B F7 C9 78 1F A1 05 8B F6 8B 42
   00020:   8C 24 FB CF 68 15 B1 AF 65 FE 47 75 95 B4 97 59
   00030:   19 65 A5 00 58 0D 50 23 72 1B E9 90 E1 83 30 E9
   00040:   56 D8 34 F4 6F 0F 4D E6 20 53 A9 5C B5 F6 3C 14
   00050:   66 68 2B 8B DD 6E B2 7E DE C7 51 D6 2F 45 A5 45
   00060:   7F 4D 87 F9 CA E9 56 09 79 C4 FA FE 34 0B 45 34



   CFB-ACPKM-Master mode with AES-256
   **********************************
   k = 256
   n = 128
   c for the CTR-ACPKM mode = 64
   N = 256
   T* = 512

   Initial key K:
   00000:   88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77
   00010:   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF

   Initial vector IV:
   00000:   12 34 56 78 90 AB CE F0 A1 B2 C3 D4 E5 F0 01 12

   Plaintext P:
   00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88
   00010:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A
   00020:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00
   00030:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11
   00040:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22
   00050:   44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33
   00060:   55 66 77 88 99 AA BB CC

   K^1 | K^2 | K^3 | K^4
   00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
   00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60
```

```
00020:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
00030:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3
00040:   E8 76 2B 30 8B 08 EB CE 3E 93 9A C2 C0 3E 76 D4
00050:   60 9A AB D9 15 33 13 D3 CF D3 94 E7 75 DF 3A 94
00060:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
00070:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12

Section_1

K^1:
00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60

Plaintext block P_1:
00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88

Encrypted block E_{K^1}(C_0):
00000:   1C 39 9D 59 F8 5D 91 91 A9 D2 12 9F 63 15 90 03

Output block C_1 = E_{K^1}(C_0) (xor) P_1:
00000:   0D 1B AE 1D AD 3B E6 91 56 3C CF 53 D8 BF 09 8B

Plaintext block P_2:
00000:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A

Encrypted block E_{K^1}(C_1):
00000:   6B A2 C5 42 52 69 C6 0B 15 14 06 87 90 46 F6 2E

Output block C_2 = E_{K^1}(C_1) (xor) P_2:
00000:   6B B3 E7 71 16 3C A0 7C 9D 8D AC 3C 5C A8 09 24

Section_2

K^2:
00000:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
00010:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3

Plaintext block P_3:
00000:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00

Encrypted block E_{K^2}(C_2):
00000:   95 45 5F DB C3 9E 0A 13 9F CB 10 F5 BD 79 A3 88

Output block C_3 = E_{K^2}(C_2) (xor) P_3:
00000:   84 67 6C 9F 96 F8 7D 9B 06 61 AB 39 53 86 A9 88

Plaintext block P_4:
00000:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11
```

```
Encrypted block E_{K^2}(C_3):
00000:   E0 AA 32 5D 80 A4 47 95 BA 42 BF 63 F8 4A C8 B2

Output block C_4 = E_{K^2}(C_3) (xor) P_4:
00000:   C2 99 76 08 E6 D3 CF 0C 10 F9 73 8D 07 40 C8 A3

Section_3

K^3:
00000:   E8 76 2B 30 8B 08 EB CE 3E 93 9A C2 C0 3E 76 D4
00010:   60 9A AB D9 15 33 13 D3 CF D3 94 E7 75 DF 3A 94

Plaintext block P_5:
00000:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22

Encrypted block E_{K^3}(C_4):
00000:   FE 42 8C 70 C2 51 CE 13 36 C1 BF 44 F8 49 66 89

Output block C_5 = E_{K^3}(C_4) (xor) P_5:
00000:   CD 06 D9 16 B5 D9 57 B9 8D 0D 51 BB F2 49 77 AB

Plaintext block P_6:
00000:   44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33

Encrypted block E_{K^3}(C_5):
00000:   01 24 80 87 86 18 A5 43 11 0A CC B5 0A E5 02 A3

Output block C_6 = E_{K^3}(C_5) (xor) P_6:
00000:   45 71 E6 F0 0E 81 0F F8 DD E4 33 BF 0A F4 20 90

Section_4

K^4:
00000:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
00010:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12

Plaintext block P_7:
00000:   55 66 77 88 99 AA BB CC

Encrypted block MSB_{|P_7|}(E_{K^4}(C_6)):
00000:   97 5C 96 37 55 1E 8C 7F

Output block C_7 = MSB_{|P_7|}(E_{K^4}(C_6)) (xor) P_7
00000:   C2 3A E1 BF CC B4 37 B3

Cipher text C:
00000:   0D 1B AE 1D AD 3B E6 91 56 3C CF 53 D8 BF 09 8B
00010:   6B B3 E7 71 16 3C A0 7C 9D 8D AC 3C 5C A8 09 24
```

```
00020:   84 67 6C 9F 96 F8 7D 9B 06 61 AB 39 53 86 A9 88
00030:   C2 99 76 08 E6 D3 CF 0C 10 F9 73 8D 07 40 C8 A3
00040:   CD 06 D9 16 B5 D9 57 B9 8D 0D 51 BB F2 49 77 AB
00050:   45 71 E6 F0 0E 81 0F F8 DD E4 33 BF 0A F4 20 90
00060:   C2 3A E1 BF CC B4 37 B3
```

OMAC-ACPKM-Master mode with AES-256

****

k = 256
n = 128
c for the CTR-ACPKM mode = 64
N = 256
T* = 768

Initial key K:

```
00000:   88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77
00010:   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF
```

Plaintext M:

```
00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88
00010:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A
00020:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00
00030:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11
00040:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22
```

K^1 | K^1_1 | K^2 | K^2_1 | K^3 | K^3_1:

```
00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60
00020:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
00030:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3
00040:   9D CC 66 42 0D FF 45 5B 21 F3 93 F0 D4 D6 6E 67
00050:   BB 1B 06 0B 87 66 6D 08 7A 9D A7 49 55 C3 5B 48
00060:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
00070:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12
00080:   78 21 C7 C7 6C BD 79 63 56 AC F8 8E 69 6A 00 07
```

Section_1

K^1:

```
00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60
```

K^1_1:

```
00000:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
```

```
   Plaintext block M_1:
   00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88

   Input block M_1 (xor) C_0:
   00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88

   Output block C_1:
   00000:   0B A5 89 BF 55 C1 15 42 53 08 89 76 A0 FE 24 3E

   Plaintext block M_2:
   00000:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A

   Input block M_2 (xor) C_1:
   00000:   0B B4 AB 8C 11 94 73 35 DB 91 23 CD 6C 10 DB 34

   Output block C_2:
   00000:   1C 53 DD A3 6D DC E1 17 ED 1F 14 09 D8 6A F3 2C

   Section_2

   K^2:
   00000:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3
   00010:   9D CC 66 42 0D FF 45 5B 21 F3 93 F0 D4 D6 6E 67

   K^2_1:
   00000:   BB 1B 06 0B 87 66 6D 08 7A 9D A7 49 55 C3 5B 48

   Plaintext block M_3:
   00000:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00

   Input block M_3 (xor) C_2:
   00000:   0D 71 EE E7 38 BA 96 9F 74 B5 AF C5 36 95 F9 2C

   Output block C_3:
   00000:   4E D4 BC A6 CE 6D 6D 16 F8 63 85 13 E0 48 59 75

   Plaintext block M_4:
   00000:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11

   Input block M_4 (xor) C_3:
   00000:   6C E7 F8 F3 A8 1A E5 8F 52 D8 49 FD 1F 42 59 64

   Output block C_4:
   00000:   B6 83 E3 96 FD 30 CD 46 79 C1 8B 24 03 82 1D 81

   Section_3

   K^3:
```

```
00000:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
00010:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12

K^3_1:
00000:   78 21 C7 C7 6C BD 79 63 56 AC F8 8E 69 6A 00 07

MSB1(K1) == 0 -> K2 = K1 << 1

K1:
00000:   78 21 C7 C7 6C BD 79 63 56 AC F8 8E 69 6A 00 07

K2:
00000:   F0 43 8F 8E D9 7A F2 C6 AD 59 F1 1C D2 D4 00 0E

Plaintext M_5:
00000:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22

Using K1, padding is not required

Input block M_5 (xor) C_4:
00000:   FD E6 71 37 E6 05 2D 8F 94 A1 9D 55 60 E8 0C A4

Output block C_5:
00000:   B3 AD B8 92 18 32 05 4C 09 21 E7 B8 08 CF A0 B8

Message authentication code T:
00000:   B3 AD B8 92 18 32 05 4C 09 21 E7 B8 08 CF A0 B8
```

Appendix B. Contributors

- Russ Housley
  Vigil Security, LLC
  housley@vigilsec.com

- Evgeny Alekseev
  CryptoPro
  alekseev@cryptopro.ru

- Ekaterina Smyshlyaeva
  CryptoPro
  ess@cryptopro.ru

- Shay Gueron
  University of Haifa, Israel
  Intel Corporation, Israel Development Center, Israel
  shay.gueron@gmail.com

- Daniel Fox Franke
  Akamai Technologies
  dfoxfranke@gmail.com

- Lilia Ahmetzyanova
  CryptoPro
  lah@cryptopro.ru

Appendix C. Acknowledgments

We thank Mihir Bellare, Scott Fluhrer, Dorothy Cooley, Yoav Nir, Jim Schaad, Paul Hoffman, Dmitry Belyavsky, Yaron Sheffer, Alexey Melnikov and Spencer Dawkins for their useful comments.

Author's Address

Stanislav Smyshlyaev (editor)
CryptoPro
18, Suschevskiy val
Moscow 127018
Russian Federation

Phone: +7 (495) 995-48-20
Email: svs@cryptopro.ru

Network Working Group
Internet-Draft
Intended status: Informational
Expires: April 19, 2018

W. Ladd
UC Berkeley
B. Kaduk, Ed.
Akamai

October 16, 2017

SPAKE2, a PAKE
draft-irtf-cfrg-spake2-04

Abstract

This Internet-Draft describes SPAKE2, a secure, efficient password based key exchange protocol.

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 April 19, 2018.

Copyright Notice

Table of Contents

1. Introduction

This document describes a means for two parties that share a password to derive a shared key. This method is compatible with any group, is computationally efficient, and has a strong security proof.

2. Definition of SPAKE2

2.1. Setup

Let G be a group in which the Diffie-Hellman problem is hard of order ph, with p a big prime and h a cofactor. We denote the operations in the group additively. Let H be a hash function from arbitrary strings to bit strings of a fixed length. Common choices for H are SHA256 or SHA512. We assume there is a representation of elements of G as byte strings: common choices would be SEC1 uncompressed [SEC1] for elliptic curve groups or big endian integers of a particular length for prime field DH.

|| denotes concatenation of strings. We also let len(S) denote the length of a string in bytes, represented as an eight-byte little-endian number.

We fix two elements M and N as defined in the table in this document for common groups, as well as a generator G of the group. G is specified in the document defining the group, and so we do not recall it here.

Let A and B be two parties. We will assume that A and B are also representations of the parties such as MAC addresses or other names (hostnames, usernames, etc). We assume they share an integer w. Typically w will be the hash of a user-supplied password, truncated and taken mod p. Protocols using this protocol must define the method used to compute w: it may be necessary to carry out normalization. The hashing algorithm SHOULD be designed to slow down brute force attackers.

We present two protocols below. Note that it is insecure to use the same password with both protocols, this MUST NOT be done.

2.2. SPAKE2

A picks x randomly and uniformly from the integers in [0,ph) divisible by h, and calculates X=xG and T=wM+X, then transmits T to B.

B selects y randomly and uniformly from the integers in [0,ph), divisible by h and calculates Y=yG, S=wN+Y, then transmits S to A.

Both A and B calculate a group element K. A calculates it as x(S-wN), while B calculates it as y(T-wM). A knows S because it has received it, and likewise B knows T.

This K is a shared secret, but the scheme as described is not secure. K MUST be combined with the values transmitted and received via a hash function to have a secure protocol. If higher-level protocols prescribe a method for doing so, that SHOULD be used. Otherwise we can compute K' as H(len(A)||A||len(B)||B||len(S)||S|| len(T)||T||len(K)||K || len(w) || w) and use K' as the key.

2.3. SPAKE2+

This protocol and security proof appear in [TDH]. We use the same setup as for SPAKE2, except that we have two secrets, w0 and w1. The server, here Bob, stores L=w1*g and w0.

When executing SPAKE2+, A selects x uniformly at random from the numbers in the range [0, ph) divisible by h, and lets X=xG+w0*M, then transmits X to B. B selects y uniformly at random from the numbers in [0, ph) divisible by h, then computes Y=yG+w0*N, and transmits it to Alice.

A computes Z as x(Y-w0*N), and V as w1(Y-w0*N). B computes Z as y(X-w0*M) and V as yL. Both share Z and V as common keys. It is essential that both Z and V be used in combination with the transcript to derive the keying material. For higher-level protocols without sufficient transcript hashing, let K' be H(len(A)||A||len(B)||B||len(X)||X||len(Y)||Y||len(Z)||Z||len(V)||V) and use K' as the established key.

3. Table of points for common groups

Every curve presented in the table below has an OID from [RFC5480]. We construct a string using the OID and the needed constant, for instance "1.3.132.0.35 point generation seed (M)" for P-512. This

string is turned into an infinite sequence of bytes by hashing with SHA256, and hashing that output again to generate the next 32 bytes, and so on. This pattern is repeated for each group and value, with the string modified appropriately.

The initial segment of bytes of length equal to that of an encoded group element is taken, and is then formatted as required for the group. In the case of Weierstrass points, this means setting the first byte to 0x02 or 0x03 depending on the low-order bit. For Ed25519 style formats this means taking all the bytes as the representation of the group element. This string of bytes is then interpreted as a point in the group. If this is impossible, then the next non-overlapping segment of sufficient length is taken. We multiply that point by the cofactor h, and if that is not the identity, output it.

These bytestrings are compressed points as in [SEC1] for curves from [SEC1].

For P256:

M =
02886e2f97ace46e55ba9dd7242579f2993b64e16ef3dcab95afd497333d8fa12f

N =
03d8bbd6c639c62937b04d997f38c3770719c629d7014d49a24b4f98baa1292b49

For P384:

M =
030ff0895ae5ebf6187080a82d82b42e2765e3b2f8749c7e05eba366434b363d3dc
36f15314739074d2eb8613fceec2853

N =
02c72cf2e390853a1c1c4ad816a62fd15824f56078918f43f922ca21518f9c543bb
252c5490214cf9aa3f0baab4b665c10

For P521:

M =
02003f06f38131b2ba2600791e82488e8d20ab889af753a41806c5db18d37d85608
cfae06b82e4a72cd744c719193562a653ea1f119eef9356907edc9b56979962d7aa

N =
0200c7924b9ec017f3094562894336a53c50167ba8c5963876880542bc669e494b25
32d76c5b53dfb349fdf69154b9e0048c58a42e8ed04cef052a3bc349d95575cd25

The following python snippet generates the above points:

```
def canon_pointstr(self, s):
    return chr(ord(s[0]) & 1 | 2) + s[1:]

def iterated_hash(seed, n):
    h = seed
    for i in xrange(n):
        h = SHA256.new(h).digest()
    return h

def bighash(seed, start, sz):
    n = -(-sz // 32)
    hashes = [iterated_hash(seed, i) for i in xrange(start, start + n)]
    return ''.join(hashes)[:sz]

def gen_point(seed, ec, order):
    for i in xrange(1, 1000):
        pointstr = ec.canon_pointstr(bighash(seed, i, ec.nbytes_point()))
        try:
            p = ec.decode_point(pointstr)
            if ec.mul(p, order) == ec.identity():
                return pointstr, i
        except Exception:
            pass
```

4. Security Considerations

A security proof of SPAKE2 for prime order groups is found in [REF]. Note that the choice of M and N is critical for the security proof. The generation method specified in this document is designed to eliminate concerns related to knowing discrete logs of M and N.

SPAKE2+ appears in [TDH], along with proof.

There is no key-confirmation as this is a one round protocol. It is expected that a protocol using this key exchange mechanism provides key confirmation separately if desired.

Elements should be checked for group membership: failure to properly validate group elements can lead to attacks. In particular it is essential to verify that received points are valid compressions of points on an elliptic curve when using elliptic curves. It is not necessary to validate membership in the prime order subgroup: the multiplication by cofactors eliminates this issue.

The choices of random numbers MUST BE uniform. Note that to pick a random multiple of h in [0, ph) one can pick a random integer in [0,p) and multiply by h. Reuse of ephemerals results in dictionary attacks and MUST NOT be done.

SPAKE2 does not support augmentation. As a result, the server has to store a password equivalent. This is considered a significant drawback, and so SPAKE2+ also appears in this document.

As specified the shared secret K is not suitable for use as a shared key. It MUST be passed to a hash function along with the public values used to derive it and the party identities to avoid attacks. In protocols which do not perform this separately, the value denoted K' MUST be used instead.

5. IANA Considerations

No IANA action is required.

6. Acknowledgments

Special thanks to Nathaniel McCallum for generation of test vectors. Thanks to Mike Hamburg for advice on how to deal with cofactors. Greg Hudson suggested addition of warnings on the reuse of x and y. Thanks to Fedor Brunner, Adam Langley, and the members of the CFRG for comments and advice. Trevor Perrin informed me of SPAKE2+.

7. References

[REF] Abdalla, M. and D. Pointcheval, "Simple Password-Based Encrypted Key Exchange Protocols.", Feb 2005.

Appears in A. Menezes, editor. Topics in Cryptography-CT-RSA 2005, Volume 3376 of Lecture Notes in Computer Science, pages 191-208, San Francisco, CA, US. Springer-Verlag, Berlin, Germany.

[RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, DOI 10.17487/RFC5480, March 2009, <https://www.rfc-editor.org/info/rfc5480>.

[SEC1] SEC, "STANDARDS FOR EFFICIENT CRYPTOGRAPHY, "SEC 1: Elliptic Curve Cryptography", version 2.0", May 2009.

[TDH] Cash, D., Kiltz, E., and V. Shoup, "The Twin-Diffie Hellman Problem and Applications", 2008.

EUROCRYPT 2008. Volume 4965 of Lecture notes in Computer Science, pages 127-145. Springer-Verlag, Berlin, Germany.

Authors' Addresses

Watson Ladd
UC Berkeley

Email: watsonbladd@gmail.com

Benjamin Kaduk (editor)
Akamai Technologies

Email: kaduk@mit.edu

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 12, 2019

W. Ladd
UC Berkeley
B. Kaduk, Ed.
Akamai

March 11, 2019

SPAKE2, a PAKE
draft-irtf-cfrg-spake2-08

Abstract

This document describes SPAKE2 and its augmented variant SPAKE2+, which are protocols for two parties that share a password to derive a strong shared key with no risk of disclosing the password. This method is compatible with any prime order group, is computationally efficient, and SPAKE2 (but not SPAKE2+) has a security proof.

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 September 12, 2019.

Table of Contents

1. Introduction

This document describes SPAKE2, a means for two parties that share a password to derive a strong shared key with no risk of disclosing the password. This password-based key exchange protocol is compatible with any group (requiring only a scheme to map a random input of fixed length per group to a random group element), is computationally efficient, and has a security proof. Predetermined parameters for a selection of commonly used groups are also provided for use by other protocols.

2. Requirements Notation

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. Definition of SPAKE2

3.1. Setup

Let G be a group in which the computational Diffie-Hellman (CDH) problem is hard. Suppose G has order p*h where p is a large prime; h will be called the cofactor. Let I be the unit element in G, e.g., the point at infinity if G is an elliptic curve group. We denote the operations in the group additively. We assume there is a representation of elements of G as byte strings: common choices would be SEC1 compressed [SEC1] for elliptic curve groups or big endian integers of a fixed (per-group) length for prime field DH. We fix

two elements M and N in the prime-order subgroup of G as defined in the table in this document for common groups, as well as a generator P of the (large) prime-order subgroup of G. P is specified in the document defining the group, and so we do not repeat it here.

|| denotes concatenation of strings. We also let len(S) denote the length of a string in bytes, represented as an eight-byte little-endian number. Finally, let nil represent an empty string, i.e., len(nil) = 0.

KDF is a key-derivation function that takes as input a salt, intermediate keying material (IKM), info string, and derived key length L to derive a cryptographic key of length L. MAC is a Message Authentication Code algorithm that takes a secret key and message as input to produce an output. Let Hash be a hash function from arbitrary strings to bit strings of a fixed length. Common choices for H are SHA256 or SHA512 [RFC6234]. Let MHF be a memory-hard hash function designed to slow down brute-force attackers. Scrypt [RFC7914] is a common example of this function. The output length of MHF matches that of Hash. Parameter selection for MHF is out of scope for this document. Section 5 specifies variants of KDF, MAC, Hash, and MHF suitable for use with the protocols contained herein.

Let A and B be two parties. A and B may also have digital representations of the parties' identities such as Media Access Control addresses or other names (hostnames, usernames, etc). A and B may share Additional Authenticated Data (AAD) of length at most 2^16 - 1 bits that is separate from their identities which they may want to include in the protocol execution. One example of AAD is a list of supported protocol versions if SPAKE2(+) were used in a higher-level protocol which negotiates use of a particular PAKE. Including this list would ensure that both parties agree upon the same set of supported protocols and therefore prevent downgrade attacks. We also assume A and B share an integer w; typically w = MHF(pw) mod p, for a user-supplied password pw. Standards such NIST.SP.800-56Ar3 suggest taking mod p of a hash value that is 64 bits longer than that needed to represent p to remove statistical bias introduced by the modulation. Protocols using this specification must define the method used to compute w: it may be necessary to carry out various forms of normalization of the password before hashing [RFC8265]. The hashing algorithm SHOULD be a MHF so as to slow down brute-force attackers.

We present two protocols below. Note that it is insecure to use the same password with both protocols; passwords MUST NOT be used for both SPAKE2 and SPAKE2+.

3.2. SPAKE2

To begin, A picks x randomly and uniformly from the integers in [0,p), and calculates X=x*P and T=w*M+X, then transmits T to B. Upon receipt of T, B computes T*h and aborts if the result is equal to I. (This ensures T is in the prime order subgroup of G.)

B selects y randomly and uniformly from the integers in [0,p), and calculates Y=y*P, S=w*N+Y, then transmits S to A. Upon receipt of S, A computes S*h and aborts if the result is equal to I.

Both A and B calculate a group element K. A calculates it as x*(S-wN), while B calculates it as y*(T-w*M). A knows S because it has received it, and likewise B knows T. A and B multiply protocol messages from each peer by h so as to avoid small subgroup attacks, but the result of the multiplication is not used for operations other than the comparison against I and the non-multiplied value is used in subsequent calculations.

K is a shared value, though it MUST NOT be used as a shared secret. Both A and B must derive two shared secrets from K and the protocol transcript. This prevents man-in-the-middle attackers from inserting themselves into the exchange. The transcript TT is encoded as follows:

```
TT = len(A) || A || len(B) || B || len(S) || S || len(T) || T
     || len(K) || K || len(w) || w
```

If an identity is absent, it is omitted from the transcript entirely. For example, if both A and B are absent, then TT = len(S) || S || len(T) || T || len(K) || K || len(w) || w. Likewise, if only A is absent, TT = len(B) || B || len(S) || S || len(T) || T || len(K) || K || len(w) || w. This must only be done for applications in which identities are implicit. Otherwise, the protocol risks Unknown Key Share attacks (discussion of Unknown Key Share attacks in a specific protocl is given in [I-D.ietf-mmusic-sdp-uks].

Upon completion of this protocol, A and B compute shared secrets Ke, KcA, and KcB as specified in Section 4. A MUST send B a key confirmation message so both parties agree upon these shared secrets. This confirmation message F is computed as a MAC over the protocol transcript TT using KcA, as follows: F = MAC(KcA, TT). Similarly, B MUST send A a confirmation message using a MAC computed equivalently except with the use of KcB. Key confirmation verification requires computing F and checking for equality against that which was received.

3.3. SPAKE2+

This protocol appears in [TDH]. We use the same setup as for SPAKE2, except that we have two secrets, w0 and w1, derived by hashing the password pw with the identities of the two participants, A and B. Specifically, w0s || w1s = MHF(len(pw) || pw || len(A) || A || len(B) || B), and then computing w0 = w0s mod p and w1 = w1s mod p. The length of each of w0s and w1s is equal to half of the MHF output, e.g., |w0s| = |w1s| = 128 bits for scrypt. w0 and w1 MUST NOT equal I. If they are, they MUST be iteratively regenerated by computing w0s || w1s = MHF(len(pw) || pw || len(A) || A || len(B) || B || 0x0000), where 0x0000 is 16-bit increasing counter. This process must repeat until valid w0 and w1 are produced. B stores L=w1*P and w0.

When executing SPAKE2+, A selects x uniformly at random from the numbers in the range [0, p), and lets X=x*P+w0*M, then transmits X to B. Upon receipt of X, A computes h*X and aborts if the result is equal to I. B then selects y uniformly at random from the numbers in [0, p), then computes Y=y*P+w0*N, and transmits Y to A. Upon receipt of Y, A computes Y*h and aborts if the result is equal to I.

A computes Z as x*(Y-w0*N), and V as w1*(Y-w0*N). B computes Z as y*(X- w0*M) and V as y*L. Both share Z and V as common keys. It is essential that both Z and V be used in combination with the transcript to derive the keying material. The protocol transcript encoding is shown below.

```
TT = len(A) || A || len(B) || B || len(X) || X || len(Y) || Y
 || len(Z) || Z || len(V) || V || len(w0) || w0
```

As in Section 3.2, inclusion of A and B in the transcript is optional depending on whether or not the identities are implicit.

Upon completion of this protocol, A and B follow the same key derivation and confirmation steps as outlined in Section 3.2.

4. Key Schedule and Key Confirmation

The protocol transcript TT, as defined in Sections Section 3.3 and Section 3.2, is unique and secret to A and B. Both parties use TT to derive shared symmetric secrets Ke and Ka as Ke || Ka = Hash(TT). The length of each key is equal to half of the digest output, e.g., |Ke| = |Ka| = 128 bits for SHA-256.

Both endpoints use Ka to derive subsequent MAC keys for key confirmation messages. Specifically, let KcA and KcB be the MAC keys used by A and B, respectively. A and B compute them as KcA || KcB =

KDF(nil, Ka, "ConfirmationKeys" || AAD), where AAD is the associated data each given to each endpoint, or nil if none was provided. The length of each of KcA and KcB is equal to half of the KDF output, e.g., |KcA| = |KcB| = 128 bits for HKDF(SHA256).

The resulting key schedule for this protocol, given transcript TT and additional associated data AAD, is as follows.

```
TT   -> Hash(TT) = Ke || Ka
AAD -> KDF(nil, Ka, "ConfirmationKeys" || AAD) = KcA || KcB
```

A and B output Ke as the shared secret from the protocol. Ka and its derived keys are not used for anything except key confirmation.

5. Ciphersuites

This section documents SPAKE2 and SPAKE2+ ciphersuite configurations. A ciphersuite indicates a group, cryptographic hash algorithm, and pair of KDF and MAC functions, e.g., SPAKE2-P256-SHA256-HKDF-HMAC. This ciphersuite indicates a SPAKE2 protocol instance over P-256 that uses SHA256 along with HKDF [RFC5869] and HMAC [RFC2104] for G, Hash, KDF, and MAC functions, respectively.

G	Hash	KDF	MAC	MHF
P-256	SHA256 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]	scrypt [RFC7914]
P-256	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]	scrypt [RFC7914]
P-384	SHA256 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]	scrypt [RFC7914]
P-384	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]	scrypt [RFC7914]
P-512	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]	scrypt [RFC7914]
edwards25519 [RFC7748]	SHA256 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]	scrypt [RFC7914]
edwards448 [RFC7748]	SHA512 [RFC6234]	HKDF [RFC5869]	HMAC [RFC2104]	scrypt [RFC7914]
P-256	SHA256 [RFC6234]	HKDF [RFC5869]	CMAC-AES-128 [RFC4493]	scrypt [RFC7914]
P-256	SHA512 [RFC6234]	HKDF [RFC5869]	CMAC-AES-128 [RFC4493]	scrypt [RFC7914]

Table 1: SPAKE2(+) Ciphersuites

The following points represent permissible point generation seeds for the groups listed in the Table Table 1, using the algorithm presented in Appendix A. These bytestrings are compressed points as in [SEC1] for curves from [SEC1].

For P256:

M =
02886e2f97ace46e55ba9dd7242579f2993b64e16ef3dcab95afd497333d8fa12f
seed: 1.2.840.10045.3.1.7 point generation seed (M)

N =
03d8bbd6c639c62937b04d997f38c3770719c629d7014d49a24b4f98baa1292b49
seed: 1.2.840.10045.3.1.7 point generation seed (N)

For P384:

M =
030ff0895ae5ebf6187080a82d82b42e2765e3b2f8749c7e05eba366434b363d3dc
36f15314739074d2eb8613fceec2853
seed: 1.3.132.0.34 point generation seed (M)

N =
02c72cf2e390853a1c1c4ad816a62fd15824f56078918f43f922ca21518f9c543bb
252c5490214cf9aa3f0baab4b665c10
seed: 1.3.132.0.34 point generation seed (N)

For P521:

M =
02003f06f38131b2ba2600791e82488e8d20ab889af753a41806c5db18d37d85608
cfae06b82e4a72cd744c719193562a653ea1f119eef9356907edc9b56979962d7aa
seed: 1.3.132.0.35 point generation seed (M)

N =
0200c7924b9ec017f3094562894336a53c50167ba8c5963876880542bc669e494b25
32d76c5b53dfb349fdf69154b9e0048c58a42e8ed04cef052a3bc349d95575cd25
seed: 1.3.132.0.35 point generation seed (N)

For edwards25519:

M =
d048032c6ea0b6d697ddc2e86bda85a33adac920f1bf18e1b0c6d166a5cecdaf
seed: edwards25519 point generation seed (M)

N =
d3bfb518f44f3430f29d0c92af503865a1ed3281dc69b35dd868ba85f886c4ab
seed: edwards25519 point generation seed (N)

For edwards448:

M =
b6221038a775ecd007a4e4dde39fd76ae91d3cf0cc92be8f0c2fa6d6b66f9a12
942f5a92646109152292464f3e63d354701c7848d9fc3b8880
seed: edwards448 point generation seed (M)

N =
6034c65b66e4cd7a49b0edec3e3c9ccc4588afd8cf324e29f0a84a072531c4db
f97ff9af195ed714a689251f08f8e06e2d1f24a0ffc0146600
seed: edwards448 point generation seed (N)

6. Security Considerations

A security proof of SPAKE2 for prime order groups is found in [REF]. Note that the choice of M and N is critical for the security proof. The generation method specified in this document is designed to eliminate concerns related to knowing discrete logs of M and N.

SPAKE2+ appears in [TDH] along with a path to a proof that server compromise does not lead to password compromise under the DH assumption (though the corresponding model excludes precomputation attacks).

Elements received from a peer MUST be checked for group membership: failure to properly validate group elements can lead to attacks. Beyond the cofactor multiplication checks to ensure that these elements are in the prime order subgroup of G, it is essential that endpoints verify received points are members of G.

The choices of random numbers MUST BE uniform. Randomly generated values (e.g., x and y) MUST NOT be reused; such reuse may permit dictionary attacks on the password.

SPAKE2 does not support augmentation. As a result, the server has to store a password equivalent. This is considered a significant drawback, and so SPAKE2+ also appears in this document.

7. IANA Considerations

No IANA action is required.

8. Acknowledgments

Special thanks to Nathaniel McCallum and Greg Hudson for generation of test vectors. Thanks to Mike Hamburg for advice on how to deal with cofactors. Greg Hudson also suggested the addition of warnings on the reuse of x and y. Thanks to Fedor Brunner, Adam Langley, and the members of the CFRG for comments and advice. Chris Wood contributed substantial text and reformatting to address the excellent review comments from Kenny Paterson. Trevor Perrin informed me of SPAKE2+.

9. References

9.1. Normative References

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[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>.

[RFC4493] Song, JH., Poovendran, R., Lee, J., and T. Iwata, "The AES-CMAC Algorithm", RFC 4493, DOI 10.17487/RFC4493, June 2006, <https://www.rfc-editor.org/info/rfc4493>.

[RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, DOI 10.17487/RFC5480, March 2009, <https://www.rfc-editor.org/info/rfc5480>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[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, <https://www.rfc-editor.org/info/rfc6234>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[RFC7914] Percival, C. and S. Josefsson, "The scrypt Password-Based Key Derivation Function", RFC 7914, DOI 10.17487/RFC7914, August 2016, <https://www.rfc-editor.org/info/rfc7914>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[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>.

[SEC1] SEC, "STANDARDS FOR EFFICIENT CRYPTOGRAPHY, "SEC 1: Elliptic Curve Cryptography", version 2.0", May 2009.

9.2. Informative References

[I-D.ietf-mmusic-sdp-uks]
Thomson, M. and E. Rescorla, "Unknown Key Share Attacks on uses of TLS with the Session Description Protocol (SDP)", draft-ietf-mmusic-sdp-uks-03 (work in progress), January 2019.

[REF] Abdalla, M. and D. Pointcheval, "Simple Password-Based Encrypted Key Exchange Protocols.", Feb 2005.

Appears in A. Menezes, editor. Topics in Cryptography-CT-RSA 2005, Volume 3376 of Lecture Notes in Computer Science, pages 191-208, San Francisco, CA, US. Springer-Verlag, Berlin, Germany.

[RFC8265] Saint-Andre, P. and A. Melnikov, "Preparation, Enforcement, and Comparison of Internationalized Strings Representing Usernames and Passwords", RFC 8265, DOI 10.17487/RFC8265, October 2017, <https://www.rfc-editor.org/info/rfc8265>.

[TDH] Cash, D., Kiltz, E., and V. Shoup, "The Twin-Diffie Hellman Problem and Applications", 2008.

EUROCRYPT 2008. Volume 4965 of Lecture notes in Computer Science, pages 127-145. Springer-Verlag, Berlin, Germany.

Appendix A. Algorithm used for Point Generation

This section describes the algorithm that was used to generate the points (M) and (N) in the table in Section 5.

For each curve in the table below, we construct a string using the curve OID from [RFC5480] (as an ASCII string) or its name, combined with the needed constant, for instance "1.3.132.0.35 point generation seed (M)" for P-512. This string is turned into a series of blocks by hashing with SHA256, and hashing that output again to generate the next 32 bytes, and so on. This pattern is repeated for each group and value, with the string modified appropriately.

A byte string of length equal to that of an encoded group element is constructed by concatenating as many blocks as are required, starting from the first block, and truncating to the desired length. The byte string is then formatted as required for the group. In the case of Weierstrass curves, we take the desired length as the length for representing a compressed point (section 2.3.4 of [SEC1]), and use the low-order bit of the first byte as the sign bit. In order to

obtain the correct format, the value of the first byte is set to 0x02 or 0x03 (clearing the first six bits and setting the seventh bit), leaving the sign bit as it was in the byte string constructed by concatenating hash blocks. For the [RFC8032] curves a different procedure is used. For edwards448 the 57-byte input has the least-significant 7 bits of the last byte set to zero, and for edwards25519 the 32-byte input is not modified. For both the [RFC8032] curves the (modified) input is then interpreted as the representation of the group element. If this interpretation yields a valid group element with the correct order (p), the (modified) byte string is the output. Otherwise, the initial hash block is discarded and a new byte string constructed from the remaining hash blocks. The procedure of constructing a byte string of the appropriate length, formatting it as required for the curve, and checking if it is a valid point of the correct order, is repeated until a valid element is found.

The following python snippet generates the above points, assuming an elliptic curve implementation following the interface of Edwards25519Point.stdbase() and Edwards448Point.stdbase() in Appendix A of [RFC8032]:

```
  def iterated_hash(seed, n):
      h = seed
      for i in range(n):
          h = hashlib.sha256(h).digest()
      return h

  def bighash(seed, start, sz):
      n = -(-sz // 32)
      hashes = [iterated_hash(seed, i) for i in range(start, start + n)]
      return b''.join(hashes)[:sz]

  def canon_pointstr(ecname, s):
      if ecname == 'edwards25519':
          return s
      elif ecname == 'edwards448':
          return s[:-1] + bytes([s[-1] & 0x80])
      else:
          return bytes([(s[0] & 1) | 2]) + s[1:]

  def gen_point(seed, ecname, ec):
      for i in range(1, 1000):
          hval = bighash(seed, i, len(ec.encode()))
          pointstr = canon_pointstr(ecname, hval)
          try:
              p = ec.decode(pointstr)
              if p != ec.zero_elem() and p * p.l() == ec.zero_elem():
                  return pointstr, i
          except Exception:
              pass
```

Authors' Addresses

Watson Ladd
UC Berkeley

Email: watsonbladd@gmail.com

Benjamin Kaduk (editor)
Akamai Technologies

Email: kaduk@mit.edu