127 018, Москва, Сущевский Вал, 18
Телефон: (495) 995 4820
Факс: (495) 995 4820
http://www.CryptoPro.ru
E-mail: info@CryptoPro.ru

Средство Криптографической Защиты Информации

КриптоПро CSP

Версия 5.0 КС1

Исполнение 1-Java

Руководство программиста

ЖТЯИ.00087-01 90 09

Листов 91

2018

Оглавление

1.Введение....7

2.Использование основной функциональности криптопровайдера КриптоПро JCSP через стандартный интерфейс JCA....9

2.1.Генерация ключевых пар электронной подписи и ключевого обмена в соответствии со стандартами ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012....9

2.1.1.Создание объекта генерации ключевых пар электронной подписи и ключевого обмена....9

2.1.2.Определение параметров генерации ключевой пары электронной подписи и ключевого обмена....10

2.2.Хэширование данных в соответствии с алгоритмом ГОСТ Р 34.11-94 и ГОСТ Р 34.11-2012....11

2.2.1.Создание объекта хэширования данных....11

2.2.2.Определение параметров хэширования данных....12

2.2.3.Копирование объекта хэширования данных....12

2.2.4.Вычисление хэша данных в соответствии с алгоритмами ГОСТ Р 34.11-94 и ГОСТ Р 34.11-2012....13

2.2.4.1.Обработка хэшируемых данных....13

2.2.4.2.Завершение операции хэширования....14

2.3.Формирование электронной подписи в соответствии с алгоритмами ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012....14

2.3.1.Создание объекта формирования электронной подписи....15

2.3.2.Инициализация объекта формирования электронной подписи....17

2.3.3.Определение параметров формирования электронной подписи....17

2.3.4.Формирование электронной подписи....17

2.3.4.1.Обработка подписываемых данных....17

2.3.4.2.Вычисление значения электронной подписи....18

2.4.Проверка электронной подписи в соответствии с алгоритмом ГОСТ Р 34.10-2012....18

2.4.1.Создание объекта проверки электронной подписи....18

2.4.2.Инициализация объекта проверки электронной подписи....19

2.4.3.Определение параметров проверки электронной подписи....19

2.4.4.Проверка электронной подписи....19

2.4.4.1.Обработка подписанных данных....19

2.4.4.2.Проверка электронной подписи....19

2.5.Создание ключей парной связи с помощью алгоритма ключевого обмена....20

2.5.1.Создание объекта генерации ключей парной связи....20

2.5.2.Инициализация генератора ключей парной связи....21

2.5.3.Выполнение фазы согласования ключей....21

2.5.4.Генерация ключа парной связи....21

2.6.Работа с ключевыми носителями....22

2.6.1.Запись ключей электронной подписи и ключевого обмена с алгоритмами ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012 на ключевые носители....22

2.6.1.1.Определение типа используемого ключевого носителя....22

2.6.1.2.Загрузка содержимого ключевого носителя....23

2.6.1.3.Запись ключа на носитель....24

2.6.2.Чтение ключей электронной подписи и ключевого обмена с алгоритмами ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012 с ключевых носителей....25

2.6.3.Запись сертификата открытого ключа на ключевой носитель в соответствии с хранящимся на нем ключом....26

2.6.4.Чтение сертификата открытого ключа с ключевого носителя....26

2.6.5.Удаление секретного ключа с ключевого носителя....27

2.7.Работа с хранилищем доверенных сертификатов....27
2.7.1.Запись сертификатов в хранилище доверенных сертификатов....27
2.7.1.1.Инициализация хранилища доверенных сертификатов....27
2.7.1.2.Загрузка содержимого хранилища....28
2.7.1.3.Запись сертификата в хранилище....28
2.7.1.4.Сохранение содержимого хранилища....28
2.7.2.Чтение сертификатов из хранилища доверенных сертификатов....28
2.8.Работа с симметричными ключами шифрования, соответствующими алгоритму ГОСТ 28147-89....29
2.8.1.Создание объекта генерации симметричных ключей шифрования....29
2.8.2.Определение параметров генерации симметричных ключей шифрования....29
2.8.3.Генерация симметричного ключа шифрования....30
2.9.Имитозащита данных в соответствии с алгоритмом ГОСТ 28147-89....30
2.9.1.Создание объекта имитозащиты данных....30
2.9.2.Инициализация объекта имитозащиты данных и определение его параметров....30
2.9.3.Копирование объекта имитозащиты данных....31
2.9.4.Выработка имитовставки данных....31
2.9.4.1.Обработка защищаемых данных....31
2.9.4.2.Вычисление значения имитовставки....32
2.10.Использование алгоритма HMAC....32
2.10.1.Использование 512-битных ключей в алгоритмах HMAC....33
2.11.Шифрование данных и ключей в соответствии с алгоритмом ГОСТ 28147-89....33
2.11.1.Создание объекта шифрования данных и ключей (шифратора)....33
2.11.2.Инициализация шифратора и определение его параметров....35
2.11.3.Зашифрование и расшифрование данных....37
2.11.3.1.Последовательное зашифрование (расшифрование) данных....38
2.11.3.2.Завершение операции зашифрования (расшифрования)....38
2.11.4.Зашифрование и расшифрование ключей....39
2.11.4.1.Зашифрование ключа....39
2.11.4.2.Расшифрование ключа....39
2.12.Диверсификация ключей....39
2.13.Генерация случайных чисел....40
2.13.1.Создание генератора случайных чисел....40
2.13.2.Использование генератора случайных чисел....40
2.13.3.Доинициализация датчика....40
2.13.4.Возможные ошибки датчика....40
2.13.5.Биологический датчик....40
2.14.Работа с сертификатами через стандартный интерфейс JCA....41
2.14.1.Генерация X509-сертификатов....41
2.14.2.Кодирование сертификата в DER-кодировку....42
2.14.3.Получение открытого ключа из сертификата....42
2.14.4.Построение и проверка цепочки сертификатов....42
2.14.4.1.Совместимость с КриптоПро УЦ при проверке цепочки сертификатов....42
2.14.4.2.Проверка цепочки сертификатов с использованием OCSP....43
3.Работа с параметрами в криптопровайдере КриптоПро JCSP....44
3.1.Работа с набором параметров для генерации ключей электронной подписи и ключевого обмена....44

3.2.Работа с параметрами алгоритмов подписи/ключевого обмена ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012....46
3.3.Работа с параметрами алгоритма хэширования ГОСТ Р 34.11-94 и ГОСТ Р 34.11-2012....46
3.4.Работа с параметрами алгоритма шифрования ГОСТ 28147-89....47
4.Дополнительные возможности работы с сертификатами....48
4.1.Инициализация генератора запросов и сертификатов....48
4.2.Генерация запроса на сертификат....50
4.2.1.Определение параметров открытого ключа субъекта....50
4.2.2.Определение имени субъекта....50
4.2.3.Кодирование и подпись запроса....50
4.2.4.Печать подписанного запроса....51
4.3.Отправка запроса центру сертификации и получение соответствующего запросу сертификата от центра....51
4.3.1.Получение сертификата непосредственно после генерации запроса....51
4.3.2.Получение сертификата из запроса, представленного в DER-кодировке....51
4.3.3.Получение сертификата из запроса, представленного в BASE64-кодировке....52
4.3.4.Получение корневого сертификата центра сертификации....52
4.4.Генерация самоподписанного сертификата....53
5.Дополнительные возможности работы с сертификатами для УЦ 1.5....54
5.1.Получение набора параметров для регистрации пользователя....54
5.2.Регистрация пользователя, получение токена и пароля и проверка статуса....55
5.3.Получение списка корневых сертификатов УЦ....56
5.4.Получение списка запросов на сертификаты пользователя....56
5.5.Генерация запроса на сертификат, проверка статуса сертификата и получение соответствующего запросу сертификата....56
6.Дополнительные возможности работы с сертификатами для УЦ 2.0....59
6.1.Получение набора параметров для регистрации пользователя в УЦ 2.0....60
6.2.Регистрация пользователя, получение токена и пароля и проверка статуса....60
6.3.Получение списка корневых сертификатов УЦ 2.0....61
6.4.Получение списка запросов на сертификаты пользователя....61
6.5.Подтверждение факта установки сертификата пользователя и авторизация по токену и паролю или сертификату пользователя....62
6.6.Генерация запроса на сертификат, проверка статуса сертификата и получение соответствующего запросу сертификата....63
6.7.Получение списка шаблонов сертификатов УЦ 2.0....65
6.8.Получение списка запросов на отзыв сертификатов....65
7.Работа с электронной подписью для XML-документов....66
8.КриптоПро Java CSP и Cryptographic Message Syntax (CMS)....69
8.1.Использование утилиты ComLine....69
8.2.Особенности использования с другими модулями....69
9.Использование библиотеки CAdES.jar для создания, проверки и усовершенствования подписи формата CAdES-BES, CAdES-T и CAdES-X Long Type 1....70
10.Использование библиотеки XAdES.jar для создания и проверки подписи формата XadES-BES, XadES-T и XadES-X Long Type 1....77
11.Использование утилиты keytool....81

11.1.Просмотр содержимого ключевого носителя....81
11.2.Генерация ключа и соответствующего ему самоподписанного сертификата и запись их на носитель....81
11.3.Генерация ключевой пары запись ее на носитель....82
11.4.Генерация запроса на сертификат ключа проверки электронной подписи в соответствии с хранящимся на носителе ключом электронной подписи и запись запроса в файл....83
11.5.Генерация самоподписанного сертификата ключа проверки электронной подписи в соответствии с хранящимся на носителе ключом электронной подписи и запись сертификата на носитель....83
11.6.Чтение сертификата ключа проверки электронной подписи с носителя и запись его в файл....84
11.7.Чтение сертификата ключа проверки электронной подписи из файла и запись его на носитель в соответствии с хранящимся на носителе ключом электронной подписи....84
11.8.Чтение доверенного сертификата из хранилища и запись его в файл....85
11.9.Чтение доверенного сертификата из файла и запись его в хранилище....85
11.10.Удаление ключа и соответствующего ему самоподписанного сертификата с носителя...86
11.11.Удаление доверенного сертификата из хранилища....86
12.Использование утилиты ComLine....87
12.1.Проверка установки и настроек провайдеров....87
12.2.Проверка работоспособности провайдеров....87
12.3.Работа с ключами и сертификатами....87
12.3.1.Генерация ключевой пары и соответствующего ей самоподписанного сертификата. Запись их на носитель. Генерация запроса на сертификат и запись его в файл....87
12.3.2.Получение сертификата из запроса. Запись сертификата в хранилище и в файл....88
12.3.3.Построение цепочки сертификатов....89
12.3.4.Формирование электронной подписи....89
12.3.5.Проверка электронной подписи....90
12.4.Использование КриптоПро JTLS....90
12.4.1.Запуск сервера из командной строки....90
12.4.2.Запуск клиента из командной строки....91
12.4.3.Запуск клиента нагрузочного примера из командной строки (samples.jar/JTLS_samples/HighLoadExample)....91
12.4.4.Запуск клиента на основе apache http client 4.x из командной строки (samples.jar/JTLS_samples/ApacheHttpClient4XExample)....93

1. Введение

Настоящее руководство содержит описание криптопровайдера КриптоПро JCSP (основной класс провайдера ru.CryptoPro.JCSP.JCSP).

Криптопровайдер КриптоПро JCSP является средством криптографической защиты информации (СКЗИ КриптоПро CSP 5.0 КС1 исполнение 1-Java), реализующим российские криптографические алгоритмы и функционирующим под управлением виртуальной машины Java 2 Runtime Environment версии 1.7 и 1.8, соответствующей спецификации Sun Java 2 ™ Virtual Machine.

Криптопровайдер КриптоПро JCSP должен использоваться с сертифицированными SUN Java-машинами, соответствующим требованиям безопасности SUN. Защищенность криптографических объектов, создаваемых и обрабатываемых криптопровайдером, зависит от степени защищенности и корректности Java-машины, и может быть снижена при использовании виртуальных машин, не имеющих сертификата SUN. Список сертифицированных Java-машин находится на сайте SUN по адресу: http://www.oracle.com/technetwork/indexes/downloads/index.html

Криптопровайдер КриптоПро JCSP реализует стандартный интерфейс Java Cryptography Architecture (JCA) в соответствии с российскими криптографическими алгоритмами и в соответствии с этим интерфейсом обеспечивает выполнение следующих операций:

- генерация ключей электронной подписи (256 бит) и ключей проверки электронной подписи (512 бит) в соответствии с алгоритмом ГОСТ Р 34.10-2001;
- генерация секретных (256 бит) и открытых (512 бит) ключей алгоритма ключевого обмена в соответствии с алгоритмом ГОСТ Р 34.10-2001;
- генерация ключей электронной подписи (256 бит) и ключей проверки электронной подписи (512 бит) в соответствии с алгоритмом ГОСТ Р 34.10-2012 (256 бит);
- генерация секретных (256 бит) и открытых (512 бит) ключей алгоритма ключевого обмена в соответствии с алгоритмом ГОСТ Р 34.10-2012;
- генерация ключей электронной подписи (512 бит) и ключей проверки электронной подписи (1024 бит) в соответствии с алгоритмом ГОСТ Р 34.10-2012 (512 бит);
- генерация секретных (512 бит) и открытых (1024 бит) ключей алгоритма ключевого обмена в соответствии с алгоритмом ГОСТ Р 34.10-2012;
- запись ключей электронной подписи и ключевого обмена с алгоритмом ГОСТ Р 34.10-2001 на носители (интерфейс хранилища ключей JCA);
- чтение ключей электронной подписи и ключевого обмена с алгоритмом ГОСТ Р 34.10-2001 с перечисленных носителей (интерфейс хранилища ключей JCA);
- запись ключей электронной подписи и ключевого обмена с алгоритмом ГОСТ Р 34.10-2012 на носители (интерфейс хранилища ключей JCA);
- чтение ключей электронной подписи и ключевого обмена с алгоритмом ГОСТ Р 34.10-2012 с перечисленных носителей (интерфейс хранилища ключей JCA);
- запись сертификата ключа проверки электронной подписи и ключевого обмена на ключевой носитель в соответствии с хранящимся на носителе ключом электронной подписи (интерфейс хранилища ключей JCA);
- чтение сертификатов ключей проверки электронной подписи и ключевого обмена с ключевых носителей (интерфейс хранилища ключей JCA);
- запись доверенных корневых сертификатов в стандартное хранилище JCA и чтение из него;
- генерация ключей с различными параметрами в соответствии с ГОСТ Р 34.10-2001;
- генерация ключей с различными параметрами в соответствии с ГОСТ Р 34.10-2012;
- хэширование данных с различными параметрами в соответствии с ГОСТ Р 34.11-94;

- хэширование данных в соответствии с ГОСТ Р 34.11-2012;
- формирование электронной подписи с различными параметрами в соответствии с ГОСТ Р 34.10-2001;
- формирование электронной подписи с различными параметрами в соответствии с ГОСТ Р 34.10-2012;
- проверка электронной подписи с различными параметрами в соответствии с ГОСТ Р 34.10-2001;
- проверка электронной подписи с различными параметрами в соответствии с ГОСТ Р 34.10-2012;
- осуществление выработки ключей парной связи с использованием алгоритма ключевого обмена с ключами ГОСТ Р 34.10-2001;
- осуществление выработки ключей парной связи с использованием алгоритма ключевого обмена с ключами ГОСТ Р 34.10-2012;
- генерация секретных ключей в соответствии с ГОСТ 28147-89;
- генерация секретных ключей длиной 512 бит для использования в функциях HMAC, основанных на использовании хэш-функций ГОСТ Р 34.11-2012 (256 бит) и ГОСТ Р 34.11-2012 (512 бит);
- шифрование данных в соответствии с алгоритмом ГОСТ 28147-89;
- шифрование секретных ключей ГОСТ 28147-89;
- выработка имитовставки в соответствии с алгоритмом ГОСТ 28147-89;
- выработка значений функций HMAC, основанных на использовании хэш-функций ГОСТ Р 34.11-94, ГОСТ Р 34.11-2012 (256 бит) и ГОСТ Р 34.11-2012 (512 бит);
- выработка случайных байтовых последовательностей.

Помимо перечисленных операций, осуществляемых в соответствии со стандартным интерфейсом JCA, модули криптопровайдера КриптоПро JCP, устанавливаемые совместно с криптопровайдером КриптоПро JCSP, предоставляют дополнительные возможности работы с сертификатами:

- генерация запроса на сертификат;
- отправка запроса серверу и получение от сервера соответствующего запросу сертификата;
- генерация самоподписанных сертификатов.

Основные технические данные и характеристики СКЗИ, а также информацию о совместимости с другими продуктами КриптоПро см. в «Руководстве администратора безопасности».

2. Использование основной функциональности криптопровайдера КриптоПро JCSP через стандартный интерфейс JCA

2.1.Генерация ключевых пар электронной подписи и ключевого обмена в соответствии со стандартами ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012

2.1.1.Создание объекта генерации ключевых пар электронной подписи и ключевого обмена

Для создания объекта генератора ключевой пары электронной подписи в соответствии с алгоритмом ГОСТ Р 34.10-2001 статическому методу *getInstance()* класса KeyPairGenerator необходимо в качестве параметра передать имя, идентифицирующее данный алгоритм ("GOST3410EL" или JCP.GOST_EL_DEGREE_NAME). При таком вызове метода *getInstance()* совместно с определением требуемого алгоритма генерации ключевой пары необходимо передавать имя криптопровайдера, используемого для выполнения требуемой операции, то есть JCSP.PROVIDER_NAME. Таким образом, создание генератора ключевой пары электронной подписи осуществляется одним из следующих способов:

```
KeyPairGenerator kg = KeyPairGenerator.getInstance("GOST3410EL", "JCSP");
```

```
KeyPairGenerator kg = KeyPairGenerator.getInstance(JCP.GOST_EL_DEGREE_NAME, JCSP.PROVIDER_NAME);
```

Для создания генератора ключевой пары электронной подписи в соответствии с алгоритмом ГОСТ Р 34.10-2012 (256 бит) методу *getInstance()* необходимо в качестве параметра передать имя, идентифицирующее данный алгоритм ("GOST3410_2012_256" или JCP.GOST_EL_2012_256_NAME). При таком вызове метода *getInstance()* совместно с определением требуемого алгоритма генерации ключевой пары необходимо передавать имя криптопровайдера, используемого для выполнения требуемой операции, то есть JCSP.PROVIDER_NAME. Таким образом, создание генератора ключевой пары электронной подписи осуществляется одним из следующих способов (на примере ГОСТ Р 34.10-2012 (256 бит)):

```
KeyPairGenerator kg = KeyPairGenerator.getInstance("GOST3410_2012_256", "JCSP");
```

```
KeyPairGenerator kg = KeyPairGenerator.getInstance(JCP.GOST_EL_2012_256_NAME, JCSP.PROVIDER_NAME);
```

Для создания генератора ключевой пары электронной подписи в соответствии с алгоритмом ГОСТ Р 34.10-2012 (512 бит) методу *getInstance()* необходимо в качестве параметра передать имя, идентифицирующее данный алгоритм ("GOST3410_2012_512" или JCP.GOST_EL_2012_512_NAME). При таком вызове метода *getInstance()* совместно с определением требуемого алгоритма генерации ключевой пары необходимо передавать имя криптопровайдера, используемого для выполнения требуемой операции, то есть JCSP.PROVIDER_NAME. Таким образом, создание генератора ключевой пары электронной подписи осуществляется одним из следующих способов:

```
KeyPairGenerator kg = KeyPairGenerator.getInstance("GOST3410_2012_512", "JCSP");
```

```
KeyPairGenerator kg = KeyPairGenerator.getInstance(JCP.GOST_EL_2012_512_NAME, JCSP.PROVIDER_NAME);
```

Для генерации ключевой пары ключевого обмена, соответствующей ГОСТ Р 34.10-2001, необходимо при создании объекта генератора в качестве имени, идентифицирующего требуемые алгоритмы, указывать имя "GOST3410DHEL" или JCP.GOST_EL_DH_NAME. В случае же генерации ключевой пары ключевого обмена, соответствующей ГОСТ Р 34.10-2012 (256) или ГОСТ Р 34.10-2012 (512), необходимо указывать имя "GOST3410DH_2012_256" или JCP.GOST_DH_2012_256_NAME, или "GOST3410DH_2012_512" или JCP.GOST_DH_2012_512_NAME. Таким образом, в криптопровайдере КриптоПро JCSP генератор ключевой пары ключевого обмена, соответствующей алгоритмам обмена Диффи-Хеллмана и подписи ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012 производится одним из следующих способов:

```
KeyPairGenerator kg = KeyPairGenerator.getInstance("GOST3410DHEL");
```

```
KeyPairGenerator kg = KeyPairGenerator.getInstance("GOST3410DHEL", "JCSP");
```

```
KeyPairGenerator kg = KeyPairGenerator.getInstance(JCP.GOST_EL_DH_NAME,
JCSP.PROVIDER_NAME);

KeyPairGenerator kg = KeyPairGenerator.getInstance("GOST3410DH_2012_256");

KeyPairGenerator kg = KeyPairGenerator.getInstance("GOST3410DH_2012_256", "JCSP");

KeyPairGenerator kg = KeyPairGenerator.getInstance(JCP.GOST_DH_2012_256_NAME,
JCSP.PROVIDER_NAME);

KeyPairGenerator kg = KeyPairGenerator.getInstance("GOST3410DH_2012_512");

KeyPairGenerator kg = KeyPairGenerator.getInstance("GOST3410DH_2012_512", "JCSP");

KeyPairGenerator kg = KeyPairGenerator.getInstance(JCP.GOST_DH_2012_512_NAME,
JCSP.PROVIDER_NAME);
```

Отличительной особенностью JCSP является использование биологического датчика случайных чисел криптопровайдера КриптоПро CSP при генерации долговременных ключевых пар электронной подписи и ключевого обмена. Генерация ключевой пары может происходить по одному из двух сценариев: создаётся временный контейнер, который копируется в рабочий, или сразу создаётся рабочий контейнер.

2.1.2.Определение параметров генерации ключевой пары электронной подписи и ключевого обмена

Возможны следующие способы подачи параметров инициализации:

1) стандартный (на примере ГОСТ Р 34.10-2001)

```
KeyPairGenerator kg = KeyPairGenerator.getInstance(JCP.GOST_EL_DEGREE_NAME,
JCSP.PROVIDER_NAME);

KeyPair keyPair = kg.generateKeyPair();
```

В этом случае происходит создание временного контейнера с неким произвольным паролем и именем (алиасом), для выработки ключа потребуется набрать достаточную энтропию в стандартном окне БиоДСЧ криптопровайдера КриптоПро CSP.

2) с заданием алиаса контейнера (на примере ГОСТ Р 34.10-2001)

```
KeyPairGenerator kg = KeyPairGenerator.getInstance(JCP.GOST_EL_DEGREE_NAME,
JCSP.PROVIDER_NAME);

String container =
KeyStoreConfig.getHDImage().makeContainerName(containerName); // Создание
контейнера типа HDIMAGE с именем containerName

AlgIdSpec params = new NameAlgIdSpec(container); // Параметры инициализации
генератора

kg.initialize(params);

KeyPair keyPair = kg.generateKeyPair();
```

В этом случае создается рабочий контейнер с именем containerName, для выработки ключа потребуется набрать достаточную энтропию в стандартном окне БиоДСЧ криптопровайдера КриптоПро CSP, в окне ввода и подтверждения пароля для контейнера указывается пин-код. Параметр container может содержать значение в формате FQCN, например, \\.\HDIMAGE\test.

Если происходит генерация ключей алгоритма обмена, соответствующих ГОСТ Р 34.10-2001, то требуется формировать параметры так:

```
AlgIdSpec params = new NameAlgIdSpec(AlgIdSpec.getDHDefault(), container); //
Параметры инициализации генератора GOST3410DH

kg.initialize(params);
```

3) для алгоритмов ГОСТ Р 34.10-2012 (256 бит) и ГОСТ Р 34.10-2012 (512 бит) на примере первого

```
KeyPairGenerator kg = KeyPairGenerator.getInstance(JCP.GOST_EL_2012_256_NAME,
JCSP.PROVIDER_NAME);

String container =
KeyStoreConfig.getHDImage().makeContainerName(containerName); // Создание
контейнера типа HDIMAGE с именем containerName

AlgIdSpec params = new NameAlgIdSpec(AlgIdSpec.OID_PARAMS_SIG_2012_512,
container); // Параметры инициализации генератора
```

```
kg.initialize(params);

KeyPair keyPair = kg.generateKeyPair();
```

В этом случае создается рабочий контейнер с именем containerName, для выработки ключа потребуется набрать достаточную энтропию в стандартном окне БиоДСЧ криптопровайдера КриптоПро CSP, в окне ввода и подтверждения пароля для контейнера указывается пин-код. Параметр container может содержать значение в формате FQCN, например, \\.\HDIMAGE\test.

Если происходит генерация ключей алгоритма ключевого обмена, соответствующих ГОСТ Р 34.10-2001, то возможно указать параметры так:

```
AlgIdSpec params = new NameAlgIdSpec(AlgIdSpec.OID_98, container); // Параметры инициализации генератора GOST3410DH_2012_256

kg.initialize(params);
```

Аналогичным образом задаются параметры для ГОСТ Р 34.10-2012 - идентификаторами наборов параметров AlgIdSpec.OID_PARAMS_EXC_2012_256 и AlgIdSpec.OID_PARAMS_EXC_2012_512.

После того, как генератор ключевой пары был создан, может также возникнуть необходимость установить набор параметров ключевой пары электронной подписи или ключевого обмена, отличный от параметров, установленных в контрольной панели JCSP. Операция изменения существующего набора параметров осуществляется при помощи метода *initialize()* класса KeyPairGenerator. Этому методу в качестве параметра передается объект `AlgIdInterface`, представляющий собой интерфейс набора устанавливаемых параметров (создание объектов такого типа описывается ниже). Тогда изменение набора параметров генератора ключевой пары производится следующим образом:

```
AlgIdInterface keyParams; // интерфейс набора параметров ключа

kg.initialize(keyParams); // установка параметров, определенных интерфейсом keyParams
```

Следует помнить о том, что изменение параметров генерации ключевой пары имеет смысл только до выполнения непосредственно генерации пары.

Стандартный интерфейс JCA допускает вызовы метода *initialize()* класса KeyPairGenerator и с другими параметрами (например, длина ключа), но при использовании криптопровайдера КриптоПро JCSP такие вызовы не имеют смысла, поскольку они не изменяют набор параметров, установленный ранее генератору.

Ключевые контейнеры поддерживают также флаг *dhAllowed*, означающий возможность производить на закрытом ключе согласование сессионных ключей шифрования. Этот бит автоматически устанавливается при генерации ключей ключевого обмена (DH) на всех алгоритмах, при генерации ключей подписи на алгоритме ГОСТ Р 34.10-2001, но не устанавливается по умолчанию для ключей подписи на алгоритме ГОСТ Р 34.10-2012. Для того, чтобы ключ подписи алгоритма ГОСТ Р 34.10-2012 был создан с установленным флагом *dhAllowed* и мог использоваться для ключевого обмена, нужно проинициализировать объект генератора следующим образом:

```
kg.initialize(new CrypdDhAllowedSpec());
```

2.2.Хэширование данных в соответствии с алгоритмом ГОСТ Р 34.11-94 и ГОСТ Р 34.11-2012

КриптопровайдерКриптоПро JCSP осуществляет хэширование данных в соответствии с алгоритмом ГОСТ Р 34.11-94 и ГОСТ Р 34.11-2012, через стандартный интерфейс JCA при помощи класса MessageDigest.

2.2.1.Создание объекта хэширования данных

Объект хэширования данных в соответствии с алгоритмами ГОСТ Р 34.11-94 и ГОСТ Р 34.11-2012 создается посредством вызова метода *getInstance()* класса MessageDigest. Этот метод является статическим и возвращает ссылку на объект класса MessageDigest, который обеспечивает выполнение требуемой операции.

Для создания объекта хэширования в соответствии с алгоритмом ГОСТ Р 34.11-94, ГОСТ Р 34.11-2012 (256 бит) и ГОСТ Р 34.11-2012 (512 бит) методу *getInstance()* необходимо в качестве параметра передать имя, идентифицирующее данный алгоритм ("GOST3411" или JCP.GOST_DIGEST_NAME, "GOST3411_2012_256" или JCP.GOST_DIGEST_2012_256_NAME, "GOST3411_2012_512" или JCP.GOST_DIGEST_2012_512_NAME соответственно). При таком вызове метода *getInstance()* совместно с определением требуемого алгоритма хэширования данных осуществляется также определение требуемого типа криптопровайдера (КриптоПро JCSP). Также стандартный интерфейс JCA позволяет в качестве параметра функции *getInstance()* класса MessageDigest вместе с именем алгоритма указывать имя криптопровайдера, используемого для

выполнения требуемой операции. Таким образом, создание генератора ключевой пары осуществляется одним из следующих способов (на примере ГОСТ Р 34.11-2012 (256 бит)):

```
MessageDigest digest = MessageDigest.getInstance("GOST3411", "JCSP");

MessageDigest digest = MessageDigest.getInstance(JCP.GOST_DIGEST_NAME,
JCSP.PROVIDER_NAME);

MessageDigest digest = MessageDigest.getInstance("GOST3411_2012_256", "JCSP");

MessageDigest digest = MessageDigest.getInstance(JCP.GOST_DIGEST_2012_256_NAME,
JCSP.PROVIDER_NAME);

MessageDigest digest = MessageDigest.getInstance("GOST3411_2012_512", "JCSP");

MessageDigest digest = MessageDigest.getInstance(JCP.GOST_DIGEST_2012_512_NAME,
JCSP.PROVIDER_NAME);
```

В случае использования алгоритма ГОСТ Р 34.11-94 будут использоваться параметры, установленные в контрольной панели параметрами (параметрами по умолчанию). Алгоритмы хэширования ГОСТ Р 34.11-2012 – не параметризованы. Стандартный интерфейс JCA класса MessageDigest не позволяет изменять параметры созданного объекта хэширования, но если существует такая необходимость, то при помощи дополнительных возможностей криптопровайдера КриптоПро JCSP можно установить требуемые параметры хэширования (отличные от параметров, установленных в контрольной панели).

2.2.2.Определение параметров хэширования данных

После того, как объект хэширования данных был создан, может возникнуть необходимость изменить параметры хэширования, установленные ранее в контрольной панели. Операция изменения существующего набора параметров не может быть осуществлена при помощи стандартного интерфейса JCA класса MessageDigest, поэтому для ее реализации следует привести созданный объект хэширования к типу `GostDigest` и уже для объекта этого класса воспользоваться методом *reset()*, передавая данному методу идентификатор устанавливаемых параметров (`OID`):

```
// ВНИМАНИЕ! для совместимости с другими продуктами КриптоПро

// допустимо использовать только параметры по умолчанию:

// "1.2.643.2.2.30.1"

OID digestOid = new OID("1.2.643.2.2.30.1");

/* преобразование к типу GostDigest */

GostDigest gostDigest = (GostDigest)digest;

/* установка требуемых параметров */

gostDigest.reset(digestOid);
```

Метод *reset()* (без параметров) стандартного интерфейса JCA класса MessageDigest изменяет установленные параметры хэширования на параметры по умолчанию.

Данная операция имеет смысл только до начала выполнения непосредственной операции создания хэша данных.

2.2.3.Копирование объекта хэширования данных

В некоторых случаях требуется создать копию уже существующего объекта хэширования данных, например, когда требуется осуществить хэширование как и части данных, так и всего исходного массива данных. В этом случае после того, как была обработана требуемая часть данных, необходимо сохранить (при помощи копирования) объект хэширования, и продолжить обработку оставшейся части (в результате чего будут обработаны все входные данные). Уже после выполняется подсчет значения хэша для обоих объектов (исходного - соответствующего всем данным и скопированного - соответствующего части данных).

Для этих целей используется метод *clone()* класса MessageDigest, который возвращает точную копию существующего объекта хэширования, включая внутреннее состояние. Этот метод может быть вызван на любом этапе выполнения операции хэширования после того, как объект хэширования был проинициализирован и до того, как операция хэширования была завершена.

2.2.4.Вычисление хэша данных в соответствии с алгоритмами ГОСТ Р 34.11-94 и ГОСТ Р 34.11-2012

После того, как объект хэширования был создан, вычисление хэша данных в соответствии с алгоритмами ГОСТ Р 34.11-94 и ГОСТ Р 34.11-2012 производится в два этапа: обработка данных и последующее завершение операции хэширования.

2.2.4.1.Обработка хэшируемых данных

Обработка хэшируемых данных может быть осуществлена двумя способами:

- при помощи метода *update()* класса MessageDigest;
- при помощи метода *read()* класса DigestInputStream.

Для обработки любым из этих способов хэшируемые данные должны быть представлены в виде байтового массива.

- **Использование метода *update()***.

Метод *update()* класса MessageDigest осуществляет обработку хэшируемых данных, представленных в виде байтового массива и подаваемых ему в качестве параметра. Существует 3 варианта обработки байтового массива данных при помощи этого метода:

1) Последовательная обработка каждого байта данных (при этом количество вызовов метода *update(byte b)* равно длине массива данных):

```
byte[] data;
for(int i = 0; i < data.length; i++)
    digest.update(data[i]);
```

2) Блочная обработка данных (данные обрабатываются блоками определенной длины):

```
byte[] data;
int BLOC_LEN = 1024;
// если длина исходных данных меньше длины блока
if(data.length/BLOC_LEN == 0)
    digest.update[data];
else {
    // цикл по блокам
    for (int i = 0; i < data.length/BLOC_LEN; i++) {
        byte[] bloc = new byte[BLOC_LEN];
        for(int j = 0; j < BLOC_LEN; j++) bloc[j] = data[j + i * BLOC_LEN];
        digest.update(bloc);
    }
    // обработка остатка
    byte[] endBloc = new byte[data.length % BLOC_LEN];
    for(int j = 0; j < data.length % BLOC_LEN; j++)
        bloc[j] = data[j + data.length - data.length % BLOC_LEN - 1];
    digest.update(bloc);
  }
```

3) Обработка данных целиком:

```
byte[] data;
digest.update(data);
```

Допускается комбинирование первого и второго варианта, обработка блоками различной длины, а также использование метода *update(byte[]data, int offset, int len)* - обработка массива данных со смещением. Но в любом случае следует помнить, что для корректного подсчета хэша на этапе завершения операции хэширования необходимо обработать все байты массива данных.

- **Использование метода *read()***.

Помимо использования метода *update()* класса MessageDigest обработка хэшируемых данных может быть осуществлена посредством метода *read()* класса DigestInputStream. Фактически, этот метод в зависимости от способа обработки данных (см. ниже) вызывает соответствующий вариант обработки при помощи метода *update()*. Для осуществления обработки данных из исходного байтового массива данных необходимо создать новый объект типа ByteArrayInputStream, а затем из него и созданного ранее объекта хэширования данных получить новый объект типа DigestInputStream:

```
byte[] data;

ByteArrayInputStream stream = new ByteArrayInputStream(data);

DigestInputStream digestStream = new DigestInputStream(stream, digest);
```

После того как объект типа DigestInputStream создан, обработка хэшируемых данных осуществляется при помощи метода *read()* класса DigestInputStream. При этом, как и в случае метода *update()*, существует 3 варианта использования метода *read()*:

1) Последовательная обработка каждого байта данных (при этом количество вызовов метода *read(byte b)* равно длине массива данных):

```
while (digestStream.available() != 0)
    digestStream.read();
```

2) Блочная обработка данных (данные обрабатываются блоками определенной длины, при этом считанные данные записываются в передаваемый функции *read()* массив):

```
int BLOC_LEN = 1024;
int DATA_LEN = digestStream.available();
// если длина исходных данных меньше длины блока
if(DATA_LEN/BLOC_LEN == 0) {
    byte[] data = new byte[DATA_LEN];
    digestStream.read(data, 0, DATA_LEN);
}
else {
    // цикл по блокам
    for (int i = 0; i < DATA_LEN/BLOC_LEN; i++) {
        byte[] bloc = new byte[BLOC_LEN];
        digestStream.read(bloc, 0, BLOC_LEN);
    }

    // обработка остатка
    byte[] endBloc = new byte[DATA_LEN % BLOC_LEN];
    digestStream.read(endBloc, 0, DATA_LEN % BLOC_LEN);
}
```

3) Обработка данных целиком (при этом считанные данные записываются в передаваемый функции *read()* массив):

```
byte[] data = new byte[digestStream.available()];
digestStream.read(data, 0, digestStream.available());
```

Допускается комбинирование первого и второго варианта, обработка блоками различной длины, а также использование метода *read(byte[]data, int offset, int len)* - запись считанных данных в массив со смещением. Но в любом случае следует помнить, что для корректного подсчета хэша на этапе завершения операции хэширования необходимо обработать все байты массива данных.

2.2.4.2.Завершение операции хэширования

После того, как все данные были обработаны, следует завершить операцию хэширования. Завершение осуществляется при помощи метода digest() класса MessageDigest. В результате выполнения этой функции подсчитывается значение хэша. Получить это значение можно двумя способами:

1. Вызовом метода без параметров - *digest()*.
   В этом случае метод возвращает байтовый массив, содержащий значение хэша;

2. Вызовом метода с параметрами - *digest(byte[] buf, int offset, int len)*.
   В этом случае метод записывает значение хэша в передаваемый ему массив со смещением.

2.3.Формирование электронной подписи в соответствии с алгоритмами ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012

Криптопровайдер КриптоПро JCSP осуществляет формирование электронной подписи данных, соответствующей алгоритмам ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012, через стандартный интерфейс JCA при помощи класса Signature. Формирование электронной подписи для любого другого алгоритма при помощи криптопровайдера КриптоПро JCSP запрещается.

В криптопровайдере КриптоПро JCSP, кроме хэша по алгоритмам ГОСТ Р 34.11-94 или ГОСТ Р 34.11-2012, можно подписывать непосредственно данные. Подпись данных осуществляет блок так называемых Raw-алгоритмов.

2.3.1.Создание объекта формирования электронной подписи

Объект формирования электронной подписи данных создается посредством вызова метода *getInstance()* класса Signature. Этот метод является статическим и возвращает ссылку на объект класса Signature, который обеспечивает выполнение требуемой операции.

Для создания объекта формирования электронной подписи в соответствии с алгоритмами ГОСТ Р 34.10-2001, ГОСТ Р 34.10-2012 (256 бит) и ГОСТ Р 34.10-2012 (512 бит) методу *getInstance()* необходимо в качестве параметра передать имя, идентифицирующее данный алгоритм:

- "GOST3411withGOST3410EL" или JCP.GOST_EL_SIGN_NAME

- "CryptoProSignature" или JCP.CRYPTOPRO_SIGN_NAME (для совместимости с КриптоПро CSP)
- "GOST3411_2012_256withGOST3410_2012_256" или JCP.GOST_SIGN_2012_256_NAME
- "GOST3411_2012_512withGOST3410_2012_512" или JCP.GOST_SIGN_2012_512_NAME
- "CryptoProSignature_2012_256" или JCP.CRYPTOPRO_SIGN_2012_256_NAME (для совместимости с КриптоПро CSP)
- "CryptoProSignature_2012_512" или JCP.CRYPTOPRO_SIGN_2012_512_NAME (для совместимости с КриптоПро CSP)

При таком вызове метода *getInstance()* совместно с определением требуемого алгоритма формирования электронной подписи осуществляется также определение требуемого типа криптопровайдера (КриптоПро JCSP). Также стандартный интерфейс JCA позволяет в качестве параметра функции *getInstance()* класса Signature вместе с именем алгоритма передавать имя криптопровайдера, используемого для выполнения требуемой операции.

Таким образом, создание объекта формирования электронной подписи осуществляется одним из следующих способов:

```
    Signature sig = Signature.getInstance("GOST3411withGOST3410EL");
    Signature sig = Signature.getInstance("GOST3411withGOST3410EL", "JCSP");
    Signature sig = Signature.getInstance(JCP.GOST_EL_SIGN_NAME, JCSP.PROVIDER_NAME);
    //для совместимости с КриптоПро CSP (подпись имеет обратный порядок байт)
    Signature sig = Signature.getInstance("CryptoProSignature");
    Signature sig = Signature.getInstance("CryptoProSignature", "JCSP");
    Signature sig = Signature.getInstance(JCP.CRYPTOPRO_SIGN_NAME, JCSP.PROVIDER_NAME);


    Signature sig = Signature.getInstance("GOST3411_2012_256withGOST3410_2012_256");
    Signature sig = Signature.getInstance("GOST3411_2012_256withGOST3410_2012_256",
"JCSP");
    Signature sig = Signature.getInstance(JCP.GOST_SIGN_2012_256_NAME,
JCSP.PROVIDER_NAME);
    //для совместимости с КриптоПро CSP (подпись имеет обратный порядок байт)
    Signature sig = Signature.getInstance("CryptoProSignature_2012_256");
    Signature sig = Signature.getInstance("CryptoProSignature_2012_256", "JCSP");
    Signature sig = Signature.getInstance(JCP.CRYPTOPRO_SIGN_2012_256_NAME,
JCSP.PROVIDER_NAME);


    Signature sig = Signature.getInstance("GOST3411_2012_512withGOST3410_2012_512");
    Signature sig = Signature.getInstance("GOST3411_2012_512withGOST3410_2012_512",
"JCSP");
    Signature sig = Signature.getInstance(JCP.GOST_SIGN_2012_512_NAME,
JCSP.PROVIDER_NAME);
    //для совместимости с КриптоПро CSP (подпись имеет обратный порядок байт)
    Signature sig = Signature.getInstance("CryptoProSignature_2012_512");
    Signature sig = Signature.getInstance("CryptoProSignature_2012_512", "JCSP");
    Signature sig = Signature.getInstance(JCP.CRYPTOPRO_SIGN_2012_512_NAME,
JCSP.PROVIDER_NAME);
```

Для создания объекта формирования электронной подписи в соответствии с алгоритмами ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012 без хэширования данных (Raw-алгоритмы), необходимо методу *getInstance()* передать имя, идентифицирующее данный алгоритм:

- "NONEwithGOST3410EL" или JCP.RAW_GOST_EL_SIGN_NAME

- "NONEwithCryptoProSignature" или JCP.RAW_CRYPTOPRO_SIGN_NAME (для совместимости с КриптоПро CSP)

или

- "NONEwithGOST3410_2012_256" или JCP.RAW_GOST_SIGN_2012_256_NAME
- "NONEwithCryptoProSignature_2012_256" или JCP.RAW_CRYPTOPRO_SIGN_2012_256_NAME (для совместимости с КриптоПро CSP)

или

- "NONEwithGOST3410_2012_512" или JCP.RAW_GOST_SIGN_2012_512_NAME
- "NONEwithCryptoProSignature_2012_512" или JCP.RAW_CRYPTOPRO_SIGN_2012_512_NAME (для совместимости с КриптоПро CSP)

В данном случае также можно определять и криптопровайдер:

```
Signature sig = Signature.getInstance("NONEwithGOST3410EL");
Signature sig = Signature.getInstance("NONEwithGOST3410EL", "JCSP");
Signature sig = Signature.getInstance(JCP.RAW_GOST_EL_SIGN_NAME, JCSP.PROVIDER_NAME);
//для совместимости с КриптоПро CSP (подпись имеет обратный порядок байт)
Signature sig = Signature.getInstance("NONEwithCryptoProSignature");
Signature sig = Signature.getInstance("NONEwithCryptoProSignature", "JCSP");
Signature sig = Signature.getInstance(JCP.RAW_CRYPTOPRO_SIGN_NAME,
JCSP.PROVIDER_NAME);
Signature sig = Signature.getInstance("NONEwithGOST3410_2012_256");
Signature sig = Signature.getInstance("NONEwithGOST3410_2012_256", "JCSP");
Signature sig = Signature.getInstance(JCP.RAW_GOST_SIGN_2012_256_NAME,
JCSP.PROVIDER_NAME);
//для совместимости с КриптоПро CSP (подпись имеет обратный порядок байт)
Signature sig = Signature.getInstance("NONEwithCryptoProSignature_2012_256");
Signature sig = Signature.getInstance("NONEwithCryptoProSignature_2012_256", "JCSP");
Signature sig = Signature.getInstance(JCP.RAW_CRYPTOPRO_SIGN_2012_256_NAME,
JCSP.PROVIDER_NAME);
Signature sig = Signature.getInstance("NONEwithGOST3410_2012_512");
Signature sig = Signature.getInstance("NONEwithGOST3410_2012_512", "JCSP");
Signature sig = Signature.getInstance(JCP.RAW_GOST_SIGN_2012_512_NAME,
JCSP.PROVIDER_NAME);
//для совместимости с КриптоПро CSP (подпись имеет обратный порядок байт)
Signature sig = Signature.getInstance("NONEwithCryptoProSignature_2012_512");
Signature sig = Signature.getInstance("NONEwithCryptoProSignature_2012_512", "JCSP");
Signature sig = Signature.getInstance(JCP.RAW_CRYPTOPRO_SIGN_2012_512_NAME,
JCSP.PROVIDER_NAME);
```

2.3.2.Инициализация объекта формирования электронной подписи

После того, как объект формирования электронной подписи был создан, необходимо определить набор параметров алгоритмов ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012, в соответствии с которыми будет осуществляться операция формирования электронной подписи. Определение параметров электронной подписи осуществляется во время инициализации операции создания подписи методом *initSign()* класса Signature. в соответствии с параметрами ключа электронной подписи, передаваемыми данному методу. Этот ключ не только определяет параметры формирования электронной подписи, но и используется в процессе ее формирования.

Необходимо помнить, что ключи электронной подписи, подаваемые на инициализацию объекта формирования электронной подписи, созданного описанным выше способом, должны соответствовать алгоритмам ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012. Способ генерации таких ключей описан выше. Создание подписи для любых других ключей запрещено.

Таким образом, инициализация операции формирования электронной подписи, во время которой происходит определение параметров подписи, осуществляется следующим образом:

```
PrivateKey privateKey; // обязательно ключ с алгоритмом "GOST3410EL" или "GOST3410_2012_256" или "GOST3410_2012_512"
// (соответствующий алгоритму ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012)
sig.initSign(privateKey);
```

2.3.3.Определение параметров формирования электронной подписи

После инициализации объекта формирования электронной подписи ключом подписи может возникнуть необходимость изменить параметры формирования электронной подписи (установить параметры, отличные от параметров ключа электронной подписи). Изменять разрешается только параметры хэширования, используемые в процессе формирования электронной подписи, причем изменение этих параметров допустимо только до начала операции формирования электронной подписи. Изменение параметров хэширования осуществляется при помощи метода *setParameter()* класса Signature. Этому методу в качестве параметра передается объект `ParamsInterface`, являющийся интерфейсом устанавливаемых параметров хэширования (создание объектов такого типа описывается ниже). Тогда изменение параметров хэширования для формирования электронной подписи осуществляется следующим образом:

```
ParamsInterface digestParams; // интерфейс параметров хэширования
sig.setParameter(digestParams); // установка параметров, определенных интерфейсом digestParams
```

Следует помнить, что использование данного метода имеет смысл только после того, как объект формирования подписи был проинициализирован. Если параметры хэширования были изменены при формировании подписи, то они должны быть соответствующим образом изменены в процессе проверки подписи. Также следует учесть, что для Raw-алгоритмов подобные установки не имеют смысла.

2.3.4.Формирование электронной подписи

После того, как объект подписи был создан и проинициализирован, операция формирования подписи производится в два этапа: обработка данных и последующее вычисление подписи, завершающее операцию формирования электронной подписи.

2.3.4.1.Обработка подписываемых данных

Обработка подписываемых данных осуществляется при помощи метода *update()* класса Signature. Этот метод осуществляет обработку подписываемых данных, представленных в виде байтового массива и подаваемых ему в качестве параметра. Существует 3 варианта обработки байтового массива данных при помощи этого метода:

1. Последовательная обработка каждого байта данных (при этом количество вызовов метода *update(byte b)* равно длине массива данных):

```
byte[] data;
for(int i = 0; i < data.length; i++)
sig.update(data[i]);
```

2. Блочная обработка данных (данные обрабатываются блоками определенной длины):

```
byte[] data;
int BLOC_LEN = 1024;

// если длина исходных данных меньше длины блока
if(data.length/BLOC_LEN == 0)
    sig.update[data];
else {
    // цикл по блокам
    for (int i = 0; i < data.length/BLOC_LEN; i++) {
        byte[] bloc = new byte[BLOC_LEN];
        for(int j = 0; j < BLOC_LEN; j++) bloc[j] = data[j + i * BLOC_LEN];
        sig.update(bloc);
```

```
        }

        // обработка остатка
        byte[] endBloc = new byte[data.length % BLOC_LEN];
        for(int j = 0; j < data.length % BLOC_LEN; j++)
            bloc[j] = data[j + data.length - data.length % BLOC_LEN - 1];
        sig.update(bloc);
    }
```

3. Обработка данных целиком:

```
byte[] data;
sig.update(data);
```

Допускается комбинирование первого и второго варианта, обработка блоками различной длины, а также использование метода *update(byte[]data, int offset, int len)* - обработка массива данных со смещением. Но в любом случае следует помнить, что для корректного вычисления подписи на этапе завершения операции создания подписи необходимо обработать все байты массива данных.

В случае, если вычисляется подпись по Raw-алгоритму, общее количество переданных байтов должно быть равно длине значения используемой хэш-функции. Будут ли они переданы за один вызов, или несколькими вызовами, значения не имеет.

2.3.4.2.Вычисление значения электронной подписи

После того, как все данные были обработаны, следует завершить операцию формирования электронной подписи. Завершение осуществляется при помощи метода *sign()* класса Signature. В результате выполнения этой функции вычисляется значение подписи. Получить это значение можно двумя способами:

1. Вызов метода без параметров - *sign()*. В этом случае метод возвращает байтовый массив, содержащий значение подписи;

2. Вызов метода с параметрами - *sign(byte[] buf, int offset, int len)*. В этом случае метод записывает значение подписи в передаваемый ему массив со смещением.

Методы для Raw-алгоритмов подписывают данные, переданные методами *update*, интерпретируя их как значение хэша. Алгоритмы, использующие ГОСТ Р 34.11-94 или ГОСТ Р 34.11-2012, вычисляют хэш переданных данных, а затем его подписывают.

2.4.Проверка электронной подписи в соответствии с алгоритмом ГОСТ Р 34.10-2012

Криптопровайдер КриптоПро JCSP осуществляет проверку электронной подписи данных, соответствующую алгоритмам ГОСТ Р 34.10-2001 и ГОСТ Р Р 34.10-2012, через стандартный интерфейс JCA при помощи класса Signature.

2.4.1.Создание объекта проверки электронной подписи

Объект проверки электронной подписи данных создается посредством вызова метода *getInstance()* класса Signature. Этот метод является статическим и возвращает ссылку на класс Signature, который обеспечивает выполнение требуемой операции.

Для создания объекта проверки электронной подписи в соответствии с алгоритмами ГОСТ Р 34.10-2001 (а точнее, алгоритм подписи ГОСТ Р 34.10-2001 с алгоритмом хэширования ГОСТ Р 34.11-94), или ГОСТ Р 34.10-2012 (256) (а точнее, алгоритм подписи ГОСТ Р 34.10-2012 (256) с алгоритмом хэширования ГОСТ Р 34.11-2012 (256)) , или ГОСТ Р 34.10-2012 (512) (а точнее, алгоритм подписи ГОСТ Р 34.10-2012 (512) с алгоритмом хэширования ГОСТ Р 34.11-2012 (512)) методу *getInstance()* необходимо в качестве параметра передать имя, идентифицирующее требуемый алгоритм проверки электронной подписи (см. создание объекта формирования электронной подписи; для алгоритма ГОСТ Р 34.10-2001 - "GOST3411withGOST3410EL" или JCP.GOST_EL_SIGN_NAME, для алгоритма ГОСТ Р 34.10-2012 (256) - "GOST3411withGOST3410_2012_256" или JCP.GOST_SIGN_2012_256_NAME, для алгоритма ГОСТ Р 34.10-2012 (512) - "GOST3411withGOST3410_2012_512" или JCP.GOST_SIGN_2012_512_NAME).

Для создания объекта проверки электронной подписи в соответствии с алгоритмами ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012 без подсчета значения хэш-функции, методу *getInstance()* необходимо в качестве параметра передать имя, идентифицирующее требуемый алгоритм проверки электронной подписи (см. создание объекта формирования электронной подписи; для алгоритма ГОСТ Р 34.10-2001 - "NONEwithGOST3410EL" или JCP.RAW_GOST_EL_SIGN_NAME, для алгоритма ГОСТ Р 34.10-2012 (256) - "NONEwithGOST3410_2012_256" или

JCP.RAW_GOST_SIGN_2012_256_NAME, для алгоритма ГОСТ Р 34.10-2012 (512) - "NONEwithGOST3410_2012_512" или JCP.RAW_GOST_SIGN_2012_512_NAME).

2.4.2.Инициализация объекта проверки электронной подписи

После того, как объект проверки электронной подписи был создан, необходимо определить набор параметров заданного при создании объекта алгоритма (ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012), в соответствии с которыми будет осуществляться операция проверки электронной подписи. Определение параметров электронной подписи осуществляется во время инициализации операции проверки подписи методом *initVerify()* класса Signature. в соответствии с параметрами ключа проверки электронной подписи, передаваемыми данному методу. Этот ключ не только определяет параметры проверки электронной подписи, но и используется в самом процессе проверки.

Необходимо помнить, что ключи проверки электронной подписи, подаваемые на инициализацию объекта проверки электронной подписи, созданного описанным выше способом должны соответствовать алгоритму этого объекта (соответственно, алгоритмам ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012). Способ генерации ключей для алгоритмов ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012 описан выше. Помимо этого для корректной проверки электронной подписи требуется, чтобы ключ проверки электронной подписи соответствовал ключу электронной подписи, на котором осуществлялось формирование подписи.

Таким образом, инициализация операции проверки электронной подписи, во время которой происходит определение параметров электронной подписи, осуществляется следующим образом:

```
PublicKey publicKey;  // алгоритм ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012
sig.initVerify(publicKey);
```

2.4.3.Определение параметров проверки электронной подписи

После инициализации объекта проверки электронной подписи ключом проверки электронной подписи может возникнуть необходимость изменить параметры проверки электронной подписи (установить параметры, отличные от параметров ключа проверки электронной подписи). Такая необходимость может возникнуть в случае, когда параметры электронной подписи были некоторым образом изменены при ее формировании. Тогда для корректной проверки этой электронной подписи требуется аналогичным образом изменить параметры объекта проверки подписи (установить те же самые параметры). Изменение параметров проверки электронной подписи осуществляется аналогично изменению параметров формирования электронной подписи.

2.4.4.Проверка электронной подписи

После того, как объект проверки подписи был создан и проинициализирован, операция проверки подписи производится в два этапа: обработка данных и последующая проверка подписи, завершающая текущую операцию.

2.4.4.1.Обработка подписанных данных

Обработка подписанных данных осуществляется при помощи метода *update()* класса Signature и полностью аналогична обработке данных при создании подписи.

2.4.4.2.Проверка электронной подписи

После того, как все данные были обработаны, следует завершить операцию проверки электронной подписи. Завершение осуществляется при помощи метода *verify()* класса Signature. Этой функции передается проверяемое значение подписи и в результате ее работы возвращается логическое значение: **true** - подпись верна, **false** - подпись не верна. Значение проверяемой подписи можно передать двумя способами:

- в качестве байтового массива - *verify(byte[] signature)*;
- в качестве байтового массива со смещением - *verify(byte[] signature, int offset, int length)*;

Проверка электронной подписи в сертификате ключа проверки электронной подписи осуществляется так же, как при проверке цепочки сертификатов. При реализации проверки электронной подписи документа с использованием сертификата ключа проверки подписи, исключается возможность проверки электронной подписи электронного документа без проверки электронной подписи в сертификате ключа проверки подписи или без наличия положительного результата проверки электронной подписи в сертификате ключа проверки электронной подписи.

2.5.Создание ключей парной связи с помощью алгоритма ключевого обмена

Криптопровайдер КриптоПро JCSP осуществляет работу с ключами парной связи в соответствии с алгоритмом ключевого обмена (см. RFC 4357 и методические рекомендации технического комитета по стандартизации 26 «Криптографическая защита информации» «ИСПОЛЬЗОВАНИЕ КРИПТОГРАФИЧЕСКИХ АЛГОРИТМОВ, СОПУТСТВУЮЩИХ ПРИМЕНЕНИЮ СТАНДАРТОВ ГОСТ Р 34.10-2012 И ГОСТ Р 34.11-2012»), через стандартный интерфейс JCE при помощи класса KeyAgreement.

2.5.1.Создание объекта генерации ключей парной связи

Объект генерации ключей парной связи (далее *генератор*) создается посредством вызова метода *getInstance()* класса KeyAgreement. Этот метод является статическим и возвращает ссылку на класс KeyAgreement, который обеспечивает выполнение требуемой операции.

Для создания генератора ключей парной связи в соответствии с алгоритмом ключевого обмена методу *getInstance()* необходимо в качестве параметра передать имя, идентифицирующее данный алгоритм ("GOST3410DHEL" или JCP.GOST_EL_DH_NAME и "GOST3410DH_2012_256" или JCP.GOST_DH_2012_256_NAME или "GOST3410DH_2012_512" или JCP.GOST_DH_2012_512_NAME). При таком вызове метода *getInstance()* совместно с определением требуемого алгоритма генерации ключей согласования осуществляется также определение требуемого типа криптопровайдера (КриптоПро JCSP). Также стандартный интерфейс JCE позволяет в качестве параметра функции *getInstance()* класса KeyAgreement вместе с именем алгоритма передавать имя криптопровайдера, используемого для выполнения требуемой операции.

Таким образом, создание генератора ключей обмена в соответствии с алгоритмом ключевого обмена осуществляется одним из следующих способов:

```
KeyAgreement ka = KeyAgreement.getInstance("GOST3410DHEL");

KeyAgreement ka = KeyAgreement.getInstance("GOST3410DHEL", "JCSP");

KeyAgreement ka = KeyAgreement.getInstance(JCP.GOST_EL_DH_NAME,
JCSP.PROVIDER_NAME);
```

Для ключей на алгоритме ГОСТ Р 34.10-2012 (256):

```
KeyAgreement ka = KeyAgreement.getInstance("GOST3410DH_2012_256");

KeyAgreement ka = KeyAgreement.getInstance("GOST3410DH_2012_256", "JCSP");

KeyAgreement ka = KeyAgreement.getInstance(JCP.GOST_DH_2012_256_NAME,
JCSP.PROVIDER_NAME);
```

Для ключей на алгоритме ГОСТ Р 34.10-2012 (512):

```
KeyAgreement ka = KeyAgreement.getInstance("GOST3410DH_2012_512");

KeyAgreement ka = KeyAgreement.getInstance("GOST3410DH_2012_512", "JCSP");

KeyAgreement ka = KeyAgreement.getInstance(JCSP.GOST_DH_2012_512_NAME,
CryptoProvider.PROVIDER_NAME);
```

Созданный таким образом генератор осуществляет выработку ключей согласования из ключей обмена, соответствующих алгоритму ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012 для использования в алгоритме ключевого обмена. Ключи ключевого обмена, соответствующие алгоритму ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012, могут быть как прочитаны из контейнера, так и получены при помощи генератора ключевой пары обмена (см. выше).

2.5.2.Инициализация генератора ключей парной связи

После того, как генератор ключей парной связи был создан, необходимо проинициализировать его закрытым ключом обмена, в соответствии с которым будет осуществляться выработка ключей согласования, а также параметрами, на основе которых эта выработка будет производиться. Такая инициализация осуществляется при помощи метода *init(Key key, AlgorithmParameterSpec params)* класса KeyAgreement. Этому методу в качестве параметра key передается закрытый ключ обмена, соответствующий алгоритму ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012. В качестве параметров params, участвующих в выработке ключей согласования, передается объект класса IvParameterSpec, представляющий собой байтовый вектор для выработки ключей согласования (UKM).

Инициализация генератора ключей согласования осуществляется следующим образом:

```
PrivateKey privateKey;  // обязательно закрытый ключ обмена, соответствующий
                        // алгоритму ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012
```

```
IvParameterSpec spec;   // стартовый вектор
ka.init(key, spec);
```

2.5.3.Выполнение фазы согласования ключей

После того, как генератор ключей парной связи был создан и проинициализирован закрытым ключом и байтовым вектором, требуется проинициализировать его открытым ключом, в соответствии с которым будет сформирован ключ парной связи (фаза согласования ключей). Выполнение фазы осуществляется при помощи метода *doPhase(Key key, boolean lastPhase)* класса KeyAgreement. Этому методу в качестве параметра передается открытый ключ, соответствующий алгоритму ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012. Следует помнить, что параметры хэширования и электронной подписи этого ключа должны совпадать с соответствующими параметрами поданного на инициализацию закрытого ключа. Второй параметр метода *doPhase()* игнорируется. Считается, что эта величина всегда равна `true`.

```
PublicKey publicKey;    // обязательно открытый ключ обмена, соответствующий
                        // закрытому ключу privateKey
ka.doPhase(publicKey, true);
```

Стандартный интерфейс JCE допускает возвращение данным методом объекта класса Key. В криптопровайдере КриптоПро JCSP в качестве такого объекта возвращается переданный методу *doPhase()* открытый ключ. Для получения самого ключа согласования необходимо сделать еще один шаг.

2.5.4.Генерация ключа парной связи

После того, как определены закрытый и открытый ключи, на основе которых осуществляется выработка ключа парной связи, а также параметры (байтовый вектор), участвующие в этой выработке, необходимо выработать собственно ключ парной связи. Генерация такого ключа осуществляется при помощи вызова метода `generateSecret(String algorithm)` класса KeyAgreement. В качестве параметра этот метода получает строковое представление алгоритма ключа парной связи ("GOST28147" или JCSP.GOST_CIPHER_NAME). Возвращает этот метод объект класса SecretKey, представляющий собой ключ парной связи. Этот ключ является ключом шифрования алгоритма ГОСТ 28147-89 с параметрами шифрования, соответствующими параметрам открытого ключа.

Таким образом, генерация ключа согласования осуществляется:

```
SecretKey agreeKey = ka.generateSecret("GOST28147");
SecretKey agreeKey = ka.generateSecret(JCSP.GOST_CIPHER_NAME);
```

Для выработки 512-битного секретного ключа для использования в функциях HMAC, основанных на использовании хэш-функций ГОСТ Р 34.11-2012 (256 бит) и ГОСТ Р 34.11-2012 (512 бит) следует передать методу строку "SYMMETRIC512" (JCSP.SYMMETRIC_512).

```
SecretKey agreeKey = ka.generateSecret("SYMMETRIC512");
SecretKey agreeKey = ka.generateSecret(JCSP.SYMMETRIC_512);
```

2.6.Работа с ключевыми носителями

Криптопровайдер КриптоПро JCSP осуществляет хранение ключей электронной подписи и ключевого обмена и соответствующих им сертификатов на ключевых носителях, которые поддерживаются криптопровайдером КриптоПро CSP, фактически выполняющим криптографические операции. Доступ к ключевым хранилищам осуществляется через стандартный интерфейс JCA при помощи класса KeyStore. Следует заметить, что использование интерфейса этого класса является общим как для работы с ключевыми носителями, так и для работы с хранилищем сертификатов. Однако, существуют и некоторые особенности, описанные ниже.

Поскольку генерация ключевых пар электронной подписи и ключевого обмена разрешена только для алгоритмов ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012, то и запись ключей электронной подписи и ключевого обмена на ключевые носители разрешена только для ключей этого алгоритма. Чтение ключей электронной подписи и ключевого обмена с носителей допустимо для ключей, соответствующих алгоритмам ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012. Запись и чтение сертификатов открытых ключей допустимы для обоих алгоритмов и удовлетворяет следующим правилам:

- для каждого ключа, хранящегося на ключевом носителе, допустимо хранение одного соответствующего ключу сертификата на этом носителе;

- если на ключевом носителе уже имеется сертификат, соответствующий ключу, то при записи нового сертификата существующий сертификат уничтожается;

- если записываемый сертификат не соответствует ни одному из ключей, хранящихся на носителе, то сертификат записывается в хранилище доверенных сертификатов.

2.6.1.Запись ключей электронной подписи и ключевого обмена с алгоритмами ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012 на ключевые носители

После того, как ключ электронной подписи или ключевого обмена, соответствующий алгоритму ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012 (256), был создан, криптопровайдер КриптоПро JCSP позволяет записать его на один из доступных ключевых носителей. Также как и генерация, сохранение на ключевые носители разрешается только для ключей электронной подписи, соответствующих алгоритму ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012.

Осуществление операций с ключевыми носителями (в том числе и запись ключа электронной подписи или ключевого обмена) производится через стандартный интерфейс хранилища ключей JCA (интерфейс класса KeyStore) посредством выполнения следующих действий:

2.6.1.1.Определение типа используемого ключевого носителя

Определение типа используемого ключевого носителя осуществляется посредством вызова метода *getInstance()* класса KeyStore. Этот метод является статическим и возвращает ссылку на класс KeyStore, который обеспечивает выполнение требуемой операции.

Для определения конкретного типа ключевого носителя методу *getInstance()* необходимо в качестве параметра передать имя, идентифицирующее необходимый тип. В криптопровайдере КриптоПро JCSP реализовано несколько типов носителей:

- имя "HDIMAGE" определяет жесткий диск;

- имя "REGISTRY" определяет реестр (в случае ОС Windows);

- другие типы носителей.

При таком вызове метода *getInstance()* совместно с определением требуемого типа ключевого носителя осуществляется также определение требуемого типа криптопровайдера (КриптоПро JCSP). Также стандартный интерфейс JCA позволяет в качестве параметра функции *getInstance()* класса KeyStore вместе с типом носителя указывать имя криптопровайдера, используемого для выполнения требуемой операции. Таким образом, определение типа используемого ключевого носителя осуществляется одним из следующих способов:

Таким образом, определение типа используемого ключевого носителя осуществляется одним из следующих способов:

```
KeyStore ks = KeyStore.getInstance("HDIMAGE", "JCSP");
KeyStore ks = KeyStore.getInstance("REGISTRY", "JCSP");
```

Определение типа используемого ключевого носителя представляет собой инициализацию стандартного ключевого хранилища JCA, поэтому операции записи на ключевой носитель или чтения с него следует осуществлять в соответствии с интерфейсом JCA, а именно, требуется предварительная загрузка содержимого носителя и последующее после выполнения операции сохранение содержимого.

2.6.1.2.Загрузка содержимого ключевого носителя

Согласно интерфейсу стандартного ключевого хранилища JCA перед началом выполнения каких либо операций требуется загрузка всего содержимого хранилища, следовательно, перед выполнением операций с ключевым носителем следует загрузить его содержимое. Загрузка содержимого стандартного хранилища JCA осуществляется посредством вызова метода *load()* класса KeyStore. Согласно интерфейсу JCA функции *load()* следует передавать два параметра: поток, из которого осуществляется чтение содержимого ключевого хранилища, и пароль на хранилище.

Поскольку работа с ключевыми носителями (чтение/запись ключей электронной подписи и ключевого обмена и соответствующих им сертификатов) и с хранилищем сертификатов (чтение/запись доверенных сертификатов) в криптопровайдере КриптоПро JCSP реализована согласно общему интерфейсу JCA класса KeyStore, то в некоторых случаях вызов функции *load()*

осуществляет как загрузку содержимого ключевого носителя, так и содержимого хранилища сертификатов, проинициализированного именем данного носителя. Ввиду этого возникают особенности использования параметров функции *load()*:

- Первый параметр (входной поток из которого осуществляется загрузка содержимого хранилища) используется криптопровайдером КриптоПро JCSP только в случае работы с хранилищем сертификатов. Поэтому, при осуществлении операции записи на ключевой носитель или чтения с него в качестве данного параметра, можно указывать `null`. Если проинициализированное именем носителя стандартное хранилище KeyStore используется как для работы с данным ключевым носителем, так и для работы с хранилищем сертификатов, то в качестве данного параметра следует указывать поток содержимого хранилища сертификатов. Тогда загруженное содержимое потока будет использоваться только при работе с хранилищем сертификатов, а при работе с носителями будет игнорироваться. Если проинициализированного именем данного носителя хранилища сертификатов на момент вызова функции *load()* не существует, то в качестве этого параметра следует указывать `null`.

- Второй параметр является паролем на хранилище сертификатов, которое было проинициализировано именем данного носителя. При операциях с ключевыми носителями этот параметр фактически не используется, но ввиду общего интерфейса для носителей и хранилища сертификатов, при любых операциях с ключевым носителем пароль следует указывать. Если на момент вызова метода *load()* не существует хранилища сертификатов, проинициализированного именем данного носителя, то в качестве этого параметра следует указывать `null`.

Таким образом, перед началом выполнения операции с ключевым носителем следует выполнить загрузку содержимого этого носителя (и, если это требуется, загрузку проинициализированного именем носителя хранилища сертификатов) следующим образом:

```
ks.load(null, null);    // не существует хранилища сертификатов,
                        // проинициализированного именем данного
                        // ключевого носителя
char[] passwd;
ks.load(null, passwd);  // хранилище сертификатов существует,
                        // на него установлен пароль passwd,
                        // но последующие операции будут производиться
                        // только с носителем
InputStream stream;
ks.load(stream, passwd);// хранилище сертификатов существует,
                        // на него установлен пароль passwd,
                        // последующие операции будут производиться
                        // как с носителем, так и с хранилищем
                        // сертификатов. Содержимое хранилища
                        // записано в stream.
```

2.6.1.3.Запись ключа на носитель

После того, как содержимое носителя (и, если это требуется, содержимое проинициализированного именем носителя хранилища сертификатов) было загружено, осуществляется собственно запись ключа на носитель. Данная операция реализуется при помощи вызова метода *setKeyEntry()* или метода *setEntry()* класса KeyStore:

```
String alias;       // идентификатор (уникальное имя) ключа и
                    // соответствующего ему сертификата
```

```
                        // открытого ключа
    PrivateKey key;     // ключ электронной подписи или ключевого обмена
                        // от пароля на хранилище, используемого при загрузке)
    Certificate[] chain;// цепочка сертификатов, начиная с корневого и
                        // и заканчивая сертификатом открытого ключа,
                        // соответствующего открытому ключу


    ks.setKeyEntry(alias, key, null, chain);
```

либо

```
    KeyStore.ProtectionParameter params = new KeyStore.PasswordProtection(password);
    KeyStore.Entry entry = new JCPPrivateKeyEntry(key, chain);
    ks.setEntry(alias, entry, params);
```

В функцию setKeyEntry вместо пароля передается null, т.к. далее ввод пин-кода потребуется произвести в окне криптопровайдера КриптоПро CSP; т.е. указание пароля непосредственно при вызове *setKeyEntry* излишне. Использование функции setEntry позволяет, наоборот, избежать ввода пин-кода в окне криптопровайдера КриптоПро CSP, т.е. передать его программно.

При вызове setKeyEntry после генерации пары на неинициализированном рабочим контейнером генераторе произойдет создание рабочего контейнера с именем alias и типом keyStoreType, копирование в него закрытого ключа и сертификата из временного контейнера и удаление последнего. При создании контейнера появится стандартное окно КриптоПро CSP для ввода и подтверждения пароля на контейнер. При вызове данной функции после генерации пары на инициализированном генераторе, если alias соответствует containerName в генераторе, произойдет только установка сертификата, так как контейнер уже создан; если alias не соответствует containerName в генераторе, то произойдет копирование ключа и установка сертификата в новый рабочий контейнер alias с соответствующим вводом пин-кода и его подтверждением для данного контейнера.

В качестве параметра chain выступает цепочка сертификатов, состоящая либо из единственного сертификата, открытый ключ которого соответствует записываемому открытому ключу, либо из собственно цепочки сертификатов, в которой сертификат соответствующий ключу находится в нулевом элементе массива. Сертификатом, соответствующим ключу, может быть как самоподписанный сертификат, так и сертификат, заверенный доверенным центром сертификации. Если при этом передаваемый сертификат не соответствует открытому ключу key, то метод *setKeyEntry()* вызовет исключение.

В обоих сценариях после генерации ключевой пары можно использовать функцию setEntry вместо setKeyEntry, что означает:

- создание рабочего контейнера с именем alias и типом контейнера, копирование в него закрытого ключа и сертификата из временного контейнера и удаление последнего. При создании контейнера будет использован пароль из параметра ProtectionParameter функции setEntry;

- если alias соответствует containerName, то произойдет смена текущего пароля на тот, что указан в ProtectionParameter, и установка сертификата; если alias не соответствует containerName, то произойдет создание нового контейнера alias с паролем из ProtectionParameter и копирование в него ключа и сертификатов.

2.6.2.Чтение ключей электронной подписи и ключевого обмена с алгоритмами ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012 с ключевых носителей

Чтение ключей электронной подписи и ключевого обмена в криптопровайдере КриптоПро JCSP производится через стандартный интерфейс хранилища ключей JCA (через интерфейс класса KeyStore) посредством выполнения следующих действий:

- определение типа используемого ключевого носителя;

- загрузка содержимого ключевого носителя;
- чтение ключа с носителя;
- сохранение содержимого ключевого носителя.

После того, как содержимое носителя (и, если это требуется, содержимое проинициализированного именем носителя хранилища сертификатов) было загружено, осуществляется собственно чтение ключа с носителя. Данная операция реализуется при помощи вызова метода *getKey()* класса KeyStore, возвращающего требуемый ключ электронной подписи или ключевого обмена, следующим образом:

- с помощью метода getKey класса KeyStore

```
String alias;   // Идентификатор (уникальное имя) получаемого ключа
PrivateKey key = (PrivateKey)ks.getKey(alias, null);
```

- с помощью метода getEntry класса KeyStore

```
String alias;   // Идентификатор (уникальное имя) получаемого ключа
KeyStore.ProtectionParameter params = new KeyStore.PasswordProtection(password);
JCPPrivateKeyEntry entry = ks.getEntry(alias, params);
```

В метод getKey вместо пароля передается null, так как ввод пин-кода потребуется только при непосредственной работе с ключом; осуществляться ввод пароля будет в стандартном окне криптопровайдера КриптоПро CSP. В метод getEntry пароль передается в виде параметра ProtectionParameter, при этом вводить пин-код (если он был правильным) в окне CSP в дальнейшем не потребуется. Это означает, что операции вида getKey/setKeyEntry используют ввод пин-кода в окне CSP, а функции getEntry/setEntry - нет.

Если использовать функцию getEntry и передать в ProtectionParameter неправильный пароль, то будет предложено ввести пароль заново в окне CSP (оставшиеся 2 попытки).

Если необходимо сразу получать уведомление о неправильном пароле и прекращать работу (по инициированному исключению), избежав повторного ввода пароля в окне CSP, можно воспользоваться классом *JCPProtectionParameter (пакет ru.CryptoPro.JCP.params)*. Он расширяет класс *KeyStore.PasswordProtection* и содержит дополнительный параметр, обозначающий режим работы, эквивалентный CRYPT_SILENT (0x00000040). Данный параметр может быть задан в конструкторе класса:

```
ProtectionParameter pp = new JCPProtectionParameter(password, true);    // Задание CRYPT_SILENT
JCPPrivateKeyEntry entry = ks.getEntry(alias, pp);   // Открытие контейнера в режиме CRYPT_SILENT
```

Таким образом, в случае передачи неправильного пароля будет сгенерировано исключение, и попытки повторного ввода пароля будут исключены.

2.6.3.Запись сертификата открытого ключа на ключевой носитель в соответствии с хранящимся на нем ключом

Криптопровайдер КриптоПро JCSP позволяет осуществлять запись на ключевые носители сертификатов открытых ключей, соответствующих хранящимся на носителе ключам электронной подписи или ключевого обмена. Таким образом, операция записи сертификатов открытых ключей допустима для ключей, соответствующих алгоритмам ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012, и приводит к следующим результатам:

- добавление сертификата на ключевой носитель в соответствии с хранящимся на носителе ключом электронной подписи или ключевого обмена, если ранее на носителе не было такого сертификата;
- перезапись существующего на носителе сертификата открытого ключа, соответствующего ключу электронной подписи или ключевого обмена, новым сертификатом.

При этом осуществляется проверка соответствия открытого ключа, указанного в записываемом сертификате, с открытом ключом ключевой пары электронной подписи или ключевого обмена.

Операция записи сертификата открытого ключа производится через стандартный интерфейс хранилища ключей JCA (через интерфейс класса KeyStore) посредством выполнения следующих действий:

- определение типа используемого ключевого носителя;
- загрузка содержимого ключевого носителя;
- запись сертификата открытого ключа на носитель;
- сохранение содержимого ключевого носителя.

После того, как содержимое носителя (и, если это требуется, содержимое проинициализированного именем носителя хранилища сертификатов) было загружено, осуществляется собственно запись сертификата открытого ключа на носитель. Данная операция реализуется при помощи вызова метода *setCertificateEntry()* класса KeyStore следующим образом:

```
String alias;        // идентификатор (уникальное имя) ключа,
                         // которому соответствует ключ проверки электронной подписи сертификата
Certificate cert;    // записываемый сертификат
ks.setCertificateEntry(alias, cert);
```

Следует отметить некоторые особенности вызова функции *setCertificateEntry()*:

- параметр `alias` является уникальным именем ключа, которому соответствует ключ проверки электронной подписи записываемого сертификата. Если ключу с заданным именем `alias` уже соответствует некоторый сертификат на носителе, то этот сертификат будет перезаписан новым. В противном случае передаваемый сертификат будет просто добавлен на носитель. При этом после записи сертификату будет присвоено имя соответствующего ему ключа - передаваемый методу *setCertificateEntry()* в качестве параметра `alias`. Если же на носителе не существует ключа с заданным `alias`, то передаваемый сертификат будет добавлен в хранилище доверенных сертификатов;
- параметр `cert` представляет собой записываемый на носитель сертификат. Ключ проверки электронной подписи этого сертификата должен соответствовать ключу с именем `alias`, если он существует, в противном случае метод *setCertificateEntry()* сгенерирует исключение.

2.6.4.Чтение сертификата открытого ключа с ключевого носителя

Криптопровайдер КриптоПро JCSP позволяет осуществлять чтение с ключевых носителей сертификатов открытых ключей, соответствующих алгоритму ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012 через стандартный интерфейс хранилища ключей электронной подписи и ключевого обмена JCA (интерфейс класса KeyStore) посредством выполнения следующих действий:

- определение типа используемого ключевого носителя;
- загрузка содержимого ключевого носителя;
- чтение сертификата открытого ключа с носителя;
- сохранение содержимого ключевого носителя.

После того, как содержимое носителя (и, если это требуется, содержимое проинициализированного именем носителя хранилища сертификатов) было загружено, осуществляется собственно чтение сертификата открытого ключа с носителя. Данная операция реализуется при помощи вызова метода *getCertificate()* класса KeyStore, возвращающего запрашиваемый сертификат, следующим образом:

```
String alias;    // идентификатор (уникальное имя) сертификата,
                 // установленный при записи сертификата на носитель
Certificate cert = ks.getCertificate(alias);
```

Следует отметить некоторые особенности вызова функции *getCertificate()* с передаваемым параметром `alias`, являющимся уникальным именем запрашиваемого сертификата:

- если на носителе существует сертификат с заданным именем `alias`, то метод *getCertificate()* вернет сертификат с носителя;

- если на носителе не существует сертификата с именем `alias`, но в хранилище сертификатов есть сертификат с таким именем, то метод *getCertificate()* вернет сертификат из хранилища сертификатов;

- если сертификата с заданным именем `alias` не существует ни на носителе, ни в хранилище сертификатов, то метод *getCertificate()* вернет `null`.

2.6.5.Удаление секретного ключа с ключевого носителя

Удаление секретного ключа с ключевого носителя осуществляется вызовом функции *deleteEntry* с передаваемым параметром `alias`, являющимся уникальным именем ключа. Для носителей требующих пароля для удаления контейнера (например, смарт-карт и токенов), будет выведено окно криптопровайдера КриптоПро CSP с предложением ввести пароль.

2.7.Работа с хранилищем доверенных сертификатов

Криптопровайдер КриптоПро JCSP поддерживает работу с хранилищами доверенных сертификатов только для операционных систем семейства Google Android. Доступ к хранилищам доверенных сертификатов для данных операционных систем осуществляется через стандартный интерфейс JCA (класс KeyStore). Следует заметить, что использование интерфейса этого класса является общим как для работы хранилищем сертификатов, так и для работы с ключевыми носителями. Особенности работы с хранилищем сертификатов описаны ниже.

2.7.1.Запись сертификатов в хранилище доверенных сертификатов

Криптопровайдер КриптоПро JCSP позволяет осуществлять запись доверенных сертификатов в определяемое пользователем хранилище доверенных сертификатов, соответствующее стандартному интерфейсу хранилища JCA (класс KeyStore). Для этого необходимо выполнить последовательность действий, аналогичную последовательности при работе с ключевыми носителями:

2.7.1.1.Инициализация хранилища доверенных сертификатов

Аналогично определению типа используемого ключевого носителя.

Для удобства пользователя был также создан тип хранилища "CertStore". В хранилище данного типа могут храниться только сертификаты. Инициализация такого хранилища может осуществляться одним из следующих способов:

```
KeyStore ks = KeyStore.getInstance("CertStore");

KeyStore ks = KeyStore.getInstance("CertStore", "JCSP");
```

Инициализация хранилища доверенных сертификатов представляет собой инициализацию стандартного хранилища JCA, поэтому для операций записи сертификатов в хранилище или чтения из него требуется предварительная загрузка содержимого хранилища и последующее сохранение содержимого.

2.7.1.2.Загрузка содержимого хранилища

Согласно интерфейсу стандартного хранилища JCA перед началом выполнения каких либо операций требуется загрузка всего содержимого хранилища. Загрузка содержимого стандартного хранилища JCA осуществляется посредством вызова метода *load()* класса KeyStore. Согласно интерфейсу JCA функции *load()* следует передавать два параметра: поток, из которого осуществляется чтение содержимого ключевого хранилища, и пароль на хранилище.

Особенности вызова метода *load()* ввиду общего интерфейса работы с ключевыми носителями и с хранилищем сертификатов подробно описаны выше.

2.7.1.3.Запись сертификата в хранилище

После того, как содержимое хранилища сертификатов было загружено, осуществляется собственно запись доверенного сертификата. Данная операция реализуется при помощи вызова метода *setCertificateEntry()* класса KeyStore следующим образом:

```
String alias;       // идентификатор (уникальное имя) устанавливаемого
                    // в хранилище сертификата
Certificate cert;   // записываемый сертификат
ks.setCertificateEntry(alias, cert);
```

Следует отметить некоторые особенности вызова функции *setCertificateEntry()*. с передачей ему параметра `alias`, являющегося уникальным именем записываемого сертификата.

- если в хранилище уже существует сертификат с именем `alias`, то он будет перезаписан передаваемым сертификатом `cert`;

- если в хранилище нет сертификата с именем `alias`, но на носителе, чьим именем было проинициализировано хранилище сертификатов, существует ключ (и, возможно, сертификат соответствующего открытого ключа) с заданным `alias`, то передаваемый сертификат будет добавлен на этот носитель. При этом будет осуществлена проверка соответствия передаваемого сертификата `cert` ключу, который хранится на носителе с именем `alias` (подробнее см. выше);

- если ни в хранилище, ни на соответствующем ему носителе нет сертификата (на носителе - ключа) с заданным `alias`, то передаваемый сертификат будет просто добавлен в хранилище доверенных сертификатов с именем `alias`.

2.7.1.4.Сохранение содержимого хранилища

Производится аналогично сохранению содержимого ключевого носителя.

2.7.2.Чтение сертификатов из хранилища доверенных сертификатов

Криптопровайдер КриптоПро JCSP позволяет осуществлять чтение доверенных корневых сертификатов из хранилища, определенного пользователем через стандартный интерфейс JCA класса KeyStore, посредством выполнения следующих действий:

- инициализация хранилища доверенных сертификатов;

- загрузка содержимого хранилища;

- чтение сертификата из хранилища;

- сохранение содержимого хранилища .

После того, как содержимое хранилища сертификатов было загружено, осуществляется собственно чтение сертификата из этого хранилища. Данная операция реализуется при помощи вызова метода *getCertificate()* класса KeyStore, возвращающего запрашиваемый сертификат, следующим образом:

```
String alias;   // идентификатор (уникальное имя) сертификата,
                // установленный при записи сертификата в хранилище
Certificate cert = ks.getCertificate(alias);
```

Следует отметить некоторые особенности вызова функции *getCertificate()* с передаваемым параметром `alias`, являющимся уникальным именем запрашиваемого сертификата.

- если в хранилище существует сертификат с заданным именем `alias`, то метод *getCertificate()* вернет сертификат из хранилища сертификатов;

- если в хранилище нет сертификата именем `alias`, но такой сертификат есть на носителе, чьим именем было проинициализировано хранилище сертификатов, то метод *getCertificate()* вернет сертификат с носителя;

- если сертификата с заданным именем `alias` не существует ни в хранилище сертификатов, ни на носителе, то метод *getCertificate()* вернет `null`.

Примечание: Согласно интерфейсу JCA существует возможность загрузки хранилища сертификатов без пароля `ks.load(stream, null)`. При этом провайдер КриптоПро JCSP позволяет осуществлять все операции связанные с чтением и запрещает операции связанные с изменением хранилища (изменять хранилище можно только при загрузке его с паролем).

2.8.Работа с симметричными ключами шифрования, соответствующими алгоритму ГОСТ 28147-89

Криптопровайдер КриптоПро JCSP осуществляет генерацию симметричных ключей шифрования, через стандартный интерфейс JCE при помощи класса KeyGenerator.

2.8.1.Создание объекта генерации симметричных ключей шифрования

Объект генерации симметричных ключей шифрования (далее *генератор*) создается посредством вызова метода *getInstance()* класса KeyGenerator. Этот метод является статическим и возвращает ссылку на класс KeyGenerator, который обеспечивает выполнение требуемой операции.

Для создания генератора симметричных ключей шифрования, соответствующих алгоритму ГОСТ 28147-89 методу *getInstance()* необходимо в качестве параметра передать имя, идентифицирующее данный алгоритм ("GOST28147" или JCSP.GOST_CIPHER_NAME). При таком вызове метода *getInstance()* совместно с определением требуемого алгоритма генерации ключей согласования осуществляется также определение требуемого типа криптопровайдера (КриптоПро JCSP). Также стандартный интерфейс JCE позволяет в качестве параметра функции *getInstance()* класса KeyGenerator вместе с именем алгоритма передавать имя криптопровайдера, используемого для выполнения требуемой операции.

Таким образом, создание генератора симметричных ключей шифрования, соответствующих алгоритму ГОСТ 28147-89 осуществляется одним из следующих способов:

```
KeyGenerator kg = KeyGenerator.getInstance("GOST28147");

KeyGenerator kg = KeyGenerator.getInstance("GOST28147", "JCSP");

KeyGenerator kg = KeyGenerator.getInstance(JCSP.GOST_CIPHER_NAME,
JCSP.PROVIDER_NAME);
```

Генерация симметричных ключей шифрования при помощи такого генератора `kg` будет осуществляться в соответствии с алгоритмом ГОСТ 28147-89 и с установленными в контрольной панели параметрами (параметрами по умолчанию). Если существует необходимость использования другого набора параметров (отличного от параметров по умолчанию), то следует установить требуемый набор параметров созданному генератору.

2.8.2.Определение параметров генерации симметричных ключей шифрования

После того, как генератор симметричных ключей был создан, может возникнуть необходимость установить некий набор параметров генерации симметричных ключей шифрования, отличный от параметров, установленных в контрольной панели. Операция изменения существующего набора параметров осуществляется при помощи метода *init(AlgorithmParameterSpec params)* класса KeyGenerator. Этому методу в качестве параметра может быть передан объект следующих двух классов:

- интерфейс `AlgIdInterface` (определяет набор параметров для генерации ключевой пары обмена, но может быть использован и для генерации симметричных ключей шифрования. В этом случае из передаваемого набора параметров в процессе генерации будут использованы лишь параметры шифрования);

- интерфейс `ParamsInterface` параметров алгоритма шифрования ГОСТ 28147-89 (см. «Работа с параметрами алгоритма шифрования ГОСТ 28147-89»).

В обоих случаях имеют значение лишь параметры шифрования (параметры алгоритма ГОСТ 28147-89), поскольку создаваемый генератором симметричный ключ может быть использован только для шифрования данных (операции создания электронной подписи, ключевого обмена и записи на носитель запрещены).

Таким образом, изменение набора параметров генератора симметричных ключей шифрования производится следующим образом:

```
AlgIdInterface keyParams;     // интерфейс набора параметров

ParamsInterface cryptParams;  // интерфейс параметров шифрования

kg.init(keyParams);           // установка полного набора параметров

kg.init(cryptParams);         // установка параметров шифрования
```

Следует помнить о том, что изменение параметров генерации симметричных ключей имеет смысл только до выполнения непосредственно генерации.

В качестве параметров для инициализации генератора может использоваться любой из параметров шифрования, представленных провайдером: параметры по умолчанию, параметры шифрования 1, параметры шифрования 2, параметры шифрования 3, параметры Оскар 1.1, параметры Оскар 1.0, параметры РИК1, TK26 2, TK26 1, TK26 3, TK26 4, TK26 5, TK26 6, TK26 Z.

2.8.3.Генерация симметричного ключа шифрования

Генерация симметричных ключей шифрования, соответствующих алгоритму ГОСТ 28147-89, осуществляется только после создания генератора и, если это необходимо, определения его параметров. Вызов метода *generateKey()* класса KeyGenerator возвращает ключ шифрования, соответствующий алгоритму ГОСТ 28147-89 и установленному набору параметров (или параметрам по умолчанию):

```
SecretKey key = kg.generateKey();
```

2.9.Имитозащита данных в соответствии с алгоритмом ГОСТ 28147-89

Криптопровайдер КриптоПро JCSP осуществляет имитозащиту данных в соответствии с алгоритмом ГОСТ 28147-89, через стандартный интерфейс JCE при помощи класса Mac.

2.9.1.Создание объекта имитозащиты данных

Объект имитозащиты данных в соответствии с алгоритмом ГОСТ 28147-89 создается посредством вызова метода *getInstance()* класса Mac. Этот метод является статическим и возвращает ссылку на класс Mac, который обеспечивает выполнение требуемой операции.

Для создания объекта имитозащиты в соответствии с алгоритмом ГОСТ 28147-89 методу *getInstance()* необходимо в качестве параметра передать имя, идентифицирующее данный алгоритм ("GOST28147" или JCSP.GOST_CIPHER_NAME). При вызове метода *getInstance()* совместно с определением требуемого алгоритма имитозащиты данных осуществляется также определение требуемого типа криптопровайдера (КриптоПро JCSP). Также стандартный интерфейс JCE позволяет в качестве параметра функции *getInstance()* класса Mac вместе с именем алгоритма указывать имя криптопровайдера, используемого для выполнения требуемой операции.

Таким образом, создание объекта имитозащиты данных осуществляется одним из следующих способов:

```
Mac mac = Mac.getInstance("GOST28147");

Mac mac = Mac.getInstance("GOST28147", "JCSP");

Mac mac = Mac.getInstance(JCSP.GOST_CIPHER_NAME, JCSP.PROVIDER_NAME);
```

2.9.2.Инициализация объекта имитозащиты данных и определение его параметров

После того, как объект имитозащиты данных был создан, необходимо определить набор параметров алгоритма ГОСТ 28147-89, в соответствии с которыми будет осуществляться операция имитозащиты. Определение параметров алгоритма осуществляется во время инициализации операции имитозащиты данных методом *init()* класса Mac одним из следующих способов:

- при вызове функции *init(Key key)* параметры алгоритма ГОСТ 28147-89 определяются в соответствии с параметрами шифрования передаваемого функции ключа `key`;
- при вызове функции *init(Key key, AlgorithmParameterSpec params)* параметры алгоритма ГОСТ 28147-89 определяются в соответствии с передаваемыми параметрами `params`.

В обоих случаях передаваемый ключ `key` используется в процессе выработки имитовставки.

Существуют некоторые ограничения на параметры, передаваемые функции *init()*. В обоих способах вызова этой функции ключ `key` должен соответствовать типу SecretKey и удовлетворять алгоритму шифрования ГОСТ 28147-89 (такие ключи могут быть получены при помощи операции согласования ключей или при помощи генерации симметричных ключей шифрования). Необходимо также, чтобы параметры, передаваемые функции *init()* во втором способе ее вызова соответствовали интерфейсу `ParamsInterface` параметров алгоритма шифрования ГОСТ 28147-89 (см. «Работа с параметрами алгоритма шифрования ГОСТ 28147-89»). Таким образом, инициализация операции имитозащиты данных, во время которой происходит определение параметров имитовставки, осуществляется одним из следующих способов:

```
ParamsInterface cryptParams;    // интерфейс параметров шифрования

SecretKey key;        // обязательно ключ шифрования с алгоритмом "GOST28147"
                      // (соответствующий алгоритму ГОСТ 28147-89)

mac.init(key);

mac.init(key, cryptParams);
```

2.9.3.Копирование объекта имитозащиты данных

В некоторых случаях требуется создать копию уже существующего объекта имитозащиты данных, например, когда требуется выработать имитовставку как и части данных, так и всего исходного массива данных. В этом случае после того, как была обработана требуемая часть данных, необходимо сохранить (при помощи копирования) объект имитозащиты, и продолжить обработку оставшейся части (в результате чего будут обработаны все исходные данные). Уже после выполняется вычисление значения имитовставки для обоих объектов (исходного - соответствующего всем данным и скопированного - соответствующего части данных).

Для этих целей используется метод *clone()* класса Mac, который возвращает точную копию существующего объекта имитозащиты. Этот метод может быть вызван на любом этапе выполнения операции выработки имитовставки после того, как объект имитозащиты был проинициализирован и до того, как операция имитозащиты данных была завершена.

2.9.4.Выработка имитовставки данных

После того, как объект имитозащиты был создан и проинициализирован, выработка имитовставки данных в соответствии с алгоритмом ГОСТ 28147-89 производится в два этапа: обработка данных и последующее вычисление значения имитовставки, завершающее операцию имитозащиты данных.

2.9.4.1.Обработка защищаемых данных

Обработка защищаемых данных осуществляется при помощи метода *update()* класса Mac. Этот метод осуществляет обработку защищаемых данных, представленных в виде байтового массива и подаваемых ему в качестве параметра. Существует 3 варианта обработки байтового массива данных при помощи этого метода:

1. Последовательная обработка каждого байта данных (при этом количество вызовов метода *update(byte b)* равно длине массива данных):

```
byte[] data;
for(int i = 0; i < data.length; i++)
    mac.update(data[i]);
```

2. Блочная обработка данных (данные обрабатываются блоками определенной длины):

```
byte[] data;
int BLOC_LEN = 1024;

//если длина исходных данных меньше длины блока
if(data.length/BLOC_LEN == 0)
    mac.update(data);
else {
    //цикл по блокам
    for (int i = 0; i < data.length/BLOC_LEN; i++) {
        byte[] bloc = new byte[BLOC_LEN];
        for(int j = 0; j < BLOC_LEN; j++)
            bloc[j] = data[j + i * BLOC_LEN];
        mac.update(bloc);
    }

    //обработка остатка
    byte[] endBloc = new byte[data.length % BLOC_LEN];
    for(int j = 0; j < data.length % BLOC_LEN; j++)
        endBloc[j] = data[j + data.length - data.length % BLOC_LEN];
    mac.update(endBloc);
}
```

3. Обработка данных целиком:

```
byte[] data;
mac.update(data);
```

Допускается комбинирование первого и второго варианта, обработка блоками различной длины, а также использование метода *update(byte[]data, int offset, int len)* - обработка массива данных со смещением.

2.9.4.2.Вычисление значения имитовставки

Вычисление значения имитовставки осуществляется при помощи метода *doFinal()* класса Mac. Существуют различные варианты вызова данного метода. Если обработке подверглись все

защищаемые данные, то остается только получить значение имитовставки. Получить это значение можно двумя способами:

1. Вызов метода без параметров - *doFinal()*. В этом случае метод возвращает байтовый массив, содержащий значение имитовставки;
2. Вызов метода с параметрами - *doFinal(byte[] buf, int offset)*. В этом случае метод записывает значение имитовставки в передаваемый ему массив со смещением.

Если же часть данных осталось не обработанной, то следует предварительно обработать ее, а лишь потом получать значение имитовставки. Для этих целей используется функция *doFinal(byte[] buf)*, обрабатывающая переданные в массиве buf данные и возвращающая байтовый массив, содержащий значение имитовставки.

2.10.Использование алгоритма HMAC

С помощью криптопровайдера КриптоПро JCSP можно осуществлять защиту данных в соответствии с алгоритмом HMAC_GOSTR3411, HMAC_GOSTR3411_2012_256 (OID: 1.2.643.7.1.1.4.1) или HMAC_GOSTR3411_2012_512 (OID: 1.2.643.7.1.1.4.2). Стандарт HMAC_GOSTR3411 описан в документе rfc4357, который базируется на стандарте rfc2104. OID "1.2.643.2.2.10" для алгоритма определен в rfc4490

Работать с реализацией алгоритма можно через стандартный интерфейс JCE при помощи класса Mac. Использование HMAC_GOSTR3411, HMAC_GOSTR3411_2012_256 или HMAC_GOSTR3411_2012_512 аналогично работе с имитозащитой по алгоритму ГОСТ 28147-89, только при создании объекта в качестве имени надо указывать "HMAC_GOSTR3411". Примеры создания объекта:

```
Mac mac = Mac.getInstance("HMAC_GOSTR3411");

Mac mac = Mac.getInstance("HMAC_GOSTR3411", "JCSP");

Mac mac = Mac.getInstance(ru.CryptoPro.JCSP.Digest.JCSPGostHMAC.STR_NAME,
JCSP.PROVIDER_NAME);

Mac mac = Mac.getInstance("1.2.643.2.2.10", "JCSP");
```

или

```
Mac mac = Mac.getInstance("HMAC_GOSTR3411_2012_256");

Mac mac = Mac.getInstance("HMAC_GOSTR3411_2012_256", "JCSP");

Mac mac =
Mac.getInstance(ru.CryptoPro.JCSP.Digest.JCSPGostHMAC2012_256.STR_NAME,
JCSP.PROVIDER_NAME);

Mac mac = Mac.getInstance("1.2.643.7.1.1.4.1", "JCSP");
```

или

```
Mac mac = Mac.getInstance("HMAC_GOSTR3411_2012_512");

Mac mac = Mac.getInstance("HMAC_GOSTR3411_2012_512", "JCSP");

Mac mac =
Mac.getInstance(ru.CryptoPro.JCSP.Digest.JCSPGostHMAC2012_512.STR_NAME,
JCSP.PROVIDER_NAME);

Mac mac = Mac.getInstance("1.2.643.7.1.1.4.2", "JCSP");
```

2.10.1.Использование 512-битных ключей в алгоритмах HMAC

В функциях HMAC, основанных на хэш-функциях ГОСТ Р 34.11-2012 (256 бит) и ГОСТ Р 34.11-2012 (512 бит), могут быть использованы 512-битные ключи. Эти ключи представляются в виде объектов класса Symmetric512Key.

Ключи Symmetric512Key могут быть получены как путем выработки с помощью алгоритма ВКО (см. соответствующий раздел), так и с помощью объекта генератора JCSPSymmetric512KeyGenerator.

```
KeyGenerator kg = KeyGenerator.getInstance("SYMMETRIC512");

KeyGenerator kg = KeyGenerator.getInstance("SYMMETRIC512", "JCSP");

KeyGenerator kg = KeyGenerator.getInstance(JCSP.SYMMETRIC_512,
JCSP.PROVIDER_NAME);
```

Непосредственно выработка ключа производится так:

```
kg.init(null);
SecretKey key = kg.generateKey();
```

После этого ключ можно использовать в HMAC:

```
Mac mac;
Symmetric512Key key;
mac.init(key);
```

2.11.Шифрование данных и ключей в соответствии с алгоритмом ГОСТ 28147-89

Криптопровайдер КриптоПро JCSP осуществляет шифрование данных и ключей в соответствии с алгоритмом ГОСТ 28147-89, через стандартный интерфейс JCE при помощи класса Cipher.

2.11.1.Создание объекта шифрования данных и ключей (шифратора)

Объект шифрования данных и ключей (далее *шифратор*) в соответствии с алгоритмом ГОСТ 28147-89 создается посредством вызова метода *getInstance()* класса Cipher. Этот метод является статическим и возвращает ссылку на класс Cipher, который обеспечивает выполнение требуемой операции.

Следуя интерфейсу класса Cipher, методу *getInstance()* в качестве параметра следует передавать строковое представление алгоритма шифратора вида *"algorithm/mode/padding"* либо *"algorithm"*. Ниже описаны допустимые варианты значений каждого из элементов такого строкового представления.

1. в качестве элемента *"algorithm"* указывается собственно алгоритм шифрования. Алгоритм ГОСТ 28147-89 идентифицируется именем "GOST28147" или JCSP.GOST_CIPHER_NAME.
2. в качестве элемента *"mode"* указывается режим шифрования. В криптопровайдере КриптоПро JCSP допустимы следующие режимы:
   - "ECB" - режим простой замены (используется при шифровании данных);
   - "CBC" - режим простой замены с зацеплением (используется при шифровании данных);
   - "CNT" - режим гаммирования (используется при шифровании данных);
   - "CFB" - режим гаммирования с обратной связью (используется при шифровании данных);
   - "SIMPLE_EXPORT" - режим шифрования симметричного ключа без усложнения (используется при шифровании ключей);
   - "PRO_EXPORT" - режим шифрования симметричного ключа с усложнением (используется при шифровании ключей);
   - "PRO12_EXPORT" - режим шифрования симметричного ключа с усложнением (используется при шифровании ключей).

Если элемент *"mode"* отсутствует (то есть в качестве строкового представления алгоритма шифратора указан лишь алгоритм шифрования), то при осуществлении последующей операции шифрования будет использован режим по умолчанию: для операций шифрования данных (см. ниже) используется режим "CFB", для операций шифрования ключей - режим "SIMPLE_EXPORT".

3. в качестве элемента *"padding"* указывается алгоритм заполнения последнего неполного блока данных при выполнении операции блочного шифрования. В криптопровайдере КриптоПро JCSP допустимы следующие алгоритмы:
   - "NoPadding" - заполнение последнего блока не используется (использование допускается только в режимах "CNT" и "CFB");
   - "PKCS5_PADDING" - заполнение последнего блока осуществляется по алгоритму PKCS5;
   - "PKCS5Padding" - другой допустимый вариант написания алгоритма PKCS5;

- "ISO10126Padding" - заполнение последнего блока осуществляется по алгоритму ISO 10126;
- "ANSIX923Padding" - заполнение последнего блока осуществляется по алгоритму ANSI X.923;
- "ZeroPadding" - заполнение последнего блока нулями, при кратности исходного открытого текста размеру блока шифра дополнительный блок не добавляется, паддинг не является однозначным;
- "RandomPadding" - заполнение последнего блока случайными байтами, при кратности исходного открытого текста размеру блока шифра дополнительный блок не добавляется, паддинг не является однозначным.

Если элемент *"padding"* отсутствует (то есть в качестве строкового представления алгоритма шифратора указан лишь алгоритм шифрования), то при осуществлении последующей операции шифрования будет использован алгоритм "PKCS5Padding".

Следует помнить, что установление отличного от "NoPadding" алгоритма заполнения имеет смысл только при использовании режимов "ECB" и "CBC". Если используются режимы "CNT" или "CFB", то установленный режим заполнения будет проигнорирован. В случае выполнения шифрования в режимах "ECB" и "CBC" отсутствие заполнения ("NoPadding") не допускается.

При вызове метода *getInstance()* со строковым представлением алгоритма, состоящем из перечисленных выше допустимых значений имени алгоритма, режима шифрования и алгоритма заполнения последнего неполного блока, помимо определения в соответствии с этим строковым представлением алгоритма работы шифратора осуществляется также определение требуемого типа криптопровайдера (КриптоПро JCSP). Также стандартный интерфейс JCE позволяет в качестве параметра функции *getInstance()* класса Cipher вместе с именем алгоритма указывать имя криптопровайдера, используемого для выполнения требуемой операции. Таким образом, создание, например, объекта шифрования данных в соответствии с алгоритмом ГОСТ 28147-89 в режиме простой замены с зацеплением с алгоритмом PKCS5 заполнения последнего неполного блока осуществляется одним из следующих способов:

```
Cipher cipher = Cipher.getInstance("GOST28147/CBC/PKCS5_PADDING");
```

```
Cipher cipher = Cipher.getInstance("GOST28147/CBC/PKCS5_PADDING", "JCSP");
```

2.11.2.Инициализация шифратора и определение его параметров

После того, как шифратор был создан, необходимо определить собственно метод шифрования и набор параметров, в соответствии с которым будет осуществляться операция шифрования. Инициализация шифратора осуществляется посредством вызова метода *init()* класса Cipher. Вызов данного метода может быть осуществлен двумя способами:

- *init(int opmode, Key key)*;
- *init(int opmode, Key key, AlgorithmParameterSpec params)*.

Ниже описываются особенности каждого из этих способов для криптопровайдера КриптоПро JCSP.

- для обоих случаев вызова функции *init()* в качестве параметра opmode передается число, определяющее собственно метод шифрования. Криптопровайдер КриптоПро JCSP поддерживает четыре метода шифрования, определяемых следующим образом:
  - метод зашифрования данных (определяется константой ENCRYPT_MODE класса Cipher);
  - метод расшифрования данных (определяется константой DECRYPT_MODE класса Cipher);
  - метод зашифрования ключа (определяется константой WRAP_MODE класса Cipher);
  - метод расшифрования ключа (определяется константой UNWRAP_MODE класса Cipher);
- для обоих случаев в качестве параметра key передается ключ, на котором будет производится указанная операция шифрования. Требуется, чтобы этот ключ являлся объектом типа SecretKey и удовлетворял алгоритму шифрования ГОСТ

28147-89 (такие ключи могут быть получены при помощи алгоритма выработки ключей парной связи или при помощи генерации симметричных ключей шифрования).

- поскольку в первом способе вызова функции *init(int opmode, Key key)* никакие дополнительные параметры не передаются, то такой вызов функции обладает следующими особенностями:
  - параметры алгоритма шифрования (узел замены) определяются в соответствии с параметрами шифрования передаваемого ключа `key`;
  - при инициализации операции зашифрования данных (`ENCRYPT_MODE`) в режиме, требующем вектора инициализации (такими режимами являются "CBC", "CFB" и "CNT") вектор инициализации будет выработан случайным образом;
  - при инициализации операции расшифрования данных (`DECRYPT_MODE`) в режиме, требующем вектора инициализации (такими режимами являются "CBC", "CFB" и "CNT") будет выдано исключение;
  - при инициализации операции шифрования ключа (`WRAP_MODE` или `UNWRAP_MODE`) ключом `key`, полученным при помощи операции согласования ключей, необходимый для дальнейшего выполнения операции шифрования вектор инициализации будет пронаследован от стартового вектора, который участвовал в процессе создания ключа согласования `key`;
  - при инициализации операции зашифрования ключа (`WRAP_MODE`) ключом `key`, полученным при помощи генерации симметричных ключей шифрования, необходимый для дальнейшего выполнения операции шифрования вектор инициализации будет выработан случайным образом;
  - при инициализации операции расшифрования ключа (`UNWRAP_MODE`) ключом `key`, полученным при помощи генерации симметричных ключей шифрования, будет выдано исключение.
- во втором способе вызова функции *init(int opmode, Key key, AlgorithmParameterSpec params)* передаются параметры `params`, определяющие работу шифратора. В качестве таких параметров в криптопровайдере КриптоПро JCSP допускается передавать объекты следующих типов:
  - интерфейс `ParamsInterface` параметров алгоритма шифрования ГОСТ 28147-89 (описание работы с этим интерфейсом приводится в соответствующем разделе документации). Таким образом, объект типа `ParamsInterface`, передаваемый в качестве параметров `params` определяет параметры шифрования (узел замены)
  - IvParameterSpec - стандартный класс JCE. Объект такого класса, переданный в качестве параметров `params` определяет вектор инициализации.
  - `GostCipherSpec` - этот класс представляет собой набор параметров алгоритма ГОСТ 28147-89, а также вектор инициализации шифратора, и является реализацией стандартного класса AlgorithmParameterSpec. Создание объекта класса `GostCipherSpec` в соответствии с требуемыми параметрами (узлом) шифрования и вектором инициализации производится следующим образом:

```
ParamsInterface cryptParams; // интерфейс параметров шифрования
byte[] iv;                   //вектор инициализации
GostCipherSpec spec = new GostCipherSpec(iv, cryptParams);
```

Таким образом, объект типа GostCipherSpec, передаваемый в качестве параметров params определяет параметры шифрования (узел замены) и вектор инициализации.

- ввиду поддерживаемых типов объектов, передаваемых в качестве параметров `params` для второго способа вызова функции *init(int opmode, Key key, AlgorithmParameterSpec params)* возникают следующие особенностями такого вызова функции:

  - если параметры алгоритма шифрования (узел замены) заданы (в случае передачи в качестве параметров `params` объекта интерфейса параметров шифрования `ParamsInterface` или класса `GostCipherSpec`), то они и будут использованы в процессе выполнения операции шифрования. В противном случае, параметры шифрования определяются в соответствии с параметрами шифрования передаваемого ключа `key`;

  - если вектор инициализации задан (в случае передачи в качестве параметров `params` объекта класса `IvParameterSpec` или класса `GostCipherSpec`), то он и будет использован в процессе выполнения операции шифрования. В противном случае:

    - при инициализации операции зашифрования данных (`ENCRYPT_MODE`) в режиме, требующем вектора инициализации (такими режимами являются "CBC", "CFB" и "CNT") вектор инициализации будет сгенерирован случайным образом;

    - при инициализации операции расшифрования данных (`DECRYPT_MODE`) в режиме, требующем вектора инициализации (такими режимами являются "CBC", "CFB" и "CNT") будет выдано исключение;

    - при инициализации операции шифрования ключа (`WRAP_MODE` или `UNWRAP_MODE`) ключом `key`, полученным при помощи операции согласования ключей, необходимый для дальнейшего выполнения операции шифрования вектор инициализации будет пронаследован от стартового вектора, который участвовал в процессе создания ключа согласования `key`;

    - при инициализации операции зашифрования ключа (`WRAP_MODE`) ключом `key`, полученным при помощи генерации симметричных ключей шифрования, необходимый для дальнейшего выполнения операции шифрования вектор инициализации будет выработан случайным образом;

    - при инициализации операции расшифрования ключа (`UNWRAP_MODE`) ключом `key`, полученным при помощи генерации симметричных ключей шифрования, будет выдано исключение.

В обоих случаях необходимо следить за согласованностью параметров шифрования (узла шифрования), а также вектора инициализации при выполнении соответствующих друг другу операций зашифрования и расшифрования. Так, например, если инициализация операции зашифровании данных производилась без передачи вектора инициализации, то созданный в процессе выполнения этой операции вектор инициализации следует передать вместе с зашифрованным текстом для корректного выполнения операции расшифрования. Получить вектор инициализации можно при помощи метода *getIV()*, возвращающего его в виде байтового массива. Аналогичные действия следует производить в случае зашифрования ключа на симметричном ключе шифрования, полученным при помощи генерации. Необходимо помнить, что вызов этой функции следует производить только после того, как шифратор был проинициализирован, и до того как операция зашифрования была завершена.

Таким образом, инициализация, соответствующих друг другу операций зашифрования и расшифрования данных для алгоритма ГОСТ 28147-89 может быть осуществлена следующим образом:

```
SecretKey key;    // обязательно ключ шифрования с алгоритмом "GOST28147"
```

```
                    // (соответствующий алгоритму ГОСТ 28147-89)
                    // считаем, что известен обоим сторонам
/*Зашифрование*/
// инициализация операции зашифрования. Вектор будет сгенерирован
// в процессе выполнения операции
cipher.init(Cipher.ENCRYPT_MODE, key);
// получение сгенерированного вектора инициализации
byte[] iv = cipher.getIV();
/*Расшифрование*/
// создание параметров, содержащих требуемый вектор
IvParameterSpec spec = new IvParameterSpec(iv);
// инициализация операции расшифрования с тем же
// ключом и вектором
cipher.init(Cipher.DECRYPT_MODE, key, spec);
```

2.11.3.Зашифрование и расшифрование данных

После того, как шифратор был создан и проинициализирован методом шифрования данных (ENCRYPT_MODE или DECRYPT_MODE), операция зашифрования (расшифрования) данных производится в два этапа: обработка данных и последующее завершение операции.

2.11.3.1.Последовательное зашифрование (расшифрование) данных

Последовательное зашифрование (расшифрование) данных осуществляется при помощи метода *update()* класса Cipher. Этот метод выполняет последовательное зашифрование (расшифрование) данных, представленных в виде байтового массива и подаваемых ему в качестве параметра. Существует четыре способа вызова метода:

- *update(byte[] input)* - зашифрование (расшифрования) всего содержимого массива `input`. Результат шифрования выдается в качестве байтового массива;
- *update(byte[] input, int inputOffset, int inputLen)* - зашифрование (расшифрования) содержимого массива `input`, начиная с позиции `inputOffset` в количестве `inputLen` байт. Результат шифрования выдается в качестве байтового массива;
- *update(byte[] input, int inputOffset, int inputLen, byte[] output)* - зашифрование (расшифрования) содержимого массива `input`, начиная с позиции `inputOffset` в количестве `inputLen` байт. Результат шифрования записывается в массив `output`. Такой вызов функции возвращает количество записанных в выходной буфер байт;
- *update(byte[] input, int inputOffset, int inputLen, byte[] output, int outputOffset)* - зашифрование (расшифрования) содержимого массива `input`, начиная с позиции `inputOffset` в количестве `inputLen` байт. Результат шифрования записывается в массив `output`, начиная с позиции `outputOffset`. Такой вызов функции возвращает количество записанных в выходной буфер байт;

Допускается комбинирование этих методов, причем подаваемые на каждый вызов функции *update()* массивы зашифровываемых (расшифровываемых) данных могут быть разной длины.

2.11.3.2.Завершение операции зашифрования (расшифрования)

Завершение операции зашифрования (расшифрования) данных осуществляется при помощи метода *doFinal()* класса Cipher. Даже если все входные данные были поданы на последовательное шифрование, то это не гарантирует того, что результат шифрования был выдан полностью. Такое, например, возможно при зашифровании в режимах CBC и ECB данных, длина которых не кратна длине блока. Ввиду этого операцию шифрования следует завершить (для получения последней части результата шифрования, либо же для обработки и получения конечного результата еще не обработанных входных данных).

Если в процессе последовательного шифрования все входные данные были поданы на зашифрование (расшифрование), то завершить операцию шифрования можно двумя способами:

- вызов метода без параметров - *doFinal()*. В этом случае метод возвращает байтовый массив, содержащий последнюю часть зашифрованных (расшифрованных) данных;
- вызов метода с параметрами - *doFinal(byte[] output, int outputOffset)*. В этом случае метод записывает последнюю часть зашифрованных (расшифрованных) данных в передаваемый ему массив со смещением.

Если же не все входные данные были поданы на последовательное шифрование, то следует обработать оставшиеся данные, и лишь потом завершать операцию шифрования для получения конечного результата. Для этих целей может быть использовано несколько вариантов вызова функции *doFinal*:

- *doFinal(byte[] input)* - обрабатывает последнюю часть входных данных, содержащуюся в массиве `input` и выдает окончательный результат шифрования в виде байтового массива;
- *doFinal(byte[] input, int inputOffset, int inputLen)* - обрабатывает последнюю часть входных данных, содержащуюся в массиве `input`, начиная с позиции `inputOffset` в количестве `inputLen` и выдает окончательный результат шифрования в виде байтового массива;
- *doFinal(byte[] input, int inputOffset, int inputLen, byte[] output)* - обрабатывает последнюю часть входных данных, содержащуюся в массиве `input`, начиная с позиции `inputOffset` в количестве `inputLen` и записывает окончательный результат шифрования в массив `output`. При этом метод возвращает количество записанных в выходной буфер байт;
- *doFinal(byte[] input, int inputOffset, int inputLen, byte[] output, int outputOffset)* - обрабатывает последнюю часть входных данных, содержащуюся в массиве `input`, начиная с позиции `inputOffset` в количестве `inputLen` и записывает окончательный результат шифрования в массив `output`, начиная с позиции `outputOffset`. При этом метод возвращает количество записанных в выходной буфер байт.

2.11.4.Зашифрование и расшифрование ключей

После того, как шифратор был создан и проинициализирован методом шифрования ключа (`WRAP_MODE` или `UNWRAP_MODE`), выполняется собственно операция зашифрования или расшифрования ключа.

2.11.4.1.Зашифрование ключа

Операция зашифрования ключа выполняется методом *wrap(Key key)* класса Cipher. В качестве параметра данному методу подается ключ `key`, который подлежит зашифрованию. Зашифрование переданного ключа производится на ключе, которым был проинициализирован шифратор. Криптопровайдер КриптоПро JCSP допускает зашифрование только следующих типов ключей:

- закрытых ключей обмена или ключей электронной подписи, соответствующих алгоритмам обмена Диффи-Хеллмана и ЭЦП ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012 (такие ключи могут быть прочитаны из контейнера, либо могут быть получены посредством генерации ключевой пары в соответствии с данными алгоритмами);
- симметричных ключей шифрования, соответствующих алгоритму ГОСТ 28147-89 (такие ключи могут быть получены при генерации симметричных ключей).

Зашифрованный ключ возвращается в виде байтового массива.

2.11.4.2.Расшифрование ключа

Операция расшифрования ключа выполняется методом *unwrap(byte[] wrappedKey, String wrappedKeyAlgorithm, int wrappedKeyType)* класса Cipher. Этот метод возвращает расшифрованный ключ. В качестве параметра данному методу передается зашифрованный ключ `wrappedKey`, представленный в виде байтового массива. Параметр `wrappedKeyAlgorithm` в криптопровайдере КриптоПро JCSP не используется. Расшифрование ключа производится на ключе, которым был проинициализирован шифратор. Криптопровайдер КриптоПро JCSP допускает расшифрование только симметричных ключей шифрования (соответствующих типу SecretKey и удовлетворяющих алгоритму шифрования ГОСТ 28147-89). Ввиду этого в качестве параметра `wrappedKeyType` допускает использование только типа ключа, соответствующего классу SecretKey. Тип такого ключа определяется константой SECRET_KEY класса Cipher.

2.12.Диверсификация ключей

Для выработки ключей шифрования и ключевого хэширования для блоков (пакетов) данных возможно использование алгоритмов диверсификации ключей. Алгоритм диверсификации ключа принимает на вход ключ (это может быть как симметричный ключ шифрования, так и ключи подписи и обмена, в том числе и долговременные), дополнительные данные для диверсификации (набор байтов) и возвращает симметричный ключ ГОСТ 28147-89. В КриптоПро JCSP реализовано два алгоритма диверсификации: PRO_DIVERS (п.7 RFC 4357) и PRO12_DIVERS (п. 5.4 рекомендаций по стандартизации "Использование криптографических алгоритмов, соответствующих применению стандартов ГОСТ Р 34.10-2012 И ГОСТ Р 34.11-2012", утвержденных ТК 26 "Криптографическая защита информации").

Для осуществления диверсификации ключа используется класс SecretKeyFactory. Для получения объекта этого класса следует использовать метод *getInstance().*

```
SecretKeyFactory secretKeyFactory = SecretKeyFactory.getInstance("GOST28147");
```

```
SecretKeyFactory secretKeyFactory = SecretKeyFactory.getInstance("GOST28147", “JCSP”);
```

```
SecretKeyFactory secretKeyFactory = SecretKeyFactory.getInstance(JCSP.GOST_CIPHER_NAME, JCSP.PROVIDER_NAME);
```

Данные, необходимые для диверсификации, передаются в виде объекта класса *DiversKeySpec*. Объект класса можно создать следующим образом:

```
DiversKeySpec diversKeySpec = new DiversKeySpec(key, data, diversAlg, dwMagic),
```

где key – объект ключа, который будет диверсифицирован, data – массив байтов, содержащий данные для диверсификации, diversAlg – параметр, указывающий на алгоритм диверсификации (DiversKeySpec.PRO_DIVERS и DiversKeySpec.PRO12_DIVERS соответственно), dwMagic – дополнительные данные для диверсификации в виде int (только для алгоритма PRO12_DIVERS, алгоритм PRO_DIVERS эти данные проигнорирует).

Для непосредственно выработки симметричного ключа ГОСТ 28147-89 необходимо выполнить команду:

```
SecretKey secretKey = secretKeyFactory.generateSecretKey(diversKeySpec);
```

Необходимо отметить, что алгоритм PRO_DIVERS не может работать с ключами ГОСТ Р 34.10-2012, а также принимать на вход данные для диверсификации короче 4 байт и длиннее 40 байт. При попытке использования неправильных ключей или данных будет инициировано исключение *IllegalArgumentException*.

2.13.Генерация случайных чисел

Криптопровайдер КриптоПро JCSP позволяет осуществлять генерацию случайных чисел, через стандартный интерфейс JCA при помощи класса SecureRandom. Для этого происходит обращение к датчику случайных чисел КриптоПро CSP.

2.13.1.Создание генератора случайных чисел

Объект генератора случайных чисел создается посредством вызова метода *getInstance()* класса SecureRandom. Этот метод является статическим и возвращает ссылку на созданный объект класса SecureRandom.

Для создания генератора методу *getInstance()* необходимо в качестве параметра передать имя, идентифицирующее алгоритм ("CPRandom" или JCP.CP_RANDOM). При таком вызове метода *getInstance()* совместно с определением требуемого алгоритма осуществляется также определение требуемого типа криптопровайдера (КриптоПро JCSP). Стандартный интерфейс JCA позволяет в качестве параметра функции *getInstance()* класса SecureRandom вместе с именем алгоритма указывать имя криптопровайдера, используемого для выполнения операции. Таким образом, создание генератора осуществляется одним из следующих способов:

```
SecureRandom rnd = SecureRandom.getInstance("CPRandom");
```

```
SecureRandom rnd = SecureRandom.getInstance("CPRandom", "JCSP");
```

```
SecureRandom rnd = SecureRandom.getInstance(JCP.CP_RANDOM, JCSP.PROVIDER_NAME);
```

2.13.2.Использование генератора случайных чисел

Некоторые функции JCA предусматривают возможность установки необходимого SecureRandom для выполнения конкретных операций. Например, можно установить конкретный генератор случайных чисел в класс Signature при создании электронной подписи с помощью функции

```
void initSign(PrivateKey privateKey, SecureRandom random).
```

При генерации ключевой пары в класс KeyPairGenerator можно установить конкретный генератор функцией

```
void initialize(AlgorithmParameterSpec params, SecureRandom random).
```

Однако, чтобы обеспечить необходимое качество случайных последовательностей, КриптоПро JCSP **игнорирует** генераторы, переданные таким способом в качестве параметров. Поэтому для увеличения производительности не стоит создавать новые генераторы только для того, чтобы проинициализировать ими другие классы JCA/JCE.

Для других целей, после того, как генератор случайных чисел был создан, можно получить случайную последовательность функцией void nextBytes(byte[] bytes)

2.13.3.Доинициализация датчика

Метод public byte[] generateSeed(int numBytes) класса SecureRandom используется в JCA для доинициализации датчика дополнительной энтропией. В случае использования криптопровайдера КриптоПро JCSP данный метод не производит никаких действий; все необходимые манипуляции с датчиком производятся криптопровайдером КриптоПро CSP самостоятельно.

2.13.4.Возможные ошибки датчика

В процессе работы генератор случайных чисел КриптоПро JCSP контролирует качество выходной последовательности и проводит периодический контроль целостности. В случае обнаружения нарушения целостности, генератор возбуждает исключение *ru.CryptoPro.JCP.Random.RandomRefuseException.* Возникновение этой ошибки возможно в любом месте, где используется генератор, например, при генерации ключ или выработке подписи. Использование криптопровайдера КриптоПро JCSP в этом случае не допускается.

2.13.5.Биологический датчик

При генерировании ключевой пары с помощью класса KeyPairGenerator для гарантии качества выходной последовательности (и, следовательно, секретного ключа) генератор случайных чисел необходимо доинициализировать дополнительной энтропией. Для этого криптопровайдер КриптоПро CSP представляет специальной окно или консоль для получения энтропии от нажатия таст клавиатуры и движения мыши.

2.14.Работа с сертификатами через стандартный интерфейс JCA

Для осуществления операций, реализуемых криптопровайдером КриптоПро JCSP, зачастую требуется использование стандартных методов JCA работы с сертификатами. Такими операциями, например, являются:

- запись ключа электронной подписи или ключевого обмена на носитель;
- запись на носитель сертификата открытого ключа, в соответствии с хранящимся на носителе ключом, и сертификата с носителя;
- запись сертификата в хранилище доверенных сертификатов и чтение сертификата из хранилища;
- проверка электронной подписи при помощи ключа проверки электронной подписи, читаемого из сертификата;
- построение и проверка цепочек сертификатов.

Поскольку криптопровайдер КриптоПро JCSP не реализует стандартные методы работы с сертификатами, а лишь обеспечивает их поддержку, то в данной документации приводится лишь описание использования этих методов при выполнении перечисленных выше операций.

2.14.1.Генерация X509-сертификатов

Генерация X509-сертификатов осуществляется при помощи метода *generateCertificate()* класса CertificateFactory следующим образом:

```
InputStream inStream;

CertificateFactory cf = CertificateFactory.getInstance("X509");

//или

//CertificateFactory cf =
CertificateFactory.getInstance(JCP.CERTIFICATE_FACTORY_NAME);

Certificate cert = cf.generateCertificate(inStream);
```

Метод *generateCertificate()* получает в качестве параметра входной поток, в который записан закодированный в DER-кодировке сертификат, и возвращает объект класса Certificate. Инициализацию объекта класса CertificateFactory следует производить именем "X509" или JCP.CERTIFICATE_FACTORY_NAME (как показано выше). В этом случае выдаваемый методом *generateCertificate()* сертификат будет удовлетворять стандарту X.509, а значит являться объектом класса X509Certificate (этот класс является расширением класса Certificate). Криптопровайдер КриптоПро JCSP поддерживает только стандарт X.509.

Генерация X509-сертификатов используется в тех операциях, которые согласно стандартному интерфейсу JCA требуют для своего выполнения объекты класса Certificate. Такими

операциями являются, например, запись ключа электронной подписи или ключевого обмена на носитель и запись сертификата в хранилище доверенных сертификатов или на носитель.

Закодированный в DER-кодировке сертификат, передаваемый функции *generateCertificate()* во входном потоке может быть получен при помощи методов класса `GostCertificateRequest` (см. раздел «Дополнительные возможности работы с сертификатами»). Получение закодированного сертификата при помощи класса `GostCertificateRequest` используется в тех случаях, когда требуется соответствие открытого ключа сертификата только что созданному ключу электронной подписи или ключевого обмена (например, при осуществлении записи этого ключа на носитель). Если же этот ключ не известен (например, при проверке электронной подписи), либо секретный ключ, которому соответствует открытый ключ сертификата был создан ранее (например, при записи сертификата на носитель, на котором уже существует секретный ключ), то в этом случае закодированный сертификат, передаваемый функции *generateCertificate()*, может быть прочитан из файла.

2.14.2.Кодирование сертификата в DER-кодировку

Кодирование существующего сертификата (объекта класса Certificate) осуществляется при помощи метода *getEncoded()* класса Certificate следующим образом:

```
Certificate cert;

byte[] encoded = cert.getEncoded();
```

Закодированный в DER-кодировке методом *getEncoded()* сертификат возвращается в виде байтового массива. Сертификат `cert`, для которого осуществляется кодирование, может быть получен различными методами: генерацией X509-сертификата, чтением сертификата открытого ключа с носителя, либо чтением доверенного сертификата из хранилища (все эти методы возвращают объект класса Certificate).

Операция кодирования сертификата используется в случае, когда требуется сохранить в файл только что сгенерированный или прочитанный с носителя (или из хранилища) сертификат.

2.14.3.Получение открытого ключа из сертификата

Получение открытого ключа из сертификата (объекта класса Certificate) осуществляется при помощи метода *getPublicKey()* класса Certificate следующим образом:

```
Certificate cert;

PublicKey publicKey = cert.getPublicKey();
```

Сертификат `cert`, из которого получается открытый ключ `publicKey`, может быть получен различными методами: генерацией X509-сертификата, чтением сертификата открытого ключа с носителя, либо чтением доверенного сертификата из хранилища (все эти методы возвращают объект класса Certificate).

Операция получения открытого ключа из сертификата используется при осуществлении проверки электронной подписи.

2.14.4.Построение и проверка цепочки сертификатов

При выполнении операции записи ключа на носитель согласно стандартному интерфейсу JCA вместе с ключом на носитель следует записывать и цепочку сертификатов, начинающуюся с сертификата открытого ключа, соответствующего секретному ключу и заканчивающуюся доверенным корневым сертификатом. Благодаря этому требованию подпись сертификата открытого ключа, соответствующего записываемому секретному ключу, всегда заверена цепочкой сертификатов.

Цепочка, как уже говорилось выше, может состоять только из одного сертификата (ключ проверки электронной подписи которого соответствует ключу электронной подписи). Если сертификат ключа проверки электронной подписи является самоподписанным, то проверка подписи этого сертификата не требуется (сертификат заверен ключом электронной подписи, которому соответствует ключ проверки электронной подписи этого сертификата). Дело обстоит иначе, когда сертификат подписан на некотором корневом сертификате центра, либо на некотором промежуточном сертификате (который в свою очередь подписан на корневом или на другом промежуточном сертификате). Тогда для обеспечения проверки подписи такого сертификата при его чтении, совместно с записью сертификата ключа проверки электронной подписи на носитель в хранилище доверенных сертификатов следует класть всю цепочку сертификатов, которой заверен этот сертификат.

2.14.4.1.Совместимость с КриптоПро УЦ при проверке цепочки сертификатов

Для проверки цепочки в режиме совместимости с КриптоПро УЦ в следует воспользоваться провайдером RevCheck (JCPRevCheck.jar), входящим в поставку криптопровайдера КриптоПро JCP.

Для его вызова в примерах, описанных ниже, следует заменить вызов алгоритма "PKIX" на вызов алгоритма "CPPKIX"

```
    //для проверки цепочки online
    System.setProperty("com.sun.security.enableCRLDP", "true");//если используется SUN
JVM
    или
    System.setProperty("com.ibm.security.enableCRLDP", "true");//если используется IBM
JVM
    //для построения цепочки
    CertPathBuilder builder = CertPathBuilder.getInstance("CPPKIX");
    и
    //для проверки цепочки
    CertPathValidator validator = CertPathValidator.getInstance("CPPKIX");
```

В остальных случаях применения в JTLS (cpSSL.jar) процедура проверки цепочки сертификатов регулируется с помощью панели JCP (вкладка "Настройки сервера").

2.14.4.2.Проверка цепочки сертификатов с использованием OCSP

Начиная с версии java 1.5 появилась возможность проверять цепочку сертификатов используя On-Line Certificate Status Protocol (OCSP). Для проверки используются стандартные средства java-машины.

3. Работа с параметрами в криптопровайдере КриптоПро JCSP

Зачастую при работе с ключами электронной подписи ключевого обмен возникает необходимость изменить набор параметров того или иного криптографического алгоритма для выполнения требуемой операции. Для этих целей в криптопровайдере КриптоПро JCSP реализован интерфейс `ParamsInterface` параметров алгоритма, являющийся реализацией стандартного класса AlgorithmParameterSpec. Объект типа `ParamsInterface`, в зависимости от способа его создания, определяет

- интерфейс набора параметров ключа подписи;
- интерфейс параметров алгоритма подписи/ключевого обмена ГОСТ Р 34.10-2001;
- интерфейс параметров алгоритма подписи/ ключевого обмена ГОСТ Р 34.10-2012;
- интерфейс параметров алгоритма хэширования ГОСТ Р 34.11-94;
- интерфейс параметров алгоритма шифрования ГОСТ 28147-89.

Во всех случаях, интерфейс параметров/набора параметров `ParamsInterface` позволяет:

- получать идентификатор текущих параметров алгоритма / набора параметров ключа при помощи функции *getOID()*;
- получать идентификатор по умолчанию для параметров алгоритма / набора параметров ключа при помощи функции *getDefault(OID paramSetOid)*;
- проверять права на изменение идентификатора по умолчанию для параметров алгоритма / набора параметров ключа при помощи функции *setDefaultAvailable()*;
- устанавливать идентификатор по умолчанию для параметров алгоритма / набора параметров ключа при помощи функции *setDefault(OID paramSetOid, OID def)*;
- получать список допустимых идентификаторов для параметров алгоритма / набора параметров ключа при помощи функций *getOIDs()* или *getOIDs(OID paramSetOid)*;
- получать список строковых представлений допустимых идентификаторов для параметров алгоритма / набора параметров ключа при помощи функции *getNames()*;
- получать строковое представление одного из допустимых идентификаторов для параметров алгоритма / набора параметров ключа при помощи функции *getNameByOID(OID oid)*;
- получать один из допустимых идентификаторов для параметров алгоритма / набора параметров ключа по его строковому представлению при помощи функции *getOIDbyName(String oid)*.

3.1.Работа с набором параметров для генерации ключей электронной подписи и ключевого обмена

В процессе генерации ключевой пары подписи существует возможность изменить набор параметров, в соответствии с которым будет создана ключевая пара (как описывалось выше, по умолчанию создается пара с параметрами, установленными в контрольной панели). В этот набора параметров входят:

- идентификатор набора параметров для генерации ключевой пары;
- параметры алгоритма подписи ГОСТ Р 34.10-2001;
- параметры алгоритма подписи ГОСТ Р 34.10-2012;
- параметры алгоритма хэширования ГОСТ Р 34.11-94;

- параметры алгоритма шифрования ГОСТ 28147-89.

Изменение такого набора параметров осуществляется при помощи интерфейса AlgIdInterface, являющегося расширением интерфейса ParamsInterface. Объект типа AlgIdInterface представляет собой интерфейс устанавливаемого набора параметров. Такой объект может быть создан при помощи класса AlgIdSpec, являющегося реализацией этого интерфейса несколькими способами:

```
    // идентификатор набора параметров для ключа подписи, соответствующего алгоритму ГОСТ
Р 34.10-2001
    // "1.2.643.2.2.19" или JCP.GOST_EL_KEY_OID
    String keyOIDStr;
    OID keyOid = new OID(keyOIDStr);
    // идентификаторы параметров алгоритмов
    OID signOid;
    OID digestOid;
    OID cryptOid;
    // параметры алгоритмов
    ParamsInterface signParams;
    ParamsInterface digestParams;
    ParamsInterface cryptParams;
    /* определение набора параметров по умолчанию (установленного в контрольной панели)
*/
    AlgIdSpec keyParams1 = new AlgIdSpec(null);
    /* определение набора параметров по идентификатору алгоритма генерации ключевой пары
    (в этом случае устанавливается набор параметров по умолчанию, соответствующий этому
    идентификатору). На данный момент единственным допустимым идентификатором набора
параметров
    для генерации ключа подписи является "1.2.643.2.2.19" (или JCP.GOST_EL_KEY_OID),
    поэтому такой способ создания идентичен первому способу*/
    AlgIdSpec keyParams2 = new AlgIdSpec(keyOid);
    /* определение набора параметров по идентификатору алгоритма генерации ключевой пары
    и заданным идентификаторам параметров. Получение таких идентификаторов описано
ниже.*/
    AlgIdSpec keyParams3 = new AlgIdSpec(keyOid, signOid, digestOid, cryptOid);
    /* определение набора параметров по идентификатору алгоритма генерации ключевой пары
    и заданным параметрам алгоритмов. Получение таких параметров описано ниже.*/
    AlgIdSpec keyParams4 = new AlgIdSpec(keyOid, signParams, digestParams, cryptParams);
```

В случае использования идентификатора набора параметров для ключа подписи, соответствующего алгоритму ГОСТ Р 34.10-2012 (256), объект класса AlgIdSpec создается следующим образом:

```
    /* определение набора параметров по умолчанию (установленного в контрольной панели)
*/
    AlgIdSpec keyParams1 = new AlgIdSpec(AlgIdSpec.OID_PARAMS_SIG_2012_256);
```

В случае использования идентификатора набора параметров для ключа подписи, соответствующего алгоритму ГОСТ Р 34.10-2012 (512), объект класса AlgIdSpec создается следующим образом:

```
    /* определение набора параметров по умолчанию (установленного в контрольной панели)
*/
    AlgIdSpec keyParams1 = new AlgIdSpec(AlgIdSpec.OID_PARAMS_SIG_2012_512);
```

3.2.Работа с параметрами алгоритмов подписи/ключевого обмена ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012

Явно изменение параметров алгоритма подписи/ключевого обмена ГОСТ Р 34.10-2001 не встречается в функциях стандартного интерфейса JCE, но оно может быть использовано в процессе определения набора параметров для генерации ключевых пар подписи (см. выше). Для изменения используемых параметров алгоритма подписи/ключевого обмена, необходимо в первую очередь получить интерфейс параметров алгоритма подписи ГОСТ Р 34.10-2001 по умолчанию (установленных в контрольной панели). Данная операция может быть выполнена при помощи статического метода getDefaultSignParams() класса `AlgIdSpec`, возвращающего ссылку на интерфейс `ParamsInterface`:

```
ParamsInterface signParams = AlgIdSpec.getDefaultSignParams();
```

Получение набора параметров электронной подписи/ключевого обмена и установка параметров по умолчанию (будут использоваться в дальнейшем):

```
/* получение всех допустимых идентификаторов параметров алгоритма подписи*/
Enumeration signOids = signParams.getOIDs(paramSetOid); // paramSetOid – идентификатор набора параметров
/* получение одного из идентификаторов. Он может быть передан в соответствующий
конструктор keyParams3 для класса AlgIdSpec*/
OID signOid  = (OID)signOids.nextElement();
/* изменение идентификатора параметров по умолчанию. Измененные таким образом
параметры могут быть переданы в соответствующий
конструктор keyParams4 для класса AlgIdSpec*/
signOids.setDefault(paramSetOid, signOid);
```

3.3.Работа с параметрами алгоритма хэширования ГОСТ Р 34.11-94 и ГОСТ Р 34.11-2012

Изменение параметров алгоритма хэширования может быть использовано в явном виде в процессе создания электронной подписи, а также в неявном виде в процессе определения набора параметров для генерации ключевых пар подписи (см. выше). Для изменения используемых параметров хэширования, необходимо в первую очередь получить интерфейс параметров алгоритма хэширования ГОСТ Р 34.11-94 по умолчанию (установленных в контрольной панели). Данная операция может быть выполнена при помощи статического метода getDefaultDigestParams() класса `AlgIdSpec`, возвращающего ссылку на интерфейс `ParamsInterface`:

```
ParamsInterface digestParams = AlgIdSpec.getDefaultDigestParams();
```

Получение набора параметров хэширования и установка параметров по умолчанию (будут использоваться в дальнейшем):

```
/* получение всех допустимых идентификаторов параметров алгоритма хэширования*/
 Enumeration digestOids = digestParams.getOIDs(paramSetOid); // paramSetOid – идентификатор набора параметров
/* получение одного из идентификаторов. Он может быть передан в соответствующий
конструктор keyParams3 для класса AlgIdSpec*/
OID digestOid  = (OID)digestOids.nextElement();
 /* изменение идентификатора параметров хэширования по умолчанию. Измененные таким образом
параметры хэширования могут быть переданы в соответствующий
конструктор keyParams4 для класса AlgIdSpec. Помимо этого они могут быть использованы
для изменения параметров хэширования при создании электронной подписи.*/
digestOids.setDefault(paramSetOid, digestOid);
```

3.4.Работа с параметрами алгоритма шифрования ГОСТ 28147-89

Явно изменение параметров алгоритма шифрования ГОСТ 28147-89 не встречается в функциях стандартного интерфейса JCE, но оно может быть использовано в процессе определения набора параметров для генерации ключевых пар подписи (см. выше). Для изменения используемых параметров электронной подписи, необходимо в первую очередь получить интерфейс параметров алгоритма шифрования ГОСТ 28147-89 по умолчанию (установленных в контрольной панели). Данная операция может быть выполнена при помощи статического метода getDefaultCryptParams класса `AlgIdSpec`, возвращаюшего ссылку на интерфейс `ParamsInterface`:

```
ParamsInterface cryptParams = AlgIdSpec.getDefaultCryptParams();
```

Получение набора параметров шифрования и установка параметров по умолчанию (будут использоваться в дальнейшем):

```
/* получение всех допустимых идентификаторов параметров алгоритма шифрования*/
   Enumeration cryptOids = cryptParams.getOIDs(paramSetOid); // paramSetOid – идентификатор набора параметров
/* получение одного из идентификаторов. Он может быть передан в соответствующий
конструктор keyParams3 для класса AlgIdSpec*/
OID cryptOid  = (OID)cryptOids.nextElement();
 /* изменение идентификатора параметров шифрования по умолчанию. Измененные таким образом
параметры шифрования могут быть переданы в соответствующий
конструктор keyParams4 для класса AlgIdSpec*/
cryptOids.setDefault(paramSetOid, cryptOid);
```

4. Дополнительные возможности работы с сертификатами

Помимо возможности работы с сертификатами через стандартный интерфейс JCA, в криптопровайдере КриптоПро JCSP реализованы некоторые дополнительные функции работы с сертификатами:

- генерация запроса на сертификат;
- отправка запроса серверу и получение от сервера соответствующего запросу сертификата;
- генерация самоподписанного сертификата.

Перечисленные операции осуществляются при помощи специального класса `GostCertificateRequest`. Данный класс реализует функционал генерации запросов на сертификат открытого ключа в соответствии с алгоритмами ГОСТ Р 34.10-2001 с алгоритмом хэширования ГОСТ Р 34.11-94 и ГОСТ Р 34.10-2012 с алгоритмом хэширования ГОСТ Р 34.11-2012. Открытые ключи, для которых генерируются запросы на сертификат, также должны соответствовать алгоритмам ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012 (способ генерации таких ключей описан выше).

Вырабатываемый запрос имеет структуру, описанную в стандарте PKCS#10 «Certification Request Syntax Specification» (см. RFC 2986).

4.1.Инициализация генератора запросов и сертификатов

Когда требуется создать запрос или сертификат сначала надо воспользоваться конструктором, при вызове которого передать в качестве параметра название криптопровайдера (JCSP):

```
GostCertificateRequest request = new
GostCertificateRequest(JCSP.PROVIDER_NAME);
```

Установить способ использования ключа keyUsage можно методом `setKeyUsage()`, параметром которого передается целочисленная 32-битная перемнная int - битовая маска способов использования ключа. По умолчанию используется комбинация `DIGITAL_SIGNATURE` "цифровая подпись" и `NON_REPUDIATION` "неотрекаемость" или константа `SIGN_DEFAULT`, объединяющая два эти значения. Если Вы создаете запрос для ключа шифрования (то есть для алгоритма "GOST3410DHEL" или "GOST3410DH_2012_256") стоит добавить `KEY_ENCIPHERMENT` "шифрование ключей" и `KEY_AGREEMENT` "согласование ключей". Можно воспользоваться константой `CRYPT_DEFAULT` которая объединяет все четыре значения.

```
int keyUsage = GostCertificateRequest.DIGITAL_SIGNATURE |
        GostCertificateRequest.NON_REPUDIATION |
        GostCertificateRequest.KEY_ENCIPHERMENT |
        GostCertificateRequest.KEY_AGREEMENT;
request.setKeyUsage(keyUsage);
```

Добавить ExtendedKeyUsage "улучшенный ключ" можно методом `addExtKeyUsage()`. Параметр методу `addExtKeyUsage()` можно указывать массивом `int[]{1, 3, 6, 1, 5, 5, 7, 3, 4}` или можно строкой `"1.3.6.1.5.5.7.3.3"` или объектом типа `ru.CryptoPro.JCP.params.OID`. По умолчанию список будет пустым.

```
request.addExtKeyUsage(GostCertificateRequest.INTS_PKIX_EMAIL_PROTECTION);
request.addExtKeyUsage("1.3.6.1.5.5.7.3.2"); // "Проверка подлинности клиента"
```

Допустимые OIDы для ExtendedKeyUsage и номера битов маски keyUsage описаны в RFC 5280.

В классе `GostCertificateRequest` определены следующие константы:

```
public static final int[] INTS_PKIX_SERVER_AUTH = {1, 3, 6, 1, 5, 5, 7, 3, 1};
public static final int[] INTS_PKIX_CLIENT_AUTH = {1, 3, 6, 1, 5, 5, 7, 3, 2};
public static final int[] INTS_PKIX_CODE_SIGNING = {1, 3, 6, 1, 5, 5, 7, 3, 3};
```

```
public static final int[] INTS_PKIX_EMAIL_PROTECTION = {1, 3, 6, 1, 5, 5, 7, 3, 4};
public static final int[] INTS_PKIX_IPSEC_END_SYSTEM = {1, 3, 6, 1, 5, 5, 7, 3, 5};
public static final int[] INTS_PKIX_IPSEC_TUNNEL = {1, 3, 6, 1, 5, 5, 7, 3, 6};
public static final int[] INTS_PKIX_IPSEC_USER = {1, 3, 6, 1, 5, 5, 7, 3, 7};
public static final int[] INTS_PKIX_TIME_STAMPING = {1, 3, 6, 1, 5, 5, 7, 3, 8};
public static final int[] INTS_PKIX_OCSP_SIGNING = {1, 3, 6, 1, 5, 5, 7, 3, 9};
```

При необходимости можно в запрос добавить собственное расширение, помимо KeyUsage и ExtendedKeyUsage. Пример добавления расширения основные ограничения BasicConstraints в запрос:

```
Extension ext = new Extension();
int[] extOid = {2, 5, 29, 19};
ext.extnID = new Asn1ObjectIdentifier(extOid);
ext.critical = new Asn1Boolean(true);
byte[] extValue = {48, 6, 1, 1, -1, 2, 1, 5};
ext.extnValue = new Asn1OctetString(extValue);
request.addExtension(ext);
```

Такое расширение автоматически добавляется в сертификат при генерации самоподписанного сертификата (без обращения к центру сертификации) методами класса `GostCertificateRequest` Это расширение имеет значения "Тип субъекта = ЦС", "Ограничение на длину пути = 5" и является критическим.

Использовать метод `addExtension()` для установки в запрос KeyUsage и ExtendedKeyUsage **нельзя**, для этого надо воспользоваться методами `setKeyUsage()` и `addExtKeyUsage()`

Ранее для инициализации объектов типа `GostCertificateRequest` использовался метод init:

```
request.init("GOST3410EL"); // JCP.GOST_EL_DEGREE_NAME - для ключей подписи,
```

и

```
request.init("GOST3410DHEL", isServer); // JCP.GOST_EL_DH_NAME - для ключей обмена.
```

или

```
request.init("GOST3410_2012_256"); // JCP.GOST_EL_2012_256_NAME - для ключей подписи,
```

и

```
request.init("GOST3410DH_2012_256", isServer); // JCP.GOST_DH_2012_256_NAME - для ключей обмена.
```

или

```
request.init("GOST3410_2012_512"); // JCP.GOST_EL_2012_512_NAME - для ключей подписи,
```

и

```
request.init("GOST3410DH_2012_512", isServer); // JCP.GOST_DH_2012_512_NAME - для ключей обмена.
```

Начиная с версии 1.0.48 JCP данный метод объявлен нежелательным к использованию («deprecated»). Вызов `init("GOST3410EL")` или `init("GOST3410_2012_256")` или `init("GOST3410_2012_512")` эквивалентен вызову

```
request.setKeyUsage( GostCertificateRequest.SIGN_DEFAULT);
```

Вызов `init("GOST3410DHEL", isServer)` или `init("GOST3410DH_2012_256", isServer)` или `init("GOST3410DH_2012_512", isServer)` эквивалентен двум вызовам:

```
request.setKeyUsage( GostCertificateRequest.CRYPT_DEFAULT);
request.addExtKeyUsage(GostCertificateRequest.INTS_PKIX_CLIENT_AUTH);
request.addExtKeyUsage(GostCertificateRequest.INTS_PKIX_SERVER_AUTH); // только для сервера
```

или трем, если второй параметр метода `init()` установлен в true.

Использовавшийся ранее флаг "Подписывание сертификатов" исключен из списка по умолчанию, теперь его надо указывать явно.

4.2.Генерация запроса на сертификат

Для генерации запроса на сертификат при помощи класса `GostCertificateRequest` необходимо выполнить следующую последовательность действий:

4.2.1.Определение параметров открытого ключа субъекта

После того, как генератор был проинициализирован требуемыми алгоритмом ключа и назначением сертификата, до начала непосредственно генерации запроса, заключающейся в кодировании и подписании содержимого полей запроса, следует определить параметры и значение открытого ключа субъекта, в соответствии с которым и будет создаваться запрос на сертификат. Эта операция осуществляется при помощи метода *setPublicKeyInfo()*, которому в качестве параметра передается открытый ключ:

```
PublicKey publicKey;

request.setPublicKeyInfo(publicKey);
```

Открытый ключ `publicKey` должен соответствовать алгоритму, которым был проинициализирован генератор.

Функция *setPublicKeyInfo()* позволяет переустанавливать значение и параметры открытого ключаы, в соответствии с которым создается запрос на сертификат. Но такие изменения допустимы лишь до тех пор, пока запрос не был подписан. В противном случае этот метод инициирует исключение.

4.2.2.Определение имени субъекта

Для осуществления генерации запроса на сертификат объекту типа `GostCertificateRequest` следует передать всю необходимую информацию о субъекте-владельце открытого ключа. Определение имени субъекта осуществляется при помощи метода *setSubjectInfo()*, которому в качестве параметра передается строковое представление имени в соответствии со стандартом X.500:

```
String name = "CN=Ivanov, OU=Security, O=CryptoPro, C=RU";

request.setSubjectInfo(name);
```

При повторном вызове функции *setSubjectInfo()* осуществляется замена установленного предыдущим ее вызовом имени на новое. Таким образом, метод *setPublicKeyInfo()* позволяет переопределять имя субъекта, для которого осуществляется генерация запроса на сертификат. Но такие изменения допустимы лишь до тех пор, пока запрос не был подписан. В противном случае этот метод инициирует исключение.

4.2.3.Кодирование и подпись запроса

После того, как все необходимые данные о субъекте внесены (открытый ключ и имя), осуществляется непосредственно генерация запроса, заключающееся в кодировании переданных объекту типа `GostCertificateRequest` данных и их подписи. Эта операция осуществляется при помощи метода *encodeAndSign()*, которому в качестве параметра передается секретный ключ, используемый для подписи запроса на сертификат, а также алгоритм подписи:

```
PrivateKey privateKey;

request.encodeAndSign(privateKey, JCP.GOST_EL_SIGN_NAME); // в случае алгоритма ключа ГОСТ Р 34.10-2001
```

или

```
request.encodeAndSign(privateKey, JCP.GOST_SIGN_2012_256_NAME); // в случае алгоритма ключа ГОСТ Р 34.10-2012 (256)
```

или

```
request.encodeAndSign(privateKey, JCP.GOST_SIGN_2012_512_NAME); // в случае алгоритма ключа ГОСТ Р 34.10-2012 (512)
```

Передаваемый ключ электронной подписи/ключевого обмена `privateKey` должен соответствовать алгоритму, которым был проинициализирован генератор. Каждый создаваемый запрос может быть подписан лишь один раз. При попытке вызова этой функции повторно, будет инициировано исключение. В результате вызова функции *encodeAndSign()* запрос представляется приобретает описанный выше вид, и в памяти он хранится в DER-кодировке.

4.2.4.Печать подписанного запроса

После того, как запрос был подписан и закодирован (другими словами, сгенерирован), требуется получить его из памяти. Класс GostCertificateRequest позволяет получать запрос в трех видах:

```
PrintStream stream;                    // выходной поток, в который печатается
                                       // сформированный запрос
request.printToDER(stream);            // записывается в поток в DER-кодировке
request.printToBASE64(stream);         // записывается в поток в BASE64-кодировке
byte[] encoded = request.getEncoded(); // возвращается в виде байтового
                                       // массива в DER-кодировке
```

Таким образом, сформированный запрос может быть получен как в DER-кодировке, так и в BASE64-кодировке. Запрос может быть записан либо в поток, либо в байтовый массив.

Запись в поток удобна в тех случаях, когда запрос требуется сохранить в некоторый файл (метод *printToDER()* сохраняет запрос в DER-кодировке, а метод *printToBASE64()* - в BASE64-кодировке). Если же предполагается дальнейшее использование данного запроса (например, отправка его центру сертификации), то удобнее его получать в виде байтового массива при помощи метода *getEncoded()*.

4.3.Отправка запроса центру сертификации и получение соответствующего запросу сертификата от центра

После того, как запрос был создан, его можно отправить центру сертификации для получения соответствующего запросу сертификата. Класс GostCertificateRequest позволяет осуществить эту операцию различными способами:

4.3.1.Получение сертификата непосредственно после генерации запроса

После того, как запрос был создан, можно предварительно не сохранять его в массив или поток, а сразу после генерации отправить центру сертификации для получения запрашиваемого сертификата. Операция отправки запроса центру непосредственно после его генерации осуществляется при помощи функции *getEncodedCert()*, которая получает в качестве параметра URL центра сертификации и возвращает закодированный в DER-кодировке сертификат, соответствующий подписанному запросу, в виде байтового массива:

```
String httpAddress = "http://www.cryptopro.ru/certsrv/";
byte[] encodedCert = request.getEncodedCert(httpAddress);
```

Полученный таким образом закодированный в DER-кодировке сертификат может в дальнейшем использоваться стандартными средствами JCA (например, функциями класса CertificateFactory).

Следует заметить, что отправлен центру сертификации может быть только подписанный запрос. В противном случае метод *getEncodedCert()* инициирует исключение.

4.3.2.Получение сертификата из запроса, представленного в DER-кодировке

Сохраненный в DER-кодировке запрос может быть отправлен центру сертификации для получения запрашиваемого сертификата при помощи статического метода *getEncodedCertFromDER()* двумя способами:

```
String httpAddress = "http://www.cryptopro.ru/certsrv/";
InputStrean stream;    // входной поток, в который записан
                       // запрос в DER-кодировке
byte[] encoded;        // DER-закодированный запрос
byte[] encodedCert =
    GostCertificateRequest.getEncodedCertFromDER(httpAddress, stream);
byte[] encodedCert =
    GostCertificateRequest.getEncodedCertFromDER(httpAddress, encoded);
```

Оба вызова метода *getEncodedCertFromDER()* получают в качестве одного из параметров URL центра сертификации и возвращают закодированный в DER-кодировке сертификат, соответствующий подписанному запросу, в виде байтового массива.

Полученный таким образом закодированный в DER-кодировке сертификат может в дальнейшем использоваться стандартными средствами JCA (например, функциями класса CertificateFactory).

Разница заключается в том, что первому способу вызова функции *getEncodedCertFromDER()* в качестве параметра передается входной поток, в который записан закодированный в DER-кодировке запрос. Такой поток обычно направлен на файл, содержащий запрос. Запись же запроса в файл может быть осуществлена при помощи метода *printToDER()* класса `GostCertificateRequest` (подробнее см. сохранение запроса). Второму же способу вызова функции *getEncodedCertFromDER()* в качестве параметра передается байтовый массив, содержащий в себе DER-закодированный запрос. Такой массив может быть получен при помощи метода *getEncoded()* класса `GostCertificateRequest` (подробнее см. сохранение запроса).

4.3.3.Получение сертификата из запроса, представленного в BASE64-кодировке

Сохраненный в BASE64-кодировке запрос может быть отправлен центру сертификации для получения запрашиваемого сертификата при помощи статического метода *getEncodedCertFromDER()* следующим образом:

```
String httpAddress = "http://www.cryptopro.ru/certsrv/";
InputStrean stream;     // входной поток, в который записан
                        // запрос в BASE64-кодировке
byte[] encodedCert =
    GostCertificateRequest.getEncodedCertFromBASE64(httpAddress, stream);
```

Метод *getEncodedCertFromBASE64()* получает в качестве параметров URL центра сертификации и входной поток, в который записан закодированный в BASE64-кодировке запрос. Такой поток обычно направлен в файл, содержащий запрос. Запись же запроса в файл может быть осуществлена при помощи метода *printToBASE64()* класса `GostCertificateRequest` (подробнее см. сохранение запроса). Метод *getEncodedCertFromBASE64()* возвращает закодированный в DER-кодировке сертификат, соответствующий подписанному запросу, в виде байтового массива.

Полученный таким образом закодированный в DER-кодировке сертификат может в дальнейшем использоваться стандартными средствами JCA (например, функциями класса CertificateFactory).

4.3.4.Получение корневого сертификата центра сертификации

После того, как соответствующий запросу сертификат был получен от центра, зачастую требуется выполнить построение цепочки сертификатов, начинающейся с корневого сертификата центра, и заканчивающейся полученным от этого центра сертификатом. Класс `GostCertificateRequest` позволяет получать корневой сертификат центра сертификации при помощи статического метода *getEncodedRootCert()* следующим образом:

```
String httpAddress = "http://www.cryptopro.ru/certsrv/";
byte[] encodedRootCert =
    GostCertificateRequest.getEncodedRootCert(httpAddress);
```

Функция *getEncodedRootCert()* получает в качестве параметра URL центра сертификации и возвращает закодированный в DER-кодировке корневой сертификат центра в виде байтового массива.

Полученный таким образом закодированный в DER-кодировке корневой сертификат `encodedRootCert` может в дальнейшем быть обработан функциями класса CertificateFactory, и после может использоваться, например, для построения цепочек. Обработанный такой сертификат может быть добавлен в хранилище доверенных сертификатов.

4.4.Генерация самоподписанного сертификата

Для генерации самоподписанных сертификатов можно воспользоваться методом *getEncodedSelfCert()* класса `GostCertificateRequest`. Этот метод получает в качестве параметра ключевую пару субъекта (он же издатель сертификата), а также имя субъекта (оно же имя издателя). Передаваемая ключевая пара должна соответствовать алгоритму, которым был проинициализирован генератор. Сертификат возвращается в DER-кодировке в виде байтового массива.

После того, как объект класса GostCertificateRequest проинициализирован, осуществляется собственно генерация сертификата:

```
KeyPair pair;    // ключевая пара субъекта (она же пара издателя)
String name;     // имя субъекта (оно же имя издателя)
String signAlgorithm; // алгоритм подписи
byte[] encodedCert =
    request.getEncodedSelfCert(pair, name, signAlgorithm);
```

Полученный таким образом закодированный в DER-кодировке самоподписанный сертификат encodedCert может в дальнейшем быть обработан функциями класса CertificateFactory, и после может использоваться, например, для записи ключ на носитель.

При генерации самоподписанного сертификата (без обращения к центру сертификации) методами класса GostCertificateRequest ему устанавливаются те же расширения, что и при генерации запроса, а также расширение basicConstraints - основные ограничения. Это расширение имеет значения "Тип субъекта = ЦС", "Ограничение на длину пути = 5" и является критическим.

Необходимо помнить, что генерация самоподписанных сертификатов имеет смысл только для тестовых целей. Для реальной работы следует пользоваться генерацией запросов для отправки их центрам сертификации.Дополнительные возможности работы с сертификатами для УЦ 1.5

5. Дополнительные возможности работы с сертификатами для УЦ 1.5

В криптопровайдере КриптоПро JCP для взаимодействия с УЦ 1.5 реализованы следующие функции работы с сертификатами:

- получение набора параметров для регистрации пользователя;
- регистрация пользователя и получение токена и пароля;
- проверка статуса регистрации пользователя;
- получение списка корневых сертификатов УЦ;
- получение списка запросов на сертификаты пользователя;
- генерация запроса на сертификат;
- отправка запроса серверу;
- проверка статуса сертификата;
- получение от сервера соответствующего запросу сертификата.

Перечисленные операции осуществляются при помощи специального класса `CA15GostCertificateRequest`, потомка класса `GostCertificateRequest`. Все условия формирования и структура описаны в соответствующем разделе выше.

Особенностью функционала является необходимость использовать протокол HTTPS с применением JTLS (модуль `cpSSL.jar`, см. «Инструкция по использованию (JTLS)»). В этом случае необходимо настроить JTLS и указать в коде хранилище доверенных сертификатов с корневым сертификатом сервера:

```
System.setProperty("javax.net.ssl.trustStoreType", JCP.HD_STORE_NAME);
System.setProperty("javax.net.ssl.trustStore", "путь_к_файлу_хранилища");
System.setProperty("javax.net.ssl.trustStorePassword", "пароль_к_хранилищу");
```

5.1.Получение набора параметров для регистрации пользователя

Для получения набора параметров (или полей) для регистрации пользователя в УЦ 1.5 следует вызвать статическую функцию `getUserRegistrationFields` класса `CA15User` и передать ей адрес УЦ, например:

```
Vector<CA15UserRegistrationField> userRegistrationFields =
CA15User.getUserRegistrationFields(“https://www.cryptopro.ru:5555/ui”);
```

Здесь `CA15UserRegistrationField` – класс, описывающий поле для заполнения перед регистрацией пользователя. Список полей может быть достаточно большим и отличаться в разных УЦ. Его – класса – подробное описание есть в Javadoc-документации пакета `JCPRequest`. Данный класс содержит набор функций, определяющих необходимость заполнения поля (`mandatory`), читаемое имя поля (`name`), зарегистрированное имя (`formName`), максимальный размер значения (`maxLength`), значение по умолчанию (`value`), тип поля (`componentType: edit, textarea, select, separator`) и список допустимых значений для `componentType:select` (`allowedValues`). При регистрации пользователя в качестве имени поля следует использовать `formName`.

Пример использования этого класса и функции getUserRegistrationFields есть в пакете userSamples.ca15 модуля samples.jar и называется RegisterUserExample.

5.2.Регистрация пользователя, получение токена и пароля и проверка статуса

Для регистрации пользователя и получения токена (идентификатора) и пароля следует использовать следующий код:

```
Map<String, String> fields = new HashMap<String, String>(); // список пар ключ=значение, заполняется с помощью formName=Value
fields.put(“RDN_CN_1”, “test”); // RDN_CN_1 был получен из ранее загруженнного списка полей для заполнения
fields.put(“RDN_C_1”, “RU”); // RDN_C_1 был получен из ранее загруженнного списка полей для заполнения

CA15User newUser = new CA15User(fields);
CA15UserRegisterInfoStatus userStatus =
newUser.registerUser(“https://www.cryptopro.ru:5555/ui”); // регистрация
```

Список полей, которое нужно передать в УЦ для регистрации пользователя, заполняется парами «имя поля»=«значение»; имена полей могут быть получены заранее с помощью функции getUserRegistrationFields. В примере заполняются только два поля, хотя на самом деле может понадобиться заполнить больше полей (в зависимости от свойства mandatory поля). Список полей передается в класс CA15User с последующим вызовом функции registerUser.

Класс CA15UserRegisterInfoStatus показывает результат регистрации – статус CR_DISP_ERROR в случае ошибки, CR_DISP_ISSUED – если операция завершена успешно, CR_DISP_UNDER_SUBMISSION – если операция еще выполняется (в этом случае необходимо периодически проверять состояние с помощью функции checkUserStatus класса CA15User). В случае задержки регистрации или ее успешного завершения объект класса CA15UserRegisterInfoStatus будет содержать токен и пароль пользователя:

```
if (userStatus.getValue() == CA15Status.CR_DISP_UNDER_SUBMISSION) {
    Thread.sleep(30 * 1000); // ждем 30 секунд
    CA15User userInfo = new CA15User(userStatus.getTokenID(), userStatus.getPassword());
 CA15UserRegisterStatus status = userInfo.checkUserStatus(
    “https://www.cryptopro.ru:5555/ui”); // проверяем статус регистрации
}
```

Подробное описание классов CA15User и CA15UserRegisterInfoStatus есть в Javadoc-документации пакета JCPRequest.

Пример использования этих классов и функций есть в пакете userSamples.ca15 модуля samples.jar и называется RegisterUserExample.

5.3.Получение списка корневых сертификатов УЦ

Для получения списка корневых сертификатов УЦ следует вызвать следующую статическую функцию getRootCertList класса CA15GostCertificateRequest:

```
Certificate[] rootCerts = CA15GostCertificateRequest
```

```
.getRootCertList("http://www.cryptopro.ru/ui");
```

Будет получен список сертификатов, в данном случае – по протоколу HTTP.

Пример использования этой функции есть в пакете userSamples.ca15 модуля samples.jar и называется GetRootCertificateExample.

5.4.Получение списка запросов на сертификаты пользователя

Для получения списка запросов на сертификаты зарегистрированного ранее пользователя следует использовать статическую функцию getCertificateRequestList класса CA15GostCertificateRequest и класс CA15User:

```
CA15User userInfo = new CA15User(“token”, “password”); // зарегистрированный пользователь
Map<String, CA15CertificateRequestRecord> requestMap =
CA15GostCertificateRequest.getCertificateRequestList(
“https://www.cryptopro.ru:5555/ui”, userInfo); // список пар «идентификатор_запроса»=«описание_запрос»
```

В requestMap будут помещены пары «идентификатор_запроса» = «описание_запроса». Подробное описание класса есть в Javadoc-документации пакета JCPRequest. Класс CA15CertificateRequestRecord содержит описание запроса пользователя, частности: идентификатор запроса (requestIdentifier), дату отправки запроса (sentDate), дату обработки (approvalDate), комментарий (comment), статус обработки запроса (status) и сам запрос в формате PKCS10 (pkcs10).

Пример использования этих классов и функций есть в пакете userSamples.ca15 модуля samples.jar и называется GetUserCertificateRequestListExample.

5.5.Генерация запроса на сертификат, проверка статуса сертификата и получение соответствующего запросу сертификата

Для выполнения генерации запроса на сертификат для зарегистрированного польлзователя можно следовать разделу 4.2, но использовать класс CA15GostCertificateRequest, например:

```
CA15User userInfo = new CA15User(“token”, “password”); // зарегистрированный пользователь

KeyPairGenerator kg = KeyPairGenerator.getInstance(JCP.GOST_EL_DH_NAME); // алгоритм ключа
KeyPair pair = kg.generateKeyPair(); // генерация

CA15GostCertificateRequest req = new
CA15GostCertificateRequest(JCP.PROVIDER_NAME);

req.init(JCP.GOST_EL_DH_NAME, false);
req.setPublicKeyInfo(pair.getPublic());
req.setSubjectInfo(“CN=test,C=RU”); // список полей запроса (subject name) должен совпадать со списком, переданным ранее для регистрации пользователя
req.encodeAndSign(pair.getPrivate(), JCP.GOST_EL_SIGN_NAME); // подпись запроса
```

```
CA15RequestStatus requestStatus =
req.sendCertificateRequest(“https://www.cryptopro.ru:5555/ui”, userInfo); // отправка запроса
```

С помощью класса CA15User задается зарегистрированный пользователь, генерируется ключевая пара на алгоритме ГОСТ Р 34.10-2001 DH. Затем формируется запрос с информацией о владельце (subject), полностью соответствующей списку полей, переданному при регистрации данного пользователя. С помощью функции sendCertificateRequest класса CA15GostCertificateRequest запрос передается в УЦ. Информация со статусом обработки операции помещается в объект класс CA15RequestStatus.

Подробное описание класса CA15RequestStatus есть в Javadoc-документации пакета JCPRequest. Он позволяет узнать идентификатор запроса и статус обработки: CR_DISP_ERROR в случае ошибки, CR_DISP_ISSUED в случае успешной обработки, CR_DISP_UNDER_SUBMISSION в случае продолжающейся обработки, CR_DISP_DENIED в случае отказа в обработке.

Чтобы узнать статус обработки и установить факт выпуска сертификата, следует выполнить проверку:

```
CA15RequestStatus certStatus = CA15GostCertificateRequest.checkCertificateStatus(
	“https://www.cryptopro.ru:5555/ui”, userInfo,
	requestStatus.getRequestIdentifier()); // используем идентификатор запроса для проверки, выпущен ли сертификат
```

Объект certStatus также может вернуть один из статусов, перечисленных выше. Если был получен CR_DISP_ISSUED, то можно загрузить сертификат в DER-кодировке с помощью статической функции getCertificateByRequestId класса CA15GostCertificateRequest:

```
byte[] certificateEncoded =
CA15GostCertificateRequest.getCertificateByRequestId(
	“https://www.cryptopro.ru:5555/ui”, userInfo,
	requestStatus.getRequestIdentifier()); // используем идентификатор запроса
```

Если в certStatus был получен статус CR_DISP_UNDER_SUBMISSION, то можно выполнить проверку статуса с помощью checkCertificateStatus позже и повторно обратиться к getCertificateByRequestId.

Преобразовать полученный массив байтов, содержащий сертификат, можно так:

```
X509Certificate certificate =
	(X509Certificate) CertificateFactory.getInstance("X.509")
	.generateCertificate(new ByteArrayInputStream(certificateEncoded));
```

Пример использования этих классов и функций есть в пакете userSamples.ca15 модуля samples.jar и называется SendRequestAndGetCertificateExample.

6. Дополнительные возможности работы с сертификатами для УЦ 2.0

В криптопровайдере КриптоПро JCP для взаимодействия с УЦ 2.0 реализованы следующие функции работы с сертификатами:

- получение набора параметров для регистрации пользователя в УЦ 2.0;
- регистрация пользователя и получение токена и пароля;
- проверка статуса регистрации пользователя;
- получение списка корневых сертификатов УЦ 2.0;
- получение списка запросов на сертификаты пользователя;
- подтверждение факта установки сертификата пользователя;
- авторизация пользователя по токену и паролю или сертификату;
- получение списка запросов на отзыв сертификатов;
- получение списка шаблонов сертификатов УЦ 2.0;
- генерация запроса на сертификат;
- отправка запроса серверу;
- проверка статуса сертификата;
- получение от сервера соответствующего запросу сертификата.

Перечисленные операции осуществляются при помощи специального класса `CA20GostCertificateRequest`, потомка класса `GostCertificateRequest`. Все условия формирования и структура описаны в соответствующем разделе выше. Большинство методов классов `CA20GostCertificateRequest` и `CA20User` асинхронные.

В API для УЦ 2.0 пользователь УЦ обладает еще одним дополнительным параметром — папка пользователя, в которой он будет зарегистрирован.

Особенностью функционала является необходимость использовать протокол HTTPS с применением JTLS (модуль `cpSSL.jar`, см. «Инструкция по использованию (JTLS)»). В этом случае необходимо настроить JTLS и указать в коде хранилище доверенных сертификатов с корневым сертификатом сервера:

```
System.setProperty("javax.net.ssl.trustStoreType", JCP.CERT_STORE_NAME);
System.setProperty("javax.net.ssl.trustStore", "путь_к_файлу_хранилища");
System.setProperty("javax.net.ssl.trustStorePassword", "пароль_к_хранилищу");
```

В ситуации, когда требуется авторизация по сертификату пользователя, может потребовать указание типа контейнера пользователя и пароля к нему:

```
System.setProperty("javax.net.ssl.keyStoreType", JCP.HD_STORE_NAME);
System.setProperty("javax.net.ssl.keyStorePassword", "пароль_к_контейнеру");
```

6.1.Получение набора параметров для регистрации пользователя в УЦ 2.0

Для получения набора параметров (или полей) для регистрации пользователя в УЦ 2.0 следует вызвать статическую функцию getUserRegistrationFields класса CA20User и передать ей адрес УЦ, например:

```
Vector<CA20UserRegistrationField> userRegistrationFields =
CA20User.getUserRegistrationFields(“https://www.cryptopro.ru/ui”, «папка_пользователя»);
```

Здесь «папка_пользователя» - папка в которой предполагается зарегистрировать пользователя. CA20UserRegistrationField – класс, описывающий поле для заполнения перед регистрацией пользователя. Список полей может быть достаточно большим. Его – класса – подробное описание есть в Javadoc-документации пакета JCPRequest. Данный класс содержит набор функций, определяющих OID элемента учетной записи пользователя (oid), имя элемента (name), локализованное имя (localizedName), список возможных значений элемента (settingsValues), значение по умолчанию (defaultValue) и еще несколько флагов. При регистрации пользователя в качестве OID'а поля следует использовать OID элемента.

Пример использования этого класса и функции getUserRegistrationFields есть в пакете userSamples.ca20 модуля samples.jar и называется CA20StepByStepExample.

6.2.Регистрация пользователя, получение токена и пароля и проверка статуса

Для регистрации пользователя и получения токена и пароля следует использовать следующий код:

```
Map<String, String> fields = new HashMap<String, String>(); // список пар ключ=значение, заполняется с помощью OID=Value
fields.put(“2.5.4.3”, “test”); // 2.5.4.3 был получен из ранее загруженнного списка полей для заполнения
fields.put(“2.5.4.6”, “RU”); // 2.5.4.6 был получен из ранее загруженнного списка полей для заполнения
CA20User newUser = new CA20User(fields, “папка_пользователя”);
CA20AuxiliaryUserInfo userInfo = new CA20AuxiliaryUserInfo("comment", "description", "test@cryptopro.ru", "key phrase"); // дополнительная информация о пользователе
CA20UserRegisterInfoStatus userStatus =
newUser.registerUser(“https://www.cryptopro.ru/ui”); // регистрация
```

Список полей, которое нужно передать в УЦ для регистрации пользователя, заполняется парами «oid»=«значение»; OID'ы полей могут быть получены заранее с помощью функции getUserRegistrationFields. В примере заполняются только два поля, хотя количество полей может быть иным.

Далее заполняется создается объект класса CA20AuxiliaryUserInfo с дополнительной информацией о пользователе. Список полей передается в класс CA20User с последующим вызовом функции registerUser.

Класс CA20UserRegisterInfoStatus показывает результат регистрации – статус E в случае ошибки, C – если операция завершена успешно, A – если запрос принят, Q - если операция еще выполняется (необходимо периодически проверять состояние с помощью функции checkUserStatus класса CA20User). В случае задержки регистрации или ее успешного завершения

объект класса CA20UserRegisterInfoStatus будет содержать токен и пароль пользователя и идентификатор запроса регистрации пользователя:

```
if (!userStatus..getStatus().equalsIgnoreCase(CA20Status.STATUS_REQUEST_C)) {
        Thread.sleep(30 * 1000); // ждем 30 секунд
        CA20User userInfo = new CA20User(userStatus.getTokenID(), userStatus.getPassword(),
“папка_пользователя”);
    CA20Status status = userInfo.checkUserStatus(
        “https://www.cryptopro.ru/ui”); // проверяем статус регистрации
}
```

Класс CA20Status — базовый класс с описанием всех основных статусов, возвращаемых всеми методами классов пакета ca20.

Подробное описание классов CA20User, CA20Status и CA20UserRegisterInfoStatus есть в Javadoc-документации пакета JCPRequest.

Пример использования этих классов и функций есть в пакете userSamples.ca20 модуля samples.jar и называется CA20StepByStepExample.

6.3.Получение списка корневых сертификатов УЦ 2.0

Для получения списка корневых сертификатов УЦ следует вызвать следующую статическую функцию getRootCertList класса CA20GostCertificateRequest:

```
Certificate[] rootCerts = CA20GostCertificateRequest
.getRootCertList("https://www.cryptopro.ru/ui");
```

Будет получен список корневых сертификатов УЦ, в данном случае – по протоколу HTTPS.

Пример использования этой функции есть в пакете userSamples.ca20 модуля samples.jar и называется GetCA20RootCertificateExample.

6.4.Получение списка запросов на сертификаты пользователя

Для получения списка запросов на сертификаты зарегистрированного ранее пользователя следует использовать статическую функцию getCertificateRequestList класса CA20GostCertificateRequest и класс CA20User:

```
CA20User userInfo = new CA20User(“token”, “password”, “папка_пользователя”);
// зарегистрированный пользователь
Vector<CA20CertificateRequestRecord> requests =
CA20GostCertificateRequest.getCertificateRequestList(
  “https://www.cryptopro.ru/ui”, userInfo); // список пар запросов
```

В requests будут помещены запросы на сертификаты. Подробное описание класса есть в Javadoc-документации пакета JCPRequest. Класс CA20CertificateRequestRecord содержит описание запроса пользователя, частности: идентификатор запроса (certRequestId), идентификатор пользователя (userId) и список других полей.

При передаче в функцию getCertificateRequestList объекта пользователя с указанием токена и пароля авторизация будет происходить по токену и паролю. Однако если пользователь отправил подтверждение установки сертификата на сервер, после того, как он его — сертификат - получил, то потребуется авторизация по сертификату пользователя. Ее можно выполнить несколькими способами, описанными в следующем разделе.

Пример использования этих классов и функций есть в пакете userSamples.ca20 модуля samples.jar и называется CA20StepByStepExample.

6.5.Подтверждение факта установки сертификата пользователя и авторизация по токену и паролю или сертификату пользователя

После того, как пользователь получил сертификат (см. разделы далее), рекомендуется отправить подтверждение о том, что данный сертификат установлен в ключевой контейнер:

```
CA20User userInfo = new CA20User(“token”, “password”, “папка_пользователя”); // зарегистрированный пользователь
CA20RequestStatus status = CA20GostCertificateRequest.
markCertificateInstalled(“https://www.cryptopro.ru/ui”, userInfo, «идентификатор_запроса_на_сертификат»);
```

Пользователь имеет на момент отправки уведомления токен и пароль и авторизуется с их помощью. После отработки запроса в поле status будет содержаться информация об обработанном запросе и статус К. При последующих обращениях к УЦ потребуется авторизация по сертификату пользователя.

Если уведомление об установке сертификата не было отправлено, то можно продолжать авторизоваться по токену и паролю.

```
KeyStore trustStore = KeyStore.getInstance(JCP.CERT_STORE_NAME);
trustStore.load(new FileInputStream(«путь_к_хранилищу_доверенных_сертификатов»), «пароль_к_хранилищу»);
KeyStore keyStore = KeyStore.getInstance(JCP.HD_STORE_NAME);
keyStore.load(null, null);
CA20CertAuthUser userInfo =  new CA20CertAuthUser(keyStore, «пароль_к_контейнеру_пользователя», trustStore, «папка_пользователя»); // пользователь УЦ, авторизующийся по сертификату
```

Теперь, при передаче userInfo, например, в функцию получения списка запросов на сертификаты (см. предыдущий пункт), авторизация будет выполняться по сертификату пользователя, а не токену и паролю.

Другой вариант авторизации по сертификату — это использование System.setProperty, например, так:

```
System.setProperty("javax.net.ssl.trustStoreType", JCP.CERT_STORE_NAME);
System.setProperty("javax.net.ssl.trustStore", "путь_к_файлу_хранилища");
System.setProperty("javax.net.ssl.trustStorePassword", "пароль_к_хранилищу");
System.setProperty("javax.net.ssl.keyStoreType", JCP.HD_STORE_NAME);
System.setProperty("javax.net.ssl.keyStorePassword", "пароль_к_контейнеру_пользователя");
```

```
CA20CertAuthUser userInfo = new CA20CertAuthUser(«папка_пользователя»); // пользователь УЦ, авторизующийся по сертификату, и последующее использование userInfo
```

Пример использования этих классов и функций есть в пакете userSamples.ca20 модуля samples.jar и называется CA20StepByStepExample.

6.6.Генерация запроса на сертификат, проверка статуса сертификата и получение соответствующего запросу сертификата

Для выполнения генерации запроса на сертификат для зарегистрированного пользователя можно следовать разделу 4.2, но использовать класс CA20GostCertificateRequest, например:

```
CA20User userInfo = new CA20User(“token”, “password”, «папка_пользователя»); // зарегистрированный пользователь

KeyPairGenerator kg = KeyPairGenerator.getInstance(JCP.GOST_EL_DH_NAME); // алгоритм ключа
KeyPair pair = kg.generateKeyPair(); // генерация

CA15GostCertificateRequest req = new
CA15GostCertificateRequest(JCP.PROVIDER_NAME);

req.init(JCP.GOST_EL_DH_NAME, false);
req.setPublicKeyInfo(pair.getPublic());
req.setSubjectInfo(“2.5.4.3=test,2.5.4.6=RU”); // список полей запроса (subject name) должен совпадать со списком, переданным ранее для регистрации пользователя
// Добавление OID'а шаблона сертификата
final String szOID_CERTIFICATE_TEMPLATE = "1.3.6.1.4.1.311.21.7"; // OID расширения в запросе
OID OID_CERTIFICATE_TEMPLATE = new OID(szOID_CERTIFICATE_TEMPLATE);
OID  selectedTemplateOid = new OID(template.getOid());

// Формат: шаблон, 1, 0.
CertificateTemplate certificateTemplate = new CertificateTemplate(
      new Asn1ObjectIdentifier(selectedTemplateOid.value),
new Asn1Integer(1), new Asn1Integer(0));

Asn1DerEncodeBuffer buffer = new Asn1DerEncodeBuffer();
certificateTemplate.encode(buffer);

byte[] encodedCertificateTemplate = buffer.getMsgCopy();
Asn1OctetString certificateTemplateValue = new
Asn1OctetString(encodedCertificateTemplate);

Extension templateExtension = new Extension(new Asn1ObjectIdentifier(
OID_CERTIFICATE_TEMPLATE.value), certificateTemplateValue);
```

```
req.addExtension(templateExtension);

req.encodeAndSign(pair.getPrivate(), JCP.GOST_EL_SIGN_NAME); // подпись запроса

CA20RequestStatus requestStatus =
req.sendCertificateRequest(“https://www.cryptopro.ru/ui”, userInfo); // отправка
запроса
```

С помощью класса CA20User задается зарегистрированный пользователь, генерируется ключевая пара на алгоритме ГОСТ Р 34.10-2001 DH. Затем формируется запрос с информацией о владельце (subject), полностью соответствующей списку полей, переданному при регистрации данного пользователя. В тело запроса добавляется некритическое рсширение 1.3.6.1.4.1.311.21.7", содержащее информацию об используемом шаблоне (о том, как получить шаблоны, описано в следующем разделе). С помощью функции sendCertificateRequest класса CA20GostCertificateRequest запрос передается в УЦ. Информация со статусом обработки операции помещается в объект класс CA20RequestStatus.

Подробное описание класса CA20RequestStatus есть в Javadoc-документации пакета JCPRequest. Он позволяет узнать идентификатор запроса и статус обработки: E в случае ошибки, C в случае успешной обработки, A – если запрос принят, Q в случае продолжающейся обработки, D в случае отказа в обработке.

Чтобы узнать статус обработки и установить факт выпуска сертификата, следует выполнить проверку:

```
CA20RequestStatus certStatus = CA20GostCertificateRequest.checkCertificateStatus(
      “https://www.cryptopro.ru/ui”, userInfo,
      requestStatus.getCertRequestId()); // используем идентификатор запроса для
проверки, выпущен ли сертификат
```

Объект certStatus также может вернуть один из статусов, перечисленных выше. Если был получен C, то можно загрузить сертификат в DER-кодировке с помощью статической функции getCertificateByRequestId класса CA20GostCertificateRequest:

```
byte[] certificateEncoded =
CA20GostCertificateRequest.getCertificateByRequestId(
      “https://www.cryptopro.ru/ui”, userInfo,
       requestStatus.getCertRequestId ()); // используем идентификатор запроса
```

Если в certStatus был получен статус A или Q, то можно выполнить проверку статуса с помощью checkCertificateStatus позже и повторно обратиться к getCertificateByRequestId.

Преобразовать полученный массив байтов, содержащий сертификат, можно так:

```
X509Certificate certificate =
      (X509Certificate) CertificateFactory.getInstance("X.509")
       .generateCertificate(new ByteArrayInputStream(certificateEncoded));
```

Пример использования этих классов и функций есть в пакете userSamples.ca20 модуля samples.jar и называется CA20StepByStepExample.

6.7.Получение списка шаблонов сертификатов УЦ 2.0

С помощью функции getUserCertificateTemplates класса CA20User можно получить список шаблонов сертификатов папки, в которой зарегистрирован пользователь:

```
CA20User userInfo = new CA20User(“token”, “password”, «папка_пользователя»); // зарегистрированный пользователь

Vector<CA20GostTemplateField> templates = userInfo.getUserCertificateTemplates(“https://www.cryptopro.ru/ui”);
```

Список templates будет содержать перечисление шаблонов в виде объектов класса CA20GostTemplateField и позволит установить поддерживаемый тип ключей (keySpec), имя шаблона (name), локализованное имя шаблона (localizedName), OID шаблона (oid) и флаг права автоматически выпустить сертификат (authApproval). Подробное описание класса CA20GostTemplateField есть в Javadoc-документации пакета JCPRequest.

Пример использования этих классов и функций есть в пакете userSamples.ca20 модуля samples.jar и называется CA20StepByStepExample.

6.8.Получение списка запросов на отзыв сертификатов

С помощью статической функции getRequestRevocationList класса CA20GostCertificateRequest можно получить список запросов отзыва сертификатов пользователя:

```
CA20User userInfo = new CA20User(“token”, “password”, «папка_пользователя»); // зарегистрированный пользователь

Vector<CA20RevocationRecord> revocations = CA20GostCertificateRequest.getRequestRevocationList(“https://www.cryptopro.ru/ui”, userInfo);
```

Список revocations будет содержать перечисление шаблонов в виде объектов класса CA20RevocationRecord и позволит узнать идентификатор запроса отзыва (revRequestId) и другие параметры (идентификатор запроса на сертификат, идентификатор пользователя и т.д.). Подробное описание класса CA20GostTemplateField есть в Javadoc-документации пакета JCPRequest.

Пример использования этих классов и функций есть в пакете userSamples.ca20 модуля samples.jar и называется CA20StepByStepExample.

7. Работа с электронной подписью для XML-документов

Криптопровайдер КриптоПро JCSP обеспечивает формирование и проверку электронной подписи в соответствии с алгоритмом ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012 для отдельного объекта XML-документа, для всего XML-документа, а также для двух независимых подписей всего XML-документа.

Криптопровайдер КриптоПро JCSP позволяет осуществлять формирование и проверку электронной подписи XML-документа в соответствии с алгоритмом ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012. При этом криптопровайдер КриптоПро JCP использует четыре библиотеки, обеспечивающие работу с электронной подписью XML-документов:

commons-logging.jar

serializer.jar

xalan.jar

xmlsec.jar

Для корректной работы криптопровайдера все эти библиотеки должны быть скачаны с сайта http://www.apache.org. Рекомендуется воспользоваться ссылкой http://xml.apache.org/mirrors.cgi, выбирая последнюю версию продукта.

Основные операции осуществляются при помощи функций следующих классов: org.apache.xml.security.algorithms, org.apache.xml.security.exceptions, org.apache.xml.security.keys, org.apache.xml.security.signature, org.apache.xml.security.transforms, org.apache.xml.security.utils. Поскольку криптопровайдер КриптоПро JCSP не реализует методы перечисленных выше пакетов, а лишь обеспечивает их поддержку для алгоритма подписи ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012, то в данной документации подробное описание этих методов не приводится.

Перед началом использования классов из библиотеки XML Security необходимо зарегистрировать ГОСТ алгоритмы. Сделать это можно двумя способами:

Во-первых, вызовом метода `ru.CryptoPro.JCPxml.XmlInit.init()` (старый метод `ru.CryptoPro.JCPxml.xmldsig.JCPXMLDSigInit.init()` тоже поддерживается).

Во-вторых, вызовом стандартного инициализатора `org.apache.xml.security.Init.init()`, который обязателен при работе с библиотекий XML Security, но с предварительно установленным свойством `System.setProperty("org.apache.xml.security.resource.config", "resource/jcp.xml")` или указывая это свойство при запуске Java-машины следующим образом: `java -Dorg.apache.xml.security.resource.config=resource/jcp.xml`. Таким образом, регистрация ГОСТ алгоритмов не требует перекомпиляции приложения. Соответствующие константы определены в файле ru.CryptoPro.JCPxml.Consts

```
/**
 * имя Property настройки конфигурации.
 */
public static final String PROPERTY_NAME = "org.apache.xml.security.resource.config";
/**
 * Имя ресурса конфигурации.
 */
public static final String CONFIG = "resource/jcp.xml";
/**
 * алгоритм подписи (ГОСТ Р 34.10-2001)
 */
public static final String URI_GOST_SIGN = "http://www.w3.org/2001/04/xmldsig-
more#gostr34102001-gostr3411";
/**
 * алгоритм хэширования, используемый при подписи (ГОСТ Р 34.11-94)
```

```
 */
public static final String URI_GOST_DIGEST = "http://www.w3.org/2001/04/xmldsig-
more#gostr3411";
/**
 * алгоритм подписи (ГОСТ Р 34.10-2001) по новому стандарту
 */
public static final String URN_GOST_SIGN =
"urn:ietf:params:xml:ns:cpxmlsec:algorithms:gostr34102001-gostr3411";
/**
 * алгоритм хэширования, ГОСТ Р 34.11-94  по новому стандарту
 */
public static final String URN_GOST_DIGEST =
"urn:ietf:params:xml:ns:cpxmlsec:algorithms:gostr3411";
/**
 * URN алгоритма подписи по новому стандарту
 * http://tools.ietf.org/html/draft-chudov-cryptopro-cpxmldsig-07
 */
public static final String URN_GOST_HMAC_GOSTR3411 =
"urn:ietf:params:xml:ns:cpxmlsec:algorithms:hmac-gostr3411";
/**
 * алгоритм подписи (ГОСТ Р 34.10-2012? 256)
 */
public static final String URN_GOST_SIGN_2012_256 =
"urn:ietf:params:xml:ns:cpxmlsec:algorithms:gostr34102012-gostr34112012-256";
/**
 * алгоритм хэширования, ГОСТ Р 34.11-2012 (256)
 */
public static final String URN_GOST_DIGEST_2012_256 =
"urn:ietf:params:xml:ns:cpxmlsec:algorithms:gostr34112012-256";
/**
 * алгоритм подписи (ГОСТ Р 34.10-2012? 512)
 */
public static final String URN_GOST_SIGN_2012_512 =
"urn:ietf:params:xml:ns:cpxmlsec:algorithms:gostr34102012-gostr34112012-512";
/**
 * алгоритм хэширования, ГОСТ Р 34.11-2012 (512)
 */
public static final String URN_GOST_DIGEST_2012_512 =
"urn:ietf:params:xml:ns:cpxmlsec:algorithms:gostr34112012-512";
```

Для идентификации российских алгоритмов подписи и хэширования внутри XML в КриптоПро CSP 2.0, 3.0, 3.6 и ранних версиях КриптоПро JCP использовались следующие пространства имен: Для алгоритма хэширования использовалось пространство имен `http://www.w3.org/2001/04/xmldsig-more#gostr3411`, а для алгоритма подписи `http://www.w3.org/2001/04/xmldsig-more#gostr34102001-gostr3411`. Эти пространства имен использовать не рекомендуется, хотя работоспособность полностью сохранена для совместимости. С появлением нового проекта стандарта рекомендуется использовать для алгоритма подписи `urn:ietf:params:xml:ns:cpxmlsec:algorithms:gostr34102001-gostr3411` и для алгоритма хэширования `urn:ietf:params:xml:ns:cpxmlsec:algorithms:gostr3411`.

На основе методов перечисленных выше пакетов криптопровайдер КриптоПро JCSP позволяет осуществлять формирование подписи как отдельного объекта XML-документа, так и всего содержимого XML-документа (соответственно, проверку подписи как объекта, так и всего содержимого документа). Помимо этого существует возможность формирования и проверки двух независимых подписей одного XML-документа. Все перечисленные способы создания и проверки электронной подписи XML-документа для алгоритма ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2001 подробно описываются в примерах, которые входят в комплект поставки программного обеспечения КриптоПро JCSP (`samples/samples_src.jar/xmlSign/`).

Внимание! Библиотека JCPxml.jar должна находиться вместе с библиотекой xmlsec.jar, так чтобы ClassLoader при загрузке класса, реализующего алгоритм ГОСТ из библиотеки JCPxml.jar, имел доступ к базовому классу, который находится в библиотеке xmlsec.jar. Например, если серверное приложение, которое должно проверять подпись, запущено на сервере J2EE и включает в себя библиотеку xmlsec.jar, а JCPxml.jar установлена в lib/ext, появится конфликт, который сделает невозможной подпись/проверку. Системный ExtClassLoader, который осуществляет загрузку из lib/ext, не будет иметь доступ к базовому классу и не сможет загрузить классы из JCPxml.jar. В свою очередь ClassLoader приложения (WebappClassLoader) не будет иметь доступ к классам из JCPxml.jar. Для устранения конфликта можно переложить библиотеку JCPxml.jar в приложение к библиотеке xmlsec.jar.

8. КриптоПро Java CSP и Cryptographic Message Syntax (CMS)

Криптопровайдер КриптоПро JCSP позволяет осуществлять формирование и проверку электронной подписи в соответствии с алгоритмами подписи ГОСТ Р 34.10-2001 и ГОСТ Р 34.10-2012 и алгоритмами хэширования ГОСТ Р 34.11-94 и ГОСТ Р 34.11-2012 для сообщений, созданных на основе Cryptographic Message Syntax (CMS).

Примеры создания и подписи сообщений CMS, а также проверки подписи входят в комплект поставки программного обеспечения КриптоПро JCSP (`samples/samples_src.jar/CMS_samples/`). В соответствии с Cryptographic Message Syntax подпись может быть 2х видов: подпись на данные и подпись на подписываемые атрибуты подписи (если они существуют (см. CMS)), что и реализовано в примерах.

8.1. Использование утилиты ComLine

При выполнении команды CheckConfFull происходит проверка всех установленных модулей, в том числе и JCSP. Проверки JCSP включают: выполнение пробной подписи и проверки подписи, генерацию ключей и создание хранилища доверенных сертификатов, зашифрование и расшифрование текста, соединение по протоколу TLS 1.0 с использованием клиентской аутентификации и без (JCSP + cpSSL).

8.2. Особенности использования с другими модулями

Чтобы использовать установленный провайдер JCSP в CAdES вместо JCP, перед выполнением кода следует записать следующую команду:

```
CAdESConfig.setDefaultDigestSignatureProvider("JCSP"); // класс CAdESConfig входит в пакет ru.CryptoPro.CAdES
```

либо

```
System.setProperty("ru.CryptoPro.defaultProv", "JCSP");
```

Контексты ключей, хэшей, контейнера освобождаются в методах finalize() объектов соответствующих классов.

9. Использование библиотеки CAdES.jar для создания, проверки и усовершенствования подписи формата CAdES-BES, CAdES-T и CAdES-X Long Type 1

В состав дистрибутива «КриптоПро JCSP» версия 5.0 входит библиотека CAdES.jar. Ее назначение — создание, проверка и усовершенствование подписи формата CAdES-BES, CAdES-T и CAdES-X Long Type 1.

CAdES (CMS Advanced Electronic Signatures) — это стандарт электронной подписи, расширяющий версию стандарта электронной подписи CMS и разработанный ETSI. Главным документом, который описывает данный стандарт, является ETSI TS 101 733 Electronic Signature and Infrastructure (ESI) (https://tools.ietf.org/html/rfc5126, https://www.etsi.org/deliver/etsi_ts/101700_101799/101733/01.08.01_60/ts_101733v010801p.pdf).

CAdES-BES (Basic Electronic Signature) - основной и простейший формат электронной подписи, описываемый в стандарте CAdES. Он обеспечивает базовую проверку подлинности данных и защиту их целостности. Включенные в него атрибуты должны присутствовать и в других форматах CadES. CAdES-BES содержит следующие атрибуты:

•набор обязательных подписываемых атрибутов (определено в CAdES). Атрибуты называются подписанными, если генерация подписи происходит от совокупности этих атрибутов и данных пользователя;

•значение цифровой подписи, вычисленное для данных пользователя и подписываемых атрибутов. Для вычисления этого значения обычно используются алгоритмы генерации цифровой подписи.

Также CAdES-BES может содержать:

•набор дополнительных атрибутов;

•набор необязательных подписываемых атрибутов.

Может содержать подписываемые данные пользователя, под которыми понимается документ или сообщение подписывающей стороны.

Помимо подписываемых атрибутов, в CAdES-BES может быть включен неподписываемый атрибут counter-signature. Он определяет факт многократного подписывания сообщения.

CAdES-T (Timestamp) - это формат электронной подписи с доверенным временем. Доверенное время может быть указано следующими способами:

•с помощью неподписываемого атрибута signature-time-stamp, включенного в электронную подпись;

•с помощью отметки времени, представленной поставщиком доверенных услуг (Trusted Service Provider).

Иногда возникает ситуация, в которой использованные сертификаты, будучи действительными на момент генерации подписи, были отозваны после этого. Поэтому для доказательства того, что данные были подписаны до отзыва сертификатов, и что эти данные существовали на определенный момент времени, используются штампы времени.

CAdES-X Long Type 1 представляет собой подпись формата CAdES-T, в которую добавлены неподписываемые атрибуты complete-certificate-references и complete-revocation-references, certificate-values и revocation-values и штамп времени CAdES-C-time-stamp. complete-certificate-references содержит идентификаторы всех сертификатов, использующихся при проверке подписи. complete-revocation-references содержит идентификаторы сертификатов из списка отзыва сертификатов (Certificate Revocation Lists, CRL) и/или ответы протокола установления статуса сертификатов (Online Certificate Status Protocol, OCSP), которые используются для проверки подписи. Атрибут CAdES-C-time-stamp содержит штамп времени на всей подписи (бинарной подписи и ее

атрибутов). Это обеспечивает целостность и наличие доверенного времени во всех элементах подписи. Тем самым, этот атрибут позволяет защитить сертификаты, списки отзыва сертификатов и ответы протокола установления статуса сертификатов, информация о которых записана в подписи, при компрометации ключа центра сертификации, ключа издателя списка отзыва сертификатов или ключа издателя протокола установления статуса сертификатов. Атрибуты certificate-values и revocation-values представляют собой полные данные сертификатов и списки отзыва сертификатов. Этим обеспечивается доступ ко всей информации о сертификатах и отзывах, необходимых для проверки подписи (даже если их исходный источник недоступен), и предотвращается возможность утери этой информации. Внутренний штамп времени на подпись signature-timestamp также дополняется атрибутами complete-certificate-references, complete-revocation-references, certificate-values и revocation-values.

Библиотека CAdES.jar имеет несколько зависимостей:

•установленный «КриптоПро JCSP» версия 5.0;

•зависимость от библиотек bouncycastle версии jdk15on-1.50: bcpkix-jdk15on-1.50.jar и bcprov-jdk15on-1.50.jar;

•зависимость от библиотеки AdES-core.jar.

Перед установкой CAdES.jar рекомендуется скопировать в папку, куда производится установка, библиотеки bouncycastle и AdES-core.jar, убедившись, что «КриптоПро JCSP» версия 5.0 установлен. Установить CAdES можно несколькими способами:

•с помощью setup.exe (в ОС Windows) – в группе ”CAdES, XAdES”;

•с помощью setup_console – в группе ”CAdES, XAdES”;

•с помощью командной строки, путем последовательного вызова классов-установщиков сначала модуля AdES-core (java -jar AdES-core.jar), затем — установщика модуля CAdES.jar (java -jar CAdES.jar);

•простым копированием файлов, например, в папку JRE/lib/ext.

Документация CAdES, включающая описание классов и методов, а также примеры работы, находится в папке javadoc дистрибутива в файле CAdES-javadoc.jar. Полные тексты примеров создания, проверки, усовершенствования, заверения и т. д. находятся в пакете CAdES файла samples-sources.jar.

CAdES предоставляет несколько классов: CAdESSigner, CAdESSignature и EnvelopedSignature.

Поддерживается создание подписей формата:

•CAdES-BES

•CAdES-T

•CAdES-X Long Type 1

Поддерживается усовершенствование подписей формата:

•CAdES-BES до CAdES-T

•CAdES-BES до CAdES-X Long Type 1

•CAdES-T до CAdES-X Long Type 1

Пример создания CAdES-BES и CAdES-X Long Type 1 подписей.

```
System.setProperty("com.sun.security.enableCRLDP", "true");
System.setProperty("com.ibm.security.enableCRLDP", "true");

// Закрытый ключ подписи.
PrivateKey privateKey = ...;

// Цепочка сертификатов подписи.
List<X509Certificate> chain = ...;

// Создаем CAdES подпись.
CAdESSignature cadesSignature = new CAdESSignature(false);

// Добавляем CAdES-BES подпись №1. Также можно передать CRL для проверки цепочки подписанта вместо использования enableCRLDP
cadesSignature.addSigner(JCP.PROVIDER_NAME, JCP.GOST_DIGEST_OID,
JCP.GOST_EL_KEY_OID, privateKey, chain, CAdESType.CAdES_BES, null, false);

// Добавляем CAdES-X Long Type 1 подпись №2.
cadesSignature.addSigner(JCP.PROVIDER_NAME, JCP.GOST_DIGEST_OID,
JCP.GOST_EL_KEY_OID, privateKey, chain, CAdESType.CAdES_X_Long_Type_1,
"http://www.cryptopro.ru:80/tsp/", false);

// Данные для подписи в виде массиве.
byte[] data = ...;

// Будущая подпись в виде массива.
ByteArrayOutputStream signatureStream = new ByteArrayOutputStream();

cadesSignature.open(signatureStream); // подготовка контекста
cadesSignature.update(data); // хеширование

cadesSignature.close(); // создание подписи с выводом в signatureStream
signatureStream.close();

// Получаем подпись с двумя подписантами в виде массиве.
byte[] cadesCms = signatureStream.toByteArray();
```

Пример проверки подписи CAdES-BES.

```
// Исходная CAdES-BES подпись в виде потока байтов из файла.
FileInputStream cadesCms = new FileInputStream("signature.file");



// Цепочка сертификатов подписи.
```

```
List<X509Certificate> chain = ...;
// Сертификаты для проверки подписи.
Set<X509Certificate> certs = ...;
// CRL для проверки подписи.
Set<X509CRL> cRLs = ...;

// Декодируем и проверяем совмещенную CAdES-BES подпись.
CAdESSignature cadesSignature = new CAdESSignature(cadesCms, null,
CAdESType.CAdES_BES); // декодирование с типом CAdESType.CAdES_BES

cadesSignature.verify(certs, cRLs); // проверка, если необходима
cadesCms.close();
```

Если список CRL отсутствует, то можно включить проверку цепочки сертификатов онлайн с обращением к CRL по сети:

```
System.setProperty("com.sun.security.enableCRLDP", "true");
System.setProperty("com.ibm.security.enableCRLDP", "true");

// Исходная CAdES-BES подпись в виде потока байтов из файла.
FileInputStream cadesCms = new FileInputStream("signature.file");

// Цепочка сертификатов подписи.
List<X509Certificate> chain = ...;
// Сертификаты для проверки подписи.
Set<X509Certificate> certs = ...;

// Декодируем и проверяем совмещенную CAdES-BES подпись.
CAdESSignature cadesSignature = new CAdESSignature(cadesCms, null,
CAdESType.CAdES_BES); // декодирование с типом CAdESType.CAdES_BES

cadesSignature.verify(certs); // проверка, если необходима
cadesCms.close();
```

Проверка подписи формата CAdES-X Long Type 1, находящей на первом месте в списке подписантов.

```
// Исходная подпись в виде потока байтов из файла.
FileInputStream cadesCms = new FileInputStream("signature.file");

// Декодируем совмещенную подпись с автоопределением типов.
CAdESSignature cadesSignature = new CAdESSignature(cadesCms, null, null);



// Подписант с типом CadES-X Long Type 1.
```

```
CAdESSigner signer = cadesSignature.getCAdESSignerInfo(0);

// Проверка подписи.
signer.verify(null);
```

Пример усовершенствования подписи формата CAdES-BES до CAdES-X Long Type 1.

```
// Исходная CAdES-BES подпись в виде потока байтов из файла.
FileInputStream cadesCms = new FileInputStream("signature.file");

// Цепочка сертификатов подписи.
List<X509Certificate> chain = ...;

// Декодируем совмещенную подпись с автоопределением типов.
// В этой подписи только один подписант!
CAdESSignature cadesSignature = new CAdESSignature(cadesCms, null, null);

// Подписант с типом CadES-BES.
CAdESSigner signer = cadesSignature.getCAdESSignerInfo(0);

// Усовершенствуем подпись данного подписанта до CAdES-X Long Type 1.
// Подписант нового класса будет возвращен функцией.
signer = signer.enhance(JCP.PROVIDER_NAME, JCP.GOST_DIGEST_OID, chain,
"http://www.cryptopro.ru:80/tsp/", CAdESType.CAdES_X_Long_Type_1);

// Получаем усовершенствованную подпись.
SignerInformation enhSigner = signer.getSignerInfo();

// Составляем новый список, чтобы заменить подписанта.
// В этой подписи только один подписант!
SignerInformationStore dstSignerInfoStore =
new SignerInformationStore(Collections.singletonList(enhSigner));

// Исходная подпись в файле.
FileInputStream srcSignedData = new FileInputStream("signature.file");

// Будущая усовершенствованная подпись в файле.
FileOutputStream dstSignedData = new FileOutputStream("enhanced_signature.file");

// В исходной подписи srcSignedData заменяем подписанта на нового.
CAdESSignature.replaceSigners(srcSignedData, dstSignerInfoStore, dstSignedData);

srcSignedData.close();
dstSignedData.close();
```

Пример зашифрования сообщения в адрес получателя.

```
// Буфер для сохранения подписи Enveloped CMS
ByteArrayOutputStream envelopedByteArrayOutStream = new ByteArrayOutputStream();

// Создание объекта Enveloped CMS
EnvelopedSignature signature = new EnvelopedSignature();

// Добавление получателя (сертификат). При расшифровании получатель
// будет использовать закрытый ключ, соответствующий данному сертификату
signature.addKeyTransRecipient(recipientCertificate); // структура key_trans,
  допускается только ключ обмена
// или
// signature.addKeyAgreeRecipient(recipientCertificate); // структура key_agree

// Инициализация Enveloped CMS буфером для сохранения подписи
signature.open(envelopedByteArrayOutStream);

// Подготовленные данные для зашифрования - строка или подпись,
// полученная с помощью CMSSign (samples.jar) или CAdES API
byte[] data = ...

// Зашифрование данных data
signature.update(data, 0, data.length);

// Формирование подписи Enveloped CMS
signature.close();

// Получение подписи в формате Enveloped CMS в буфер
byte[] envelopedByteData = envelopedByteArrayOutStream.toByteArray();
```

Пример расшифрования сообщения получателем.

```
// Буфер для сохранения расшифрованных данных
ByteArrayOutputStream decryptedByteDataStream = new ByteArrayOutputStream();

// Прочитанное в буфер сообщение формата Enveloped CMS
byte[] envelopedByteData = ...

// Создание объекта Enveloped CMS с передачей ему буфера подписи для расшифрования
signature = new EnvelopedSignature(new ByteArrayInputStream(envelopedByteData));

// Расшифрование подписи на закрытом ключе получателя с записью
// расшифрованных данных в буфер decryptedByteDataStream
signature.decrypt(recipientCertificate, recipientPrivateKey,
  decryptedByteDataStream);
```

```
// Получение расшифрованных данных - строки или подпись, которую можно
// далее проверить с помощью CMSVerify (samples.jar) или CAdES.jar
byte[] decryptedByteData = decryptedByteDataStream.toByteArray();
```

Класс CadESSignature используется для декодирования подписи формата CAdES перед проверкой или для подготовки подписи при ее создании. Данные подписи могут быть переданы в виде входного потока InputStream. При проверке подпись может быть декодирована как с автоматическим определением типа, так и с заданным типом. Подпись формата CAdES-X Long Type 1 может быть проверена, например, как CAdES-BES или CAdES-T, если при декодировании подписи передать в конструктор соответствующий тип.

Класс CAdESSigner используется для представления декодированного подписанта, и объекты этого типа доступны только при проверке подписи. В подписанном сообщении их может быть несколько. Интерфейсы CAdESSignerT и CAdESSignerXLT1 предоставляют дополнительные функции для получения различных сведений о подписи. Класс CAdESSigner содержит функцию verify(), которая также, как и CAdESSignature, позволяет указать, с каким типом проверить подпись. Так, например, объект класса CAdESSignerTImpl, т. е. подпись формата CAdES-T, может быть проверена, как CAdES-BES, если в функцию verify() подписанта передать требуемый тип.

До версии 2.0.39267 включительно полная проверка цепочки сертификатов оператора службы внутреннего штампа не выполнялась. В текущей версии данная проверка выполняется (для T-подписи), но может быть отключена с помощью параметра ru.CryptoPro.AdES.validate_tsp (например, -Dru.CryptoPro.AdES.validate_tsp=false). Текущая версия также отличается более жесткой политикой в отношении наличия доказательства (CRL, OCSP) для сертификата службы штампа в усовершенствованном внутреннем штампе времени, однако в целях совместимости с предыдущими версиями проверка отключена. Она может быть включена с помощью параметра ru.CryptoPro.AdES.require_tsp_evidence (например, -Dru.CryptoPro.AdES.require_tsp_evidence=true).

В текущей версии также при создании подписи (addSigner) или ее усовершенствовании (enhance) можно передать CRL для проверки цепочки подписанта или сертификатов службы штампов или в качестве дополнительного источника доказательств.

Класс EnvelopedSignature используется для создания зашифрованного сообщения типа Enveloped CMS или его расшифрования. В качестве входных данных может выступать как подпись формата CAdES или CMS, так и данные любого другого формата (это следует учитывать при расшифровании сообщения адресатом). Шифруемые или расшифровываемые данные могут быть переданы в виде входного потока InputStream.

10.Использование библиотеки XAdES.jar для создания и проверки подписи формата XadES-BES, XadES-T и XadES-X Long Type 1

В состав дистрибутива «КриптоПро JCSP» версия 5.0 входит библиотека XAdES.jar. Ее назначение — создание и проверка подписи формата XAdES-BES, XadES-T и XadES-X Long Type 1.

XAdES (XMLDSig Advanced Electronic Signatures) — это стандарт электронной подписи, расширяющий версию стандарта электронной подписи XMLDSig (https://www.etsi.org/deliver/etsi_ts/101900_101999/101903/01.03.02_60/ts_101903v010302p.pdf).

Форматы XAdES во многом совпадают с форматами CAdES, но оформлены в соответствии со стандартом XMLDSig.

Библиотека XAdES.jar имеет несколько зависимостей:

•установленный «КриптоПро JCSP» версия 5.0;

•зависимость от библиотек bouncycastle версии jdk15on-1.50: bcpkix-jdk15on-1.50.jar и bcprov-jdk15on-1.50.jar;

•зависимость от библиотек AdES-core.jar и CAdES.jar.

Перед установкой XAdES.jar рекомендуется скопировать в папку, куда производится установка, библиотеки bouncycastle и установить/скопировать AdES-core.jar и CAdES.jar, убедившись, что «КриптоПро JCSP» версия 5.0 установлен. Установить XAdES можно, как и CAdES.jar, несколькими способами:

•с помощью setup.exe (в ОС Windows) – в группе ”CAdES, XAdES”;

•с помощью setup_console – в группе ”CAdES, XAdES”;

•с помощью командной строки, путем последовательного вызова классов-установщиков сначала модуля AdES-core (java -jar AdES-core.jar), затем — установщика модуля CAdES.jar (java -jar CAdES.jar), и XAdES.jar (java -jar XAdES.jar);

•простым копированием файлов, например, в папку JRE/lib/ext.

Документация XAdES, включающая описание классов и методов, а также примеры работы, находится в папке javadoc дистрибутива в файле XAdES-javadoc.jar. Полные тексты примеров создания и проверки находятся в пакете xades файла samples-sources.jar.

XAdES предоставляет XAdES API, в который входят классы XAdESSignature и XAdESSigner.

Поддерживается создание подписей формата:

•XAdES-BES

•XadES-T

•XadES-X Long Type 1

Пример создания XAdES-BES подписи.

```
System.setProperty("com.sun.security.enableCRLDP", "true");
```

```
System.setProperty("com.ibm.security.enableCRLDP", "true");

String documentContext =
"<?xml version=\"1.0\"?>\n" +
"<PatientRecord>    \n" +
"    <Name>John Doe</Name>    \n" +
"    <Account Id=\"acct\">123456</Account>    \n" +
"    <BankInfo Id=\"bank\">HomeBank</BankInfo>    \n" +
"    <Visit date=\"10pm March 10, 2002\">    \n" +
"        <Diagnosis>Broken second metacarpal</Diagnosis>    \n" +
"    </Visit>\n" +
"</PatientRecord>";

String ref_acct = "acct"; // ссылка на подписываемый узел

// декодирование документа
DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
dbFactory.setNamespaceAware(true);
Document document = dbFactory.newDocumentBuilder().parse(
new ByteArrayInputStream(documentContext.getBytes()));

XPathFactory factory = XPathFactory.newInstance();
XPath xpath = factory.newXPath();

XPathExpression expr = xpath.compile(String.format("//*[@Id='%s']", ref_acct));
NodeList nodes = (NodeList) expr.evaluate(document, XPathConstants.NODESET);

Node node = nodes.item(0);
String referenceURI = "#" + ref_acct;

// Подписываемая ссылка.
DataObjects dataObjects = new DataObjects(Arrays.asList(referenceURI));
dataObjects.addTransform(new EnvelopedTransform());

PrivateKey privateKey = ... // ключ подписи
List<X509Certificate> chain = ... // цепочка сертификатов подписи

XAdESSignature xAdESSignature = new XAdESSignature();

// добавляем подписанта формата XadES-BES. Также можно передать CRL для проверки
цепочки подписанта вместо использования enableCRLDP
xAdESSignature.addSigner(JCP.PROVIDER_NAME, null, privateKey, chain,
  XAdESType.XAdES_BES, null);

FileOutputStream fileOutputStream = new FileOutputStream("signed.xml");
xAdESSignature.open(fileOutputStream);
```

```
// Подписание.
xAdESSignature.update((Element) node, dataObjects);
xAdESSignature.close();
```

Пример проверки всех подписей формата XAdES в XML документе.

```
// декодирование документа с подписью
DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
dbFactory.setNamespaceAware(true);
Document document = dbFactory.newDocumentBuilder().parse(new
  FileInputStream("signed.xml"));

Set<X509Certificate> certs = ... // дополнительные сертификаты для построения
цепочки
Set<X509CRL> cRLs = ... // CRL для проверки цепочки сертификатов
XAdESSignature xAdESSignature = new
  XAdESSignature(document.getDocumentElement(), XAdESType.XAdES_BES);
xAdESSignature.verify(certs, cRLs);
```

Если список CRL отсутствует, то можно включить проверку цепочки сертификатов онлайн с обращением к CRL по сети:

```
System.setProperty("com.sun.security.enableCRLDP", "true");
System.setProperty("com.ibm.security.enableCRLDP", "true");

DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
dbFactory.setNamespaceAware(true);
Document document = dbFactory.newDocumentBuilder().parse(new
  FileInputStream("signed.xml"));

Set<X509Certificate> certs = ... // дополнительные сертификаты для построения
цепочки
XAdESSignature xAdESSignature = new
  XAdESSignature(document.getDocumentElement(), XAdESType.XAdES_BES);
xAdESSignature.verify(certs);
```

Пример проверки отдельной подписи XAdES-BES в XML документе.

```
// декодирование документа с подписью
DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
dbFactory.setNamespaceAware(true);
Document document = dbFactory.newDocumentBuilder().parse(new
FileInputStream("signed.xml"));
Set<X509Certificate> certs = ... // дополнительные сертификаты для построения
цепочки
Set<X509CRL> cRLs = ... // CRL для проверки цепочки сертификатов
```

```
	XAdESSignature xAdESSignature = new XAdESSignature(document.getDocumentElement(), XadESType.XAdES_BES); // декодирование с типом XAdES-BES

	XAdESSigner xAdESSigner = xAdESSignature.getXAdESSignerInfo(0);


	// Проверка отдельной подписи с порядковым номером 0

	xAdESSigner.verify(certs, cRLs);
```

Класс XAdESSignature используется для декодирования подписи формата XAdES перед проверкой или для подготовки подписи при ее создании. При проверке подпись может быть декодирована как с автоматическим определением типа, так и с заданным типом. Подпись формата XAdES-T может быть проверена, например, как XAdES-BES, если при декодировании подписи передать в конструктор соответствующий тип.

Класс XAdESSigner используется для представления декодированного подписанта, и объекты этого типа доступны только при проверке подписи. В подписанном сообщении их может быть несколько. Интерфейс XAdESSignerT предоставляет дополнительные функции для получения различных сведений о подписи. Класс XAdESSigner содержит функцию verify(), которая также, как и XAdESSignature, позволяет указать, с каким типом проверить подпись. Так, например, объект класса XAdESSignerTImpl, т. е. подпись формата XAdES-T, может быть проверена, как XAdES-BES, если в функцию verify() подписанта передать требуемый тип.

До версии 2.0.39267 включительно полная проверка цепочки сертификатов оператора службы внутреннего штампа не выполнялась. В текущей версии данная проверка выполняется (для T-подписи), но может быть отключена с помощью параметра ru.CryptoPro.AdES.validate_tsp (например, -Dru.CryptoPro.AdES.validate_tsp=false). Текущая версия также отличается более жесткой политикой в отношении наличия доказательства (CRL, OCSP) для сертификата службы штампа в усовершенствованном внутреннем штампе времени, однако в целях совместимости с предыдущими версиями проверка отключена. Она может быть включена с помощью параметра ru.CryptoPro.AdES.require_tsp_evidence (например, -Dru.CryptoPro.AdES.require_tsp_evidence=true).

В текущей версии также при создании подписи (addSigner) или ее усовершенствовании (enhance) можно передать CRL для проверки цепочки подписанта или сертификатов службы штампов или в качестве дополнительного источника доказательств.

11.Использование утилиты keytool

При работе криптопровайдером КриптоПро JCSP операции

- генерации ключа электронной подписи с алгоритмом ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012 и соответствующего ему самоподписанного сертификата с записью их на один из носителей;
- генерации запроса на сертификат ключа проверки электронной подписи в соответствии с хранящимся на носителе ключом электронной подписи;
- генерации самоподписанного сертификата ключа проверки электронной подписи в соответствии с хранящимся на носителе ключом электронной подписи и запись сертификата на носитель;
- чтение сертификата ключа проверки электронной подписи с носителя и запись его в файл;
- чтение сертификата открытого ключа из файла и запись его на носитель в соответствии с хранящимся на носителе ключом электронной подписи;
- чтение доверенного сертификата из хранилища и запись его в файл;
- чтение доверенного сертификата из файла и запись его в хранилище

могут осуществляться не только через стандартный интерфейс JCA, но также при помощи утилиты keytool.

При генерации самоподписанного сертификата при помощи утилиты keytool никакие расширения в сертификат не устанавливаются. Для генерации сертификатов с расширениями следует воспользоваться методами класса `GostCertificateRequest` (см. выше).

Ниже приводятся примеры осуществления перечисленных операций при помощи данной утилиты.

11.1.Просмотр содержимого ключевого носителя

В данном примере осуществляется просмотр содержимого ключевого носителя (жесткий диск) и проинициализированного именем этого носителя хранилища доверенных сертификатов.

Просмотр содержимого осуществляется при помощи команды **`-list`**, которой в качестве параметров передаются:

- тип провайдера (КриптоПро JCSP): **`-provider`** `ru.CryptoPro.JCP.JCP`
- имя ключевого носителя (жесткий диск): **`-storetype`** `HDIMAGE`
- путь к хранилищу доверенных сертификатов, проинициализированному именем носителя: **`-keystore`** `c:\.keystore`
- пароль на хранилище доверенных сертификатов: **`-storepass`** `123456`

Таким образом, просмотр содержимого носителя и соответствующего ему хранилища доверенных сертификатов осуществляется:

```
keytool -list -provider ru.CryptoPro.JCSP.JCSP -storetype HDIMAGE -keystore
c:\.keystore -v -storepass 123456
```

11.2.Генерация ключа и соответствующего ему самоподписанного сертификата и запись их на носитель

В данном примере осуществляется генерация ключа электронной подписи и соответствующего ему самоподписанного сертификата в соответствии с алгоритмом ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012 и запись их на носитель.

Генерация и запись ключа и сертификата осуществляется при помощи команды **`-genkey`**, которой в качестве параметров передаются:

- уникальное имя создаваемого ключа и соответствующего ему сертификата: **-alias** myKey
- длина создаваемого ключа (в соответствии с алгоритмом ГОСТ Р 34.10.2001): **-keysize** 512
- тип провайдера (КриптоПро JCSP): **-provider** ru.CryptoPro.JCSP.JCSP
- пароль на записываемый ключ: **-keypass** 11111111
- имя ключевого носителя (жесткий диск): **-storetype** HDIMAGE
- имя создаваемого сертификата по стандарту X.500: **-dname** CN=myKey,O=CryptoPro,C=RU
- путь к хранилищу доверенных сертификатов, проинициализированному именем носителя: **-keystore** c:\.keystore
- пароль на хранилище доверенных сертификатов: **-storepass** 123456
- алгоритм генерации ключа ГОСТ Р 34.10-2001: **-keyalg** GOST3410EL
- алгоритм подписи сертификата (ГОСТ Р 34.10-2001): **-sigalg** GOST3411withGOST3410EL

Таким образом, генерация ключа электронной подписи и соответствующего ему самоподписанного сертификата и запись их на носитель осуществляется:

```
keytool -genkey -alias myKey -keysize 512 -provider ru.CryptoPro.JCSP.JCSP -keypass 11111111 -storetype HDIMAGE -dname CN=myKey,O=CryptoPro,C=RU -keystore c:\.keystore -storepass 123456 -keyalg GOST3410EL -sigalg GOST3411withGOST3410EL
```

11.3.Генерация ключевой пары запись ее на носитель

В данном примере осуществляется генерация ключевой пары в соответствии с алгоритмом ГОСТ Р 34.10-2001 или ГОСТ Р 34.10-2012 и запись ее на носитель.

Генерация и запись ключевой пары осуществляется при помощи команды **-genkeypair**, которой в качестве параметров передаются:

- уникальное имя создаваемого ключа и соответствующего ему сертификата: **-alias** myKey
- длина создаваемого ключа (в соответствии с алгоритмом ГОСТ Р 34.10.2001): **-keysize** 512
- имя провайдера (КриптоПро JCSP): **-providername** JCSP
- имя ключевого носителя (жесткий диск): **-storetype** HDIMAGE
- алгоритм генерации ключа ГОСТ Р 34.10-2001: **-keyalg** GOST3410EL
- алгоритм подписи сертификата (ГОСТ Р 34.10-2001): **-sigalg** GOST3411withGOST3410EL

Таким образом, генерация ключевой пары и запись ее на носитель осуществляется:

```
keytool -genkeypair -alias myKey -keysize 512 -providername JCSP -storetype HDIMAGE -keyalg GOST3410EL -sigalg GOST3411withGOST3410EL -keystore c:\.keystore -storepass 123456 -keypass 11111111
```

11.4.Генерация запроса на сертификат ключа проверки электронной подписи в соответствии с хранящимся на носителе ключом электронной подписи и запись запроса в файл

В данном примере осуществляется генерация запроса на сертификат ключа проверки электронной подписи в соответствии с хранящимся на носителе ключом электронной подписи и запись запроса в файл.

Генерация и запись в файл запроса осуществляется при помощи команды **-certreq**, которой в качестве параметров передаются:

- уникальное имя ключа электронной подписи на носителе, в соответствии с которым осуществляется генерация запроса на сертификат: **-alias** myKey
- тип провайдера (КриптоПро JCSP): **-provider** ru.CryptoPro.JCSP.JCSP
- пароль на ключ электронной подписи: **-keypass** 11111111
- имя ключевого носителя (жесткий диск): **-storetype** HDIMAGE
- путь к хранилищу доверенных сертификатов, проинициализированному именем носителя: **-keystore** c:\.keystore
- пароль на хранилище доверенных сертификатов: **-storepass** 123456
- алгоритм подписи запроса на сертификат (ГОСТ Р 34.10-2001): **-sigalg** GOST3411withGOST3410EL
- путь к файлу для записи в него запроса: **-file** c:\request.bin

Таким образом, генерация запроса на сертификат ключа проверки электронной подписи в соответствии с хранящимся на носителе ключом электронной подписи и запись запроса в файл осуществляется:

```
keytool -certreq -alias myKey -provider ru.CryptoPro.JCSP.JCSP -keypass 11111111
-storetype HDIMAGE -keystore c:\.keystore -storepass 123456 -sigalg
GOST3411withGOST3410EL -file c:\request.bin
```

11.5.Генерация самоподписанного сертификата ключа проверки электронной подписи в соответствии с хранящимся на носителе ключом электронной подписи и запись сертификата на носитель

В данном примере осуществляется генерация самоподписанного сертификата ключа проверки электронной подписи в соответствии с хранящимся на носителе ключом электронной подписи и запись сертификата на носитель. Если на носителе уже существует сертификат ключа проверки электронной подписи, соответствующий данному ключу электронной подписи, то он будет перезаписан.

Генерация и запись на носитель самоподписанного сертификата осуществляется при помощи команды **-selfcert**, которой в качестве параметров передаются:

- уникальное имя ключа электронной подписи на носителе, в соответствии с которым осуществляется генерация самоподписанного сертификата: **-alias** myKey
- тип провайдера (КриптоПро JCSP): **-provider** ru.CryptoPro.JCSP.JCSP
- пароль на ключ электронной подписи: **-keypass** 11111111
- имя ключевого носителя (жесткий диск): **-storetype** HDIIMAGE
- путь к хранилищу доверенных сертификатов, проинициализированному именем носителя: **-keystore** c:\.keystore
- пароль на хранилище доверенных сертификатов: **-storepass** 123456

- алгоритм подписи сертификата (ГОСТ Р 34.10-2001): **-sigalg** GOST3411withGOST3410EL

- имя создаваемого сертификата по стандарту X.500: **-dname** CN=myKey,O=CryptoPro,C=RU

Таким образом, генерация самоподписанного сертификата ключа проверки электронной подписи в соответствии с хранящимся на носителе ключом электронной подписи и запись сертификата на носитель осуществляется:

```
keytool -selfcert -alias myKey -provider ru.CryptoPro.JCSP.JCSP -keypass 11111111
-storetype HDIMAGE -keystore c:\.keystore -storepass 123456 -sigalg
GOST3411withGOST3410EL -dname CN=myKey,O=CryptoPro,C=RU
```

11.6.Чтение сертификата ключа проверки электронной подписи с носителя и запись его в файл

В данном примере осуществляется чтение сертификата ключа проверки электронной подписи с носителя и запись сертификата в файл.

Чтение сертификата ключа проверки электронной подписи с носителя и запись его в файл осуществляется при помощи команды **-export**, которой в качестве параметров передаются:

- уникальное имя читаемого с носителя сертификата ключа проверки электронной подписи: **-alias** myKey

- тип провайдера (КриптоПро JCSP): **-provider** ru.CryptoPro.JCSP.JCSP

- имя ключевого носителя (жесткий диск): **-storetype** HDIMAGE

- путь к хранилищу доверенных сертификатов, проинициализированному именем носителя: **-keystore** c:\.keystore

- пароль на хранилище доверенных сертификатов: **-storepass** 123456

- путь к файлу для записи в него сертификата: **-file** c:\myKeyCert.cer

Таким образом, чтение сертификата ключа проверки электронной подписи с носителя и запись сертификата в файл осуществляется:

```
keytool -export -alias myKey -provider ru.CryptoPro.JCSP.JCSP -storetype HDIMAGE
-keystore c:\.keystore -storepass 123456 -file c:\myKeyCert.cer
```

11.7.Чтение сертификата ключа проверки электронной подписи из файла и запись его на носитель в соответствии с хранящимся на носителе ключом электронной подписи

В данном примере осуществляется чтение сертификата ключа проверки электронной подписи из файла и запись его на носитель в соответствии с хранящимся на носителе ключом электронной подписи.

Чтение сертификата ключа проверки электронной подписи из файла и запись его на носитель осуществляется при помощи команды **-import**, которой в качестве параметров передаются:

- уникальное имя ключа электронной подписи на носителе, в соответствии с которым на носитель записывается сертификат: **-alias** myKey

- тип провайдера (КриптоПро JCSP): **-provider** ru.CryptoPro.JCSP.JCSP

- пароль на ключ электронной подписи: **-keypass** 11111111

- имя ключевого носителя (жесткий диск): **-storetype** HDIMAGE

- путь к хранилищу доверенных сертификатов, проинициализированному именем носителя: **-keystore** c:\.keystore

- пароль на хранилище доверенных сертификатов: **-storepass** 123456

- путь к файлу для чтения из него сертификата: **-file** c:\myKeyCert.cer

Таким образом, чтение сертификата ключа проверки электронной подписи из файла и запись его на носитель в соответствии с хранящимся на носителе ключом электронной подписи осуществляется:

```
keytool -import -alias myKey -provider ru.CryptoPro.JCSP.JCSP -keypass 11111111
-storetype HDIMAGE -keystore c:\.keystore -storepass 123456 -file c:\myKeyCert.cer
```

11.8. Чтение доверенного сертификата из хранилища и запись его в файл

В данном примере осуществляется чтение доверенного сертификата из хранилища и запись сертификата в файл.

Чтение доверенного сертификата из хранилища и запись его в файл осуществляется при помощи команды **-export**, которой в качестве параметров передаются:

- уникальное имя читаемого из хранилища доверенного сертификата (предполагается, что на носителе нет ключа с тем же именем): **-alias** myCert
- тип провайдера (КриптоПро JCSP): **-provider** ru.CryptoPro.JCSP.JCSP
- имя ключевого носителя (жесткий диск): **-storetype** HDIMAGE
- путь к хранилищу доверенных сертификатов, проинициализированному именем носителя: **-keystore** c:\.keystore
- пароль на хранилище доверенных сертификатов: **-storepass** 123456
- путь к файлу для записи в него сертификата: **-file** c:\myCert.cer

Таким образом, чтение доверенного сертификата из хранилища и запись сертификата в файл осуществляется:

```
keytool -export -alias myCert -provider ru.CryptoPro.JCSP.JCSP -storetype HDIMAGE
-keystore c:\.keystore -storepass 123456 -file c:\myCert.cer
```

11.9. Чтение доверенного сертификата из файла и запись его в хранилище

В данном примере осуществляется чтение доверенного сертификата из файла и запись его в хранилище.

Чтение доверенного сертификата и запись его в хранилище осуществляется при помощи команды **-import**, которой в качестве параметров передаются:

- уникальное записываемого сертификата (предполагается, что на носителе нет ключа с тем же именем): **-alias** myCert
- тип провайдера (КриптоПро JCSP): **-provider** ru.CryptoPro.JCSP.JCSP
- имя ключевого носителя (жесткий диск): **-storetype** HDIMAGE
- путь к хранилищу доверенных сертификатов, проинициализированному именем носителя: **-keystore** c:\.keystore
- пароль на хранилище доверенных сертификатов: **-storepass** 123456
- путь к файлу для чтения из него сертификата: **-file** c:\myCert.cer

Таким образом, чтение сертификата ключа проверки электронной подписи из файла и запись его на носитель в соответствии с хранящимся на носителе ключом электронной подписи осуществляется:

```
keytool -import -alias myCert -provider ru.CryptoPro.JCSP.JCSP -storetype HDIMAGE
-keystore c:\.keystore -storepass 123456 -file c:\myCert.cer
```

11.10.Удаление ключа и соответствующего ему самоподписанного сертификата с носителя

В данном примере осуществляется удаление ключа электронной подписи и соответствующего ему самоподписанного сертификата с носителя.

Удаление ключа и соответствующего ему самоподписанного сертификата с носителя осуществляется при помощи команды **-delete**, которой в качестве параметров передаются:

- уникальное имя удаляемого ключа: **-alias** myKey
- тип провайдера (КриптоПро JCSP): **-provider** ru.CryptoPro.JCSP.JCSP
- имя ключевого носителя (жесткий диск): **-storetype** HDIMAGE
- путь к хранилищу доверенных сертификатов, проинициализированному именем носителя: **-keystore** c:\.keystore
- пароль на хранилище доверенных сертификатов: **-storepass** 123456

Таким образом, удаление ключа электронной подписи и соответствующего ему самоподписанного сертификата с носителя осуществляется:

```
keytool -delete -alias myKey -provider ru.CryptoPro.JCSP.JCSP -storetype HDIMAGE
-keystore c:\.keystore -v -storepass 123456
```

При удалении ключа с носителя, требующего пароля доступа к ключу на носителе при удалении (например, HDIMAGE) в качестве имени необходимо передать FQCN, при этом пароль будет запрошен через окно криптопровайдера КриптоПро CSP. Таким образом, удаление ключа электронной подписи и соответствующего ему самоподписанного сертификата со считывателя HDIMAGE осуществляется:

```
keytool -delete -alias //./HDIMAGE/cnt -provider ru.CryptoPro.JCSP.JCSP -storetype
HDIMAGE -keystore c:\.keystore -v -storepass 123456
```

11.11.Удаление доверенного сертификата из хранилища

В данном примере осуществляется удаление доверенного сертификата из хранилища.

Удаление доверенного сертификата из хранилища осуществляется при помощи команды **-delete**, которой в качестве параметров передаются:

- уникальное имя удаляемого сертификата (предполагается, что на носителе нет ключа с тем же именем): **-alias** myCert
- тип провайдера (КриптоПро JCSP): **-provider** ru.CryptoPro.JCSP.JCSP
- имя ключевого носителя (жесткий диск): **-storetype** HDIMAGE
- путь к хранилищу доверенных сертификатов, проинициализированному именем носителя: **-keystore** c:\.keystore
- пароль на хранилище доверенных сертификатов: **-storepass** 123456

Таким образом, удаление доверенного сертификата из хранилища осуществляется:

```
keytool -delete -alias myCert -provider ru.CryptoPro.JCSP.JCSP -storetype HDIMAGE
-keystore c:\.keystore -v -storepass 123456
```

12.Использование утилиты ComLine

Также можно воспользоваться готовыми классами пакета ComLine из модуля Samples, входящего в состав КриптоПро JCP. Запустите ComLine с вызовом нужного класса либо сам класс, используя следующие параметры командной строки:

`java ComLine NameofClass args` или `java NameofClass args`

например:

```
java ComLine KeyPairGen -alias name_of_key -dname CN=autor,OU=Security,O=CryptoPro,C=RU
-reqCertpath C:/req.txt
```

или

```
java KeyPairGen -alias name_of_key -dname CN=autor,OU=Security,O=CryptoPro,C=RU
-reqCertpath C:/req.txt
```

12.1.Проверка установки и настроек провайдеров.

Проверку установки и основных настроек провайдера можно осуществить запуском:

`CheckConf (без параметров)`

12.2.Проверка работоспособности провайдеров.

Запуском:

`CheckConfFull [-servDir C:/*.*]`

-servDir

рабочая директория

(по умолчанию текущая)

можно проверить работоспособность провайдеров.

Выполняются тесты на генерацию ключей, генерацию и проверку подписи, а также тесты на создание ssl-соединения (если установлен КриптоПро JTLS). (Запуск возможен при условии, что КриптоПро JCSP был установлен успешно).

12.3.Работа с ключами и сертификатами

12.3.1.Генерация ключевой пары и соответствующего ей самоподписанного сертификата. Запись их на носитель. Генерация запроса на сертификат и запись его в файл.

Генерация ключевой пары осуществляется в соответствии с алгоритмами ключевого обмена и подписи ГОСТ Р 34.10-2001.

`KeyPairGen -alias name_of_key [-alg GOST3410EL] [-storetype HDIMAGE] [-storepath null] [-storepass null] [-keypass password] [-isServer true] -dname CN=autor,OU=Security,O=CryptoPro,C=RU -reqCertpath C:/*.* -encoding der`

-alias

уникальное имя записываемого ключа

-alg

алгоритм для генерации

(по умолчанию GOST3410EL)

-storetype

имя ключевого носителя HDIMAGE (жесткий диск), REGISTRY (реестр ОС Windows), имя карточки или токена

(по умолчанию HDIMAGE)

-storepath

путь к хранилищу доверенных сертификатов

(по умолчанию null)

-storepass

пароль на хранилище доверенных сертификатов

(по умолчанию null)

-keypass

пароль на записываемый ключ

(по умолчанию null)

-isServer

если ключ серверный, то значение true

(по умолчанию false)

-dname

имя субъекта для генерации самоподписанного сертификата

-encoding

кодировка (DER/BASE64)

(по умолчанию DER)

-reqCertpath

путь для записи запроса

Полученные таким образом ключи можно использовать как для генерации электронной подписи, так и для обмена.

12.3.2.Получение сертификата из запроса. Запись сертификата в хранилище и в файл.

```
getCert -alias name_of_key [-storetype HDIMAGE] [-storepath null]
     [-storepass null] -http http://www.cryptopro.ru/certsrv/ -certpath C:/*.cer
-reqCertpath C:/*.*
```

-alias

уникальное имя ключа

-storetype

имя ключевого носителя HDIMAGE (жесткий диск), REGISTRY (реестр ОС Windows), имя карточки или токена

(по умолчанию HDIMAGE)

-storepath

путь к хранилищу доверенных сертификатов

(по умолчанию null)

-storepass

пароль на хранилище доверенных сертификатов

(по умолчанию null)

-http

путь к центру сертификации

-reqCertpath

путь к файлу с запросом

-encoding

кодировка запроса (DER/BASE64)

(по умочанию DER)

-certpath

путь к файлу для записи сертификата

12.3.3.Построение цепочки сертификатов.

```
Certs -alias name_of_key [-storetype HDIMAGE] [-storepath null] [-storepass null] [-keypass password] -certs C:/my.cer,C:/*.cer,...,C:/root.cer
```

-alias

уникальное имя ключа

-keypass

пароль на ключ

(по умолчанию null)

-storetype

имя ключевого носителя HDIMAGE (жесткий диск), REGISTRY (реестр ОС Windows), имя карточки или токена

(по умолчанию HDIMAGE)

-storepath

путь к хранилищу доверенных сертификатов

(по умолчанию null)

-storepass

пароль на хранилище доверенных сертификатов

(по умолчанию null)

-certs

пути к сертификатам

12.3.4.Формирование электронной подписи.

Формирование электронной подписи осуществляется в соответствии с алгоритмом ГОСТ Р 34.10-2001.

```
Signature -alias name_of_key [-storetype HDIMAGE] [-storepath null] [-storepass null] [-keypass password] -signpath C:/*.* -filepath C:/*.*
```

-alias

уникальное имя ключа

-keypass

пароль на записываемый ключ

(по умолчанию null)

-storetype

имя ключевого носителя HDIMAGE (жесткий диск), REGISTRY (реестр ОС Windows), имя карточки или токена

(по умолчанию HDIMAGE)

-storepath

путь к хранилищу доверенных сертификатов

(по умолчанию null)

-storepass

пароль на хранилище доверенных сертификатов

(по умолчанию null)

-signpath

путь к файлу подписи

-filepath

путь к подписываемому файлу

12.3.5.Проверка электронной подписи.

Проверка электронной подписи осуществляется в соответствии с алгоритмами ГОСТ Р ГОСТ Р 34.10-2001.

```
SignatureVerif -alias name_of_key [-storetype HDIMAGE] [-storepath null] [-storepass null] -signpath C:/*.* -filepath C:/*.*
```

-alias

уникальное имя ключа

-keypass

пароль на записываемый ключ

(по умолчанию null)

-storetype

имя ключевого носителя HDIMAGE (жесткий диск), REGISTRY (реестр ОС Windows), имя карточки или токена

(по умолчанию HDIMAGE)

-storepath

путь к хранилищу доверенных сертификатов

(по умолчанию null)

-storepass

пароль на хранилище доверенных сертификатов

(по умолчанию null)

-signpath

путь к файлу подписи

-filepath

путь к проверяемому файлу

12.4.Использование КриптоПро JTLS

12.4.1.Запуск сервера из командной строки.

```
Server [-port port] [-auth true] [-keyStoreType HDIMAGE] [-trustStoreType HDImageStore] -trustStorePath C:/*.* -trustStorePassword trust_pass -keyStorePassword key_pass
```

-port

порт сервера

(по умолчанию 443)

-auth

нужна ли аутентификация клиента

(по умолчанию false)

-keyStoreType

имя ключевого носителя HDIMAGE (жесткий диск), REGISTRY (реестр ОС Windows), имя карточки или токена

(по умолчанию HDIMAGE)

-trustStoreType

тип носителя для хранилища доверенных сертификатов HDImageStore (жесткий диск), FloppyStore (дискета)

(по умолчанию HDImageStore)

-trustStorePath

путь к хранилищу доверенных сертификатов

-trustStorePassword

пароль на хранилище доверенных сертификатов

-keyStorePassword

пароль на ключ

-servDir

рабочая директория сервера

(по умолчанию текущая)

При запросе ресурса shutdown сервер останавливается, предварительно послав клиенту ответ, который содержит сообщение об остановке сервера по окончании сессии.

12.4.2.Запуск клиента из командной строки.

```
Client [-port port] [-server serverName] [-keyStoreType HDIMAGE] [-trustStoreType HDIMAGE] -trustStorePath C:/*.* -trustStorePassword trust_pass -keyStorePassword key_pass [-fileget gettingFileName] [-fileout outputFilePath]
```

-port

порт сервера

(по умолчанию 443)

-server

имя сервера

(по умолчанию localhost)

-keyStoreType

имя ключевого носителя HDIMAGE (жесткий диск), REGISTRY (реестр ОС Windows), имя карточки или токена

(по умолчанию HDIMAGE)

-trustStoreType

имя ключевого носителя HDIMAGE (жесткий диск), REGISTRY (реестр ОС Windows), имя карточки или токена

(по умолчанию HDIMAGE)

-trustStorePath

путь к хранилищу доверенных сертификатов

-trustStorePassword

пароль на хранилище доверенных сертификатов

-keyStorePassword

пароль на ключ

-fileget

имя ресурса

(по умолчанию index.html)

-fileout

путь к файлу вывода

(по умолчанию out.html)

12.4.3.Запуск клиента нагрузочного примера из командной строки (samples.jar/JTLS_samples/HighLoadExample).

```
JTLS_samples.HighLoadExample -client [-port hostPort] [-host hostName] [-get sourcePage] [-t T] [-n N] -source sourceDir -store tempDir -trustStorePath C:/*.* [-trustStoreType trust_type] -trustStorePassword trust_pass [-keyStoreType keystoreType] [-keyStorePassword key_pass] [-ct X] [-external] [-apache4] [-trace] [-help]
```

При выполнении команды, возможно, потребуется указать параметры **-Dcom.sun.security.enableCRLDP=true -Dcom.ibm.security.enableCRLDP=true** для осуществления проверки цепочки сертификатов online.

-port

порт сервера

(по умолчанию 443)

-host

имя сервера

(по умолчанию 127.0.0.1)

-get

имя загружаемого ресурса

(по умолчанию default.htm)

-t

количество потоков (подключений)

(по умолчанию 2)

-n

количество запросов на поток (подключение)

(по умолчанию 2)

-source

папка с ресурсами для передачи сервером клиенту (пока не используется)

-store

папка для сохранения загружаемого ресурса

-trustStorePath

путь к хранилищу доверенных сертификатов

-trustStoreType

имя ключевого носителя HDIMAGE (жесткий диск), REGISTRY (реестр ОС Windows), имя карточки или токена

(по умолчанию HDIMAGE)

-trustStorePassword

пароль на хранилище доверенных сертификатов

-keyStoreType

имя ключевого носителя HDIMAGE (жесткий диск), REGISTRY (реестр ОС Windows), имя карточки или токена

(по умолчанию HDIMAGE)

-keyStorePassword

пароль на ключ

-ct

таймаут работы потока клиента (сек.)

(по умолчанию 5 мин.)

-external

означает подключение к "внешнему" (не созданному в этом же примере) серверу

-apache4

означает использование apache http client 4.x вместо внутреннего класса Client. Библиотеки apache должны быть в каталоге lib/ext

-trace

означает подробный вывод в консоль

-help

информация о том, какие команды можно использовать

12.4.4.3апуск клиента на основе apache http client 4.x из командной строки (samples.jar/JTLS_samples/ApacheHttpClient4XExample).

```
JTLS_samples.ApacheHttpClient4XExample [-port hostPort] [-host hostName] [-get sourcePage] [-allow] [-auth] [-save path] -trustStorePath C:/*.* [-trustStoreType trust_type] -trustStorePassword trust_pass [-keyStoreType keystoreType] [-keyStorePassword key_pass] [-help]
```

При выполнении команды, возможно, потребуется указать параметры **-Dcom.sun.security.enableCRLDP=true** **-Dcom.ibm.security.enableCRLDP=true** для осуществления проверки цепочки сертификатов online.

-port

порт сервера

(по умолчанию 443)

-host

имя сервера

(по умолчанию 127.0.0.1)

-get

имя загружаемого ресурса

(по умолчанию default.htm)

-save

полный путь для сохранения загруженного ресурса

-allow

для отключения проверки соответствия адреса ресурса и CN серверного сертификата

-auth

указывает на необходимость клиентской аутентификации

-trustStorePath

путь к хранилищу доверенных сертификатов

-trustStoreType

имя ключевого носителя HDIMAGE (жесткий диск), REGISTRY (реестр ОС Windows), имя карточки или токена

(по умолчанию HDIMAGE)

-trustStorePassword

пароль на хранилище доверенных сертификатов

-keyStoreType

имя ключевого носителя HDIMAGE (жесткий диск), REGISTRY (реестр ОС Windows), имя карточки или токена

(по умолчанию HDIMAGE)

-keyStorePassword

пароль на ключ

-help

информация о том, какие команды можно использовать