Abstract

This document specifies a set of cipher suites for the Transport Layer Security (TLS) protocol Version 1.3 to support the Russian cryptographic standard 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 May 1, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of
1. Introduction

This document specifies four new cipher suites for the Transport Layer Security (TLS) Protocol Version 1.3 [RFC8446] to support the set of Russian cryptographic standard algorithms called GOST algorithms (see Section 4). These cipher suites use the hash algorithm GOST R 34.11-2012 [GOST3411-2012] (the English version can be found in [RFC6986]), the AEAD algorithm MGM [DraftMGM] and the block ciphers GOST R 34.12-2015 [GOST3412-2015] (the English version
can be found in [RFC7801]). The GOST cipher suites have the following values:

- TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_L = {TBD1};
- TLS_GOSTR341112_256_WITH_MAGMA_MGM_L = {TBD2};
- TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_S = {TBD3};
- TLS_GOSTR341112_256_WITH_MAGMA_MGM_S = {TBD4}.

These cipher suites have different key lifetime value (see [RFC8645]), so they are divided into two types: the _S (strong) cipher suites and the _L (light) cipher suites (see Section 4.1.1.3, Section 4.1.1.2).

This document specifies seven new signature schemes (see Section 5) to be used with GOST cipher suites. These signature schemes use the signature algorithm GOST R 34.10-2012 [GOST3410-2012] (the English version can be found in [RFC7091]) and have the following values:

- gostr34102012_256a = {TBD5};
- gostr34102012_256b = {TBD6};
- gostr34102012_256c = {TBD7};
- gostr34102012_256d = {TBD8};
- gostr34102012_512a = {TBD9};
- gostr34102012_512b = {TBD10};
- gostr34102012_512c = {TBD11}.

Additionally this document specifies the key exchange and authentication process in case of negotiating 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", "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:
B\_t\) the set of byte strings of length \(t\), \(t \geq 0\), 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 \(A = (a\_1, a\_2, \ldots, a\_t)\), where 
\(a\_1, a\_2, \ldots, a\_t\) are in \([0, \ldots, 255]\);

\(B^*\) the set of all byte strings of a finite length (hereinafter 
referred to as strings), including the empty string;

\(A[i..j]\) the string \(A[i..j] = (a\_i, a\_(i+1), \ldots, a\_j)\) in \(B\_{j-i+1}\)
where \(A = (a\_1, \ldots, a\_t)\) in \(B\_t\) and \(1 \leq i \leq j \leq t\);

\(|A|\) the byte length of the byte string \(A\);

\(A \mid C\) 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 \(j\);

\(\text{STR}\_t\) the transformation that maps an integer 
\(i = 256^{t-1} \times i\_1 + \ldots + 256 \times i\_{t-1} + i\_t\) into the byte string 
\(\text{STR}\_t(i) = (i\_1, \ldots, i\_t)\) in \(B\_t\) (the interpretation of the integer as 
a byte string in big-endian format);

\(\text{str}\_t\) the transformation that maps an integer 
\(i = 256^{t-1} \times i\_t + \ldots + 256 \times i\_2 + i\_1\) into the byte string 
\(\text{str}\_t(i) = (i\_1, \ldots, i\_t)\) in \(B\_t\) (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;

\(E\_i\) elliptic curve indicated by client in "supported_group" 
extension;

\(Q\_c\) the public key stored in the client’s certificate;

\(d\_c\) the private key that corresponds to the \(Q\_c\) key;

\(Q\_s\) the public key stored in the server’s certificate;

\(d\_s\) the private key that corresponds to the \(Q\_s\) key;

\(q\_s\) subgroup order of group of points of the elliptic curve that 
corresponds to \(Q\_s\);

\(P\_s\) the point of order \(q\_s\) that belongs to the same curve as \(Q\_s\);
4. Cipher Suite Definition

This document defines the following new values for the "Cipher Suites" registry, that can be used in the ClientHello.cipher_suites and ServerHello.cipher_suites fields for the particular cipher suite:

CipherSuite TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_L = TBD1;
CipherSuite TLS_GOSTR341112_256_WITH_MAGMA_MGM_L = TBD2;
CipherSuite TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_S = TBD3;
CipherSuite TLS_GOSTR341112_256_WITH_MAGMA_MGM_S = TBD4;

Each cipher suite defines the pair of the AEAD algorithm and the hash algorithm. AEAD algorithm is used for the payload protection (see Section 4.1 for more details) and hash algorithm is used for key derivation and handshake message authentication.

4.1. Payload Protection

All of the cipher suites described in this document use the block cipher in Authenticated Encryption with Associated Data (AEAD) mode (see Section 4.1.1) to protect records. The TLSCiphertext structure is specified in accordance with [RFC8446] as follows.

```c
struct {
    ContentType opaque_type = application_data; /* 23 */
    ProtocolVersion legacy_record_version = 0x0303; /*TLSv1.2*/
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

The encrypted_record field of TLSCiphertext is set to AEADEncrypted, where AEADEncrypted is computed as follows:

```c
AEADEncrypted = AEAD-Encrypt(sender_write_key, nonce, additional_data, plaintext),
```

where

- AEAD-Encrypt function is defined in Section 4.1.1;
- the traffic key material that consists of the sender_write_key (either the client_write_key or the server_write_key) and the sender_write_IV (either the client_write_IV or the server_write_IV).
server_write_IV) parameters is generated in accordance with Section 7.3 of [RFC8446];

- the nonce is derived from the sequence number and the sender_write_iv (see Section 5.3 of [RFC8446]);

- the plaintext input to the AEAD algorithm is the encoded TLSInnerPlaintext structure;

- the additional data input is the record header:

    \[
    \text{additional}_\text{data} = \text{TLSCiphertext}.\text{opaque}_\text{type} \oplus \text{TLSCiphertext}.\text{legacy}_\text{record}_\text{version} \oplus \text{TLSCiphertext}.\text{length};
    \]

In order to decrypt and verify, the cipher takes as input the key, nonce, additional data and the AEADEncrypted value. The output is either the plaintext or an error indicating that the decryption failed.

\[
\text{plaintext of encrypted_record} = \text{AEAD-Decrypt}(\text{sender_write_key}, \text{nonce}, \text{additional}_\text{data}, \text{AEADEncrypted})
\]

where AEAD-Decrypt function is defined in Section 4.1.1.

4.1.1. AEAD Algorithm

The GOST cipher suites use the block cipher in MGM authenticated encryption with associated data mode defined in [DraftMGM]. The base block cipher is defined in Section 4.1.1.1. The size \( S \) of the authentication field is \( n \) bytes. The IVlen parameter is \( n \) bytes.

The record key material is a key material that is generated from the traffic key material and is used to protect a record with the certain sequence number. All of the cipher suites described in this document use TLSTREE algorithm to derive record key material (Section 4.1.1.2).

For each record with sequence number \( \text{seqnum} \) 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 $B_{IVlen}$,
- additional authenticated data A in $B_s$, $s \geq 0$,
- plaintext P in $B_s$, $s \geq 0$.

Output:
- ciphertext C in $B_{|P|}$,
- authentication tag T in $B_S$.

1. $K^\text{seqnum} = \text{TLSTREE}(K, \text{seqnum})$
2. $\text{MGMnonce} = \text{nonce}[1..1] \& 0x7f | \text{nonce}[2..\text{IVlen}]
3. $(\text{MGMnonce}, A, C, T) = \text{MGM-Encrypt}(K^\text{seqnum}, \text{MGMnonce}, A, P)$
4. Return C | T.

AEAD-Decrypt(K, nonce, A, C | T)

Input:
- encryption key K in $B_k$,
- unique vector nonce in $B_{IVlen}$,
- additional authenticated data A in $B_s$, $s \geq 0$,
- ciphertext C in $B_s$, $s \geq 0$
- authentication tag T in $B_S$.

Output:
- plaintext P in $B_{|C|}$ or FAIL.

1. $K^\text{seqnum} = \text{TLSTREE}(K, \text{seqnum})$
2. $\text{MGMnonce} = \text{nonce}[1..1] \& 0x7f | \text{nonce}[2..\text{IVlen}]
3. $\text{res}' = \text{MGM-Decrypt}(K^\text{seqnum}, \text{MGMnonce}, A, C, T)$
4. IF $\text{res}' = \text{FAIL}$ then return FAIL; else return P.

MGM-Encrypt and MGM-Decrypt functions are defined in [DraftMGM], TLSTREE function is defined in Section 4.1.1.2. The maximum value of the sequence number seqnum during one TLS 1.3 connection is defined in Section 4.1.1.3.

4.1.1.1. Block Cipher

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 and the key length $k$ is 32 bytes.
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 and the key length k is 32 bytes.

4.1.1.2.  TLSTREE

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

\[ \text{TLSTREE}(K_{\text{root}}, i) = \text{KDF}_3(\text{KDF}_2(\text{KDF}_1(K_{\text{root}}, \text{STR}_8(i \& C_1)), \text{STR}_8(i \& C_2)), \text{STR}_8(i \& C_3)), \]

where

- \( K_{\text{root}} \) in \( B_{32} \);
- \( i \) in \( \{0, 1, \ldots, 2^{64} - 1\} \);
- \( \text{KDF}_j(K, D), j = 1, 2, 3, K \) in \( B_{32} \), \( D \) in \( B_{8} \), is the key derivation function based on the KDF_GOSTR3411_2012_256 function defined in [RFC7836]:

\[
\begin{align*}
\text{KDF}_1(K, D) &= \text{KDF}_\text{GOSTR3411_2012_256}(K, "level1", D); \\
\text{KDF}_2(K, D) &= \text{KDF}_\text{GOSTR3411_2012_256}(K, "level2", D); \\
\text{KDF}_3(K, D) &= \text{KDF}_\text{GOSTR3411_2012_256}(K, "level3", D).
\end{align*}
\]

- \( C_1, C_2, C_3 \) are constants defined by the particular cipher suite:
4.1.1.3. SNMAX parameter

The SNMAX parameter defines the maximal value of the sequence number seqnum during one TLS 1.3 connection and is defined as follows:

<table>
<thead>
<tr>
<th>CipherSuites</th>
<th>SNMAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_L</td>
<td>$2^{64} - 1$</td>
</tr>
<tr>
<td>TLS_GOSTR341112_256_WITH_MAGMA_MGM_L</td>
<td>$2^{64} - 1$</td>
</tr>
<tr>
<td>TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_S</td>
<td>$2^{42} - 1$</td>
</tr>
<tr>
<td>TLS_GOSTR341112_256_WITH_MAGMA_MGM_S</td>
<td>$2^{39} - 1$</td>
</tr>
</tbody>
</table>

4.2. HASH algorithm

The function HASH for all the cipher suites defined in this document is the GOST R 34.11-2012 [RFC6986] hash algorithm with 32-byte (256-bit) hash code.
The function HASH is used for key derivation process (see Section 7.1 of [RFC8446]), Finished message calculation, Transcript-Hash function, PSK binder value calculation, derivation of record key material (see Section 4.1.1.2) and other purposes during key exchange process.

5. Signature Scheme Definition

The signature scheme values are used to indicate to the server/client which signature algorithms and curves can be used in digital signatures and are defined by the SignatureSchemeList structure (see Section 4.2.3 of [RFC8446]) as follows:

```c
struct {
    SignatureScheme supported_signature_algorithms<2..2^16-2>;
} SignatureSchemeList;
```

This document defines new values for the "SignatureAlgorithm" registry that can be used in the SignatureSchemeList.supported_signature_algorithms field for the particular signature scheme:

```c
enum {
    gostr34102012_256a(TBD5),
    gostr34102012_256b(TBD6),
    gostr34102012_256c(TBD7),
    gostr34102012_256d(TBD8),
    gostr34102012_512a(TBD9),
    gostr34102012_512b(TBD10),
    gostr34102012_512c(TBD11)
} SignatureScheme;
```

where the values correspond to the following signature algorithms and curves:
### Table 3

<table>
<thead>
<tr>
<th>SignatureScheme Value</th>
<th>Signature Algorithm</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>gostr34102012_256a</td>
<td>GOST R 34.10-2012 + 32-byte key length</td>
<td>RFC 7091</td>
</tr>
<tr>
<td>gostr34102012_256b</td>
<td>GOST R 34.10-2012 + 32-byte key length</td>
<td>RFC 7091</td>
</tr>
<tr>
<td>gostr34102012_256c</td>
<td>GOST R 34.10-2012 + 32-byte key length</td>
<td>RFC 7091</td>
</tr>
<tr>
<td>gostr34102012_256d</td>
<td>GOST R 34.10-2012 + 32-byte key length</td>
<td>RFC 7091</td>
</tr>
<tr>
<td>gostr34102012_512a</td>
<td>GOST R 34.10-2012 + 64-byte key length</td>
<td>RFC 7091</td>
</tr>
<tr>
<td>gostr34102012_512b</td>
<td>GOST R 34.10-2012 + 64-byte key length</td>
<td>RFC 7091</td>
</tr>
<tr>
<td>gostr34102012_512c</td>
<td>GOST R 34.10-2012 + 64-byte key length</td>
<td>RFC 7091</td>
</tr>
</tbody>
</table>

### Table 4

<table>
<thead>
<tr>
<th>SignatureScheme Value</th>
<th>Curve Identifier Value</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>gostr34102012_256a</td>
<td>id-tc26-gost-3410-2012-256-paramSetA</td>
<td>RFC 7836</td>
</tr>
<tr>
<td>gostr34102012_256b</td>
<td>id-GostR3410-2001-CryptoPro-A-ParamSet</td>
<td>RFC 4357</td>
</tr>
<tr>
<td>gostr34102012_256c</td>
<td>id-GostR3410-2001-CryptoPro-B-ParamSet</td>
<td>RFC 4357</td>
</tr>
<tr>
<td>gostr34102012_256d</td>
<td>id-GostR3410-2001-CryptoPro-C-ParamSet</td>
<td>RFC 4357</td>
</tr>
<tr>
<td>gostr34102012_512a</td>
<td>id-tc26-gost-3410-12-512-paramSetA</td>
<td>RFC 7836</td>
</tr>
<tr>
<td>gostr34102012_512b</td>
<td>id-tc26-gost-3410-12-512-paramSetB</td>
<td>RFC 7836</td>
</tr>
<tr>
<td>gostr34102012_512c</td>
<td>id-tc26-gost-3410-2012-512-paramSetC</td>
<td>RFC 7836</td>
</tr>
</tbody>
</table>

This document defines the SIGN function which is used for computing signature value in CertificateVerify message (see Section 6.3.3) as follows.
SIGN(M, d_sign)

Input:
- the byte string M in B*;
- the sign key d_sign: 0 <= d_sign <= q.

Output:
- signature value sgn in B_{2*l}.

1. (r, s) = SIGNGOST(M, d_sign)
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 values of the SignatureScheme field;
  * l = 64 for gostr34102012_512a, gostr34102012_512b and gostr34102012_512c values of the SignatureScheme field;
- SIGNGOST(M, d_sign) is the GOST R 34.10-2012 [RFC7091] signature algorithm.

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

According to [RFC7091] the GOST R 34.10-2012 signature algorithm with 32-byte (256-bit) or 64-byte (512-bit) key length use the GOST R 34.11-2012 [RFC6986] hash algorithm with 32-byte (256-bit) or 64-byte (512-bit) hash code respectively (the hash algorithm is intrinsic to the signature algorithm).

6. Key Exchange and Authentication

Key exchange and authentication process in case of using GOST cipher suites is defined in [RFC8446] and Section 6.1, Section 6.2. Additionally the proposed cipher suites specify some Handshake messages (see Section 6.3).
6.1. Key Exchange

TLS 1.3 supports three basic key exchange modes in accordance with [RFC8446]:

1. (EC)DHE
2. PSK-only
3. PSK with (EC)DHE

All of the cipher suites described in this document use Diffie-Hellman over elliptic curves in (EC)DHE and PSK with (EC)DHE key exchange modes. Diffie-Hellman over finite fields SHOULD NOT be used. This document defines the process of ECDHE shared secret calculation (see Section 6.1.1) and specifies the elliptic curves that can be used during this process (see Section 6.1.2).

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.

This document defines that PSK-only key exchange mode SHOULD be used only with the internal PSKs. External PSKs SHOULD be used together with certificates in PSK with (EC)DHE mode only.

6.1.1. ECDHE Shared Secret Calculation

6.1.1.1. ECDHE Parameters

ECDHE parameters for both clients and servers are encoded in the opaque key_exchange field of a KeyShareEntry in a KeyShare structure.

For GOST cipher suites the contents are the serialized value of the following struct:

```c
struct {
    opaque X[coordinate_length];
    opaque Y[coordinate_length];
} PlainPointRepresentation;
```
X and Y, respectively, contain the binary representations of the x and y values of point Q (Q = (x, y)) in the little-endian format and are specified as follows:

\[
X = \text{str\_coordinate\_length}(x);
\]

\[
Y = \text{str\_coordinate\_length}(y).
\]

The coordinate\_length value is defined in Table 5.

6.1.1.2. 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 first message;
   - r = 1 in the case of responding to HelloRetryRequest message, E_{i_1} SHOULD correspond 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 E_{i_1}, ..., E_{i_r} curves, 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} specified in accordance with Section 4.2.8 of [RFC8446];
   - supported_group extension with supported curves E_1, ..., E_R specified in accordance with Section 4.2.7 of [RFC8446].

4. In case of receiving HelloRetryRequest message client should return to step 1 and correct parameters according to 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_res belongs to E_res. If this check fails, the client MUST abort the handshake with "handshake_failure" alert.

6. Generates Q^ECDHE:

\[
Q^\text{ECDHE} = (X^\text{ECDHE}, Y^\text{ECDHE}) = (h_{\text{res}} \times d_C^{\text{res}}) \times Q_S^{\text{res}}.
\]

7. Client MUST check whether the computed shared secret Q^ECDHE is not equal to zero point. 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:

\[
\text{ECDHE} = \text{str}_{\text{coordinate_length}}(X^\text{ECDHE}),
\]

where coordinate_length is defined in Table 5.

6.1.1.3. 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 curves E_1, ..., E_R indicated in supported_group extension in ClientHello message and the corresponding public ephemeral key value Q_C^{i_1}, ..., Q_C^{i_r}, 1 \leq i_1 \leq i_r \leq R, indicated in key_share extension. If no corresponding public ephemeral key value is found (res in \{1, ..., R\}\{i_1, ..., i_r\}), server MAY send HelloRetryRequest message with key_share extension indicated E_res and wait for the next ClientHello message.

2. Checks whether Q_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} \times P_{\text{res}}.

4. Sends ServerHello message specified in accordance with Section 4.1.3 of [RFC8446] and Section 6.3.1 with key_share extension indicated public ephemeral key value Q_S^{res} corresponding to E_res.
5. Generates $Q^{ECDHE}$:

$$Q^{ECDHE} = (X^{ECDHE}, Y^{ECDHE}) = (h_{res} \times d_{S^{res}}) \times Q_{C^{res}}.$$

6. Server MUST check whether the computed shared secret $Q^{ECDHE}$ is not equal to zero point. 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:

$$ECDHE = \text{str}_{\text{coordinate_length}}(X^{ECDHE}),$$

where coordinate_length is defined in Table 5.

6.1.2. Values for the Supported Groups Registry

The supported_groups extension indicates the set of elliptic curves supported by the client and is defined in [RFC8446].

This document defines the following values for the "Supported Groups" registry:

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

Where the values correspond to the following curves and the following values of coordinate_length parameter ("cl" column).
### Table 5

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

These values 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. This document defines that external PSKs SHOULD be combined only with the mutual authentication.

Certificate-based authentication happens via authentication messages. This document defines the signature schemes that are used for certificate-based authentication (see Section 5) and specifies some authentication messages (see Section 6.3.2, Section 6.3.3).

#### 6.3. Handshake Messages

The proposed cipher suites specify the ClientHello, ServerHello, CertificateRequest and CertificateVerify handshake messages, that are described in further detail below.
6.3.1. Hello Messages

The ClientHello message is generated in accordance with the following requirements.

- The ClientHello.cipher_suites field SHOULD contain cipher suites with the values defined in Section 4.

- The ClientHello.extensions field MAY contain the signature_algorithms and signature_algorithms_cert extensions. The extension_data field of these extensions contains a SignatureSchemeList value, where SignatureSchemeList.supported_signature_algorithms field SHOULD contain only the values defined in Section 5.

- The ClientHello.extensions field MAY contain the supported_group extension. The extension_data field of this extension contains a NamedGroupList value, where NamedGroupList.named_group_list field SHOULD contain only the values defined in Section 6.1.2.

- The ClientHello.extensions field SHOULD NOT contain early_data extension.

The ServerHello message is generated in accordance with the following requirements.

- The ServerHello.cipher_suites field SHOULD contain cipher suites with the values defined in Section 4.

6.3.2. CertificateRequest

This message is sent when requesting client authentication 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 the GOST cipher suite is negotiated, the CertificateRequest message MUST meet the following requirements.

- The CertificateRequest.extensions field MUST contain the signature_algorithms extension. The "extension_data" field of this extension contains a SignatureSchemeList value, where...
SignatureSchemeList.supported_signature_algorithms field SHOULD contain only the values defined in Section 5.

- The CertificateRequest.extensions field MAY contain the signature_algorithms_cert extension. The "extension_data" field of this extension contains a SignatureSchemeList value, where SignatureSchemeList.supported_signature_algorithms field SHOULD contain only the values defined in Section 5.

### 6.3.3. CertificateVerify

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

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

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

- The CertificateVerify.algorithm field SHOULD contain only the signature schemes with the values defined in Section 5.
- The CertificateVerify.signature field contains sgn value, which is computed as follows:
  ```
  sgn = SIGN(M, d_sign),
  ```
  where
  ```
  * function SIGN is defined in Section 5,
  * structure of M is defined in Section 4.4.3 of [RFC8446],
  * 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.
  ```

### 7. IANA Considerations

IANA is asked to assign numbers TBD1, TBD2, TBD3 and TBD4 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.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
<th>DTLS-OK</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBD1</td>
<td>TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_L</td>
<td>?</td>
<td>this RFC</td>
</tr>
<tr>
<td>TBD2</td>
<td>TLS_GOSTR341112_256_WITH_MAGMA_MGM_L</td>
<td>?</td>
<td>this RFC</td>
</tr>
<tr>
<td>TBD3</td>
<td>TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_S</td>
<td>?</td>
<td>this RFC</td>
</tr>
<tr>
<td>TBD4</td>
<td>TLS_GOSTR341112_256_WITH_MAGMA_MGM_S</td>
<td>?</td>
<td>this RFC</td>
</tr>
</tbody>
</table>

Table 6

IANA is asked to assign numbers TBD5, TBD6, TBD7, TBD8, TBD9, TBD10 
and TBD11 with the names gostr34102012_256a, gostr34102012_256b, 
gostr34102012_256c, gostr34102012_256d, gostr34102012_512a, 
gostr34102012_512b, gostr34102012_512c to the "TLS 
SignatureAlgorithm" registry, as shown below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
<th>DTLS-OK</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBD5</td>
<td>gostr34102012_256a</td>
<td>?</td>
<td>this RFC</td>
</tr>
<tr>
<td>TBD6</td>
<td>gostr34102012_256b</td>
<td>?</td>
<td>this RFC</td>
</tr>
<tr>
<td>TBD7</td>
<td>gostr34102012_256c</td>
<td>?</td>
<td>this RFC</td>
</tr>
<tr>
<td>TBD8</td>
<td>gostr34102012_256d</td>
<td>?</td>
<td>this RFC</td>
</tr>
<tr>
<td>TBD9</td>
<td>gostr34102012_512a</td>
<td>?</td>
<td>this RFC</td>
</tr>
<tr>
<td>TBD10</td>
<td>gostr34102012_512b</td>
<td>?</td>
<td>this RFC</td>
</tr>
<tr>
<td>TBD11</td>
<td>gostr34102012_512c</td>
<td>?</td>
<td>this RFC</td>
</tr>
</tbody>
</table>

Table 7
8. Historical considerations

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Curve Identifier Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>gostr34102012_256b</td>
<td>id-GostR3410_2001-CryptoPro-XchA-ParamSet</td>
</tr>
<tr>
<td></td>
<td>id-tc26-gost-3410-2012-256-paramSetB</td>
</tr>
<tr>
<td>gostr34102012_256c</td>
<td>id-tc26-gost-3410-2012-256-paramSetC</td>
</tr>
<tr>
<td>gostr34102012_256d</td>
<td>id-GostR3410-2001-CryptoPro-XchB-ParamSet</td>
</tr>
<tr>
<td></td>
<td>id-tc26-gost-3410-2012-256-paramSetD</td>
</tr>
</tbody>
</table>

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 and to resist attacks client and server SHOULD stick to the following rules.

1. While using TLSSTREE algorithm function KDF_j, j = 1, 2, 3, SHOULD be invoked only if sequence number seqnum reaches such 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.

3. Server SHOULD NOT send Application Data prior to receiving first client’s Finished message in a mutually authenticated connection.
10. References

10.1. Normative References

[DraftGostTLS12]

[DraftMGM]


10.2. Informative References

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

[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.

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

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