**Federal Agency for Technical Regulation and Metrology   
(ROSSTANDARD)**

**Technical Committee 026**

**Cryptographic Information Protection**

Information Technology

**GUIDELINES**

**The PKCS#11 extensions for implementing the**

**GOST R 34.10-2012 and GOST R 34.11-2012 Russian standards.**

**2017**

**Table of Contents**

[1. Abstract 3](#_Toc509224060)

[2. References. 3](#_Toc509224061)

[3. Note on the use of numeric identifiers. 4](#_Toc509224062)

[4. GOST R 34.10-2012 key type. 4](#_Toc509224063)

[5. Elliptic curve algorithm parameters. 4](#_Toc509224064)

[6. Using the CKO\_DOMAIN\_PARAMETERS objects. 4](#_Toc509224065)

[7. Hash mechanisms. 5](#_Toc509224066)

[8. HMAC mechanisms 5](#_Toc509224067)

[9. PRF mechanisms. 6](#_Toc509224068)

[10. Using the GOST R 34.11-2012 hash algorithm for the PBKD2 key generation. 6](#_Toc509224069)

[11. Key creation mechanism. 7](#_Toc509224070)

[12. Public key derivation mechanism. 7](#_Toc509224071)

[13. Signature and verification mechanisms. 7](#_Toc509224072)

[14. Common key derivation. 8](#_Toc509224073)

[Appendix 1. Header file with the identifier list. 9](#_Toc509224074)

[Appendix 2. Examples of using the CKO\_DOMAIN\_PARAMETERS objects to determine the elliptic curve parameter set availability. 9](#_Toc509224075)

[Appendix 3. Examples of using the CKM\_GOSTR3411\_2012\_512 mechanism. 11](#_Toc509224076)

[Appendix 4. Examples of using theCKM\_GOSTR3411\_2012\_256 mechanism. 11](#_Toc509224077)

[Appendix 5. Examples of using the CKM\_GOSTR3411\_2012\_512\_HMAC mechanism. 12](#_Toc509224078)

[Appendix 6. Examples of using the CKM\_GOSTR3411\_2012\_256\_HMAC mechanism. 13](#_Toc509224079)

[Appendix 7. Examples of using the CKM\_TLS\_GOST\_PRF\_2012\_256 mechanism. 13](#_Toc509224080)

[Appendix 8. Examples of using the CKM\_TLS\_GOST\_PRF\_2012\_512 mechanism. 15](#_Toc509224081)

[Appendix 9. Examples of using the GOST R 34.11-2012 hash algorithm in the PKCS #5 PBKDF2 key generation mechanism. 16](#_Toc509224082)

[Appendix 10. Examples of using the CKM\_GOSTR3410\_512\_KEY\_PAIR\_GEN mechanism for the key pair generation. 17](#_Toc509224083)

[Appendix 11. Examples of using the CKM\_GOSTR3410\_PUBLIC\_KEY\_DERIVE mechanism for the public key derivation. 18](#_Toc509224084)

[Appendix 12. Examples of using mechanisms for the signature and verification processes according to GOST R 34.10-2012, 256 bit. 19](#_Toc509224085)

[Appendix 13. Examples of using mechanisms for the signature and verification processes according to GOST R 34.10-2012, 512 bit. 21](#_Toc509224086)

[Appendix 14. Examples of using the CKM\_GOSTR3410\_2012\_DERIVE mechanism for the common secret key derivation. 24](#_Toc509224087)

[Appendix 15. Examples of using the CKM\_GOSTR3410\_2012\_DERIVE mechanism gor the common secret key derivation. 26](#_Toc509224088)

# Abstract

This document defines the PKCS#11 standard extension, which enables the use of the cryptographic algorithms described in **GOST R 34.10-2012, GOST R 34.11-2012** federal standards, and the algorithms built on their basis, as well as the use of keys, certificates, algorithm parameters, and other objects suitable for these standards.

# References.

* **[GOST3410-2012]** "Information technology. Cryptographic data security. Signature and verification processes of [electronic] digital signature", GOST R 34.10-2012 Federal Agency on Technical Regulating and Metrology (In Russian), 2012.
* **[GOST3411-2012]** "Information technology. Cryptographic Data Security. Hashing function", GOST R 34.11-2012 Federal Agency on Technical Regulating and Metrology (In Russian), 2012.
* **[PKCS #11** **Base] —** PKCS #11. Cryptographic Token Interface. Base Specification. Version 2.40
* **[PKCS #11** **Mechanisms] —** PKCS #11. Cryptographic Token Interface. Current Mechanisms Specification. Version 2.40
* **[RFC 2104] —** H. Krawczyk, M. Bellare and R. Canetti, HMAC: Keyed-Hashing for Message Authentication, Informational, IETF RFC 2104, February 1997)
* **[RFC 2898] —** B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, September 2000.
* **[RFC 4357]**  — Popov V., Kurepkin I. and S. Leontiev, Additional Cryptographic Algorithms for Use with GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms, IETF RFC 4357, January 2006.
* **[RFC 6986]** —V. Dolmatov, Ed. A. Degtyarev, "GOST R 34.11-2012: Hash Function", RFC 6986, DOI 10.17487/RFC6986, August 2013.
* **[RFC 7091]** — V. Dolmatov, Ed. A. Degtyarev, "GOST R 34.10-2012: Digital Signature Algorithm", RFC 7091, DOI 10.17487/RFC7091, December 2013.
* **[RFC 7836]** — S. Smyshlyaev, Ed., E. Alekseev, I. Oshkin, V. Popov, S. Leontiev, V. Podobaev, 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.

# Note on the use of numeric identifiers.

Constant definitions, code fragments and samples in the document text are given in the “C” programming language notation.

Before this extension is added to the official PKCS#11 specification and permanent values are provided for all numeric identifiers, constants are chosen according to the following rules:

1) If the high bit (0x80000000) is set, it is a sign of a non-standard vendor defined value.

2) In every independent namespace, numerical identifiers can be chosen arbitrarily.

We chose the following value as the vendor identifier:

#define NSSCK\_VENDOR\_PKCS11\_RU\_TEAM 0xd4321000 //0x80000000|0x54321000

#define CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 NSSCK\_VENDOR\_PKCS11\_RU\_TEAM

For the header file with the list of identifiers, see Appendix 1.

# GOST R 34.10-2012 key type.

This PKCS#11extension introduces a new CKK\_GOSTR3410\_512 key type.

The CKA\_KEY\_TYPE attribute can have the CKK\_GOSTR3410\_512 value for the CKO\_PUBLIC\_KEY, CKO\_PRIVATE\_KEY and CKO\_DOMAIN\_PARAMETERS objects.

A key object of the CKK\_GOSTR3410\_512 type is intended to be used with GOST R 34.10-2012 algorithm with a 512-bit private key.

Such keys can be used with the following mechanisms:

CKM\_GOSTR3410\_512\_KEY\_PAIR\_GEN

CKM\_GOSTR3410\_512

CKM\_GOSTR3410\_2012\_DERIVE

# Elliptic curve algorithm parameters.

GOST R 34.10-2012 algorithm can use a set of elliptic curve parameters as recommended by **[RFC 4357]** and **[RFC 7836, section 5].**

A parameter set, with OID “1.2.643.7.1.2.1.2.1” is the default parameter set for a 512-bit private key. The default set is used if the curve parameters are not specified in the key template.

The elliptic curve parameters are represented as a DER-encoded OID of ASN.1, and used as the values of the CKA\_GOSTR3410\_PARAMS attribute in the key objects and templates, transmitted to functions.

# Using the CKO\_DOMAIN\_PARAMETERS objects.

The PKCS#11 implementation may support key mechanism, but it does not necessarily support all possible parameters. Special objects may be used to determine the list of elliptic parameters, supported in the implementation.

If the key storage has one of such objects, it means that one set of parameters is supported. The CKA\_CLASS attribute is set to CKO\_DOMAIN\_PARAMETERS for such objects.

Besides the common attributes, an object must have the CKA\_KEY\_TYPE attribute, set to the key type to which the CKK\_GOSTR3410 or CKK\_GOSTR3410\_512 parameters apply.

The mandatory CKA\_OBJECT\_ID attribute must have an ASN.1 encoded OID, according to the rules defined in the previous section.

The CKA\_VALUE attribute may be absent or unavailable.

For an example of using the CKO\_DOMAIN\_PARAMETERS objects to determine the availability of the elliptic curve parameter set, see Appendix 2.

# Hash mechanisms.

**This document extends the mechanism list to include the GOST R 34.11-2012 hash functions [RFC 6986].**

**The GOST R 34.11-2012 [RFC 6986] document** defines two hash functions with value lengths of 256 and 512 bits. Two mechanisms - CKM\_GOSTR3411\_2012\_256 and CKM\_GOSTR3411\_2012\_512 - are i**ntroduced** to enable the use of two hash functions with value lengths of 32 and 64 bytes.

These mechanisms do not use any parameters.

These mechanisms can be used in the C\_DigestInit function **[PKCS #11** **Base, section 5.10]**.

For examples of using the CKM\_GOSTR3411\_2012\_512 and CKM\_GOSTR3411\_2012\_256 mechanisms, see Appendix 3 and 4, respectively.

# HMAC mechanisms

The document defines two HMAC functions based on the **GOST R 34.11-2012** hash algorithm **[RFC 6986]** and the HMAC function definition in **[RFC 2104]. [RFC 7836 4.1]** also describes these functions**.**

CKM\_GOSTR3411\_2012\_256\_HMAC is the HMAC mechanism with B = 64, L = 32 and a value length of 32 bytes.

CKM\_GOSTR3411\_2012\_512\_HMAC is the HMAC mechanism with B = 64, L = 64 and a value length of 64 bytes.

These mechanisms do not use any parameters.

Both of these HMAC mechanisms can use keys of the CKK\_GOST28147 and CKK\_GENERIC\_SECRET types.

These mechanisms are used in the C\_SignInit function **[PKCS #11** **Base, section 5.11]**.

For examples of using the CKM\_GOSTR3411\_2012\_512\_HMAC and CKM\_GOSTR3411\_2012\_256\_HMAC mechanisms, see Appendix 5 and 6, respectively.

# PRF mechanisms.

The Transport Layer Security (TLS) protocol (1.0 and higher versions) with the GOST R 34.10-2012 and GOST R 34.11-2012 Russian algorithms uses the PRF\_TLS\_GOSTR3411\_2012\_256 and PRF\_TLS\_GOSTR3411\_2012\_512 HMAC-based pseudo-random function (PRF) transformations according to **[RFC 6986]** as described in **[RFC 7836, section 4.2.1]** «PRFs for the TLS Protocol».

According to **[PKCS#11** **Base, section 5.13],** the C\_DeriveKeyfunction is used to obtain the PRF values. The CKM\_TLS\_GOST\_PRF\_2012\_256 and CKM\_TLS\_GOST\_PRF\_2012\_512 mechanisms are used to apply the PRF\_TLS\_GOSTR3411\_2012\_256 и PRF\_TLS\_GOSTR3411\_2012\_512 functions.

The CK\_TLS\_PRF\_PARAMS structure determines the set of parameters for these mechanisms. The C\_DeriveKey function produces a pseudo-random sequence of *pulOutputLen* bytes and returns the result in the *pOutput* field, instead of the key object*.* The *phKey* argument is not used and must be set to NULL\_PTR.

Both these PRF mechanisms can use keys of the CKK\_GOST28147 and CKK\_GENERIC\_SECRET types.

For examples of using the CKM\_TLS\_GOST\_PRF\_2012\_512 and CKM\_TLS\_GOST\_PRF\_2012\_256 mechanisms, see Appendix 7 and 8, respectively.

# Using the GOST R 34.11-2012 hash algorithm for the PBKD2 key generation.

The **[PKCS #11** **Mechanisms, section 2.26] s**tandard supports the password-based key derivation using the PBKDF2 algorithm, according **[RFC 2898]**. The C\_GenerateKey function is used with the CKM\_PKCS5\_PBKD2 mechanism.

The CK\_PKCS5\_PBKD2\_PARAMS2 structure defines the mechanism parameters. To use the GOST R 34.11-2012 hash algorithm, in the field

CK\_PKCS5\_PBKD2\_PSEUDO\_RANDOM\_FUNCTION\_TYPE prf;

of the CK\_PKCS5\_PBKD2\_PARAMS2 structure, the CKP\_PKCS5\_PBKD2\_HMAC\_GOSTR3411\_2012\_512 value must be set. The value is defined as follows

#define CKP\_PKCS5\_PBKD2\_HMAC\_GOSTR3411\_2012\_512 \

(CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x003)

Additional parameters are not used, the *pPrfData* value must be *NULL, and* *ulPrfDataLen* must be zero.

For an example of password-based key creation see Appendix 9.

# Key creation mechanism.

To create a key pair for the GOST R 34.10-2012 algorithm with the private key length of 512 bits, the CKM\_GOSTR3410\_512\_KEY\_PAIR\_GEN mechanism is used.

The C\_GenerateKeyPair function is used as described in **[PKCS #11** **Base, section 5.13]**.

The mechanism do not use parameters. Elliptic curve parameters can be set in the key object templates as an ASN.1 DER encoded OID, as described in section 5, “Elliptic curve algorithm parameters”. The creation mechanism uses the default parameter set with OID “1.2.643.7.1.2.1.2.1”, if no elliptic curve is specified.

For an example of creating a 512-bit key pair for the GOST R 34.11-2012 algorithm, see Appendix 10

# Public key derivation mechanism.

To derive a public key from a private one, the CKM\_GOSTR3410\_PUBLIC\_KEY\_DERIVE mechanism is used.

The C\_DeriveKey function is used as described in **[PKCS #11** **Base, section 5.13].**

This mechanism do not use parameters. The derivation mechanism uses the elliptic curve defined in the private key.

For an example of the public key creation from the private key, see Appendix 11.

# Signature and verification mechanisms.

We introduce the CKM\_GOSTR3410\_512 mechanism for the GOST R 34.10-2012 signature and verification processes. It is used for a calculated hash function value using the GOST R 34.11-2012 algorithm with a 512-bit key.

The CKM\_GOSTR3410\_WITH\_GOSTR3411\_2012\_256 mechanism should be used for the GOST R 34.10-2012 signature and verification processes together with the GOST R 34.11-2012 hash algorithm with 256-bit message digest length.

The CKM\_GOSTR3410\_WITH\_GOSTR3411\_2012\_512 mechanism should be used for the GOST R 34.10-2012 signature and verification processes together with the GOST R 34.11-2012 hash algorithm with 512-bit message digest length.

These mechanisms are used in the C\_SignInit and C\_VerifyInit functions, according to **[PKCS #11** **Base, section 5.11]**.

These mechanisms do not use parameters.

For examples of using the GOST R 34.10-2012 signature and verification mechanisms, see Appendix 12 and 13.

# Common key derivation.

We introduce the CKM\_GOSTR3410\_2012\_DERIVE mechanism for the Diffie-Hellman protocol for GOST R 34.10-2012 keys.

This mechanism implements the algorithm of the common secret key derivation from the public and private GOST R 34.10-2012 keys with the length of 256 and 512-bit. It allows obtaining the 256-bit key material, which can be used in cryptographic protocols.

The key material obtained using this mechanism can be optionally modified by the diversification algorithm.

This mechanism implements the algorithm described in **[RFC 7836, section 4.3]**.

The CKM\_GOSTR3410\_2012\_DERIVE mechanism uses parameters, with a byte array, which has the following structure:

- 4 bytes that specify the KDF value in the little-endian format, as defined in **[PKCS #11** **Mechanisms, section 2.45.5].**

- 4 bytes that specify the public key length in the little-endian format (64 or 128 bytes)

- Public key, the length of which is defined by the previous field (64 or 128 bytes).

- 4 bytes that specify the UKM length in the little-endian format. (minimum 8 bytes).

- UKM value, the length of which is defined by the previous field.

This mechanism is used in the C\_DeriveKey function according to **[PKCS #11** **Base, section 5.13]**.

For examples of using the CKM\_GOSTR3410\_2012\_DERIVE mechanism to create a common secret key, see Appendix 14 and 15

# Appendix 1. Header file with the identifier list.

#define NSSCK\_VENDOR\_PKCS11\_RU\_TEAM 0xD4321000 /\* 0x80000000 | 0x54321000 \*/

#define CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 NSSCK\_VENDOR\_PKCS11\_RU\_TEAM

/\* GOST KEY TYPES \*/

#define CKK\_GOSTR3410\_256 CKK\_GOSTR3410

#define CKK\_GOSTR3410\_512 (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x003)

#define CKK\_KUZNECHIK (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x004)

#define CKK\_MAGMA (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x005)

/\* GOST OBJECT ATTRIBUTES \*/

#define CKA\_GOSTR3410\_256PARAMS CKA\_GOSTR3410\_PARAMS

/\* PKCS #5 PRF Functions \*/

#define CKP\_PKCS5\_PBKD2\_HMAC\_GOSTR3411\_2012\_512 (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x003)

/\* GOST MECHANISMS \*/

#define CKM\_GOSTR3410\_256\_KEY\_PAIR\_GEN CKM\_GOSTR3410\_KEY\_PAIR\_GEN

#define CKM\_GOSTR3410\_512\_KEY\_PAIR\_GEN (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x005)

#define CKM\_GOSTR3410\_256 CKM\_GOSTR3410

#define CKM\_GOSTR3410\_512 (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x006)

#define CKM\_GOSTR3410\_2012\_DERIVE (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x007)

#define CKM\_GOSTR3410\_12\_DERIVE CKM\_GOSTR3410\_2012\_DERIVE

#define CKM\_GOSTR3410\_WITH\_GOSTR3411\_94 CKM\_GOSTR3410\_WITH\_GOSTR3411

#define CKM\_GOSTR3410\_WITH\_GOSTR3411\_2012\_256 (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x008)

#define CKM\_GOSTR3410\_WITH\_GOSTR3411\_2012\_512 (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x009)

#define CKM\_GOSTR3410\_WITH\_GOSTR3411\_12\_256 CKM\_GOSTR3410\_WITH\_GOSTR3411\_2012\_256

#define CKM\_GOSTR3410\_WITH\_GOSTR3411\_12\_512 CKM\_GOSTR3410\_WITH\_GOSTR3411\_2012\_512

#define CKM\_GOSTR3410\_PUBLIC\_KEY\_DERIVE (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x00A)

#define CKM\_GOSTR3411\_94 CKM\_GOSTR3411

#define CKM\_GOSTR3411\_2012\_256 (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x012)

#define CKM\_GOSTR3411\_2012\_512 (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x013)

#define CKM\_GOSTR3411\_12\_256 CKM\_GOSTR3411\_2012\_256

#define CKM\_GOSTR3411\_12\_512 CKM\_GOSTR3411\_2012\_512

#define CKM\_GOSTR3411\_94\_HMAC CKM\_GOSTR3411\_HMAC

#define CKM\_GOSTR3411\_2012\_256\_HMAC (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x014)

#define CKM\_GOSTR3411\_2012\_512\_HMAC (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x015)

#define CKM\_GOSTR3411\_12\_256\_HMAC CKM\_GOSTR3411\_2012\_256\_HMAC

#define CKM\_GOSTR3411\_12\_512\_HMAC CKM\_GOSTR3411\_2012\_512\_HMAC

#define CKM\_TLS\_GOST\_PRF\_2012\_256 (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x016)

#define CKM\_TLS\_GOST\_PRF\_2012\_512 (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x017)

#define CKM\_TLS\_GOST\_PRE\_MASTER\_KEY\_GEN CKM\_GOST28147\_KEY\_GEN

#define CKM\_TLS\_GOST\_MASTER\_KEY\_DERIVE\_2012\_256 (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x018)

#define CKM\_KDF\_4357 (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x025)

#define CKM\_KDF\_GOSTR3411\_2012\_256 (CK\_VENDOR\_PKCS11\_RU\_TEAM\_TC26 |0x026)

# Appendix 2. Examples of using the CKO\_DOMAIN\_PARAMETERS objects to determine the elliptic curve parameter set availability.

CK\_RV sample\_domain\_param(CK\_FUNCTION\_LIST\_PTR pF, CK\_SESSION\_HANDLE hSession)

{

CK\_RV rv = CKR\_OK;

CK\_OBJECT\_HANDLE objectList[10] = { 0 };

CK\_ULONG objectCount = sizeof(objectList) / sizeof(\*objectList);

CK\_OBJECT\_CLASS domainParamsClass = CKO\_DOMAIN\_PARAMETERS;

CK\_KEY\_TYPE key\_type\_gost3410\_256 = CKK\_GOSTR3410;

CK\_KEY\_TYPE key\_type\_gost3410\_512 = CKK\_GOSTR3410\_512;

CK\_ATTRIBUTE domainParams\_256[] =

{

{ CKA\_CLASS, &domainParamsClass, sizeof(domainParamsClass) },

{ CKA\_KEY\_TYPE, &key\_type\_gost3410\_256, sizeof(key\_type\_gost3410\_256) },

};

CK\_ATTRIBUTE domainParams\_512[] =

{

{ CKA\_CLASS, &domainParamsClass, sizeof(domainParamsClass) },

{ CKA\_KEY\_TYPE, &key\_type\_gost3410\_512, sizeof(key\_type\_gost3410\_512) },

};

const CK\_BYTE default\_param\_256[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x01}; // "1.2.643.2.2.35.1"

const CK\_BYTE default\_param\_512[] = {0x06, 0x09, 0x2a, 0x85, 0x03, 0x07, 0x01, 0x02, 0x01, 0x02, 0x01}; // "1.2.643.2.2.7.1.2.1.2.1"

CK\_BYTE domainParamValue[20];

CK\_ATTRIBUTE attr = { CKA\_OBJECT\_ID, domainParamValue, sizeof(domainParamValue) };

rv = pF->C\_FindObjectsInit( hSession, domainParams\_256, sizeof( domainParams\_256 ) / sizeof( CK\_ATTRIBUTE ) );

assert(rv == CKR\_OK);

rv = pF->C\_FindObjects( hSession, objectList, sizeof(objectList) / sizeof(\*objectList), &objectCount );

assert(rv == CKR\_OK);

rv = pF->C\_FindObjectsFinal( hSession );

assert(rv == CKR\_OK);

for ( CK\_ULONG i = 0; i < objectCount; i++ )

{

attr.ulValueLen = sizeof(domainParamValue);

rv = pF->C\_GetAttributeValue( hSession, objectList[i], &attr, 1 );

assert(rv == CKR\_OK);

if ( memcmp( domainParamValue, default\_param\_256, attr.ulValueLen ) == 0) {

printf("default 256-bit param set \"1.2.643.2.2.35.1\" supported.\n");

break;

}

}

rv = pF->C\_FindObjectsInit( hSession, domainParams\_512, sizeof( domainParams\_512 ) / sizeof( CK\_ATTRIBUTE ) );

assert(rv == CKR\_OK);

rv = pF->C\_FindObjects( hSession, objectList, sizeof(objectList) / sizeof(\*objectList), &objectCount );

assert(rv == CKR\_OK);

rv = pF->C\_FindObjectsFinal( hSession );

assert(rv == CKR\_OK);

for ( CK\_ULONG i = 0; i < objectCount; i++ )

{

attr.ulValueLen = sizeof(domainParamValue);

rv = pF->C\_GetAttributeValue( hSession, objectList[i], &attr, 1 );

assert(rv == CKR\_OK);

if ( memcmp( domainParamValue, default\_param\_512, attr.ulValueLen ) == 0) {

printf("default 512-bit param set \"1.2.643.2.2.7.1.2.1.2.1\" supported.\n");

break;

}

}

return rv;

}

# Appendix 3. Examples of using the CKM\_GOSTR3411\_2012\_512 mechanism.

CK\_RV sample\_digest\_stribog\_512(CK\_FUNCTION\_LIST\_PTR pF, CK\_SESSION\_HANDLE hSession)

{

CK\_RV rv;

CK\_BYTE pDigest[64];

CK\_ULONG ulDigestLen = sizeof(pDigest);

CK\_MECHANISM digestMechanism = { CKM\_GOSTR3411\_12\_512, NULL\_PTR, 0 };

// data from GOST R 3411-2012

CK\_BYTE data[] = "012345678901234567890123456789012345678901234567890123456789012";

const CK\_BYTE ETALON[] = {

0x1B, 0x54, 0xD0, 0x1A, 0x4A, 0xF5, 0xB9, 0xD5,

0xCC, 0x3D, 0x86, 0xD6, 0x8D, 0x28, 0x54, 0x62,

0xB1, 0x9A, 0xBC, 0x24, 0x75, 0x22, 0x2F, 0x35,

0xC0, 0x85, 0x12, 0x2B, 0xE4, 0xBA, 0x1F, 0xFA,

0x00, 0xAD, 0x30, 0xF8, 0x76, 0x7B, 0x3A, 0x82,

0x38, 0x4C, 0x65, 0x74, 0xF0, 0x24, 0xC3, 0x11,

0xE2, 0xA4, 0x81, 0x33, 0x2B, 0x08, 0xEF, 0x7F,

0x41, 0x79, 0x78, 0x91, 0xC1, 0x64, 0x6F, 0x48,

};

rv = pF->C\_DigestInit(hSession, &digestMechanism);

assert(rv == CKR\_OK);

rv = pF->C\_DigestUpdate(hSession, data, sizeof(data)-1);

assert(rv == CKR\_OK);

rv = pF->C\_DigestFinal(hSession, pDigest, &ulDigestLen);

assert(rv == CKR\_OK);

return memcmp(pDigest, ETALON, ulDigestLen) ? CKR\_FUNCTION\_FAILED : CKR\_OK;

}

# Appendix 4. Examples of using theCKM\_GOSTR3411\_2012\_256 mechanism.

CK\_RV sample\_digest\_stribog\_256(CK\_FUNCTION\_LIST\_PTR pF, CK\_SESSION\_HANDLE hSession)

{

CK\_RV rv;

CK\_BYTE pDigest[32];

CK\_ULONG ulDigestLen = sizeof(pDigest);

CK\_MECHANISM digestMechanism = { CKM\_GOSTR3411\_12\_256, NULL\_PTR, 0 };

// data from GOST R 3411-2012

CK\_BYTE data[] = "012345678901234567890123456789012345678901234567890123456789012";

const CK\_BYTE ETALON[] = {

0x9D, 0x15, 0x1E, 0xEF, 0xD8, 0x59, 0x0B, 0x89,

0xDA, 0xA6, 0xBA, 0x6C, 0xB7, 0x4A, 0xF9, 0x27,

0x5D, 0xD0, 0x51, 0x02, 0x6B, 0xB1, 0x49, 0xA4,

0x52, 0xFD, 0x84, 0xE5, 0xE5, 0x7B, 0x55, 0x00,

};

rv = pF->C\_DigestInit(hSession, &digestMechanism);

assert(rv == CKR\_OK);

rv = pF->C\_DigestUpdate(hSession, data, sizeof(data)-1);

assert(rv == CKR\_OK);

rv = pF->C\_DigestFinal(hSession, pDigest, &ulDigestLen);

assert(rv == CKR\_OK);

return memcmp(pDigest, ETALON, ulDigestLen) ? CKR\_FUNCTION\_FAILED : CKR\_OK;

}

# Appendix 5. Examples of using the CKM\_GOSTR3411\_2012\_512\_HMAC mechanism.

CK\_RV sample\_hmac\_stribog\_512(CK\_FUNCTION\_LIST\_PTR pF, CK\_SESSION\_HANDLE hSession)

{

CK\_RV rv;

CK\_MECHANISM hmacMechanism = { CKM\_GOSTR3411\_12\_512\_HMAC, NULL\_PTR, 0};

CK\_BBOOL bTrue = CK\_TRUE;

CK\_BBOOL bFalse = CK\_FALSE;

CK\_KEY\_TYPE keyType = CKK\_GENERIC\_SECRET;

CK\_OBJECT\_CLASS keyObject = CKO\_SECRET\_KEY;

CK\_BYTE keyValue[] = {

0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,

0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,

0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,

0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,

};

CK\_ATTRIBUTE secretKeyTemplate[] = {

{CKA\_CLASS, &keyObject, sizeof(keyObject)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &bFalse, sizeof(bTrue)},

{CKA\_SIGN, &bTrue, sizeof(bTrue)},

{CKA\_VERIFY, &bTrue, sizeof(bTrue)},

{CKA\_EXTRACTABLE, &bTrue, sizeof(bTrue)},

{CKA\_VALUE, keyValue, sizeof(keyValue)},

};

CK\_BYTE testData[] = {

0x01, 0x26, 0xBD, 0xB8, 0x78, 0x00, 0xAF, 0x21,

0x43, 0x41, 0x45, 0x65, 0x63, 0x78, 0x01, 0x00,

};

CK\_OBJECT\_HANDLE secretKey = CK\_INVALID\_HANDLE;

CK\_BYTE hmacValue[64];

CK\_ULONG hmacValueLength = sizeof(hmacValue);

const CK\_BYTE ETALON[] = {

0xA5, 0x9B, 0xAB, 0x22, 0xEC, 0xAE, 0x19, 0xC6,

0x5F, 0xBD, 0xE6, 0xE5, 0xF4, 0xE9, 0xF5, 0xD8,

0x54, 0x9D, 0x31, 0xF0, 0x37, 0xF9, 0xDF, 0x9B,

0x90, 0x55, 0x00, 0xE1, 0x71, 0x92, 0x3A, 0x77,

0x3D, 0x5F, 0x15, 0x30, 0xF2, 0xED, 0x7E, 0x96,

0x4C, 0xB2, 0xEE, 0xDC, 0x29, 0xE9, 0xAD, 0x2F,

0x3A, 0xFE, 0x93, 0xB2, 0x81, 0x4F, 0x79, 0xF5,

0x00, 0x0F, 0xFC, 0x03, 0x66, 0xC2, 0x51, 0xE6,

};

rv = pF->C\_CreateObject(hSession, secretKeyTemplate, sizeof(secretKeyTemplate) / sizeof(CK\_ATTRIBUTE), &secretKey);

assert(rv == CKR\_OK);

rv = pF->C\_SignInit(hSession, &hmacMechanism, secretKey);

assert(rv == CKR\_OK);

rv = pF->C\_Sign(hSession, testData, sizeof(testData), hmacValue, &hmacValueLength);

assert(rv == CKR\_OK);

rv = pF->C\_DestroyObject(hSession, secretKey);

assert(rv == CKR\_OK);

return memcmp(hmacValue, ETALON, hmacValueLength) ? CKR\_FUNCTION\_FAILED : CKR\_OK;

}

# Appendix 6. Examples of using the CKM\_GOSTR3411\_2012\_256\_HMAC mechanism.

CK\_RV sample\_hmac\_stribog\_256(CK\_FUNCTION\_LIST\_PTR pF, CK\_SESSION\_HANDLE hSession)

{

CK\_RV rv = CKR\_OK;

CK\_MECHANISM hmacMechanism = { CKM\_GOSTR3411\_12\_256\_HMAC, NULL\_PTR, 0};

CK\_BBOOL bTrue = CK\_TRUE;

CK\_BBOOL bFalse = CK\_FALSE;

CK\_KEY\_TYPE keyType = CKK\_GENERIC\_SECRET;

CK\_OBJECT\_CLASS keyObject = CKO\_SECRET\_KEY;

CK\_BYTE keyValue[] = {

0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,

0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,

0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,

0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,

};

CK\_ATTRIBUTE secretKeyTemplate[] = {

{CKA\_CLASS, &keyObject, sizeof(keyObject)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &bFalse, sizeof(bTrue)},

{CKA\_SIGN, &bTrue, sizeof(bTrue)},

{CKA\_VERIFY, &bTrue, sizeof(bTrue)},

{CKA\_EXTRACTABLE, &bTrue, sizeof(bTrue)},

{CKA\_VALUE, keyValue, sizeof(keyValue)},

};

CK\_BYTE testData[] = {

0x01, 0x26, 0xBD, 0xB8, 0x78, 0x00, 0xAF, 0x21,

0x43, 0x41, 0x45, 0x65, 0x63, 0x78, 0x01, 0x00,

};

CK\_OBJECT\_HANDLE secretKey = CK\_INVALID\_HANDLE;

CK\_BYTE hmacValue[32];

CK\_ULONG hmacValueLength = sizeof(hmacValue);

const CK\_BYTE ETALON[] = {

0xA1, 0xAA, 0x5F, 0x7D, 0xE4, 0x02, 0xD7, 0xB3,

0xD3, 0x23, 0xF2, 0x99, 0x1C, 0x8D, 0x45, 0x34,

0x01, 0x31, 0x37, 0x01, 0x0A, 0x83, 0x75, 0x4F,

0xD0, 0xAF, 0x6D, 0x7C, 0xD4, 0x92, 0x2E, 0xD9,

};

rv = pF->C\_CreateObject(hSession, secretKeyTemplate, sizeof(secretKeyTemplate) / sizeof(CK\_ATTRIBUTE), &secretKey);

assert(rv == CKR\_OK);

rv = pF->C\_SignInit(hSession, &hmacMechanism, secretKey);

assert(rv == CKR\_OK);

rv = pF->C\_Sign(hSession, testData, sizeof(testData), hmacValue, &hmacValueLength);

assert(rv == CKR\_OK);

rv = pF->C\_DestroyObject(hSession, secretKey);

assert(rv == CKR\_OK);

return memcmp(hmacValue, ETALON, hmacValueLength) ? CKR\_FUNCTION\_FAILED : CKR\_OK;

}

# Appendix 7. Examples of using the CKM\_TLS\_GOST\_PRF\_2012\_256 mechanism.

CK\_RV sample\_prf\_2012\_256(CK\_FUNCTION\_LIST\_PTR pF, CK\_SESSION\_HANDLE hSession)

{

CK\_RV rv;

CK\_OBJECT\_HANDLE keyHandle = CK\_INVALID\_HANDLE;

CK\_BYTE keyValue[] =

{

0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,

0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,

0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,

0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,

};

CK\_BYTE seed[] = {

0x18, 0x47, 0x1D, 0x62, 0x2D, 0xC6, 0x55, 0xC4,

0xD2, 0xD2, 0x26, 0x96, 0x91, 0xCA, 0x4A, 0x56,

0x0B, 0x50, 0xAB, 0xA6, 0x63, 0x55, 0x3A, 0xF2,

0x41, 0xF1, 0xAD, 0xA8, 0x82, 0xC9, 0xF2, 0x9A,

};

CK\_BYTE label[] = {

0x11, 0x22, 0x33, 0x44, 0x55,

};

const CK\_BYTE ETALON[] = {

0xFF, 0x09, 0x66, 0x4A, 0x44, 0x74, 0x58, 0x65,

0x94, 0x4F, 0x83, 0x9E, 0xBB, 0x48, 0x96, 0x5F,

0x15, 0x44, 0xFF, 0x1C, 0xC8, 0xE8, 0xF1, 0x6F,

0x24, 0x7E, 0xE5, 0xF8, 0xA9, 0xEB, 0xE9, 0x7F,

0xC4, 0xE3, 0xC7, 0x90, 0x0E, 0x46, 0xCA, 0xD3,

0xDB, 0x6A, 0x01, 0x64, 0x30, 0x63, 0x04, 0x0E,

0xC6, 0x7F, 0xC0, 0xFD, 0x5C, 0xD9, 0xF9, 0x04,

0x65, 0x23, 0x52, 0x37, 0xBD, 0xFF, 0x2C, 0x02,

};

CK\_BBOOL bTrue = CK\_TRUE;

CK\_BBOOL bFalse = CK\_FALSE;

CK\_KEY\_TYPE keyType = CKK\_GOST28147;

CK\_OBJECT\_CLASS keyObject = CKO\_SECRET\_KEY;

CK\_ATTRIBUTE secretKeyTemplate[] = {

{CKA\_CLASS, &keyObject, sizeof(keyObject)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &bFalse, sizeof(bTrue)},

{CKA\_VALUE, keyValue, sizeof(keyValue)},

{CKA\_DERIVE, &bTrue, sizeof(bTrue)},

{CKA\_EXTRACTABLE, &bTrue, sizeof(bTrue)},

};

CK\_TLS\_PRF\_PARAMS prfParams = { 0 };

CK\_ULONG outputLen = sizeof( ETALON );

CK\_BYTE outputBuf[sizeof( ETALON )];

CK\_MECHANISM prfMech = { CKM\_TLS\_GOST\_PRF\_2012\_256, &prfParams, sizeof( prfParams ) };

rv = pF->C\_CreateObject(hSession, secretKeyTemplate, sizeof(secretKeyTemplate) / sizeof(CK\_ATTRIBUTE), &keyHandle);

assert(rv == CKR\_OK);

prfParams.pSeed = seed;

prfParams.ulSeedLen = sizeof( seed );

prfParams.pLabel = label;

prfParams.ulLabelLen = sizeof( label );

prfParams.pOutput = outputBuf;

prfParams.pulOutputLen = &outputLen;

rv = pF->C\_DeriveKey( hSession, &prfMech, keyHandle, NULL, 0, NULL );

assert(rv == CKR\_OK && \*prfParams.pulOutputLen == sizeof(ETALON));

rv = pF->C\_DestroyObject( hSession, keyHandle );

assert(rv == CKR\_OK);

return memcmp(prfParams.pOutput, ETALON, sizeof(ETALON)) ? CKR\_FUNCTION\_FAILED : CKR\_OK;

}

# Appendix 8. Examples of using the CKM\_TLS\_GOST\_PRF\_2012\_512 mechanism.

CK\_RV sample\_prf\_2012\_512(CK\_FUNCTION\_LIST\_PTR pF, CK\_SESSION\_HANDLE hSession)

{

CK\_RV rv;

CK\_OBJECT\_HANDLE keyHandle = CK\_INVALID\_HANDLE;

CK\_BYTE keyValue[] =

{

0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,

0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,

0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,

0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,

};

CK\_BYTE seed[] = {

0x18, 0x47, 0x1D, 0x62, 0x2D, 0xC6, 0x55, 0xC4,

0xD2, 0xD2, 0x26, 0x96, 0x91, 0xCA, 0x4A, 0x56,

0x0B, 0x50, 0xAB, 0xA6, 0x63, 0x55, 0x3A, 0xF2,

0x41, 0xF1, 0xAD, 0xA8, 0x82, 0xC9, 0xF2, 0x9A,

};

CK\_BYTE label[] = {

0x11, 0x22, 0x33, 0x44, 0x55,

};

const CK\_BYTE ETALON[] = {

0xF3, 0x51, 0x87, 0xA3, 0xDC, 0x96, 0x55, 0x11,

0x3A, 0x0E, 0x84, 0xD0, 0x6F, 0xD7, 0x52, 0x6C,

0x5F, 0xC1, 0xFB, 0xDE, 0xC1, 0xA0, 0xE4, 0x67,

0x3D, 0xD6, 0xD7, 0x9D, 0x0B, 0x92, 0x0E, 0x65,

0xAD, 0x1B, 0xC4, 0x7B, 0xB0, 0x83, 0xB3, 0x85,

0x1C, 0xB7, 0xCD, 0x8E, 0x7E, 0x6A, 0x91, 0x1A,

0x62, 0x6C, 0xF0, 0x2B, 0x29, 0xE9, 0xE4, 0xA5,

0x8E, 0xD7, 0x66, 0xA4, 0x49, 0xA7, 0x29, 0x6D,

0xE6, 0x1A, 0x7A, 0x26, 0xC4, 0xD1, 0xCA, 0xEE,

0xCF, 0xD8, 0x0C, 0xCA, 0x65, 0xC7, 0x1F, 0x0F,

0x88, 0xC1, 0xF8, 0x22, 0xC0, 0xE8, 0xC0, 0xAD,

0x94, 0x9D, 0x03, 0xFE, 0xE1, 0x39, 0x57, 0x9F,

0x72, 0xBA, 0x0C, 0x3D, 0x32, 0xC5, 0xF9, 0x54,

0xF1, 0xCC, 0xCD, 0x54, 0x08, 0x1F, 0xC7, 0x44,

0x02, 0x78, 0xCB, 0xA1, 0xFE, 0x7B, 0x7A, 0x17,

0xA9, 0x86, 0xFD, 0xFF, 0x5B, 0xD1, 0x5D, 0x1F,

};

CK\_BBOOL bTrue = CK\_TRUE;

CK\_BBOOL bFalse = CK\_FALSE;

CK\_KEY\_TYPE keyType = CKK\_GENERIC\_SECRET;

CK\_OBJECT\_CLASS keyObject = CKO\_SECRET\_KEY;

CK\_ATTRIBUTE secretKeyTemplate[] = {

{CKA\_CLASS, &keyObject, sizeof(keyObject)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &bFalse, sizeof(bTrue)},

{CKA\_VALUE, keyValue, sizeof(keyValue)},

{CKA\_DERIVE, &bTrue, sizeof(bTrue)},

{CKA\_EXTRACTABLE, &bTrue, sizeof(bTrue)},

};

CK\_TLS\_PRF\_PARAMS prfParams = { 0 };

CK\_ULONG outputLen = sizeof( ETALON );

CK\_BYTE outputBuf[sizeof( ETALON )];

CK\_MECHANISM prfMech = { CKM\_TLS\_GOST\_PRF\_2012\_512, &prfParams, sizeof( prfParams ) };

rv = pF->C\_CreateObject(hSession, secretKeyTemplate, sizeof(secretKeyTemplate) / sizeof(CK\_ATTRIBUTE), &keyHandle);

assert(rv == CKR\_OK);

prfParams.pSeed = seed;

prfParams.ulSeedLen = sizeof( seed );

prfParams.pLabel = label;

prfParams.ulLabelLen = sizeof( label );

prfParams.pOutput = outputBuf;

prfParams.pulOutputLen = &outputLen;

rv = pF->C\_DeriveKey( hSession, &prfMech, keyHandle, NULL, 0, NULL );

assert(rv == CKR\_OK && \*prfParams.pulOutputLen == sizeof(ETALON));

rv = pF->C\_DestroyObject( hSession, keyHandle );

assert(rv == CKR\_OK);

return memcmp(prfParams.pOutput, ETALON, sizeof(ETALON)) ? CKR\_FUNCTION\_FAILED : CKR\_OK;

}

# Appendix 9. Examples of using the GOST R 34.11-2012 hash algorithm in the PKCS #5 PBKDF2 key generation mechanism.

CK\_RV sample\_pbkdf\_stribog\_512(CK\_FUNCTION\_LIST\_PTR pF, CK\_SESSION\_HANDLE hSession)

{

CK\_RV rv = CKR\_OK;

const char salt[] = "saltSALTsalt";

const char password[] = "password";

CK\_ULONG password\_length = sizeof(password) - 1;

CK\_PKCS5\_PBKD2\_PARAMS2 params = {

CKZ\_SALT\_SPECIFIED,

(CK\_BYTE\_PTR)salt, sizeof(salt) - 1,

2048,

CKP\_PKCS5\_PBKD2\_HMAC\_GOSTR3411\_2012\_512,

NULL, 0,

(CK\_UTF8CHAR\_PTR)password, password\_length,

};

CK\_MECHANISM pbkdfMechanism = { CKM\_PKCS5\_PBKD2, &params, sizeof(CK\_PKCS5\_PBKD2\_PARAMS2) };

CK\_BBOOL bTrue = CK\_TRUE;

CK\_BBOOL bFalse = CK\_FALSE;

CK\_KEY\_TYPE keyType = CKK\_GOST28147;

CK\_OBJECT\_CLASS keyObject = CKO\_SECRET\_KEY;

CK\_ATTRIBUTE secretKeyTemplate[] = {

{CKA\_CLASS, &keyObject, sizeof(keyObject)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &bFalse, sizeof(bTrue)},

{CKA\_ENCRYPT, &bTrue, sizeof(bTrue)},

{CKA\_DECRYPT, &bTrue, sizeof(bTrue)},

{CKA\_EXTRACTABLE,&bTrue, sizeof(bTrue)},

};

CK\_OBJECT\_HANDLE secretKey;

CK\_BYTE keyValue[32];

CK\_ATTRIBUTE keyAttribute = {CKA\_VALUE, keyValue, sizeof(keyValue)};

const CK\_BYTE ETALON[] = {

0x96, 0x85, 0x54, 0x56, 0xF3, 0x1E, 0x87, 0xD8,

0xCA, 0x4F, 0x55, 0x62, 0x91, 0xDE, 0x76, 0x7C,

0x97, 0xEF, 0x3F, 0x59, 0x7E, 0x65, 0xBA, 0x86,

0x82, 0x70, 0xE9, 0x41, 0x24, 0xCF, 0x68, 0x24,

};

rv = pF->C\_GenerateKey(hSession, &pbkdfMechanism,

secretKeyTemplate, sizeof(secretKeyTemplate) / sizeof(CK\_ATTRIBUTE), &secretKey);

assert(rv == CKR\_OK);

rv = pF->C\_GetAttributeValue(hSession, secretKey, &keyAttribute, 1);

assert(rv == CKR\_OK);

rv = pF->C\_DestroyObject(hSession, secretKey);

assert(rv == CKR\_OK);

return memcmp(keyValue, ETALON, sizeof(keyValue)) ? CKR\_FUNCTION\_FAILED : CKR\_OK;

}

# Appendix 10. Examples of using the CKM\_GOSTR3410\_512\_KEY\_PAIR\_GEN mechanism for the key pair generation.

CK\_RV sample\_generate\_key\_pair\_512(CK\_FUNCTION\_LIST\_PTR pF, CK\_SESSION\_HANDLE hSession)

{

CK\_RV rv = CKR\_OK;

CK\_MECHANISM asymKeyMechanism512 = {CKM\_GOSTR3410\_512\_KEY\_PAIR\_GEN, 0, 0};

CK\_OBJECT\_HANDLE publicKey512 = CK\_INVALID\_HANDLE;

CK\_OBJECT\_HANDLE privateKey512 = CK\_INVALID\_HANDLE;

CK\_BBOOL bTrue = CK\_TRUE;

CK\_BBOOL bFalse = CK\_FALSE;

CK\_KEY\_TYPE keyType2012 = CKK\_GOSTR3410\_512;

CK\_OBJECT\_CLASS pubkeyClass = CKO\_PUBLIC\_KEY;

CK\_OBJECT\_CLASS privkeyClass = CKO\_PRIVATE\_KEY;

CK\_ATTRIBUTE PublicKey512Template[] = {

{CKA\_CLASS, &pubkeyClass, sizeof(pubkeyClass) },

{CKA\_KEY\_TYPE, &keyType2012, sizeof(keyType2012) },

{CKA\_TOKEN, &bFalse, sizeof(bFalse) },

{CKA\_PRIVATE, &bFalse, sizeof(bFalse) },

{CKA\_EXTRACTABLE, &bTrue, sizeof(bTrue) },

{CKA\_VERIFY, &bTrue, sizeof(bTrue) },

};

CK\_ATTRIBUTE PrivateKey512Template[] = {

{CKA\_CLASS, &privkeyClass, sizeof(privkeyClass) },

{CKA\_KEY\_TYPE, &keyType2012, sizeof(keyType2012) },

{CKA\_TOKEN, &bFalse, sizeof(bFalse) },

{CKA\_PRIVATE, &bTrue, sizeof(bTrue) },

{CKA\_EXTRACTABLE, &bTrue, sizeof(bTrue) },

{CKA\_SIGN, &bTrue, sizeof(bTrue) },

};

CK\_ATTRIBUTE keyAttribute = {CKA\_VALUE, NULL, 0};

rv = pF->C\_GenerateKeyPair(hSession, &asymKeyMechanism512,

PublicKey512Template, sizeof(PublicKey512Template) / sizeof(CK\_ATTRIBUTE),

PrivateKey512Template, sizeof(PrivateKey512Template) / sizeof(CK\_ATTRIBUTE),

&publicKey512, &privateKey512);

assert(rv == CKR\_OK);

rv = pF->C\_GetAttributeValue(hSession, publicKey512, &keyAttribute, 1);

assert(rv == CKR\_OK && keyAttribute.ulValueLen == 128);

rv = pF->C\_DestroyObject(hSession, publicKey512);

assert(rv == CKR\_OK);

rv = pF->C\_DestroyObject(hSession, privateKey512);

assert(rv == CKR\_OK);

return CKR\_OK;

}

# Appendix 11. Examples of using the CKM\_GOSTR3410\_PUBLIC\_KEY\_DERIVE mechanism for the public key derivation.

CK\_RV sample\_public\_derive(CK\_FUNCTION\_LIST\_PTR pF, CK\_SESSION\_HANDLE hSession)

{

CK\_RV rv = CKR\_OK;

CK\_OBJECT\_HANDLE publicKeyHandle = CK\_INVALID\_HANDLE;

CK\_OBJECT\_HANDLE privateKeyHandle = CK\_INVALID\_HANDLE;

CK\_MECHANISM deriveMechanism = {CKM\_GOSTR3410\_PUBLIC\_KEY\_DERIVE, 0, 0};

CK\_BYTE keyValue[] = {

0xD9, 0x2D, 0x43, 0x1D, 0x20, 0x37, 0x5C, 0xD2,

0xA5, 0x37, 0xCD, 0x64, 0x8E, 0x14, 0xB6, 0x0B,

0x4C, 0x21, 0xA1, 0x5A, 0x57, 0x98, 0x61, 0xB7,

0xBE, 0x41, 0x9B, 0x16, 0xED, 0x86, 0x18, 0x74,

};

CK\_BYTE gost3410\_defOid[] = { 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x01 };

CK\_BBOOL bTrue = CK\_TRUE;

CK\_BBOOL bFalse = CK\_FALSE;

CK\_KEY\_TYPE keyType = CKK\_GOSTR3410;

CK\_OBJECT\_CLASS privateClass = CKO\_PRIVATE\_KEY;

CK\_OBJECT\_CLASS publicClass = CKO\_PUBLIC\_KEY;

CK\_ATTRIBUTE privateKeyTemplate[] = {

{ CKA\_CLASS, &privateClass, sizeof(privateClass) },

{ CKA\_KEY\_TYPE, &keyType, sizeof(keyType) },

{ CKA\_TOKEN, &bFalse, sizeof(bFalse) },

{ CKA\_PRIVATE, &bTrue, sizeof(bTrue) },

{ CKA\_VALUE, keyValue, sizeof(keyValue) },

{ CKA\_SIGN, &bTrue, sizeof(bTrue) },

{ CKA\_DERIVE, &bTrue, sizeof(bTrue) },

{ CKA\_GOSTR3410\_PARAMS, gost3410\_defOid, sizeof(gost3410\_defOid) },

};

CK\_ATTRIBUTE publicKeyTemplate[] =

{

{ CKA\_CLASS, &publicClass, sizeof(publicClass)},

{ CKA\_KEY\_TYPE, &keyType, sizeof(keyType) },

{ CKA\_TOKEN, &bFalse, sizeof(bFalse) },

{ CKA\_PRIVATE, &bFalse, sizeof(bFalse) },

{ CKA\_VERIFY, &bTrue, sizeof(bTrue) },

};

const CK\_BYTE ETALON[] = {

0x03, 0x06, 0x54, 0xAC, 0xD1, 0x4A, 0xD8, 0x5D,

0x6B, 0x24, 0x6E, 0xC4, 0xA1, 0x95, 0xB3, 0x34,

0xEC, 0xFE, 0xF9, 0x3C, 0x1F, 0x22, 0xB6, 0x7C,

0xF8, 0x1F, 0xF7, 0xD3, 0x5E, 0x8D, 0xD6, 0x18,

0xE5, 0x38, 0xC3, 0xB3, 0x27, 0xE9, 0x3B, 0x13,

0x66, 0x97, 0xED, 0x5C, 0x86, 0x17, 0x3B, 0x44,

0x34, 0x1C, 0x5F, 0x5B, 0x97, 0x92, 0xE9, 0x53,

0x62, 0x17, 0x0A, 0x99, 0x3D, 0x84, 0xA4, 0x72,

};

CK\_BYTE publicKeyValue[64];

CK\_ATTRIBUTE keyAttribute = {CKA\_VALUE, publicKeyValue, sizeof(publicKeyValue)};

rv = pF->C\_CreateObject( hSession, privateKeyTemplate, sizeof( privateKeyTemplate ) / sizeof( CK\_ATTRIBUTE ), &privateKeyHandle );

assert(rv == CKR\_OK);

rv = pF->C\_DeriveKey( hSession, &deriveMechanism, privateKeyHandle, publicKeyTemplate, sizeof( publicKeyTemplate ) / sizeof( CK\_ATTRIBUTE ), &publicKeyHandle );

assert(rv == CKR\_OK);

rv = pF->C\_GetAttributeValue(hSession, publicKeyHandle, &keyAttribute, 1);

assert(rv == CKR\_OK);

rv = pF->C\_DestroyObject(hSession, publicKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_DestroyObject(hSession, privateKeyHandle);

assert(rv == CKR\_OK);

return memcmp(publicKeyValue, ETALON, sizeof(publicKeyValue)) ? CKR\_FUNCTION\_FAILED : CKR\_OK;

}

# Appendix 12. Examples of using mechanisms for the signature and verification processes according to GOST R 34.10-2012, 256 bit.

CK\_RV sample\_sign\_verify(CK\_FUNCTION\_LIST\_PTR pF, CK\_SESSION\_HANDLE hSession)

{

CK\_RV rv = CKR\_OK;

CK\_OBJECT\_HANDLE publicKeyHandle = CK\_INVALID\_HANDLE;

CK\_OBJECT\_HANDLE privateKeyHandle = CK\_INVALID\_HANDLE;

CK\_BYTE privateValue[] = {

0xD9, 0x2D, 0x43, 0x1D, 0x20, 0x37, 0x5C, 0xD2,

0xA5, 0x37, 0xCD, 0x64, 0x8E, 0x14, 0xB6, 0x0B,

0x4C, 0x21, 0xA1, 0x5A, 0x57, 0x98, 0x61, 0xB7,

0xBE, 0x41, 0x9B, 0x16, 0xED, 0x86, 0x18, 0x74,

};

CK\_BYTE gost3410\_defOid[] = { 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x01 };

CK\_BBOOL bTrue = CK\_TRUE;

CK\_BBOOL bFalse = CK\_FALSE;

CK\_KEY\_TYPE keyType = CKK\_GOSTR3410;

CK\_OBJECT\_CLASS privateClass = CKO\_PRIVATE\_KEY;

CK\_OBJECT\_CLASS publicClass = CKO\_PUBLIC\_KEY;

CK\_ATTRIBUTE privateKeyTemplate[] = {

{ CKA\_CLASS, &privateClass, sizeof(privateClass) },

{ CKA\_KEY\_TYPE, &keyType, sizeof(keyType) },

{ CKA\_TOKEN, &bFalse, sizeof(bFalse) },

{ CKA\_PRIVATE, &bTrue, sizeof(bTrue) },

{ CKA\_VALUE, privateValue, sizeof(privateValue) },

{ CKA\_SIGN, &bTrue, sizeof(bTrue) },

{ CKA\_DERIVE, &bTrue, sizeof(bTrue) },

{ CKA\_GOSTR3410\_PARAMS, gost3410\_defOid, sizeof(gost3410\_defOid) },

};

CK\_BYTE publicValue[] = {

0x03, 0x06, 0x54, 0xAC, 0xD1, 0x4A, 0xD8, 0x5D,

0x6B, 0x24, 0x6E, 0xC4, 0xA1, 0x95, 0xB3, 0x34,

0xEC, 0xFE, 0xF9, 0x3C, 0x1F, 0x22, 0xB6, 0x7C,

0xF8, 0x1F, 0xF7, 0xD3, 0x5E, 0x8D, 0xD6, 0x18,

0xE5, 0x38, 0xC3, 0xB3, 0x27, 0xE9, 0x3B, 0x13,

0x66, 0x97, 0xED, 0x5C, 0x86, 0x17, 0x3B, 0x44,

0x34, 0x1C, 0x5F, 0x5B, 0x97, 0x92, 0xE9, 0x53,

0x62, 0x17, 0x0A, 0x99, 0x3D, 0x84, 0xA4, 0x72,

};

CK\_ATTRIBUTE publicKeyTemplate[] =

{

{ CKA\_CLASS, &publicClass, sizeof(publicClass)},

{ CKA\_KEY\_TYPE, &keyType, sizeof(keyType) },

{ CKA\_TOKEN, &bFalse, sizeof(bFalse) },

{ CKA\_PRIVATE, &bFalse, sizeof(bFalse) },

{ CKA\_VALUE, publicValue, sizeof(publicValue) },

{ CKA\_VERIFY, &bTrue, sizeof(bTrue) },

{ CKA\_GOSTR3410\_PARAMS, gost3410\_defOid, sizeof(gost3410\_defOid) },

};

CK\_BYTE pangram[] = "The quick brown fox jumps over the lazy dog";

CK\_BYTE pangramDigest[] = {

0x3e, 0x7d, 0xea, 0x7f, 0x23, 0x84, 0xb6, 0xc5,

0xa3, 0xd0, 0xe2, 0x4a, 0xaa, 0x29, 0xc0, 0x5e,

0x89, 0xdd, 0xd7, 0x62, 0x14, 0x50, 0x30, 0xec,

0x22, 0xc7, 0x1a, 0x6d, 0xb8, 0xb2, 0xc1, 0xf4,

};

CK\_MECHANISM signOnlyMechanism = {CKM\_GOSTR3410\_256, 0, 0};

CK\_MECHANISM signWithHashMechanism = {CKM\_GOSTR3410\_WITH\_GOSTR3411\_2012\_256, 0, 0};

CK\_ULONG signatureLen = 0;

CK\_BYTE hashSignature[64];

CK\_BYTE dataSignature[64];

CK\_BYTE ETALON[] = {

0x68, 0x13, 0x4d, 0x22, 0xa3, 0xf3, 0xb0, 0x70,

0x7a, 0x85, 0xc9, 0xb8, 0x8f, 0xaf, 0x12, 0x9c,

0x1b, 0x83, 0xca, 0x26, 0x31, 0x1c, 0x1f, 0x47,

0xbd, 0x5f, 0xaa, 0x00, 0x13, 0x45, 0x45, 0x19,

0xcf, 0x17, 0x36, 0x16, 0x8b, 0xa1, 0xb7, 0x01,

0x48, 0xd8, 0x86, 0xf2, 0x67, 0x7c, 0xa4, 0xc6,

0x8e, 0xd9, 0xf2, 0xbe, 0x42, 0x4b, 0x25, 0x08,

0x40, 0x00, 0x08, 0x70, 0xd6, 0xd3, 0x98, 0xba,

};

rv = pF->C\_CreateObject( hSession, privateKeyTemplate, sizeof( privateKeyTemplate ) / sizeof( CK\_ATTRIBUTE ), &privateKeyHandle );

assert(rv == CKR\_OK);

rv = pF->C\_CreateObject( hSession, publicKeyTemplate, sizeof( publicKeyTemplate ) / sizeof( CK\_ATTRIBUTE ), &publicKeyHandle );

assert(rv == CKR\_OK);

rv = pF->C\_SignInit(hSession, &signOnlyMechanism, privateKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Sign(hSession, pangramDigest, sizeof(pangramDigest), NULL, &signatureLen);

assert(rv == CKR\_OK);

assert(signatureLen == 64);

rv = pF->C\_Sign(hSession, pangramDigest, sizeof(pangramDigest), hashSignature, &signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_SignInit(hSession, &signWithHashMechanism, privateKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Sign(hSession, pangram, sizeof(pangram) - 1, NULL, &signatureLen);

assert(rv == CKR\_OK);

assert(signatureLen == 64);

rv = pF->C\_Sign(hSession, pangram, sizeof(pangram) - 1, dataSignature, &signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_VerifyInit(hSession, &signOnlyMechanism, publicKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Verify(hSession, pangramDigest, sizeof(pangramDigest), hashSignature, signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_VerifyInit(hSession, &signOnlyMechanism, publicKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Verify(hSession, pangramDigest, sizeof(pangramDigest), dataSignature, signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_VerifyInit(hSession, &signOnlyMechanism, publicKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Verify(hSession, pangramDigest, sizeof(pangramDigest), ETALON, signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_VerifyInit(hSession, &signWithHashMechanism, publicKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Verify(hSession, pangram, sizeof(pangram) - 1, hashSignature, signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_VerifyInit(hSession, &signWithHashMechanism, publicKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Verify(hSession, pangram, sizeof(pangram) - 1, dataSignature, signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_VerifyInit(hSession, &signWithHashMechanism, publicKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Verify(hSession, pangram, sizeof(pangram) - 1, ETALON, signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_DestroyObject(hSession, publicKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_DestroyObject(hSession, privateKeyHandle);

assert(rv == CKR\_OK);

return CKR\_OK;

}

# Appendix 13. Examples of using mechanisms for the signature and verification processes according to GOST R 34.10-2012, 512 bit.

CK\_RV sample\_sign\_verify\_512(CK\_FUNCTION\_LIST\_PTR pF, CK\_SESSION\_HANDLE hSession)

{

CK\_RV rv = CKR\_OK;

CK\_OBJECT\_HANDLE publicKeyHandle = CK\_INVALID\_HANDLE;

CK\_OBJECT\_HANDLE privateKeyHandle = CK\_INVALID\_HANDLE;

CK\_BYTE privateValue[] = {

0xC2, 0x48, 0x02, 0x82, 0x70, 0xE0, 0xFF, 0x17,

0xD4, 0xDD, 0x9D, 0xA7, 0x19, 0xE2, 0xBD, 0xB6,

0xDF, 0x60, 0x17, 0x2B, 0xCB, 0xC1, 0x70, 0x9A,

0xBC, 0x4B, 0xAA, 0x80, 0xD2, 0xB6, 0x56, 0x9B,

0x69, 0xDC, 0xED, 0x7A, 0x02, 0x66, 0xAC, 0xE0,

0xA2, 0x64, 0x2C, 0xB4, 0x3A, 0x35, 0x87, 0x8F,

0x82, 0x5F, 0x30, 0x2F, 0x14, 0x63, 0xDE, 0xC0,

0xB7, 0x41, 0x33, 0xAF, 0x55, 0x81, 0x65, 0x40,

};

CK\_BYTE gost3410\_defOid[] = { 0x06, 0x09, 0x2a, 0x85, 0x03, 0x07, 0x01, 0x02, 0x01, 0x02, 0x01 }; //1.2.643.7.1.2.1.2.1

CK\_BBOOL bTrue = CK\_TRUE;

CK\_BBOOL bFalse = CK\_FALSE;

CK\_KEY\_TYPE keyType = CKK\_GOSTR3410\_512;

CK\_OBJECT\_CLASS privateClass = CKO\_PRIVATE\_KEY;

CK\_OBJECT\_CLASS publicClass = CKO\_PUBLIC\_KEY;

CK\_ATTRIBUTE privateKeyTemplate[] = {

{ CKA\_CLASS, &privateClass, sizeof(privateClass) },

{ CKA\_KEY\_TYPE, &keyType, sizeof(keyType) },

{ CKA\_TOKEN, &bFalse, sizeof(bFalse) },

{ CKA\_PRIVATE, &bTrue, sizeof(bTrue) },

{ CKA\_VALUE, privateValue, sizeof(privateValue) },

{ CKA\_SIGN, &bTrue, sizeof(bTrue) },

{ CKA\_DERIVE, &bTrue, sizeof(bTrue) },

{ CKA\_GOSTR3410\_PARAMS, gost3410\_defOid, sizeof(gost3410\_defOid) },

};

CK\_BYTE publicValue[] = {

0xf1, 0xa7, 0x56, 0x64, 0xfd, 0xa4, 0x27, 0x64,

0xe4, 0x9f, 0x0d, 0x73, 0xae, 0x95, 0x56, 0x65,

0xba, 0x6c, 0x27, 0x97, 0x2f, 0x8e, 0x79, 0x30,

0xe6, 0x77, 0x7f, 0xb8, 0xd1, 0xf7, 0xa5, 0xc8,

0x97, 0x4c, 0x5f, 0x15, 0xa5, 0x75, 0x94, 0x84,

0x53, 0x9c, 0x21, 0xea, 0x8b, 0x15, 0xba, 0x29,

0x02, 0x82, 0x54, 0x30, 0x72, 0xdf, 0x48, 0xea,

0x62, 0x32, 0x41, 0xf0, 0x21, 0xb5, 0x0e, 0xab,

0xb3, 0x34, 0x59, 0x11, 0x82, 0x83, 0x0c, 0xb6,

0x7c, 0x5a, 0x33, 0x9d, 0x53, 0x78, 0xf3, 0x42,

0x51, 0x8b, 0xeb, 0xcb, 0xa9, 0x49, 0x1e, 0xb6,

0xcf, 0xb9, 0x75, 0x51, 0x7f, 0x17, 0x4a, 0xab,

0x5b, 0x5d, 0x3b, 0xc1, 0x03, 0x61, 0x85, 0xa9,

0x25, 0x26, 0x9d, 0xca, 0x4a, 0xe5, 0xb3, 0xe9,

0x1e, 0x17, 0x3d, 0xda, 0xb1, 0x64, 0xfa, 0x98,

0x6d, 0x17, 0xd4, 0x1c, 0x3f, 0x33, 0x7b, 0x4c,

};

CK\_ATTRIBUTE publicKeyTemplate[] =

{

{ CKA\_CLASS, &publicClass, sizeof(publicClass)},

{ CKA\_KEY\_TYPE, &keyType, sizeof(keyType) },

{ CKA\_TOKEN, &bFalse, sizeof(bFalse) },

{ CKA\_PRIVATE, &bFalse, sizeof(bFalse) },

{ CKA\_VALUE, publicValue, sizeof(publicValue) },

{ CKA\_VERIFY, &bTrue, sizeof(bTrue) },

{ CKA\_GOSTR3410\_PARAMS, gost3410\_defOid, sizeof(gost3410\_defOid) },

};

CK\_BYTE pangram[] = "The quick brown fox jumps over the lazy dog";

CK\_BYTE pangramDigest[] = {

0xd2, 0xb7, 0x93, 0xa0, 0xbb, 0x6c, 0xb5, 0x90,

0x48, 0x28, 0xb5, 0xb6, 0xdc, 0xfb, 0x44, 0x3b,

0xb8, 0xf3, 0x3e, 0xfc, 0x06, 0xad, 0x09, 0x36,

0x88, 0x78, 0xae, 0x4c, 0xdc, 0x82, 0x45, 0xb9,

0x7e, 0x60, 0x80, 0x24, 0x69, 0xbe, 0xd1, 0xe7,

0xc2, 0x1a, 0x64, 0xff, 0x0b, 0x17, 0x9a, 0x6a,

0x1e, 0x0b, 0xb7, 0x4d, 0x92, 0x96, 0x54, 0x50,

0xa0, 0xad, 0xab, 0x69, 0x16, 0x2c, 0x00, 0xfe,

};

CK\_MECHANISM signOnlyMechanism = {CKM\_GOSTR3410\_512, 0, 0};

CK\_MECHANISM signWithHashMechanism = {CKM\_GOSTR3410\_WITH\_GOSTR3411\_2012\_512, 0, 0};

CK\_ULONG signatureLen = 0;

CK\_BYTE hashSignature[128];

CK\_BYTE dataSignature[128];

CK\_BYTE ETALON[] = {

0x38, 0x11, 0x87, 0x26, 0xe0, 0x05, 0xa3, 0x86,

0x7e, 0xe5, 0xd4, 0xa8, 0x89, 0x41, 0x8d, 0x41,

0x17, 0x66, 0x1b, 0x4d, 0xdc, 0x15, 0x95, 0x89,

0xb1, 0x45, 0xcf, 0x42, 0x49, 0x1c, 0xb9, 0xe5,

0xf6, 0x30, 0x69, 0x13, 0x55, 0x9b, 0x10, 0xd8,

0xa9, 0x0d, 0xee, 0xd6, 0x55, 0xf2, 0xbb, 0xff,

0x6c, 0xac, 0xa6, 0xcd, 0xea, 0xcc, 0x56, 0x67,

0x03, 0x52, 0xeb, 0xf2, 0x70, 0xee, 0x12, 0xba,

0x52, 0x42, 0x9f, 0x17, 0x3d, 0xd2, 0xd1, 0x02,

0x98, 0x4c, 0x67, 0xce, 0xea, 0xcb, 0xf3, 0x98,

0xcc, 0x17, 0x4f, 0x06, 0x7d, 0x4b, 0xeb, 0xf5,

0xe5, 0xe5, 0xf8, 0x6b, 0x19, 0x36, 0x95, 0x25,

0xb4, 0x2e, 0xca, 0x0a, 0xa8, 0xe3, 0x69, 0xa9,

0xe7, 0xd3, 0x86, 0x21, 0x0c, 0x7a, 0x25, 0x42,

0x2c, 0xff, 0x04, 0x3f, 0x9d, 0xe9, 0xe4, 0xef,

0xfb, 0x33, 0x70, 0xa4, 0x3e, 0xa9, 0x17, 0x7c,

};

rv = pF->C\_CreateObject( hSession, privateKeyTemplate, sizeof( privateKeyTemplate ) / sizeof( CK\_ATTRIBUTE ), &privateKeyHandle );

assert(rv == CKR\_OK);

rv = pF->C\_CreateObject( hSession, publicKeyTemplate, sizeof( publicKeyTemplate ) / sizeof( CK\_ATTRIBUTE ), &publicKeyHandle );

assert(rv == CKR\_OK);

rv = pF->C\_SignInit(hSession, &signOnlyMechanism, privateKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Sign(hSession, pangramDigest, sizeof(pangramDigest), NULL, &signatureLen);

assert(rv == CKR\_OK);

assert(signatureLen == 128);

rv = pF->C\_Sign(hSession, pangramDigest, sizeof(pangramDigest), hashSignature, &signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_SignInit(hSession, &signWithHashMechanism, privateKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Sign(hSession, pangram, sizeof(pangram) - 1, NULL, &signatureLen);

assert(rv == CKR\_OK);

assert(signatureLen == 128);

rv = pF->C\_Sign(hSession, pangram, sizeof(pangram) - 1, dataSignature, &signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_VerifyInit(hSession, &signOnlyMechanism, publicKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Verify(hSession, pangramDigest, sizeof(pangramDigest), hashSignature, signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_VerifyInit(hSession, &signOnlyMechanism, publicKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Verify(hSession, pangramDigest, sizeof(pangramDigest), dataSignature, signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_VerifyInit(hSession, &signOnlyMechanism, publicKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Verify(hSession, pangramDigest, sizeof(pangramDigest), ETALON, signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_VerifyInit(hSession, &signWithHashMechanism, publicKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Verify(hSession, pangram, sizeof(pangram) - 1, hashSignature, signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_VerifyInit(hSession, &signWithHashMechanism, publicKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Verify(hSession, pangram, sizeof(pangram) - 1, dataSignature, signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_VerifyInit(hSession, &signWithHashMechanism, publicKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_Verify(hSession, pangram, sizeof(pangram) - 1, ETALON, signatureLen);

assert(rv == CKR\_OK);

rv = pF->C\_DestroyObject(hSession, publicKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_DestroyObject(hSession, privateKeyHandle);

assert(rv == CKR\_OK);

return CKR\_OK;

}

# Appendix 14. Examples of using the CKM\_GOSTR3410\_2012\_DERIVE mechanism for the common secret key derivation.

CK\_RV sample\_dh\_2012(CK\_FUNCTION\_LIST\_PTR pF, CK\_SESSION\_HANDLE hSession)

{

CK\_RV rv = CKR\_OK;

CK\_OBJECT\_HANDLE aliceKeyHandle = CK\_INVALID\_HANDLE;

CK\_OBJECT\_HANDLE secretKeyHandle = CK\_INVALID\_HANDLE;

CK\_BYTE ukm[] = {1, 2, 3, 4, 5, 6, 7, 8};

CK\_BYTE aliceKeyValue[] = {

0xD9, 0x2D, 0x43, 0x1D, 0x20, 0x37, 0x5C, 0xD2,

0xA5, 0x37, 0xCD, 0x64, 0x8E, 0x14, 0xB6, 0x0B,

0x4C, 0x21, 0xA1, 0x5A, 0x57, 0x98, 0x61, 0xB7,

0xBE, 0x41, 0x9B, 0x16, 0xED, 0x86, 0x18, 0x74,

};

CK\_BYTE bobKeyValue[] = {

0x4F, 0xC5, 0xF5, 0x7A, 0xB0, 0x9A, 0xA6, 0xF0,

0xF7, 0x43, 0x3E, 0xDE, 0xFB, 0xB4, 0xBC, 0xBE,

0x43, 0x68, 0xD6, 0x4F, 0xCF, 0x5E, 0xC6, 0x94,

0x52, 0x98, 0x2C, 0xFA, 0xEF, 0x61, 0xFD, 0xC6,

0xAE, 0x37, 0x76, 0x4B, 0xC9, 0xF9, 0x10, 0x90,

0x59, 0x95, 0xE9, 0x23, 0x89, 0x53, 0x7F, 0xF3,

0xB6, 0x32, 0x93, 0x8A, 0x4A, 0x6B, 0x8E, 0x5D,

0x1B, 0xEE, 0x20, 0xDE, 0xE3, 0x71, 0xE2, 0x58,

};

CK\_BYTE ETALON[] = {

0x13, 0xC0, 0xBC, 0xD5, 0x1E, 0x44, 0x9B, 0x2C,

0x20, 0x5D, 0xF0, 0x5D, 0xFD, 0xCA, 0xCE, 0xE8,

0x54, 0xB6, 0xC4, 0xE9, 0xC5, 0x57, 0xF1, 0x3A,

0xCF, 0xF6, 0x03, 0x93, 0x59, 0x92, 0xF9, 0xC2,

};

CK\_BYTE gost3410\_defOid[] = { 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x01 };

CK\_BBOOL bTrue = CK\_TRUE;

CK\_BBOOL bFalse = CK\_FALSE;

CK\_KEY\_TYPE secretType = CKK\_GOST28147;

CK\_KEY\_TYPE privateType = CKK\_GOSTR3410;

CK\_OBJECT\_CLASS privateClass = CKO\_PRIVATE\_KEY;

CK\_OBJECT\_CLASS secretClass = CKO\_SECRET\_KEY;

CK\_ATTRIBUTE aliceKeyTemplate[] = {

{ CKA\_CLASS, &privateClass, sizeof(privateClass) },

{ CKA\_KEY\_TYPE, &privateType, sizeof(privateType) },

{ CKA\_TOKEN, &bFalse, sizeof(bFalse) },

{ CKA\_PRIVATE, &bTrue, sizeof(bTrue) },

{ CKA\_VALUE, aliceKeyValue, sizeof(aliceKeyValue) },

{ CKA\_DERIVE, &bTrue, sizeof(bTrue) },

{ CKA\_GOSTR3410\_PARAMS, gost3410\_defOid,sizeof(gost3410\_defOid) },

};

CK\_ULONG aliceKeyTemplateSize = sizeof( aliceKeyTemplate ) / sizeof( CK\_ATTRIBUTE );

CK\_ATTRIBUTE secretTemplate[] =

{

{ CKA\_CLASS, &secretClass, sizeof(secretClass)},

{ CKA\_KEY\_TYPE, &secretType, sizeof(secretType) },

{ CKA\_TOKEN, &bFalse, sizeof(bFalse) },

{ CKA\_PRIVATE, &bFalse, sizeof(bFalse) },

{ CKA\_EXTRACTABLE, &bTrue, sizeof(bTrue) },

};

CK\_ULONG secretTemplateSize = sizeof( secretTemplate ) / sizeof( CK\_ATTRIBUTE );

uint32\_t deriveParams[21] = {0};

CK\_MECHANISM deriveMechanism = { CKM\_GOSTR3410\_2012\_DERIVE, &deriveParams, sizeof( deriveParams ) };

CK\_BYTE secretKeyValue[32];

CK\_ATTRIBUTE keyAttribute = {CKA\_VALUE, secretKeyValue, sizeof(secretKeyValue)};

rv = pF->C\_CreateObject( hSession, aliceKeyTemplate, aliceKeyTemplateSize, &aliceKeyHandle );

assert(rv == CKR\_OK);

deriveParams[0] = CKD\_NULL;

deriveParams[1] = 64;

memcpy( deriveParams + 2, bobKeyValue, sizeof(bobKeyValue) );

deriveParams[18] = 8;

memcpy( deriveParams + 19, ukm, sizeof( ukm ) );

rv = pF->C\_DeriveKey( hSession, &deriveMechanism, aliceKeyHandle, secretTemplate, secretTemplateSize, &secretKeyHandle );

assert(rv == CKR\_OK);

rv = pF->C\_GetAttributeValue(hSession, secretKeyHandle, &keyAttribute, 1);

assert(rv == CKR\_OK);

rv = pF->C\_DestroyObject(hSession, aliceKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_DestroyObject(hSession, secretKeyHandle);

assert(rv == CKR\_OK);

return memcmp(secretKeyValue, ETALON, sizeof(ETALON)) ? CKR\_FUNCTION\_FAILED : CKR\_OK;

}

# Appendix 15. Examples of using the CKM\_GOSTR3410\_2012\_DERIVE mechanism gor the common secret key derivation.

CK\_RV sample\_dh\_2012\_512(CK\_FUNCTION\_LIST\_PTR pF, CK\_SESSION\_HANDLE hSession)

{

CK\_RV rv = CKR\_OK;

CK\_OBJECT\_HANDLE aliceKeyHandle = CK\_INVALID\_HANDLE;

CK\_OBJECT\_HANDLE secretKeyHandle = CK\_INVALID\_HANDLE;

CK\_BYTE ukm[] = {1, 2, 3, 4, 5, 6, 7, 8};

CK\_BYTE aliceKeyValue[] = {

0xC2, 0x48, 0x02, 0x82, 0x70, 0xE0, 0xFF, 0x17,

0xD4, 0xDD, 0x9D, 0xA7, 0x19, 0xE2, 0xBD, 0xB6,

0xDF, 0x60, 0x17, 0x2B, 0xCB, 0xC1, 0x70, 0x9A,

0xBC, 0x4B, 0xAA, 0x80, 0xD2, 0xB6, 0x56, 0x9B,

0x69, 0xDC, 0xED, 0x7A, 0x02, 0x66, 0xAC, 0xE0,

0xA2, 0x64, 0x2C, 0xB4, 0x3A, 0x35, 0x87, 0x8F,

0x82, 0x5F, 0x30, 0x2F, 0x14, 0x63, 0xDE, 0xC0,

0xB7, 0x41, 0x33, 0xAF, 0x55, 0x81, 0x65, 0x40,

};

CK\_BYTE bobKeyValue[] = {

0x73, 0x50, 0x68, 0x39, 0x15, 0x51, 0x22, 0x45,

0x0C, 0x15, 0x30, 0x88, 0xCD, 0xA0, 0x1A, 0xBC,

0xBE, 0xA0, 0x4D, 0x9B, 0x3F, 0xCC, 0xB1, 0xB6,

0x95, 0xF9, 0x49, 0x63, 0x0D, 0x02, 0xEF, 0xFE,

0x0D, 0xA2, 0xC2, 0xCB, 0x84, 0x88, 0x43, 0x7B,

0x05, 0x03, 0xB3, 0x31, 0x43, 0x0E, 0xD1, 0x6D,

0xFF, 0xB9, 0x11, 0x1D, 0x44, 0xF1, 0x35, 0x23,

0xF1, 0x38, 0x5B, 0x79, 0x03, 0x17, 0xB8, 0xEE,

0x9B, 0x2E, 0xC2, 0x56, 0x6F, 0x78, 0xD6, 0xB1,

0xF7, 0xDB, 0xD7, 0xC8, 0xBE, 0x89, 0x33, 0x8D,

0x72, 0xD4, 0x1E, 0x4A, 0x60, 0x11, 0x0E, 0x16,

0x4A, 0x01, 0x3C, 0x52, 0xBF, 0xF5, 0x8C, 0x9B,

0x83, 0xB8, 0xDB, 0x64, 0xFB, 0xA1, 0xE7, 0x03,

0x64, 0xD5, 0x26, 0xF6, 0x79, 0x3C, 0x4E, 0x35,

0x5F, 0x58, 0x87, 0x69, 0x59, 0x28, 0xFA, 0x1F,

0xCC, 0x20, 0x0F, 0x42, 0x46, 0x55, 0xF3, 0x95,

};

CK\_BYTE ETALON[] = {

0xef, 0xc6, 0x4a, 0x95, 0x35, 0x7b, 0xb7, 0x21,

0x57, 0x6f, 0x25, 0xbd, 0x2a, 0xb9, 0x22, 0xf1,

0x16, 0x69, 0xf3, 0xb1, 0xa2, 0x32, 0xd4, 0x7b,

0xae, 0xb9, 0x2a, 0xaf, 0xa6, 0x10, 0x25, 0x64,

};

CK\_BYTE gost3410\_defOid[] = { 0x06, 0x09, 0x2a, 0x85, 0x03, 0x07, 0x01, 0x02, 0x01, 0x02, 0x01 }; //1.2.643.7.1.2.1.2.1

CK\_BBOOL bTrue = CK\_TRUE;

CK\_BBOOL bFalse = CK\_FALSE;

CK\_KEY\_TYPE secretType = CKK\_GOST28147;

CK\_KEY\_TYPE privateType = CKK\_GOSTR3410\_512;

CK\_OBJECT\_CLASS privateClass = CKO\_PRIVATE\_KEY;

CK\_OBJECT\_CLASS secretClass = CKO\_SECRET\_KEY;

CK\_ATTRIBUTE aliceKeyTemplate[] = {

{ CKA\_CLASS, &privateClass, sizeof(privateClass) },

{ CKA\_KEY\_TYPE, &privateType, sizeof(privateType) },

{ CKA\_TOKEN, &bFalse, sizeof(bFalse) },

{ CKA\_PRIVATE, &bTrue, sizeof(bTrue) },

{ CKA\_VALUE, aliceKeyValue, sizeof(aliceKeyValue) },

{ CKA\_DERIVE, &bTrue, sizeof(bTrue) },

{ CKA\_GOSTR3410\_PARAMS, gost3410\_defOid,sizeof(gost3410\_defOid) },

};

CK\_ULONG aliceKeyTemplateSize = sizeof( aliceKeyTemplate ) / sizeof( CK\_ATTRIBUTE );

CK\_ATTRIBUTE secretTemplate[] =

{

{ CKA\_CLASS, &secretClass, sizeof(secretClass)},

{ CKA\_KEY\_TYPE, &secretType, sizeof(secretType) },

{ CKA\_TOKEN, &bFalse, sizeof(bFalse) },

{ CKA\_PRIVATE, &bFalse, sizeof(bFalse) },

{ CKA\_EXTRACTABLE, &bTrue, sizeof(bTrue) },

};

CK\_ULONG secretTemplateSize = sizeof( secretTemplate ) / sizeof( CK\_ATTRIBUTE );

uint32\_t deriveParams[37] = {0};

CK\_MECHANISM deriveMechanism = { CKM\_GOSTR3410\_2012\_DERIVE, &deriveParams, sizeof( deriveParams ) };

CK\_BYTE secretKeyValue[32];

CK\_ATTRIBUTE keyAttribute = {CKA\_VALUE, secretKeyValue, sizeof(secretKeyValue)};

rv = pF->C\_CreateObject( hSession, aliceKeyTemplate, aliceKeyTemplateSize, &aliceKeyHandle );

assert(rv == CKR\_OK);

deriveParams[0] = CKD\_NULL;

deriveParams[1] = 128;

memcpy( deriveParams + 2, bobKeyValue, sizeof(bobKeyValue) );

deriveParams[34] = 8;

memcpy( deriveParams + 35, ukm, sizeof( ukm ) );

rv = pF->C\_DeriveKey( hSession, &deriveMechanism, aliceKeyHandle, secretTemplate, secretTemplateSize, &secretKeyHandle );

assert(rv == CKR\_OK);

rv = pF->C\_GetAttributeValue(hSession, secretKeyHandle, &keyAttribute, 1);

assert(rv == CKR\_OK);

rv = pF->C\_DestroyObject(hSession, aliceKeyHandle);

assert(rv == CKR\_OK);

rv = pF->C\_DestroyObject(hSession, secretKeyHandle);

assert(rv == CKR\_OK);

return memcmp(secretKeyValue, ETALON, sizeof(ETALON)) ? CKR\_FUNCTION\_FAILED : CKR\_OK;

}