ГОСУДАРСТВЕННЫЙ СТАНДАРТ
РЕСПУБЛИКИ БЕЛАРУСЬ

СТБ 34.101.65-2014

Информационные технологии и безопасность
ПРОТОКОЛ ЗАЩИТЫ ТРАНСПОРТНОГО УРОВНЯ (TLS)

Інфармацыйныя тэхналогіі і бяспека
ПРАТАКОЛ АХОВЫ ТРАНСПАРТНАГА ЎЗРОЎНЮ (TLS)

Госстандарт
Минск

УДК 004.056.5.057.4(083.74)(476) МКС 35.240.01 КП 05

Ключевые слова: транспортный уровень, криптографический протокол, шифрование, контроль целостности, аутентификация

Предисловие

Цели, основные принципы, положения по государственному регулированию и управлению в области технического нормирования и стандартизации установлены Законом Республики Беларусь «О техническом нормировании и стандартизации».

1 РАЗРАБОТАН учреждением Белорусского государственного университета «Научно-исследовательский институт прикладных проблем математики и информатики» (НИИ ППМИ)

ВНЕСЕН Оперативно-аналитическим центром при Президенте Республики Беларусь

2 УТВЕРЖДЕН И ВВЕДЕН В ДЕЙСТВИЕ постановлением Госстандарта Республики Беларусь от 22 мая 2014 г. № 23

3 Настоящий стандарт разработан на основе документа RFC 5246:2008 The Transport Layer Security (TLS) Protocol. Version 1.2 (Протокол защиты транспортного уровня. Версия 1.2). Документ RFC 5246 разработан Специальной комиссией интернет-разработок (Internet Engineering Task Force)

4 ВВЕДЕН ВПЕРВЫЕ

Издан на русском языке

Содержание

ГОСУДАРСТВЕННЫЙ СТАНДАРТ РЕСПУБЛИКИ БЕЛАРУСЬ

Информационные технологии и безопасность ПРОТОКОЛ ЗАЩИТЫ ТРАНСПОРТНОГО УРОВНЯ (TLS)

Інфармацыйныя тэхналогіі і бяспека ПРАТАКОЛ АХОВЫ ТРАНСПАРТНАГА ЎЗРОЎНЮ (TLS)

Information technology and security
The Transport Layer Security Protocol

Дата введения 2014-09-01

1 Область применения

Настоящий стандарт устанавливает протокол защиты транспортного (коммуникационного) уровня, известный как протокол TLS (Transport Layer Security) версии 1.2. Стандарт определяет действия сторон протокола и форматы сообщений, которыми стороны обмениваются.

Настоящий стандарт применяется при разработке средств криптографической защиты информации, передаваемой в сети Интернет.

2 Нормативные ссылки

В настоящем стандарте использованы ссылки на следующие технические нормативные правовые акты в области технического нормирования и стандартизации (далее — ТНПА):

СТБ 34.101.19-2012 Информационная технологии и безопасность. Форматы сертификатов и списков отозванных сертификатов инфраструктуры открытых ключей

СТБ 34.101.31-2011 Информационные технологии. Защита информации. Криптографические алгоритмы шифрования и контроля целостности

СТБ 34.101.45-2013 Информационные технологии и безопасность. Алгоритмы электронной цифровой подписи и транспорта ключа на основе эллиптических кривых

СТБ 34.101.47-2012 Информационные технологии и безопасность. Криптографические алгоритмы генерации псевдослучайных чисел

ГОСТ 34.973-91 (ИСО 8824-87) Информационная технология. Взаимосвязь открытых систем. Спецификация абстрактно-синтаксической нотации версии 1 (АСН.1)

ГОСТ 34.974-91 (ИСО 8825-87) Информационная технология. Взаимосвязь открытых систем. Описание базовых правил кодирования для абстрактно-синтаксической нотации версии 1 (АСН.1)

ГОСТ 27463-87 Системы обработки информации. 7-битные кодированные наборы символов

Примечание — При пользовании настоящим стандартом целесообразно проверить действие ТНПА по каталогу, составленному по состоянию на 1 января текущего года, и по соответствующим информационным указателям, опубликованным в текущем году.

Если ссылочные ТНПА заменены (изменены), то при пользовании настоящим стандартом следует руководствоваться замененными (измененными) ТНПА. Если ссылочные ТНПА отменены без замены, то положение, в котором дана ссылка на них, применяется в части, не затрагивающей эту ссылку.

3 Термины и определения

В настоящем стандарте применяют термины, установленные в СТБ 34.101.19, СТБ 34.101.31, СТБ 34.101.45 и СТБ 34.101.47, а также следующие термины с соответствующими определениями:

3.1 алгоритм сжатия: Алгоритм, который выполняет обратимое сжатие данных для уменьшения их размера при передаче по каналам связи; в настоящем стандарте алгоритм сжатия используется всегда вместе с алгоритмом восстановления сжатых данных.

3.2 аутентификация: Проверка подлинности стороны.

3.3 клиент (client): Сторона, которая инициирует выполнение протокола TLS.

3.4 ключ имитозащиты клиента (client write MAC key): Криптографический ключ, используемый для имитозащиты данных, отправляемых клиентом.

3.5 ключ имитозащиты сервера (server write MAC key): Криптографический ключ, используемый для имитозащиты данных, отправляемых сервером.

3.6 ключ шифрования клиента (client write encryption key): Криптографический ключ, используемый для шифрования данных, отправляемых клиентом.

3.7 ключ шифрования сервера (server write encryption key): Криптографический ключ, используемый для шифрования данных, отправляемых сервером.

3.8 криптонабор (cipher suite): Криптоопределение, дополненное алгоритмом формирования общего ключа, который используется в протоколе Handshake для построения предварительного мастер-ключа.

3.9 криптоопределение (cipher spec): Точно определенный перечень алгоритмов шифрования, имитозащиты, генерации псевдослучайных чисел, которые используются в протоколе Record для обеспечения конфиденциальности и контроля целостности данных.

3.10 мастер-ключ (master secret): Криптографический ключ, который включается в параметры защиты и используется для построения ключей соединений.

3.11 неявная часть синхропосылки (implicit nonce): Часть синхропосылки, которая не передается вместе с обработанными на ней данными, а определяется по контексту, в котором обработка выполнялась (например, по состоянию соединения).

3.12 параметры защиты (security parameters): Параметры, которые являются частью состояния сеанса и используются для формирования состояний соединения; создаются при согласовании параметров связи с использованием согласованного алгоритма формирования общего ключа и с применением согласованных методов аутентификации.

3.13 переустановка связи (renegotiate): Согласование параметров связи, которое выполняется во время соединения и заканчивается привязкой к созданному ранее сеансу (возможно текущему), формированием по параметрам защиты этого сеанса новых состояний соединения и переходом к этим состояниям.

3.14 предварительный мастер-ключ (preliminary master secret): Криптографический ключ, который формируется при согласовании параметров связи и используется для построения мастер-ключа.

3.15 прикладной протокол (application protocol): Протокол, который выполняется поверх протокола TLS.

3.16 протокол защиты транспортного уровня; протокол TLS (transport layer security protocol): Определеяемый в настоящем стандарте криптографический протокол, который обеспечивает взаимную аутентификацию сторон протокола, конфиденциальность и контроль целостности данных, передаваемых между сторонами на транспортном коммуникационном уровне.

3.17 протокол Alert: Протокол обмена сигнальными сообщениями; часть протокола TLS.

3.18 протокол Change Cipher Spec: Протокол оповещения о переходе к новым состояниям соединения после согласования параметров связи; часть протокола TLS.

3.19 протокол Handshake: Протокол установки, возобновления или переустановки связи; часть протокола TLS.

3.20 протокол Record: Протокол, обеспечивающий обратимое сжатие, конфиденциальность и контроль целостности данных, передаваемых на транспортном коммуникационном уровне; часть протокола TLS; используется протоколами Handshake, Change Cipher Spec и Alert.

3.21 сеанс (session): Логическая связь между клиентом и сервером, которая описывается идентификатором, параметрами защиты и другими согласованными между сторонами данными, которые могут быть использованы в нескольких соединениях.

3.22 сервер (server): Сторона, которая выполняет протокол TLS с клиентами по их запросам.

3.23 сигнальное сообщение (alert message): Сообщение о закрытии соединения или сообщение о внештатной ситуации во время выполнения протокола TLS.

3.24 согласование параметров связи (negotiate): Согласование между клиентом и сервером алгоритма сжатия, криптонаборов, метода аутентификации, параметров защиты; часть протокола Handshake.

3.25 соединение (connection): Связь между сторонами на транспортном коммуникационном уровне.

3.26 состояние соединения (connection state): Набор параметров, определяющих способ обработки принимаемых или отправляемых в рамках соединения данных, в том числе порядковый номер обрабатываемого фрагмента данных, состояния алгоритмов криптоопределения, состояние алгоритма сжатия; состояние соединения формируется по параметрам защиты сеанса.

3.27 сокращенная установка связи; возобновление связи (abbreviated handshake; session resume): Согласование параметров связи, которое выполняется в начале соединения и заканчивается привязкой к созданному ранее сеансу, формированием по параметрам защиты этого сеанса новых состояний соединения и переходом к этим состояниям.

3.28 установка связи (handshake; full handshake): Согласование параметров связи, которое выполняется в начале соединения и заканчивается созданием нового сеанса, формированием по параметрам защиты этого сеанса новых состояний соединения и переходом к этим состояниям.

3.29 фрагмент (record): Порция данных, отправляемых или принимаемых на транспортном коммуникационном уровне.

3.30 эфемерный ключ (ephemeral key): Криптографический ключ, который генерируется, используется и уничтожается при установке связи.

3.31 явная часть синхропосылки (explicit nonce): Часть синхропосылки, которая передается вместе с обработанными на ней данными.

4 Сокращения, соглашения и обозначения

В настоящем стандарте применяют следующее сокращение:

ЭЦП — электронная цифровая подпись.

В настоящем стандарте ключевые слова «ДОЛЖЕН», «НЕЛЬЗЯ», «СЛЕДУЕТ», «НЕ СЛЕДУЕТ», «РЕКОМЕНДУЕТСЯ» и «МОЖЕТ», выделенные прописными буквами, должны интерпретироваться, как описано в приложении А.

В настоящем стандарте для описания форматов данных используются соглашения и обозначения, определенные в приложении Б. Применяемый синтаксис основан на языке программирования Си [1] и правилах, заданных в [2].

5 Общие положения

Настоящий стандарт определяет криптографический протокол, предназначенный для защиты соединений между клиентом и сервером в сети Интернет. Данный протокол соответствует спецификации [3], ее расширениям [4], [5] и, следуя этим документам, обозначается TLS. Действия сторон протокола и форматы пересылаемых между сторонами сообщений определяются с такой степенью детализации, которая позволяет разрабатывать полностью совместимые между собой реализации TLS.

TLS обеспечивает аутентификацию сторон протокола, конфиденциальность и контроль целостности передаваемых между сторонами данных. TLS встраивается в стек коммуникационных протоколов поверх транспортного уровня и обеспечивает защиту данных этого уровня. TLS выполняется независимо от протоколов верхнего уровня и прозрачен для них.

Для организации защиты используются криптографические алгоритмы, которые оформляются в виде криптонаборов. В TLS предусмотрена возможность расширения перечня криптонаборов. Расширяемость TLS избавляет от необходимости создавать и реализовывать новый протокол для каждого нового набора криптографических алгоритмов. В приложении В определены криптонаборы, основанные на криптографических алгоритмах действующих ТНПА.

В TLS предусмотрена возможность сохранения состояний сеансов, что позволяет восстанавливать связь между сторонами, а не устанавливать ее каждый раз заново. Возможность сохранения состояний сеансов снижает нагрузку на сервер и уменьшает объем пересылаемых между сторонами данных.

TLS является объединением нескольких субпротоколов, разбитых на два уровня. На нижнем уровне действует протокол Record, который обеспечивает защищенный транспорт данных, поступающих от прикладных протоколов. На верхнем уровне действуют протоколы Handshake, Change Cipher Spec, Alert и прикладные протоколы.

Протокол Record обеспечивает конфиденциальность и контроль целостности транспортируемых данных. Для обеспечения конфиденциальности используются симметичные алгоритмы шифрования, а для контроля целостности — алгоритмы имитозащиты. Для каждого соединения стороны вырабатывают уникальные общие ключи шифрования и имитозащиты. Ключи строятся по секретным данным, согласованным с помощью других протоколов (как правило, Handshake). Протокол Record может выполняться без шифрования и имитозащиты. Однако режимы, обеспечивающие и конфиденциальность, и контроль целостности, являются основными.

Протокол Handshake позволяет клиенту и серверу аутентифицировать друг друга, а также согласовать криптографические алгоритмы и общие ключи до того, как прикладной протокол начнет прием или передачу данных. Аутентификация сторон, как правило, производится с помощью асимметричных алгоритмов (алгоритмов с открытым ключом). Аутентификация может быть необязательной, но, как правило, хотя бы одна сторона проверяет подлинность другой. Общие ключи согласуются так, чтобы их не мог определить злоумышленник, который перехватывает все сообщения протокола. Более того, в режи-

мах с аутентификацией общие ключи недоступны злоумышленнику, даже если он выдает себя за одну из сторон протокола. Протокол Handshake строится так, что всякое изменение пересылаемых между сторонами данных будет выявлено этими сторонами.

Протокол Change Cipher Spec сообщает о смене параметров защиты на новые, согласованные при выполнении протокола Handshake.

Протокол Alert извещает о закрытии соединений и об ошибках, произошедших при выполнении TLS.

6 Криптонаборы и методы аутентификации

6.1 Криптонаборы

Криптонабор определяет:

– алгоритмы шифрования (зашифрования и расшифрования), которые используются для обеспечения конфиденциальности данных, передаваемых между клиентом и сервером;

– алгоритм имитозащиты, который используется для контроля целостности данных;

– алгоритм генерации псевдослучайных чисел, который используется для генерации ключей и синхропосылок по мастер-ключу, а также для верификации сообщений;

– алгоритм формирования общего ключа, который используется для согласования или передачи предварительного мастер-ключа с последующим построением мастер-ключа.

Включенные в криптонабор алгоритмы шифрования, имитозащиты и генерации псевдослучайных чисел в совокупности составляют криптоопределение. Каждому криптонабору назначается уникальный идентификатор типа `CipherSuite`:

```
uint8 CipherSuite[2];
```

Идентификатор `{0, 0}` зарезервирован для криптонабора `TLS_NULL_WITH_NULL_NULL`:

```
CipherSuite TLS_NULL_WITH_NULL_NULL = {0, 0};
```

В этом криптонаборе все алгоритмы являются «пустыми» (обозначаются `null`), т. е. не выполняют никаких вычислений. Для «пустых» алгоритмов длины ключей, синхропосылок и имитовставок полагаются равными 0.

6.2 Алгоритмы криптонаборов

6.2.1 Алгоритмы шифрования

Для шифрования могут использоваться алгоритмы поточного шифрования, алгоритмы блочного шифрования или алгоритмы одновременного шифрования и имитозащиты. Тип алгоритма задается с помощью перечисления

```
enum {stream, block, aead} CipherType;
```

Элемент `stream` этого перечисления указывает на поточное шифрование, элемент `block` — на блочное, элемент `aead` — на одновременное шифрование и имитозащиту.

При поточном зашифровании открытый текст побитово суммируется с гаммой (двоичной последовательностью), полученной с помощью ключезависимого генератора псевдослучайных чисел. Гамма может вырабатываться для каждого отдельного фрагмента, передаваемого в рамках соединения, или для всех фрагментов сразу. В первом случае при генерации гаммы должна использоваться синхропосылка, уникальная для обрабатываемого фрагмента (например, номер фрагмента). Во втором случае состояние генератора гаммы должно сохраняться после зашифрования очередного фрагмента и использоваться при зашифровании следующего.

При блочном зашифровании каждый блок открытого текста преобразуется в блок шифртекста. Шифрование выполняется в режиме сцепления блоков. Перед зашифрованием открытый текст дополняется незначащими байтами, чтобы получить строку байтов, длина которой кратна длине блока алгоритма. При зашифровании используется синхропосылка, которая передается вместе с шифртекстом.

Алгоритмы одновременного шифрования и имитозащиты — это алгоритм установки защиты и алгоритм снятия защиты. Алгоритм установки защиты берет на вход ключ, синхропосылку, критические данные, для которых будет обеспечиваться шифрование и имитозащита, и открытые данные, для которых будет обеспечиваться только имитозащита. Алгоритм установки защиты возвращает зашифрованные критические данные и имитовставку открытых и критических данных. Алгоритм снятия защиты берет на вход ключ, синхропосылку, имитовставку, зашифрованные критические данные и открытые данные. Алгоритм снятия защиты либо расшифровывает критические данные, либо возвращает признак нарушения целостности данных.

Допустимые алгоритмы шифрования задаются типом `BulkCipherAlgorithm`:

```
enum {null,...} BulkCipherAlgorithm;
```

Элемент `null` соответствует «пустому» алгоритму шифрования, который не изменяет поступающие на его вход данные. Алгоритм `null` классифицируется как алгоритм поточного шифрования.

6.2.2 Алгоритмы имитозащиты

Допустимые алгоритмы имитозащиты задаются типом `MACAlgorithm`:

```
enum {null,...} MACAlgorithm;
```

Элемент `null` соответствует «пустому» алгоритму имитозащиты, который не вычисляет имитовставку.

Если в качестве алгоритмов шифрования выбраны алгоритмы типа `aead`, то в качестве алгоритма имитозащиты должен быть выбран `null`.

6.2.3 Алгоритмы генерации псевдослучайных чисел

Алгоритм генерации псевдослучайных чисел принимает на вход три параметра, которые обозначаются `secret`, `label` и `seed`, и возвращает строку байтов требуемой длины, которая обозначается `PRF(secret, label, seed)`. Входные параметры `secret`, `label` и `seed` — это строки байтов произвольной длины, причем `label` можно задать строкой символов (см. Б.2, приложение Б).

Вычисление `PRF(secret, label, seed)` состоит в обращении к алгоритму генерации псевдослучвйных чисел в режиме HMAC, определенному в СТБ 34.101.47 (подраздел 6.3), с передачей `secret` в качестве ключа и `label + seed` в качестве синхропосылки. Используемый алгоритм СТБ 34.101.47 строится на основе алгоритма HMAC, который, в свою очередь, строится на некотором алгоритме хэширования.

Могут использоваться различные базовые алгоритмы хэширования. Используемый алгоритм ДОЛЖЕН однозначно определяться в криптонаборе. СЛЕДУЕТ использовать криптографически стойкие алгоритмы.

Выходные данные `PRF(secret, label, seed)` формируются блоками по n байтов, где n — длина хэш-значения (определяется используемым алгоритмом хэширования). Для генерации m байтов следует сформировать k блоков, где k — минимальное целое такое, что $nk \geqslant m$. При необходимости в последнем блоке следует отбросить последние байты.

Допустимые алгоритмы генерации псевдослучайных чисел задаются типом `PRFAlgorithm`:

```
enum {...} PRFAlgorithm;
```

6.2.4 Алгоритмы формирования общего ключа

Алгоритм формирования общего ключа является интерактивным. Это значит, что его выполняют совместно клиент и сервер, обмениваясь между собой сообщениями, которые содержат промежуточные результаты вычислений. Сообщения алгоритма — это сообщения протокола Handshake, описанные в 8.3. По завершении алгоритма стороны формируют предварительный мастер-ключ, известный только им. По этому ключу и случайным данным, вырабатываемым каждой из сторон, строится окончательный мастер-ключ.

В криптонаборе, вообще говоря, можно определить любой алгоритм формирования общего ключа. Тем не менее в основной спецификации [3] и ее расширении [4] описаны семь типов таких алгоритмов, которые покрывают большинство существующих на сегодняшний день решений. Для описания этих типов используются обозначения, близкие к обозначениям СТБ 34.101.45: G — элемент аддитивной (алгебраической) группы, который порождает циклическую группу $\langle G\rangle$ порядка q; $d_S, d_C \in \{1, 2, \ldots, q-1\}$ — личные ключи сервера и клиента соответственно; $Q_S = d_S G$, $Q_C = d_C G$ — открытые ключи сторон.

DH_anon (протокол Диффи – Хеллмана без аутентификации сторон). Сервер в сообщении `ServerKeyExchange` пересылает клиенту описание группы $\langle G\rangle$ и свой открытый ключ Q_S. Клиент в сообщении `ClientKeyExchange` пересылает серверу свой открытый ключ Q_C. Стороны вычисляют общий ключ $d_S d_C G = d_S Q_C = d_C Q_S$, по которому строится предварительный мастер-ключ. Сертификаты не используются.

DH_fixed (протокол Диффи – Хеллмана со статическим ключом). Сервер в сообщении `Certificate` передает свой сертификат, который содержит описание $\langle G\rangle$ и открытый ключ Q_S. Сообщение `ServerKeyExchange` не передается. Клиент в сообщении `Certificate` (по запросу сервера) передает в своем сертификате статический (неизменяемый) открытый ключ Q_C. Если запроса от сервера нет, то клиент передает эфемерный (одноразовый) открытый ключ Q_C в сообщении `ClientKeyExchange`. Стороны вычисляют общий ключ $d_S d_C G$, по которому определяется предварительный мастер-ключ.

DHE (протокол Диффи – Хеллмана с эфемерными ключами). Сервер в сообщении `Certificate` передает клиенту сертификат, открытый ключ которого можно использовать для проверки ЭЦП. Затем сервер в сообщении `ServerKeyExchange` передает описание группы $\langle G\rangle$, свой эфемерный открытый ключ Q_S и подписывает эти данные, а также случайные данные обеих сторон, на личном ключе, который соответствует переданному сертификату. Клиент проверяет ЭЦП и в сообщении `ClientKeyExchange` передает свой открытый ключ Q_C. Стороны вычисляют общий ключ $d_S d_C G = d_S Q_C = d_C Q_S$, по которому строится предварительный мастер-ключ.

T (транспорт). Сервер в сообщении `Certificate` передает свой сертификат, открытый ключ которого можно использовать для шифрования. Клиент выполняет на этом ключе зашифрование предварительного мастер-ключа и передает зашифрованный ключ в сообщении `ClientKeyExchange`. Сервер выполняет расшифрование на своем личном ключе. Сообщение `ServerKeyExchange` не передается.

PSK (на основе предварительного распределения секретов, от английского pre-shared key). Клиент и сервер предварительно распределяют между собой набор общих секретов. Клиент выбирает секрет из набора и в сообщении `ClientKeyExchange` передает идентификатор выбранного секрета. Для помощи при выборе секрета сервер в `ServerKeyExchange`

может передать подсказку (например, номер секрета). Если подсказка не нужна, то сообщение `ServerKeyExchange` не передается. Предварительный мастер-ключ строится по выбранному секрету. Сертификаты не используются.

DHE_PSK (совмещение PSK и DHE). Сервер в сообщении `ServerKeyExchange` передает описание группы $\langle G \rangle$ и свой эфемерный открытый ключ Q_S. Дополнительно сервер может передать в `ServerKeyExchange` PSK-подсказку. Клиент в `ClientKeyExchange` передает свой эфемерный открытый ключ Q и идентификатор выбранного PSK-секрета. Стороны определяют общий ключ Диффи – Хеллмана $d_S d_C G = d_S Q_C = d_C Q_S$ и общий PSK-секрет. Предварительный мастер-ключ является результатом конкатенации этих общих секретных данных. Сертификаты не используются.

T_PSK (совмещение PSK и транспорта). Дополнительно к сообщениям механизма PSK, сервер в сообщении `Certificate` посылает клиенту свой сертификат, открытый ключ которого можно использовать для шифрования. Клиент выполняет на этом ключе зашифрование предварительного мастер-ключа и передает зашифрованный ключ в сообщении `ClientKeyExchange` вместе с идентификатором выбранного PSK-секрета. Сервер выполняет расшифрование на своем личном ключе и объединяет полученный предварительный мастер-ключ с общим PSK-секретом.

Примечание 1 — Описание группы $\langle G \rangle$ может задаваться явно или косвенно. Явное описание задается набором параметров, описывающих структуру группы, ее порядок, правила представления элементов и др. Клиент ДОЛЖЕН проверять корректность присланного ему явного описания группы $\langle G \rangle$. Косвенное описание задается ссылкой на фиксированные параметры, известные клиенту и серверу, например, на параметры из ТНПА или из сертификата сервера.

Примечание 2 — В алгоритмах типа DH_anon не используются ни сертификаты, ни общие секретные данные и, таким образом, не проверяется подлинность сторон. Поэтому алгоритмы типа DH_anon не обеспечивают защиту от атак «противник посередине», и их рекомендуется использовать только в специальных случаях.

Примечание 3 — В алгоритмах типа DH_fixed, T при компрометации личного ключа сервера все сообщения предыдущих соединений TLS могут быть раскрыты. Данные алгоритмы не обеспечивают защиту от атак по «чтению назад» и их рекомендуется использовать только в специальных случаях. Защиту от «чтения назад» не обеспечивают также алгоритмы типа PSK (при компрометации PSK-секрета) и T_PSK (при компрометации PSK-секрета и личного ключа сервера).

Примечание 4 — Алгоритмы типа PSK не обеспечивают защиту от словарных атак по подбору PSK-секрета со стороны злоумышленника, который перехватывает все сообщения протокола. Алгоритмы типа DHE_PSK, T_PSK защищают от таких атак, но не обеспечивают защиту от словарных атак злоумышленника, который выдает себя за сервер (DHE_PSK) или является таковым (T_PSK) и пытается узнать PSK-секрет у клиента.

Примечание 5 — Если клиент C взаимодействует с сервером A, контролируемым злоумышленником, и использует алгоритм формирования общего ключа типа T, то злоумышленник может провести атаку, описанную в [6]. Злоумышленник может организовать защищенное соединение между C и другим сервером S, при котором и S будут считать, что взаимодействуют с A и не смогут обнаружить, что взаимодействуют между собой. СЛЕДУЕТ учитывать возможность данной атаки при использовании алгоритмов типа T.

6.3 Методы аутентификации

6.3.1 Аутентификация сервера

Аутентификация сервера основана, как правило, на проверке сертификата открытого ключа сервера и на проверке владения сервером соответствующим личным ключом. Успешное завершение протокола Handshake означает, что аутентификация завершена успешно: сертификат сервера действителен и сервер действительно владеет личным ключом. В алгоритмах формирования общего ключа на основе предварительно распределенных секретов проводится неявная аутентификация сервера, основанная на проверке владения PSK-секретом. Успешное завершение протокола Handshake означает, что сервер действительно владеет этим секретом. Если секрет распределялся по защищенным каналам только клиенту и серверу, то владение секретом доказывает клиенту подлинность сервера.

6.3.2 Аутентификация клиента

Для аутентификации клиента сервер, как правило, запрашивает его сертификат в сообщении `CertificateRequest`. Сертификат должен содержать открытый ключ определенного алгоритма ЭЦП. Сервер указывает в `CertificateRequest` список подходящих типов открытых ключей. Данные типы задаются одним байтом и называются методами аутентификации клиента. Допустимые методы аутентификации клиента задаются типом `ClientCertificateType`:

```
enum {..., (255)} ClientCertificateType;
```

В ответ на запрос сервера клиент представляет сертификат одного из запрашиваемых типов и подписывает на личном ключе, соответствующем открытому ключу сертификата, определенные данные. При этом используется пара «алгоритм хэширования, алгоритмы ЭЦП». Данная пара используется также в некоторых алгоритмах формирования общего ключа. В алгоритмах формирования общего ключа на основе предварительно распределенных секретов проводится неявная аутентификация клиента. При соблюдении мер защиты PSK-секрета успешное завершение Handshake означает, что клиент действительно знает этот секрет, т. е. является подлинным.

6.3.3 Алгоритмы хэширования и электронной цифровой подписи

Пара «алгоритм хэширования, алгоритмы ЭЦП» задается следующим типом:

```
struct {
  HashAlgorithm hash;
  SignatureAlgorithm signature;
} SignatureAndHashAlgorithm;
```

Поле `hash` определяет алгоритм хэширования и описывается следующим типом:

```
enum {none(0),..., (255)} HashAlgorithm;
```

Элемент `none` означает, что алгоритмы ЭЦП не требуют хэширования данных перед выработкой или проверкой подписи.

Поле `signature` определяет алгоритмы ЭЦП и описывается следующим типом:

```
enum {anonymous(0),..., (255)} SignatureAlgorithm;
```

Элемент `anonymous` означает, что подпись не вырабатывается.

7 Протокол Record

7.1 Общие сведения

Протокол Record получает данные для передачи, разбивает их на фрагменты, при необходимости сжимает, вычисляет имитовставку, зашифровывает и передает полученный результат. После получения данные расшифровываются, проверяется их целостность, при необходимости данные преобразуются из сжатой формы в исходную, объединяются и доставляются протоколам верхних уровней.

Протокол Record используется протоколами Handshake, Change Cipher Spec и Alert, которые описываются в следующих разделах, а также прикладными протоколами. Каждый фрагмент данных при передаче протоколом Record дополняется полями, которые указывают на тип содержимого и длину фрагмента.

Тип содержимого является элементом перечисления

```
enum {
  change_cipher_spec(20), alert(21), handshake(22),
  application_data(23), (255)
} ContentType;
```

Значения `change_cipher_spec`, `alert`, `handshake` и `application_data` указывают на то, что данные отправлены протоколами Change Cipher Spec, Alert, Handshake и прикладным протоколом соответственно.

Перечисление `ContentType` может быть расширено в будущем. Но пока реализации TLS НЕ ДОЛЖНЫ посылать данные, типы содержимого которых отличны от определенных выше. Если сторона TLS получает сообщение с содержимым неопределенного типа, то она ДОЛЖНА выслать критическое сигнальное сообщение `unexpected_message` (см. 10.3).

Любой протокол, предназначенный для использования поверх TLS, должен быть тщательно продуман на предмет защиты от возможных атак. Это значит, что разработчик протокола верхнего уровня должен быть осведомлен о том, какие механизмы безопасности поддерживает протокол TLS, а какие нет, и не полагаться на последние. В частности, разработчик должен учитывать, что в TLS тип содержимого и длина фрагмента не защищаются шифрованием. Поэтому, если эти данные являются критическими, разработчику следует принять меры для минимизации утечки информации при передаче (например, следует дополнять фрагменты неиспользуемыми байтами или маскировать трафик).

7.2 Состояния соединения

Логически соединение TLS описывается четырьмя состояниями: имеются текущие (активные) состояния чтения и записи, а также ожидаемые состояния чтения и записи. Состояние — это структура данных, которая определяет используемые алгоритмы сжатия, шифрования и имитозащиты, а также состояния и параметры этих алгоритмов. В том числе состояние определяет ключи шифрования и имитозащиты, которые используются или будут использоваться для защиты соединения в одном из направлений «чтения или записи».

Протокол Record обрабатывает отправляемые данные с использованием текущего состояния записи, а принимаемые — с использованием текущего состояния чтения. Начальные текущие состояния устанавливаются пустыми. В этих состояниях шифрование,

сжатие и имитозащита не используются. Пустое состояние соответствует криптонабору TLS_NULL_WITH_NULL_NULL.

Состояния чтения и записи разделяют общие параметры защиты. Параметры защиты для ожидаемых состояний устанавливаются протоколом Handshake. Протокол Change Cipher Spec может переводить ожидаемое состояние в текущее с переносом параметров защиты, при этом ранее установленное текущее состояние заменяется ожидаемым, а ожидаемое состояние сбрасывается в пустое. Если для состояния не установлены параметры защиты, то его нельзя сделать текущим.

Параметры защиты описываются следующей структурой:

```
struct {
  ConnectionEnd           entity;
  PRFAlgorithm            prf_algorithm;
  BulkCipherAlgorithm     bulk_cipher_algorithm;
  CipherType              cipher_type;
  uint8                   enc_key_length;
  uint8                   block_length;
  uint8                   fixed_iv_length;
  uint8                   record_iv_length;
  MACAlgorithm            mac_algorithm;
  uint8                   mac_length;
  uint8                   mac_key_length;
  CompressionMethod       compression_algorithm;
  opaque                  master_secret[48];
  opaque                  client_random[32];
  opaque                  server_random[32];
} SecurityParameters;
```

Поля структуры SecurityParameters имеют следующее значение:

– entity — сторона соединения (клиент или сервер);

– prf_algorithm — алгоритм генерации псевдослучайных чисел;

– bulk_cipher_algorithm — алгоритм шифрования (зашифрования или расшифрования);

– cipher_type — тип алгоритма шифрования;

– enc_key_length — длина ключа алгоритма шифрования;

– block_length — длина блока алгоритма шифрования (только для алгоритмов блочного шифрования);

– fixed_iv_length — длина неявной части синхропосылки, используемой при шифровании;

– record_iv_length — длина явной части синхропосылки, используемой при шифровании;

– mac_algorithm — алгоритм имитозащиты;

– mac_length — длина имитовставки алгоритма имитозащиты;

– mac_key_length — длина ключа алгоритма имитозащиты;

– compression_algorithm — алгоритм, который используется для сжатия данных;

– master_secret — мастер-ключ, который является общим для двух сторон соединения;

– client_random — случайное число клиента;

– server_random — случайное число сервера.

Используемые в `SecurityParameters` типы `PRFAlgorithm`, `BulkCipherAlgorithm`, `CipherType`, `MACAlgorithm` описаны выше. Оставшиеся типы определяются следующим образом:

```
enum {server, client} ConnectionEnd;
enum {null(0), (255)} CompressionMethod;
```

Элементы `server` и `client` перечисления `ConnectionEnd` соответствуют серверу и клиенту. Элемент `null` перечисления CompressionMethod соответствует «пустому» алгоритму сжатия, который не изменяет поступающие на его вход данные.

Протокол Record использует параметры защиты для формирования следующих объектов:

- `client_write_MAC_key` — ключ имитозащиты клиента;
- `server_write_MAC_key` — ключ имитозащиты сервера;
- `client_write_key` — ключ шифрования клиента;
- `server_write_key` — ключ шифрования сервера;
- `client_write_IV` — неявная часть синхропосылки клиента;
- `server_write_IV` — неявная часть синхропосылки сервера.

Объекты строятся по параметрам защиты с помощью алгоритма, определенного в 6.2.3. Не все объекты могут быть задействованы и поэтому некоторые из них могут быть пустыми. Ключи и синхропосылки клиента используются клиентом при отправке данных, а сервером при получении данных, и наоборот.

После того, как параметры защиты согласованы, необходимые ключи и синхропосылки построены, ожидаемые состояния соединения могут быть переведены в текущие. Смена состояний ДОЛЖНА быть учтена при обработке всех следующих фрагментов данных.

Каждое состояние соединения включает следующие элементы:

- состояние алгоритма сжатия;
- состояние алгоритма шифрования, в том числе ключ шифрования для данного соединения. Для алгоритма поточного шифрования в этом состоянии сохраняется вся информация, необходимая для продолжения шифрования следующих фрагментов данных;
- состояние алгоритма имитозащиты, в том числе ключ имитозащиты;
- порядковый номер `seq_num`.

Для состояний чтения и записи должны поддерживаться независимые порядковые номера. Порядковый номер ДОЛЖЕН сбрасываться в 0 всякий раз, когда состояние соединения становится текущим, и увеличиваться на единицу после обработки (передачи или приема) каждого фрагмента данных. Порядковый номер является числом типа `uint64` и не может превышать значения $2^{64} - 1$. Порядковые номера не должны повторяться. Поэтому при достижении максимального порядкового номера реализации TLS должны переустанавливать соединение.

7.3 Выполнение протокола

7.3.1 Фрагментация

Протокол Record принимает неструктурированные данные от протоколов верхнего уровня непустыми блоками произвольной длины. Протокол преобразует блоки в структуры типа `TLSPlaintext`, содержащие фрагменты из не более чем 2^{14} байтов. Протокол Record не обязательно сохраняет размер поступающих блоков, т. е. несколько блоков данных, полученных от одного и того же протокола верхнего уровня, МОГУТ быть объединены в один фрагмент, или, наоборот, один блок МОЖЕТ быть разбит на несколько фрагментов. Тип `TLSPlaintext` определяется следующим образом:

```
struct {
  ContentType type;
  ProtocolVersion version;
  uint16 length;
  opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

Поля структуры `TLSPlaintext` имеют следующее значение:

– `type` — тип содержимого передаваемых данных (см. 7.1);

– `version` — используемая версия TLS;

– `length` — длина в байтах поля `TLSPlaintext.fragment`. Длина НЕ ДОЛЖНА быть больше 2^{14};

– `fragment` — фрагмент данных от протокола верхнего уровня, заданного полем `TLSPlaintext.type`.

Тип `ProtocolVersion` определяется следующим образом:

```
struct {
  uint8 major;
  uint8 minor;
} ProtocolVersion;
```

Поля `major` и `minor` задают старшую и младшую части номера версии протокола. Настоящий стандарт определяет протокол TLS версии 1.2, которому соответствует значение `{3, 3}`.

Реализации TLS НЕ ДОЛЖНЫ посылать фрагменты нулевой длины с типом содержимого `change_cipher_spec`, `alert` и `handshake`. Фрагменты нулевой длины с типом содержимого `application_data` МОГУТ высылаться, например, для анализа помех в канале связи.

Фрагменты с различными типами содержимого МОГУТ чередоваться. Фрагменты прикладных протоколов, как правило, имеют более низкий приоритет для передачи по сравнению с фрагментами других типов. Тем не менее фрагменты ДОЛЖНЫ доставляться на сетевой уровень в том же порядке, в каком к ним применялась защита протоколом Record. Если сторона получает фрагмент прикладного протокола во время выполнения Handshake, то этот фрагмент ДОЛЖЕН обрабатываться с использованием параметров защиты, установленных по завершении предыдущего сеанса Handshake.

7.3.2 Сжатие и восстановление сжатых данных

Все фрагменты сжимаются с помощью алгоритма сжатия, определенного в текущем состоянии сеанса. Всегда существует активный алгоритм сжатия. Первоначально он определяется как «пустой» и задается идентификатором `CompressionMethod.null`. Алгоритм сжатия преобразует структуру типа `TLSPlaintext` в структуру типа `TLSCompressed`:

```
struct {
  ContentType type;
  ProtocolVersion version;
  uint16 length;
  opaque fragment[TLSCompressed.length];
} TLSCompressed;
```

Поля структуры `TLSCompressed` имеют следующее значение:

– `type` — поле, аналогичное `TLSPlaintext.type`;

– `version` — поле, аналогичное `TLSPlaintext.version`;

– `length` — длина в байтах поля `TLSCompressed.fragment`. Длина НЕ ДОЛЖНА быть больше, чем $2^{14} + 1024$;

– `fragment` — сжатое поле `TLSPlaintext.fragment`.

Состояние алгоритма сжатия инициализируются значениями по умолчанию в тот момент, когда состояние соединения становится активным.

Сжатие должно выполняться без потери информации. При сжатии длина содержимого не может увеличиваться (за счет дополнительных заголовков, таблиц сжатия и др.) более чем на 1024 байтов. Если объем данных, восстановленных из `TLSCompressed.fragment`, превысит 2^{14} байтов, то ДОЛЖНО быть отправлено критическое сигнальное сообщение `decompression_failure` (см. 10.3).

Восстановление сжатых данных ДОЛЖНО быть реализовано так, чтобы восстановление не могло привести к переполнению внутренних буферов памяти.

Примечание 1 — Некоторые алгоритмы сжатия для TLS приводятся в [7].

Примечание 2 — Известны атаки на прикладные протоколы, в которых злоумышленник знает формат фрагмента открытых данных и даже частично управляет его содержимым. Злоумышленник использует уровень сжатия (разницу между `TLSPlaintext.length` и `TLSCompressed.length`) для определения недостающих частей фрагмента. Следует учитывать возможность таких атак при проектировании пакетов прикладных протоколов и планировании сжатия в TLS.

7.3.3 Защита данных

Алгоритмы зашифрования и имитозащиты преобразуют структуру типа `TLSCompressed` в структуру типа `TLSCiphertext`. Алгоритм расшифрования выполняет обратный процесс.

Тип `TLSCiphertext` определяется следующим образом:

```
struct {
  ContentType type;
  ProtocolVersion version;
  uint16 length;
  select (SecurityParameters.cipher_type) {
    case stream: GenericStreamCipher;
    case block:  GenericBlockCipher;
    case aead:   GenericAEADCipher;
  } fragment;
} TLSCiphertext;
```

Поля структуры `TLSCiphertext` имеют следующее значение:

– `type` — поле, аналогичное `TLSCompressed.type`;

– `version` — поле, аналогичное `TLSCompressed.version`;

– `length` — длина в байтах поля `TLSCiphertext.fragment`. Длина не должна быть больше, чем $2^{14} + 2048$;

– `fragment` — зашифрованное поле `TLSCompressed.fragment` вместе с имитовставкой.

7.3.3.1 Поточное шифрование

Алгоритмы поточного шифрования преобразуют структуру `TLSCompressed.fragment` в структуру `TLSCiphertext.fragment` типа `GenericStreamCipher` и наоборот.

Тип `GenericStreamCipher` определяется следующим образом:

```
stream-ciphered struct {
  opaque content[TLSCompressed.length];
  opaque MAC[SecurityParameters.mac_length];
} GenericStreamCipher;
```

Поля структуры TLSCompressed имеют следующее значение:

– `content` — поле, аналогичное `TLSCiphertext.fragment`;

– `MAC` — имитовставка, вычисленная с помощью алгоритма, заданного полем `SecurityParameters.mac_algorithm`.

Имитовставка вычисляется на ключе имитозащиты от составной строки

```
seq_num + TLSCompressed.type + TLSCompressed.version +
  TLSCompressed.length + TLSCompressed.fragment,
```

где `seq_num` — порядковый номер фрагмента. Имитовставка вычисляется до зашифрования отправляемого фрагмента и после расшифрования принятого.

Зашифровывается составная строка `content + mac`. Длина `TLSCiphertext.length` равна сумме значений полей `TLSCompressed.length` и `SecurityParameters.mac_length`.

7.3.3.2 Блочное шифрование

Алгоритмы блочного шифрования используются в режиме сцепления блоков. Они преобразуют структуру `TLSCompressed.fragment` в структуру `TLSCiphertext.fragment` типа `GenericBlockCipher` и наоборот.

Тип `GenericBlockCipher` определяется следующим образом:

```
struct {
  opaque IV[SecurityParameters.record_iv_length];
  block-ciphered struct {
    opaque content[TLSCompressed.length];
    opaque MAC[SecurityParameters.mac_length];
    uint8 padding[GenericBlockCipher.padding_length];
    uint8 padding_length;
  };
} GenericBlockCipher;
```

Поля структуры `GenericBlockCipher` имеют следующее значение:

– `IV` — синхропосылка, которую СЛЕДУЕТ выбирать случайно и которая ДОЛЖНА быть непредсказуемой. Для алгоритмов блочного шифрования длина синхропосылки определяется полем `SecurityParameters.record_iv_length`, значение которого ДОЛЖНО совпадать со значением поля `SecurityParameters.block_size`;

– `content` — поле, аналогичное `TLSCompressed.fragment`;

– `MAC` — имитовставка, которая вычисляется в соответствии с 7.3.3.1;

– `padding` — дополнение, которое добавляется к открытым данным для выравнивания их на границу блока алгоритма шифрования. Дополнение МОЖЕТ быть любой длины, не превосходящей 255, при этом длина дополнения ДОЛЖНА выбираться так, чтобы значение поля `TLSCiphertext.length` (т. е. размер структуры типа `GenericBlockCipher`) было кратно длине блока алгоритма шифрования. Дополнения, размеры которых больше минимально необходимых, могут использоваться для предотвращения атак, основанных на анализе длин передаваемых сообщений. Каждый байт дополнения ДОЛЖЕН быть заполнен значением длины дополнения. Получатель ДОЛЖЕН проверить дополнение и ДОЛЖЕН отправить критическое сигнальное сообщение `bad_record_mac` (см. 10.3), если дополнение некорректно;

– padding_length — длина дополнения в байтах. Может принимать значения от 0 до 255 включительно.

При блочном шифровании значение поля TLSCiphertext.length на единицу больше, чем сумма значений полей SecurityParameters.block_length, TLSCompressed.length, SecurityParameters.mac_length и padding_length.

Пример — Пусть длина блока (в байтах) алгоритма шифрования равна 8, длина содержимого (TLSCompressed.length***) равна 61, а длина имитовставки — 20. Тогда суммарная длина полей*** content, MAC, padding_length ***равняется 82. Для того, чтобы общая длина зашифровываемых данных была кратна длине блока, необходимо выбрать дополнение, длина которого должна принимать одно из следующих значений: 6, 14, 22, . . . , 254. Дополнение минимальной длины будет состоять из 6 байтов, каждый из которых содержит значение 6. Таким образом, последние 8 байтов структуры*** GenericBlockCipher ***до их зашифрования будут иметь следующий вид: XX 06 06 06 06 06 06 06, где XX — последний байт имитовставки (поле MAC).***

Примечание 1 — При шифровании в режиме сцепления блоков весь фрагмент открытого текста ДОЛЖЕН быть известен до передачи какой-либо части соответствующего шифртекста.

Примечание 2 — В [8] описана атака на механизм дополнения данных перед зашифрованием, основанная на замерах времени вычисления имитовставки расшифрованных данных. Для защиты от данной атаки реализации ДОЛЖНЫ обрабатывать зашифрованные фрагменты данных за одно и то же время, вне зависимости от того, какое дополнение получено при расшифровании, корректное или нет. Для этого рекомендуется вычислять имитовставку даже в том случае, когда дополнение некорректно, и только после этого браковать фрагмент. Для защиты от атаки, предложенной в [9], рекомендуется и при корректном, и при некрректном дополнении вычислять имитовставку за максимальное время. Это время, за которые были бы выполнены вычисления, если бы дополнение имело нулевую длину.

7.3.3.3 Одновременное шифрование и имитозащита

Алгоритм установки защиты преобразует структуру TLSCompressed.fragment в структуру TLSCiphertext.fragment типа GenericAEADCipher, а алгоритм снятия защиты выполняет обратное преобразование.

Тип GenericAEADCipher определяется следующим образом:

```
struct {
  opaque nonce_explicit[SecurityParameters.record_iv_length];
  aead-ciphered struct {
    opaque content[TLSCompressed.length];
  };
} GenericAEADCipher;
```

При установке защиты клиент использует ключ шифрования client_write_key, а сервер — server_write_key. Ключи имитозащиты не используются. Каждый криптонабор, поддерживающий алгоритмы одновременного шифрования и имитозащиты, ДОЛЖЕН определять, каким образом формируется синхропосылка.

Синхропосылка состоит из двух частей — явной, которая передается вместе с фрагментами данных, и неявной, которая определяется по параметрам защиты и не передается вместе с данными.

Явная часть синхропосылки размещается в поле GenericAEAEDCipher. nonce_explicit. Неявную часть СЛЕДУЕТ определять по строке client_write_iv

(при отправке данных клиентом) или server_write_iv (при отправке данных сервером). Формирование этих строк описывается в 7.4.

Критическими данными алгоритмов одновременного шифрования и имитозащиты является поле TLSCompressed.fragment. Дополнительные открытые данные, обозначаемые additional_data, определяются следующим образом:

```
additional_data = seq_num + TLSCompressed.type +
  TLSCompressed.version + TLSCompressed.length,
```

где seq_num — порядковый номер фрагмента.

Длина выходных данных алгоритма установки защиты, как правило, больше, чем длина критических данных (т. е. больше значения поля TLSCompressed.length). Увеличение длины определяется спецификой используемого алгоритма. Так как алгоритмы одновременного шифрования и имитозащиты могут включать в себя механизмы выравнивания данных на границу блока, длина выходных данных алгоритма установки защиты может меняться в зависимости от значения поля TLSCompressed.length. При установке защиты длина критических данных НЕ ДОЛЖНА увеличиваться более чем на 1024 байтов.

Если при обработке данных алгоритм снятия защиты возвратил признак нарушения целостности данных, то ДОЛЖНО быть отправлено критическое сигнальное сообщение bad_record_mac (см. 10.3).

7.4 Формирование ключей

Протокол Record использует алгоритм генерации псевдослучайных чисел для формирования ключей текущего состояния соединения. С помощью этого алгоритма могут также формироваться синхропосылки алгоритмов одновременного шифрования и имитозащиты.

При генерации используются параметры защиты (см. 7.2), выработанные сторонами по протоколу Handshake. По мастер-ключу и случайным данным клиента и сервера генерируется строка байтов key_block:

```
key_block = PRF(SecurityParameters.master_secret,
  "key expansion", SecurityParameters.server_random +
     SecurityParameters.client_random);
```

Ключи и синхропосылки формируются по этой строке в следующем порядке (объекты нулевой длины считаются пустыми):

- ключ client_write_MAC_key длины SecurityParameters.mac_key_length;
- ключ server_write_MAC_key длины SecurityParameters.mac_key_length;
- ключ client_write_key длины SecurityParameters.enc_key_length;
- ключ server_write_key длины SecurityParameters.enc_key_length;
- синхропосылка client_write_IV длины SecurityParameters.fixed_iv_length;
- синхропосылка server_write_IV длины SecurityParameters.fixed_iv_length.

Генерируется столько байтов key_block, сколько требуется для построения всех ключей и синхропосылок.

8 Протокол Handshake

8.1 Состояние сеанса

Протокол Handshake выполняется поверх протокола Record и отвечает за установку, возобновление или переустановку связи. При установке связи создается новый сеанс. Сеанс описывается состоянием со следующими элементами:

– идентификатор сеанса — произвольная строка байтов, выбираемая сервером для идентификации активного (текущего) или возобновляемого сеанса;

– сертификат стороны — сертификат открытого ключа стороны. Формат сертификата должен соответствовать СТБ 34.101.19. Этот элемент состояния может отсутствовать;

– алгоритм сжатия;

– криптоопределение;

– мастер-ключ — секретная строка из 48 байтов, общая для клиента и сервера;

– флаг возобновления — признак, который определяет, может ли сеанс использоваться для создания по нему новых состояний соединения.

По состоянию сеанса создаются параметры защиты, по которым затем формируются состояния соединения. Эти состояния используются протоколом Record при защите данных прикладных протоколов. Благодаря наличию в протоколе Handshake механизмов возобновления и переустановки связи, с использованием одного и того же сеанса может быть установлено множество соединений.

8.2 Шаги протокола

При установке связи клиент и сервер согласуют версию TLS, выбирают криптографические алгоритмы, проводят аутентификацию друг друга (при необходимости), формируют общий секретный ключ. При установке связи протокол Handshake состоит из следующих этапов:

– обмен приветственными сообщениями с целью согласования алгоритмов шифрования, передачи случайных данных, проверки возобновляемости сеанса;

– обмен необходимыми криптографическими параметрами для согласования клиентом и сервером предварительного мастер-ключа `pre_master_secret`;

– обмен сертификатами и другими данными для взаимной аутентификации клиента и сервера (при необходимости);

– выработка мастер-ключа `master_secret` с использованием `pre_master_secret` и случайных данных, которыми обменялись клиент и сервер;

– передача параметров защиты протоколу Record;

– проверка того, что противоположная сторона вычислила те же параметры защиты, и что сеанс создан без вмешательства злоумышленника.

Протоколам, выполняемым поверх TLS, не следует полностью полагаться на то, что стороны согласуют наиболее надежный для них способ соединения. Злоумышленник может провести атаку типа «человек посередине» и навязать обеим сторонам наименее безопасный способ связи из тех, которые они могут поддержать. TLS сводит риск таких атак к минимуму, но тем не менее атаки возможны. Например, злоумышленник может заблокировать доступ к коммуникационному порту, который соответствует TLS, или заставить стороны взаимодействовать без TLS. Для протоколов верхних уровней должен быть определен необходимый уровень безопасности, эти протоколы не должны передавать информацию по каналам связи менее безопасным способом, чем необходимо. TLS безопасен в том смысле, что каждый криптонабор предоставляет заявленный уровень защиты.

Этапы протокола Handshake детализируются следующим образом (содержание и значение каждого сообщения определяются в следующих подразделах).

Клиент отправляет сообщение `ClientHello`, на которое сервер должен ответить сообщением `ServerHello`. Если ответ от сервера не получен, то генерируется ошибка и соединение не устанавливается. Сообщения `ClientHello` и `ServerHello` используются для выбора наиболее безопасного из возможных способов связи между клиентом и сервером.

С помощью сообщений `ClientHello` и `ServerHello` согласуются следующие параметры связи: версия протокола, идентификатор сеанса, криптонабор и алгоритм сжатия. Дополнительно с помощью данных сообщений стороны обмениваются случайными данными клиента `ClientHello.random` и сервера `ServerHello.random`.

Для формирования общего ключа используется до четырех сообщений: сообщения сервера `Certificate`, `ServerKeyExchange` и сообщения клиента `Certificate`, `ClientKeyExchange`. TLS допускает разработку новых алгоритмов формирования общего ключа путем определения форматов указанных сообщений и правил их обработки. Общий ключ ДОЛЖЕН быть достаточно длинным, рекомендуется использовать ключ длиной не менее 48 байтов.

Если сервер должен быть аутентифицирован, то следом за приветственными сообщениями он отправляет свой сертификат в сообщении `Certificate`. Дополнительно, если требуется, может быть отправлено сообщение `ServerKeyExchange` (например, если сервер не имеет сертификата или его сертификат содержит только открытый ключ ЭЦП). Сервер может потребовать сертификат от клиента, если сервер аутентифицируется и если запрос сертификата предусмотрен выбранным криптонабором. Сервер отправляет сообщение `ServerHelloDone`, указывающее на то, что фаза обмена приветственными сообщениями закончена, и ждет ответа клиента. Если сервер отправил сообщение `CertificateRequest`, то клиент ДОЛЖЕН отправить сообщение `Certificate`, а затем сообщение `ClientKeyExchange`. Содержание сообщения `ClientKeyExchange` зависит от алгоритма с открытым ключом, который был выбран при обмене сообщениями `ClientHello` и `ServerHello`. Если клиент послал сертификат открытого ключа, который может использоваться для проверки ЭЦП, то отправляется сообщение `CertificateVerify` с атрибутом `digitally-signed`. Это сообщение отправляется для того, чтобы доказать владение личным ключом, соответствующим открытому ключу сертификата.

После выполнения описанных действий клиент отправляет сообщение `ChangeCipherSpec` и переводит ожидаемое состояние записи соединения, содержащее согласованное криптоопределение, в текущее. Сразу после этого клиент посылает сообщение `Finished`, защищенное с помощью алгоритмов и ключей из нового криптоопределения. В ответ сервер выполняет аналогичные действия: отправляет свое сообщение `ChangeCipherSpec`, переводит свое ожидаемое состояние записи в текущее и посылает свое сообщение `Finished`, защищенное с использованием согласованного криптоопределения. При получении `ChangeCipherSpec` стороны переводят свои ожидаемые состояния чтения в текущие. Стороны проверяют сообщения `Finished` и на этом установка связи завершается.

После установки связи клиент и сервер могут начать обмен данными прикладных протоколов. На рисунке 1 изображен полный обмен сообщениями при установке связи. Звездочкой помечены необязательные сообщения, символом «+» — сообщения, связанные с аутентификацией клиента. Сообщения `ChangeCipherSpec`, выделенные квадратными скобками, не относятся к протоколу Handshake.

Данные прикладных протоколов НЕЛЬЗЯ посылать до того, как завершится установка связи (т. е. до того, как будет согласован и применен криптонабор, отличный от `TLS_NULL_WITH_NULL_NULL`).

Когда клиент и сервер решают возобновить или переустановить связь, используя параметры уже существующего сеанса (вместо того, чтобы согласовывать новые параметры), выполняется сокращенный обмен сообщениями.

Клиент отправляет сообщение `ClientHello`, в котором указывает идентификатор сеанса, который должен использоваться для возобновления связи. При получении данного

клиент		сервер
ClientHello	→	
	←	ServerHello Certificate* ServerKeyExchange* CertificateRequest*+ ServerHelloDone
Certificate* ClientKeyExchange CertificateVerify*+ [ChangeCipherSpec] Finished	→	
	←	[ChangeCipherSpec] Finished
[Данные прикладного протокола]	↔	[Данные прикладного протокола]

Рисунок 1 — Полный обмен сообщениями протокола Handshake

сообщения сервер проверяет список сеансов, которые были созданы ранее (кэш сеансов), и ищет сеанс с присланным идентификатором. Если совпадение найдено, и сервер желает возобновить связь, используя найденный сеанс, то он отправляет сообщение ServerHello с тем же самым идентификатором сеанса. После этого клиент и сервер ДОЛЖНЫ отправить сообщения ChangeCipherSpec и перейти к пересылке сообщений Finished. На рисунке 2 изображен сокращенный обмен сообщениями между клиентом и сервером при возобновлении связи. Аналогичный обмен выполняется при переустановке связи.

Если сервер не нашел сеанс с нужным идентификатором, то он генерирует новое значение идентификатора. После чего клиент и сервер выполняют полный обмен сообщениями по протоколу Handshake.

клиент		сервер
ClientHello	→	
	←	ServerHello [ChangeCipherSpec] Finished
[ChangeCipherSpec] Finished	→	
[Данные прикладного протокола]	↔	[Данные прикладного протокола]

Рисунок 2 — Сокращенный обмен сообщениями протокола Handshake

8.3 Сообщения протокола

Сообщения протокола Handshake передаются протоколу Record, который помещает их в одну или более структур TLSPlaintext. Данные структуры обрабатываются и передаются в соответствии с текущим активным состоянием сеанса.

Сообщения протокола Handshake определяются следующим образом:

```
struct {
  HandshakeType msg_type;
  uint24 length;
  select (HandshakeType) {
```

```
      case hello_request:       HelloRequest;
      case client_hello:        ClientHello;
      case server_hello:        ServerHello;
      case certificate:         Certificate;
      case server_key_exchange: ServerKeyExchange;
      case certificate_request: CertificateRequest;
      case server_hello_done:   ServerHelloDone;
      case certificate_verify:  CertificateVerify;
      case client_key_exchange: ClientKeyExchange;
      case finished:            Finished;
    } body;
} Handshake;
enum {
    hello_request(0), client_hello(1), server_hello(2),
    certificate(11), server_key_exchange(12),
    certificate_request(13), server_hello_done(14),
    certificate_verify(15), client_key_exchange(16),
    finished(20), (255)
} HandshakeType;
```

Поля структуры `Handshake` имеют следующее значение:

– `msg_type` — тип сообщения, который описывается перечислением `HandshakeType`. Элементы этого перечисления соответствуют сообщениям, определенным в 8.4 — 8.15;

– `length` — длина в байтах поля `Handshake.body`;

– `body` — тело сообщения.

Сообщения протокола Handshake определяются в следующих подразделах в порядке, в котором они ДОЛЖНЫ отправляться. Отправка сообщений в порядке, который не предусмотрен настоящим стандартом, приводит к ошибке. Ненужные сообщения могут быть пропущены. Сообщение `HelloRequest` не подчиняется данному порядку следования: оно может быть отправлено в любое время, но клиенту СЛЕДУЕТ его игнорировать, если оно приходит во время установки связи.

Сообщения `HelloRequest`, `ClientHello`, `ServerHello` являются приветственными. Эти сообщения используются для выбора наиболее безопасного из возможных способов связи между клиентом и сервером. Когда создается новый сеанс, в состояниях соединения заданы «пустые» алгоритмы сжатия, шифрования и имитозащиты. Эти алгоритмы используются для обработки сообщений вплоть до согласования новых алгоритмов.

8.4 Сообщение `HelloRequest`

Сообщение `HelloRequest` сервер МОЖЕТ отправить в любое время. Данное сообщение сообщает клиенту, что он должен снова начать согласование параметров связи. В ответ на сообщение `HelloRequest` клиент, если посчитает нужным, должен отправить сообщение `ClientHello`. Сообщение `HelloRequest` не предназначено для того, чтобы определить, какая из сторон является клиентом, а какая — сервером. Серверу НЕ СЛЕДУЕТ отправлять сообщение сразу после того, как клиент инициировал соединение (отправил сообщение `ClientHello`).

Клиенту следует игнорировать сообщение `HelloRequest`, если в момент его получения он уже находится в процессе установки связи. Клиент МОЖЕТ проигнорировать

сообщение, если он не желает переустанавливать связь. Клиент также может ответить сигнальным сообщением `no_renegotiation` (см. 10.3).

Между отправкой `HelloRequest` и получением `ClientHello` серверу могут приходить от клиента фрагменты данных прикладных протоколов. Ожидается, что сервер получит только малое число таких фрагментов, поскольку сообщения протокола Handshake при пересылке имеют приоритет над данными прикладных протоколов. Если сервер отправляет `HelloRequest`, но не получает в ответ `ClientHello`, то он может закрыть соединение с критическим сигнальным сообщением.

После отправки сообщения `HelloRequest`, серверу НЕ СЛЕДУЕТ повторять запрос до тех пор, пока не завершится последующий обмен сообщениями протокола Handshake.

Сообщение `HelloRequest` имеет следующую структуру:

```
struct {} HelloRequest;
```

Это сообщение НЕ ДОЛЖНО учитываться при вычислении хэш-значений, используемых в сообщениях `Finished` и `CertificateVerify`.

8.5 Сообщение `ClientHello`

Клиент посылает сообщение `ClientHello` при установке связи (полной или сокращенной). Клиент также может послать это сообщение после установки связи либо в ответ на сообщение `HelloRequest`, либо по собственной инициативе с целью переустановки связи.

Сообщение `ClientHello` определяется следующим образом:

```
struct {
  ProtocolVersion client_version;
  Random random;
  SessionID session_id;
  CipherSuite cipher_suites<2..2^16-2>;
  CompressionMethod compression_methods<1..2^8-1>;
  select (extensions_present) {
    case false:
      struct {};
    case true:
      Extension extensions<0..2^16-1>;
  };
} ClientHello;
struct {
  uint32 gmt_unix_time;
  opaque random_bytes[28];
} Random;
opaque SessionID<0..32>;
```

Поля структуры `ClientHello` имеют следующее значение:

– `client_version` — версия TLS, которую клиент предлагает использовать в течение сеанса. Тип данного поля определен в 7.3.1;

– `random` — структура со случайными данными, сгенерированными клиентом;

– `session_id` — идентификатор сеанса, параметры защиты которого клиент предлагает использовать в данном соединении. В этом поле задается пустая строка, если клиент желает создать новый сеанс. Непустая строка может соответствовать идентификатору, использованному в 1) соединении, которое было закрыто, 2) текущем соединении, 3) другом

активном в настоящее время соединении. Второй вариант приводит только к обновлению поля `random` и производных от него параметров соединения. Третий вариант позволяет установить несколько независимых безопасных соединений по сокращенной схеме обмена сообщениями протокола Handshake (эти независимые соединения могут устанавливаться последовательно или одновременно);

– `cipher_suites` — список идентификаторов криптонаборов, поддерживаемых клиентом. Список упорядочивается клиентом в порядке убывания приоритета. Если поле `session_id` не пустое (подразумевается запрос на возобновление или переустановку связи), то список ДОЛЖЕН содержать криптонабор `ServerHello.cipher_suite`, согласованный в сеансе с идентификатором `session_id`. Сервер выбирает криптонабор из полученного списка или, если отсутствуют приемлемые для него варианты, отправляет критическое сигнальное сообщение `handshake_failure` (см. 10.3) и закрывает соединение. Если список содержит криптонаборы, которые сервер не распознает, не поддерживает или не желает использовать, то он ДОЛЖЕН проигнорировать эти криптонаборы, а оставшиеся криптонаборы обработать обычным образом;

– `compression_methods` — список поддерживаемых клиентом алгоритмов сжатия, упорядоченный клиентом в порядке убывания приоритета. Тип данного поля определен в 7.2. Если поле `session_id` не пустое, то список ДОЛЖЕН содержать алгоритм `ServerHello.compression_method`, согласованный в сеансе с идентификатором `session_id` . Список ДОЛЖЕН содержать и все реализации ДОЛЖНЫ поддерживать алгоритм `CompressionMethod.null`. Это позволит клиенту и серверу всегда согласовать алгоритм сжатия;

– `extensions` — список расширений, которые клиенты МОГУТ отправить серверу для согласования дополнительных функциональных возможностей. Тип данного поля определен в 8.7.

Поля структуры `Random` имеют следующее значение:

– `gmt_unix_time` — текущее время и дата согласно таймеру отправителя в стандартном формате UNIX (т. е. количество секунд, прошедших начиная с 0 часов 1 января 1970 года в стандартном времени по Гринвичу, игнорируя секунды координации). TLS не требует корректной установки таймера (прикладные протоколы или протоколы более высокого уровня могут определять дополнительные требования);

– `random_bytes` — 28 байтов, сгенерированных с помощью криптографически стойкого генератора случайных или псевдослучайных чисел.

Идентификатор сеанса становится действительным, когда протокол Handshake завершается сообщениями `Finished`, и сохраняется до тех пор, пока не будет удален по истечении срока хранения состояния сеанса или из-за ошибки в соединении, связанном с данным сеансом. Фактическое значение идентификатора сеанса определяется сервером.

Так как идентификатор сеанса передается в незашифрованном виде, сервер НЕ ДОЛЖЕН помещать в идентификатор конфиденциальную информацию. Целостность идентификатора контролируется отложенно (при обработке `Finished`), поэтому стороны должны корректно обрабатывать любые, даже заведомо неверные идентификаторы, которые может посылать злоумышленник.

Сообщения `ClientHello` могут содержать расширения. Для проверки наличия расширений следует проверить, что сообщение не заканчивается полем `compression_methods` и что за этим полем имеются дполнительные байты. Такой метод выявления необязательных данных отличается от обычного в TLS способа обработки полей переменной длины. Метод выбран для совместимости с более ранними версиями TLS, в которых расширения не были предусмотрены.

Если клиент, используя `extensions`, запрашивает дополнительные функциональные возможности и эти функциональные возможности сервер не поддерживает, клиент МОЖЕТ аварийно прекратить установку связи. Сервер ДОЛЖЕН принимать сообщение `ClientHello` как с полем `extensions`, так и без него. При этом он ДОЛЖЕН проверить (как и для всех других сообщений), что объем данных в сообщении точно совпадает с объемом данных, определяемых форматом сообщения. Если объемы данных не совпадают, то сервер ДОЛЖЕН отправить критическое сигнальное сообщение `decode_error` (см. 10.3).

После отправки сообщения `ClientHello` клиент ожидает получения сообщения сервера `ServerHello`. Отправка сервером любого другого сообщение протокола Handshake, за исключением сообщения `HelloRequest`, рассматривается как ошибка.

8.6 Сообщение `ServerHello`

Сообщение `ServerHello` отправляется сервером в ответ на сообщение `ClientHello`, когда он может выбрать из предложенного клиентом списка приемлемый для него криптонабор. Если сервер не может выбрать такой криптонабор, то он отправляет критическое сигнальное сообщение `handshake_failure` (см. 10.3).

Сообщение `ServerHello` определяется следующим образом:

```
struct {
  ProtocolVersion server_version;
  Random random;
  SessionID session_id;
  CipherSuite cipher_suite;
  CompressionMethod compression_method;
  select (extensions_present) {
    case false:
      struct {};
    case true:
      Extension extensions<0..2^16-1>;
  };
} ServerHello;
```

Поля структуры `ServerHello` имеют следующее значение:

– `server_version` — поле содержит согласованную между клиентом и сервером версию TLS. Тип данного поля определен в 7.3.1;

– `random` — поле со случайными данными, сгенерированными сервером. Тип данного поля определен в 8.5. Случайные данные ДОЛЖНЫ генерироваться независимо от `ClientHello.random`;

– `session_id` — идентификатор сеанса, соответствующего данному соединению. Если поле `session_id` в сообщении `ClientHello` не было пустым, то сервер ищет присланный идентификатор в кэше сеансов. Если сеанс с нужным идентификатором найден, и сервер желает установить связь, используя состояние этого сеанса, то он отвечает тем же идентификатором, который был прислан клиентом. Этот ответ сервера указывает на возобновление связи и требует от сторон перехода сразу к сообщениям `Finished`. В противном случае поле `session_id` должно содержать значение, соответствующее новому сеансу. Сервер может вернуть пустое поле `session_id`, чтобы указать на то, что сеанс не будет кэшироваться и не может использоваться для возобновления связи в будущем. При возобновлении связи должен использоваться тот самый криптонабор, который был согласован ранее. От сервера не требуется возобновлять связь, даже если сеанс с нужным

идентификатором имеется в кэше. Клиенты ДОЛЖНЫ быть готовы выполнить полное согласование параметров связи (включая согласование криптонаборов) всякий раз при запросе на возобновление связи;

– `cipher_suite` — идентификатор криптонабора, который сервер выбрал из списка, переданного клиентом в поле `ClientHello.cipher_suites`. При возобновлении связи сервер не выбирает криптонабор, а задает его по сохраненному состоянию сеанса;

– `compression_method` — алгоритм сжатия, который сервер выбрал из списка, переданного клиентом в поле `ClientHello.compression_methods`. При возобновлении связи сервер не выбирает алгоритм, а задает его по сохраненному состоянию сеанса;

– `extensions` — список расширений. Только расширения, предложенные клиентом, могут включаться в список сервера. Признаком передачи расширений является наличие байтов за полем `compression_method` (см. 8.5).

Определенные выше правила возобновления связи действуют также при переустановке связи.

8.7 Расширения приветственных сообщений

Расширения приветственных сообщений описываются следующим типом:

```
struct {
  ExtensionType extension_type;
  opaque extension_data<0..2^16-1>;
} Extension;
```

Поля структуры `Extension` имеют следующее значение:

– `extension_type` — тип расширения;

– `extension_data` — данные расширения, т. е. информация, специфическая для конкретного типа расширения.

Вложенный тип `ExtensionType` определяется следующим образом:

```
enum {
  signature_algorithms(13), renegotiation_info(65281), (65535)
} ExtensionType;
```

Примечание — Некоторые распространенные дополнительные расширения определены в [10].

В сообщение `ServerHello` НЕЛЬЗЯ включать расширения с типом, который не встречается в соответствующем сообщении `ClientHello`. Если клиент получает в `ServerHello` расширения с типом, который не был указан в соответствующем `ClientHello`, то он ДОЛЖЕН прервать установку связи с сигнальным сообщением `unsupported_extension` (см. 10.3). Тем не менее предусмотрена возможность использования расширений, ориентированных на отправку сервером. Такие расширения с пустым полем `extension_data` клиент посылает серверу в сообщении `ClientHello`. Этим клиент демонстрирует серверу, что он может обрабатывать расширение данного типа. В ответ на сообщение клиента сервер высылает расширение того же типа, но уже с заполненными данными.При реализации TLS следует учитывать, что в сообщениях `ClientHello`, `ServerHello` расширения различных типов МОГУТ встречаться в любом порядке. Расширения одного типа НЕЛЬЗЯ включать в список более одного раза.

Расширение может посылаться и при установке связи, и при ее возобновлении. Необходимость отправки расширения при возобновлении связи объясняется тем, что клиент, который запрашивает возобновление, не знает наверняка, примет ли сервер запрос на

возобновление или нет. Следовательно, клиенту СЛЕДУЕТ отправить серверу те же расширения, какие бы он послал, если бы начинал новый сеанс.

В общем случае, при описании каждого расширения требуется определить способ его обработки как при полном, так и при сокращенном обмене сообщениями протокола Handshake. С другой стороны, большинство используемых расширений TLS применяются только при установке связи. При возобновлении связи сервер не обрабатывает такие расширения, даже если они присутствуют в сообщении `ClientHello`, и не включает их в сообщение `ServerHello`.

При разработке новых расширений необходимо учитывать следующее:

– сервер может отказаться обрабатывать расширение как из-за ошибки, так и ввиду отсутствия необходимости поддерживать определенные функциональные возможности. В первом случае следует использовать сигнальные сообщения об ошибке. Во втором случае серверу следует объяснить отказ в поле `extension_data`;

– расширения должны, по возможности, разрабатываться так, чтобы предотвращать атаку, которая навязывает использование (или наоборот отключает) определенные функциональные возможности путем манипуляции сообщениями протокола Handshake. Подлинность сообщений протокола Handshake, в том числе подлинность расширений, контролируется при обработке `Finished`. Тем не менее необходимо уделять особое внимание случаям, когда расширение изменяет смысл сообщений, посылаемых при установке связи. Пока установка связи не завершена, злоумышленник может модифицировать сообщения и вставить, удалить или переместить расширения;

– существует техническая возможность использовать расширения для изменения базовых конструктивных схем протокола TLS, например, схемы согласования криптонабора. Поддерживать изменение базовых схем не рекомендуется. Более подходящим будет определить новую версию TLS.

8.7.1 Расширение `signature_algorithms`

Клиент использует расширение с типом `signature_algorithms`, чтобы указать серверу, какие пары «алгоритм хэширования, алгоритмы ЭЦП» могут использоваться при выработке и проверке ЭЦП. Поле `extension_data` этого расширения содержит значение `supported_signature_algorithms`, которое является списком с элементами типа `SignatureAndHashAlgorithm`:

```
SignatureAndHashAlgorithm supported_signature_algorithms
   <2..2^16-2>;
```

Элементы в списке указываются в порядке убывания приоритета.

Если клиент поддерживает только определенную пару «алгоритм хэширования, алгоритмы ЭЦП», которая по умолчанию используется вместе с согласуемыми криптонаборами, то он МОЖЕТ опустить расширение `signature_algorithms`. Если клиент не поддерживает пару по умолчанию или поддерживает другие пары (и намеревается их использовать для проверки сообщений сервера, т. е. для проверки сертификатов и подписи в `ServerKeyExchange`), то клиент ДОЛЖЕН выслать расширение `signature_algorithms`, перечислив поддерживаемые пары.

Серверам НЕЛЬЗЯ посылать расширение с типом `signature_algorithms`. Серверы ДОЛЖНЫ поддерживать получение этого расширения.

Когда выполняется возобновление сеанса, расширение с типом `signature_algorithms` не включается в сообщение `ServerHello` и сервер игнорирует данное расширение в сообщении `ClientHello`.

8.7.2 Расширение `renegotiation_info`

Расширение с типом `renegotiation_info` используется для связывания соединения, в котором выполняется переустановка связи, с предыдущим соединением. Такое связывание защищает от атак, в которых злоумышленник сначала устанавливает связь с сервером, а затем инициирует переустановку связи и передает выполнение протокола клиенту. По завершении атаки между клиентом и сервером устанавливается обычное защищенное соединение, но сервер ошибочно считает, что оно является продолжением предыдущего соединения, которое было установлено со злоумышленником.

Если клиент и сервер хотят использовать механизм переустановки связи, то они ДОЛЖНЫ хранить три дополнительных параметра соединения:

– `secure_renegotiation` — флаг того, что поддерживается переустановка связи;

– `client_verify_data` — поле `verify_data` сообщения `Finished`, которое было выслано клиентом по завершении предыдущего сеанса протокола Handshake;

– `server_verify_data` — поле `verify_data` сообщения `Finished`, которое было выслано сервером по завершении предыдущего сеанса протокола Handshake.

Эти параметры касаются соединения в целом (в обоих направлениях). Параметры не относятся к сеансу, и поэтому их не требуется сохранять в кэше сеансов.

Содержимое расширения описывается следующим образом:

```
struct {
  opaque renegotiated_connection<0..255>;
} RenegotiationInfo;
```

При установке связи поле `renegotiated_connection` устанавливается пустым как в сообщении `ClientHello`, так и в сообщении `ServerHello`. При переустановке связи поле устанавливается равным `client_verify_data` (12 байтов) в сообщении `ClientHello` и `client_verify_data + server_verify_data` (24 байта) в сообщении `ServerHello`.

Действия клиента при установке связи. Если клиент собирается переустанавливать связь, то при установке связи (полной или сокращенной) он ДОЛЖЕН включить расширение `renegotiation_info` с пустым содержимым в свое сообщение `ClientHello`. После получения сообщения `ServerHello` от сервера клиент ДОЛЖЕН проверить, что это сообщение включает расширение `renegotiation_info`. Отсутствие расширения означает, что сервер не поддерживает безопасную переустановку связи. Если расширение отсутствует, то клиент должен сбросить флаг `secure_renegotiation` в 0. Более того, клиент может разорвать связь, если безопасная переустановка является для него критичной.

Примечание — Здесь и далее в параграфе разрыв связи означает отправку сигнального сообщения `handshake_failure` с последующим закрытием соединения.

Если расширение присутствует, то клиент устанавливает флаг `secure_renegotiation` в 1. Клиент ДОЛЖЕН проверить, что длина поля `renegotiated_connection` равняется 0. Если последнее условие нарушается, то клиент ДОЛЖЕН разорвать связь.

После установки связи клиент должен сохранить атрибуты `client_verify_data` и `server_verify_data`.

Действия клиента при переустановке связи. При переустановке связи клиент проверяет значение флага `secure_renegotiation`. Если флаг равняется 0, то клиент не должен инициировать переустановку. Если флаг равняется 0 и переустановку инициировал сервер, то клиенту РЕКОМЕНДУЕТСЯ отвергнуть ее. При этом клиент ДОЛЖЕН выслать предупредительное сигнальное сообщение `no_renegotiate`.

Если флаг равняется 1, то клиент может переустанавливать связь. Клиент включает расширение `renegotiation_info` в свое сообщение `ClientHello`. В содержимом рас-

ширения должна передаваться строка client_verify_data. После получения сообщения ServerHello клиент ДОЛЖЕН проверить, что расширение renegotiation_info в нем присутствует, первая половина поля renegotiated_connection совпадает с сохраненным значением client_verify_data, а вторая половина совпадает с сохраненным значением server_verify_data. При нарушении любого из этих условий клиент ДОЛЖЕН прервать переустановку связи.

После переустановки связи клиент должен сохранить новые значения client_verify_data и server_verify_data.

Действия сервера при установке связи. При установке связи (полной или сокращенной) сервер ДОЛЖЕН проверить, что сообщение ClientHello включает расширение renegotiation_info. Отсутствие расширения означает, что клиент не поддерживает безопасную переустановку связи. Если расширение отсутствует, то сервер должен сбросить флаг secure_renegotiation в 0. Более того, сервер может разорвать связь, если безопасная переустановка является для него критичной.

Если расширение renegotiation_info включено в ClientHello, то сервер проверяет, что его содержимое является пустым. При нарушении этого условия сервер ДОЛЖЕН разорвать связь. Если содержимое расширения пусто, то сервер устанавливает флаг secure_renegotiation в 1 и возвращает такое же пустое расширение в своем сообщении ServerHello.

После установки связи сервер должен сохранить атрибуты client_verify_data и server_verify_data.

Действия сервера при переустановке связи. При переустановке связи клиент проверяет значение флага secure_renegotiation. Если флаг равняется 0, то сервер не должен инициировать переустановку. Если флаг равняется 0 и переустановку инициировал клиент, то серверу РЕКОМЕНДУЕТСЯ отвергнуть ее.

Если флаг равняется 1, то сервер может переустанавливать связь. Сервер проверяет расширение renegotiation_info в сообщении ClientHello. Если расширения нет или его содержимое отличается от client_verify_data, то сервер ДОЛЖЕН разорвать связь. В противном случае сервер формирует расширение renegotiation_info по server_verify_data, client_verify_data и отправляет его в сообщении ServerHello.

После переустановки связи сервер должен сохранить новые значения client_verify_data и server_verify_data.

8.8 Сообщение сервера Certificate

Сервер ДОЛЖЕН посылать сообщение Certificate всегда, когда в согласованном алгоритме формирования общего ключа используются его сертификаты. Сообщение Certificate обязательно высылается при использовании алгоритмов типа DH_fixed, T, DHE и T_PSK. Данное сообщение всегда следует сразу за сообщением ServerHello.

Сообщение Certificate определяется следующим образом:

```
struct {
  ASN.1Cert certificate_list<0..2^24-1>;
} Certificate;
opaque ASN.1Cert<1..2^24-1>;
```

Поле certificate_list структуры Certificate определяет последовательность (цепочку) сертификатов. Непосредственный сертификат сервера ДОЛЖЕН быть первым в списке. Каждый следующий сертификат в списке ДОЛЖЕН использоваться для проверки подписи предыдущего. Так как для признания сертификатов требуется, чтобы откры-

тые ключи корневых удостоверяющих центров распределялись отдельно, самоподписанный сертификат корневого удостоверяющего центра может не указываться в цепочке, при условии, что получатель его уже имеет и может проверить его подлинность.

К сертификатам, отправляемым сервером, применяются следующие правила:

– сертификаты ДОЛЖНЫ соответствовать СТБ 34.101.19;

– открытый ключ сертификата сервера (первый сертификат в списке) и связанные с ключом параметры ДОЛЖНЫ быть совместимы с выбранным алгоритмом формирования общего ключа.

Примечание — Для управления выбором сертификатов, могут использоваться расширения `server_name` и `trusted_ca_keys`, определенные в [10].

Если клиент выслал расширение `signature_algorithms` в своем сообщении `ClientHello`, то все сертификаты, предоставленные сервером, ДОЛЖНЫ быть подписаны с использованием пар «алгоритм хэширования, алгоритмы ЭЦП» из этого расширения. Пары МОГУТ быть различны и сертификат, содержащий открытый ключ одного алгоритма ЭЦП, может быть подписан с помощью другого алгоритма. Алгоритмы формирования общего ключа не накладывают ограничений на алгоритмы подписи сертификатов. Поэтому сертификат может быть подписан с помощью любой согласованной пары «алгоритм хэширования, алгоритмы ЭЦП», вне зависимости от назначения открытого ключа сертификата.

Если сервер имеет несколько сертификатов, то он выбирает один из них на основе приведенных выше правил. Могут использоваться дополнительные правила, учитывающие, например, сетевые адреса сторон протокола, локальную конфигурацию, настройки безопасности. Если сервер имеет только один сертификат, то ему СЛЕДУЕТ проверить, что сертификат удовлетворяет заданным правилам.

8.9 Сообщение `ServerKeyExchange`

Сообщение `ServerKeyExchange` сервер отправляет после сообщения `Certificate`, если согласование параметров связи выполняется с сертификатами, или после сообщения `ServerHello`, если сертификаты не используются. Сообщение `ServerKeyExchange` содержит криптографические данные (например, эфемерный открытый ключ), с помощью которых клиент может сформировать предварительный мастер-ключ `pre_master_secret`. Сообщение `ServerKeyExchange` должно отправляться тогда и только тогда, когда сообщение сервера `Certificate` содержит недостаточно данных для согласования `pre_master_secret` или это сообщение не отправлялось.

Сообщение `ServerKeyExchange` обязательно высылается при использовании алгоритмов DH_anon, DHE и DHE_PSK, может высылаться при использовании алгоритмов PSK и T_PSK и не высылается при использовании алгоритмов DH_fixed и T. Формат и содержание сообщения `ServerKeyExchange` определяется в зависимости от используемого алгоритма.

При определении криптонаборов, которые включают новые алгоритмы формирования общего ключа, ДОЛЖНО быть сказано, следует ли отправлять сообщение `ServerKeyExchange` и, если сообщение отправлять следует, ДОЛЖЕН быть определен его формат.

Если клиент выслал расширение `signature_algorithms` в своем сообщении `ClientHello`, а сервер при формировании криптографических данных в `ServerKeyExchange` подписывает их, то при выработке ЭЦП сервер ДОЛЖЕН использовать только те пары «алгоритм хэширования, алгоритмы ЭЦП», которые были

заданы в расширении. Могут быть противоречия между выбранными клиентом парами и выбранными им криптонаборами. Поэтому перед выбором криптонабора сервер ДОЛЖЕН проверять его соответствие присланному расширению `signature_algorithms`.

8.10 Сообщение `CertificateRequest`

Сервер, который предъявил свой сертификат, может запросить сертификат клиента, если это не противоречит правилам согласованного криптонабора. Запросом сертификата является сообщение `CertificateRequest`. Это сообщение сервер отправляет либо за `ServerKeyExchange`, либо за `ServerHello`, в зависимости от того, высылалось `ServerKeyExchange` или нет.

Сообщение `CertificateRequest` определяется следующим образом:

```
struct {
  ClientCertificateType certificate_types<1..2^8-1>;
  SignatureAndHashAlgorithm
    supported_signature_algorithms<2^16-1>;
  DistinguishedName certificate_authorities<0..2^16-1>;
} CertificateRequest;
opaque DistinguishedName<1..2^16-1>;
```

Поля структуры `CertificateRequest` имеют следующее значение:

– `certificate_types` — список методов аутентификации, т. е. типов сертификатов, которые клиенту предлагается предъявить (см. 6.3.2);

– `supported_signature_algorithms` — список пар «алгоритм хэширования, алгоритмы ЭЦП», которые сервер может использовать при проверке подписи сертификатов, в порядке убывания приоритета. Тип данного поля определяется в 6.3.3;

– `certificate_authorities` — список отличительных имен признаваемых удостоверяющих центров. Формат имен определен в [11]. Каждое имя в списке кодируется строкой байтов (октетов) по отличительным правилам, описанным в СТБ 34.101.19 (приложение Б). Элементы списка могут определять желательные корневые или промежуточные удостоверяющие центры и, таким образом, список задает пространство доверия. Если список пуст, то клиент МОЖЕТ отправить любой сертификат, тип которого указан в `certificate_types` (при условии, что у клиента нет дополнительных ограничений).

Поля `certificate_types` и `supported_signature_algorithms` структуры `CertificateRequest` связаны следующими правилами:

– каждый сертификат, предоставленный клиентом, ДОЛЖЕН быть подписан с использованием одной из пар «алгоритм хэширования, алгоритмы ЭЦП», указанной в поле `supported_signature_algorithms`;

– непосредственный сертификат клиента ДОЛЖЕН иметь тип, включенный в `certificate_types`. Если ключ сертификата является ключом подписи, то он ДОЛЖЕН быть пригоден к использованию с некоторой парой «алгоритм хэширования, алгоритмы ЭЦП» из поля `supported_signature_algorithms`. Тип сертификата не ограничивает алгоритм, используемый для подписи сертификата.

Примечание — Если сервер, который не предъявил свой сертификат, запрашивает сертификат клиента, то это считается ошибкой типа `handshake_failure` (см. 10.3).

8.11 Сообщение `ServerHelloDone`

Сообщение `ServerHelloDone` сервер отправляет для того, чтобы известить клиента о завершении пересылки `ServerHello` и других сообщений, нужных для формирования

общего ключа. Сообщение означает, что клиент может приступить к выполнению своей части алгоритма формирования общего ключа. После отправки `ServerHelloDone` сервер ожидает ответ клиента.

Получив сообщение `ServerHelloDone`, клиенту СЛЕДУЕТ проверить, что сервер представил действительный сертификат (если сертификат предоставлялся) и параметры, указанные сервером в `ServerHello`, приемлемы.

Сообщение `ServerHelloDone` определяется следующим образом:

```
struct {} ServerHelloDone;
```

8.12 Сообщение клиента `Certificate`

Сообщение клиента `Certificate` является первым сообщением, которое клиент посылает серверу после получения от него сообщения `ServerHelloDone`. Сообщение Certificate высылается только тогда, когда сервер запросил сертификат клиента. Если у клиента нет подходящих сертификатов, то он ДОЛЖЕН отправить сообщение `Certificate` без сертификатов (поле `certificate_list` этого сообщения имеет нулевую длину).

Если сообщение клиента `Certificate` не содержит сертификатов или некоторый сертификат из списка неприемлем (например, был подписан недоверенным удостоверяющим центром), то сервер МОЖЕТ, по своему усмотрению, либо продолжить обмен сообщениями без аутентификации клиента, либо ответить сигнальным сообщением `handshake_failure`.

Формат сообщения `Certificate` определен в 8.8. В этом сообщении клиент передает серверу цепочку сертификатов. Непосредственный сертификат клиента ДОЛЖЕН быть первым в списке. Сервер использует переданную цепочку при верификации сообщения `CertificateVerify` (например, если используется алгоритм формирования общего ключа типа DHE), либо при вычислении предварительного мастер-ключа (если используется алгоритм формирования общего ключа типа DH_fixed).

К сертификатам, отправляемым клиентом, применяются следующие правила:

– сертификаты ДОЛЖНЫ соответствовать СТБ 34.101.19;

– сертификат клиента (первый сертификат в списке) ДОЛЖЕН иметь один из типов, перечисленных в сообщении `CertificateRequest`;

– сертификат клиента ДОЛЖЕН соответствовать алгоритму формирования общего ключа из согласованного криптонабора и всем согласованным расширениям;

– если список `certificate_authorities` в сообщении `CertificateRequest` не был пустым, то некоторому сертификату в цепочке СЛЕДУЕТ быть выпущенным одним из перечисленных удостоверяющих центров;

– сертификаты ДОЛЖНЫ быть подписаны с использованием подходящей пары «алгоритм хэширования, алгоритмы ЭЦП», как это описано в 8.7.

8.13 Сообщение `ClientKeyExchange`

Сообщение `ClientKeyExchange` всегда отправляется клиентом. Если клиент отправляет сообщение `Certificate`, то сразу за ним он ДОЛЖЕН отправить `ClientKeyExchange`. В противном случае, `ClientKeyExchange` ДОЛЖНО быть первым сообщением, отправленным клиентом после того, как он получил `ServerHelloDone`.

Сообщение `ClientKeyExchange` используется сторонами для завершения процесса формирования предварительного мастер-ключа `pre_master_secret`. Этот ключ может формировать одна сторона (и передавать его другой в зашифрованном виде), либо обе стороны могут участвовать в формировании `pre_master_secret`.

Формат `ClientKeyExchange` определяется в зависимости от используемого алгоритма формирования общего ключа.

8.14 Сообщение `CertificateVerify`

Сообщение `CertificateVerify` используется для явной верификации сертификата клиента. Оно посылается только в том случае, когда клиент представил свой сертификат и открытый ключ этого сертификата может использоваться в алгоритмах ЭЦП. Сообщение `CertificateVerify` отправляется сразу после сообщения `ClientKeyExchange`.

Сообщение `CertificateVerify` определяется следующим образом:

```
struct {
  digitally-signed struct {
    opaque handshake_messages[handshake_messages_length];
  }
} CertificateVerify;
```

Подписанное поле `handshake_messages` содержит все сообщения протокола Handshake, отосланные и полученные клиентом, начиная с сообщения `ClientHello` и заканчивая сообщением, которое предшествует `CertificateVerify`. Должны учитываться все поля этих сообщений, в том числе поля типа и длины. Другими словами, поле `handshake_messages` представляет собой конкатенацию всех структур протокола Handshake, которыми стороны обменялись до отправки сообщения `CertificateVerify`.

Алгоритмы хэширования и ЭЦП, используемые для формирования подписи, ДОЛЖНЫ быть указаны в поле `supported_signature_algorithms` сообщения `CertificateRequest`. Кроме того, алгоритмы хэширования и ЭЦП ДОЛЖНЫ быть согласованы с открытым ключом сертификата клиента.

Для того, чтобы сформировать поле `handshake_messages` стороны должны либо сохранять обработанные сообщения вплоть до `CertificateVerify`, либо обрабатывать их последовательно с помощью потенциально возможных алгоритмов хэширования. Серверы могут минимизировать вычислительные затраты, предложив ограниченный набор пар «алгоритм хэширования, алгоритмы ЭЦП» в сообщении `CertificateRequest`.

8.15 Сообщение `Finished`

Сообщение `Finished` всегда высылается сразу после сообщения `ChangeCipherSpec` для подтверждения того, что аутентификация и формирование общего ключа прошли успешно. Существенно, что сообщение `ChangeCipherSpec` будет получено между другими сообщениями протокола Handshake и сообщением `Finished`. Поэтому `Finished` является первым сообщением, которое защищено с использованием только что согласованных алгоритмов и ключей. Если сообщению `Finished` не предшествует сообщение `ChangeCipherSpec`, то это является ошибкой.

Получатель сообщения `Finished` ДОЛЖЕН проверить, что его содержимое корректно. Как только одна из сторон отправила свое сообщение `Finished` и подтвердила правильность полученного от другой стороны сообщения `Finished`, она может начать отправлять и получать данные прикладных протоколов в установленном соединении.

Сообщение `Finished` определяется следующим образом:

```
struct {
  opaque verify_data[verify_data_length];
} Finished;
```

Поле verify_data длины verify_data_length формируется с помощью алгоритма генерации псевдослучайных чисел по правилу:

```
verify_data = PRF(master_secret, finished_label,
  Hash(handshake_messages))[0..verify_data_length-1],
```

где параметры имеют следующее значение:

– finished_label — строка "client finished" для сообщения Finished, отправленного клиентом, и строка "server finished" для сообщения Finished, отправленного сервером;

– handshake_messages — все данные сообщений протокола Handshake (без сообщения HelloRequest), отправленных и полученных стороной, не включая само сообщение Finished. В handshake_messages содержатся только данные, видимые на уровне протокола Handshake (без заголовков протокола Record). Другими словами, handshake_messages представляет собой конкатенацию всех структур протокола Handshake, которыми стороны обменялись к данному моменту;

– Hash(handshake_messages) — хэш-значение, вычисленное от handshake_messages с помощью алгоритма хэширования, используемого в алгоритме генерации псевдослучайных чисел. Если криптонабор определяет свой собственный алгоритм генерации псевдослучайных чисел, отличный от алгоритма из 6.2.3, то в криптонаборе ДОЛЖЕН быть определен алгоритм хэширования, который используется при формировании сообщения Finished.

Длина поля verify_data определяется в зависимости от криптонабора. В криптонаборе, который явно не определяет значение verify_data_length, это значение принимается равным 12. При разработке криптонаборов МОГУТ определяться другие длины поля verify_data, при этом длина ДОЛЖНА быть не меньше 12 байтов.

Значение handshake_messages содержит в себе все сообщения протокола Handshake, начиная с сообщения ClientHello до сообщения Finished, не включая последнее. Данное значение может отличаться от значения handshake_messages в сообщении CertificateVerify, так как handshake_messages в сообщении Finished может дополнительно содержать сообщение CertificateVerify. Кроме этого, handshake_messages в сообщении Finished клиента будет отличаться от handshake_messages в сообщении Finished сервера, так как в сообщении сервера будет учтено сообщение Finished клиента.

Сообщения ChangeCipherSpec, сигнальные сообщения об ошибках и любые другие сообщения, которые не являются сообщениями протокола Handshake, не учитываются при вычислении хэш-значений при формировании verify_data. При вычислении хэш-значений опускаются также сообщения HelloRequest.

8.16 Вычисление мастер-ключа

Мастер-ключ master_secret определяется по pre_master_secret, ClientHello.random и ServerHello.random с помощью алгоритма генерации псевдослучайных чисел, определенного в 6.2.3:

```
master_secret = PRF(pre_master_secret, "master secret",
  ClientHello.random + ServerHello.random)[0..47];
```

Длина pre_master_secret может зависеть от используемого алгоритма формирования общего ключа, но мастер-ключ всегда состоит из 48 байтов.

Ключ `pre_master_secret` должен быть уничтожен после того, как `master_secret` вычислен.

9 Протокол Change Cipher Spec

Протокол Change Cipher Spec предназначен для оповещения о смене криптоопределений и ключей. В этом протоколе используется единственное сообщение `ChangeCipherSpec`, которое защищается и сжимается с использованием текущего состояния соединения. Сообщение состоит из одного байта со значением 1:

```
struct {
  enum {change_cipher_spec(1), (255)} type;
} ChangeCipherSpec;
```

Сообщение `ChangeCipherSpec` может отправлять как клиент, так и сервер. Данное сообщение уведомляет принимающую сторону о том, что последующие фрагменты данных будут защищены с использованием вновь согласованных криптоопределения и ключей. Сообщение `ChangeCipherSpec` отправляется во время установки связи после того, как согласованы параметры защиты, но перед отправкой сообщения `Finished`.

При получении данного сообщения ожидаемое состояние чтения ДОЛЖНО быть переведено в активное состояние чтения. Сразу после отправки этого сообщения ожидаемое состояние записи ДОЛЖНО быть переведено в активное состояние записи (см. 7.2).

Если во время передачи данных по соединению инициируется переустановка связи, то взаимодействующие стороны могут продолжить обмен данными, используя старое криптоопределение. Однако сразу после отправки сообщения `ChangeCipherSpec` отправитель ДОЛЖЕН использовать новое криптоопределение. Так как при получении сообщения `ChangeCipherSpec` принимающей стороне требуется время для вычисления новых ключей, то МОЖЕТ существовать определенный временной интервал, в течение которого получатель должен буферизировать данные. На практике, как правило, этот интервал довольно короткий.

10 Протокол Alert

10.1 Сигнальные сообщения

Протокол Record поддерживает передачу сигнальных сообщений. Эти сообщения формируются протоколом Alert и передают уровень сигнала (предупреждение или критическая ошибка) и его описание. Сигнальные сообщения, соответствующие критическим ошибкам, приводят к немедленному прерыванию соединения. В этом случае другие соединения данного сеанса могут продолжаться, но идентификатор сеанса ДОЛЖЕН быть признан недействительным, чтобы предотвратить использование сеанса, в котором возникла ошибка, для возобновления или переустановки связи. Как и другие сообщения, сигнальные сообщения защищаются и сжимаются в соответствии с текущим состоянием соединения.

Сообщения протокола Alert описываются следующим образом:

```
struct {
  AlertLevel level;
  AlertDescription description;
} Alert;
```

Поля структуры `Alert` имеют следующее значение:

– `level` — уровень сигнального сообщения (предупреждение или критическая ошибка);

– `description` — описание сигнального сообщения.

Типы вложенных в `Alert` полей определяются следующим образом:

```
enum {warning(1), fatal(2), (255)} AlertLevel;
enum {
  close_notify(0),
  unexpected_message(10),
  bad_record_mac(20),
  record_overflow(22),
  decompression_failure(30),
  handshake_failure(40),
  bad_certificate(42),
  unsupported_certificate(43),
  certificate_revoked(44),
  certificate_expired(45),
  certificate_unknown(46),
  illegal_parameter(47),
  unknown_ca(48),
  access_denied(49),
  decode_error(50),
  decrypt_error(51),
  protocol_version(70),
  insufficient_security(71),
  internal_error(80),
  user_canceled(90),
  no_renegotiation(100),
  unsupported_extension(110),
  unknown_psk_identity(115)
  (255)
} AlertDescription;
```

Элемент `warning` перечисления `AlertLevel` соответствует предупредительному сигнальному сообщению, элемент `fatal` — критическому. Элементы перечисления `AlertDescription` описываются в 10.2 и 10.3.

10.2 Сообщения о закрытии соединения

Обе стороны, клиент и сервер, должны знать, что соединение закрывается, чтобы избежать ситуации, в которой одна сторона считает, что соединение закрыто, а вторая сторона считает наоборот.

Каждая из сторон может инициировать закрытие соединения, отправив предупредительное сигнальное сообщение `close_notify`. Это сообщение извещает получателя о том, что отправитель больше не будет посылать сообщения в данном соединении. Любые данные, полученные после этого сигнального сообщения, игнорируются.

При получении сообщения о закрытии соединения получатель ДОЛЖЕН ответить своим сигнальным сообщением `close_notify` и немедленно закрыть соединение, отбросив любые задержанные сообщения. От инициатора закрытия соединения не требуется ожидать ответное сообщение `close_notify`, он может сразу прекратить прием данных.

При закрытии соединения передача сообщения `close_notify` является обязательной для каждой из сторон, если они до этого не получали критических сигнальных сообщений.

Если прикладной протокол предусматривает передачу данных после закрытия соединения TLS, то реализация TLS должна дождаться ответного сообщения `close_notify` и проинформировать прикладной протокол о том, что соединение закрыто. Если же передача данных после завершения соединения не предусмотрена, то реализация TLS МОЖЕТ закрыть соединение, не дожидаясь `close_notify`. В настоящем стандарте не определяются правила использования соединений TLS прикладными протоколами, в том числе правила открытия и закрытия соединений.

10.3 Сообщения об ошибках

Сторона, обнаружившая ошибку во время выполнения TLS, должна отправить другой стороне соответствующее сигнальное сообщение. При передаче или получении критического сигнального сообщения, обе стороны должны немедленно закрыть соединение. Стороны НЕ ДОЛЖНЫ использовать идентификаторы сеанса и ключи, связанные с соединением, закрытым из-за ошибки. Таким образом, возобновлять связь после критических ошибок НЕЛЬЗЯ.

Всякий раз, когда при выполнении TLS возникает критическая ошибка, противоположной стороне до закрытия соединения ДОЛЖНО быть отправлено соответствующее сигнальное сообщение. Для ошибки, уровень соответствующего сигнального сообщения которой явно не определен, отправитель МОЖЕТ, по своему усмотрению, считать сигнальное сообщение критическим или предупредительным. Однако, если отправитель намеревается закрыть соединение сразу после передачи сигнального сообщения, то он ДОЛЖЕН выбрать критический уровень.

Если стороны отправляют или принимают предупредительное сигнальное сообщение, то соединение может продолжаться. Если принимающая сторона решает не продолжать соединение (например, после получения сигнального сообщения `no_renegotiation`), то ей СЛЕДУЕТ отправить другой стороне критическое сигнальное сообщение, чтобы прервать соединение. Отправляющая сторона может не знать, как принимающая сторона будет реагировать на предупредительное сигнальное сообщение. Поэтому, если у одной из сторон возникает ошибка, которая трактуется как предупреждение, и эта сторона желает продолжить соединение, то соответствующее сигнальное сообщение может не высылаться. Например, если одна из сторон решает принять недействительный сертификат другой стороны (возможно, после подтверждения принятия конечным пользователем) и продолжить соединение, то ей не следует посылать сигнальное сообщение `certificate_expired`.

Определены следующие сигнальные сообщения:

– `unexpected_message` — получено некорректное сообщение. Сообщение всегда является критическим. Оно никогда не будет выслано при взаимодействии корректных реализаций TLS;

– `bad_record_mac` — получен фрагмент с некорректным значением имитовставки. Данное сообщение означает также, что фрагмент расшифрован неверно (длина зашифрованных данных не кратна длине блока алгоритма шифрования или дополнение при проверке оказалось некорректным). Сообщение всегда является критическим. Оно никогда не будет выслано при взаимодействии кор-ректных реализаций TLS (при условии, что в канале связи нет помех);

– `record_overflow` — получен фрагмент `TLSCiphertext.fragment`, длина которого превышает $2^{14} + 2048$ байтов, или после расшифрования которого длина `TLSCompressed .fragment` превышает $2^{14} + 1024$ байтов. Сообщение всегда является критическим. Оно

никогда не будет выслано при взаимодействии корректных реализаций TLS (при условии, что в канале связи нет помех);

– `decompression_failure` — алгоритм восстановления сжатых данных получил некорректные входные данные (например, при восстановлении получены данные, объем которых превышает допустимое значение). Сообщение всегда является критическим. Оно никогда не будет выслано при взаимодействии корректных реализаций TLS (при условии, что в канале связи нет помех);

– `handshake_failure` — отправитель не смог согласовать приемлемый набор параметров связи. Сообщение всегда является критическим;

– `bad_certificate` — сертификат поврежден, содержит некорректную ЭЦП и т. д.;

– `unsupported_certificate` — тип сертификата не поддерживается;

– `certificate_revoked` — сертификат отозван выпустившей его стороной;

– `certificate_expired` — срок действия сертификата истек или еще не наступил;

– `certificate_unknown` — при обработке сертификата возникла ошибка, не определенная выше, которая не позволяет использовать сертификат;

– `illegal_parameter` — в сообщении протокола Handshake значение некоторого поля лежит вне допустимого диапазона или не согласуется со значениями других полей. Сообщение всегда является критическим;

– `unknown_ca` — получена достоверная цепочка сертификатов или ее часть, но сертификат не был признан действительным, поскольку сертификат соответствующего удостоверяющего центра не был найден или не соответствует имеющемуся сертификату надежного удостоверяющего центра. Сообщение всегда является критическим;

– `access_denied` — был получен действительный сертификат, но при проверке прав доступа отправитель решил не продолжать согласование. Сообщение всегда является критическим;

– `decode_error` — сообщение не может быть декодировано, так как значение одного из полей лежит вне допустимого диапазона или длина сообщения некорректна. Сообщение всегда является критическим. Оно никогда не будет выслано при взаимодействии корректных реализаций TLS (при условии, что в канале связи нет помех);

– `decrypt_error` — неудачное завершение некоторой криптографической операции во время выполнения Handshake, в том числе некорректное завершение алгоритма проверки ЭЦП или ошибка при проверке правильности сообщения `Finished`. Сообщение всегда является критическим;

– `protocol_version` — номер версии протокола, предложенный клиентом для согласования, не поддерживается. Сообщение всегда является критическим;

– `insufficient_security` — это сообщение используется вместо `handshake_failure` в тех случаях, когда процесс согласования параметров связи завершился неудачно по причине того, что сервер потребовал использования более надежных параметров, чем те, которые поддерживает клиент. Сообщение всегда является критическим;

– `internal_error` — внутренняя ошибка, которая не связана с противоположной стороной или логикой протокола (например, ошибка выделения памяти), которая делает невозможным дальнейшее выполнение протокола. Сообщение всегда является критическим;

– `user_canceled` — выполнение протокола Handshake остановлено по причине, не связанной с логикой протокола. Если пользователь останавливает выполнение после того, как протокол завершен, то рекомендуется закрыть соединение отправкой сигнального сообщения `close_notify`. Это сообщение должно следовать за сообщением `user_canceled`. Сообщение `user_canceled` обычно является предупредительным;

– `no_renegotiation` — сообщение отсылается клиентом в ответ на запрос сервера `HelloRequest` или сервером в ответ на запрос клиента `ClientHello` после установки связи в тех случаях, когда получатель запроса не желает переустанавливать связь. После получения сообщения запрашивающая сторона может отказаться от переустановки связи. Например, этой стороной может быть сервер, на котором уже запущена программа, ожидающая переустановку, эта программа получила параметры защиты, и ход ее выполнения не может быть изменен. Сообщение `no_renegotiation` всегда является предупредительным;

– `unsupported_extension` — это сообщение высылает клиент, который получает сообщение сервера `ServerHello` с расширением, не включенным клиентом в соответствующее сообщение `ClientHello`. Сообщение всегда является критическим;

– `unknown_psk_identity` — это сообщение высылает сервер, если он не может определить предварительно распределенный секрет по полученному идентификатору (см. 6.2.4). Сообщение всегда является критическим.

Приложение А
(обязательное)
Трактование ключевых слов

В настоящем приложении приводится разъяснение значений ключевых слов «ДОЛЖЕН», «НЕЛЬЗЯ», «СЛЕДУЕТ», «НЕ СЛЕДУЕТ», «РЕКОМЕНДУЕТСЯ» и «МОЖЕТ» используемых в настоящем стандарте.

Ключевое слово «ДОЛЖЕН» означает, что действия, к которым применено данное ключевое слово, необходимо в точности выполнять.

Ключевое слово «НЕЛЬЗЯ» выражает абсолютный запрет на выполнение соответствующих действий.

Ключевые слова «СЛЕДУЕТ» и «РЕКОМЕНДУЕТСЯ» необходимо понимать так, что в некоторых случаях существует реальная причина их игнорировать, но последствия таких действий должны быть очевидными и хорошо взвешенными.

Ключевое слово «НЕ СЛЕДУЕТ» употребляется в тех случаях, когда действие, к которому применено данное ключевое слово, будет в некоторых случаях правильным и даже полезным, однако при этом его последствия должны быть очевидными и хорошо взвешенными.

Ключевое слово «МОЖЕТ» применяется к действиям (предметам), выполнение или невыполнение (наличие или отсутствие) которых не влияет на ситуацию в целом. Это означает, что программы, работающие с чем-то, помеченным данными ключевыми словами, должны учитывать обе ситуации и корректно их обрабатывать.

Данные ключевые слова введены, в первую очередь, для выражения требований к действиям, которые влияют на безопасность и надежность рассматриваемых объектов, а также в интересах унификации последних.

Приложение Б
(обязательное)
Правила описания данных

Б.1 Общие соглашения

Комментарии начинаются с символов «`/*`» и заканчиваются символами «`*/`».

Необязательные компоненты заключаются в двойные квадратные скобки: «`[[ ]]`».

Тип `opaque` описывает один байт неструктурированных данных.

Число `a` в степени `b` обозначается через `a^b`.

Б.2 Данные

Данные, которыми обмениваются стороны протокола TLS, представляют собой структурированные объединения элементов. Элементы данных и структуры данных в целом представляют собой строки (последовательности) байтов. Байты записываются слева направо и сверху вниз. Например, в строке `{1, 2, 3, 4}` первым байтом является 1, последним — 4. Длина строки `data` обозначается `len(data)`.

Байты строки нумеруются, начиная с 0. Для выделения диапазона байтов с номерами от `lo` до `hi` можно использовать выражение `[lo..hi]`. Например, строка `{1, 2, 3, 4}[1..3]` состоит из байтов 2, 3, 4.

Символом «`+`» обозначается конкатенация (объединение) строк. Например, `{1, 2} + {3, 4}` есть `{1, 2, 3, 4}`.

Строка, составленная из байтов со значениями от 32 до 126, может представляться строкой символов. Соответствие между символами и байтами задает ГОСТ 27463. Например, строка символов `"slithy toves"` представляет строку байтов `{115, 108, 105, 116, 104, 121, 32, 116, 111, 118, 101, 115}`.

Б.3 Векторы

Вектор — это одномерный массив однотипных элементов. Длина вектора может быть определена заранее, либо может определяться в момент выполнения протокола. В любом случае, длина определяет количество байтов, а не количество элементов в векторе. Для определения нового типа `T'`, представляющего собой вектор фиксированной длины с элементами типа `T`, используется синтаксис:

```
T T'[n];
```

Здесь `T'` занимает `n` байтов, где `n` кратно длине элемента типа `T`. Длина вектора не записывается в поток данных.

В следующем примере тип `Datum` определяет тройку неструктурированных байтов, а тип `Data` — тройку последовательных элементов типа `Datum`, которые в сумме занимают девять байтов:

```
opaque Datum[3];
Datum Data[9];
```

Векторы переменной длины определяются путем задания диапазона допустимых длин (включая границы) с помощью обозначения `<floor..ceiling>`:

```
T T'<floor..ceiling>;
```

Число `floor` задает минимальную длину, а `ceiling` — максимальную.

При передаче вектора переменной длины в поток данных записывается сначала фактическая длина вектора, а затем его содержимое. Длина представляется числом, занимающим столько байтов, сколько требуется для определения максимальной длины (т. е. значения `ceiling`). Вектор переменной длины, в котором поле фактической длины равняется нулю, соответствует пустому вектору.

В следующем примере вектор типа `mandatory` содержит от 300 до 400 байтов, а вектор типа `longer` содержит до 800 байтов или до 400 элементов типа `uint16`:

```
opaque mandatory<300..400>;
uint16 longer<0..800>;
```

Вектор `mandatory` не может быть пустым. Поле его фактической длины занимает два байта (`uint16`), которых достаточно для представления значения 400 (см. Б.5). Представление вектора `longer` будет включать двухбайтовое поле фактической длины, предшествующее вектору. Длина вектора должна быть кратна длине его элементов (например, 17-байтовый вектор из элементов типа `uint16` недопустим).

Б.4 Числа

Элементами данных могут быть неотрицательные целые числа. Тип `uint8` представляет беззнаковый байт и является базовым числовым типом данных. Все числа, которые не укладываются в байт, представляются вектором беззнаковых байтов фиксированной длины. При этом первый байт считается старшим, последний — младшим. Такой порядок байтов называется сетевым или от «старших к младшим» (big-endian). Например, строке байтов `{1, 2, 3, 4}` соответствует число $16909060 = 1 * 2^{24} + 2 * 2^{16} + 3 * 2^8 + 4$.

Следующие числовые типы предопределены:

```
uint8 uint16[2];
uint8 uint24[3];
uint8 uint32[4];
uint8 uint64[8];
```

Если неотрицательное целое число необходимо представить вектором с элементами типа opaque, то число представляются как беззнаковое (без дополнительных нулевых старших байтов даже в тех случаях, когда установлен старший бит числа).

Б.5 Перечисления

Тип `enum` поддерживает перечисления. Переменная типа `enum` может принимать только те значения, которые объявлены в определении перечисления. Каждое такое определение задает новый тип. Только переменные одного и того же перечислительного типа могут присваиваться друг другу или сравниваться между собой. Элементам перечисления должны назначаться числовые значения:

```
enum {e1(v1), e2(v2) [[,...]] [[, (n)]]} Te;
```

Здесь `e1`, `e2`,... — имена элементов, `v1`, `v2`,... — назначенные значения. Так как элементы перечисления не упорядочены, им могут назначаться любые уникальные значения и в любом порядке.

Если в определении перечисления есть многоточие, то перечисление может дополняться новыми элементами. Добавление элементов обозначается следующим образом:

```
Te += {e3(v3), e4(v4)};
```

Примечание — Возможность доопределения перечислительного типа является единственным расширением правил описания данных, заданных в [3].

Для перечислений, которые никогда не преобразовываются во внешние представления, числовые значения могут опускаться. Например,

```
enum {low, medium, high} Amount;
```

Перечисление занимает столько байтов в потоке данных, сколько необходимо для записи максимального из значений его элементов. Элементы следующего перечисления занимают один байт:

```
enum {red(3), blue(5), white(7)} Color;
```

Для задания количества байтов, которое будет занимать переменная перечислительного типа, можно указывать в перечислении элементы без имени. В следующем примере переменные типа Taste будут занимать два байта в потоке данных, хотя могут принимать только значения 1, 2 или 4:

```
enum {sweet(1), sour(2), bitter(4), (32000)} Taste;
```

Областью видимости имен элементов перечисления является сам перечислительный тип. При ссылке на имя можно указывать этот тип явно или опускать его, если он ясен из контекста:

```
Color color = Color.blue;      /* задано явно */
Color color = blue;            /* определяется неявно */
```

Б.6 Структурированные типы и варианты

Примитивные типы могут объединяться в структурированный тип (структуру). Структуры могут содержать вложенные структуры. Каждое определение структуры задает новый уникальный тип. Синтаксис определения структур аналогичен синтаксису языка Си:

```
struct {
  T1 f1;
T2 f2;
...
Tn fn;
} [[T]];
```

При ссылке на элемент структуры указывается тип и имя поля (вложенного элемента). Например, `T.f2` — ссылка на второе поле структуры, определенной выше.

Структуры могут содержать варианты, выбор в которых производится на основании некоторой переменной, доступной внутри структуры. На выбор варианта указывает ключевое слово `select`. Механизм разбора вариантов во время выполнения TLS не задается настоящими правилами и определяется в конкретной реализации TLS.

Переменная, на основании которой производится выбор варианта, должна иметь перечислительный тип. Элементы этого типа определяют возможные варианты определения структуры. Для каждого элемента должен быть указан вариант выбора. Варианты могут объединяться: например, если два элемента следуют сразу друг за другом, то им соответствует один и тот же вариант.

Для определения вариантов используется следующий синтаксис:

```
struct {
  T1 f1;
  T2 f2;
  ...
  Tn fn;
  select (E) {
    case e1: Te1;
    case e2: Te2;
    case e3: case e4: Te3;
    ...
    case en: Ten;
  } [[fv]];
} [[Tv]];
```

Здесь `E` — перечислительный тип. Для ссылки на вариант ему может быть присвоена метка `fv`.

В следующем примере за выбор варианта отвечают переменные типа `VariantTag`:

```
enum {apple, orange, banana} VariantTag;
struct {
  uint16 number;
  opaque string<0..10>; /* переменная длина */
} V1;
struct {
  uint32 number;
  opaque string[10];    /* фиксированная длина */
} V2;
struct {
  select (VariantTag) { /* значение задано неявно */
  case apple:
    V1;                 /* apple */
  case orange:
  case banana:
    V2;                 /* orange или banana */
  } variant_body;       /* необязательная метка */
} VariantRecord;
```

Б.7 Криптографические атрибуты

Для указания на криптографическую обработку данных используются специальные атрибуты, которые предшествуют описанию типа данных. Ключи, которые используются при криптографической обработке, определяются текущим состоянием соединения TLS (см. 7.2).

Атрибут `digitally-signed` используется для обозначения выходных данных алгоритмов выработки ЭЦП. Элемент с атрибутом `digitally-signed` преобразуется в структуру `DigitallySigned`:

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

Здесь поле `algorithm` определяет используемую при выработке ЭЦП пару «алгоритм хэширования, алгоритмы ЭЦП» (см. 8.7). Поле `signature` определяет значение ЭЦП, выработанное от содержимого элемента. Само содержимое элемента не включается в структуру. Длина подписи определяется используемым алгоритмом выработки ЭЦП и его ключами.

Атрибут `stream-ciphered` (`block-ciphered`) указывает, что данные защищены с помощью алгоритмов поточного (блочного) шифрования. Длина данных, которые зашифровываются блочно, должна быть кратна длине блока.

Атрибут `aead-ciphered` указывает, что данные зашифрованы и дополнены имитовставкой. Обработка данных выполняется с помощью алгоритмов одновременного шифрования и имитозащиты. Входные данные алгоритма могут иметь произвольную длину.

Атрибут `public-key-encrypted` используется для обозначения выходных данных алгоритмов шифрования с открытым ключом. Данные зашифровываются таким образом, что расшифровать их можно только на соответствующем личном ключе. Элемент с атрибутом `public-key-encrypted` преобразуется в следующий вектор переменной длины:

```
opaque vector <0..2^16-1>;
```

Фактическая длина вектора определяется используемым алгоритмом шифрования и его ключами.

Рассмотрим пример использования крипографических атрибутов:

```
stream-ciphered struct {
  uint8 field1;
  uint8 field2;
  digitally-signed struct {
    uint8 field3<0..255>;
    uint8 field4;
  };
} UserType;
```

В этом примере содержимое внутренней структуры (поля `field3` и `field4`) используется в качестве входных данных для пары «алгоритм хэширования, алгоритмы ЭЦП», т. е. от полей `field3` и `field4` вырабатывается ЭЦП. После этого вся структура `UserType` зашифровывается поточно. Длина зашифрованной структуры будет равна сумме длин полей `field1` и `field2` (два байта), поля `algorithm`, определяющего используемые алгоритмы хэширования и ЭЦП (два байта), поля с длиной ЭЦП (два байта), а также поля, содержащего ЭЦП. Длина ЭЦП известна, так как известны используемый алгоритм выработки ЭЦП и его ключи.

Б.8 Константы

Элементам данных могут назначаться значения. Для этого должны использоваться типизированные константы. Например,

```
struct {
  uint8 f1;
  uint8 f2;
} Example1;
Example1 ex1 = {1, 4};  /* f1 = 1, f2 = 4 */
```

Переменным некоторых типов значения назначены быть не могут. К таким типам относятся opaque, структуры, содержащие поля типа opaque, векторы переменной длины.

Приложение В
(обязательное)
Криптонаборы семейства BIGN_WITH_BELT

В настоящем приложении определяются криптонаборы и методы аутентификации сторон протокола TLS, основанные на криптографических алгоритмах СТБ 34.101.31 и СТБ 34.101.45.

В.1 Криптонаборы

Определяемые криптонаборы перечислены в таблице В.1.

Таблица В.1 — Криптонаборы семейства BIGN_WITH_BELT

Криптонабор	Идентификатор	Алгоритм формирования общего ключа
TLS_DHE_BIGN_WITH_BELT_CTR_MAC_HBELT	{192, 21}	DHE_BIGN
TLS_DHE_BIGN_WITH_BELT_DWP_HBELT	{192, 22}	DHE_BIGN
TLS_DHT_BIGN_WITH_BELT_CTR_MAC_HBELT	{192, 23}	DHT_BIGN
TLS_DHT_BIGN_WITH_BELT_DWP_HBELT	{192, 24}	DHT_BIGN
TLS_DHE_PSK_BIGN_WITH_BELT_CTR_MAC_HBELT	{192, 25}	DHE_PSK_BIGN
TLS_DHE_PSK_BIGN_WITH_BELT_DWP_HBELT	{192, 26}	DHE_PSK_BIGN
TLS_DHT_PSK_BIGN_WITH_BELT_CTR_MAC_HBELT	{192, 27}	DHT_PSK_BIGN
TLS_DHT_PSK_BIGN_WITH_BELT_DWP_HBELT	{192, 28}	DHT_PSK_BIGN

Криптонабор TLS_DHE_BIGN_WITH_BELT_CTR_MAC_HBELT является обязательным в семействе BIGN_WITH_BELT.

Во всех криптонаборах используется один и тот же алгоритм генерации псевдослучайных чисел (см. В.2.2). Отличаются алгоритмы шифрования и имитозащиты (см. В.2.1), алгоритмы формирования общего ключа (см. В.2.3).

В.2 Алгоритмы криптонаборов

В.2.1 Алгоритмы шифрования и имитозащиты

В.2.1.1 Шифрование в режиме счетчика

Если в название криптонабора входит строка _CTR, то должен использоваться алгоритм шифрования в режиме счетчика, определенный в СТБ 34.101.31 (пункт 6.5). Этот алгоритм включается в перечисление `BulkCipherAlgorithm` под именем `belt_ctr`:

```
BulkCipherAlgorithm += {belt_ctr};
```

Алгоритм классифицируется как алгоритм поточного шифрования (`stream`) и действуют правила, заданные в 7.3.3.1.

Ключ шифрования (`client_write_key` или `server_write_key`) вырабатывается в процессе выполнения протокола Handshake. Длина ключа (`SecurityParameters.enc_key_length`) ДОЛЖНА равняться 32.

В качестве синхропосылки ДОЛЖЕН использоваться порядковый номер `seq_num`, определенный в 7.2, 7.3.3.1. Порядковый номер состоит из 8 байтов. Для получения синхропосылки данный номер ДОЛЖЕН быть дополнен 8 нулевыми байтами.

B.2.1.2 Имитозащита

Если в название криптонабора входит строка _MAC, то должен использоваться алгоритм выработки имитовставки, определенный в СТБ 34.101.31 (пункт 6.6). Этот алгоритм включается в перечисление `MACAlgorithm` под именем `belt_mac`:

```
MACAlgorithm += {belt_mac};
```

Ключ имитозащиты вырабатывается в процессе выполнения протокола Handshake. Длина ключа (`SecurityParameters.mac_key_length`) ДОЛЖНА равняться 32.

Имитовставка добавляется к открытым данным перед их зашифрованием (см. 7.3.3.1, 7.3.3.2). Длина имитовставки (`SecurityParameters.mac_lenght`) равняется 8.

B.2.1.3 Одновременное шифрование и имитозащита

Если в название криптонабора входит строка _DWP, то должны использоваться алгоритмы одновременного шифрования и имитозащиты данных, определенные в СТБ 34.101.31 (пункт 6.7). Эти алгоритмы включаются в перечисление `BulkCipherAlgorithm` под именем `belt_dwp`:

```
BulkCipherAlgorithm += {belt_dwp};
```

Алгоритмы классифицируются как `aead` и действуют правила, заданные в 7.3.3.3.

Ключ защиты (`client_write_key` или `server_write_key`) вырабатывается в процессе выполнения протокола Handshake. Длина ключа (`enc_key_length`) ДОЛЖНА равняться 32.

Неявная часть синхропосылки (`client_write_IV` или `server_write_IV`) вырабатывается в процессе выполнения протокола Handshake. Длина неявной части (`SecurityParameters.fixed_iv_length`) равняется 8. Явная часть синхропосылки передается вместе с данными в поле `GenericAEADCipher.nonce_explicit`. Длина явной части (`SecurityParameters.record_iv_length`) также равняется 8. В рамках одного соединения все явные части синхропосылок ДОЛЖНЫ различаться. Явная часть МОЖЕТ совпадать с порядковым номером `seq_num`, определенном в 7.2, 7.3.3.3. Общая синхропосылка определяется как объединение `client_write_IV + GenericAEADCipher.nonce_explicit` (для данных, отправляемых клиентом) или `server_write_IV + GenericAEADCipher.nonce_explicit` (для данных, отправляемых сервером).

B.2.2 Алгоритм генерации псевдослучайных чисел

Для генерации псевдослучайных чисел ДОЛЖЕН использоваться алгоритм, определенный в 6.2.3. При построении алгоритма генерации псевдослучайных чисел ДОЛЖЕН использоваться алгоритм хэширования, определенный в СТБ 34.101.31 (пункт 6.9). Названия криптонаборов снабжены суффиксом _HBELT, который указывает на выбор данного алгоритма хэширования.

B.2.3 Алгоритмы формирования общего ключа

Для формирования общего ключа используются алгоритмы, перечисленные в таблице В.2.

В алгоритмах DHT_BIGN, DHE_BIGN, DHT_PSK_BIGN сервер пересылает клиенту свой сертификат. Формат сертификата ДОЛЖЕН соответствовать СТБ 34.101.19 с уточнениями, заданными в СТБ 34.101.45 (приложение Д). Сертификат ДОЛЖЕН иметь расширение KeyUsage.

Таблица В.2 — Алгоритмы формирования общего ключа криптонаборов BIGN_WITH_BELT

Алгоритм формирования общего ключа	Описание
DHE_BIGN	Протокол Диффи – Хеллмана с эфемерными ключами и базовыми операциями СТБ 34.101.45
DHT_BIGN	Транспорт ключа согласно СТБ 34.101.45
DHE_PSK_BIGN	На основе предварительно распределенных общих секретов и протокола Диффи – Хеллмана с эфемерными ключами
DHT_PSK_BIGN	На основе предварительно распределенных общих секретов ключей и транспорта ключа согласно СТБ 34.101.45

В.2.3.1 Алгоритм DHE_BIGN

Алгоритм DHE_BIGN состоит в выполнении протокола Диффи – Хеллмана с эфемерными ключами. В протоколе используются эллиптические кривые, определенные в СТБ 34.101.45 (пункт 5.3). Для генерации пар эфемерных ключей используются алгоритмы, определенные в СТБ 34.101.45 (пункт 6.2). При выполнении DHE_BIGN сервер подписывает свой эфемерный ключ и дополнительные данные. Для выработки и проверки ЭЦП используются алгоритмы, определенные в СТБ 34.101.45 (пункт 7.1).

Для применения DHE_BIGN сервер ДОЛЖЕН переслать клиенту свой сертификат в сообщении `Certificate`. Сертификат ДОЛЖЕН содержать открытый ключ алгоритмов ЭЦП. В расширении `KeyUsage` сертификата ДОЛЖЕН быть установлен бит `digitalSignature`.

Сервер использует параметры эллиптической кривой из своего сертификата, генерирует эфемерные личный ключ `dS` и открытый ключ `QS` с помощью алгоритма, определенного в СТБ 34.101.45 (пункт 6.2.2). Затем сервер подписывает структуру, составленную из `QS` и случайных данных `client_random`, `server_random`. При выработке ЭЦП используется личный ключ, которому соответствует открытый ключ сертификата сервера. Сервер передает `QS` и выработанную подпись клиенту в сообщении `ServerKeyExchange`.

Клиент проверяет полученные сертификат и ЭЦП. Если при проверке произошла ошибка, то клиент ДОЛЖЕН прекратить протокол Handshake. В противном случае клиент использует параметры эллиптической кривой из сертификата сервера, генерирует эфемерные личный ключ `dC` и открытый ключ `QC` с помощью алгоритма, определенного в СТБ 34.101.45 (пункт 6.2.2). Клиент передает открытый ключ серверу в сообщении `ClientKeyExchange`.

Сервер проверяет `QC` с помощью алгоритма, определенного в СТБ 34.101.45 (пункт 6.2.3). Если при проверке произошла ошибка, то сервер ДОЛЖЕН прекратить протокол Handshake.

Открытые ключи `QS`, `QC` являются точками эллиптической кривой, заданной в сертификате сервера. Сервер находит `dS`-кратное точки `QC`, а клиент находит `dC`-кратное точки `QS`. Вычисление кратной точки описано в СТБ 34.101.45 (пункт 4.2.4). Если все шаги протокола выполнены корректно и злоумышленник не вмешивался в обмен сообщениями, то в результате вычислений стороны получат одну и ту же секретную точку `Q`.

Точка `Q` представляется строкой байтов (октетов) по правилам, описанным в СТБ 34.101.45 (пункт 5.4). Полученная строка выступает в роли `pre_master_secret`. По `pre_master_secret` строится `master_secret` (см. 8.16).

B.2.3.2 Алгоритм DHT_BIGN

Алгоритм DHT_BIGN состоит в выполнении сторонами алгоритмов транспорта ключа, определенных в СТБ 34.101.45 (пункт 7.2). К алгоритмам транспорта относится алгоритм создания токена транспортируемого ключа и алгоритм разбора токена. При создании и разборе токена ДОЛЖЕН использоваться нулевой заголовок ключа [см. СТБ 34.101.45 (пункт 5.6)].

Для применения DHT_BIGN сервер ДОЛЖЕН переслать клиенту свой сертификат в сообщении `Certificate`. Сертификат ДОЛЖЕН содержать открытый ключ алгоритмов транспорта ключа. В расширении KeyUsage сертификата ДОЛЖЕН быть установлен бит `keyEncipherment`.

Клиент проверяет полученный сертификат и определяет размещенные в сертификате параметры эллиптической кривой и открытый ключ. Клиент генерирует случайный ключ `pre_master_secret` из 48 байтов, создает токен этого ключа, используя открытый ключ сертификата, и пересылает токен в сообщении `ClientKeyExchange`.

Сервер разбирает токен, используя свой личный ключ. Если при разборе произошла ошибка, то сервер ДОЛЖЕН прекратить протокол Handshake. В противном случае сервер использует полученный по токену `pre_master_secret` для построения `master_secret` (см. 8.16).

B.2.3.3 Алгоритм DHE_PSK_BIGN

Алгоритм DHE_PSK_BIGN состоит в построении `pre_master_secret` по предварительно распределенному между клиентом и сервером секрету. Для того, чтобы `pre_master_secret` не повторялся и чтобы у злоумышленника не было возможности определить предварительно распределенный секрет по сообщениям протокола Handshake, при формировании `pre_master_secret` используется дополнительный ключ, который формируется по протоколу Диффи – Хеллмана.

Для применения DHE_PSK_BIGN сервер выбирает параметры эллиптической кривой, группа точек которой будет использоваться в протоколе Диффи – Хеллмана. Сервер ДОЛЖЕН использовать стандартные параметры, заданные в СТБ 34.101.45 (приложение Б). Для ссылки на параметры должны использоваться идентификаторы, определенные в СТБ 34.101.45 (приложение Д). Идентифакторы должны кодироваться по правилам, заданным в ГОСТ 34.973 и кратко описанным в СТБ 34.101.45 (приложение А).

После выбора параметров сервер генерирует эфемерные личный ключ `dS` и открытый ключ `QS` с помощью алгоритма, определенного в СТБ 34.101.45 (пункт 6.2.2). Сервер передает описание параметров и открытый ключ `QS` клиенту в сообщении `ServerKeyExchange`. Дополнительно в `ServerKeyExchange` сервер МОЖЕТ переслать клиенту подсказку по выбору общего секрета (см. 6.2.4).

Клиент определяет параметры эллиптической кривой по полученному идентификатору и проверяет открытый ключ `QS` с помощью алгоритма, определенного в СТБ 34.101.45 (пункт 6.2.3). Если при определении параметров или при проверке открытого ключа произошла ошибка, то клиент ДОЛЖЕН прекратить Handshake. В противном случае клиент выбирает общий секрет, используя подсказку сервера. Если подсказки нет, то клиент выбирает общий секрет самостоятельно. Клиент высылает серверу идентификатор выбранного секрета в `ClientKeyExchange`. Кроме этого, клиент генерирует эфемерные личный ключ `dC` и открытый ключ `QC` с помощью алгоритма, определенного в СТБ 34.101.45 (пункт 6.2.2). Открытый ключ `QC` также пересылается в `ClientKeyExchange`.

По идентификатору секрета сервер определяет сам секрет, который обозначается `psk` и представляет собой строку байтов. Если сервер не может определить сек-

рет по полученному идентификатору, то он МОЖЕТ ответить сообщением об ошибке `unknown_psk_identity` (см. 10.3). В качестве альтернативы, если сервер хочет скрыть факт незнания секрета, то он МОЖЕТ продолжить протокол, как если бы он знал секрет, но секрет оказался неверным. При этом сервер в конце концов отвечает сообщением об ошибке `decrypt_error`.

Сервер проверяет открытый ключ `QC` с помощью алгоритма, определенного в СТБ 34.101.45 (пункт 6.2.3). Если при проверке открытого ключа произошла ошибка, то сервер ДОЛЖЕН прекратить Handshake.

Сервер находит `dS`-кратное точки `QC`, а клиент находит `dC`-кратное точки `QS`. Вычисление кратной точки описано в СТБ 34.101.45 (пункт 4.2.4). Если все шаги протокола выполнены корректно и злоумышленник не вмешивался в обмен сообщениями, то в результате вычислений стороны получат одну и ту же секретную точку `Q`.

Точка `Q` представляется строкой байтов (октетов) по правилам, описанным в СТБ 34.101.45 (пункт 5.4). Полученная строка состоит из 64, 96 или 128 байтов и обозначается `other_secret`.

Ключ `pre_master_secret` является результатом конкатенации:

```
len(other_secret) + other_secret + len(psk) + psk,
```

где `len(other_secret)`, `len(psk)` — длины `other_secret` и `psk`, представленные двумя байтами.

По `pre_master_secret` строится `master_secret` (см. 8.16).

Реализации ДОЛЖНЫ поддерживать идентификаторы секретов, состоящие из 128 байтов, и секреты, состоящие из 64 байтов. Поддержка идентификаторов и секретов больших длин НЕ РЕКОМЕНДУЕТСЯ.

B.2.3.4 Алгоритм DHT_PSK_BIGN

Алгоритм DHT_PSK_BIGN состоит в построении `pre_master_secret` по предварительно распределенному между клиентом и сервером секрету. Для того, чтобы `pre_master_secret` не повторялся и чтобы у злоумышленника не было возможности определить предварительно распределенный секрет по сообщениям протокола Handshake, при формировании `pre_master_secret` используется дополнительный ключ. Ключ передается серверу с помощью алгоритмов транспорта, определенных в СТБ 34.101.45 (пункт 7.2). К алгоритмам транспорта относится алгоритм создания токена транспортируемого ключа и алгоритм разбора токена. При создании и разборе токена ДОЛЖЕН использоваться нулевой заголовок ключа (см. СТБ 34.101.45 (пункт 5.6)).

Для применения DHT_PSK_BIGN сервер ДОЛЖЕН переслать клиенту свой сертификат в сообщении `Certificate`. Сертификат ДОЛЖЕН содержать открытый ключ алгоритмов транспорта ключа. В расширении `KeyUsage` сертификата ДОЛЖЕН быть установлен бит `keyEnchipherment`.

Дополнительно в `ServerKeyExchange` сервер МОЖЕТ переслать клиенту подсказку по выбору общего секрета (см. 6.2.4). Если подсказка не используется, то `ServerKeyExchange` может не передаваться. Используя подсказку, клиент выбирает общий секрет. Если подсказки нет, то клиент выбирает общий секрет самостоятельно. Клиент высылает серверу идентификатор выбранного секрета в `ClientKeyExchange`.

Кроме этого, клиент проверяет сертификат сервера и определяет размещенные в сертификате параметры эллиптической кривой и открытый ключ. Клиент генерирует случайный ключ `other_secret` из 48 байтов. Клиент создает токен этого ключа, используя открытый ключ сертификата, и пересылает токен в сообщении `ClientKeyExchange`.

По идентификатору секрета стороны определяют секрет psk. Если сервер не может определить секрет по полученному идентификатору, то он МОЖЕТ ответить сообщением об ошибке unknown_psk_identity (см. 10.3). В качестве альтернативы, если сервер хочет скрыть факт незнания секрета, то он МОЖЕТ продолжить протокол, как если бы он знал секрет, но секрет оказался неверным. При этом сервер в конце концов отвечает сообщением об ошибке decrypt_error.

Сервер разбирает токен клиента, используя свой личный ключ. Если при разборе произошла ошибка, то сервер ДОЛЖЕН прекратить протокол Handshake. В противном случае сервер определяет переданный клиентом ключ other_secret.

Ключ pre_master_secret является результатом конкатенации:

```
len(other_secret) + other_secret + len(psk) + psk,
```

где len(other_secret), len(psk) — длины other_secret и psk, представленные двумя байтами.

По pre_master_secret строится master_secret (см. 8.16).

Реализации ДОЛЖНЫ поддерживать идентификаторы секретов, состоящие из 128 байтов, и секреты, состоящие из 64 байтов. Поддержка идентификаторов и секретов больших длин НЕ РЕКОМЕНДУЕТСЯ.

B.3 Методы аутентификации

B.3.1 Аутентификация сервера

При использовании алгоритмов DHE_BIGN, DHT_BIGN, DHT_PSK_BIGN сервер пересылает клиенту свой сертификат. Последующее успешное завершение протокола Handshake означает, что проведена аутентификация сервера: сертификат сервера действителен и сервер знает личный ключ, соответствующий открытому ключу сертификата.

При использовании алгоритмов DHE_PSK_BIGN и DHT_PSK_BIGN клиент и сервер договариваются об использовании предварительного распределенного секрета psk. Если psk был распределен по защищенным каналам и был распределен только этим сторонам, то успешное завершение Handshake означает, что сервер знает psk, т. е. является подлинным.

B.3.2 Аутентификация клиента

Для аутентификации клиента могут использоваться методы bign128_auth, bign192_auth и bign256_auth, которые включаются в перечисление ClientCertificateType:

```
ClientCertificateType += {bign128_auth(121),
    bign192_auth(122), bign256_auth(123)};
```

Для применения данных методов сервер в сообщении CertificateRequest ДОЛЖЕН запросить у клиента его сертификат. После получения сообщения CertificateRequest клиент ДОЛЖЕН отправить запрашиваемый сертификат в сообщении Certificate.

В СТБ 34.101.45 (пункт 5.2) для алгоритмов ЭЦП определены три уровня стойкости: $l = 128$, $l = 192$ и $l = 256$. При использовании методов bign128_auth, bign192_auth и bign256_auth сертификат ДОЛЖЕН содержать открытый ключ, который можно использовать на уровнях стойкости $l \geqslant 128$, $l \geqslant 192$ и $l = 256$ соответственно.

Формат сертификата ДОЛЖЕН соответствовать СТБ 34.101.19 с уточнениями, заданными в СТБ 34.101.45 (приложение Д). Сертификат ДОЛЖЕН иметь расширение KeyUsage и в этом расширении ДОЛЖЕН быть установлен бит digitalSignature.

Успешное завершение протокола Handshake означает, что проведена аутентификация клиента: сертификат клиента действителен и клиент знает личный ключ, соответствующий открытому ключу сертификата.

Методы аутентификации не применяются и предыдущие ограничения не действуют, если клиент передает в сообщении `Certificate` пустую цепочку сертификатов. В этом случае сервер может либо прервать Handshake, либо продолжить его, но уже без аутентификации клиента (см. 8.12).

При использовании алгоритмов DHE_PSK_BIGN, DHT_PSK_BIGN проводится неявная аутентификация клиента. При соблюдении мер защиты общего секрета `psk` успешное завершение Handshake означает, что клиент знает `psk`, т. е. является подлинным.

B.3.3 Алгоритмы хэширования и ЭЦП

В методах аутентификации используется алгоритм хэширования, определенный в СТБ 34.101.31 (пункт 6.9). Этот алгоритм добавляется в перечисление `HashAlgorithm` под именем `belt_hash`:

```
HashAlgorithm += {belt_hash(121)};
```

В методах аутентификации используются также алгоритмы ЭЦП, определенные в СТБ 34.101.45 (пункт 7.1). Эти алгоритмы добавляются в перечисление `SignatureAlgorithm` под именем `bign_sign`:

```
SignatureAlgorithm += {bign_sign(121)};
```

Допустимая пара «алгоритм хэширования, алгоритмы ЭЦП» описывается значением `{belt_hash, bign_sign}` типа `SignatureAndHashAlgorithm`.

В алгоритмах `bign_sign` на уровне стойкости l должна использоваться функция хэширования с $2l$-битовыми значениями. Поскольку длина хэш-значения `belt_hash` составляет 256 битов, алгоритмы `bign_sign` могут выполняться только на уровне $l = 128$. Тем не менее, в будущем список разрешенных функций хэширования может пополняться, и при этом могут быть поддержаны остальные уровни стойкости `bign_sign`. Сказанное означает, что значение `{belt_hash, bign_sign}` соответствует методу аутентификации `bign128_auth`, и не соответствует методам `bign192_auth`, `bign256_auth`.

B.4 Сообщения и шаги

B.4.1 Уточнения

В настоящем подразделе уточняются сообщения и шаги протокола Handshake. Уточнения касаются деталей использования определенных выше криптонаборов и методов аутентификации при выполнении протокола.

Сообщения и шаги протоколов Record, Change Cipher Spec и Alert не уточняются, в этом нет необходимости.

B.4.2 Приветственные сообщения

Для использования криптонаборов настоящего приложения стороны указывают их идентификаторы (см. B.1) в поле `cipher_suites` сообщения `ClientHello` и в поле `cipher_suite` сообщения `ServerHello`.

Для криптонаборов, определенных в настоящем приложении, сообщение `ClientHello` ДОЛЖНО содержать расширение `signature_algorithms` (см. 8.7) за исключением случая, когда клиент поддерживает только одну пару `{belt_hash,bign_sign}`. Клиент МОЖЕТ опустить передачу этой пары в расширении. При получении ClientHello с отсут-

ствующим расширением signature_algorithms сервер ДОЛЖЕН считать, что клиент поддерживает только пару {belt_hash,bign_sign}.

Сообщение HelloRequest не уточняется.

B.4.3 Сообщение сервера Certificate

Сообщение сервера Certificate высылается при использовании любого из алгоритмов DHT_BIGN, DHE_BIGN, DHT_PSK_BSUITES. Содержание сообщения было определено при описании алгоритмов.

B.4.4 Сообщение ServerKeyExchange

Сообщение ServerKeyExchange обязательно высылается при использовании алгоритмов DHE_BIGN, DHE_PSK_BIGN может высылаться при использовании алгоритма DHT_PSK_BIGN и не высылается при использовании алгоритма DHT_BSUITES. Содержание сообщения было определено при описании алгоритмов.

Формат сообщения ServerKeyExchange определяется следующим образом:

```
struct {
  select (KeyExchangeAlgorithm) {
  case dhe_bign:
    opaque public<64..128>;
    digitally-signed struct {
      opaque client_random[32];
      opaque server_random[32];
      opaque public<64..128>;
    } signed_params;
    case dhe_psk_bign:
      opaque psk_identity_hint<0..2^16-1>;
      opaque oid<3..255>;
      opaque public<64..128>;
    case dht_psk_bign:
      opaque psk_identity_hint<0..2^16-1>;
  };
} ServerKeyExchange;
```

Поля структуры ServerKeyExchange имеют следующее значение:

– client_random — случайные данные клиента (см. 7.2);

– server_random — случайные данные сервера (см. 7.2);

– public — эфемерный открытый ключ сервера. Открытый ключ представляется строкой байтов (октетов) по правилам, описанным в СТБ 34.101.45 (пункт 5.4). В зависимости от уровня стойкости строка состоит из 64, 96 или 128 байтов;

– signed_params — ЭЦП полей client_random, server_random, public;

– psk_identity_hint — подсказка по выбору общего предварительно распределенного секрета;

– oid — кодированное представление идентификатора параметров эллиптической кривой.

Используемое в структуре ServerKeyExchange перечисление KeyExchangeAlgorithm определяется следующим образом:

```
enum {dhe_bign, dhe_psk_bign, dht_psk_bign}
   KeyExchangeAlgorithm;
```

Для данного перечисления элементы означают следующее:
– `dhe_bign` — используется алгоритм DHE_BIGN;
– `dhe_psk_bign` — используется алгоритм DHE_PSK_BIGN;
– `dht_psk_bign` — используется алгоритм DHT_PSK_BSUITES.

B.4.5 Сообщение `CertificateRequest`

Сервер может отправить сообщение `CertificateRequest` для аутентификации клиента. В этом случае поля структуры `CertificateRequest` должны быть заданы следующим образом:

– `certificate_types` — идентификаторы методов аутентификации `bign128_auth`, `bign192_auth` или `bign256_auth`;

– `supported_signature_algorithms` — определенные в B.3.3 пары идентификаторов «алгоритм хэширования, алгоритмы ЭЦП».

Поле `certificate_authorities` определяется в соответствии с 8.10.

B.4.6 Сообщение `ServerHelloDone`

Сообщение `ServerHelloDone` не уточняется.

B.4.7 Сообщение клиента `Certificate`

Сообщение посылается клиентом, если сервер прислал запрос `CertificateRequest`. В этом сообщении клиент высылает серверу свой сертификат, соответствующий одному из запрошенных методов аутентификации: `bign128_auth`, `bign192_auth` или `bign256_auth`. Если клиент не имеет подходящего сертификата, то он может послать в сообщении `Certificate` пустую цепочку сертификатов. В этом случае сервер принимает решение о возможности продолжения протокола Handshake без аутентификации клиента.

B.4.8 Сообщение `ClientKeyExchange`

Сообщение `ClientKeyExchange` обязательно высылается при использовании любого из алгоритмов DHE_BIGN, DHT_BIGN, DHE_PSK_BIGN и DHT_PSK_BSUITES. Содержание сообщения было определено при описании алгоритмов.

Формат сообщения `ClientKeyExchange` определяется следующим образом:

```
struct {
  select (KeyExchangeAlgorithm) {
  case dht_bign:
    opaque token<96..160>;
  case dhe_bign:
    opaque public<64..128>;
  case dhe_psk_bign:
    opaque psk_identity<0..2^16-1>;
    opaque public<64..128>;
  case dht_psk_bign:
    opaque psk_identity<0..2^16-1>;
  opaque token<96..160>;
  };
} ClientKeyExchange;
```

Поля структуры `ClientKeyExchange` имеют следующее значение:

– `token` — токен ключа `pre_master_secret`. В зависимости от уровня стойкости токен состоит из 96, 128 или 160 байтов;

– `public` — эфемерный открытый ключ клиента. Открытый ключ представляется строкой октетов по правилам, определенным в СТБ 34.101.45 (пункт 5.4). В зависимости от уровня стойкости строка состоит из 64, 96 или 128 байтов;

– `psk_identity` — идентификатор общего предварительно распределенного секрета.

Используемое в структуре `ClientKeyExchange` перечисление `KeyExchangeAlgorithm` определено в В.4.4 и доопределяется следующим образом:

```
KeyExchangeAlgorithm += {dht_bign};
```

Дополнительный элемент перечисления означает следующее:

– `dht_bign` — используется алгоритм DHT_BSUITES.

В.4.9 Сообщения `CertificateVerify, Finished`

Для выработки ЭЦП при формировании сообщения `CertificateVerify` используются алгоритмы ЭЦП, определенные в В.3.3.

При формировании сообщения `Finished` ДОЛЖЕН использоваться алгоритм генерации псевдослучайных чисел, определенный в В.2.2. Для хэширования сообщений протокола Handshake ДОЛЖЕН использоваться алгоритм, определенный в СТБ 34.101.31 (пункт 6.9). Параметр `verify_data_length` ДОЛЖЕН быть равен 12.

Библиография

[1] ISO/IEC 9899:2011 Information technology — Programming languages + C
Информационные технологии. Языки программирования. Си

[2] RFC 4506:2006 XDR: External Data Representation Standard
(Стандарт представления внешних данных)

[3] RFC 5246:2008 The Transport Layer Security (TLS) Protocol. Version 1.2
(Протокол защиты транспортного уровня. Версия 1.2)

[4] RFC 4279:2005 PreShared Key Ciphersuites for Transport Layer Security (TLS)
(Криптонаборы TLS на основе предварительно распределенного ключа)

[5] RFC 5746:2010 Transport Layer Security (TLS) Renegotiation Indication Extension
(Расширение TLS для индикации переустановки связи)

[6] Bhargavan K., Delignat-Lavaud A., Fournet C., Pironti A., Strub P.-Y. Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS. `https://secure-resumption.com/tlsauth.pdf`, 2014
(Тройное рукопожатие и обрезка куки: взлом и исправление аутентификации на основе TLS)

[7] RFC 3749:2004 Transport Layer Security Protocol Compression Methods
(Методы сжатия протокола защиты транспортного уровня)

[8] Canvel B., Hiltgen A., Vaudenay S., Vuagnoux M. Password Interception in a SSL/TLS Channel. Advances in Cryptology – CRYPTO 2003, LNCS 2729, 2003
(Перехват пароля в канале SSL/TLS)

[9] AlFardan N. J., Paterson K. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. Royal Holloway, University of London. `http://www.isg.rhul.ac.uk/tls/TLStiming.pdf`, 2013
(Лаки Тринадцать: Атаки на протоколы записи TLS и DTLS)

[10] RFC 6066:2011 Transport Layer Security (TLS) Extensions: Extension Definitions
(Расширения TLS: определения расширений)

[11] ITU-T Recommendation X.501: Information Technology — Open Systems Interconnection — The Directory: Models, 1993
(Информационные технологии — Взаимосвязь открытых систем — Директория: Модели)

Поправка к официальной редакции

В каком месте	Напечатано	Должно быть
Приложение В, таблица В.1, вторая колонка	{192, 21} {192, 22} {192, 23} {192, 24} {192, 25} {192, 26} {192, 27} {192, 28}	{255, 21} {255, 22} {255, 23} {255, 24} {255, 25} {255, 26} {255, 27} {255, 28}
Приложение В, пункт В.1, последний абзац	Во всех криптонаборах используется один и тот же алгоритм генерации псевдослучайных чисел (см. В.2.4). Отличаются алгоритмы шифрования (см. В.2.1), алгоритмы имитозащиты (см. В.2.2) и алгоритмы формирования общего ключа (см. В.2.5). Алгоритмы шифрования и имитозащиты могут быть совмещены (см. В.2.3).	Во всех криптонаборах используется один и тот же алгоритм генерации псевдослучайных чисел (см. В.2.2). Отличаются алгоритмы шифрования и имитозащиты (см. В.2.1), алгоритмы формирования общего ключа (см. В.2.3).
Приложение В, пункт В.2, заголовки	В.2.3 Одновременное шифрование и имитозащита	В.2.1.3 Одновременное шифрование и имитозащита
	В.2.4 Алгоритм генерации псевдослучайных чисел	В.2.2 Алгоритм генерации псевдослучайных чисел
	В.2.5 Алгоритмы формирования общего ключа	В.2.3 Алгоритмы формирования общего ключа
	В.2.5.1 Алгоритм DHE_BIGN	В.2.3.1 Алгоритм DHE_BIGN
	В.2.5.2 Алгоритм DHT_BIGN	В.2.3.2 Алгоритм DHT_BIGN
	В.2.5.3 Алгоритм DHE_PSK_BIGN	В.2.3.3 Алгоритм DHE_PSK_BIGN
	В.2.5.4 Алгоритм DHT_PSK_BIGN	В.2.3.4 Алгоритм DHT_PSK_BIGN
Приложение В, пункт В.3.2	`ClientCertificateType += {bign128_auth(121), bign192_auth(122), bign256_auth(123)};`	`ClientCertificateType += {bign128_auth(231), bign192_auth(232), bign256_auth(233)};`
Приложение В, пункт В.3.3	`HashAlgorithm += {belt_hash(121)};` `SignatureAlgorithm += {bign_sign(121)};`	`HashAlgorithm += {belt_hash(231)};` `SignatureAlgorithm += {bign_sign(231)};`