Network Working Group
Internet-Draft
Intended status: Informational
Expires: June 17, 2021

S. Smyshlyaev, Ed.
E. Alekseev
E. Griboedova
A. Babueva
CryptoPro

December 14, 2020

GOST Cipher Suites for Transport Layer Security (TLS) Protocol Version 1.3
draft-smyshlyaev-tls13-gost-suites-03

Abstract

The purpose of this document is to make the Russian cryptographic standards available to the Internet community for their implementation in the Transport Layer Security (TLS) Protocol Version 1.3.

This specification defines four new cipher suites and seven new signature schemes based on GOST R 34.12-2015, GOST R 34.11-2012 and GOST R 34.10-2012 algorithms.

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 June 17, 2021.

Copyright Notice

Table of Contents

1. Introduction

This document defines four new cipher suites (the TLS13_GOST cipher suites) and seven new signature schemes (the TLS13_GOST signature schemes) for the Transport Layer Security (TLS) Protocol Version 1.3, that are based on Russian cryptographic standards GOST R 34.12-2015 [GOST3412-2015] (the English version can be found in [RFC7801]), GOST R 34.11-2012 [GOST3411-2012] (the English version can be found in [RFC6986]) and GOST R 34.10-2012 [GOST3410-2012] (the English version can be found in [RFC7091]).

The TLS13_GOST cipher suites (see Section 4) have the following values:

```
TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_L = {0xC1, 0x03};
TLS_GOSTR341112_256_WITH_MAGMA_MGM_L = {0xC1, 0x04};
TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_S = {0xC1, 0x05};
TLS_GOSTR341112_256_WITH_MAGMA_MGM_S = {0xC1, 0x06}.
```

Each TLS13_GOST cipher suite specifies a pair (record protection algorithm, hash algorithm) such that:

- The record protection algorithm is the AEAD algorithm (see Section 4.1.1) based on the GOST R 34.12-2015 block cipher [RFC7801] in the Multilinear Galois Mode (MGM) [DraftMGM] and the external re-keying approach (see [RFC8645]) intended for increasing the lifetime of symmetric keys used to protect records.

- The hash algorithm is the GOST R 34.11-2012 algorithm [RFC6986].

Note: The TLS13_GOST cipher suites are divided into two types (depending on the key lifetime limitations, see Section 4.1.2 and Section 4.1.3): the "_S" (strong) cipher suites and the "_L" (light) cipher suites.

The TLS13_GOST signature schemes that can be used with the TLS13_GOST cipher suites have the following values:

```
gostr34102012_256a = 0x0709;

gostr34102012_256b = 0x070A;

gostr34102012_256c = 0x070B;

gostr34102012_256d = 0x070C;

gostr34102012_512a = 0x070D;
```

gostr34102012_512b = 0x070E;

gostr34102012_512c = 0x070F.

Each TLS13_GOST signature scheme specifies a pair (signature algorithm, elliptic curve) such that:

- The signature algorithm is the GOST R 34.10-2012 algorithm [RFC7091].
- The elliptic curve is one of the curves defined in Section 5.2.

Additionally, this document specifies the key exchange and authentication process in case of negotiating TLS13_GOST cipher suites (see Section 6).

2. Conventions Used in This Document

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. Basic Terms and Definitions

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

B_t	the set of byte strings of length t, t >= 0, for t = 0 the B_t set consists of a single empty string of zero length. If A is an element of B_t, then A = (a_1, a_2, ... , a_t), where a_1, a_2, ... , a_t are in {0, ... , 255};
B*	the set of all byte strings of a finite length (hereinafter referred to as strings), including the empty string;
A[i..j]	the string A[i..j] = (a_i, a_{i+1}, ... , a_j) in B_{j-i+1}, where A = (a_1, a_2, ... , a_t) in B_t and 1<=i<=j<=t;
|A|	the byte length of the string A;
A | C	the concatenation of strings A and C both belonging to B*, i.e., a string in B_{|A|+|C|}, where the left

	substring in B_|A| is equal to A, and the right substring in B_|C| is equal to C;
i & j	bitwise AND of integers i and j;
STR_t	the byte string STR_t(i) = (i_1, ... , i_t) in B_t corresponding to an integer i = 256^{t-1} * i_1 + ... + 256 * i_{t-1} + i_t (the interpretation of the integer as a byte string in big-endian format);
str_t	the byte string str_t(i) = (i_1, ... , i_t) in B_t corresponding to an integer i = 256^{t-1} * i_t + ... + 256 * i_2 + i_1 (the interpretation of the integer as a byte string in little-endian format);
k	the byte-length of the block cipher key;
n	the byte-length of the block cipher block;
IVlen	the byte-length of the initialization vector;
S	the byte-length of the authentication tag;
E_i	the elliptic curve indicated by client in "supported_groups" extension;
O_i	the zero point of the elliptic curve E_i;
m_i	the order of group of points belonging to the elliptic curve E_i;
q_i	the cyclic subgroup order of group of points belonging to the elliptic curve E_i;
h_i	the cyclic subgroup cofactor which is equal to m_i / q_i;
Q_sign	the public key stored in endpoint's certificate;
d_sign	the private key that corresponds to the Q_sign key;
P_i	the point of the elliptic curve E_i of the order q_i;
(d_C^i, Q_C^i)	the client's ephemeral key pair which consists of the private key and the public key corresponding to the elliptic curve E_i;

(d_S^i, Q_S^i) the server's ephemeral key pair which consists of the private key and the public key corresponding to the elliptic curve E_i.

4. Cipher Suite Definition

The cipher suite value is used to indicate a record protection algorithm and a hash algorithm which an endpoint supports (see Section 4.1.2 of [RFC8446]).

This section defines the following four TLS13_GOST cipher suites that can be used to support Russian cryptographic algorithms:

```
CipherSuite TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_L = {0xC1, 0x03};
CipherSuite TLS_GOSTR341112_256_WITH_MAGMA_MGM_L = {0xC1, 0x04};
CipherSuite TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_S = {0xC1, 0x05};
CipherSuite TLS_GOSTR341112_256_WITH_MAGMA_MGM_S = {0xC1, 0x06};
```

Each cipher suite specifies a pair of the record protection algorithm (see Section 4.1) and the hash algorithm (Section 4.2).

4.1. Record Protection Algorithm

In accordance with Section 5.2 of [RFC8446] the record protection algorithm translates a TLSPlaintext structure into a TLSCiphertext structure. If TLS13_GOST cipher suite is negotiated, the encrypted_record field of the TLSCiphertext structure MUST be set to the AEADEncrypted value computed as follows:

```
AEADEncrypted = AEAD-Encrypt(sender_record_write_key, nonce,
associated_data, plaintext),
```

where

- the AEAD-Encrypt function is defined in Section 4.1.1;
- the sender_record_write_key is derived from the sender_write_key (see Section 7.3 of [RFC8446]) using TLSTREE function defined in Section 4.1.2 and sequence number seqnum as follows:

```
sender_record_write_key = TLSTREE(sender_write_key, seqnum);
```

- the nonce value is derived from the record sequence number seqnum and the sender_write_iv value (see Section 7.3 of [RFC8446]) in accordance with Section 5.3 of [RFC8446];

- the associated_data value is the record header that is generated in accordance with Section 5.2 of [RFC8446];

- the plaintext value is the TLSInnerPlaintext structure encoded in accordance with Section 5.2 of [RFC8446].

Note1: The AEAD-Encrypt function is exactly the same as the AEAD-Encrypt function defined in [RFC8446] except the key (the first argument) is calculated from the sender_write_key and sequence number seqnum for each message separately to support external re-keying approach according to [RFC8645].

Note2: The record sequence number is the value in the range 0-SNMAX, where the SNMAX value is defined in Section 4.1.3. The SNMAX parameter is specified by the particular TLS13_GOST cipher suite to limit the amount of data that can be encrypted under the same traffic key material (sender_write_key, sender_write_iv).

The record deprotection algorithm reverses the process of the record protection. In order to decrypt and verify the protected record with sequence number seqnum the algorithm takes as input the sender_record_write_key is derived from the sender_write_key, nonce, associated_data and the AEADEncrypted value and outputs the res value which is either the plaintext or an error indicating that the decryption failed. If TLS13_GOST cipher suite is negotiated, the res value MUST be computed as follows:

```
res = AEAD-Decrypt(sender_record_write_key, nonce,
associated_data, AEADEncrypted),
```

where the AEAD-Decrypt function is defined in Section 4.1.1.

Note: The AEAD-Decrypt function is exactly the same as the AEAD-Decrypt function defined in [RFC8446] except the key (the first argument) is calculated from the sender_write_key and sequence number seqnum for each message separately to support external re-keying approach according to [RFC8645].

4.1.1. AEAD Algorithm

The AEAD-Encrypt and AEAD-Decrypt functions are defined as follows.

```
+--------------------------------------------------------------------+
|  AEAD-Encrypt(K, nonce, A, P)                                      |
|--------------------------------------------------------------------|
|  Input:                                                            |
|  - encryption key K in B_k,                                        |
|  - unique vector nonce in B_IVlen,                                 |
|  - associated authenticated data A in B_r, r >= 0,                 |
|  - plaintext P in B_t, t >= 0.                                     |
|  Output:                                                           |
|  - ciphertext C in B_{|P|},                                        |
|  - authentication tag T in B_S.                                    |
|--------------------------------------------------------------------|
|  1. MGMnonce = nonce[1..1] & 0x7f | nonce[2..IVlen];                |
|  2. (MGMnonce, A, C, T) = MGM-Encrypt(K, MGMnonce, A, P);          |
|  3. Return C | T.                                                  |
+--------------------------------------------------------------------+

+--------------------------------------------------------------------+
|  AEAD-Decrypt(K, nonce, A, C | T)                                  |
|--------------------------------------------------------------------|
|  Input:                                                            |
|  - encryption key K in B_k,                                        |
|  - unique vector nonce in B_IVlen,                                 |
|  - associated authenticated data A in B_r, r >= 0,                 |
|  - ciphertext C in B_t, t >= 0,                                    |
|  - authentication tag T in B_S.                                    |
|  Output:                                                           |
|  - plaintext P in B_{|C|} or FAIL.                                 |
|--------------------------------------------------------------------|
|  1. MGMnonce = nonce[1..1] & 0x7f | nonce[2..IVlen];                |
|  2. res' = MGM-Decrypt(K, MGMnonce, A, C, T);                      |
|  3. IF res' = FAIL then return FAIL;                               |
|  4. IF res' = (A, P) then return P.                                |
+--------------------------------------------------------------------+
```

where

- MGM-Encrypt and MGM-Decrypt functions are defined in [DraftMGM]. The size of the authentication tag T is equal to n bytes (S = n). The size of the nonce parameter is equal to n bytes (IVlen = n).

The cipher suites TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_L and TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_S MUST use Kuznyechik [RFC7801] as a base block cipher for the AEAD algorithm. The block length n is 16 bytes (n = 16) and the key length k is 32 bytes (k = 32).

The cipher suites TLS_GOSTR341112_256_WITH_MAGMA_MGM_L and TLS_GOSTR341112_256_WITH_MAGMA_MGM_S MUST use Magma [GOST3412-2015] as a base block cipher for the AEAD algorithm. The block length n is 8 bytes (n = 8) and the key length k is 32 bytes (k = 32).

4.1.2. TLSTREE

The TLS13_GOST cipher suites use the TLSTREE function for the external re-keying approach (see [RFC8645]). The TLSTREE function is defined as follows:

```
TLSTREE(K_root, i) = KDF_3(KDF_2(KDF_1(K_root, STR_8(i & C_1)),
STR_8(i & C_2)), STR_8(i & C_3)),
```

where

- K_root in B_32;
- i in {0, 1, ... , 2^64 - 1};
- KDF_j(K, D), j = 1, 2, 3, is the key derivation function defined as follows:

  ```
  KDF_1(K, D) = KDF_GOSTR3411_2012_256(K, "level1", D),
  KDF_2(K, D) = KDF_GOSTR3411_2012_256(K, "level2", D),
  KDF_3(K, D) = KDF_GOSTR3411_2012_256(K, "level3", D),
  ```

  where the KDF_GOSTR3411_2012_256 function is defined in [RFC7836], K in B_32, D in B_8.

- C_1, C_2, C_3 are constants defined by the particular cipher suite as follows:

CipherSuites	C_1, C_2, C_3
TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_L	C_1=0xf800000000000000 C_2=0xfffffff000000000 C_3=0xfffffffffffe000
TLS_GOSTR341112_256_WITH_MAGMA_MGM_L	C_1=0xffe0000000000000 C_2=0xffffffffc0000000 C_3=0xffffffffffffff80
TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_S	C_1=0xfffffffe0000000 C_2=0xfffffffffff0000 C_3=0xfffffffffffffff8
TLS_GOSTR341112_256_WITH_MAGMA_MGM_S	C_1=0xfffffffffc000000 C_2=0xfffffffffffe000 C_3=0xffffffffffffffff

Table 1

4.1.3. SNMAX parameter

The SNMAX parameter is the maximum number of records encrypted under the same traffic key material (sender_write_key and sender_write_iv) and is defined by the particular cipher suite as follows:

CipherSuites	SNMAX
TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_L	SNMAX = 2^64 - 1
TLS_GOSTR341112_256_WITH_MAGMA_MGM_L	SNMAX = 2^64 - 1
TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_S	SNMAX = 2^42 - 1
TLS_GOSTR341112_256_WITH_MAGMA_MGM_S	SNMAX = 2^39 - 1

Table 2

4.2. Hash Algorithm

The Hash algorithm is used for key derivation process (see Section 7.1 of [RFC8446]), Finished message calculation (see Section 4.4.4 of [RFC8446]), Transcript-Hash function computation

(see Section 4.4.1 of [RFC8446]), PSK binder value calculation (see Section 4.2.11.2 of [RFC8446]), external re-keying approach (see Section 4.1.2) and other purposes.

In case of negotiating a TLS13_GOST cipher suite the Hash algorithm MUST be the GOST R 34.11-2012 [RFC6986] hash algorithm with 32-byte (256-bit) hash value.

5. Signature Scheme Definition

The signature scheme value is used to indicate a single signature algorithm and a curve that can be used in digital signature (see Section 4.2.3 of [RFC8446]).

This section defines the following seven TLS13_GOST signature schemes that can be used to support Russian cryptographic algorithms:

```
enum {
    gostr34102012_256a(0x0709),
    gostr34102012_256b(0x070A),
    gostr34102012_256c(0x070B),
    gostr34102012_256d(0x070C),
    gostr34102012_512a(0x070D),
    gostr34102012_512b(0x070E),
    gostr34102012_512c(0x070F)
} SignatureScheme;
```

If TLS13_GOST cipher suite is negotiated and authentication via certificates is required one of the TLS13_GOST signature schemes listed above SHOULD be used.

Each signature scheme specifies a pair of the signature algorithm (see Section 5.1) and the elliptic curve (see Section 5.2).

5.1. Signature Algorithm

Signature algorithms corresponding to the TLS13_GOST signature schemes are defined as follows:

SignatureScheme Value	Signature Algorithm	References
gostr34102012_256a	GOST R 34.10-2012 , 32-byte key length	RFC 7091
gostr34102012_256b	GOST R 34.10-2012 , 32-byte key length	RFC 7091
gostr34102012_256c	GOST R 34.10-2012 , 32-byte key length	RFC 7091
gostr34102012_256d	GOST R 34.10-2012 , 32-byte key length	RFC 7091
gostr34102012_512a	GOST R 34.10-2012 , 64-byte key length	RFC 7091
gostr34102012_512b	GOST R 34.10-2012 , 64-byte key length	RFC 7091
gostr34102012_512c	GOST R 34.10-2012 , 64-byte key length	RFC 7091

Table 3

5.2. Elliptic Curve

Elliptic curves corresponding to the TLS13_GOST signature schemes are defined as follows:

SignatureScheme Value	Curve Identifier Value	References
gostr34102012_256a	id-tc26-gost-3410-2012-256-paramSetA	RFC 7836
gostr34102012_256b	id-GostR3410-2001-CryptoPro-A-ParamSet	RFC 4357
gostr34102012_256c	id-GostR3410-2001-CryptoPro-B-ParamSet	RFC 4357
gostr34102012_256d	id-GostR3410-2001-CryptoPro-C-ParamSet	RFC 4357
gostr34102012_512a	id-tc26-gost-3410-12-512-paramSetA	RFC 7836
gostr34102012_512b	id-tc26-gost-3410-12-512-paramSetB	RFC 7836
gostr34102012_512c	id-tc26-gost-3410-2012-512-paramSetC	RFC 7836

Table 4

5.3. SIGN function

If TLS13_GOST signature scheme is used, the signature value in CertificateVerify message (see Section 6.3.4) MUST be calculated using the SIGN function defined as follows:

```
+-----------------------------------------------------+
|  SIGN(d_sign, M)                                    |
|-----------------------------------------------------|
|  Input:                                             |
|  - the sign key d_sign: 0 < d_sign < q;             |
|  - the byte string M in B*.                         |
|  Output:                                            |
|  - signature value sgn in B_{2*l}.                  |
|-----------------------------------------------------|
|  1. (r, s) = SIGNGOST(d_sign, M)                    |
|  2. Return str_l(r) | str_l(s).                     |
|-----------------------------------------------------+
```

where

- q is the subgroup order of group of points of the elliptic curve;
- l is defined as follows:
  - l = 32 for gostr34102012_256a, gostr34102012_256b, gostr34102012_256c and gostr34102012_256d signature schemes;
  - l = 64 for gostr34102012_512a, gostr34102012_512b and gostr34102012_512c signature schemes;
- SIGNGOST is an algorithm which takes as an input private key d_sign and message M and returns a pair of integers (r, s) calculated during signature generation process in accordance with the GOST R 34.10-2012 signature algorithm (see Section 6.1 of [RFC7091]).

Note: The signature value sgn is the concatenation of two strings that are byte representations of r and s values in the little-endian format.

6. Key Exchange and Authentication

Key exchange and authentication process in case of using TLS13_GOST cipher suites is defined in Section 6.1, Section 6.2 and Section 6.3.

6.1. Key Exchange

TLS13_GOST cipher suites support three basic key exchange modes which are defined in [RFC8446]: ECDHE, PSK-only and PSK-with-ECDHE.

Note: In accordance with [RFC8446] TLS 1.3 also supports key exchange modes based on Diffie-Hellman protocol over finite fields. However, TLS13_GOST cipher suites MUST use only modes based on Diffie-Hellman protocol over elliptic curves.

In accordance with [RFC8446] PSKs can be divided into two types:

- internal PSKs which can be established during the previous connection;
- external PSKs which can be established out of band.

If TLS13_GOST cipher suite is negotiated, PSK-only key exchange mode SHOULD be established only via the internal PSKs, and external PSKs SHOULD be used only in PSK-with-ECDHE mode (see more in Section 9).

If TLS13_GOST cipher suite is negotiated and ECDHE or PSK-with-ECDHE key exchange mode is used the ECDHE shared secret value should be calculated in accordance with Section 6.1.1 on the basis of one of the elliptic curves defined in Section 6.1.2.

6.1.1. ECDHE Shared Secret Calculation

If TLS13_GOST cipher suite is negotiated, ECDHE shared secret value should be calculated in accordance with Section 6.1.1.1 and Section 6.1.1.2. The public ephemeral keys used to obtain ECDHE shared secret value should be represented in format described in Section 6.1.1.3.

6.1.1.1. ECDHE Shared Secret Calculation on Client Side

The client calculates ECDHE shared secret value in accordance with the following steps:

1. Chooses from all supported curves E_1, ..., E_R the set of curves E_{i_1}, ..., E_{i_r}, 1 <= i_1 <= i_r <= R, where

- r >= 1 in the case of the first ClientHello message;
- r = 1 in the case of responding to HelloRetryRequest message, E_{i_1} corresponds to the curve indicated in the "key_share" extension in the HelloRetryRequest message.

2. Generates ephemeral key pairs (d_C^{i_1}, Q_C^{i_1}), ..., (d_C^{i_r}, Q_C^{i_r}) corresponding to the curves E_{i_1}, ..., E_{i_r}, where for each i in {i_1, ..., i_r}:

- d_C^i is chosen from {1, ..., q_i - 1} at random;

- Q_C^i = d_C^i * P_i.

3. Sends ClientHello message specified in accordance with Section 4.1.2 of [RFC8446] and Section 6.3.1, which contains:

- "key_share" extension with public ephemeral keys Q_C^{i_1}, ..., Q_C^{i_r} generated in accordance with Section 4.2.8 of [RFC8446];

- "supported_groups" extension with supported curves E_1, ..., E_R generated in accordance with Section 4.2.7 of [RFC8446].

Note: Client MAY send an empty "key_share" extension in the first ClientHello in order to request group selection from the server in the HelloRetryRequest message and to generate ephemeral key for the selected group only. The ECDHE value may be calculated without sending HelloRetryRequest, if the "key_share" extension in the first ClientHello message consists the value corresponded to the curve that will be selected by the server.

4. In case of receiving HelloRetryRequest message client MUST return to step 1 and correct parameters in accordance with Section 4.1.2 of [RFC8446]. In case of receiving ServerHello message client proceeds to the next step. In other cases client MUST terminate the connection with "unexpected_message" alert.

5. Extracts curve E_res and ephemeral key Q_S^res, res in {1, ..., R}, from ServerHello message and checks whether the Q_S^res belongs to E_res. If this check fails, the client MUST abort the handshake with "handshake_failure" alert.

6. Generates Q^ECDHE:

$$Q^{ECDHE} = (X^{ECDHE}, Y^{ECDHE}) = (h_{res} * d_C^{res}) * Q_S^{res}.$$

7. Client MUST check whether the computed shared secret Q^ECDHE is not equal to the zero point O_res. If this check fails, the client MUST abort the handshake with "handshake_failure" alert.

8. Shared secret value ECDHE is the byte representation of the coordinate X^ECDHE of point Q^ECDHE in the little-endian format:

$$ECDHE = str_{coordinate\_length}(X^{ECDHE}),$$

where the coordinate_length value is defined by the particular elliptic curve (see Section 6.1.2).

6.1.1.2. ECDHE Shared Secret Calculation on Server Side

Upon receiving the ClientHello message, the server calculates ECDHE shared secret value in accordance with the following steps:

1. Chooses the curve E_res, res in {1, ..., R}, from the list of the curves E_1, ..., E_R indicated in "supported_groups" extension in ClientHello message and the corresponding public ephemeral key value Q_C^res from the list Q_C^{i_1}, ..., Q_C^{i_r}, 1 <= i_1 <= i_r <= R, indicated in "key_share" extension. If no corresponding public ephemeral key value is found (res in {1, ..., R}\{i_1, ..., i_r}), server MUST send HelloRetryRequest message with "key_share" extension indicating the selected elliptic curve E_res and wait for the new ClientHello message.

2. Checks whether Q_C^res belongs to E_res. If this check fails, the server MUST abort the handshake with "handshake_failure" alert.

3. Generates ephemeral key pair (d_S^res, Q_S^res) corresponding to E_res:

o d_S^res is chosen from {1, ..., q_res - 1} at random;

o Q_S^res = d_S^res * P_res.

4. Sends ServerHello message generated in accordance with Section 4.1.3 of [RFC8446] and Section 6.3.1 with "key_share" extension which contains public ephemeral key value Q_S^res corresponding to E_res.

5. Generates Q^ECDHE:

   Q^ECDHE = (X^ECDHE, Y^ECDHE) = (h_res * d_S^res) * Q_C^res.

6. Server MUST check whether the computed shared secret Q^ECDHE is not equal to the zero point O_res. If this check fails, the server MUST abort the handshake with "handshake_failure" alert.

7. Shared secret value ECDHE is the byte representation of the coordinate X^ECDHE of point Q^ECDHE in the little-endian format:

   ECDHE = str_{coordinate_length}(X^ECDHE),

where the coordinate_length value is defined by the particular elliptic curve (see Section 6.1.2).

6.1.1.3. Public ephemeral key representation

This section defines the representation format of the public ephemeral keys generated during ECDHE shared secret calculation (see Section 6.1.1.1 and Section 6.1.1.2).

If TLS13_GOST cipher suite is negotiated and ECDHE or PSK-with-ECDHE key exchange mode is used, the public ephemeral key Q indicated in the KeyShareEntry.key_exchange field MUST contain the data defined by the following structure:

```
struct {
    opaque X[coordinate_length];
    opaque Y[coordinate_length];
} PlainPointRepresentation;
```

where X and Y, respectively, contain the byte representations of the x and the y values of point Q (Q = (x, y)) in the little-endian format and are specified as follows:

X = str_{coordinate_length}(x);

Y = str_{coordinate_length}(y).

The coordinate_length value is defined by the particular elliptic curve (see Section 6.1.2).

6.1.2. Values for the TLS Supported Groups Registry

The "supported_groups" extension is used to indicate the set of the elliptic curves supported by an endpoint and is defined in Section 4.2.7 [RFC8446]. This extension is always contained in ClientHello message and optionally in EncryptedExtensions message.

This section defines the following seven elliptic curves that can be used to support Russian cryptographic algorithms:

```
enum {
    GC256A(0x22), GC256B(0x23), GC256C(0x24), GC256D(0x25),
    GC512A(0x26), GC512B(0x27), GC512C(0x28)
} NamedGroup;
```

If TLS13_GOST cipher suite is negotiated and ECDHE or PSK-with-ECDHE key exchange mode is established, one of the elliptic curves listed above SHOULD be used.

Each curve corresponds to the particular identifier and specifies the value of coordinate_length parameter (see "cl" column) as follows:

Description	Curve Identifier Value	cl	Reference
GC256A	id-tc26-gost-3410-2012-256-paramSetA	32	RFC 7836
GC256B	id-GostR3410-2001-CryptoPro-A-ParamSet	32	RFC 4357
GC256C	id-GostR3410-2001-CryptoPro-B-ParamSet	32	RFC 4357
GC256D	id-GostR3410-2001-CryptoPro-C-ParamSet	32	RFC 4357
GC512A	id-tc26-gost-3410-12-512-paramSetA	64	RFC 7836
GC512B	id-tc26-gost-3410-12-512-paramSetB	64	RFC 7836
GC512C	id-tc26-gost-3410-2012-512-paramSetC	64	RFC 7836

Table 5

Note: The identifier values and the corresponding elliptic curves are the same as in [DraftGostTLS12].

6.2. Authentication

In accordance with [RFC8446] authentication can happen via signature with certificate or via symmetric pre-shared key (PSK). The server side of the channel is always authenticated; the client side is optionally authenticated.

PSK-based authentication happens as a side effect of key exchange. If TLS13_GOST cipher suite is negotiated, external PSKs SHOULD be combined only with the mutual authentication (see more in Section 9).

Certificate-based authentication happens via Authentication messages and optional CertificateRequest message (sent if client authentication is required). In case of negotiating TLS13_GOST cipher suites the signature schemes used for certificate-based authentication are defined in Section 5 and the Authentication

messages are specified in Section 6.3.3 and Section 6.3.4. The CertificateRequest message is specified in Section 6.3.2.

6.3. Handshake Messages

The TLS13_GOST cipher suites specify the ClientHello, ServerHello, CertificateRequest, Certificate and CertificateVerify handshake messages that are described in further detail below.

6.3.1. Hello Messages

The ClientHello message is sent when a client first connects to a server or responds to a HelloRetryRequest message and is specified in accordance with [RFC8446] as follows.

```
struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id<0..32>;
    CipherSuite cipher_suites<2..2^16-2>;
    opaque legacy_compression_methods<1..2^8-1>;
    Extension extensions<8..2^16-1>;
} ClientHello;
```

In order to negotiate a TLS13_GOST cipher suite, the ClientHello message MUST meet the following requirements.

- The ClientHello.cipher_suites field MUST contain the values defined in Section 4.
- If server authentication via a certificate is required, the extension_data field of the "signature_algorithms" extension MUST contain the values defined in Section 5, which correspond to the GOST R 34.10-2012 signature algorithm.
- If server authentication via a certificate is required and the client uses optional "signature_algorithms_cert" extension, the extension_data field of this extension SHOULD contain the values defined in Section 5, which correspond to the GOST R 34.10-2012 signature algorithm.
- If client wants to establish TLS 1.3 connection using ECDHE shared secret value, the extension_data field of the "supported_groups" extension MUST contain the elliptic curve identifiers defined in Section 6.1.2.

The ServerHello message is sent by the server in response to a ClientHello message to negotiate an acceptable set of handshake parameters based on the ClientHello and is specified in accordance with [RFC8446] as follows.

```
struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id_echo<0..32>;
    CipherSuite cipher_suite;
    uint8 legacy_compression_method = 0;
    Extension extensions<6..2^16-1>;
} ServerHello;
```

In case of negotiating a TLS13_GOST cipher suite, the ServerHello message MUST meet the following requirements.

- The ServerHello.cipher_suite field MUST contain one of the values defined in Section 4.
- If server decides to establish TLS 1.3 connection using ECDHE shared secret value, the extension_data field of the "key_share" extension MUST contain the elliptic curve identifier and the public ephemeral key that satisfy the following conditions.
  - The elliptic curve identifier corresponds to a value that was provided in the "supported_groups" and the "key_share" extensions in the ClientHello message.
  - The elliptic curve identifier is one of the values defined in Section 6.1.2.
  - The public ephemeral key corresponds to the elliptic curve specified by the KeyShareEntry.group identifier.

6.3.2. CertificateRequest

This message is sent when server requests client authentication via a certificate and is specified in accordance with [RFC8446] as follows.

```
struct {
    opaque certificate_request_context<0..2^8-1>;
    Extension extensions<2..2^16-1>;
} CertificateRequest;
```

If TLS13_GOST cipher suite is negotiated, the CertificateRequest message MUST meet the following requirements.

- The extension_data field of the "signature_algorithms" extension MUST contain only the values defined in Section 5.

- If server uses optional "signature_algorithms_cert" extension, the extension_data field of this extension SHOULD contain only the values defined in Section 5.

6.3.3. Certificate

This message is sent to convey the endpoint's certificate chain to the peer and is specified in accordance with [RFC8446] as follows.

```
struct {
    opaque certificate_request_context<0..2^8-1>;
    CertificateEntry certificate_list<0..2^24-1>;
} Certificate;
```

If TLS13_GOST cipher suite is negotiated, the Certificate message MUST meet the following requirements.

- Each endpoint's certificate provided to the peer MUST be signed using the algorithm which corresponds to a signature scheme indicated by the peer in its "signature_algoritms_cert" extension, if present (or in the "signature_algorithms" extension, otherwise).

- The signature algorithm used for signing certificates SHOULD correspond to the one of the signature schemes defined in Section 5.

6.3.4. CertificateVerify

This message is sent to provide explicit proof that an endpoint possesses the private key corresponding to the public key indicated in its certificate and is specified in accordance with [RFC8446] as follows.

```
struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;
```

If TLS13_GOST cipher suite is negotiated, the CertificateVerify message MUST meet the following requirements.

- The CertificateVerify.algorithm field MUST contain the signature scheme identifier which corresponds to the value indicated in the peer's "signature_algorithms" extension and which is one of the values defined in Section 5.

- The CertificateVerify.signature field contains the sgn value, which is computed as follows:

  sgn = SIGN(d_sign, M),

- where

  * the SIGN function is defined in Section 5,

  * d_sign is the sender long-term private key that corresponds to the sender long-term public key Q_sign from the sender's certificate,

  * the message M is defined in accordance with Section 4.4.3 of [RFC8446].

7. IANA Considerations

IANA has added numbers {0xC1, 0x03}, {0xC1, 0x04}, {0xC1, 0x05} and {0xC1, 0x06} with the names TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_L, TLS_GOSTR341112_256_WITH_MAGMA_MGM_L, TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_S, TLS_GOSTR341112_256_WITH_MAGMA_MGM_S to the "TLS Cipher Suites" registry with this document as reference, as shown below.

Value	Description	DTLS-OK	Reference
0xC1, 0x03	TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_L	N	this RFC
0xC1, 0x04	TLS_GOSTR341112_256_WITH_MAGMA_MGM_L	N	this RFC
0xC1, 0x05	TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_S	N	this RFC
0xC1, 0x06	TLS_GOSTR341112_256_WITH_MAGMA_MGM_S	N	this RFC

Table 6

IANA has added numbers 0x0709, 0x070A, 0x070B, 0x070C, 0x070D, 0x070E and 0x070F with the names gostr34102012_256a, gostr34102012_256b, gostr34102012_256c, gostr34102012_256d, gostr34102012_512a, gostr34102012_512b, gostr34102012_512c to the "TLS SignatureScheme" registry, as shown below.

Value	Description	DTLS-OK	Reference
0x0709	gostr34102012_256a	N	this RFC
0x070A	gostr34102012_256b	N	this RFC
0x070B	gostr34102012_256c	N	this RFC
0x070C	gostr34102012_256d	N	this RFC
0x070D	gostr34102012_512a	N	this RFC
0x070E	gostr34102012_512b	N	this RFC
0x070F	gostr34102012_512c	N	this RFC

Table 7

8. Historical considerations

Due to historical reasons in addition to the curve identifier values listed in Table 5 there exist some additional identifier values that correspond to the signature schemes as follows.

Description	Curve Identifier Value
gostr34102012_256b	id-GostR3410_2001-CryptoPro-XchA-ParamSet
	id-tc26-gost-3410-2012-256-paramSetB
gostr34102012_256c	id-tc26-gost-3410-2012-256-paramSetC
gostr34102012_256d	id-GostR3410-2001-CryptoPro-XchB-ParamSet
	id-tc26-gost-3410-2012-256-paramSetD

Table 8

Client should be prepared to handle any of them correctly if corresponding signature scheme is included in the "signature_algorithms" or "signature_algorithms_cert" extensions.

9. Security Considerations

In order to create an effective implementation client and server SHOULD follow the rules below.

1. While using TLSTREE algorithm function KDF_j, j = 1, 2, 3, SHOULD be invoked only if the record sequence number seqnum reaches such a value that

   seqnum & C_j != (seqnum - 1) & C_j.

Otherwise the previous value should be used.

2. For each pre-shared key value PSK the binder_key value should be computed only once within all connections where ClientHello message contains a "pre_shared_key" extension indicating this PSK value.

In order to ensure the secure TLS 1.3 connection client and server SHOULD fulfil the following requirements.

1. An internal PSK value is NOT RECOMMENDED to be used to establish more than one TLS 1.3 connection.

2. 0-RTT data SHOULD NOT be sent during TLS 1.3 connection. The reasons for this restriction are that the 0-RTT data is not forward secret and is not resistant to replay attacks (see more in Section 2.3 of [RFC8446]).

3. If client authentication is required, server SHOULD NOT send Application Data, NewSessionTicket and KeyUpdate messages prior to receiving the client's Authentication messages since any data sent at that point is being sent to an unauthenticated peer.

4. External PSKs SHOULD be used only in PSK-with-ECDHE mode. In case of using external PSK in PSK-only mode the attack described in [Selfie] is possible which leads to the situation when client establishes connection to itself. One of the mitigations proposed in [Selfie] is to use certificates, however, in that case, an impersonation attack as in [AASS19] occurs. If the connections are established with additional usage of key_share extension (in PSK-with-ECDHE mode), then the adversary which just echoes messages cannot reveal the traffic key material (as long as the used group is secure).

5. In case of using external PSK, the mutual authentication MUST be provided by the external PSK distribution mechanism between the endpoints which guarantees that the derived external PSK is unknown to anyone but the endpoints. In addition, the endpoint roles (i.e. client and server) MUST be fixed during this mechanism and each role can match only to one endpoint during the whole external PSK lifetime.

10. References

10.1. Normative References

[DraftGostTLS12]
Smyshlyaev, S., Belyavsky, D., and M. Saarinen, "GOST Cipher Suites for Transport Layer Security (TLS) Protocol Version 1.2", 2019, <https://tools.ietf.org/html/draft-smyshlyaev-tls12-gost-suites-08>.

[DraftMGM]
Smyshlyaev, S., Nozdrunov, V., Shishkin, V., and E. Smyshlyaeva, "Multilinear Galois Mode (MGM)", 2019, <https://tools.ietf.org/html/draft-smyshlyaev-mgm-17>.

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

[RFC6986] Dolmatov, V., Ed. and A. Degtyarev, "GOST R 34.11-2012: Hash Function", RFC 6986, DOI 10.17487/RFC6986, August 2013, <https://www.rfc-editor.org/info/rfc6986>.

[RFC7091] Dolmatov, V., Ed. and A. Degtyarev, "GOST R 34.10-2012: Digital Signature Algorithm", RFC 7091, DOI 10.17487/RFC7091, December 2013, <https://www.rfc-editor.org/info/rfc7091>.

[RFC7801] Dolmatov, V., Ed., "GOST R 34.12-2015: Block Cipher "Kuznyechik"", RFC 7801, DOI 10.17487/RFC7801, March 2016, <https://www.rfc-editor.org/info/rfc7801>.

[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>.

[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>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC8645] Smyshlyaev, S., Ed., "Re-keying Mechanisms for Symmetric Keys", RFC 8645, DOI 10.17487/RFC8645, August 2019, <https://www.rfc-editor.org/info/rfc8645>.

10.2. Informative References

[AASS19] Akhmetzyanova, L., Alekseev, E., Smyshlyaeva, E., and A. Sokolov, "Continuing to reflect on TLS 1.3 with external PSK", Cryptology ePrint Archive Report 2019/421, April 2019, <https://eprint.iacr.org/2019/421.pdf>.

[GOST3410-2012] Federal Agency on Technical Regulating and Metrology, "Information technology. Cryptographic data security. Signature and verification processes of [electronic] digital signature", GOST R 34.10-2012, 2012.

[GOST3411-2012]
Federal Agency on Technical Regulating and Metrology, "Information technology. Cryptographic Data Security. Hashing function", GOST R 34.11-2012, 2012.

[GOST3412-2015]
Federal Agency on Technical Regulating and Metrology, "Information technology. Cryptographic data security. Block ciphers", GOST R 34.12-2015, 2015.

[Selfie] Drucker, N. and S. Gueron, "Selfie: reflections on TLS 1.3 with PSK", Cryptology ePrint Archive Report 2019/347, April 2019, <https://eprint.iacr.org/2019/347.pdf>.

Appendix A. Test Examples

TODO

Appendix B. Contributors

- Lilia Akhmetzyanova
  CryptoPro
  lah@cryptopro.ru

- Alexandr Sokolov
  CryptoPro
  sokolov@cryptopro.ru

- Vasily Nikolaev
  CryptoPro
  nikolaev@cryptopro.ru

- Lidia Nikiforova
  CryptoPro
  nikiforova@cryptopro.ru

Appendix C. Acknowledgments

Authors' Addresses

Stanislav Smyshlyaev (editor)
CryptoPro
18, Suschevsky val
Moscow 127018
Russian Federation

Phone: +7 (495) 995-48-20
Email: svs@cryptopro.ru

Evgeny Alekseev
CryptoPro
18, Suschevsky val
Moscow 127018
Russian Federation

Email: alekseev@cryptopro.ru

Ekaterina Griboedova
CryptoPro
18, Suschevsky val
Moscow 127018
Russian Federation

Email: griboedova.e.s@gmail.com

Alexandra Babueva
CryptoPro
18, Suschevsky val
Moscow 127018
Russian Federation

Email: babueva@cryptopro.ru