Internet Engineering Task Force (IETF) E. Rescorla
Request for Comments: 8446 Mozilla
Obsoletes: 5077, 5246, 6961 August 2018
Updates: 5705, 6066
Category: Standards Track
ISSN: 2070-1721

The Transport Layer Security (TLS) Protocol Version 1.3

Протокол TLS версии 1.3

Аннотация

Этот документ содержит спецификацию протокола TLS[1] версии 1.3, который обеспечивает клиентам и серверам возможность взаимодействия через Internet с предотвращением раскрытия, фальсификации и подмены сообщений.

Этот документ обновляет RFC 5705 и RFC 6066, а также отменяет RFC 5077, RFC 5246 и RFC 6961. Документ также задаёт новые требования для реализаций TLS 1.2.

Статус документа

Документ относится к категории Internet Standards Track.

Документ является результатом работы IETF[2] и представляет согласованный взгляд сообщества IETF. Документ прошёл открытое обсуждение и был одобрен для публикации IESG[3]. Дополнительную информацию о стандартах Internet можно найти в разделе 2 в RFC 7841.

Информацию о текущем статусе документа, ошибках и способах обратной связи можно найти по ссылке https://www.rfc-editor.org/info/rfc8446.

Оглавление

[1]Transport Layer Security - защита на транспортном уровне.
[2]Internet Engineering Task Force - комиссия по решению инженерных задач Internet.
[3]Internet Engineering Steering Group - комиссия по инженерным разработкам Internet.

1. Введение

Основной задачей протокола TLS является поддержка защищённого канала между двумя коммуникационными партнёрами. К базовому транспорту предъявляются лишь требования надёжности и сохранения порядка пакетов. Защищённый канал должен, в частности, иметь перечисленные ниже свойства.

- Проверка подлинности (Authentication). Серверная сторона соединения аутентифицируется всегда, проверка подлинности клиентской стороны не обязательна. Аутентификация может выполняться на основе асимметричной криптографии (например, RSA [RSA], ECDSA[1] [ECDSA], EdDSA[2] [RFC8032]) или симметричного, заранее известного ключа (PSK[3]).

- Конфиденциальность (Confidentiality). Переданные через канал данные доступны лишь конечным точкам соединения. TLS не скрывает размера передаваемых данных, хотя конечные точки могут дополнять записи TLS для сокрытия размера и улучшения защиты от анализа трафика.

- Целостность (Integrity). Переданные через канал данные не могут быть незаметно изменены атакующим.

Эти свойства должны сохраняться даже при полном контроле сети злоумышленником, как описано в [RFC3552]. Дополнительное описание свойств защищенности дано в Приложении E.

TLS состоит из двух основных частей, указанных ниже.

- Протокол согласования (Handshake, раздел 4) проверяет подлинность сторон, согласует криптографические режимы и параметры, организует совместно используемый ключевой материал. Протокол согласования устойчив к попыткам постороннего вмешательства и активный атакующий не сможет вынудить партнёров согласовать параметры, отличающиеся от тех, которые были бы согласованы при отсутствии атаки.

- Протокол записей (раздел 5) использует параметры, заданные протоколом согласования, для защиты трафика между партнёрами. Протокол делит трафик на последовательности записей, каждая из которых защищается независимо с использованием ключей трафика.

TLS не зависит от прикладных протоколов и совершенно прозрачен для протоколов вышележащих уровней. Однако стандарт TLS не задаёт способа защиты протоколов с помощью TLS - инициирование согласования TLS и интерпретация сертификатов подлинности отдаётся на откуп разработчикам протоколов, работающих на основе TLS и использующих такие протоколы приложений.

Этот документ определяет TLS версии 1.3. Хотя TLS 1.3 и не совместим напрямую с прежними версиями, в протокол встроен механизм указания версии, который позволяет клиентам и серверам согласовать номер версии, поддерживаемой обоими.

[1]Elliptic Curve Digital Signature Algorithm - алгоритм цифровой подписи на основе эллиптических кривых.
[2]Edwards-Curve Digital Signature Algorithm - алгоритм цифровой подписи на основе кривой Эдвардса.
[3]Pre-shared key.

Этот документ заменяет собой (и отменяет) прежние версии TLS, включая версию 1.2 [RFC5246]. Он также отменяет механизм квитанций TLS (ticket), заданный в [RFC5077], заменяя его механизмом, определенным в параграфе 2.2. Восстановление и PSK. Поскольку TLS 1.3 меняет способ создания ключей, он обновляет [RFC5705], как описано в параграфе 7.5. Экспортёры. Документ также меняет способ передачи сообщений OCSP[1], в результате чего обновляет [RFC6066] и отменяет [RFC6961], как описано в параграфе 4.4.2.1. Статус OCSP и расширения SCT.

1.1. Соглашения и термины

Ключевые слова **необходимо** (MUST), **недопустимо** (MUST NOT), **требуется** (REQUIRED), **нужно** (SHALL), **не следует** (SHALL NOT), **следует** (SHOULD), **не нужно** (SHOULD NOT), **рекомендуется** (RECOMMENDED), **не рекомендуется** (NOT RECOMMENDED), **возможно** (MAY), **необязательно** (OPTIONAL) в данном документе интерпретируются в соответствии с BCP 14 [RFC2119] [RFC8174] тогда и только тогда, когда они выделены шрифтом, как показано здесь.

Ниже приведены определения используемых в документе терминов.

client - клиент
Конечная точка, инициирующая соединение TLS.

connection - соединение
Соединение между парой конечных точек на транспортном уровне.

endpoint - конечная точка
Клиентская или серверная сторона соединения.

handshake - согласование
Начальное согласование между клиентом и сервером, задающее параметры их взаимодействия в рамках TLS.

peer - партнёр
Конечная точка. При рассмотрении конкретной точки термин партнёр относится с точке, не являющейся основной темой обсуждения.

receiver - получатель
Конечная точка, принимающая записи.

sender - отправитель
Конечная точка, передающая записи.

server - сервер
Конечная точка, которая не инициировала соединение TLS.

1.2. Основные отличия от TLS 1.2

Ниже приведён список основных функциональных различий между TLS 1.2 и TLS 1.3. Список не является исчерпывающим, но включает и мелкие различия.

- Из списка поддерживаемых симметричных алгоритмов удалены все алгоритмы, считавшиеся унаследованными. Остались только алгоритмы AEAD[2]. Изменена концепция шифронабора для отделения механизмов аутентификации и обмена ключами от алгоритма защиты записи (включая размер секретного ключа) и хэширования, которое будет применяться для функции создания ключей и согласования MAC[3].
- Добавлен режим 0-RTT[4] для экономии одного периода кругового обхода при организации соединения для некоторых данных приложений за счёт отдельных свойств защиты.
- Удалены шифронаборы Static RSA и Diffie-Hellman, все механизмы обмена на базе открытых ключей сейчас обеспечивают полную защиту.
- Все согласующие сообщения после ServerHello шифруются. Новое сообщение EncryptedExtensions позволяет расширениям, передававшимся ранее в открытом сообщении ServerHello, также пользоваться защитой конфиденциальности.
- Переработаны функции создания ключей. Новые функции позволяют упростить анализ за счёт улучшенных свойств разделения ключей. Используется основанная на HMAC функция HKDF[5] в качестве базового примитива.
- Конечный автомат согласования был реструктурирован для улучшения согласованности с удалением лишних сообщений, таких как ChangeCipherSpec (за исключением случаев их необходимости для промежуточных устройств).
- Алгоритмы на основе эллиптических кривых сейчас входят в базовую спецификацию и включены новые алгоритмы подписи, такие как EdDSA. Из TLS 1.3 удалено согласование формата точек и применяется один формат для каждой кривой.
- Внесён ряд криптографических улучшений, включая замену заполнения RSA на RSASSA-PSS[6], а также удаление компрессии, алгоритма DSA[7] и пользовательских групп DHE[8].
- Механизм согласования версии TLS 1.2 был отменен в пользу указания списка версий в расширении. Это улучшает совместимость с имеющимися серверами, в которых некорректно реализовано согласование версий.
- Восстановление сессии с состоянием сервера и без него, а также основанные на PSK шифры ранних версий TLS заменены одним новым обменом PSK.
- Обновлены ссылки с указанием новых версий RFC, где это нужно (например, RFC 5280 вместо RFC 3280).

[1]Online Certificate Status Protocol - протокол состояния сертификата в сети.
[2]Authenticated Encryption with Associated Data - аутентифицированное шифрование со связанными данными.
[3]Message authentication code - код аутентификации сообщения.
[4]A zero round-trip time - нулевой круговой обход.
[5]Extract-and-Expand Key Derivation Function - функция создания ключей «извлечь и расширить».
[6]RSA Probabilistic Signature Scheme - схема вероятностной подписи RSA.
[7]Digital Signature Algorithm - алгоритм цифровой подписи.
[8]Ephemeral Diffie-Hellman - эфемерная группа Diffie-Hellman.

1.3. Обновления, влияющие на TLS 1.2

Этот документ задаёт некоторые изменения, которые могут влиять на реализации TLS 1.2, включая те, которые не поддерживают TLS 1.3.

- Механизм защиты от снижения версии, описанный в параграфе 4.1.3. Серверное сообщение Hello.
- Схемы подписи RSASSA-PSS, определённые в параграфе 4.2.3. Алгоритмы подписи.
- Расширение supported_versions для ClientHello может применяться при согласовании используемой версии TLS вместо поля legacy_version в ClientHello.
- Расширение signature_algorithms_cert позволяет клиенту указать, какие алгоритмы подписи он может принимать в сертификатах X.509.

В параграфе 9.3. Инварианты протокола разъяснены некоторые требования совместимости для прежних версий.

2. Обзор протокола

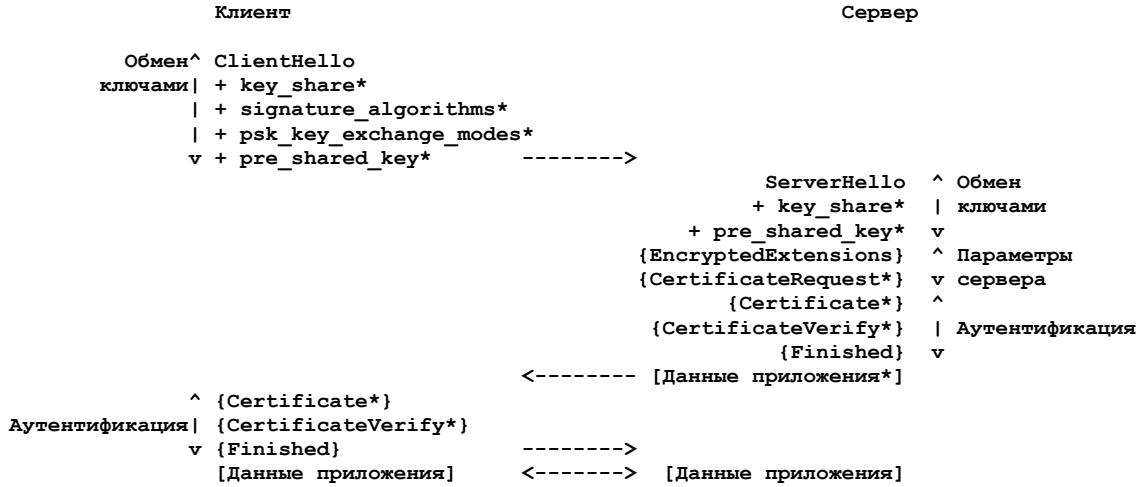
Криптографические параметры, используемые защищённым каналом, создаются протоколом согласования TLS (handshake). Этот субпротокол применяется в TLS клиентом и сервером при первом взаимодействии между ними. Протокол согласования позволяет партнёрам выбрать версию протокола и криптографические алгоритмы, (необязательно) проверить подлинность друг друга, а также создать секретный ключевой материал для общего применения. По завершении согласования стороны используют созданные ключи для защиты трафика приложений.

Отказ при согласовании или иная ошибка протокола ведёт к разрыву соединения, которому может предшествовать сигнальное сообщение (6. Протокол Alert).

TLS поддерживает три базовых режима обмена ключами:

- (EC)DHE (Diffie-Hellman на основе конечных полей или эллиптических кривых);
- PSK;
- PSK с (EC)DHE.

На рисунке 1 показан базовый вариант полного согласования TLS.



\+ заслуживающие внимания расширения из ранее отмеченного сообщения;
* необязательные или зависящие от ситуации сообщения или расширения;
{} сообщения, защищённые ключом на основе [sender]_handshake_traffic_secret;
[] сообщения, защищённые ключом на основе [sender]_application_traffic_secret_N.

Рисунок 1. Поток сообщений при полном TLS Handshake.

Согласование можно рассматривать как 3 этапа, показанных на рисунке 1.

- Обмен ключами создаёт общий ключевой материал и задаёт криптографические параметры. После этой фазы все данные шифруются.
- Параметры сервера - поддержка прикладных протоколов, проверка подлинности клиента и пр.
- Аутентификация сервера (возможно и клиента), а также подтверждение ключа и целостности согласования.

В фазе обмена ключами (Key Exchange) клиент передаёт сообщение ClientHello (параграф 4.1.2) со случайным одноразовым (nonce) ClientHello.random. Это сообщение предлагает версии протокола, список хэш-пар, шифр-HKDF, набор общих значений Diffie-Hellman (в расширении key_share, параграф 4.2.8), набор меток заранее известных ключей (в расширении pre_shared_key, параграф 4.2.11) или оба, а также может включать дополнительные расширения. Для совместимости с промежуточными устройствами могут добавляться другие поля и сообщения.

Сервер обрабатывает ClientHello и определяет подходящие криптографические параметры для соединения. Затем он отвечает сообщением ServerHello (параграф 4.1.3), которое указывает согласованные параметры соединения. Комбинация ClientHello и ServerHello определяет совместно используемые ключи. Если применяется организация ключей (EC)DHE, сообщение ServerHello включает расширение key_share со своим эфемерным значением Diffie-Hellman, которое **должно** относится к одной группе с одним из общих с клиентом значений. При использовании PSK сообщение ServerHello включает расширение pre_shared_key, указывающие какой из предложенных клиентом PSK был выбран. Реализации могут применять (EC)DHE и PSK вместе и в этом случае сообщение включает оба расширения.

Затем сервер передаёт два сообщения для организации параметров (Server Parameters).

EncryptedExtensions

Отклик на расширения ClientHello, которые не требуют определения криптографических параметров в дополнение к параметрам отдельных сертификатов. [4.3.1. Шифрованные расширения]

CertificateRequest

Если желательна аутентификация клиента по сертификату, сообщение содержит ожидаемые параметры сертификата. Сообщение не применяется, если подлинность клиента не нужно проверять. [4.3.2. Запрос сертификата]

В заключение клиент и сервер обмениваются сообщениями Authentication. TLS использует один и тот же набор сообщений при каждом случае аутентификации по сертификатам (аутентификация PSK является побочным эффектом обмена ключами).

Certificate

Сертификат конечной точки и связанные с ним расширения. Сервер пропускает сообщение, если его подлинность не проверяется по сертификату, а клиент пропускает его, если сервер не передал CertificateRequest (т. е. клиента не нужно аутентифицировать по сертификату). Отметим, что при использовании необработанных (raw) открытых ключей [RFC7250] или расширения для кэширования информации [RFC7924], это сообщение не будет включать сертификат, а вместо него будет указано некое значение, соответствующее долгосрочному ключу сервера. [4.4.2. Сообщение Certificate]

CertificateVerify

Подпись всего согласования с применением секретного ключа, соответствующего открытому ключу в сообщении Certificate. Сообщение пропускается, если конечная точка не аутентифицируется по сертификату [4.4.3. Сообщение CertificateVerify]

Finished

Код MAC для всего согласования. Это сообщение содержит подтверждение ключа, привязку конечных точек к переданным ключам, а в режиме PSK также подтверждает подлинность согласования [4.4.4. Сообщение Finished]

При получении сообщений от сервера клиент отвечает своими сообщениями Authentication - Certificate, CertificateVerify (при запросе) и Finished.

На этом согласование завершается, а клиент и сервер создают ключевой материал, нужный уровню записи для обмена данными приложений, защищённого аутентифицированным шифрованием. Данные приложений **недопустимо** передавать до отправки сообщения Finished, за исключением случая, описанного в параграфе 2.3. Данные 0-RTT. Отметим, что хотя сервер может передавать данные приложений до получения Authentication от клиента, эти данные будут передаваться неаутентифицированному партнёру.

2.1. Некорректные параметры DHE

Если клиент не представил достаточного расширения key_share (например, включил лишь непригодные или не поддерживаемые сервером группы DHE или ECDHE), сервер исправляет несоответствие путём передачи сообщения HelloRetryRequest и клиенту нужно возобновить согласование с подходящим расширением key_share, как показано на рисунке 2. Если общие криптографические параметры не согласованы, сервер **должен** прервать согласование с подходящим сигналом.

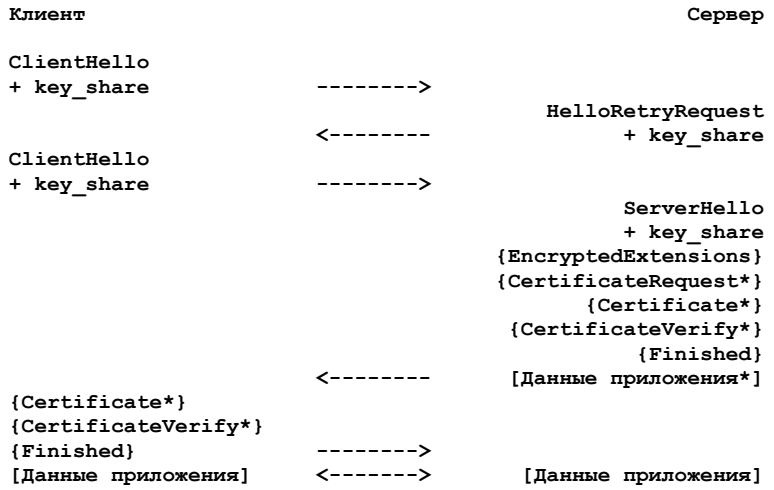


Рисунок 2. Поток сообщений полного TLS Handshake с несоответствием параметров.

Примечание. Показанное на рисунке согласование включает начальный обмен ClientHello/HelloRetryRequest и не сбрасывается новым ClientHello.

TLS поддерживает также несколько оптимизированных вариантов базового согласования, описанных ниже.

2.2. Восстановление и PSK

Хотя TLS PSK можно организовать отдельно (out of band), возможно создание ключей PSK в предшествующем соединении для использования в будущем (восстановление сессии или восстановление с PSK). По завершении согласования сервер может передать клиенту отождествление PSK, соответствующее уникальному ключу, выведенному из начального согласования (4.6.1. Сообщение NewSessionTicket). Клиент может применять это отождествление PSK в последующем согласовании для использования соответствующего PSK. Если сервер воспринимает PSK, контекст защиты нового соединения криптографически привязывается к исходному (прежнему) соединению и выведенный из начального согласования ключ применяется для загрузки криптографического состояния взамен полного согласования. В TLS 1.2 и более ранних версиях эта функциональность обеспечивалась идентификаторами сессий и квитанциями сессий (session ticket) [RFC5077]. Оба эти механизма отменены в TLS 1.3.

Ключи PSK можно использовать в обмене ключами (EC)DHE для ранней секретности в сочетании с общими ключами или отдельно за счёт потери ранней секретности для данных приложения.

На рисунке 3 показаны два согласования, первое из которых создаёт PSK, а второе использует этот секрет.

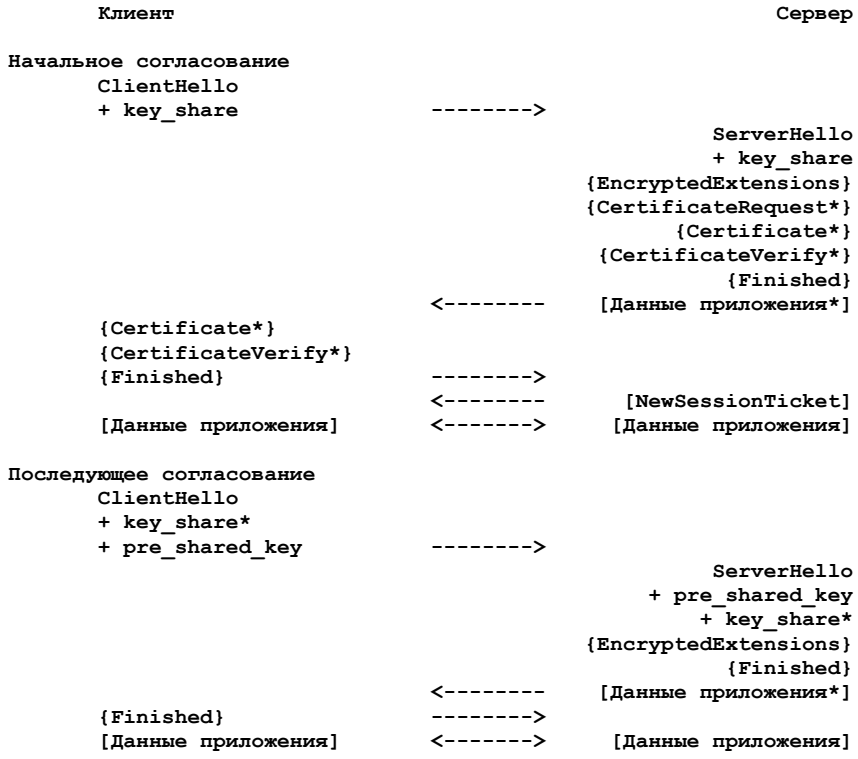


Рисунок 3. Поток сообщений для восстановления и PSK.

Поскольку подлинность сервера проверяется с помощью PSK, он не передаёт сообщений Certificate и CertificateVerify. Когда клиент предлагает восстановление с помощью PSK, ему **следует** также представить серверу расширение key_share, чтобы тот мог при необходимости отвергнуть восстановление и вернуться к полному согласованию. Сервер отвечает расширением pre_shared_key для согласования использования PSK и может (как показано здесь) ответить расширением key_share для организации ключей (EC)DHE, обеспечивающих раннюю секретность.

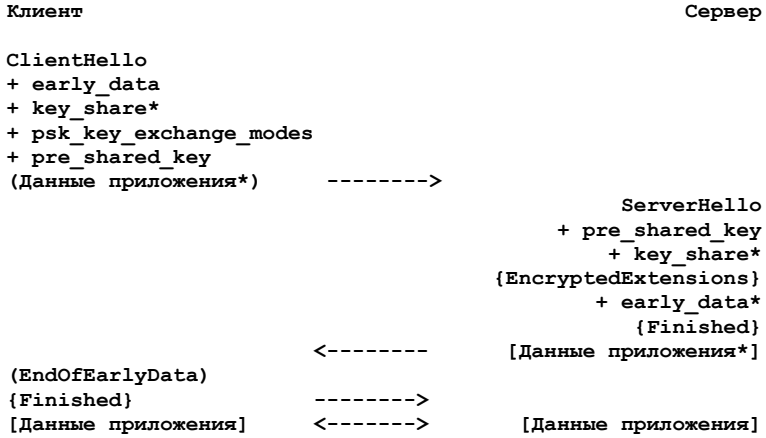
Когда ключ PSK предоставляется по отдельному каналу (out of band), **должны** быть предоставлены также отождествление PSK и алгоритм хэширования KDF, которые будут применяться с PSK.

> Примечание. При использовании переданного заранее по отдельному каналу ключа важен вопрос достаточности энтропии при создании ключа, как описано в [RFC4086]. Создание общего секрета на основе пароля или другого источника с малой энтропией не обеспечивает достаточной защиты. Секреты с малой энтропией или пароли взламываются с помощью атак по словарю, основанных на механизме привязки PSK. Указанная проверка подлинности PSK не является надёжным обменом ключами даже при использовании метода Diffie-Hellman. В частности, это не мешает атакующему, который может наблюдать согласование, провести атаку путём подбора (brute-force) пароля или заранее известного ключа.

2.3. Данные 0-RTT

Когда клиент и сервер применяют PSK (полученный извне или при предшествующем согласовании), TLS 1.3 позволяет клиентам сразу передавать данные (early data). Клиент применяет PSK для аутентификации сервера и шифрования.

Как показано на рисунке 4, данные 0-RTT просто добавляются к согласованию 1-RTT при первой передаче (first flight). Остальная часть согласования использует те же сообщения, что и согласование 1-RTT с восстановлением PSK.



\+ заслуживающие внимания расширения из ранее отмеченного сообщения;
* необязательные или зависящие от ситуации сообщения или расширения;
() сообщения, защищённые с помощью ключа, выведенного из client_early_traffic_secret;
{} сообщения, защищённые ключом на основе [sender]_handshake_traffic_secret;
[] сообщения, защищённые ключом на основе [sender]_application_traffic_secret_N.

Рисунок 4. Поток сообщений для 0-RTT Handshake.

Примечание для разработчиков. Защитные свойства для данных 0-RTT слабее, чем для остальных данных TLS.

1. Эти данные не защищены заранее, поскольку шифруются лишь ключом, выведенным из предложенного PSK.

2. Нет гарантии защиты от повторного использования (replay) между соединениями. Защита от повторного использования для обычных данных TLS 1.3 1-RTT обеспечивается с помощью серверного значения Random, но данные 0-RTT не зависят от ServerHello и поэтому имеют более слабые гарантии. Это особенно актуально при аутентификации данных с помощью проверки подлинности клиента TLS или в рамках прикладного протокола. Эти предупреждения относятся и к любому использованию early_exporter_master_secret.

Данные 0-RTT не могут дублироваться в соединении (т. е. сервер не будет дважды обрабатывать одни данные в том же соединении) и атакующий не сможет представить данные 0-RTT как данные 1-RTT (поскольку они защищены разными ключами). В Приложении E.5 приведено описание возможных атак, а в разделе 8 описаны механизмы, которые сервер может применить для минимизации эффекта повторного использования пакетов.

3. Язык представления

Этот документ имеет дело с форматированием данных для внешнего представления. Используемый синтаксис представления кратко описан ниже.

3.1. Размер базового блока

Представление всех элементов данных описано в явной форме. Базовый блок данных имеет размер 1 байт (8 битов). Многобайтовые элементы данных объединяются (конкатенация) слева направо и сверху вниз. Из байтового потока многобайтовый элемент (например, число) формируется следующим образом (используется нотация языка C)

```
значение = (байт[0] << 8*(n-1)) | (байт[1] << 8*(n-2)) | ... | байт[n-1];
```

Для многобайтовых значений используется сетевой порядок следования байтов[1].

3.2. Различные элементы

Текст комментария начинается с символов /* и заканчивается символами */.

Необязательные компоненты заключены в двойные квадратные скобки [[ ]].

Однобайтовые элементы, содержащие неинтерпретируемые данные, имеют тип opaque[2].

Псевдоним T' для имеющегося типа T определяется в виде

```
T T';
```

3.3. Числа

Базовым числовым элементом является беззнаковый байт uint8. Все остальные типы чисел формируются из базового типа путём описанной в параграфе 3.1 конкатенации фиксированного числа байтов. Ниже перечислены предопределённые типы чисел.

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

Все числовые значения, используемые в данной спецификации, сохраняются в сетевом порядке байтов. Число uint32, представленное шестнадцатеричными байтами 01 02 03 04, эквивалентно десятичному значению 16909060.

3.4. Векторы

Вектор (одномерный массив) представляет собой поток однородных элементов данных. Размер вектора может быть указан в документации или согласован во время работы. В любом случае размер задаётся в байтах, а не числом элементов вектора. Синтаксис задания нового типа T', который относится к векторам фиксированного размера типа T, имеет вид

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

Вектор T' в потоке данных занимает n байтов T. Размер вектора не включается в кодированный поток данных.

В приведённом ниже примере Datum определяется как три последовательных байта, которые протокол не интерпретирует, а Data - как три последовательных элемента Datum, занимающих в общей сложности 9 байтов.

```
opaque Datum[3];      /* три неинтерпретируемых байта */
Datum Data[9];        /* 3 последовательных 3-байтовых вектора */
```

Векторы переменной длины определяются с указанием допустимого диапазона размеров (включая крайние значения) в форме <floor..ceiling>. При кодировании в поток данных перед самим вектором помещается его реальный размер. Размер задаётся в форме числа, занимающего столько байтов, сколько требуется для хранения максимального (ceiling) размера вектора. Вектор переменной длины, имеющий нулевой размер, указывается как пустой вектор.

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

В приведённом ниже примере mandatory представляет собой вектор типа opaque размером от 300 до 400 байтов. Такой вектор никогда не может быть пустым. Поле размера занимает два байта (uint16), что достаточно для записи максимальной длины вектора 400 (3.3. Числа). Вектор longer может представлять до 800 байтов данных или до 400 элементов uint16 и может быть пустым. Кодирование вектора включает двухбайтовое поле размера, предшествующее вектору. Размер кодированного вектора должен быть кратным размеру одного элемента (например, значение 17 для вектора uint16 будет некорректным).

```
opaque mandatory<300..400>;
      /* поле размера занимает 2 байта, вектор не может быть пустым */
uint16 longer<0..800>;
      /* от 0 до 400 16-битовых целых чисел без знака */
```

[1]Network или big endian, когда первым указывается старший байт.
[2]Неинтерпретируемые данные – *Прим. перев.*

3.5. Перечисляемые значения

Используется также дополнительный тип данных - перечисляемые значения enum. Каждое определение задаёт новый тип. Сравниваться и присваиваться могут только перечисляемые значения одного типа. Каждому элементу перечисляемого типа должно быть присвоено значение, как показано в приведённом ниже примере. Поскольку элементы перечисляемого типа не упорядочены, каждый элемент должен иметь уникальное значение.

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

Будущие расширения или дополнения протокола могут определять новые значения. Реализации должны быть способны разбирать и игнорировать неизвестные значения, если определение поля не указывает иное.

Перечисляемые значения занимают в потоке байтов столько места, сколько нужно для записи значения самого большого элемента данного перечисляемого типа. Элементы определённого ниже перечисляемого типа Color будут занимать в потоке по 1 байту.

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

Можно задать значение без связанного с ним тега для расширения размера типа без создания ненужных элементов.

В приведённом ниже определении задаётся тип Taste, элементы которого занимают в потоке по 2 байта и могут принимать только значения только 1, 2 или 4 для текущей версии протокола.

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

Имена элементов перечисляемого типа доступны только в контексте данного типа. В первом примере полная ссылка на второй элемент типа Color будет иметь вид Color.blue. Полная форма представления не требуется, если целью присваивания является полностью определённый элемент.

```
Color color = Color.blue;     /* полная форма, корректно */
Color color = blue;           /* корректно, тип задан неявно */
```

Имена перечисляемых элементов не обязаны быть уникальными. Числовое значение может описывать диапазон, к которому применимо одно имя. Значение включает минимальное и максимальное (включительно) значения диапазона, разделённые двумя точками. Это полезно в основном для резервирования областей в пространстве значений.

```
enum { sad(0), meh(1..254), happy(255) } Mood;
```

3.6. Составные типы

Из примитивов могут создаваться составные типы. Каждая спецификация составного типа задаёт новый уникальный тип. Синтаксис похож на синтаксис структур языка C.

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

Разрешены векторные поля фиксированного и переменного размера, использующие стандартный синтаксис векторов. Структуры V1 и V2 в параграфе 3.8. Варианты показывают это.

Поля структуры можно указывать с использованием идентификатора типа, как для перечисляемых значений. Например, T.f2 будет указывать на второе поле определённого выше составного типа.

3.7. Константы

Полям и переменным могут назначаться фиксированные значения с использованием знака равенства =

```
struct {
    T1 f1 = 8;  /* T.f1 всегда должно иметь значение 8 */
    T2 f2;
} T;
```

3.8. Варианты

Определяемая структура может содержать варианты, выбор между которыми основывается на доступной в среде информации. Селектор вариантов должен относиться к перечисляемому типу, включающему возможные варианты, определяемые структурой. Каждый вариант выбора (см. ниже) задаёт тип поля этого варианта и необязательное поле метки. Механизм выбора варианта во время исполнения не задаётся языком представления.

```
struct {
    T1 f1;
    T2 f2;
    ....
    Tn fn;
    select (E) {
        case e1: Te1 [[fe1]];
        case e2: Te2 [[fe2]];
        ....
        case en: Ten [[fen]];
    };
} Tv;
```

Например,

```
enum { apple(0), orange(1) } VariantTag;

struct {
    uint16 number;
    opaque string<0..10>; /* переменный размер */
} V1;
```

```
    struct {
        uint32 number;
        opaque string[10];    /* фиксированный размер */
    } V2;

    struct {
        VariantTag type;
        select (VariantRecord.type) {
            case apple:  V1;
            case orange: V2;
        };
    } VariantRecord;
```

4. Протокол согласования

Протокол Handshake служит для согласования параметров защиты соединения. Сообщения согласования передаются уровню записи TLS, где они инкапсулируются в одну или несколько структур TLSPlaintext или TLSCiphertext, которые обрабатываются и передаются в соответствии с активным в данный момент состоянием.

```
    enum {
        client_hello(1),
        server_hello(2),
        new_session_ticket(4),
        end_of_early_data(5),
        encrypted_extensions(8),
        certificate(11),
        certificate_request(13),
        certificate_verify(15),
        finished(20),
        key_update(24),
        message_hash(254),
        (255)
    } HandshakeType;

    struct {
        HandshakeType msg_type;    /* тип согласования */
        uint24 length;             /* оставшиеся байты сообщения */
        select (Handshake.msg_type) {
            case client_hello:          ClientHello;
            case server_hello:          ServerHello;
            case end_of_early_data:     EndOfEarlyData;
            case encrypted_extensions:  EncryptedExtensions;
            case certificate_request:   CertificateRequest;
            case certificate:           Certificate;
            case certificate_verify:    CertificateVerify;
            case finished:              Finished;
            case new_session_ticket:    NewSessionTicket;
            case key_update:            KeyUpdate;
        };
    } Handshake;
```

Протокольные сообщения должны передаваться в порядке, заданном в параграфе 4.4.1. Transcript-Hash и показанном на рисунках в разделе 2. Обзор протокола. Партнёр, получивший сообщения согласования в неожиданном порядке, должен прервать согласование с сигналом unexpected_message.

Новые сообщения согласования назначаются IANA, как описано в разделе 11. Взаимодействие с IANA.

4.1. Сообщения обмена ключами

Сообщения обмена ключами служат для определения возможностей защиты клиента и сервера, а также организации общих секретов, включая ключи трафика, используемые для защиты остальной части согласования и данных.

4.1.1. Криптографическое согласование

В TLS криптографическое согласование выполняется клиентом и предлагает перечисленные ниже 4 набора опций в сообщении ClientHello.

- Список шифров, указывающий алгоритм AEAD/хэш-пары HKDF, которые поддерживает клиент.
- Расширение supported_groups (4.2.7. Поддерживаемые группы), которое показывает поддерживаемые клиентом группы (EC)DHE и расширение key_share (4.2.8. Совместное использование ключа), содержащее общие значения (EC)DHE для всех или части этих групп.
- Расширение signature_algorithms (4.2.3. Алгоритмы подписи), указывающее алгоритмы подписи, которые клиент может воспринимать. Расширение signature_algorithms_cert (4.2.3. Алгоритмы подписи) может добавляться для указания специфических для сертификата алгоритмов подписи.
- Расширение pre_shared_key (4.2.11. Расширение PSK) со списком отождествлений симметричных ключей, известных клиенту, и расширение psk_key_exchange_modes (4.2.9. Режимы обмена ключами PSK), которое указывает режимы обмена ключами, подходящие для использования с PSK.

Если сервер не выбирал PSK, первые 3 параметра будут ортогональны - сервер независимо выбирает шифр, группу (EC)DHE и ключ, применяемый для создания ключа, а также пару алгоритм-сертификат, используемую для подтверждения своей подлинности клиенту. Если нет пересечений между полученным supported_groups и поддерживаемыми сервером группами, сервер должен прервать согласование с возвратом сигнала handshake_failure или insufficient_security.

Если сервер выбрал PSK, он **должен** также выбрать режим организации ключей из набора, указанного в клиентском расширении psk_key_exchange_modes (только PSK или PSK вместе с (EC)DHE). Отметим, что при использовании PSK без (EC)DHE отсутствие пересечения параметров supported_groups не будет критичным, как при отказе от PSK, описанном выше. Если сервер выбрал группу (EC)DHE, а клиент не предложил совместимого расширения key_share в начальном ClientHello, сервер **должен** ответить сообщением HelloRetryRequest (4.1.4. Запрос повтора Hello).

Если сервер выбрал параметры и не требуется передавать HelloRetryRequest, он указывает выбранные параметры в ServerHello, как показано ниже.

- При использовании PSK сервер передаёт расширение pre_shared_key, указывающее выбранный ключ.
- При использовании (EC)DHE сервер включает расширение key_share. Если PSK не будет применяться, всегда используется (EC)DHE и аутентификация на основе сертификатов.
- При аутентификации по сертификату сервер будет передавать сообщения Certificate (4.4.2. Сообщение Certificate) и CertificateVerify (4.4.3. Сообщение CertificateVerify). В TLS 1.3 в соответствии с данным документом всегда применяется PSK или сертификат, но не могут использоваться сразу оба. Будущие документы могут определить совместное использование.

Если сервер не может согласовать набор параметров (т. е. параметры клиента и сервера не пересекаются), он **должен** прервать согласование с критическим сигналом handshake_failure или insufficient_security (6. Протокол Alert).

4.1.2. Клиентское сообщение Hello

При первом подключении клиента к серверу от него **требуется** передать ClientHello в качестве первого сообщения TLS. Клиент будет также передавать ClientHello, если сервер ответит на его сообщение ClientHello своим HelloRetryRequest. В таких случаях клиент **должен** передать то же самое сообщение ClientHello без его изменения за исключением перечисленных ниже случаев.

- Если в HelloRetryRequest было указано расширение key_share, список секретов заменяется списком с одной записью KeyShareEntry из указанной группы.
- При наличии расширения early_data (4.2.10. Индикация ранних данных) оно удаляется. Ранняя передача данных не разрешена после HelloRetryRequest.
- Включается расширение cookie, если оно присутствует в HelloRetryRequest.
- При наличии расширения pre_shared_key оно обновляется путём повторного расчёта obfuscated_ticket_age и значений привязок, а также (необязательно) удаления PSK, не совместимых с указанным сервером шифром.
- Необязательное добавление, удаление или изменение размера расширения padding [RFC7685].
- В будущем могут быть заданы расширения, наличие которых в HelloRetryRequest разрешит другие изменения.

Поскольку TLS 1.3 запрещает повторное согласование, сервер **должен** прервать соединение с возвратом сигнала unexpected_message, если он согласовал TLS 1.3 и потом получил сообщение ClientHello.

Если сервер организовал соединение TLS с предыдущей версией и получил TLS 1.3 ClientHello для повторного согласования, он **должен** сохранить прежнюю версию протокола. В частности, **недопустимо** согласовывать TLS 1.3.

Структура сообщения приведена ниже.

```
uint16 ProtocolVersion;
opaque Random[32];

uint8 CipherSuite[2];     /* Селектор криптографического набора (шифра) */

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

legacy_version

В прежних версиях TLS это поле служило для согласования версии и указывало старший номер поддерживаемой клиентом версии. Опыт показал, что многие серверы некорректно реализуют согласование версий, что ведёт к «непереносимости», когда сервер отвергает приемлемое по остальным параметрам сообщение ClientHello с номером версии выше поддерживаемого сервером. В TLS 1.3 клиент указывает предпочитаемую версию в расширении supported_versions (4.2.1. Поддерживаемые версии), а в поле legacy_version **должно** быть указано значение 0x0303, соответствующее TLS 1.2. Сообщения TLS 1.3 ClientHello идентифицируются по полю legacy_version = 0x0303 и расширению supported_versions со значением 0x0304 в качестве старшего поддерживаемого номера версии (совместимость с ранними версиями рассмотрена в Приложении D).

random

32 байта, созданные защищённым генератором случайных чисел (Приложение C).

legacy_session_id

Версии TLS до 1.3 поддерживали функцию возобновления сессии (session resumption), которая в данной версии была объединена с PSK (2.2. Восстановление и PSK). Клиенту, который имеет кэшированный идентификатор сессии, установленный сервером более ранней (до TLS 1.3), **следует** поместить в поле значение этого идентификатора. В режиме совместимости (Приложение D.4) это поле **должно** быть непустым, поэтому клиент, не предлагающий сессию до TLS 1.3, **должен** создать новое 32-байтовое значение. Это значение не обязано быть случайным, но ему **следует** быть непредсказуемым для предотвращения реализаций, фиксирующих определённое значение (называется окостенением - ossification). В остальных случаях значением **должен** быть вектор нулевого размера (однобайтовое поле со значением 0).

cipher_suites
Список опций симметричного шифра, поддерживаемых клиентом, в частности алгоритм защиты записей (включая размер секретного ключа) и алгоритм хэширования для использования с HKDF, в порядке убывания предпочтений клиента. Значения определены в Приложении B.4. Если список содержит непонятные, неподдерживаемые или нежелательные для сервера шифры, сервер **должен** игнорировать их и обрабатывать остальные как обычно. Если клиент пытается организовать ключ PSK, ему **следует** анонсировать хотя бы один шифр, указывающий значение Hash, связанное с PSK.

legacy_compression_methods
Версии TLS до 1.3 использовали компрессию и передавали в этом поле список поддерживаемых методов сжатия. Для каждого сообщения TLS 1.3 ClientHello этот вектор **должен** содержать в точности 1 байт со значением 0, который указывал отсутствие (null) сжатия в предыдущих версиях TLS. Если принято сообщение TLS 1.3 ClientHello с другим значением в этом поле, сервер **должен** прервать согласование с сигналом illegal_parameter. Отметим, что серверы TLS 1.3 могут получать сообщения ClientHellos TLS 1.2 и других версий с указанием методов компрессии (если согласована предыдущая версия протокола) и **должны** считать их действительными.

extensions
Клиенты запрашивают у сервера расширение функциональности с помощью поля расширений, формат которого определён в параграфе 4.2. Расширения. В TLS 1.3 использование некоторых расширений обязательно, поскольку функциональность была перенесена в расширения с целью сохранения совместимости сообщений ClientHello с прежними версиями TLS. Серверы **должны** игнорировать непонятные расширения.

Все версии TLS разрешают помещать поле extensions вслед за compression_methods. Сообщения TLS 1.3 ClientHello всегда содержат расширения (как минимум supported_versions, поскольку иначе они будут считаться сообщениями TLS 1.2). Однако серверы TLS 1.3 могут получать сообщения ClientHello без поля extensions от клиентов с прежними версиями TLS. Наличие расширений можно обнаружить по байтам после поля compression_methods в конце ClientHello. Отметим, что этот способ обнаружения необязательных данных отличается от обычного в TLS использования полей переменного размера и выбран для совместимости с определёнными раньше расширениями. Серверы TLS 1.3 должны сразу выполнять такую проверку и попытаться согласовать TLS 1.3 лишь при наличии расширения supported_versions. Если согласована версия TLS до 1.3, сервер **должен** убедиться, что сообщение не содержит данных после поля legacy_compression_methods или содержит там пригодный блок расширений, за которым не следует данных. В противном случае он **должен** прервать согласование с сигналом decode_error.

Если сервер не поддерживает запрошенную в расширении функциональность, клиент **может** прервать согласование.

После отправки сообщения ClientHello клиент ждёт в ответ ServerHello или HelloRetryRequest. Если используется ранняя передача данных, клиент может передать Application Data (2.3. Данные 0-RTT) до следующего согласующего сообщения.

4.1.3. Серверное сообщение Hello

Сервер будет передавать это сообщение в ответ на ClientHello для выполнения процедуры согласования подходящего набора параметров на основе ClientHello.

Структура сообщения приведена ниже.

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

legacy_version
В прежних версиях TLS это поле служило для согласования версий и содержало номер выбранной для соединения версии. Однако некоторые промежуточные устройства сталкивались с отказами при добавлении новой версии. В TLS 1.3 сервер указывает версию в расширении supported_versions (4.2.1. Поддерживаемые версии), а поле legacy_version **должно** иметь значение 0x0303 (TLS 1.2). Вопросы совместимости рассмотрены в Приложении D.

random
32 байта, созданные защищённым генератором случайных чисел (Приложение C). Последние 8 байтов должны переписываться, как указано ниже, если согласована версия TLS 1.2 или TLS 1.1, но остальные биты должны быть случайными. Эта структура генерируется сервером и **должна** создаваться независимо от ClientHello.random.

legacy_session_id_echo
Содержимое клиентского поля legacy_session_id. Отметим, что это поле возвращается даже в том случае, когда клиентское значение соответствует кэшированной версии до TLS 1.3, которую сервер решил не восстанавливать. Клиент, получивший значение legacy_session_id_echo, которое не соответствует переданному им в ClientHello, **должен** прервать согласование с сигналом illegal_parameter.

cipher_suite
Один шифр, выбранный сервером из списка ClientHello.cipher_suites. Клиент, получивший шифр, который он не предлагал, **должен** прервать согласование с сигналом illegal_parameter.

legacy_compression_method
Однобайтовое поле, которое **должно** иметь значение 0.

extensions
Список расширений. Сообщение ServerHello **должно** включать лишь те расширения, которые требуются для организации криптографического контекста и согласования версии протокола. Все сообщения TLS 1.3 ServerHello **должны** включать расширение supported_versions. В настоящее время сообщения ServerHello содержат также расширение pre_shared_key или key_share и могут включать оба (при использовании PSK с организацией ключа (EC)DHE). Остальные расширения (4.2. Расширения) передаются в отдельном сообщении EncryptedExtensions.

Для совместимости с промежуточными устройствами прежних версий (Приложение D.4), сообщение HelloRetryRequest имеет такую же структуру как ServerHello, но в поле Random указывается значение SHA-256 из HelloRetryRequest

```
CF 21 AD 74 E5 9A 61 11 BE 1D 8C 02 1E 65 B8 91
C2 A2 11 16 7A BB 8C 5E 07 9E 09 E2 C8 A8 33 9C
```

При получении сообщения с типом server_hello реализация **должна** сначала проверить значение Random и при совпадении с указанным выше выполнять обработку в соответствии с параграфом 4.1.4. Запрос повтора Hello.

TLS 1.3 имеет механизм защиты от понижения версии, встроенный в случайное значение сервера. Сервер TLS 1.3, который согласовал версию TLS 1.2 или ниже в ответ на ClientHello, **должен** установить специальное значение в последних 8 байтах поля Random в сообщении ServerHello.

Если согласована версия TLS 1.2, сервер TLS 1.3 **должен** установить

```
44 4F 57 4E 47 52 44 01
```

Если согласована версия TLS 1.1 или ниже, сервер TLS 1.3 **должен**, а серверу TLS 1.2 **следует** установить для 8 последних байтов ServerHello.Random значение

```
44 4F 57 4E 47 52 44 00
```

Клиенты TLS 1.3, получившие сообщение ServerHello, указывающее версию TLS 1.2 или ниже, **должны** проверить последние 8 байтов случайного значения на предмет совпадения с любой из указанных выше последовательностей. Клиентам TLS 1.2 **следует** проверить последние 8 на предмет совпадения со вторым значением, если в ServerHello указана версия TLS 1.1 или ниже. При совпадении значений клиент **должен** прервать согласование с сигналом illegal_parameter. Этот механизм обеспечивает ограниченную защиту от атак на понижение версии в дополнение к защите с помощью обмена Finished - поскольку сообщение ServerKeyExchange, присутствующее в TLS 1.2 и ниже, включает подпись учитывающую оба шифрованных значения, активный атакующий не сможет незаметно поменять случайные значения при использовании эфемерных шифров. При использовании статического шифра RSA защиты от понижения версии не обеспечивается.

> Примечание. Это отличается от [RFC5246], поэтому на практике многие клиенты и серверы TLS 1.2 не будут вести себя, как указано выше.

Устаревший клиент TLS, выполняющий повторное согласование с TLS 1.2 или более ранней версией и получивший при этом сообщение TLS 1.3 ServerHello, **должен** прервать согласование с сигналом protocol_version. Отметим, что при выборе версии TLS 1.3 повторное согласование невозможно.

4.1.4. Запрос повтора Hello

Сервер передаёт сообщение HelloRetryRequest в ответ на ClientHello, если он смог найти подходящий набор параметров, но в ClientHello недостаточно информации для обработки согласования. Как указано в параграфе 4.1.3, HelloRetryRequest имеет такой же формат, как сообщение ServerHello, а поля legacy_version, legacy_session_id_echo, cipher_suite и legacy_compression_method имеют такой же смысл. Однако для удобства HelloRetryRequest рассматривается здесь как отдельное сообщение.

Расширения сервера **должны** содержать supported_versions, а также **следует** включать минимальный набор расширений, требуемых клиенту для генерации корректной пары ClientHello. Как и для ServerHello в сообщения HelloRetryRequest **недопустимо** включать какие-либо расширения, которые не были предложены клиентом в сообщении ClientHello, за исключением необязательного расширения cookie (4.2.2. Cookie).

При получении HelloRetryRequest клиент **должен** проверить поля legacy_version, legacy_session_id_echo, cipher_suite и legacy_compression_method, как указано в параграфе 4.1.3, а затем обработать расширения, начав с определения версии из поля supported_versions. Клиент **должен** прервать согласование с сигналом illegal_parameter, если HelloRetryRequest не будет приводить к какому-либо изменению ClientHello. Если клиент получает второе сообщение HelloRetryRequest в том же соединении (т. е. в ответ сообщение ClientHello, которое уже является ответом на HelloRetryRequest), он **должен** прервать согласование с сигналом unexpected_message.

В остальных случаях клиент **должен** обработать все расширения в HelloRetryRequest и передать второе (обновлённое) сообщение ClientHello. Определённые этой спецификацией для HelloRetryRequest расширения включают:

- supported_versions (4.2.1. Поддерживаемые версии);
- cookie (4.2.2. Cookie);
- key_share (4.2.8. Совместное использование ключа).

Клиент, получивший шифр, который он не предлагал, **должен** прервать согласование. Серверы **должны** гарантировать согласование того же шифра при получении соответствующего обновлённого сообщения ClientHello (если сервер выбрал шифр на первом этапе согласования, это произойдёт автоматически). При получении ServerHello клиенты **должны** проверить совпадение шифра в ServerHello с предложенным в HelloRetryRequest и при расхождении прервать согласование с сигналом illegal_parameter.

Кроме того, в обновлённом сообщении ClientHello клиенту **не следует** предлагать какой-либо заранее известный (pre-shared) ключ хэширования, отличающимся от выбранного шифра. Это позволяет клиентам избежать необходимости частичного расчёта для множества хэш-значений во втором сообщении ClientHello.

Значение selected_version из расширения supported_versions в сообщении HelloRetryRequest **должно** сохраняться в ServerHello и клиент **должен** прерывать согласование с сигналом illegal_parameter, если получено другое значение.

4.2. Расширения

Многие сообщения TLS содержат расширения в формате TLV[1].

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

enum {
    server_name(0),                             /* RFC 6066 */
    max_fragment_length(1),                     /* RFC 6066 */
    status_request(5),                          /* RFC 6066 */
```

[1]Tag-length-value - тег, размер, значение.

```
        supported_groups(10),                   /* RFC 8422, 7919 */
        signature_algorithms(13),               /* RFC 8446 */
        use_srtp(14),                           /* RFC 5764 */
        heartbeat(15),                          /* RFC 6520 */
        application_layer_protocol_negotiation(16), /* RFC 7301 */
        signed_certificate_timestamp(18),       /* RFC 6962 */
        client_certificate_type(19),            /* RFC 7250 */
        server_certificate_type(20),            /* RFC 7250 */
        padding(21),                            /* RFC 7685 */
        pre_shared_key(41),                     /* RFC 8446 */
        early_data(42),                         /* RFC 8446 */
        supported_versions(43),                 /* RFC 8446 */
        cookie(44),                             /* RFC 8446 */
        psk_key_exchange_modes(45),             /* RFC 8446 */
        certificate_authorities(47),            /* RFC 8446 */
        oid_filters(48),                        /* RFC 8446 */
        post_handshake_auth(49),                /* RFC 8446 */
        signature_algorithms_cert(50),          /* RFC 8446 */
        key_share(51),                          /* RFC 8446 */
        (65535)
    } ExtensionType;
```

- extension_type указывает конкретный тип расширения.
- extension_data содержит информацию, относящуюся к определённому типу расширения.

Список типов расширений поддерживается IANA, как описано в разделе 11.

Расширения обычно представлены парами запрос-отклик, но некоторые являются просто указаниями без соответствующих откликов. Клиент передаёт свои запросы расширений в сообщении ClientHello, а сервер возвращает отклики в сообщениях ServerHello, EncryptedExtensions, HelloRetryRequest и Certificate. Сервер передаёт запросы расширений в сообщении CertificateRequest, на которое клиент **может** ответить сообщением Certificate. Сервер **может** также передавать незапрошенные расширения в NewSessionTicket, хотя клиент не отвечает на них непосредственно.

Реализациям **недопустимо** передавать отклики на расширения, если удалённая сторона не передавала соответствующего запроса. Единственным исключением является расширение cookie в HelloRetryRequest. При получении незапрошенного отклика на расширение конечная точка **должна** прервать согласование с сигналом unsupported_extension.

Приведённая ниже таблица указывает сообщения, в которых может присутствовать каждое из расширений (CH - ClientHello, SH - ServerHello, EE - EncryptedExtensions, CT - Certificate, CR - CertificateRequest, NST - NewSessionTicket, HRR - HelloRetryRequest). Если реализация получает расширение, которое понятно, но не предназначено для данного сообщения, она **должна** прервать согласование с сигналом illegal_parameter.

Расширение	***TLS 1.3***
server_name [RFC6066]	CH, EE
max_fragment_length [RFC6066]	CH, EE
status_request [RFC6066]	CH, CR, CT
supported_groups [RFC7919]	CH, EE
signature_algorithms (RFC 8446)	CH, CR
use_srtp [RFC5764]	CH, EE
heartbeat [RFC6520]	CH, EE
application_layer_protocol_negotiation [RFC7301]	CH, EE
signed_certificate_timestamp [RFC6962]	CH, CR, CT
client_certificate_type [RFC7250]	CH, EE
server_certificate_type [RFC7250]	CH, EE
padding [RFC7685]	CH
key_share (RFC 8446)	CH, SH, HRR
pre_shared_key (RFC 8446)	CH, SH
psk_key_exchange_modes (RFC 8446)	CH
early_data (RFC 8446)	CH, EE, NST
cookie (RFC 8446)	CH, HRR
supported_versions (RFC 8446)	CH, SH, HRR
certificate_authorities (RFC 8446)	CH, CR
oid_filters (RFC 8446)	CR
post_handshake_auth (RFC 8446)	CH
signature_algorithms_cert (RFC 8446)	CH, CR

При наличии разнотипных расширений они **могут** размещаться в произвольном порядке, за исключением того, что расширение pre_shared_key (4.2.11. Расширение PSK) **должно** быть последним в сообщении ClientHello (но может указываться в любом месте блока расширений ServerHello). В блоке расширений **недопустимо** наличие нескольких однотипных расширений.

В TLS 1.3 (в отличие от TLS 1.2) расширения согласуются для каждого «приветствия» (handshake) даже в режиме resumption-PSK. Однако при расхождении параметров 0-RTT с согласованными в предыдущем приветствии может потребоваться отвергнуть 0-RTT (4.2.10. Индикация ранних данных).

В этом протоколе имеются тонкие (и не очень тонкие) взаимодействия между новыми и имеющимися возможностями, которые могут приводить к существенному снижению общего уровня защиты. Приведённые ниже соображения следует принимать во внимание при разработке новых расширений.

- Некоторые случаи, когда сервер не согласен с предложенными расширениями, связаны с ошибками (например, согласование не может быть продолжено), а в других это может быть просто отказом от поддержки конкретной функции. В общем случае для первого варианта следует применять сигналы об ошибках, а во втором - отклик на расширение от сервера.

- Расширения, насколько это возможно, следует проектировать так, чтобы предотвратить любую атаку, вынуждающую использовать (или не использовать) конкретную функцию путём манипуляций с приветственными сообщениями. Этот принцип следует соблюдать независимо от того, связана ли функция с вопросами безопасности. Зачастую будет достаточно учёта всех полей расширений при хешировании сообщения Finished, но следует соблюдать особую осторожность, когда расширение меняет смысл сообщений, передаваемых на этапе согласования. Разработчикам следует учитывать то, что до аутентификации согласования атакующий может менять сообщения, а также добавлять, изменять и удалять расширения.

4.2.1. Поддерживаемые версии

```
struct {
    select (Handshake.msg_type) {
        case client_hello:
             ProtocolVersion versions<2..254>;

        case server_hello: /* и HelloRetryRequest */
             ProtocolVersion selected_version;
    };
} SupportedVersions;
```

Расширение supported_versions используется клиентами для указания поддерживаемых версий TLS и серверами для указания используемой версии. Расширение содержит список поддерживаемых версий в порядке снижения предпочтений. Реализации данной спецификации **должны** передавать это расширение в ClientHello с указанием всех версий TLS, которые они готовы согласовать (для данной спецификации это означает значение 0x0304, но при поддержке других версий TLS они также **должны** быть указаны). Если этого расширения нет и сервер, соответствующий данной спецификации, поддерживает также TLS 1.2, он **должен** согласовывать TLS 1.2 и более ранние версии в соответствии с [RFC5246], даже если ClientHello.legacy_version содержит 0x0304 или более позднюю версию. Сервер **может** прервать согласование при получении ClientHello с legacy_version = 0x0304 или выше.

Если это расширение включено в ClientHello, серверу **недопустимо** использовать значение ClientHello.legacy_version для согласования версии и он **должен** применять для определения клиентских предпочтений лишь расширение supported_versions. Сервер **должен** выбирать только версию TLS, предложенную в этом расширении, и **должен** игнорировать все неизвестные версии из расширения. Отметим, что этот механизм позволяет согласовать версии до TLS 1.2, если одна сторона поддерживает «разреженный» диапазон. Реализациям TLS 1.3, которые выбрали поддержку предыдущих версий TLS, **следует** поддерживать TLS 1.2. Серверы **должны** быть готовы к приёму сообщений ClientHello, включающих это расширение без значения 0x0304 в числе поддерживаемых версий.

Сервер, который согласует версию TLS до TLS 1.3, **должен** установить ServerHello.version, а передача расширения supported_versions для него **недопустима**. Сервер, согласующий TLS 1.3, должен отвечать отправкой расширения supported_versions, содержащего значение выбранной версии (0x0304). Он **должен** установить в поле ServerHello.legacy_version значение 0x0303 (TLS 1.2). Клиенты **должны** проверять это расширение до обработки остальной части ServerHello (хотя они должны разобрать ServerHello для считывания этого расширения). Если данное расширение присутствует, клиент **должен** игнорировать значение ServerHello.legacy_version и **должен** применять для определения выбранной версии лишь расширение supported_versions. Если расширение supported_versions в ServerHello содержит версию, не предлагавшуюся клиентом, или версию до TLS 1.3, клиент **должен** прервать согласование с сигналом illegal_parameter.

4.2.2. Cookie

```
struct {
    opaque cookie<1..2^16-1>;
} Cookie;
```

Расширение cookie служит двум основным целям.

- Разрешить серверу заставлять клиента демонстрировать доступность по видимому сетевому адресу (некоторая мера защиты от DoS-атак). Это полезно в первую очередь для транспорта без организации соединений (пример представлен в [RFC6347]).
- Разрешить серверу выгрузку состояния клиенту, позволяющую передавать HelloRetryRequest без сохранения состояния. Сервер может делать это, включая хэш ClientHello в поле cookie сообщения HelloRetryRequest (с подходящей защитой целостности).

При отправке HelloRetryRequest сервер **может** представить клиенту расширение cookie (исключение из правила указания лишь расширений, присутствующих в ClientHello). При отправке нового ClientHello клиент **должен** скопировать содержимое из расширения в HelloRetryRequest в своё расширение cookie нового сообщения ClientHello. Клиентам **недопустимо** использовать cookie в начальных ClientHello при последующих подключениях.

Когда сервер не хранит состояния, он может получить незащищённую запись типа change_cipher_spec между первым и вторым сообщениями ClientHello (5. Протокол Record). Поскольку сервер не хранит состояния, это будет выглядеть как первое принятое сообщение. При работе без сохранения состояний сервер **должен** игнорировать такие записи.

4.2.3. Алгоритмы подписи

TLS 1.3 поддерживает два расширения для индикации алгоритмов, которые могут применяться в цифровых подписях. Расширение signature_algorithms_cert применяется для подписей в сертификатах, а signature_algorithms, изначально включённое в TLS 1.2, - для подписей в сообщениях CertificateVerify. Ключи в сертификатах **должны** иметь тип, подходящий для используемых алгоритмов подписи. Для ключей RSA и подписей PSS возникает отдельный вопрос, рассмотренный ниже. При отсутствии расширения signature_algorithms_cert к подписям в сертификатах применяется расширение signature_algorithms. Клиенты, которые хотят, чтобы сервер подтвердил свою подлинность с помощью сертификата, **должны** передавать расширение signature_algorithms. Если сервер аутентифицируется по сертификату, а клиент не передал расширений signature_algorithms, сервер **должен** прервать согласование с сигналом missing_extension (см. 9.2. Обязательные расширения).

Расширение signature_algorithms_cert было добавлено для того, чтобы позволить реализациям, поддерживающим множество алгоритмов для сертификатов и самого TLS, чётко указать свои возможности. Реализациям TLS 1.2 также **следует** обрабатывать это расширение. Реализации с одинаковыми правилами для обоих случаев **могут** опускать расширение signature_algorithms_cert.

Поле extension_data в этом расширении содержит список SignatureSchemeList, показанный ниже.

```
enum {
    /* Алгоритмы RSASSA-PKCS1-v1_5 */
    rsa_pkcs1_sha256(0x0401),
    rsa_pkcs1_sha384(0x0501),
    rsa_pkcs1_sha512(0x0601),

    /* Алгоритмы ECDSA */
    ecdsa_secp256r1_sha256(0x0403),
    ecdsa_secp384r1_sha384(0x0503),
    ecdsa_secp521r1_sha512(0x0603),

    /* Алгоритмы RSASSA-PSS с открытым ключом OID rsaEncryption*/
    rsa_pss_rsae_sha256(0x0804),
    rsa_pss_rsae_sha384(0x0805),
    rsa_pss_rsae_sha512(0x0806),

    /* Алгоритмы EdDSA */
    ed25519(0x0807),
    ed448(0x0808),

    /* Алгоритмы RSASSA-PSS с открытым ключом OID RSASSA-PSS */
    rsa_pss_pss_sha256(0x0809),
    rsa_pss_pss_sha384(0x080a),
    rsa_pss_pss_sha512(0x080b),

    /* Унаследованные алгоритмы */
    rsa_pkcs1_sha1(0x0201),
    ecdsa_sha1(0x0203),

    /* Резервные коды */
    private_use(0xFE00..0xFFFF),
    (0xFFFF)
} SignatureScheme;

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

Примечание. Этот перечисляемый тип (enum) назван SignatureScheme потому, что уже имеется тип SignatureAlgorithm (TLS 1.2), который он заменяет. В тексте документа применяется термин «алгоритм подписи».

Каждое значение SignatureScheme включает один алгоритм подписи, который клиент хочет проверить. Значения указываются в порядке снижения предпочтительности. Отметим, что алгоритм подписи принимает на входе сообщение произвольного размера, а не дайджест. Для алгоритмов, работающих с дайджестом, в TLS нужно сначала задать алгоритм хэширования. Перечисленные в списке группы алгоритмов описаны ниже.

RSASSA-PKCS1-v1_5

Указывает алгоритм подписи, использующий RSASSA-PKCS1-v1_5 [RFC8017] с соответствующим алгоритмом хэширования, как определено в [SHS]. Эти значения относятся исключительно к подписям в сертификатах (4.4.2.2. Выбор сертификата сервера) и не определены для применения в подписанных сообщениях согласования TLS, хотя могут указываться в signature_algorithms и signature_algorithms_cert для совместимости с TLS 1.2.

ECDSA

Указывает алгоритм подписи, использующий ECDSA [ECDSA] с соответствующей кривой, как определено в ANSI X9.62 [ECDSA] и FIPS 186-4 [DSS], а также алгоритмом хэширования, как определено в [SHS]. Подпись представляется в форме структуры ECDSA-Sig-Value с DER-кодированием [X690] .

RSASSA-PSS RSAE

Указывает алгоритм подписи, использующий RSASSA-PSS [RFC8017] с функцией генерации маски 1. Дайджест, применяемый в функции генерации маски и подписываемый дайджест соответствуют алгоритму хэширования, определённому в [SHS]. Размер Salt **должен** совпадать с выходным размером алгоритма дайджеста. Если открытый ключ передаётся в сертификате X.509, он **должен** использовать rsaEncryption OID [RFC5280].

EdDSA

Указывает алгоритм подписи, использующий EdDSA, как указано в [RFC8032] и обновлениях. Отметим, что это соответствует алгоритмам PureEdDSA, а не вариантам prehash.

RSASSA-PSS PSS

Указывает алгоритм подписи, использующий RSASSA-PSS [RFC8017] с функцией генерации маски 1. Дайджест, применяемый в функции генерации маски, и подписываемый дайджест соответствуют алгоритму хэширования, определённому в [SHS]. Размер Salt **должен** совпадать с выходным размером алгоритма дайджеста. Если открытый ключ передаётся в сертификате X.509, он **должен** использовать RSASSA-PSS OID [RFC5756]. При использовании в подписях сертификатов для параметров алгоритма **должно** применяться DER-кодирование. При наличии параметров соответствующего открытого ключа параметры в подписи **должны** быть идентичны этим параметрам.

Устаревшие алгоритмы

Указывает алгоритмы, от использования которых отказались по причине наличия известных недостатков. В частности, это SHA-1 , который применяется в данном контексте с (1) RSA, использующим RSASSA-PKCS1-v1_5, или (2) ECDSA. Эти значения относятся исключительно к подписям, которые присутствуют в сертификатах (4.4.2.2. Выбор сертификата сервера), и не определены для подписанных сообщений согласования TLS, хотя **могут**

появляться в signature_algorithms и signature_algorithms_cert для совместимости с TLS 1.2. Конечным точкам не **следует** согласовывать эти алгоритмы, но они разрешены для совместимости со старыми версиями. Предлагающие эти значения клиенты **должны** перечислять их с самым низким приоритетом (после всех других алгоритмов в SignatureSchemeList). Серверам TLS 1.3 **недопустимо** предлагать сертификаты с подписями SHA-1, если можно создать действительную цепочку подписей без таких сертификатов (параграф 4.4.2.2).

Подписи в самоподписанных сертификатах и сертификатах с привязками доверия не проверяются, поскольку они являются началом пути сертификации ([RFC5280], параграф 3.2). Сертификаты, начинающиеся с пути сертификации, **могут** использовать алгоритм подписи, который не анонсируется как поддерживаемый в расширении signature_algorithms.

Отметим, что в TLS 1.2 это расширение определено иначе. Реализации TLS 1.3, желающие согласовать TLS 1.2, **должны** вести себя в соответствии с требованиями [RFC5246] при согласовании этой версии.

- В TLS 1.2 ClientHello это расширение **можно** пропускать.
- В TLS 1.2 расширение содержит пары хэш-подпись. Эти пары кодируются двумя октетами, поэтому значения SignatureScheme были выбраны с учётом кодирования TLS 1.2. Некоторые устаревшие пары остались не выделенными, а соответствующие алгоритмы запрещены в TLS 1.3. Их **недопустимо** предлагать или согласовывать в реализации. В частности, **недопустимо** применять MD5 [SLOTH], SHA-224 и DSA.
- Схемы подписи ECDSA соответствуют парам хэш-подпись TLS 1.2 ECDSA. Однако старая семантика не ограничивала кривые для подписей. При согласовании TLS 1.2 реализация **должна** быть готова к восприятию подписи, использующей любые кривые, которые анонсированы в расширении supported_groups.
- Реализации, анонсирующие поддержку RSASSA-PSS (обязательная в TLS 1.3), **должны** быть готовы воспринимать подписи, использующие эту схему, даже при согласованной версии TLS 1.2. В TLS 1.2 алгоритм RSASSA-PSS используется с шифрами RSA.

4.2.4. Удостоверяющие центры

Расширение certificate_authorities служит для указания удостоверяющих центров (Certificate authority или CA), с которым взаимодействует конечная точка и которые **следует** использовать приёмной стороне при выборе сертификата.

Телом расширения certificate_authorities является структура CertificateAuthoritiesExtension.

```
opaque DistinguishedName<1..2^16-1>;

struct {
    DistinguishedName authorities<3..2^16-1>;
} CertificateAuthoritiesExtension;
```

authorities

Список отличительных имён [X501] воспринимаемых удостоверяющих центров в формате DER [X690]. Этот список задаёт отличительные имена для привязок доверия и подчинённых CA, поэтому сообщение может служить для описания известных привязок доверия, а также желаемого пространства полномочий.

Клиент **может** передавать расширение certificate_authorities в сообщении ClientHello, а сервер **может** передавать его в CertificateRequest.

Расширение trusted_ca_keys [RFC6066], которое служит для аналогичных целей, но более сложно, не применяется в TLS 1.3 (хотя может появляться в сообщениях ClientHello от клиентов, предлагающих прежние версии TLS).

4.2.5. Фильтры OID

Расширение oid_filters позволяет серверам представить набор пар OID-значение для сопоставления с сертификатами клиентов. Это расширение, если сервер его применяет, **должно** передаваться только в сообщении CertificateRequest.

```
struct {
    opaque certificate_extension_oid<1..2^8-1>;
    opaque certificate_extension_values<0..2^16-1>;
} OIDFilter;

struct {
    OIDFilter filters<0..2^16-1>;
} OIDFilterExtension;
```

filters

Список OID [RFC5280] расширений сертификатов с разрешёнными значениями в формате DER [X690]. Некоторые OID могут иметь несколько значений (например, Extended Key Usage). Если сервер указал непустой список фильтров, включенный в отклик сертификат клиента **должен** содержать все указанные OID расширений, которые клиент распознал. Для каждого OID расширения, распознанного клиентом, все указанные значения **должны** быть представлены в клиентском сертификате (сертификат **может** иметь и другие значения). Однако клиент **должен** игнорировать и пропускать нераспознанные OID расширений сертификатов. Если клиент игнорирует некоторые их требуемых OID и представляет сертификат, который не соответствует запросу, сервер **может** по своему усмотрению продолжить соединение без аутентификации клиента или прервать согласование с сигналом unsupported_certificate. Любое данное значение OID **недопустимо** включать в список фильтров более одного раза.

PKIX RFC определяют множество OID расширений сертификатов и соответствующих им значений типов. В зависимости от типа совпадение значений расширений сертификатов не обязательно будет побитовым. Предполагается, что реализации TLS будут опираться на свои библиотеки PKI для выбора сертификатов с использованием OID расширений.

Этот документ определяет правила сопоставления для 2 стандартных расширений сертификатов, заданных в [RFC5280].

- Расширение Key Usage в сертификате соответствует запросу, когда все биты использования ключа из запроса указаны также в расширении сертификата Key Usage.

- Расширение Extended Key Usage в сертификате соответствует запросу, когда все OID назначения ключа из запроса имеются также в расширении сертификата Extended Key Usage. В запросе **недопустимо** применять anyExtendedKeyUsage OID.

Правила для других расширений сертификатов могут быть определены в отдельных документах.

4.2.6. Аутентификация клиента после согласования

Расширение post_handshake_auth служит для индикации желания клиента выполнить аутентификацию post-handshake (4.6.2. Аутентификация после согласования). Серверам **недопустимо** передавать CertificateRequest после согласования клиентам, которые не предлагают это расширение. Серверам **недопустимо** передавать это расширение.

```
struct {} PostHandshakeAuth;
```

Поле extension_data в расширении post_handshake_auth имеет нулевой размер.

4.2.7. Поддерживаемые группы

При передаче клиентом расширения supported_groups оно указывает именованные группы, которые клиент поддерживает для обмена ключами, упорядоченные по снижению предпочтительности.

Примечание. В версиях TLS до TLS 1.3 это расширение называлось elliptic_curves и содержало лишь группы эллиптических кривых (см. [RFC8422] и [RFC7919]). Расширение применялось также для согласования кривых ECDSA. Алгоритмы подписи сейчас согласуются независимо (4.2.3. Алгоритмы подписи).

Поле extension_data этого расширения содержит значение NamedGroupList, показанное ниже.

```
enum {
    /* Группы эллиптических кривых (ECDHE) */
    secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),
    x25519(0x001D), x448(0x001E),

    /* Группы конечных полей (DHE) */
    ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),
    ffdhe6144(0x0103), ffdhe8192(0x0104),

    /* Резервные коды */
    ffdhe_private_use(0x01FC..0x01FF),
    ecdhe_private_use(0xFE00..0xFEFF),
    (0xFFFF)
} NamedGroup;

struct {
    NamedGroup named_group_list<2..2^16-1>;
} NamedGroupList;
```

Elliptic Curve Groups (ECDHE)

Указывает поддержку соответствующей именованной кривой, определённой в FIPS 186-4 [DSS] или [RFC7748]. Значения 0xFE00 - 0xFEFF зарезервированы для приватного использования [RFC8126].

Finite Field Groups (DHE)

Указывает поддержку соответствующей группы конечных полей, определённой в [RFC7919]. Значения 0x01FC - 0x01FF зарезервированы для приватного использования.

Элементы в named_group_list упорядочиваются по снижению предпочтительности для отправителя.

Начиная с TLS 1.3, серверам разрешено передавать клиенту расширение supported_groups. Клиентам **недопустимо** действовать в соответствии с информацией из supported_groups до завершения согласования, но они могут использовать информацию полученную после успешного согласования для смены групп, используемых в расширении key_share при последующих соединениях. Если на сервере есть группа, которую он предпочитает указанным в расширении key_share, но сервер все равно готов воспринять ClientHello, ему **следует** передать supported_groups для обновления клиентского представления о предпочтениях. В это расширение **следует** включать все группы, поддерживаемые сервером, независимо от их поддержки клиентом в данный момент.

4.2.8. Совместное использование ключа

Расширение key_share содержит криптографические параметры конечной точки. Клиенты могут передавать пустой вектор client_shares для запроса выбора группы сервером за счёт дополнительного кругового обхода (4.1.4. Запрос повтора Hello).

```
struct {
    NamedGroup group;
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;
```

group

Именованная группа для ключа.

key_exchange

Информация обмена ключами. Содержимое этого поля задаёт указанная группа и соответствующее определение. Параметры Finite Field Diffie-Hellman [DH76] описаны в параграфе 4.2.8.1. Параметры Diffie-Hellman, а Elliptic Curve Diffie-Hellman - в параграфе 4.2.8.2. Параметры ECDHE.

В сообщении ClientHello поле extension_data этого расширения содержит значение KeyShareClientHello.

```
struct {
    KeyShareEntry client_shares<0..2^16-1>;
} KeyShareClientHello;
```

client_shares

Список предлагаемых значений KeyShareEntry в порядке снижения предпочтительности для клиента.

Этот вектор **может** быть пустым, если клиент запрашивает HelloRetryRequest. Каждое значение KeyShareEntry **должно** соответствовать группе, предложенной в расширении supported_groups, и порядок значений **должен** соответствовать порядку групп. Однако значения **могут** не образовывать непрерывного подмножества расширения supported_groups и даже наиболее предпочтительные группы **могут** быть опущены. Такая ситуация может возникать, если самые предпочтительные группы являются новыми и вряд ли поддерживаются достаточно широко, чтобы сделать для них эффективным использование заранее созданных общих ключей.

Клиенты могут предлагать столько значений KeyShareEntry, сколько они указали поддерживаемых групп, каждая из которых представляет один набор параметров обмена ключами. Например, клиент может предложить совместное использование нескольких эллиптических кривых или групп FFDHE. Значения key_exchange для каждой KeyShareEntry **должны** генерироваться независимо. Клиентам **недопустимо** предлагать несколько значений KeyShareEntry для одной группы и **недопустимо** предлагать значения KeyShareEntry для групп, не указанных в клиентском расширении supported_groups. Серверы **могут** проверять эти правила и при нарушении прерывать согласование с сигналом illegal_parameter.

В сообщении HelloRetryRequest поле extension_data этого расширения содержит значение KeyShareHelloRetryRequest.

```
struct {
    NamedGroup selected_group;
} KeyShareHelloRetryRequest;
```

selected_group

Поддерживаемая обеими сторонами группа, которую сервер намерен согласовать и запрашивает повтор ClientHello/KeyShare.

При получении этого расширения в HelloRetryRequest клиент **должен** проверить, что (1) поле selected_group соответствует группе, которая была указана в расширении supported_groups исходного ClientHello, и (2) не соответствует группе, указанной в расширении key_share исходного ClientHello. Если любое из этих условий не выполняется, клиент **должен** прервать согласование с сигналом illegal_parameter. В остальных случаях при отправке нового ClientHello клиент **должен** заменить исходное расширение key_share другим, которое содержит лишь новое значение KeyShareEntry для группы, указанной в поле selected_group вызвавшего процесс сообщения HelloRetryRequest.

В сообщении ServerHello поле extension_data этого расширения содержит значение KeyShareServerHello.

```
struct {
    KeyShareEntry server_share;
} KeyShareServerHello;
```

server_share

Одно значение KeyShareEntry, относящееся к одной из общих с клиентом групп.

При использовании (EC)DHE для организации ключей сервер предлагает в точности 1 значение KeyShareEntry в ServerHello. Это значение **должно** быть из той же группы, что и KeyShareEntry, предложенное клиентом и выбранное сервером для согласованного обмена ключами. Серверу **недопустимо** передавать KeyShareEntry для какой-либо группы, не указанной в клиентском расширении supported_groups, а также **недопустимо** передавать KeyShareEntry при использовании psk_ke PskKeyExchangeMode. Если используется организация ключей (EC)DHE и клиентом получено сообщение HelloRetryRequest с расширением key_share, клиент **должен** убедиться в том, что выбранная группа NamedGroup в ServerHello совпадает с группой в HelloRetryRequest. Если это не выполняется, сервер **должен** прервать согласование с сигналом illegal_parameter.

4.2.8.1. Параметры Diffie-Hellman

Параметры Diffie-Hellman [DH76] для клиентов и серверов кодируются в неинтерпретируемом (opaque) поле key_exchange записи KeyShareEntry в структуре KeyShare. Поле содержит неинтерпретируемое открытое значение Diffie-Hellman ($Y = g^X \bmod p$) для указанной группы (см. определения в [RFC7919]), представленное в виде целого числа (big-endian), дополняемого слева нулями до размера p байтов.

> Примечание. Для данной группы Diffie-Hellman результат заполнения для всех открытых ключей будет давать одинаковый размер.

Партнёры **должны** проверить открытый ключ другой стороны Y, который должен иметь значение $1 < Y < p-1$. Эта проверка гарантирует, что удалённый партнёр ведёт себя корректно и не вынуждает локальную систему входить в мелкую подгруппу.

4.2.8.2. Параметры ECDHE

Параметры ECDHE для клиентов и серверов кодируются в неинтерпретируемом (opaque) поле key_exchange записи KeyShareEntry в структуре KeyShare.

Для secp256r1, secp384r1 и secp521r1 содержимое является сериализованным представлением структуры

```
struct {
    uint8 legacy_form = 4;
    opaque X[coordinate_length];
    opaque Y[coordinate_length];
} UncompressedPointRepresentation;
```

X и Y, соответственно, являются двоичными представлениями значений x и y с сетевым порядком байтов. Здесь нет внутренних маркеров размера, поэтому каждое значение занимает число октетов, определяемое параметрами кривой. Для P-256 это означает, что X и Y используют по 32 октета с дополнением нулями слева при необходимости. Для P-384 они будут занимать по 48 октетов, для P-521 - по 66.

Для кривых secp256r1, secp384r1 и secp521r1 партнёры **должны** проверять, что значение открытого ключа другой стороны Q является действительной точкой эллиптической кривой. Подходящие процедуры проверки определены в параграфе 4.3.7 [ECDSA] и параграфе 5.6.2.3 [KEYAGREEMENT]. Проверка состоит из трёх этапов - (1) Q не является точкой в бесконечности (O), (2) для Q = (x, y) оба целых числа x и y находятся в корректном интервале и (3) пара (x, y) является корректным решением для уравнения эллиптической кривой. Для этих кривых от разработчиков не нужна проверка принадлежности к корректной подгруппе.

Для X25519 и X448 содержимым открытых значений являются байтовые строки ввода и вывода соответствующих функций, определённых в [RFC7748] - 32 байта для X25519 и 56 для X448.

Примечание. Версии TLS до 1.3 разрешают согласование формата точек, в версии TLS 1.3 от этого отказались в пользу одного формата для каждой кривой.

4.2.9. Режимы обмена ключами PSK

Для использования PSK клиент **должен** также передать расширение psk_key_exchange_modes. Семантика этого расширения заключается в том, что клиент поддерживает использование PSK только в режимах, которые ограничиваются набором из числа включённых в данное сообщение ClientHello и поддерживаемых сервером через NewSessionTicket.

Клиент **должен** представить расширение psk_key_exchange_modes, если он предлагает расширение pre_shared_key. Если клиент предлагает pre_shared_key без расширения psk_key_exchange_modes, сервер **должен** прервать согласование. Серверу **недопустимо** выбирать режим обмена ключами, не указанный клиентом. Это расширение также ограничивает режимы, используемые при восстановлении PSK. Серверам **не следует** передавать NewSessionTicket с квитанцией, которая не совместима с анонсированными режимами, однако если сервер делает это, эффект должен проявляться лишь в виде отказа при попытке восстановления клиентом.

Серверам **недопустимо** передавать расширение psk_key_exchange_modes.

```
enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeMode;

struct {
    PskKeyExchangeMode ke_modes<1..255>;
} PskKeyExchangeModes;
```

psk_ke

Организация ключей с использованием только PSK. В этом режиме серверу **недопустимо** представлять значение key_share.

psk_dhe_ke

Организация ключей на основе PSK с (EC)DHE. В этом режиме сервер **должен** представлять значения key_share, как описано в параграфе 4.2.8. Совместное использование ключа.

Выделяемые в будущем значения должны гарантировать, что передаваемые протокольные сообщения однозначно указывают выбранный сервером режим. В настоящий момент это указывается наличием key_share в ServerHello.

4.2.10. Индикация ранних данных

Когда применяется PSK и ранние данные для этого PSK разрешены, клиент может передавать данные приложений в первой отправке сообщений. Если клиент делает это, он **должен** представить оба расширения pre_shared_key и early_data.

Поле extension_data этого расширения содержит значение EarlyDataIndication.

```
struct {} Empty;

struct {
    select (Handshake.msg_type) {
        case new_session_ticket:   uint32 max_early_data_size;
        case client_hello:         Empty;
        case encrypted_extensions: Empty;
    };
} EarlyDataIndication;
```

Использование поля max_early_data_size подробно описано в параграфе 4.6.1. Сообщение NewSessionTicket.

Параметры для данных 0-RTT (версия, симметричный шифр, протокол ALPN[1] [RFC7301] и т. п.) связаны с применяемым PSK. Для внешних PSK связанные значения представляются вместе с ключом. Для PSK, организованных через сообщения NewSessionTicket, связанные значения согласуются в соединении, где организуется PSK. Ключ PSK, используемый для шифрования ранних данных, **должен** быть первым PSK в клиентском расширении pre_shared_key.

Для PSK, предоставленных с помощью NewSessionTicket, сервер **должен** убедиться, что возраст квитанции для отождествления выбранного PSK (рассчитывается вычитанием ticket_age_add из PskIdentity.obfuscated_ticket_age по модулю 232) находится в пределах небольшого допуска от времени, прошедшего с момента выдачи квитанции (8. 0-RTT и Anti-Replay). Если это не так, серверу **следует** обработать согласование, но отвергнуть 0-RTT, и **не следует** предпринимать других действий, предполагающих свежесть данного ClientHello.

Сообщения 0-RTT, передаваемые в первой отправке, имеют те же типы (шифрованного) содержимого, что и сообщения того же типа в других отправках (handshake и application_data), но защищены другими ключами. После получения серверного сообщения Finished, если сервер принял ранние данные, будет передаваться сообщение EndOfEarlyData для индикации смены ключа. Это сообщение шифруется с ключами трафика 0-RTT.

Сервер, получивший расширение early_data, **должен** выбрать один из 3 вариантов поведения, указанных ниже.

- Игнорировать расширение и вернуть обычный отклик 1-RTT. Затем сервер пропускает ранние данные, пытаясь снять защиту полученных записей с использованием ключа трафика согласования и отбрасывая данные, защиту которых снять не удалось (вплоть до настроенного max_early_data_size). После успешного снятия защиты с записи она считается началом второй отправки клиента и сервер обрабатывает её как обычное согласование 1-RTT.
- Запросить у клиента передачу другого ClientHello, отправив ему HelloRetryRequest. Клиенту **недопустимо** включать расширение early_data в последующее сообщение ClientHello. Затем сервер игнорирует ранние

[1]Application-Layer Protocol Negotiation - согласование протокола прикладного уровня.

данные, пропуская все записи с внешним содержимым типа application_data (указывает, что данные зашифрованы) вплоть до настроенного max_early_data_size.

- Возвратить своё расширение early_data в EncryptedExtensions, указав, что оно предназначено для обработки ранних данных. Для сервера невозможно воспринять лишь часть сообщений с ранними данными. Хотя сервер передаёт сообщение о восприятии ранних данных, сами эти данные могут ещё находиться в сети, когда сервер генерирует это сообщение.

Для восприятия ранних данных сервер **должен** иметь воспринятый шифронабор PSK и выбранный первый ключ, предложенный в клиентском расширении pre_shared_key. Кроме того, он **должен** проверить, что перечисленные ниже значения совпадают со связанными с выбранным PSK.

- Номер версии TLS.
- Выбранный шифронабор.
- Выбранный протокол ALPN [RFC7301] (если он есть).

Эти требования являются расширением требований, которые нужны для согласования 1-RTT с использованием рассматриваемого PSK. Для установленных извне PSK связанные значения представляются вместе с ключом. Для PSK, организованных через сообщение NewSessionTicket, связанные значения согласуются в соединении в процессе организации квитанции.

Будущие расширения **должны** определять взаимодействие с 0-RTT.

Если любое из приведённых выше условий не выполняется, серверу **недопустимо** отвечать с расширением и он должен отбросить все данные первой отправки, используя один из двух указанных выше механизмов (возвращаясь тем самым к 1-RTT или 2-RTT). Если клиент пытается выполнить согласование 0-RTT, но сервер отвергает его, у сервера обычно нет ключей защиты записи 0-RTT и он должен взамен использовать «пробную» расшифровку (с помощью ключей согласования 1-RTT или путём поиска открытого ClientHello в случае HelloRetryRequest), чтобы найти первое сообщение, не являющееся 0-RTT.

Если сервер решает принять расширение early_data, он **должен** соответствовать тем же требованиям к обработке ошибок, которые заданы для ранних данных. В частности, если серверу не удаётся расшифровать запись 0-RTT, следующую за принятым расширением early_data, он **должен** прервать соединение с сигналом bad_record_mac (5.2. Защита данных Record).

Если сервер отвергает расширение early_data, клиентское приложение **может** выбрать повтор передачи данных приложения, переданных как ранние данные, после завершения согласования. Отметим, что автоматический повтор передачи ранних данных может приводить к ошибочным предположениям о состоянии соединения. Например, когда согласованное соединение выбирает другой протокол ALPN, нежели был использован для ранних данных, приложению может потребоваться создание других сообщений. Точно так же, если для ранних данных использовалось допущение о состоянии соединения, это может приводить к ошибке по завершении согласования.

Реализации TLS **не следует** автоматически повторять передачу ранних данных, приложения имеют лучшую позицию для решения вопроса о повторе передачи. Реализации TLS **недопустимо** автоматически повторять передачу ранних данных, если для согласованного соединения выбран другой протокол ALPN.

4.2.11. Расширение PSK

Расширение pre_shared_key служит для согласования отождествления заранее известного ключа, который будет применяться в данном согласовании при организации ключа PSK.

Поле extension_data в этом расширении содержит значение PreSharedKeyExtension.

```
struct {
    opaque identity<1..2^16-1>;
    uint32 obfuscated_ticket_age;
} PskIdentity;

opaque PskBinderEntry<32..255>;

struct {
    PskIdentity identities<7..2^16-1>;
    PskBinderEntry binders<33..2^16-1>;
} OfferedPsks;

struct {
    select (Handshake.msg_type) {
        case client_hello: OfferedPsks;
        case server_hello: uint16 selected_identity;
    };
} PreSharedKeyExtension;
```

identity

Метка ключа. Например, квитанция (B.3.4. Создание квитанции) или метка полученного извне общего ключа.

obfuscated_ticket_age

Запутанная версия возраста ключа. В параграфе 4.2.11.1 описано формирование этого значения для элементов, созданных с помощью сообщения NewSessionTicket. Для полученных извне элементов **следует** применять obfuscated_ticket_age = 0, а серверы **должны** игнорировать это значение.

identities

Список элементов, которые клиент хочет согласовать с сервером. При передаче вместе с расширением early_data (4.2.10. Индикация ранних данных) первый элемент списка используется для данных 0-RTT.

binders

Последовательность значений HMAC (1 для каждого элемента в списке identities с сохранением порядка), рассчитанных, как описано ниже.

selected_identity

Выбранный сервером элемент, указанный номером в клиентском списке identities (отсчёт с 0).

Каждый PSK связан с одним алгоритмом хэширования. Для PSK, полученных с помощью механизма квитанций (4.6.1. Сообщение NewSessionTicket), это алгоритм KDF Hash в соединении, где была создана квитанция. Для полученных извне PSK алгоритм Hash **должен** быть установлен при создании PSK или используется SHA-256, если такой алгоритм не задан. Сервер **должен** гарантировать совместимость PSK (если он есть) и шифра.

В TLS версий до TLS 1.3 значение SNI[1] было предназначено для связывания сессии (раздел 3 [RFC6066]), при этом от сервера требовалось обеспечить соответствие связанного с сессией SNI одному из значений, заданных при согласовании восстановления. Однако в действительности реализации оказались непоследовательны при выборе одного из двух предлагаемых значений SNI и это приводило к тому, что требования совместимости фактически применялись клиентами. В TLS 1.3 значение SNI всегда явно задаётся в согласовании восстановления и серверу не требуется связывать значение SNI с квитанцией. Однако клиентам **следует** сохранять SNI с PSK для выполнения требований параграфа 4.6.1. Сообщение NewSessionTicket.

> Примечание для разработчиков. Когда восстановление сеанса является основным применением PSK, самым простым способом реализации требований к соответствию PSK и шифра является согласование сначала шифра, а затем исключение явно несовместимых PSK. Все неизвестные PSK (например, отсутствующие в базе данных или зашифрованные неизвестным ключом) **следует** просто игнорировать. Если подходящих PSK не найдено, серверу **следует** по возможности выполнить согласование без PSK. Если важна совместимость с прежними версиями, выбор шифра **следует** делать с учётом предоставленных клиентом PSK, полученных извне.

До восприятия организации ключа PSK сервер **должен** проверить соответствующее значение привязки (binder, параграф 4.2.11.2). Если такого значения нет или оно не действительно, сервер **должен** прервать согласование. Серверам **не следует** пытаться проверить множество привязок, скорее **следует** выбрать один PSK и проверить только привязку, соответствующую этому PSK. В параграфе 8.2. Запись ClientHello и приложении E.6. Раскрытие отождествления PSK приведено обоснование этого с точки зрения безопасности. Для восприятия организации PSK сервер передаёт расширение pre_shared_key, указывающее выбранный объект.

Клиент **должен** убедиться, что полученное от сервера значение selected_identity находится в диапазоне, указанном клиентом, сервер выбрал шифр, указывающий Hash, связанный с PSK, а серверное расширение key_share присутствует, если оно затребовано расширением psk_key_exchange_modes в ClientHello. Если эти значения не согласованы, клиент **должен** прервать согласование с сигналом illegal_parameter.

Если сервер представил расширение early_data, клиент **должен** убедиться, что сервер указал selected_identity = 0 и в противном случае он **должен** прервать согласование с сигналом illegal_parameter.

Расширение pre_shared_key **должно** быть последним в ClientHello (это упрощает реализацию, как описано ниже). Серверы **должны** проверять это и при наличии других расширений после указанного прерывать согласование с сигналом illegal_parameter.

4.2.11.1. Возраст квитанции

Возраст квитанции с точки зрения клиента определяется временем с момента получения NewSessionTicket. Клиентам **недопустимо** пытаться использовать квитанции с возрастом больше значения ticket_lifetime в квитанции. Поле obfuscated_ticket_age в каждом PskIdentity содержит запутанную версию возраста квитанции, сформированную из возраста в миллисекундах, сложенного со значением ticket_age_add из квитанции (4.6.1. Сообщение NewSessionTicket) по модулю 232. Это добавление препятствует пассивным наблюдателям при сопоставлении соединений, если квитанции не используются многократно. Отметим, что поле ticket_lifetime в сообщении NewSessionTicket указывает время в секундах, а obfuscated_ticket_age - в миллисекундах. Поскольку сроки действия квитанций ограничены неделей, 32 битов достаточно для представления любого срока действия даже в миллисекундах.

4.2.11.2. Привязка PSK

Значение привязки PSK формирует связь между PSK и текущим согласованием, а также между согласованием, где был создан PSK (через сообщение NewSessionTicket) и текущим согласованием. Каждая запись в списке привязок рассчитывается как HMAC для хэша стенограммы (4.4.1. Transcript-Hash), включающего часть ClientHello до поля PreSharedKeyExtension.identities, включительно, т. е. значение учитывает ClientHello без самих привязок. Поля размера для сообщения (включая общий размер, размеры блоков расширений и pre_shared_key) установлены как при наличии привязок корректного размера.

PskBinderEntry рассчитывается так же, как сообщение Finished (4.4.4. Сообщение Finished), но в качестве BaseKey используется значение binder_key, выведенное через планирование ключей из соответствующего PSK, который будет предлагаться (7.1. Планирование ключей).

Если согласование включает HelloRetryRequest, сообщения ClientHello и HelloRetryRequest включаются в стенограмму вместе с новым ClientHello. Например, если клиент передаёт ClientHello1, привязка рассчитывается для Transcript-Hash(Truncate(ClientHello1)), где Truncate() удаляет список привязок из ClientHello. Если сервер отвечает сообщением HelloRetryRequest и клиент после этого шлёт ClientHello2, его привязка рассчитывается в форме Transcript-Hash(ClientHello1, HelloRetryRequest, Truncate(ClientHello2)).

Полные сообщения ClientHello1/ClientHello2 включаются во все другие расчёты хэш-значений согласования. Отметим, что в первой отправке Truncate(ClientHello1) хэшируется напрямую, но во второй хэшируется ClientHello1 и затем повторно инжектируется как сообщение message_hash (4.4.1. Transcript-Hash).

4.2.11.3. Порядок обработки

Клиентам разрешено передавать данные 0-RTT до получения от сервера сообщения Finished, после этого передавая сообщение EndOfEarlyData, за которым следует остальная часть согласования. Для предотвращения блокировок в случае восприятия early_data сервер **должен** обработать клиентское сообщение ClientHello и сразу после этого отправлять свои сообщения вместо ожидания от клиента сообщения EndOfEarlyData до отправки своего ServerHello.

[1]Server Name Intdication - индикация имени сервера.

4.3. Параметры сервера

Следующие два сообщения от сервера (EncryptedExtensions и CertificateRequest) содержат информацию, которая задаёт остальную часть согласования. Эти сообщения шифруются с использованием ключей, выведенных из server_handshake_traffic_secret.

4.3.1. Шифрованные расширения

Во всех согласованиях сервер **должен** передавать сообщение EncryptedExtensions сразу после сообщения ServerHello. Это будет первым сообщением, шифруемым с ключами, выведенными из server_handshake_traffic_secret.

Сообщение EncryptedExtensions содержит расширения, которые могут быть защищены (т. е. не требуют организации криптографического контекста), но не связаны с индивидуальными сертификатами. Клиент **должен** проверить наличие в EncryptedExtensions запрещённых расширений и при обнаружении таковых **должен** прервать согласование с сигналом illegal_parameter.

Ниже показана структура сообщения.

```
struct {
    Extension extensions<0..2^16-1>;
} EncryptedExtensions;
```

extensions

Список расширений (см. таблицу в параграфе 4.2).

4.3.2. Запрос сертификата

При аутентификации по сертификатам сервер **может** запросить сертификат у клиента. При передаче такого сообщения оно **должно** следовать за EncryptedExtensions.

Структура сообщения показана ниже.

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

certificate_request_context

Неинтерпретируемая (opaque) строка, которая указывает запрос сертификата и будет возвращаться в клиентском сообщении Certificate. Значение certificate_request_context **должно** быть уникальным в рамках соединения (чтобы предотвратить повторное использование сообщений). Это поле **нужно** сохранять пустым, если оно не применяется в аутентификационных обменах после согласования, как описано в параграфе 4.6.2. При запросе аутентификации после согласования серверу **следует** сделать контекст непредсказуемым для клиента (например, путём случайной генерации), чтобы воспрепятствовать атакам за счёт получения временного доступа к секретному ключу клиента из заранее созданных действительных сообщений CertificateVerify.

extensions

Набор расширений, описывающий параметры запрашиваемого сертификата. Расширение signature_algorithms **должно** быть указано, а другие расширения являются необязательными. Клиенты **должны** игнорировать неопознанные расширения.

В предшествующих версиях TLS сообщение CertificateRequest содержало список алгоритмов подписи и удостоверяющих центров, которые сервер будет воспринимать. В TLS 1.3 первый список указывается расширением signature_algorithms и необязательным расширением signature_algorithms_cert, а второй - расширением certificate_authorities (4.2.4. Удостоверяющие центры).

При аутентификации с помощью PSK серверу **недопустимо** передавать CertificateRequest в основном согласовании, хотя он **может** передать это сообщение в аутентификации post-handshake (4.6.2. Аутентификация после согласования) при условии, что клиент передал post_handshake_auth (4.2.6. Аутентификация клиента после согласования).

4.4. Аутентификационные сообщения

Как отмечено в разделе 2, TLS в общем случае использует для проверки подлинности, подтверждения ключей и защиты целостности согласования набор сообщений Certificate, CertificateVerify и Finished (для привязки PSK также выполняется аналогичное подтверждение ключей). Эти три сообщения всегда передаются последними в отправке согласования. Сообщения Certificate и CertificateVerify передаются лишь в некоторых случаях, описанных ниже. Сообщение Finished передаётся всегда как часть блока аутентификации. Эти сообщения шифруются с ключом, выведенным из [sender]_handshake_traffic_secret.

Для расчётов аутентификационных сообщений используются однотипные входные данные:

- сертификат и ключ подписи;
- контекст согласования, состоящих из набора сообщений, включаемых в хэширование стенограммы;
- базовый ключ, применяемый для расчёта ключа MAC.

На основе этих данных создаются сообщения, включающие перечисленные ниже элементы.

Certificate

Сертификат, который будет использоваться для аутентификации, и все поддерживающие сертификаты в цепочке. Аутентификация клиентов на основе сертификатов недоступна в потоке согласования PSK (включая 0-RTT).

CertificateVerify

Подпись для значения Transcript-Hash(Handshake Context, Certificate).

Finished

MAC для значения Transcript-Hash(Handshake Context, Certificate, CertificateVerify) с использованием ключа MAC, выведенного из базового ключа (Base Key).

В таблице приведён контекст согласования (Handshake Context) и базовый ключ MAC для каждого варианта.

Режим	*Контекст согласования*	*Базовый ключ*
Server	ClientHello ... после EncryptedExtensions/CertificateRequest	server_handshake_traffic_secret
Client	ClientHello ... после серверного Finished/EndOfEarlyData	client_handshake_traffic_secret
Post-Handshake	ClientHello ... клиентские Finished + CertificateRequest	client_application_traffic_secret_N

4.4.1. Transcript-Hash

Многие криптографические расчёты в TLS используют хэш стенограммы (transcript hash). Это значение рассчитывается путём хеширования конкатенации всех входящих в согласование сообщений, включая заголовок, указывающий тип и размер сообщений согласования, но не включая заголовков уровня записей.

```
Transcript-Hash(M1, M2, ... Mn) = Hash(M1 || M2 || ... || Mn)
```

Исключением из общего правила является случай, когда сервер отвечает на ClientHello сообщением HelloRetryRequest - значение ClientHello1 заменяется синтетическим сообщением согласования типа message_hash, содержащим Hash(ClientHello1).

```
Transcript-Hash(ClientHello1, HelloRetryRequest, ... Mn) =
    Hash(message_hash ||        /* Тип согласования */
         00 00 Hash.length  ||  /* Размер сообщения Handshake в байтах */
         Hash(ClientHello1) ||  /* Хэш-значение ClientHello1 */
         HelloRetryRequest  || ... || Mn)
```

Такая конструкция создана для того, чтобы позволить серверу выполнять HelloRetryRequest без учёта состояния, сохраняя лишь хэш ClientHello1 в значении cookie вместо экспорта всего промежуточного состояния хэша (4.2.2. Cookie).

Для конкретности хэш стенограммы всегда создаётся для указанной далее последовательности сообщений согласования, начиная с первого ClientHello и включая лишь те сообщения, которые были были переданы: ClientHello, HelloRetryRequest, ClientHello, ServerHello, EncryptedExtensions, серверные сообщения CertificateRequest, Certificate, CertificateVerify и Finished, EndOfEarlyData, клиентские сообщения Certificate, CertificateVerify и Finished.

В общем случае реализация может хэшировать стенограмму путём сохранения хэш-значения текущей стенограммы с использованием согласованной функции. Однако следует отметить, что последующие аутентификации после согласования не включают друг друга, а учитывают лишь сообщения до завершения основного согласования.

4.4.2. Сообщение Certificate

Это сообщение передаёт партнёру цепочку сертификатов конечной точки.

Сервер **должен** передавать Certificate всякий раз, когда согласованный метод обмена ключами применяет сертификаты для проверки подлинности (все методы, определённые в этом документе, за исключением PSK).

Клиент **должен** передавать сообщение Certificate тогда и только тогда, когда сервер запросил аутентификацию сообщением CertificateRequest (4.3.2. Запрос сертификата). Если сервер запросил проверку подлинности клиента, но подходящий сертификат недоступен, клиент **должен** передать сообщение Certificate без сертификатов (т. е. с полем certificate_list размером 0). Сообщение Finished **должно** передаваться независимо от того, является ли сообщение Certificate пустым.

Структура сообщения показана ниже.

```
enum {
    X509(0),
    RawPublicKey(2),
    (255)
} CertificateType;

struct {
    select (certificate_type) {
        case RawPublicKey:
          /* Из RFC 7250 ASN.1_subjectPublicKeyInfo */
          opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

        case X509:
          opaque cert_data<1..2^24-1>;
    };
    Extension extensions<0..2^16-1>;
} CertificateEntry;

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

certificate_request_context

Если сообщение является ответом на CertificateRequest, поле содержит значение certificate_request_context из запроса. В остальных случаях (аутентификация сервера) его **нужно** оставлять пустым (размера 0).

certificate_list

Цепочка структур CertificateEntry, каждая из которых содержит сертификат и набор расширений.

extensions

Набор расширений в CertificateEntry, формат которых определён в параграфе 4.2. Расширения. Пригодными расширениями для сертификатов сервера сейчас являются OCSP Status [RFC6066] и SignedCertificateTimestamp [RFC6962], а в будущем могут быть определены расширения и для этого сообщения. Расширения в сообщении Certificate от сервера **должны** соответствовать расширениям в сообщении ClientHello. Расширения в сообщении Certificate от клиента **должны** соответствовать расширениям в сообщении CertificateRequest от сервера. Если расширение относится ко всей цепочке, его **следует** включать в первую запись CertificateEntry.

Если соответствующее расширение типа сертификата (server_certificate_type или client_certificate_type) не согласовано в EncryptedExtensions или тип сертификата X.509 не был согласован, каждая запись CertificateEntry содержит

сертификат X.509 в DER-представлении. Сертификат отправителя **должен** быть первым в списке CertificateEntry. Всем остальным сертификатам **следует** напрямую удостоверять своего предшественника. Поскольку проверка сертификатов требует независимого распространения привязок доверия, указывающий привязку сертификат **может** не включаться в цепочку при условии, что поддерживающему партнёру известны все пропущенные сертификаты.

> Примечание. До TLS 1.3 упорядочение certificate_list требовало, чтобы каждый сертификат удостоверял своего непосредственного предшественника, однако часть реализаций допускала определённую гибкость. Серверы иногда передают одновременно текущий и отменённый предыдущий сертификат для учёта перехода, а другие просто некорректно настроены, но эти случаи никогда нельзя было проверить корректно. Для максимальной совместимости всем реализациям **следует** быть готовыми к обработке избыточных сертификатов и их произвольному порядку для любой версии TLS, но сертификат конечной точки всегда **должен** быть первым.

Если был согласован тип сертификата RawPublicKey, поле certificate_list **должно** содержать не более 1 записи CertificateEntry, которая включает значение ASN1_subjectPublicKeyInfo, как определено в разделе 3 [RFC7250].

Сертификаты типа OpenPGP [RFC6091] **недопустимо** использовать в TLS 1.3.

Поле certificate_list от сервера **должно** быть непустым. Клиент будет передавать пустой список certificate_list, если у него нет подходящего сертификата для ответа на запрос сертификации от сервера.

4.4.2.1. Статус OCSP и расширения SCT

В [RFC6066] и [RFC6961] определены расширения для согласования передачи сервером откликов OCSP. В TLS 1.2 и более ранних версиях сервер отвечает пустым расширением для индикации согласования этого расширения, а информация OCSP передаётся в сообщении CertificateStatus. В TLS 1.3 серверная информация OCSP передаётся в расширении CertificateEntry, содержащем связанный сертификат. В частности, телом расширения status_request от сервера **должна** быть структура CertificateStatus, заданная в [RFC6066], которая интерпретируется в соответствии с [RFC6960].

> Примечание. Расширение status_request_v2 [RFC6961] отменено. Серверам TLS 1.3 **недопустимо** действовать в зависимости от его наличия или информации в нем при обработке ClientHello. В частности, **недопустимо** передавать расширение status_request_v2 в сообщениях EncryptedExtensions, CertificateRequest, Certificate. Сервер TLS 1.3 **должен** быть способен обрабатывать сообщения ClientHello с этим расширением, поскольку его **могут** передавать клиенты, желающие применять более ранние версии протокола.

Сервер **может** запросить у клиента представление отклика OCSP с сертификатом путём передачи пустого расширения status_request в сообщении CertificateRequest. Если клиент решает передать отклик OCSP, телом расширения status_request в нем **должна** быть структура CertificateStatus, определённая в [RFC6066].

[RFC6962] определяет для сервера механизм передачи временной метки SCT[1] как расширения в ServerHello для TLS 1.2 и более ранних версий. В TLS 1.3 серверная информация SCT передаётся в расширении CertificateEntry.

4.4.2.2. Выбор сертификата сервера

Приведённые ниже правила применяются для сертификатов, переданных сервером.

- Сертификат **должен** иметь тип X.509v3 [RFC5280], если явно не указано иное (например, [RFC7250]).
- Открытый ключ (и связанные с ним ограничения) сертификата конечного объекта для сервера **должен** быть совместим с выбранным алгоритмом аутентификации из клиентского расширения signature_algorithms (в настоящее время RSA, ECDSA или EdDSA).
- Сертификат **должен** разрешать использование ключа для подписи (т. е. бит digitalSignature **должен** быть установлен, если присутствует расширение Key Usage) со схемой подписи, указанной в расширениях signature_algorithms/signature_algorithms_cert (4.2.3. Алгоритмы подписи).
- Расширения server_name [RFC6066] и certificate_authorities служат для руководства выбором сертификата. Поскольку сервер **может** требовать наличия расширения server_name, клиентам **следует** передавать это расширения, когда оно применимо.

Все подписи, представленные сервером, **должны** быть выполнены с использованием алгоритма, анонсированного клиентом, если есть возможность обеспечить такую цепочку (4.2.3. Алгоритмы подписи). Самоподписанные сертификаты или сертификаты, для которых ожидается привязка доверия, не проверяются как часть цепочки и поэтому **могут** быть подписаны с любым алгоритмом.

Если сервер не может создать цепочку, подписанную с применением лишь указанных поддерживаемых алгоритмов, ему **следует** продолжать согласование, передавая клиенту цепочку сертификатов по своему выбору, включающую алгоритмы, для которых неизвестна поддержка клиентом. В этой резервной цепочке обычно **не следует** применять устаревший алгоритм SHA-1, но это **можно** делать, если клиентский анонс разрешает, а иные варианты **недопустимы**.

Если клиент не может построить подходящую цепочку с использованием представленных сертификатов и решает прервать согласование, он **должен** прервать его с помощью подходящего сигнала, связанного с сертификатом (по умолчанию unsupported_certificate, параграф 6.2. Сигналы ошибок).

Если у сервера есть несколько сертификатов, он выбирает 1 из них на основе приведённых выше критериев (в дополнение к таким критериям, как конечная точка транспортного уровня, локальные настройки и предпочтения).

4.4.2.3. Выбор сертификата клиента

Приведённые ниже правила применяются для сертификатов, переданных сервером.

- Сертификат **должен** иметь тип X.509v3 [RFC5280], если явно не указано иное (например, [RFC7250]).
- Если расширение certificate_authorities присутствует в CertificateRequest, хотя бы одному из сертификатов в цепочке сертификации **следует** выпущенным одним из указанных в списке CA.

[1]Signed Certificate Timestamp - временная метка подписанного сертификата.

- Сертификат **должен** быть подписан с использованием подходящего алгоритма, как указано в параграфе 4.3.2. Запрос сертификата. Это смягчает требования к алгоритмам подписи сертификатов прежних версий TLS.
- Если сообщение CertificateRequest содержит непустое расширение oid_filters, сертификат конечного объекта **должен** соответствовать OID, которые распознаются клиентом, как описано в параграфе 4.2.5.

4.4.2.4. Приём сообщения Certificate

В общем случае детализация процедур проверки пригодности сертификатов выходит за рамки TLS (см. [RFC5280]). В этом параграфе представлены относящиеся к TLS требования.

Если сервер передаёт пустое сообщение Certificate, клиент **должен** прервать согласование с сигналом decode_error.

Если клиент не передаёт никаких сертификатов (пустое сообщение Certificate), сервер **может** по своему усмотрению продолжить согласование без аутентификации клиента или прервать его с сигналом certificate_required. Если какая-то часть цепочки сертификатов неприемлема (например, не подписана известным, доверенным CA), сервер **может** по своему усмотрению продолжить (считая клиента непроверенным) или прервать согласование.

Любая конечная точка, получившая какой-либо сертификат, который требуется проверить с использованием любого алгоритма, применяющего хэш MD5, **должна** прервать согласование с сигналом bad_certificate. Алгоритм SHA-1 устарел и конечной точке, получившей сертификат, который нужно проверить с использованием алгоритма подписи, применяющего SHA-1, **рекомендуется** прервать согласование с сигналом bad_certificate. Проще говоря, это означает, что конечная точка может воспринимать эти алгоритмы для самоподписанных сертификатов и сертификатов с привязкой доверия.

Всем конечным точкам **рекомендуется** перейти на SHA-256 или лучший алгоритм для обеспечения совместимости с реализациями, которые постепенно отказываются от поддержки SHA-1.

Отметим, что сертификат, содержащий ключ для одного алгоритма подписи, **может** быть подписан с использованием другого алгоритма (например, ключ RSA, подписанный с ключом ECDSA).

4.4.3. Сообщение CertificateVerify

Это сообщение используется для явного подтверждения владения конечной точкой секретным ключом, соответствующим сертификату. Сообщение CertificateVerify также обеспечивает целостность согласования до этого момента. Сервер **должен** передавать это сообщение при аутентификации по сертификатам (сообщение Certificate не пусто). При передаче сообщения оно **должно** следовать сразу после Certificate и непосредственно перед Finished.

Структура сообщения показана ниже.

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

Поле algorithm задаёт применяемый алгоритм подписи (4.2.3. Алгоритмы подписи). Подписью служит цифровая подпись, созданная с использованием этого алгоритма. Содержимое, покрываемое подписью, является выводом хэш-функции, как описано в параграфе 4.4.1. Transcript-Hash, а именно Transcript-Hash(Handshake Context, Certificate)

Цифровая подпись затем рассчитывается для конкатенации строки октетов 32 (0x20), повторяющихся 64 раза, строки контекста, одного байта 0, служащего разделителем и подписываемого содержимого.

Эта структура предназначена для предотвращения атак на прежние версии TLS, в которых формат ServerKeyExchange позволял атакующему получить подпись с выбранным 32-байтовым префиксом (ClientHello.random). Начальное 64-байтовое заполнение очищает этот префикс вместе с контролируемым сервером значением ServerHello.random.

Строкой контекста для подписи сервера является "TLS 1.3, server CertificateVerify", а для клиента - "TLS 1.3, client CertificateVerify". Она служит разделителем между подписями, сделанными в разных контекстах, что защищает от возможных кросс-протокольных атак.

Например, если хэш стенограммы представлял собой 32 байта 01 (этот размер будет иметь смысл для SHA-256), содержимое, покрываемое цифровой подписью для серверного CertificateVerify будет иметь вид

```
2020202020202020202020202020202020202020202020202020202020202020
2020202020202020202020202020202020202020202020202020202020202020
544c5320312e332c20736572766572204365727469666963617465726966
79
00
0101010101010101010101010101010101010101010101010101010101010101
```

На стороне отправителя процесс расчёта поля signature в сообщении CertificateVerify принимает на входе содержимое, охватываемое цифровой подписью, и секретный ключ подписи, соответствующий сертификату, переданному в предыдущем сообщении.

Если сервером передано сообщение CertificateVerify, алгоритмом подписи **должен** быть один из предложенных в клиентском расширении signature_algorithms, если никакая действительная цепочка сертификации не может быть создана без неподдерживаемых алгоритмов (4.2.3. Алгоритмы подписи).

При отправке сообщения клиентом алгоритмом подписи **должен** быть один из присутствующих в поле supported_signature_algorithms расширения signature_algorithms в сообщении CertificateRequest.

В дополнение к этому алгоритм подписи **должен** быть совместим с ключом в сертификате конечного объекта передающей стороны. Подписи RSA **должны** использовать алгоритм RSASSA-PSS, независимо от присутствия алгоритмов RSASSA-PKCS1-v1_5 в signature_algorithms. Алгоритм SHA-1 **недопустимо** применять в каких-либо подписях сообщений CertificateVerify.

Все алгоритмы SHA-1 в этой спецификации определены исключительно для применения в унаследованных сертификатах и не пригодны для подписей CertificateVerify.

Получатель сообщения CertificateVerify **должен** проверить поле подписи, используя на входе содержимое, охватываемое подписью, открытый ключ из сертификата конечного объекта в сообщении Certificate и цифровую подпись, полученную в поле signature сообщения CertificateVerify.

Если проверка дала отрицательный результат, получатель **должен** прервать согласование с сигналом decrypt_error.

4.4.4. Сообщение Finished

Сообщение Finished завершает обмен в блоке проверки подлинности (Authentication Block). Оно важно для аутентификации согласования и рассчитанных ключей.

Получатель сообщения Finished **должен** проверить корректность его содержимого и при обнаружении несоответствия прервать согласование с сигналом decrypt_error.

После того, как сторона передала сообщение Finished, а также получила и проверила Finished от партнёра, она может начинать передачу и приём данных приложения через организованное соединение. Имеется две настройки, при которых разрешена передача данных до получения от партнёра сообщения Finished.

1. Клиент передал данные 0-RTT, как описано в параграфе 4.2.10. Индикация ранних данных.
2. Сервер **может** передавать данные после своей первой отправки, но по причине незавершённости согласования у него не будет уверенности в отождествлении и живучести партнёра (сообщение ClientHello могло быть воспроизведено).

Ключ, применяемый для создания сообщения Finished, рассчитывается из базового ключа (Base Key), определённого в параграфе 4.4. Аутентификационные сообщения, с использованием HKDF (7.1. Планирование ключей). В частности,

```
finished_key = HKDF-Expand-Label(BaseKey, "finished", "", Hash.length)
```

Структура сообщения показана ниже.

```
struct {
    opaque verify_data[Hash.length];
} Finished;
```

Значение verify_data определяется выражением

```
verify_data =
    HMAC(finished_key,
         Transcript-Hash(Handshake Context,
                         Certificate[1], CertificateVerify[1]))
```

HMAC [RFC2104] использует алгоритм Hash для согласования. Как отмечено выше, входные данные HMAC в общем случае могут быть получены с помощью текущего хэша, т. е. хэша согласования в этой точке.

В предыдущих версиях TLS размер verify_data всегда составлял 12 октетов, но в TLS 1.3 это размер вывода HMAC для функции Hash, используемой при согласовании.

> Примечание. Сигналы и другие типы записей, не относящиеся к согласованию, являются сообщениями, не связанными с согласованием, и не включаются в хэш.

Все записи, следующие за сообщением Finished, **должны** шифроваться с использованием соответствующего ключа трафика приложений, как описано в параграфе 7.2. Обновление секретов трафика. В частности, это включает любые сигналы, передаваемые сервером в ответ на клиентские сообщения Certificate и CertificateVerify.

4.5. Сообщение EndOfEarlyData

```
struct {} EndOfEarlyData;
```

Если сервер передал расширение early_data в EncryptedExtensions, клиент должен передать сообщение EndOfEarlyData после приёма серверного сообщения Finished. Если сервер не передал расширения early_data в EncryptedExtensions, клиенту **недопустимо** передавать сообщение EndOfEarlyData. Это сообщение указывает, что все сообщения 0-RTT application_data (если они были) переданы и последующие данные будут защищены с согласованными ключами трафика. Серверу **недопустимо** передавать это сообщение, а получивший такое сообщение клиент **должен** прервать согласование с сигналом unexpected_message. Сообщение шифруется с ключами, выведенными из client_early_traffic_secret.

4.6. Сообщения после согласования

TLS позволяет передавать после основного согласования и другие сообщения, использующие тип содержимого handshake и шифруемые подходящим ключом трафика приложений.

4.6.1. Сообщение NewSessionTicket

В любой момент после получения сервером клиентского сообщения Finished он **может** передать сообщение NewSessionTicket, которое создаёт уникальную ассоциацию между значением квитанции и секретным PSK, выведенным из первичного секрета (7. Криптографические расчёты).

Клиент **может** применять этот PSK в будущих согласованиях, включая квитанцию в расширение pre_shared_key своего ClientHello (4.2.11. Расширение PSK). Серверы могут передавать несколько квитанций в одном соединении сразу одну за другой, либо после конкретных событий (C.4. Предотвращение отслеживания клиентов). Например, сервер может передать новую квитанцию после аутентификации post-handshake, чтобы инкапсулировать дополнительное состояние аутентификации клиента. Множество квитанций полезно для клиентов по разным причинам, включая:

- создание множества параллельных соединений HTTP;
- организацию одновременных соединений через разные интерфейсы и семейства адресов, например, с помощью Happy Eyeballs [RFC8305] или других технологий.

[1]Только при наличии этого сообщения.

Любая квитанция **должна** возобновляться лишь с шифром, имеющим такой же набор алгоритмов хэширования KDF, какой применялся для организации исходного соединения.

Клиенты **должны** возобновлять сессию лишь в том случае, когда новое значение SNI действительно для сертификата сервера, представленного в исходной сессии, а также **следует** возобновлять сессию, если значение SNI соответствует одному из использованных в исходной сессии. Последнее является оптимизацией производительности и обычно нет причин предполагать, что разные серверы, охватываемые одним сертификатом, смогут принимать квитанции друг друга, поэтому попытка восстановления в таком случае приведёт к потере одноразовой квитанции. Если такое указание предоставляется (извне или иным путём), клиенты **могут** восстанавливать сессию с другим значением SNI.

Если при восстановлении значение SNI сообщается вызывающему приложению, реализации **должны** использовать значение, переданное в восстанавливающем ClientHello, а не значение из предыдущей сессии. Отметим, что если реализация сервера отвергает отождествления PSK с разными значениями SNI, эти два значения всегда совпадают.

Примечание. Хотя первичный секрет для восстановления зависит от второй отправки клиента, сервер, который не запрашивал аутентификацию клиента, **может** рассчитать остальную часть стенограммы независимо и передать NewSessionTicket сразу же после отправки Finished, не ожидая клиентского сообщения Finished. Это может быть приемлемо для случаев, когда предполагается создание клиентом множества параллельных соединений TLS и обеспечит снижение издержек при согласовании восстановления.

```
struct {
    uint32 ticket_lifetime;
    uint32 ticket_age_add;
    opaque ticket_nonce<0..255>;
    opaque ticket<1..2^16-1>;
    Extension extensions<0..2^16-2>;
} NewSessionTicket;
```

ticket_lifetime

Указывает время жизни (в секундах) с момента выпуска квитанции 32-битовым целым числом без знака с сетевым порядком байтов. Серверам **недопустимо** использовать значение больше 604800 секунд (7 дней). Нулевое значение указывает, что квитанцию следует отбросить сразу же. Клиентам **недопустимо** кэшировать квитанции больше, чем на 7 дней, независимо от значения ticket_lifetime, и они **могут** удалять квитанции раньше в соответствии с локальной политикой. Сервер **может** считать квитанцию действительной в течение времени, которое меньше указанного в ticket_lifetime.

ticket_age_add

Созданное защищённым способом случайное 32-битовое значение, которое служит для сокрытия возраста квитанции, указанного клиентом в расширении pre_shared_key. Возраст квитанции на клиентской стороне складывается с этим значением по модулю 2^{32} для получения значения, которое передаётся клиенту. Сервер **должен** генерировать новое значение для каждой передаваемой квитанции.

ticket_nonce

Значение для квитанции, уникальное в каждой квитанции данного соединения.

ticket

Значение квитанции, служащее отождествлением PSK. Само поле ticket является неинтерпретируемой меткой. Это **может** быть ключ поиска в базе данных или самозашифрованное и самоаутентифицированное значение.

extensions

Набор расширений для квитанции, формат расширения описан в параграфе 4.2. Клиент **должен** игнорировать непонятные расширения.

Для NewSessionTicket в настоящее время определено единственное расширение early_data, показывающее, что квитанцию можно применять для передачи данных 0-RTT (4.2.10. Индикация ранних данных). Оно содержит описанное ниже значение.

max_early_data_size

Максимальное число байтов данных 0-RTT, которые клиенту разрешено передать с применением этой квитанции. Учитываются только данные приложений (т. е. открытые данные без заполнения и внутреннего байта типа). Серверу, получившему больше max_early_data_size байтов данных 0-RTT, **следует** прервать соединение с сигналом unexpected_message. Отметим, что серверы, отвергающие ранние данные по причине отсутствия криптографического материала, не смогут отличить заполнение от содержимого, поэтому клиентам **не следует** полагаться на возможность передачи большого объёма заполнения в записях с ранними данными.

Связанный с квитанцией PSK рассчитывается как

```
HKDF-Expand-Label(resumption_master_secret, "resumption", ticket_nonce, Hash.length)
```

Поскольку значение ticket_nonce отличается для каждого сообщения NewSessionTicket, каждая квитанция будет иметь свой PSK.

Отметим, что в принципе можно продолжать выпуск новых квитанций, которые бесконечно расширяют срок действия ключевого материала, исходно выведенного из начального согласования без PSK (который, скорей всего, связан с сертификатом партнёра). Реализациям **рекомендуется** задавать пределы общего срока действия ключевого материала и этим пределам следует учитывать срок действия сертификата партнёра, вероятность отзыва сертификата и время, прошедшее с момента получения партнерской подписи CertificateVerify.

4.6.2. Аутентификация после согласования

Если клиент передал расширение post_handshake_auth (параграф 4.2.6), сервер **может** запросить аутентификацию клиента в любой момент после завершения согласования путём передачи сообщения CertificateRequest. Клиент **должен** ответить подходящими сообщениями Authentication (параграф 4.4). Если клиент выбрал аутентификацию, он **должен** передать сообщения Certificate, CertificateVerify и Finished. Если аутентификация отвергнута, клиент **должен** передать сообщение Certificate без сертификатов, а затем - сообщение Finished. Все клиентские сообщения для данного отклика **должны** передаваться в линию последовательно, без промежуточных сообщений других типов.

Клиент, который получил сообщение CertificateRequest, но не передавал расширения post_handshake_auth, **должен** передать критический сигнал unexpected_message.

Примечание. Поскольку аутентификация клиента может включать обращение к пользователю, серверы **должны** быть готовы к некоторой задержке, включающей получение произвольного числа других сообщений в интервале между отправкой CertificateRequest и получением отклика. Кроме того, клиенты, получившие множество CertificateRequest за короткое время, **могут** отвечать на них в порядке, отличающемся от порядка приёма (значение certificate_request_context позволяет серверу устранять неоднозначность откликов).

4.6.3. Обновление ключа и вектора инициализации

Согласующее сообщение KeyUpdate служит для индикации обновления передающей стороной своих криптографических ключей передачи. Это сообщение может передаваться любым из партнёров после отправки им сообщения Finished. Реализации, получившие KeyUpdate до сообщения Finished, **должны** прерывать соединение с сигналом unexpected_message. После передачи сообщения KeyUpdate отправителю **нужно** передавать весь трафик со следующим поколением ключей, рассчитанных в соответствии с параграфом 7.2. Обновление секретов трафика. При получении KeyUpdate принявшая сторона **должна** обновить свои приёмные ключи.

```
enum {
    update_not_requested(0), update_requested(1), (255)
} KeyUpdateRequest;

struct {
    KeyUpdateRequest request_update;
} KeyUpdate;
```

request_update

Показывает, следует ли получателю KeyUpdate отвечать своим сообщением KeyUpdate. При получении другого значения реализация **должна** разорвать соединение с сигналом illegal_parameter.

Если поле request_update имеет значение update_requested, получатель **должен** передать своё сообщение KeyUpdate с request_update = update_not_requested до отправки следующей записи с данными приложения. Этот механизм позволяет любой из сторон форсировать полное обновление соединения, но реализация, получившая несколько сообщений KeyUpdate во время своего молчания будет отвечать лишь одним обновлением. Отметим, что реализации могут получить произвольное число сообщений между отправкой KeyUpdate с request_update set = update_requested и получением партнерского KeyUpdate, поскольку эти сообщения уже были в сети. Однако ключи приёма и передачи выводятся из независимых секретов трафика, поэтому сохранение приёмного секрета не угрожает защите данных, отправленных до смены ключей отправителем.

Если реализации независимо отправляют свои KeyUpdate с request_update = update_requested и эти сообщения «пересекаются» в пути, каждая из сторон передаст отклик и в результате произойдёт двухкратная смена ключей.

Отправитель и получатель **должны** шифровать свои сообщения KeyUpdate со старыми ключами. Кроме того, обе стороны **должны** обеспечить приём KeyUpdate со старым ключом до восприятия каких-либо сообщений, зашифрованных новым ключом. Невыполнение этого требования может привести к атакам с отсечкой сообщений.

5. Протокол Record

Протокол TLS Record принимает сообщения для передачи, фрагментирует данные в управляемые блоки, защищает записи и передаёт результат. Принятые данные проверяются, расшифровываются, собираются из фрагментов и затем доставляются клиентам вышележащих уровней.

Записи TLS типизованы, что позволяет мультиплексировать множество протоколов вышележащих уровней через один уровень записи. Этот документ задаёт 4 типа - согласование (handshake), данные приложений (application_data), сигналы (alert) и смену шифра (change_cipher_spec). Запись change_cipher_spec служит для обеспечения совместимости (D.4. Режим совместимости с промежуточными устройствами).

Реализация может получить нешифрованную запись типа change_cipher_spec, состоящую из одного байта 0x01, в любой момент после передачи или приёма первого сообщения ClientHello и до получения от партнёра сообщения Finished и **должна** просто отбросить ее без обработки. Отметим, что эта запись может появиться во время согласования, где реализация ожидает защищённые записи, поэтому необходимо обнаружить это до попытки снять защиту записи. Реализация, получившая любое другое значение change_cipher_spec или защищённую запись change_cipher_spec, **должна** прервать согласование с сигналом unexpected_message. Если реализация обнаруживает запись change_cipher_spec, принятую до первого сообщения ClientHello или после сообщения Finished от партнёра, она **должна** трактовать её как запись неожиданного типа (хотя серверы без учёта состояния могут не отличить этот случай от дозволенных).

Реализациям **недопустимо** передавать записи не заданных в этом документе типов, пока это не согласовано тем или иным расширением. Если реализация TLS получает запись неожиданного типа, она **должна** прервать соединение с сигналом unexpected_message. Значения новых типов записей выделяются IANA в реестре TLS ContentType, как описано в разделе 11.

5.1. Уровень Record

Уровень записи фрагментирует блоки информации в записи TLSPlaintext, передающие данные блоками размером до 2^{14} байтов. Границы сообщений обрабатываются по-разному в зависимости от ContentType. Все будущие типы содержимого **должны** задавать подходящие правила. Отметим, что эти правила жёстче принятых в TLS 1.2.

Согласующие сообщения могут быть объединены в одну запись TLSPlaintext или фрагментированы в несколько записей при выполнении приведённых ниже условий.

- Согласующие сообщения **недопустимо** чередовать с записями других типов, т. е. при разделении согласующего сообщения на записи **недопустимо** помещать между ними другие записи.
- Для согласующих сообщений **недопустим** переход через смену ключей. Реализации должны проверять, что все сообщения, непосредственно предшествующие смене ключа, относятся к одной записи. В противном случае соединение **должно** разрываться с сигналом unexpected_message. Поскольку сообщения ClientHello,

EndOfEarlyData, ServerHello, Finished и KeyUpdate могут непосредственно предшествовать смене ключей, реализации **должны** передавать эти сообщения с выравниванием по границе записи.

Реализациям **недопустимо** передавать фрагменты типа Handshake размера 0, даже если они содержат заполнение.

Сигнальные сообщения (6. Протокол Alert) **недопустимо** фрагментировать между записями и несколько сигнальных сообщений **недопустимо** собирать в одну запись TLSPlaintext. Иными словами запись с типом Alert **должна** содержать в точности одно сообщение.

Сообщения Application Data содержат данные, которые TLS не анализирует. Эти сообщения всегда защищены. Фрагменты Application Data с нулевым размером **можно** передавать, поскольку они полезны в качестве противодействия анализу трафика. Фрагменты Application Data **можно** разделять между разными записями или собирать в одну запись.

```
enum {
    invalid(0),
    change_cipher_spec(20),
    alert(21),
    handshake(22),
    application_data(23),
    (255)
} ContentType;

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

type

Протокол вышележащего уровня, используемый для обработки вложенного фрагмента.

legacy_record_version

Должно иметь значение 0x0303 для всех записей, генерируемых реализацией TLS 1.3, за исключением начального ClientHello (т. е. не отклика на HelloRetryRequest), где это поле **может** иметь также значение 0x0301 для обеспечения совместимости. Это поле устарело и **должно** игнорироваться при обработке. Предыдущие версии TLS будут применять другие значения этого поля при некоторых обстоятельствах.

length

Размер (в байтах) следующего TLSPlaintext.fragment. **Недопустим** размер, превышающий 2^{14} байтов. Конечная точка, получившая запись избыточного размера, **должна** прервать соединение с сигналом record_overflow.

fragment

Данные, которые будут передаваться. Это значение трактуется как независимый блок данных, который будет обрабатываться вышележащим протоколом, указанным в поле type.

Этот документ описывает TLS 1.3, где используется версия 0x0304. Номер версии обусловлен исторически, поскольку ранее применялось значение 0x0301 для TLS 1.0 и 0x0300 для SSL 3.0. Для максимальной совместимости с прежними версиями в записи с начальным ClientHello **следует** указывать версию 0x0301 (соответствует TLS 1.0), а в записи со вторым ClientHello или ServerHello **должен** быть номер версии 0x0303 (соответствует TLS 1.2). При согласовании более ранних версий TLS конечные точки следуют процедуре и требованиям, представленным в приложении D.

Когда защита записей ещё не действует, структуры TLSPlaintext передаются в линию напрямую. После включения защиты записи TLSPlaintext передаются в соответствии с приведённым ниже описанием. Отметим, что данные приложений **недопустимо** передавать в линию без защиты (см. 2. Обзор протокола).

5.2. Защита данных Record

Функции защиты записей преобразуют структуру TLSPlaintext в TLSCiphertext, функции снятия защиты выполняют обратное преобразование. В TLS 1.3, в отличие от прежних версий, все шифры моделируются как AEAD [RFC5116]. Функции AEAD обеспечивают унифицированную операцию шифрования и аутентификации, которая преобразует открытые данные в шифрованные и аутентифицированные, а также обратно. Каждая шифрованная запись содержит открытый заголовок, за которым следует шифрованное тело, содержащее тип и необязательное заполнение.

```
struct {
    opaque content[TLSPlaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;

struct {
    ContentType opaque_type = application_data; /* 23 */
    ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

content

Значение TLSPlaintext.fragment, содержащее байтовое представление согласующего или сигнального сообщения или необработанные байты данных приложения для передачи.

type

Значение TLSPlaintext.type, указывающее тип содержимого записи.

zeros

Последовательность нулевых байтов произвольного размера, которая может включаться в открытые данные после поля type. Это позволяет отправителю дополнить любую запись TLS до выбранного размера в пределах заданных ограничений (5.4. Дополнение записей).

opaque_type

Внешнее поле opaque_type в записи TLSCiphertext всегда имеет значение 23 (application_data) для совместимости с промежуточными устройствами, настроенными на разбор предыдущих версий TLS. Реальный тип содержимого записи определяется из поля TLSInnerPlaintext.type после расшифровки.

legacy_record_version

Поле legacy_record_version всегда имеет значение 0x0303. Структуры TLS 1.3 TLSCiphertext не генерируются до завершения согласования TLS 1.3, поэтому не возникает проблем совместимости при получении других значений. Отметим, что протокол согласования, включая сообщения ClientHello и ServerHello, аутентифицирует версию протокола, поэтому данное значение является избыточным.

length

Размер (в байтах) следующего поля TLSCiphertext.encrypted_record, который определяется суммой размеров содержимого и заполнения, а также размера внутреннего типа содержимого (1) и расширений, добавленных алгоритмом AEAD. **Недопустимы** значения length, превышающие 214 + 256 байтов. Конечная точка, получившая запись избыточного размера, **должна** прервать соединение с сигналом record_overflow.

encrypted_record

Зашифрованная с помощью AEAD форма последовательного представления структуры TLSInnerPlaintext.

Алгоритмы AEAD принимают на входе один ключ, nonce, открытые данные (plaintext) и дополнительные данные для включения в проверку подлинности, как описано в параграфе 2.1 [RFC5116]. Ключом служит client_write_key или server_write_key, nonce выводится из порядкового номера и client_write_iv или server_write_iv (5.3. Nonce для отдельной записи), а дополнительными данными - заголовок записи

```
additional_data = TLSCiphertext.opaque_type ||
                  TLSCiphertext.legacy_record_version ||
                  TLSCiphertext.length
```

Открытые данные на входе алгоритма AEAD кодируются в структуру TLSInnerPlaintext, создание ключей трафика описано в параграфе 7.3. Расчёт ключей трафика.

Выводом AEAD являются шифроданные, возвращаемые операцией шифрования AEAD. Размер открытых данных больше соответствующего значения TLSPlaintext.length в результате включения TLSInnerPlaintext.type и предоставленного отправителем заполнения. Размер вывода AEAD в общем случае превышает размер открытых данных, а величина этого превышения зависит от алгоритма AEAD. Шифры могут включать заполнение, поэтому превышение может меняться в зависимости от размера открытых данных.

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

В поле encrypted_record структуры TLSCiphertext указывается значение AEADEncrypted.

Для расшифровки и проверки алгоритм принимает на входе ключ, nonce, дополнительные данные и значение AEADEncrypted. Выходом будут расшифрованные данные или ошибка, указывающая отказ при расшифровке. Отдельной проверки целостности не выполняется.

```
расшифровка encrypted_record =
    AEAD-Decrypt(peer_write_key, nonce, additional_data, AEADEncrypted)
```

При отказе в расшифровке получатель **должен** прервать соединение с сигналом bad_record_mac.

Алгоритмам AEAD, используемым в TLS 1.3, **недопустимо** при шифровании увеличивать размер более, чем на 255 октетов. Конечная точка, получившая от партнёра запись с TLSCiphertext.length больше (2^{14} + 256) октетов, **должна** прервать соединение с сигналом record_overflow. Это ограничение выведено из максимального размера TLSInnerPlaintext (2^{14} октетов) + 1 октет ContentType + максимальное расширение при шифровании AEAD (255 октетов).

5.3. Nonce для отдельной записи

Для чтения и записи поддерживаются отдельные 64-битовые порядковые номера. Соответствующий порядковый номер увеличивается на 1 после каждой операции чтения или записи. Отсчёт порядковых номеров начинается с 0 при организации соединения и каждой смене ключа. Первая запись, переданная с новым ключом трафика **должна** иметь порядковый номер 0.

Поскольку размер порядкового номера составляет 64 бита, они не должны переходить через максимум. При достижении максимума реализация TLS **должна** сменить ключ (4.6.3. Обновление ключа и вектора инициализации) или разорвать соединение.

Каждый алгоритм AEAD будет задавать диапазон возможных размеров nonce для записи от N_MIN до N_MAX байтов [RFC5116]. Размер nonce для записи TLS (iv_length) устанавливается больше 8 и N_MIN байтов для алгоритмов AEAD (раздел 4 в [RFC5116]). Алгоритмы AEAD с N_MAX меньше 8 байтов **недопустимо** применять в TLS. Значение nonce для записи в конструкции AEAD формируется, как показано ниже.

1. 64-битовый порядковый номер кодируется с сетевым порядком байтов и дополняется слева нулями до размера iv_length.
2. Выполняется операция XOR для дополненного порядкового номера и статического значения client_write_iv или server_write_iv (в зависимости от роли).

Полученное в результате значение размером iv_length используется в качестве nonce для записи.

Примечание. Это отличается от TLS 1.2, где применялись частично явные значения nonce.

5.4. Дополнение записей

Все шифрованные записи TLS могут дополняться для увеличения размера TLSCiphertext. Это позволяет отправителю скрыть истинный размер трафика от наблюдателя.

При генерации TLSCiphertext реализации **могут** применять заполнение. Недополненная запись считается записью с заполнением нулевого размера. Заполнение является строкой байтов 0, добавленной после поля ContentType до шифрования. Реализации **должны** устанавливать нулевое значения байтов заполнения перед шифрованием.

Записи Application Data могут содержать пустое поле TLSInnerPlaintext.content, если отправитель этого хочет. Это позволяет генерировать правдоподобный трафик в контексте, где наличие или отсутствие активности может быть важно. Реализациям **недопустимо** передавать записи Handshake и Alert с пустым полем TLSInnerPlaintext.content и при получении такой записи приёмная сторона **должна** разорвать соединение с сигналом unexpected_message.

Передаваемое заполнение автоматически проверяется механизмом защиты записей. При успешной расшифровке TLSCiphertext.encrypted_record принимающая сторона сканирует поле от конца к началу, пока не дойдёт до отличного от 0 октета, задающего тип содержимого сообщения. Такая схема заполнения выбрана для того, чтобы можно было дополнить любую запись TLS произвольного размера (от 0 до максимума) без задания новых типов содержимого. Решение также предусматривает заполнение нулями, что позволяет быстро обнаруживать ошибки заполнения.

Реализации **должны** завершать сканирование на открытых данных, возвращённых после расшифровки AEAD. Если принимающая сторона не нашла отличных от 0 октетов в открытых данных, она **должна** разорвать соединение с сигналом unexpected_message.

Наличие заполнения не меняет ограничений на общий размер записи - полной записи TLSInnerPlaintext **недопустимо** быть больше (214 + 1) октетов. Если размер фрагмента снижен (например, с помощью расширения record_size_limit из [RFC8449]), это ограничение относится ко всем открытым данным, включая тип содержимого и заполнение.

Выбор политики заполнения, указывающей когда и сколько октетов можно добавлять, является сложным вопросом и выходит за рамки этого документа. Если протокол прикладного уровня над TLS использует своё заполнение, может оказаться предпочтительным дополнять записи TLS с данными приложения в рамках прикладного уровня. Заполнение шифрованных записей Handshake и Alert должно выполняться на уровне TLS. Будущие документы могут определять алгоритмы выбора или механизмы запроса политики заполнения с помощью расширений TLS или иными средствами.

5.5. Ограничения на использование ключей

Имеются криптографические ограничения на объем открытых данных, которые можно безопасно шифровать с данным набором ключей. В [AEAD-LIMITS] приведён анализ этих ограничений для случая, когда базовый примитив (AES или ChaCha20) не имеет слабых мест. Реализациям **следует** обновлять ключи в соответствии с параграфом 4.6.3. Обновление ключа и вектора инициализации до достижения этих пределов.

Для AES-GCM можно шифровать до $2^{24,5}$ полноразмерных записей (около 24 миллионов) в данном соединении, сохраняя запас по безопасности приблизительно 2^{-57} для аутентифицированного шифрования. Для ChaCha20/Poly1305 порядковые номера достигнут максимума раньше, чем наступит предел безопасного шифрования.

6. Протокол Alert

TLS поддерживает тип содержимого Alert для индикации ошибок и событий, связанных с разрывом соединений. Подобно другим сообщениям, сигналы шифруются в соответствии с текущим состоянием соединения.

Сигнальные сообщения передают описание события и унаследованное поле level, которое указывало уровень важности события в предыдущих версиях TLS. Сигналы делятся на два класса, один из которых связан с разрывом соединения, другой - с ошибками. В TLS 1.3 уровень важности указывается неявно типом сигнала и поле level можно безбоязненно игнорировать. Сигнал close_notify служит для индикации упорядоченного закрытия одного из направлений соединения. При получении такого сигнала реализации TLS **следует** сообщить приложению о завершении передачи данных.

Сигналы об ошибках указывают разрыв соединения (6.2. Сигналы ошибок). При получении такого сигнала реализации TLS **следует** передать ошибку приложению, а продолжение передачи или приёма данных через соединение **недопустимо**. Серверы и клиенты **должны** забыть секретные значения и ключи разорванного соединения за исключением PSK, связанных с квитанциями сессии, которые по возможности **следует** отбросить.

Все сигналы, указанные в параграфе 6.2, **должны** передаваться с AlertLevel=fatal и **должны** считаться сигналами об ошибках независимо от AlertLevel в сообщении. Неизвестные типы Alert **должны** считаться сигналами об ошибках.

> Примечание. TLS определяет 2 базовых сигнала (раздел 6) для использования при отказах в разборе сообщений. Партнёры, получившие сообщение, которое не удаётся разобрать в соответствии с синтаксисом (например, недопустимо длинное), **должны** прерывать соединение с сигналом decode_error. Партнёры, получившие синтаксически корректное, но семантически недействительное сообщение (например, DHE с p = 1 или некорректное перечисляемое значение), **должны** прерывать соединение с сигналом illegal_parameter.

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
    unexpected_message(10),
    bad_record_mac(20),
    record_overflow(22),
    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),
    inappropriate_fallback(86),
```

```
        user_canceled(90),
        missing_extension(109),
        unsupported_extension(110),
        unrecognized_name(112),
        bad_certificate_status_response(113),
        unknown_psk_identity(115),
        certificate_required(116),
        no_application_protocol(120),
        (255)
    } AlertDescription;

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

6.1. Сигналы о закрытии

Клиент и сервер должны иметь общую информацию о закрытии соединения во избежание атак с отсечкой (truncation).

close_notify

Этот сигнал уведомляет получателя о том, что отправитель не будет больше передавать сообщений в этом соединении. Все данные, полученные после этого сигнала, **должны** игнорироваться.

user_canceled

Этот сигнал информирует получателя о том, что отправитель отверг согласование по причинам, не связанным с протокольным отказом, и обычно имеет AlertLevel=warning. Если пользователь прервал операцию уже после согласования, лучше закрывать соединение с сигналом close_notify. После сигнала user_canceled следует передавать close_notify.

Любая из сторон **может** инициировать закрытие записи на своей стороне соединения путём отправки сигнала close_notify. После получения этого сигнала приёмная сторона **должна** игнорировать все получаемые данные. Если сигнал о закрытии на транспортном уровне получен до close_notify, получатель не может знать, что все отправленные данные были получены.

Каждая из сторон **должна** передать close_notify до закрытия записи на своей стороне соединения, если она уже не передала какой-либо сигнал об ошибке. Это не оказывает никакого влияния на чтение из соединения на этой стороне. Отметим отличие от предшествующих версий TLS, где от реализаций требовалось реагировать на close_notify отбрасыванием ожидающих записей и незамедлительной передачей ответного close_notify, что могло приводить к отсечке чтения. Обеим сторонам не нужно ждать получения close_notify перед закрытием чтения на своей стороне, хотя это может приводить к отсечке данных.

Если использующий TLS прикладной протокол предполагает, что данные могут быть переданы через базовый транспорт после разрыва соединения TLS, реализация TLS **должна** получить сигнал close_notify перед индикацией завершения данных прикладному уровню. Ни одну часть этого стандарта не следует применять для задания способа управления транспортировкой данных профилем TLS , включая ситуации когда соединение открыто или закрыто.

> **Примечание**. Предполагается, что при закрытии записи в соединении ожидающие данные гарантированно доставляются до удаления транспорта.

6.2. Сигналы ошибок

Обработка ошибок в TLS очень проста - обнаружившая ошибку сторона передаёт сообщение партнёру. При передаче или приёме сигнала о критической ошибке обе стороны **должны** сразу же разорвать соединение.

Всякий раз, когда реализация сталкивается с критической ошибкой, ей **следует** передать подходящий сигнал и она **должна** закрыть соединение без передачи или приёма дополнительных данных. В оставшейся части документа использование слов «разрыв соединения» и «прерывание согласования» без уточнения причины означает, что реализации **следует** передать сигнал, указанный приведёнными ниже описаниями. Фразы «разорвать соединение с сигналом X» и «прервать согласование с сигналом X» означают, что реализация **должна** передать сигнал X, если она передаёт какой-либо сигнал. Все сигналы, определённые далее в этом параграфе, а также все неизвестные сигналы считаются критическими в TLS 1.3 (6. Протокол Alert). Реализациям **следует** обеспечивать способ записи в системный журнал информации об отправке и получении сигналов.

Ниже перечислены определяемые спецификацией сигналы об ошибках.

unexpected_message

Получено непригодное сообщение (ошибочное согласующее сообщение, преждевременные пользовательские данные и т. п.). Такие сигналы не должны появляться во взаимодействии между корректными реализациями.

bad_record_mac

Этот сигнал передаётся при получении записи, с которой не удалось снять защиту. Поскольку в алгоритмах AEAD сочетается защита и проверка, а также для предотвращения побочных каналов, это сообщение передаётся при всех отказах снятия защиты. Такие сигналы не должны появляться во взаимодействии между корректными реализациями за исключением случаев повреждения данных в сети.

record_overflow

Полученная запись TLSCiphertext имеет размер больше (2^{14} + 256) байтов или расшифрованная запись TLSPlaintext по размеру превышает 2^{14} (или иное согласованное значение). Такие сигналы не должны появляться во взаимодействии между корректными реализациями за исключением случаев повреждения данных в сети.

handshake_failure

Этот сигнал указывает, что его отправитель не способен согласовать приемлемый набор параметров защиты с учётом доступных опций.

bad_certificate

Сертификат повреждён, содержит подписи, которые не удалось проверить и т. п.

unsupported_certificate

Неподдерживаемый тип сертификата.

certificate_revoked
Сертификат был отозван подписавшей его стороной.

certificate_expired
Срок действия сертификата истёк или сертификат недействителен по иной причине.

certificate_unknown
Возникли те или иные (не указанные) проблемы при обработке сертификата, не позволившие принять его.

illegal_parameter
Поле в согласовании было некорректно или несовместимо с другими полями. Этот сигнал применяется для случаев, когда формальный синтаксис соблюдается, но что-то иное некорректно.

unknown_ca
Получена пригодная цепочка или часть цепочки сертификатов, но сертификат не принят, поскольку сертификат CA не удалось найти или он не соответствует известным привязкам доверия.

access_denied
Получен пригодный сертификат или PSK, но при контроле доступа отправитель решил не выполнять согласование.

decode_error
Сообщение не может быть декодировано, поскольку некоторые поля выходят за пределы диапазонов или размер сообщения некорректен. Этот сигнал применяется для случаев, когда сообщение не соответствует формальному синтаксису протокола. Такие сигналы не должны появляться во взаимодействии между корректными реализациями за исключением случаев повреждения сообщений в сети.

decrypt_error
Отказ криптографической операции при согласовании (не на уровне Record), включая невозможность проверить синтаксис, сообщение Finished или привязку PSK.

protocol_version
Предлагаемая партнёром версия протокола распознана, но не поддерживается (см. Приложение D).

insufficient_security
Возвращается вместо handshake_failure при отказе согласования в результате того, что сервер требует более защищённых параметров, нежели поддерживает клиент.

internal_error
Внутренняя ошибка, не связанная с партнёром или корректностью протокола (например, ошибка при выделении памяти), которая не позволяет продолжить работу.

inappropriate_fallback
Передаётся сервером в ответ на некорректную попытку повторного соединения от клиента (см. [RFC7507]).

missing_extension
Передаётся конечной точкой, получившей согласующее соединение, которое не содержит расширения, обязательного для передачи в предложенной версии TLS, или других согласуемых параметров.

unsupported_extension
Передаётся конечной точкой, получившей согласующее сообщение с запрещённым для этого типа расширением, ServerHello или Certificate с расширением, не предложенным в соответствующем ClientHello или CertificateRequest.

unrecognized_name
Передаётся сервером, если он не идентифицируется именем, указанным в расширении server_name ([RFC6066]).

bad_certificate_status_response
Передаётся клиентом при получении от сервера некорректного или неприемлемого отклика OCSP в расширении status_request ([RFC6066]).

unknown_psk_identity
Передаётся сервером, когда желательна организация ключа PSK, но клиент не предоставил приемлемого отождествления PSK. Передача этого сигнала **необязательна** и сервер **может** передать decrypt_error для указания непригодного отождествления PSK.

certificate_required
Передаётся сервером, когда сертификат клиента желателен, но не предоставлен клиентом.

no_application_protocol
Передаётся сервером, когда клиентское расширение application_layer_protocol_negotiation анонсирует лишь неподдерживаемые сервером протоколы (см. [RFC7301]).

Новые значения Alert назначаются IANA в соответствии с разделом 11.

7. Криптографические расчёты

Согласование TLS создаёт один или множество входных секретов, которые объединяются для генерации реального ключевого материала, как описано ниже. Процесс вывода ключей включает входные секреты и стенограмму (transcript) согласования. Отметим, что в результате наличия в стенограмме согласования случайных значений из сообщений Hello каждое согласование будет создавать разные секреты для трафика даже при использовании одинаковых входных секретов как в случае использования одного PSK для множества соединений.

7.1. Планирование ключей

Процесс вывода ключей использует функции HKDF-Extract и HKDF-Expand, определённые для HKDF [RFC5869], а также функции, определённые ниже

```
HKDF-Expand-Label(Secret, Label, Context, Length) = HKDF-Expand(Secret, HkdfLabel, Length)
```

где HkdfLabel имеет вид

```
struct {
    uint16 length = Length;
    opaque label<7..255> = "tls13 " + Label;
    opaque context<0..255> = Context;
} HkdfLabel;

Derive-Secret(Secret, Label, Messages) =
     HKDF-Expand-Label(Secret, Label, Transcript-Hash(Messages), Hash.length)
```

Функция Hash, используемая Transcript-Hash и HKDF, является хэш-функцией шифра. Hash.length указывает выходной размер в байтах. Сообщения являются конкатенацией указанных согласующих сообщений, включая их поля типа и

размера, но не включая заголовков уровня Record. Отметим, что в некоторых случаях в HKDF-Expand-Label передаётся Context нулевого размера (указывается строкой ""). Заданные в этом документе метки являются строками ASCII без NUL-символа в конце.

Примечание. При использовании обычных хэш-функций любая метка размером более 12 символов требует дополнительного вызова хэш-функции. Метки в данной спецификации выбраны с учётом этого ограничения.

Ключи выводятся из двух входных секретов с использованием функций HKDF-Extract и Derive-Secret. Общий шаблон для добавления нового секрета использует HKDF-Extract с текущим состоянием секрета в качестве Salt и новым секретом IKM[1], который будет добавлен. Входные секреты для TLS 1.3 приведены ниже.

- PSK (заранее известный общий ключ, полученный извне или выведенный из значения resumption_master_secret в предыдущем соединении).
- Общий секрет (EC)DHE (7.4. Расчёт общего секрета (EC)DHE).

Это создаёт полную процедуру вывода ключа, показанную ниже. Используемые обозначения приведены ниже.

- HKDF-Extract создаётся принятием аргумента Salt «сверху» и аргумента IKM «слева» с выводом «вниз» и выходным именем «справа».
- Аргумент Secret в Derive-Secret указывается входящей стрелкой. Например, Early Secret является Secret для генерации client_early_traffic_secret.
- 0 указывает строку с Hash.length = 0.

```
          0
          |
          v
PSK ->  HKDF-Extract = Early Secret
          |
          +-----> Derive-Secret(., "ext binder" | "res binder", "")
          |                     = binder_key
          |
          +-----> Derive-Secret(., "c e traffic", ClientHello)
          |                     = client_early_traffic_secret
          |
          +-----> Derive-Secret(., "e exp master", ClientHello)
          |                     = early_exporter_master_secret
          v
    Derive-Secret(., "derived", "")
          |
          v
(EC)DHE -> HKDF-Extract = Handshake Secret
          |
          +-----> Derive-Secret(., "c hs traffic",
          |                     ClientHello...ServerHello)
          |                     = client_handshake_traffic_secret
          |
          +-----> Derive-Secret(., "s hs traffic",
          |                     ClientHello...ServerHello)
          |                     = server_handshake_traffic_secret
          v
    Derive-Secret(., "derived", "")
          |
          v
0 -> HKDF-Extract = Master Secret
          |
          +-----> Derive-Secret(., "c ap traffic",
          |                     ClientHello...Finished от сервера)
          |                     = client_application_traffic_secret_0
          |
          +-----> Derive-Secret(., "s ap traffic",
          |                     ClientHello...Finished от сервера)
          |                     = server_application_traffic_secret_0
          |
          +-----> Derive-Secret(., "exp master",
          |                     ClientHello...Finished от сервера)
          |                     = exporter_master_secret
          |
          +-----> Derive-Secret(., "res master",
                                ClientHello...Finished от клиента)
                                = resumption_master_secret
```

Общая схема заключается в том, что секреты, показанные слева, являются просто необработанной энтропией без контекста, тогда как секреты справа включают контекст согласования и поэтому могут использоваться для создания рабочих ключей без дополнительного контекста. Отметим, что разные вызовы Derive-Secret могут давать различные аргументы Messages даже с одним секретом. В обмене 0-RTT вызовы Derive-Secret делаются с 4 разными стенограммами (transcript), а в 1-RTT вызовы используют три разных стенограммы.

При недоступности заданного секрета используется 0-значение, представляющее собой строку нулей размером Hash.length байтов. Отметим, что это не означает пропуска циклов (раундов), поэтому Early Secret все равно будет оставаться HKDF-Extract(0, 0), если PSK не используется. Для вычисления binder_key меткой служит «ext binder» для внешних PSK (полученных не от TLS) и «res binder» для восстановительных PSK (представляются как основной секрет из предыдущего согласования). Разные метки предотвращают замену одного PSK другим.

[1]Input Keying Material - входной ключевой материал.

Имеется много возможных значений Early Secret в зависимости от выбранного сервером PSK. Клиент должен рассчитать одно значение для каждого возможного PSK, а если PSK не выбран, нужно рассчитать Early Secret, соответствующий нулевому PSK.

После вывода всех нужных значений из данного секрета этот секрет **следует** уничтожить.

7.2. Обновление секретов трафика

Когда согласование завершено, любая из сторон может обновить свои ключи трафика для передачи, используя согласующее сообщение KeyUpdate, определённое в параграфе 4.6.3. Новые ключи трафика создаётся путём генерации client_/server_application_traffic_secret_N+1 из client_/server_application_traffic_secret_N, как описано в предыдущем параграфе с последующим выводом ключей трафика, описанным в параграфе7.3.

Следующее значение значение application_traffic_secret рассчитывается как

```
application_traffic_secret_N+1 =
    HKDF-Expand-Label(application_traffic_secret_N, "traffic upd", "", Hash.length)
```

После расчёта client_/server_application_traffic_secret_N+1 и связанных ключей трафика реализации **следует** удалить client_/server_application_traffic_secret_N и связанные ключи трафика.

7.3. Расчёт ключей трафика

Ключевой материал для трафика создаётся с использованием перечисленных ниже входных значений.

- Значение секрета.
- Назначение создаваемого материала.
- Размер создаваемого ключа.

Для генерации ключевого материала из входных значений применяются приведённые ниже расчёты.

```
[sender]_write_key = HKDF-Expand-Label(Secret, "key", "", key_length)
[sender]_write_iv  = HKDF-Expand-Label(Secret, "iv", "", iv_length)
```

где [sender] указывает передающую сторону. Значения Secret для каждого типа записи показаны в таблице.

Тип записи	*Секрет*
0-RTT Application	client_early_traffic_secret
Handshake	[sender]_handshake_traffic_secret
Application Data	[sender]_application_traffic_secret_N

Весь ключевой материал для трафика рассчитывается заново при каждой смене Secret (например, при переходе от ключей согласования к ключам для данных приложений или при обновлении ключа).

7.4. Расчёт общего секрета (EC)DHE

7.4.1. Конечное поле Diffie-Hellman

Для групп конечных полей выполняется традиционный расчёт Diffie-Hellman [DH76]. Согласованный ключ (Z) преобразуется в строку байтов с кодированием big-endian и дополнением нулями слева до заданного размера простого числа. Эта строка служит общим секретом при планировании ключей, как указано выше.

Отметим, что эта конструкция отличается от прежних версия TLS, в которых начальные нули удалялись.

7.4.2. Эллиптическая кривая Diffie-Hellman

Для secp256r1, secp384r1 и secp521r1 расчёты ECDH (включая генерацию параметра и ключа, а также расчёт общего секрета) выполняются в соответствии с [IEEE1363] по схеме ECKAS-DH1 с отображением отождествления в качестве функции вывода ключей (Key derivation function или KDF), так что общий секрет является координатой x точки эллиптической кривой общего секрета ECDH, представленной в виде строки октетов. Отметим, что эта строка (Z в терминологии IEEE 1363) как вывод FE2OSP[1] имеет постоянный размер для любого данного поля и отбрасывание нулей в начале строки **недопустимо**.

Использование отождествления KDF является технической деталью. Полная картина включает использование ECDH с нетривиальной KDF, поскольку TLS не использует этот секрет напрямую нигде, кроме расчёта других секретов.

Для X25519 и X448 расчёты ECDH описаны ниже.

- Открытый ключ для включения в структуру KeyShareEntry.key_exchange является результатом применения функции скалярного умножения ECDH к секретному ключу подходящего размера (вход скаляра) и стандартной открытой базовой точке (вход u-координаты).
- Общий секрет ECDH является результатом применения функции скалярного умножения ECDH к секретному ключу подходящего размера (вход скаляра) и открытому ключу партнёра (вход u-координаты). Результат используется в необработанном виде (raw).

Для этих кривых реализации **следует** применять подход, описанный в [RFC7748] для расчёта общего секрета Diffie-Hellman. Реализация **должна** проверить, не является ли рассчитанный общий секрет Diffie-Hellman нулевым значением и в этом случае прерывать расчёт, как описано в разделе 6 [RFC7748]. Если разработчики применяют альтернативную реализацию этих кривых, им **следует** выполнять дополнительные проверки, как указано в разделе 7 [RFC7748].

7.5. Экспортёры

[RFC5705] определяет экспортёров ключевого материала для TLS в терминах псевдослучайной функции TLS (PRF). Этот документ заменяет PRF на HKDF, для чего требуется новая конструкция. Интерфейс экспортёра не меняется.

Значение экспортёра определяется выражением

[1]The Field Element to Octet String Conversion Primitive - примитив для преобразования поля в строку октетов.

```
TLS-Exporter(label, context_value, key_length) =
    HKDF-Expand-Label(Derive-Secret(Secret, label, ""),
                      "exporter", Hash(context_value), key_length)
```

Значением Secret является early_exporter_master_secret или exporter_master_secret. Реализации должны использовать exporter_master_secret, если приложение явно не задаёт иное. Значение early_exporter_master_secret определено для использования в тех случаях, когда экспортёр требуется для данных 0-RTT. Для раннего экспортёра **рекомендуется** отдельный интерфейс, это избавляет от непреднамеренного использования одного экспортёра вместо другого.

Если контекст не задан, context_value имеет нулевой размер. Поэтому при отсутствии контекста вычисляется такое же значение, как при пустом контексте. Это отличается от прежних версий TLS, где пустой контекст давал результат, отличный от случая с отсутствием контекста. На момент публикации этого документа не было выделено меток экспортёров, используемых с контекстом и без него. В будущих спецификациях **недопустимо** определять экспортёры, которые разрешают использовать пустой контекст и отсутствие контекста с одной меткой. Новым применениям экспортёров **следует** обеспечивать контекст во всех расчётах экспортёра (контекст может быть пустым).

Требования к формату меток экспортёров определены в разделе 4 [RFC5705].

8. 0-RTT и Anti-Replay

Как отмечено в параграфе 2.3 и приложении E.5, TLS не обеспечивает встроенной защиты от повторного использования для данных 0-RTT. С этим связаны две угрозы, которые следует принимать во внимание.

- Атакующие в сети, которые могут организовать replay-атаку, просто дублируя поток данных 0-RTT.
- Атакующие в сети, которые используют поведение повторов клиента, чтобы организовать получение сервером нескольких копий прикладного сообщения. Эта угроза уже существует в неком виде, поскольку клиенты, желающие обеспечить отказоустойчивость, реагируют на сетевые ошибки попытками повторять запросы. Однако 0-RTT добавляет дополнительное измерение для любой серверной системы, которая не поддерживает глобально согласованное состояние сервера. В частности, если серверная система имеет множество зон и квитанции из зоны A не принимаются зоной B, атакующий может дублировать ClientHello и ранние данные, предназначенные для зоны A, также и в зону B. В зоне A данные будут восприняты в 0-RTT, но зона B на сервере будет отвергать данные 0-RTT и форсирует полное согласование. Если атакующий заблокирует ServerHello от A, клиент будет завершать согласование с B и может повторить запрос, ведущий к дублированию на серверной системе в целом.

Первый тип атак можно предотвратить с помощью общего состояния для гарантии восприятия данных 0-RTT не более одного раза. Серверам **следует** поддерживать этот уровень защиты путём реализации одного из описанных в этом разделе методов или аналогичных мер. Однако ясно, что в результате эксплуатационных проблем не все реализации будут поддерживать состояние на этом уровне. При обычной работе клиенты не будут знать, какие механизмы серверы реализуют на деле, поэтому они **должны** передавать ранние данные лишь в том случае, когда их повтор не опасен.

В дополнение к прямому влиянию повторов имеется класс атак, где даже операции, обычно считающиеся идемпотентными, могут использоваться в большом числе повторов (timing-атаки, истощение ресурсов и др., как описано в приложении E.5). Это можно смягчить путём обеспечения возможности воспроизведения данных 0-RTT лишь ограниченное число раз. Сервер **должен** гарантировать, что любой экземпляр (машина, процесс или иной элемент, относящийся к инфраструктуре сервера) будет воспринимать 0-RTT для одного и того же согласования 0-RTT не более 1 раза - это ограничит число повторов числом экземпляров сервера. Такая гарантия может быть достигнута путём локальной записи данных из недавно полученных ClientHello и отклонения повторов или иным методом, обеспечивающим такую же или более строгую гарантию. «Не более одного раза на экземпляр сервера» является минимальным требованием, серверам **следует** дополнительно ограничивать повторы 0-RTT, когда это возможно.

Второй тип атак невозможно предотвратить на уровне TLS и это должны делать все приложения. Отметим, что любое приложение, чьи клиенты реализуют какие-либо повторы, уже **должны** поддерживать ту или иную защиту от повторного использования (anti-replay).

8.1. Одноразовые квитанции

Простейшим способом защиты от повторного использования для сервера является восприятие каждой сеансовой квитанции лишь 1 раз. Например, сервер может поддерживать базу остающихся действительными квитанций, удаляя из неё запись при использовании квитанции. При получении отсутствующей в базе квитанции сервер будет возвращаться к полному согласованию.

Если квитанция не является автономной и служит ключом базы данных, соответствующий PSK исключается из употребления при организации соединения с этим PSK для обеспечения эффективной защиты. Это повышает уровень защищенности для всех данных 0-RTT и PSK при использовании PSK без (EC)DHE.

Поскольку этому механизму нужна общая база данных для всех серверов среды с множеством распределенных серверов, при высокой скорости организации соединений использование PSK 0-RTT может оказаться сложнее применения самошифрованных (автономных) квитанций. В отличии от баз данных сеансов сеансовые квитанции позволяют организовывать сессии на основе PSK без согласованного хранения, хотя при разрешённом 0-RTT все равно нужно согласованное хранение для предотвращения повторного использования данных 0-RTT, как указано в следующем параграфе.

8.2. Запись ClientHello

Другим вариантом защиты от повторного использования является запись уникального значения, выведенного из ClientHello (обычно это случайное значение или привязка PSK), и отклонение дубликатов. Запись всех ClientHello ведёт к безграничному росту состояния, но сервер может записывать сообщения в заданном временном окне и применять obfuscated_ticket_age для гарантии того, что квитанции не будут повторно применяться за пределами этого окна.

Для реализации этого сервер при получении ClientHello сначала проверяет привязку PSK, как описано в параграфе 4.2.11. Расширение PSK. Затем рассчитывается expected_arrival_time, как описано в следующем параграфе, и отвергаются 0-RTT, выходящие за пределы окна записи, с переходом к согласованию 1-RTT.

Если expected_arrival_time попадает в окно, сервер проверяет, записано ли соответствующее сообщение ClientHello. Если сообщение найдено, сервер прерывает согласование с сигналом illegal_parameter или воспринимает PSK, отвергая 0-RTT. Если соответствующего ClientHello не найдено, сервер воспринимает 0-RTT и сохраняет ClientHello, пока expected_arrival_time находится в окне. Сервер **может** также реализовать хранение данных с ложными срабатываниями, таких как фильтры Bloom, которые в этом случае **должны** реагировать на возможный повтор, отвергая 0-RTT, но прерывать согласование **недопустимо**.

Сервер **должен** выводить ключи хранилища только из проверенных разделов ClientHello. Если ClientHello содержит несколько отождествлений PSK, атакующий может создать множество ClientHello с другим значением привязки для менее предпочтительного отождествления в предположении, что сервер не проверит его (как рекомендовано в параграфе 4.2.11). Т. е. если клиент передаёт PSK A и B, но сервер предпочитает A, атакующий может изменить привязку для B, не влияя на привязку для A. Если привязка B является частью ключа хранилища, это сообщение ClientHello не будет казаться дубликатом, что приведёт к его восприятию и может вызывать побочные эффекты (например, загрязнение кэша), хотя данные 0-RTT не будут расшифрованы в результате использования другого ключа. Если в качестве ключа хранилища используется проверенная привязка или ClientHello.random, такая атака становится невозможной.

Поскольку этот механизм не требует хранить все остающиеся квитанции, его реализация может оказаться проще в распределенных системах с высокой скоростью восстановления и 0-RTT за счёт возможно снижения уровня защиты от повторов из-за сложности надёжного хранения и извлечения принятых сообщений ClientHello. Во многих таких системах непрактично поддерживать глобально согласованное хранилище всех полученных ClientHello. В этом случае лучшая защита от повторного использования обеспечивается наличием одной зоны хранения, полномочной для данной квитанции и отвергающей 0-RTT с квитанциями из других зон. Этот подход предотвращает простые повторы от атакующих, поскольку лишь одна зона будет воспринимать данные 0-RTT. Более слабым решением будет реализация отдельного хранилища для каждой зоны и восприятие 0-RTT в любой зоне. Такой подход ограничивает число повторов до одного на зону. Разумеется, дублирование прикладных сообщений остаётся возможным в обоих вариантах.

При новом запуске реализации ей **следует** отвергать 0-RTT, пока окно записи включает время старта. Иначе возникает риск получения повтора пакетов, которые были отправлены в течение этого интервала.

> Примечание. Если часы клиента работают быстрее часов сервера, сообщение ClientHello может быть получено за пределами окна (в будущем) и в этом случае оно может быть воспринято для 1-RTT, что вынудит клиента к повтору и последующему восприятию 0-RTT. Это является другим вариантом второго типа атак, описанного в параграфе 8.

8.3. Проверка свежести

Поскольку в ClientHello указано время отправки, можно убедиться в том, что сообщение отправлено недавно и принимать 0-RTT лишь для свежих ClientHello, используя для остальных согласование 1-RTT. Это требуется для механизма хранения ClientHello, описанного в параграфе 8.2, поскольку иначе серверу придётся хранить неограниченное число ClientHello, а также полезно для оптимизации автономных одноразовых квитанций, поскольку позволяет отвергать ClientHello, которые нельзя использовать для 0-RTT.

Для реализации этого механизма серверу нужно сохранять время генерации сеансовой квитанции, смещённое на оценку времени кругового обхода между клиентом и сервером adjusted_creation_time = creation_time + estimated_RTT. Это значение можно закодировать в квитанции, избавляясь от необходимости хранить каждую выпущенную квитанцию. Сервер может определить клиентское представление возраста квитанции, вычитая значение ticket_age_add в квитанции из obfuscated_ticket_age в клиентском расширении pre_shared_key. Сервер может определить expected_arrival_time для ClientHello как expected_arrival_time = adjusted_creation_time + clients_ticket_age. При получении нового ClientHello значение expected_arrival_time сравнивается с текущим временем сервера и при разнице больше определённого значения 0-RTT отвергается, хотя завершение согласования 1-RTT может быть разрешено.

Существует несколько источников ошибок, которые могут приводить к несоответствию expected_arrival_time и измеренного времени. Различия в скорости хода часов клиента и сервера вероятно будут минимальными, хотя абсолютное отклонение может быть очень большим. Задержки в сети являются наиболее вероятной причиной расхождения. Сообщения NewSessionTicket и ClientHello могут передаваться повторно, что ведёт к задержке, которая может быть скрыта протоколом TCP. Для клиентов в Internet это предполагает окно порядка 10 секунд с учётом расхождения часов и погрешностей измерения. В других случаях размер окна может отличаться. Распределение сдвига часов не является симметричным, поэтому оптимальным компромиссом будет асимметричный диапазон допустимых значений рассогласования.

Отметим, что проверки лишь свежести недостаточно для предотвращения повторов, поскольку она не обнаруживает повторов в окне ошибок, которое (в зависимости от пропускной способности и ёмкости системы) в реальных условиях может включать миллиарды повторов. Кроме того, проверка сложна лишь при получении ClientHello, а не последующих записей Application Data. После восприятия ранних данных записи могут продолжать приходить на сервер ещё достаточно долго.

9. Требования соответствия

9.1. Обязательные шифронаборы

В отсутствие стандарта профиля приложения, задающего иное, должны выполняться приведённые ниже условия.

Соответствующее TLS приложение **должно** реализовать шифр TLS_AES_128_GCM_SHA256 [GCM] и **следует** реализовать шифры TLS_AES_256_GCM_SHA384 [GCM] и TLS_CHACHA20_POLY1305_SHA256 [RFC8439] (Приложение B.4).

Соответствующее TLS приложение **должно** поддерживать цифровые подписи rsa_pkcs1_sha256 (сертификаты), rsa_pss_rsae_sha256 (CertificateVerify и сертификаты) и ecdsa_secp256r1_sha256. Соответствующее TLS приложение **должно** поддерживать обмен ключами secp256r1 (NIST P-256) и **следует** поддерживать обмен X25519 [RFC7748].

9.2. Обязательные расширения

В отсутствие стандарта профиля приложения, задающего иное, соответствующее TLS приложение **должно** реализовать перечисленные ниже расширения TLS.

- Поддерживаемые версии (supported_versions, 4.2.1. Поддерживаемые версии).
- Cookie (cookie, 4.2.2. Cookie).
- Алгоритмы подписи (signature_algorithms, 4.2.3. Алгоритмы подписи).
- Сертификаты алгоритма подписи (signature_algorithms_cert, 4.2.3. Алгоритмы подписи).
- Согласованные группы (supported_groups, 4.2.7. Поддерживаемые группы).
- Общий ключ (key_share, 4.2.8. Совместное использование ключа).
- Указание имени сервера (server_name, параграф 3 of [RFC6066]).

Все приложения **должны** передавать и использовать указанные ниже расширения в отмеченных случаях.

- supported_versions **требуется** для всех сообщений ClientHello, ServerHello, HelloRetryRequest.
- signature_algorithms **требуется** для аутентификации сертификатов.
- supported_groups **требуется** для сообщений ClientHello, использующих обмен ключами DHE или ECDHE.
- key_share **требуется** для обмена ключами DHE или ECDHE.
- pre_shared_key **требуется** для согласования ключа PSK.
- psk_key_exchange_modes **требуется** для согласования ключа PSK.

Считается, что клиент пытается выполнить согласование с использованием этой спецификации, если ClientHello содержит supported_versions со значением 0x0304 в теле. Такое сообщение ClientHello **должно** соответствовать приведённым ниже требованиям.

- Если нет расширения pre_shared_key, сообщение **должно** включать signature_algorithms и supported_groups.
- При наличии расширения supported_groups сообщение **должно** также содержать key_share и наоборот. Разрешается указывать пустой вектор KeyShare.client_shares.

Серверы, принявшие сообщение ClientHello, которое не соответствует этим требованиям, **должны** прервать согласование с сигналом missing_extension.

Кроме того, все реализации **должны** поддерживать использование расширения server_name с приложениями, которые способны его использовать. Серверы **могут** требовать от клиентов передачи пригодного расширения server_name. Требующим это расширение серверам **следует** отвечать на ClientHello без расширения server_name прерыванием соединения с сигналом missing_extension.

9.3. Инварианты протокола

В этом параграфе описаны инварианты, которым должны следовать конечные и промежуточные точки TLS. Эти применимо и к прежним версиям TLS.

Протокол TLS спроектирован для защищённого и совместимого расширения. Более новым клиентам и серверам при взаимодействии с партнёрами следует согласовывать наиболее предпочтительные общие параметры. Согласование TLS обеспечивает защиту от понижения версии и промежуточные устройства, передающие трафик между новыми клиентами и серверами без завершения соединений TLS, не должны иметь возможности влиять на согласование (Приложение E.1). Однако системы обновляются с разной скоростью, поэтому новые клиенты и серверы **могут** продолжать поддержку более старых параметров для совместимости со старыми конечными точками.

Чтобы это работало, реализации **должны** корректно обрабатывать расширяемые поля.

- Клиент, передавший ClientHello, **должен** поддерживать все анонсированные в нем параметры. Иначе могут возникнуть проблемы, если сервер выберет один из неподдерживаемых параметров.
- Сервер, получивший ClientHello, **должен** корректно игнорировать все нераспознанные шифры, расширения и другие параметры. Иначе может возникнуть отказ при работе с более новым клиентом. В TLS 1.3 клиент, получивший CertificateRequest или NewSessionTicket, **должен** игнорировать все неизвестные расширения.
- Промежуточное устройство, завершающее соединение TLS, **должно** вести себя как сервер TLS (по отношению к исходному клиенту), включая наличие сертификата, воспринимаемого клиентом, а также соответствовать требованиям к клиенту TLS (по отношению к конечному серверу), включая проверку сертификата сервера. В частности, это устройство **должно** создавать своё сообщение ClientHello, содержащее лишь понятные ему параметры, а также **должно** генерировать свежее случайное значение ServerHello вместо пересылки полученного.

  Отметим, что требования протокола TLS и анализ защиты применяются лишь к двум соединениям по отдельности. Безопасное развёртывание завершающей точки TLS требует дополнительной защиты, но этот вопрос выходит за рамки документа.
- Промежуточным устройствам, которые пересылают непонятые ими параметры ClientHello, **недопустимо** обрабатывать какие-либо сообщения, кроме ClientHello. Они **должны** пересылать весь последующий трафик без изменений. В противном случае могут возникнуть проблемы взаимодействия с новым клиентом или сервером.

  Пересылаемые ClientHello могут содержать анонсы возможностей, не поддерживаемых промежуточным устройством и отклик может включать новые дополнения TLS, которые это устройство не понимает. Такие

дополнения **могут** произвольно менять любые сообщения, кроме ClientHello. В частности, могут меняться значения в ServerHello, формат ServerHello и формат TLSCiphertext.

Разработка TLS 1.3 ограничивалась широким распространением не соответствующих TLS промежуточных устройств (Приложение D.4), однако это не отменяет инварианты. Эти промежуточные устройства остаются несоответствующими.

10. Вопросы безопасности

Вопросы безопасности рассматриваются на протяжении всего документа и, в частности, в приложениях C, D, E.

11. Взаимодействие с IANA

В этом документе используется несколько реестров, созданных в [RFC4346] и обновлённых в [RFC8447]. Агентство IANA обновило реестры в соответствии с этим документом. Реестры и правила для них указаны ниже.

- TLS Cipher Suites - значения с первым байтом из диапазона 0-254 (десятичные) выделяются по процедуре Specification Required [RFC8126], значения с первым байтом 255 (десятичное) служат для приватных приложений [RFC8126]. Агентство IANA добавило в реестр шифры, указанные в Приложении B.4. Столбцы Value и Description взяты из таблицы, в столбцах DTLS-OK и Recommended для новых шифров указано значение Y.
- TLS ContentType - новые значения выделяются по процедуре Standards Action [RFC8126].
- TLS Alerts - новые значения выделяются по процедуре Standards Action [RFC8126]. Агентство IANA внесло в этот реестр значения из приложения B.2. В колонке DTLS-OK указано Y для всех этих значений. Для значений, указанных как _RESERVED, приведены комментарии с описанием их применения.
- - новые значения выделяются по процедуре Standards Action [RFC8126]. Агентство IANA обновило этот реестр, переименовав элемент 4 (NewSessionTicket) в new_session_ticket, и заполнило значениями из Приложения B.3. В колонке DTLS-OK указано Y для всех этих значений. Для значений, указанных как _RESERVED, приведены комментарии с описанием их применения.

Этот документ использует реестр TLS ExtensionType Values, созданный в [RFC4366]. Агентство IANA обновило реестр в соответствии с этим документом. Изменения приведены ниже.

- Изменена политика регистрации. Значения с первым байтом из диапазона 0-254 (десятичные) назначаются по процедуре Specification Required [RFC8126], значения с первым байтом 255 (десятичное) служат для приватных приложений [RFC8126].
- Добавлены расширения key_share, pre_shared_key, psk_key_exchange_modes, early_data, cookie, supported_versions, certificate_authorities, oid_filters, post_handshake_auth и signature_algorithms_cert со значениями, заданными у этом документе и Recommended = Y.
- Включена колонка TLS 1.3, где указываются сообщения, в которых может присутствовать расширение. Эта колонка заполнена значениями в соответствии с таблицей из параграфа 4.2, а отсутствующие в этой таблице расширения помечены символом «-», указывающим, что они не применяются в TLS 1.3.

Этот документ обновляет запись в реестре TLS Certificate Types, созданную в [RFC6091] и обновлённую в [RFC8447]. Агентство IANA обновило запись для значения 1, указав имя OpenPGP_RESERVED, Recommended = N и комментарий «Used in TLS versions prior to 1.3.» (используется в TLS до версии 1.3).

Этот документ обновляет реестр TLS Certificate Status Types, созданный в [RFC6961]. Агентство IANA обновило запись для значения 2, указав имя ocsp_multi_RESERVED и комментарий «Used in TLS versions prior to 1.3.» (используется в TLS до версии 1.3).

Этот документ обновляет две записи в реестре TLS Supported Groups (создан в [RFC4492] с другим именем, сейчас поддерживается [RFC8422]), а также обновлён в [RFC7919] и [RFC8447]. Записи для значений 29 и 30 (x25519 и x448) обновлены со ссылкой на этот документ.

Кроме того, этот документ определяет два новых реестра, поддерживаемых IANA.

- TLS SignatureScheme. Значения с первым байтом из диапазона 0-253 (десятичные) назначаются по процедуре Specification Required [RFC8126]. Значения с первым байтом 254 или 255 (десятичные) зарезервированы как Private Use [RFC8126]. Значения с первым байтом из диапазона 0-6 или вторым байтом из диапазона 0-3 в настоящее время не выделены и зарезервированы для совместимости с прежними версиями. Этот реестр имеет колонку Recommended (рекомендовано). В реестр изначально включены значения, описанные в параграфе 4.2.3. Рекомендованы значения ecdsa_secp256r1_sha256, ecdsa_secp384r1_sha384, rsa_pss_rsae_sha256, rsa_pss_rsae_sha384, rsa_pss_rsae_sha512, rsa_pss_pss_sha256, rsa_pss_pss_sha384, rsa_pss_pss_sha512 и ed25519. В колонке Recommended указывается значение N (нет), пока явно не запрошено значение Y (да), для чего требуется процедура Standards Action [RFC8126]. Для замены Y->N **требуется** процедура IESG Approval.
- TLS PskKeyExchangeMode. Значения из диапазона 0-253 (десятичные) выделяются по процедуре Specification Required [RFC8126]. Значения 254 и 255 (десятичные) зарезервированы как Private Use [RFC8126]. Этот реестр имеет колонку Recommended (рекомендовано). В реестр изначально включены значения psk_ke (0) и psk_dhe_ke (1), указанные как рекомендуемые. В колонке Recommended указывается N (нет), пока явно не запрошено значение Y (да), для чего требуется процедура Standards Action [RFC8126]. Для замены Y->N **требуется** процедура IESG Approval.

12. Литература

12.1. Нормативные документы

[DH76] Diffie, W. and M. Hellman, "New directions in cryptography", IEEE Transactions on Information Theory, Vol. 22 No. 6, pp. 644-654, DOI 10.1109/TIT.1976.1055638, November 1976.

[ECDSA] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI ANS X9.62-2005, November 2005.

[GCM] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D, DOI 10.6028/NIST.SP.800-38D, November 2007.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

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

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <https://www.rfc-editor.org/info/rfc5705>.

[RFC5756] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Updates for RSAES-OAEP and RSASSA-PSS Algorithm Parameters", RFC 5756, DOI 10.17487/RFC5756, January 2010, <https://www.rfc-editor.org/info/rfc5756>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC6655] McGrew, D. and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)", RFC 6655, DOI 10.17487/RFC6655, July 2012, <https://www.rfc-editor.org/info/rfc6655>.

[RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <https://www.rfc-editor.org/info/rfc6960>.

[RFC6961] Pettersen, Y., "The Transport Layer Security (TLS) Multiple Certificate Status Request Extension", RFC 6961, DOI 10.17487/RFC6961, June 2013, <https://www.rfc-editor.org/info/rfc6961>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/info/rfc6962>.

[RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <https://www.rfc-editor.org/info/rfc6979>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC7507] Moeller, B. and A. Langley, "TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks", RFC 7507, DOI 10.17487/RFC7507, April 2015, <https://www.rfc-editor.org/info/rfc7507>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[RFC7919] Gillmor, D., "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)", RFC 7919, DOI 10.17487/RFC7919, August 2016, <https://www.rfc-editor.org/info/rfc7919>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <https://www.rfc-editor.org/info/rfc8439>.

[SHS] Dang, Q., "Secure Hash Standard (SHS)", National Institute of Standards and Technology report, DOI 10.6028/NIST.FIPS.180-4, August 2015.

[X690] ITU-T, "Information technology -- ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8825-1:2015, November 2015.

12.2. Дополнительная литература

[AEAD-LIMITS] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", August 2017, <http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>.

[BBFGKZ16] Bhargavan, K., Brzuska, C., Fournet, C., Green, M., Kohlweiss, M., and S. Zanella-Beguelin, "Downgrade Resilience in Key-Exchange Protocols", Proceedings of IEEE Symposium on Security and Privacy (San Jose), DOI 10.1109/SP.2016.37, May 2016.

[BBK17] Bhargavan, K., Blanchet, B., and N. Kobeissi, "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate", Proceedings of IEEE Symposium on Security and Privacy (San Jose), DOI 10.1109/SP.2017.26, May 2017.

[BDFKPPRSZZ16] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pan, J., Protzenko, J., Rastogi, A., Swamy, N., Zanella-Beguelin, S., and J. Zinzindohoue, "Implementing and Proving the TLS 1.3 Record Layer", Proceedings of IEEE Symposium on Security and Privacy (San Jose), May 2017, <https://eprint.iacr.org/2016/1178>.

[Ben17a] Benjamin, D., "Presentation before the TLS WG at IETF 100", November 2017, <https://datatracker.ietf.org/meeting/100/materials/slides-100-tls-sessa-tls13/>.

[Ben17b] Benjamin, D., "Additional TLS 1.3 results from Chrome", message to the TLS mailing list, 18 December 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25168.html>.

[Blei98] Bleichenbacher, D., "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1", Proceedings of CRYPTO '98, 1998.

[BMMRT15] Badertscher, C., Matt, C., Maurer, U., Rogaway, P., and B. Tackmann, "Augmented Secure Channels and the Goal of the TLS 1.3 Record Layer", ProvSec 2015, September 2015, <https://eprint.iacr.org/2015/394>.

[BT16] Bellare, M. and B. Tackmann, "The Multi-User Security of Authenticated Encryption: AES-GCM in TLS 1.3", Proceedings of CRYPTO 2016, July 2016, <https://eprint.iacr.org/2016/564>.

[CCG16] Cohn-Gordon, K., Cremers, C., and L. Garratt, "On Post-compromise Security", IEEE Computer Security Foundations Symposium, DOI 10.1109/CSF.2016.19, July 2015.

[CHECKOWAY] Checkoway, S., Maskiewicz, J., Garman, C., Fried, J., Cohney, S., Green, M., Heninger, N., Weinmann, R., Rescorla, E., and H. Shacham, "A Systematic Analysis of the Juniper Dual EC Incident", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS '16, DOI 10.1145/2976749.2978395, October 2016.

[CHHSV17] Cremers, C., Horvat, M., Hoyland, J., Scott, S., and T. van der Merwe, "Awkward Handshake: Possible mismatch of client/server view on client authentication in post-handshake mode in Revision 18", message to the TLS mailing list, 10 February 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg22382.html>.

[CHSV16] Cremers, C., Horvat, M., Scott, S., and T. van der Merwe, "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication", Proceedings of IEEE Symposium on Security and Privacy (San Jose), DOI 10.1109/SP.2016.35, May 2016, <https://ieeexplore.ieee.org/document/7546518/>.

[CK01] Canetti, R. and H. Krawczyk, "Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels", Proceedings of Eurocrypt 2001, DOI 10.1007/3-540-44987-6_28, April 2001.

[CLINIC] Miller, B., Huang, L., Joseph, A., and J. Tygar, "I Know Why You Went to the Clinic: Risks and Realization of HTTPSTraffic Analysis", Privacy Enhancing Technologies, pp. 143-163, DOI 10.1007/978-3-319-08506-7_8, 2014.

[DFGS15] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates", Proceedings of ACM CCS 2015, October 2015, <https://eprint.iacr.org/2015/914>.

[DFGS16] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 Full and Pre-shared Key Handshake Protocol", TRON 2016, February 2016, <https://eprint.iacr.org/2016/081>.

[DOW92] Diffie, W., van Oorschot, P., and M. Wiener, "Authentication and authenticated key exchanges", Designs, Codes and Cryptography, DOI 10.1007/BF00124891, June 1992.

[DSS] National Institute of Standards and Technology, U.S. Department of Commerce, "Digital Signature Standard (DSS)", NIST FIPS PUB 186-4, DOI 10.6028/NIST.FIPS.186-4, July 2013.

[FG17] Fischlin, M. and F. Guenther, "Replay Attacks on Zero Round-Trip Time: The Case of the TLS 1.3 Handshake Candidates", Proceedings of EuroS&P 2017, April 2017, <https://eprint.iacr.org/2017/082>.

[FGSW16] Fischlin, M., Guenther, F., Schmidt, B., and B. Warinschi, "Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3", Proceedings of IEEE Symposium on Security and Privacy (San Jose), DOI 10.1109/SP.2016.34, May 2016, <https://ieeexplore.ieee.org/document/7546517/>.

[FW15] Weimer, F., "Factoring RSA Keys With TLS Perfect Forward Secrecy", September 2015.

[HCJC16] Husak, M., Cermak, M., Jirsik, T., and P. Celeda, "HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting", EURASIP Journal on Information Security, Vol. 2016, DOI 10.1186/s13635-016-0030-7, February 2016.

[HGFS15] Hlauschek, C., Gruber, M., Fankhauser, F., and C. Schanes, "Prying Open Pandora's Box: KCI Attacks against TLS", Proceedings of USENIX Workshop on Offensive Technologies, August 2015.

[IEEE1363] IEEE, "IEEE Standard Specifications for Public Key Cryptography", IEEE Std. 1363-2000, DOI 10.1109/IEEESTD.2000.92292.

[JSS15] Jager, T., Schwenk, J., and J. Somorovsky, "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption", Proceedings of ACM CCS 2015, DOI 10.1145/2810103.2813657, October 2015, <https://www.nds.rub.de/media/nds/veroeffentlichungen/2015/08/21/Tls13QuicAttacks.pdf>.

[KEYAGREEMENT] Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", National Institute of Standards and Technology, DOI 10.6028/NIST.SP.800-56Ar3, April 2018.

[Kraw10] Krawczyk, H., "Cryptographic Extraction and Key Derivation: The HKDF Scheme", Proceedings of CRYPTO 2010, August 2010, <https://eprint.iacr.org/2010/264>.

[Kraw16] Krawczyk, H., "A Unilateral-to-Mutual Authentication Compiler for Key Exchange (with Applications to Client Authentication in TLS 1.3", Proceedings of ACM CCS 2016, October 2016, <https://eprint.iacr.org/2016/711>.

[KW16] Krawczyk, H. and H. Wee, "The OPTLS Protocol and TLS 1.3", Proceedings of EuroS&P 2016, March 2016, <https://eprint.iacr.org/2015/978>.

[LXZFH16] Li, X., Xu, J., Zhang, Z., Feng, D., and H. Hu, "Multiple Handshakes Security of TLS 1.3 Candidates", Proceedings of IEEE Symposium on Security and Privacy (San Jose), DOI 10.1109/SP.2016.36, May 2016, <https://ieeexplore.ieee.org/document/7546519/>.

[Mac17] MacCarthaigh, C., "Security Review of TLS1.3 0-RTT", March 2017, <https://github.com/tlswg/tls13-spec/issues/1001>.

[PS18] Patton, C. and T. Shrimpton, "Partially specified channels: The TLS 1.3 record layer without elision", 2018, <https://eprint.iacr.org/2018/634>.

[PSK-FINISHED] Scott, S., Cremers, C., Horvat, M., and T. van der Merwe, "Revision 10: possible attack if client authentication is allowed during PSK", message to the TLS mailing list, 31 October 2015, <https://www.ietf.org/mail-archive/web/tls/current/msg18215.html>.

[REKEY] Abdalla, M. and M. Bellare, "Increasing the Lifetime of a Key: A Comparative Analysis of the Security of Re-keying Techniques", ASIACRYPT 2000, DOI 10.1007/3-540-44448-3_42, October 2000.

[Res17a] Rescorla, E., "Preliminary data on Firefox TLS 1.3 Middlebox experiment", message to the TLS mailing list, 5 December 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25091.html>.

[Res17b] Rescorla, E., "More compatibility measurement results", message to the TLS mailing list, 22 December 2017, <https://www.ietf.org/mail-archive/web/tls/current/msg25179.html>.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, DOI 10.17487/RFC4346, April 2006, <https://www.rfc-editor.org/info/rfc4346>.

[RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 4366, DOI 10.17487/RFC4366, April 2006, <https://www.rfc-editor.org/info/rfc4366>.

[RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, DOI 10.17487/RFC4492, May 2006, <https://www.rfc-editor.org/info/rfc4492>.

[RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", RFC 5077, DOI 10.17487/RFC5077, January 2008, <https://www.rfc-editor.org/info/rfc5077>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <https://www.rfc-editor.org/info/rfc5764>.

[RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, DOI 10.17487/RFC5929, July 2010, <https://www.rfc-editor.org/info/rfc5929>.

[RFC6091] Mavrogiannopoulos, N. and D. Gillmor, "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", RFC 6091, DOI 10.17487/RFC6091, February 2011, <https://www.rfc-editor.org/info/rfc6091>.

[RFC6101] Freier, A., Karlton, P., and P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0", RFC 6101, DOI 10.17487/RFC6101, August 2011, <https://www.rfc-editor.org/info/rfc6101>.

[RFC6176] Turner, S. and T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0", RFC 6176, DOI 10.17487/RFC6176, March 2011, <https://www.rfc-editor.org/info/rfc6176>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/info/rfc6347>.

[RFC6520] Seggelmann, R., Tuexen, M., and M. Williams, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension", RFC 6520, DOI 10.17487/RFC6520, February 2012, <https://www.rfc-editor.org/info/rfc6520>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <https://www.rfc-editor.org/info/rfc7250>.

[RFC7465] Popov, A., "Prohibiting RC4 Cipher Suites", RFC 7465, DOI 10.17487/RFC7465, February 2015, <https://www.rfc-editor.org/info/rfc7465>.

[RFC7568] Barnes, R., Thomson, M., Pironti, A., and A. Langley, "Deprecating Secure Sockets Layer Version 3.0", RFC 7568, DOI 10.17487/RFC7568, June 2015, <https://www.rfc-editor.org/info/rfc7568>.

[RFC7627] Bhargavan, K., Ed., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", RFC 7627, DOI 10.17487/RFC7627, September 2015, <https://www.rfc-editor.org/info/rfc7627>.

[RFC7685] Langley, A., "A Transport Layer Security (TLS) ClientHello Padding Extension", RFC 7685, DOI 10.17487/RFC7685, October 2015, <https://www.rfc-editor.org/info/rfc7685>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <https://www.rfc-editor.org/info/rfc7924>.

[RFC8305] Schinazi, D. and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", RFC 8305, DOI 10.17487/RFC8305, December 2017, <https://www.rfc-editor.org/info/rfc8305>.

[RFC8422] Nir, Y., Josefsson, S., and M. Pegourie-Gonnard, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier", RFC 8422, DOI 10.17487/RFC8422, August 2018, <https://www.rfc-editor.org/info/rfc8422>.

[RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <https://www.rfc-editor.org/info/rfc8447>.

[RFC8449] Thomson, M., "Record Size Limit Extension for TLS", RFC 8449, DOI 10.17487/RFC8449, August 2018, <https://www.rfc-editor.org/info/rfc8449>.

[RSA] Rivest, R., Shamir, A., and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM, Vol. 21 No. 2, pp. 120-126, DOI 10.1145/359340.359342, February 1978.

[SIGMA] Krawczyk, H., "SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and its Use in the IKE Protocols", Proceedings of CRYPTO 2003, DOI 10.1007/978-3-540-45146-4_24, August 2003.

[SLOTH] Bhargavan, K. and G. Leurent, "Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH", Network and Distributed System Security Symposium (NDSS 2016), DOI 10.14722/ndss.2016.23418, February 2016.

[SSL2] Hickman, K., "The SSL Protocol", February 1995.

[TIMING] Boneh, D. and D. Brumley, "Remote Timing Attacks Are Practical", USENIX Security Symposium, August 2003.

[TLS13-TRACES] Thomson, M., "Example Handshake Traces for TLS 1.3", Work in Progress[1], draft-ietf-tls-tls13-vectors-06, July 2018.

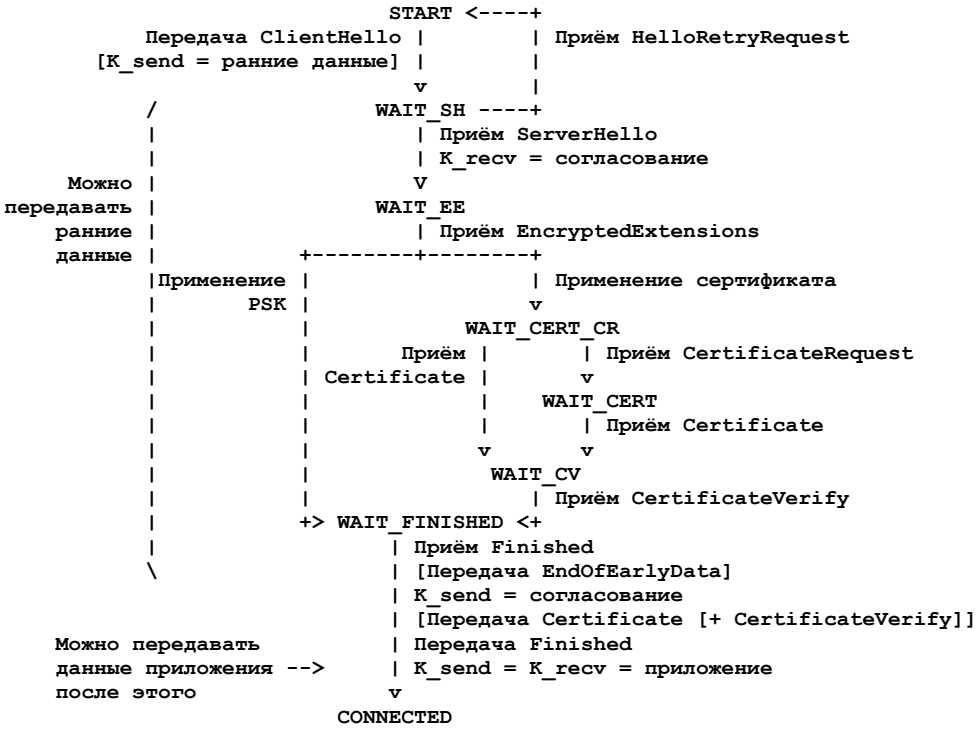
[X501] ITU-T, "Information Technology - Open Systems Interconnection - The Directory: Models", ITU-T X.501, October 2016, <https://www.itu.int/rec/T-REC-X.501/en>.

Приложение A. Конечные автоматы

В этом приложении дана сводка допустимых переходов при согласовании для клиента и сервера. Состояния (заглавные буквы, например START) не имеют формального значения и приведены для упрощения восприятия. Действия, происходящие лишь при определённых обстоятельствах, указаны в квадратных скобках []. Обозначение K_{send,recv} = foo указывает установку для ключа приёма/передачи указанного значения.

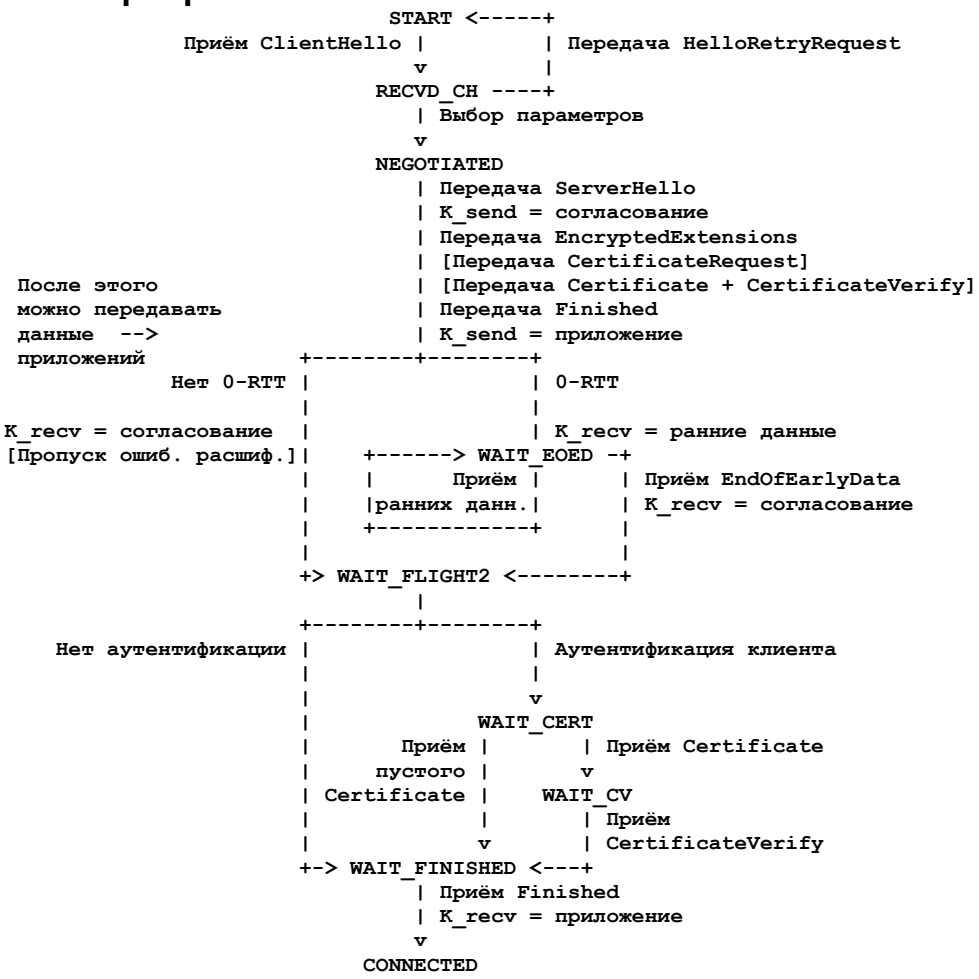
[1]Опубликовано в RFC 8448. *Прим. перев.*

A.1. Клиент



Отметим, что при показанных выше переходах клиент может передавать сигналы, выведенные из сообщений после ServerHello в открытом виде или с ключами ранних данных. Если клиенту нужно передать такой сигнал, ему **следует** сначала обновить ключи согласования, если это возможно.

A.2. Сервер



Приложение B. Структуры данных и константы протокола

В этом приложении представлены нормативные типы протоколов и определения констант. Значения с пометкой _RESERVED использовались в прежних версиях TLS и приведены здесь для полноты. Реализациям TLS 1.3 **недопустимо** передавать такие значения, но они могут получать их от старых реализаций TLS.

B.1. Уровень Record

```
enum {
    invalid(0),
    change_cipher_spec(20),
    alert(21),
    handshake(22),
    application_data(23),
    heartbeat(24),  /* RFC 6520 */
    (255)
} ContentType;
struct {
    ContentType type;
    ProtocolVersion legacy_record_version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
struct {
    opaque content[TLSPlaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;
struct {
    ContentType opaque_type = application_data;      /* 23 */
    ProtocolVersion legacy_record_version = 0x0303;  /* TLS v1.2 */
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

B.2. Сообщения Alert

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
    unexpected_message(10),
    bad_record_mac(20),
    decryption_failed_RESERVED(21),
    record_overflow(22),
    decompression_failure_RESERVED(30),
    handshake_failure(40),
    no_certificate_RESERVED(41),
    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),
    export_restriction_RESERVED(60),
    protocol_version(70),
    insufficient_security(71),
    internal_error(80),
    inappropriate_fallback(86),
    user_canceled(90),
    no_renegotiation_RESERVED(100),
    missing_extension(109),
    unsupported_extension(110),
    certificate_unobtainable_RESERVED(111),
    unrecognized_name(112),
    bad_certificate_status_response(113),
    bad_certificate_hash_value_RESERVED(114),
    unknown_psk_identity(115),
    certificate_required(116),
    no_application_protocol(120),
    (255)
} AlertDescription;

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

B.3. Протокол Handshake

```
enum {
    hello_request_RESERVED(0),
    client_hello(1),
    server_hello(2),
    hello_verify_request_RESERVED(3),
    new_session_ticket(4),
    end_of_early_data(5),
    hello_retry_request_RESERVED(6),
    encrypted_extensions(8),
    certificate(11),
```

```
        server_key_exchange_RESERVED(12),
        certificate_request(13),
        server_hello_done_RESERVED(14),
        certificate_verify(15),
        client_key_exchange_RESERVED(16),
        finished(20),
        certificate_url_RESERVED(21),
        certificate_status_RESERVED(22),
        supplemental_data_RESERVED(23),
        key_update(24),
        message_hash(254),
        (255)
    } HandshakeType;

    struct {
        HandshakeType msg_type;    /* тип согласования */
        uint24 length;             /* число байтов в сообщении */
        select (Handshake.msg_type) {
            case client_hello:          ClientHello;
            case server_hello:          ServerHello;
            case end_of_early_data:     EndOfEarlyData;
            case encrypted_extensions:  EncryptedExtensions;
            case certificate_request:   CertificateRequest;
            case certificate:           Certificate;
            case certificate_verify:    CertificateVerify;
            case finished:              Finished;
            case new_session_ticket:    NewSessionTicket;
            case key_update:            KeyUpdate;
        };
    } Handshake;
```

B.3.1. Сообщения обмена ключами

```
uint16 ProtocolVersion;
opaque Random[32];

uint8 CipherSuite[2];    /* Селектор шифра */

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

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

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

enum {
    server_name(0),                             /* RFC 6066 */
    max_fragment_length(1),                     /* RFC 6066 */
    status_request(5),                          /* RFC 6066 */
    supported_groups(10),                       /* RFC 8422, 7919 */
    signature_algorithms(13),                   /* RFC 8446 */
    use_srtp(14),                               /* RFC 5764 */
    heartbeat(15),                              /* RFC 6520 */
    application_layer_protocol_negotiation(16), /* RFC 7301 */
    signed_certificate_timestamp(18),           /* RFC 6962 */
    client_certificate_type(19),                /* RFC 7250 */
    server_certificate_type(20),                /* RFC 7250 */
    padding(21),                                /* RFC 7685 */
    RESERVED(40),                               /* Применяется, но не выделено*/
    pre_shared_key(41),                         /* RFC 8446 */
    early_data(42),                             /* RFC 8446 */
    supported_versions(43),                     /* RFC 8446 */
    cookie(44),                                 /* RFC 8446 */
    psk_key_exchange_modes(45),                 /* RFC 8446 */
    RESERVED(46),                               /* Применяется, но не выделено*/
    certificate_authorities(47),                /* RFC 8446 */
    oid_filters(48),                            /* RFC 8446 */
    post_handshake_auth(49),                    /* RFC 8446 */
    signature_algorithms_cert(50),              /* RFC 8446 */
    key_share(51),                              /* RFC 8446 */
    (65535)
```

```
} ExtensionType;

struct {
    NamedGroup group;
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;

struct {
    KeyShareEntry client_shares<0..2^16-1>;
} KeyShareClientHello;

struct {
    NamedGroup selected_group;
} KeyShareHelloRetryRequest;

struct {
    KeyShareEntry server_share;
} KeyShareServerHello;

struct {
    uint8 legacy_form = 4;
    opaque X[coordinate_length];
    opaque Y[coordinate_length];
} UncompressedPointRepresentation;

enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeMode;

struct {
    PskKeyExchangeMode ke_modes<1..255>;
} PskKeyExchangeModes;

struct {} Empty;

struct {
    select (Handshake.msg_type) {
        case new_session_ticket:   uint32 max_early_data_size;
        case client_hello:         Empty;
        case encrypted_extensions: Empty;
    };
} EarlyDataIndication;

struct {
    opaque identity<1..2^16-1>;
    uint32 obfuscated_ticket_age;
} PskIdentity;

opaque PskBinderEntry<32..255>;

struct {
    PskIdentity identities<7..2^16-1>;
    PskBinderEntry binders<33..2^16-1>;
} OfferedPsks;

struct {
    select (Handshake.msg_type) {
        case client_hello: OfferedPsks;
        case server_hello: uint16 selected_identity;
    };
} PreSharedKeyExtension;
```

B.3.1.1. Расширение Version

```
struct {
    select (Handshake.msg_type) {
        case client_hello:
             ProtocolVersion versions<2..254>;

        case server_hello: /* и HelloRetryRequest */
             ProtocolVersion selected_version;
    };
} SupportedVersions;
```

B.3.1.2. Расширение Cookie

```
struct {
    opaque cookie<1..2^16-1>;
} Cookie;
```

B.3.1.3. Расширение Signature Algorithm

```
enum {
    /* Алгоритмы RSASSA-PKCS1-v1_5 */
    rsa_pkcs1_sha256(0x0401),
    rsa_pkcs1_sha384(0x0501),
    rsa_pkcs1_sha512(0x0601),

    /* Алгоритмы ECDSA */
    ecdsa_secp256r1_sha256(0x0403),
    ecdsa_secp384r1_sha384(0x0503),
```

```
        ecdsa_secp521r1_sha512(0x0603),
        /* Алгоритмы RSASSA-PSS с OID открытого ключа rsaEncryption */
        rsa_pss_rsae_sha256(0x0804),
        rsa_pss_rsae_sha384(0x0805),
        rsa_pss_rsae_sha512(0x0806),

        /* EdDSA algorithms */
        ed25519(0x0807),
        ed448(0x0808),

        /* Алгоритмы RSASSA-PSS с OID открытого ключа RSASSA-PSS */
        rsa_pss_pss_sha256(0x0809),
        rsa_pss_pss_sha384(0x080a),
        rsa_pss_pss_sha512(0x080b),

        /* Устаревшие алгоритмы */
        rsa_pkcs1_sha1(0x0201),
        ecdsa_sha1(0x0203),

        /* Резервные коды */
        obsolete_RESERVED(0x0000..0x0200),
        dsa_sha1_RESERVED(0x0202),
        obsolete_RESERVED(0x0204..0x0400),
        dsa_sha256_RESERVED(0x0402),
        obsolete_RESERVED(0x0404..0x0500),
        dsa_sha384_RESERVED(0x0502),
        obsolete_RESERVED(0x0504..0x0600),
        dsa_sha512_RESERVED(0x0602),
        obsolete_RESERVED(0x0604..0x06FF),
        private_use(0xFE00..0xFFFF),
        (0xFFFF)
    } SignatureScheme;

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

B.3.1.4. Расширение Supported Groups

```
    enum {
        unallocated_RESERVED(0x0000),

        /* Группы эллиптических кривых (ECDHE) */
        obsolete_RESERVED(0x0001..0x0016),
        secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),
        obsolete_RESERVED(0x001A..0x001C),
        x25519(0x001D), x448(0x001E),

        /* Группы конечных полей (DHE) */
        ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),
        ffdhe6144(0x0103), ffdhe8192(0x0104),

        /* Резервные коды */
        ffdhe_private_use(0x01FC..0x01FF),
        ecdhe_private_use(0xFE00..0xFEFF),
        obsolete_RESERVED(0xFF01..0xFF02),
        (0xFFFF)
    } NamedGroup;

    struct {
        NamedGroup named_group_list<2..2^16-1>;
    } NamedGroupList;
```

Значения obsolete_RESERVED использовались в прежних версиях TLS и их **недопустимо** предлагать или согласовывать реализациям TLS 1.3. Устаревшие кривые имеют известные или предсказанные недостатки (слабость) или применялись очень редко, в некоторых случаях лишь по причине непреднамеренных проблем в конфигурации серверов. Они больше не считаются подходящими для общего пользования и их следует считать потенциально небезопасными. Указанного здесь набора кривых достаточно для обеспечения совместимости со всеми развёрнутыми в настоящее время и корректно настроенными реализациями TLS.

B.3.2. Сообщения с параметрами сервера

```
    opaque DistinguishedName<1..2^16-1>;

    struct {
        DistinguishedName authorities<3..2^16-1>;
    } CertificateAuthoritiesExtension;

    struct {
        opaque certificate_extension_oid<1..2^8-1>;
        opaque certificate_extension_values<0..2^16-1>;
    } OIDFilter;

    struct {
        OIDFilter filters<0..2^16-1>;
    } OIDFilterExtension;
```

```
struct {} PostHandshakeAuth;

struct {
    Extension extensions<0..2^16-1>;
} EncryptedExtensions;

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

B.3.3. Аутентификационные сообщения

```
enum {
    X509(0),
    OpenPGP_RESERVED(1),
    RawPublicKey(2),
    (255)
} CertificateType;

struct {
    select (certificate_type) {
        case RawPublicKey:
          /* Из RFC 7250 ASN.1_subjectPublicKeyInfo */
          opaque ASN1_subjectPublicKeyInfo<1..2^24-1>;

        case X509:
          opaque cert_data<1..2^24-1>;
    };
    Extension extensions<0..2^16-1>;
} CertificateEntry;

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

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

struct {
    opaque verify_data[Hash.length];
} Finished;
```

B.3.4. Создание квитанции

```
struct {
    uint32 ticket_lifetime;
    uint32 ticket_age_add;
    opaque ticket_nonce<0..255>;
    opaque ticket<1..2^16-1>;
    Extension extensions<0..2^16-2>;
} NewSessionTicket;
```

B.3.5. Обновление ключей

```
struct {} EndOfEarlyData;

enum {
    update_not_requested(0), update_requested(1), (255)
} KeyUpdateRequest;

struct {
    KeyUpdateRequest request_update;
} KeyUpdate;
```

B.4. Шифры

Симметричный шифр определяет пару алгоритмов AEAD и хэширования для использования с HKDF. Имя шифра следует соглашению

```
CipherSuite TLS_AEAD_HASH = VALUE;
```

Компонента	*Содержимое*
TLS	Строка "TLS"
AEAD	Алгоритм AEAD, используемый для защиты записи
HASH	Алгоритм хэширования, используемый с HKDF
VALUE	Двухбайтовый идентификатор шифра

Данная спецификация определяет перечисленные ниже шифры для использования с TLS 1.3.

Описание	*Значение*
TLS_AES_128_GCM_SHA256	{0x13,0x01}
TLS_AES_256_GCM_SHA384	{0x13,0x02}
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}
TLS_AES_128_CCM_SHA256	{0x13,0x04}
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}

Соответствующие алгоритмы AEAD_AES_128_GCM, AEAD_AES_256_GCM и AEAD_AES_128_CCM определены в [RFC5116], AEAD_CHACHA20_POLY1305 - в [RFC8439], AEAD_AES_128_CCM_8 - в [RFC6655]. Алгоритмы хэширования определены в [SHS].

Хотя TLS 1.3 использует то же пространство шифров, что и предыдущие версии TLS, шифронаборы TLS 1.3 определены путём указания только симметричных шифров и не могут применяться для TLS 1.2. Аналогично, шифры для TLS 1.2 и ниже не могут использоваться в TLS 1.3.

Новые значения, выделенные IANA для шифров, описаны в разделе 11. Взаимодействие с IANA.

Приложение C. Примечания для разработчиков

Протокол TLS не может предотвратить многие распространённые ошибки защиты. В этом приложении даны некоторые рекомендации, способные помочь разработчикам. В [TLS13-TRACES] представлены тестовые векторы для согласования TLS 1.3.

C.1. Генерация случайных чисел и затравки

TLS требует криптографически защищённого генератора псевдослучайных чисел (CSPRNG[1]). В большинстве случаев операционная система предоставляет подходящие средства (например /dev/urandom), которые следует использовать в отсутствие других (например, производительность) соображений. **Рекомендуется** применять имеющуюся реализацию CSPRNG, а не создавать новую. Уже доступно много адекватных криптографических библиотек с приемлемыми условиями лицензирования. Если они не подходят, следует использовать рекомендации [RFC4086] по созданию псевдослучайных значений.

TLS использует случайные значения (1) в открытых полях протокола (например, Random) сообщений ClientHello и ServerHello, а также (2) для генерации ключевого материала. При корректной работе CSPRNG не возникает проблем безопасности, поскольку нет возможности определить состояние CSPRNG по его выходным данным. Однако при повреждённом CSPRNG у атакующего может появиться возможность использовать общедоступный вывод для определения внутреннего состояния CSPRNG и, следовательно, предсказания ключевого материала, как указано в [CHECKOWAY]. Реализации могут обеспечить дополнительную защиту от этой формы атак, используя разные CSPRNG для генерации общедоступных и приватных значений.

C.2. Сертификаты и проверка подлинности

Реализации отвечают за проверку целостности сертификатов и в общем случае должны поддерживать сообщения с отзывом сертификатов. При отсутствии конкретного указания в профиле приложения сертификаты следует проверять всегда для обеспечения надлежащей подписи доверенного удостоверяющего центра (CA). Выбор и добавление привязок доверия следует выполнять очень осторожно. Пользователям следует иметь возможность видеть информацию о сертификате и привязке доверия. Приложениям **следует** также исполнять требования к минимальному и максимальному размеру ключей. Например, путь сертификации, содержащий ключи или подписи слабее 2048-битовых RSA или 224-битовых ECDSA, не применим для защищённых приложений.

C.3. Подводные камни реализации

Опыт реализации показал, что прежние спецификации было сложно понять и это служило причиной несовместимости и возникновения проблем защиты. Многие из таких вопросов были уточнены в этом документе, а в данном приложении приведён короткий список наиболее важных вопросов, требующих внимания разработчиков.

Протокол TLS

- Корректно ли обрабатываются согласующие сообщения, фрагментированные в несколько записей TLS (5.1. Уровень Record)? Корректно ли обрабатываются важные случаи, такие как сообщения ClientHello, разбитые на несколько мелких фрагментов? Фрагментируются ли сообщения, размер которых превышает максимальный размер фрагмента (в частности, сообщения Certificate и CertificateRequest могут достаточно велики и требовать фрагментирования)?
- Игнорируется ли номер версии уровня TLS во всех незашифрованных записях TLS (Приложение D)?
- Имеется ли уверенность в полном удалении SSL, RC4, шифров EXPORT и MD5 (через расширение signature_algorithms) из всех возможных конфигураций, которые поддерживают TLS 1.3 и выше, а также в корректном отказе при попытке использования этих устаревших возможностей (Приложение D)?
- Обрабатываются ли корректно расширения TLS в ClientHello, включая неизвестные расширения?
- При недоступности запрошенного сервером сертификата отправляется ли корректно пустое сообщение Certificate (4.4.2. Сообщение Certificate)?
- При обработке данных, расшифрованных AEAD-Decrypt, и сканировании с конца в поиске ContentType, предотвращается ли сканирование после начала открытых данных, если узел ошибочно заполнил открытые данные только нулями?
- Игнорируются ли корректно неопознанные шифры (4.1.2. Клиентское сообщение Hello), расширения при согласовании (4.2. Расширения), именованные группы (4.2.7. Поддерживаемые группы), общие ключи (4.2.8. Совместное использование ключа), поддерживаемые версии (4.2.1. Поддерживаемые версии) и алгоритмы подписи (4.2.3. Алгоритмы подписи) в ClientHello?
- Передаёт ли реализация сервера сообщения HelloRetryRequest клиентам, которые поддерживают совместимую группу (EC)DHE, но не прогнозируют её в расширении key_share? Обрабатывает ли реализация клиента корректно сообщения HelloRetryRequest от сервера?

[1]Cryptographically secure pseudorandom number generator.

- Какие меры предусмотрены для предотвращения timing-атак [TIMING]?
- При обмене ключами Diffie-Hellman обрабатываются ли корректно начальные 0 в согласованном ключе (7.4.1. Конечное поле Diffie-Hellman)?
- Проверяет ли клиент TLS приемлемость переданных сервером параметров Diffie-Hellman (4.2.8.1. Параметры Diffie-Hellman)?
- Используется ли сильный и, наиболее важно, правильно подобранный генератор случайных чисел (Приложение C.1) при создании приватных значений Diffie-Hellman, параметра ECDSA "k" и других важных для безопасности значений? Реализациям **рекомендуется** применять «детерминированный ECDSA», как указано в [RFC6979].
- Используются ли дополненные 0 до размера группы открытые ключи Diffie-Hellman и общие секреты (параграфы 4.2.8.1. Параметры Diffie-Hellman и 7.4.1. Конечное поле Diffie-Hellman)?
- Проверяются ли подписи после их создания для предотвращения утечки ключа RSA-CRT [FW15]?

C.4. Предотвращение отслеживания клиентов

Клиентам **не следует** применять одну квитанцию для нескольких соединений. Повторное использование квитанции позволяет пассивному наблюдателю сопоставлять разные соединения. Серверам, выпускающим квитанции, **следует** предлагать число квитанций не меньше числа соединений, которые может использовать клиент. Например, браузер HTTP/1.1 [RFC7230] может создавать 6 соединений с сервером. Серверам **следует** выпускать новую квитанцию для каждого соединения. Это позволит клиенту применять в каждом соединении свою квитанцию.

C.5. Неаутентифицированные операции

Прежние версии TLS предлагали явно неаутентифицированные шифры на основе анонимного согласования Diffie-Hellman. Эти режимы были исключены в TLS 1.3. Однако остаётся возможность согласовать параметры, которые не обеспечивают проверяемой аутентификации сервера несколькими способами, включая:

- необработанные (raw) открытые ключи [RFC7250];
- использование открытого ключа из сертификата без проверки его содержимого или цепочки сертификации.

Любой из этих методов уязвим для MITM-атак[1] и поэтому небезопасен для общего пользования. Однако можно связать такие соединения с внешним механизмом аутентификации с помощью проверки по отдельному каналу (out-of-band) открытого ключа сервера, доверия при первом применении или такого механизма, как привязка канала (хотя привязка, описанная в [RFC5929], не определена для TLS 1.3). Если такие механизмы не используются, соединение не будет защищено от MITM-атак и приложениям **недопустимо** использовать TLS таким способом при отсутствии явной конфигурации или конкретного профиля приложения.

Приложение D. Совместимость с прежними версиями

Протокол TLS обеспечивает встроенный механизм согласования версии между конечными точками, которые могут поддерживать разные версии TLS.

TLS 1.x и SSL 3.0 применяют несовместимые сообщения ClientHello. Серверы могут обслуживать клиентов, пытающихся применять будущие версии TLS, пока формат ClientHello остаётся совместимым и есть хотя бы одна версия, поддерживаемая клиентом и сервером.

Прежние версии TLS использовали номер версии уровня записи (TLSPlaintext.legacy_record_version и TLSCiphertext.legacy_record_version) для разных целей. Начиная с TLS 1.3, это поле отменено. Значение TLSPlaintext.legacy_record_version **должно** игнорироваться реализациями. TLSCiphertext.legacy_record_version включается в дополнительные данные для снятия защиты, но **может** игнорироваться в иных случаях или **может** проверяться на совпадение с фиксированной константой. Согласование версий выполняется с использованием лишь версии согласования (ClientHello.legacy_version и ServerHello.legacy_version), а также расширения supported_versions в ClientHello, HelloRetryRequest и ServerHello). Для максимальной совместимости со старыми конечными точками реализациям, согласующим применение TLS 1.0-1.2, **следует** устанавливать для номера версии уровня записи согласованный номер версии в ServerHello и последующих записях.

Для максимальной совместимости с прежним нестандартным поведением и некорректно настроенными реализациями, **следует** поддерживать проверку путей сертификации на основе ожиданий даже при обработке согласования с прежними версиями TLS (параграф 4.4.2.2).

Версии TLS 1.2 и ранее поддерживают расширение Extended Master Secret [RFC7627], которое переводило большие части стенограммы согласования в основной секрет. Поскольку в TLS 1.3 стенограмма всегда хэшируется вплоть до серверного сообщения Finished, реализациям, поддерживающим TLS 1.3 и более ранние версии, **следует** указывать использование Extended Master Secret в своих API всякий раз при использовании TLS 1.3.

D.1. Согласование со старым сервером

Клиент TLS 1.3 при согласовании с сервером, не поддерживающим TLS 1.3, будет передавать обычное сообщение TLS 1.3 ClientHello с 0x0303 (TLS 1.2) в поле ClientHello.legacy_version, но с корректными версиями в расширении supported_versions. Сервер, не поддерживающий TLS 1.3, будет отвечать сообщением ServerHello с номером старой версии. Если клиент согласен использовать эту версию, согласование будет продолжаться в соответствии с выбранным протоколом. Клиенту, применяющему квитанцию для восстановления, **следует** инициировать соединение, используя согласованную ранее версию.

[1] Man-in-the-middle - перехват и изменение пакетов с участием человека.

Отметим, что данные 0-RTT не совместимы со старыми серверами и их **не следует** передавать при отсутствии информации о поддержке сервером TLS 1.3 (Приложение D.3).

Если выбранная сервером версия не поддерживается клиентом (или неприемлема), клиент **должен** прервать согласование с сигналом protocol_version.

Известно, что некоторые устаревшие реализации серверов не реализуют спецификацию TLS должным образом и могут разрывать соединение в ответ на неизвестные им расширения TLS. Взаимодействие с такими серверами сложно и выходит за рамки этого документа. Для согласования совместимого со старыми версиями соединения может потребоваться несколько попыток, однако это уязвимо для атак на понижение версии и **не рекомендуется**.

D.2. Согласование со старым клиентом

Сервер TLS может получать ClientHello с номером версии, ниже поддерживаемой сервером максимальной версии. При наличии расширения supported_versions сервер **должен** выполнить согласование с использованием этого расширения, как описано в параграфе 4.2.1. Поддерживаемые версии. Если расширения supported_versions нет, сервер должен согласовать меньшее из ClientHello.legacy_version и TLS 1.2. Например, если сервер поддерживает TLS 1.0, 1.1 и 1.2, а legacy_version указывает TLS 1.0, сервер будет продолжать с TLS 1.0 ServerHello. Если расширение supported_versions отсутствует и сервер поддерживает лишь версии больше ClientHello.legacy_version, он **должен** прервать согласование с сигналом protocol_version.

Отметим, что ранние версии TLS не всегда чётко указывают номер версии уровня записи (TLSPlaintext.legacy_record_version). Серверы будут получать разные версии TLS 1.x в этом поле, но они **должны** игнорироваться.

D.3. Совместимость 0-RTT с прежними версиями

Данные 0-RTT несовместимы со старыми версиями. Старый сервер будет отвечать на ClientHello старым ServerHello, но он не будет корректно пропускать данные 0-RTT и согласование завершится отказом. Это может вызывать проблемы при попытке клиента использовать 0-RTT, особенно в среде с множеством серверов. Например, система может поэтапно внедрять TLS 1.3 и некоторые серверы будут поддерживать TLS 1.3, а другие - TLS 1.2 или для серверов TLS 1.3 версия может быть снижена до TLS 1.2.

Клиент, пытающийся передать данные 0-RTT, **должен** разорвать соединение, если он получит ServerHello с TLS 1.2 или ниже. Клиент может затем повторить попытку соединения с запретом 0-RTT. Для предотвращения атак на понижение версии клиенту **не следует** отменять TLS 1.3, достаточно отключить 0-RTT.

Для предотвращения таких ошибок системам с множеством серверов **следует** обеспечить однородное и стабильное развёртывание TLS 1.3 без 0-RTT и лишь после этого разрешать 0-RTT.

D.4. Режим совместимости с промежуточными устройствами

Исследования [Ben17a] [Ben17b] [Res17a] [Res17b] показали, что значительная часть промежуточных устройств некорректно ведёт себя при согласовании клиентом и сервером TLS 1.3. Реализации могут повысить шансы согласования через такие устройства, сделав согласование TLS 1.3 более похожим на TLS 1.2, как указано ниже.

- Клиент всегда указывает непустой идентификатор сессии в ClientHello, как описано в параграфе 4.1.2. Клиентское сообщение Hello (legacy_session_id).
- Если не используются ранние данные, клиент передаёт фиктивную запись change_cipher_spec (см. третий абзац в разделе 5) непосредственно перед своей второй отправкой (перед вторым ClientHello или перед шифрованным согласующим сообщением). При использовании ранних данных фиктивная запись передаётся сразу после первого ClientHello.
- Сервер передаёт фиктивную запись change_cipher_spec сразу после первого согласующего сообщения (ServerHello или HelloRetryRequest).

В совокупности эти изменения делают согласование TLS 1.3 похожим на восстановление сессии TLS 1.2, что повышает шансы успешного соединения через промежуточные устройства. Такой «режим совместимости» согласуется частично - клиент может указать или не указать идентификатор сессии, а сервер возвращает его (эхо). Любая сторона может передать change_cipher_spec в любой момент согласования, но если клиент передал непустой идентификатор сессии, сервер **должен** передать change_cipher_spec, как указано здесь.

D.5. Ограничения защиты, связанные с совместимостью

Реализации, согласующий более старые версии TLS, **следует** предпочитать «прямую секретность» (forward secret) и шифры AEAD, когда это доступно.

Безопасность шифров RC4 считается недостаточной по причинам, указанным в [RFC7465]. Реализациям **недопустимо** предлагать или согласовывать шифры RC4 для любой версии TLS по любым причинам.

Старые версии TLS допускали применение шифров с очень ограниченной стойкостью. Шифры со стойкостью менее 112 битов **недопустимо** предлагать или согласовывать для любой версии TLS по любым причинам.

Безопасность SSL 3.0 [RFC6101] считается недостаточной по причинам, указанным в [RFC7568], и **недопустимо** согласовывать эту версию по любым причинам.

Безопасность SSL 2.0 [SSL2] считается недостаточной по причинам, указанным в [RFC6176], и **недопустимо** согласовывать эту версию по любым причинам.

Реализации **недопустимо** передавать CLIENT-HELLO, совместимые с SSL 2.0. Реализации **недопустимо** согласовывать 1.3 или выше с использованием совместимых с SSL 2.0 сообщений CLIENT-HELLO. Реализации **не рекомендуется** воспринимать совместимые с SSL 2.0 CLIENT-HELLO для согласования ранних версий TLS.

Реализациям **недопустимо** передавать ClientHello.legacy_version или ServerHello.legacy_version 0x0300 или меньше. Любая конечная точка, получившая сообщение с ClientHello.legacy_version или ServerHello.legacy_version 0x0300, **должна** прервать согласование с сигналом protocol_version.

Реализациям **недопустимо** передавать какие-либо записи с версией меньше 0x0300. Реализациям **не следует** воспринимать какие-либо записи с версией меньше 0x0300 (они могут сделать это непреднамеренно при полном игнорировании версии записи).

Реализациям **недопустимо** использовать расширение Truncated HMAC, определённое в разделе 7 [RFC6066], поскольку оно не применимо к алгоритмам AEAD и показало свою небезопасность в некоторых ситуациях.

Приложение E. Обзор защитных свойств

Полный анализ защищенности TLS выходит за рамки этого документа. В этом приложении представлено описание желаемых свойств, а также ссылки на более подробные описания в исследовательской литературе, где приведены более формализованные определения.

Отдельно рассматриваются свойства согласования и уровня записи.

E.1. Согласование

Согласование TLS является протоколом аутентифицированного обмена ключами (Authenticated Key Exchange или AKE), который предназначен для односторонней (сервер) и взаимной (клиент и сервер) проверки подлинности. По завершении согласования каждая сторона выводит своё представление указанных ниже значений.

- Набор «сеансовых ключей» (различные секреты, выведенные из первичного), из которого будет выводиться набор рабочих ключей.
- Набор криптографических параметров (алгоритмы и т. п.).
- Отождествления взаимодействующих сторон.

Предполагается, что атакующий является активным и имеет полный контроль над сетью, используемой для взаимодействия сторон [RFC3552]. Даже при таких условиях согласование должно обеспечивать приведённые ниже свойства. Отметим, что эти свойства не обязательно являются независимыми, но отражают потребности пользователей протокола.

Создание одинаковых сеансовых ключей
Согласование должно давать одинаковые наборы сеансовых ключей на обеих сторонах при условии полного завершения согласования на каждой из сторон ([CK01], определение 1, часть 1).

Секретность сеансовых ключей
Общие сеансовые ключи следует знать только взаимодействующим сторонам, но не атакующему ([CK01], определение 1, часть 2). Отметим, что при односторонней аутентификации соединения атакующий может организовать свой сеансовый ключ с сервером, но этот ключ будет отличаться от созданного клиентом.

Проверка подлинности партнёра
Представление клиента об идентификации партнёра должно отражать отождествление сервера. Если клиент аутентифицирован, представление сервера о его идентификации должно совпадать с отождествлением клиента.

Уникальность сеансовых ключей
Любым двум разным согласованиям следует давать на выходе разные, не связанные сеансовые ключи. Отдельные сеансовые ключи, создаваемые при согласовании, также должны быть разными и независимыми.

Защита от понижения версии
Криптографические параметры должны быть одинаковыми на обеих сторонах и им следует быть такими же, какие были бы согласованы в отсутствие атаки ([BBFGKZ16], определения 8 и 9).

Секрет для долгосрочных ключей
Если долгосрочный ключевой материал (ключи подписи в режиме аутентификации по сертификатам или внешний/восстановительный PSK в режиме PSK с (EC)DHE) скомпрометированы после завершения согласования, это не снижает защиту сеансового ключа (см. [DOW92]), пока сам ключ не уничтожен. Свойство forward secrecy не выполняется, когда PSK применяется в режиме psk_ke PskKeyExchangeMode.

Устойчивость к компрометации ключей (KCI[1])
Во взаимно аутентифицированном соединении с сертификатами компрометация долгосрочного секрета на одной стороне не должна прерывать аутентификацию его партнёра в данном соединении (см. [HGFS15]). Например, если скомпрометирован ключ подписи клиента, это не должно давать возможности обмана произвольного сервера при последующих согласованиях этого клиента.

Защита отождествлений конечных точек
Отождествлению сервера (сертификат) следует быть защищённым от пассивных атак. Отождествлению клиента следует быть защищённым от пассивных и активных атак.

Неформально, основанные на подписи режимы TLS 1.3 обеспечивают создание уникального, секретного общего ключа с помощью обмена ключами (EC)DHE, аутентифицированного подписью сервера для стенограммы согласования и привязанного к отождествлению сервера с помощью MAC. Если клиент аутентифицирован по сертификату, он тоже подписывает стенограмму согласования и привязывает MAC к обоим отождествлениям. В [SIGMA] приведено устройство и анализ этого типа протокола обмена ключами. Если для каждого соединения применяются свежие ключи (EC)DHE, ключи на выходе будут иметь надёжную защиту (forward secret).

Внешний и восстановительный PSK загружаются из долгосрочного общего секрета в уникальный для соединения набор краткосрочных сеансовых ключей. Этот секрет может быть организован в предшествующем соединении. При использовании (EC)DHE для создания PSK сеансовые ключи будут также надёжным секретом (forward secret). Восстановительный PSK устроен так, что первичный секрет восстановления, рассчитанный в соединении N и требуемый для организации соединения N+1, отделен от ключей трафика, используемых соединением N, что обеспечивает надёжную секретность между соединениями. В дополнение к этому при создании множества квитанций для одного соединения они связываются с разными ключами, поэтому компрометация PSK, связанного с одной

[1]Key Compromise Impersonation.

квитанцией, не будет угрожать соединениям, использующим PSK, связанные с другими квитанциями. Это свойство наиболее интересно при хранении квитанций в базе данных (могут быть удалены) вместо их самошифровки.

Значение привязки PSK формирует связь PSK с текущим согласованием, а также между сессией, создавшей PSK, и текущей сессией. Эта привязка транзитивно включает копию исходного согласования, поскольку эта копия преобразуется в значения, которые создают первичный секрет восстановления. Это требует от KDF, используемой для создания первичного секрета, и функции MAC, используемой для расчёта привязки, устойчивости к конфликтам (Приложение E.1.1).

> Примечание. Привязка не охватывает значений привязок из других PSK, хотя они включаются в MAC сообщения Finished.

TLS в настоящее время не позволяет серверу передавать сообщение certificate_request при согласовании, не основанном на сертификатах (например, PSK). Если в будущем это ограничения будет смягчено, подпись клиента не будет напрямую охватывать сертификат сервера. Однако при организации PSK с помощью NewSessionTicket подпись клиента будет транзитивно охватывать сертификат сервера за счёт привязки PSK. В [PSK-FINISHED] описана конкретная атака на конструкции без привязки к сертификату сервера (см. также [Kraw16]). Небезопасно применять аутентификацию клиентов на основе сертификатов, когда клиент может использовать пару «PSK - идентификатор ключа» совместно двумя разными конечными точками. Реализациям **недопустимо** комбинировать внешние PSK с аутентификацией на основе сертификата клиента или сервера, пока это не согласовано тем или иным расширением.

При использовании экспортёра тот создаёт уникальные и секретные значения (поскольку они генерируются из уникального сеансового ключа). Экспортёры, рассчитанные с разными метками и контекстом, независимы вычислительно, поэтому нет возможности рассчитать одно из другого или получить сеансовый секрет из экспортированного значения.

> Примечание. Экспортёры могут создавать значения произвольного размера. Если экспортёр применяется в качестве привязки канала, экспортируемое значение **должно** быть достаточно большим для обеспечения устойчивости к конфликтам. Экспортёры в TLS 1.3 выводятся из того же контекста согласования, что и ключи для ранних данных и трафика приложений, поэтому имеют схожие свойства безопасности. Отметим, что они не включают сертификат клиента. В будущих приложениях, желающих привязать сертификат клиента, может потребоваться определять новые экспортёры, которые будут включать полную стенограмму согласования.

Для всех режимов согласования Finished MAC (и подпись при её наличии) предотвращает атаки на понижение версии. В дополнение к этому использование определённых битов в случайных nonce, как описано в параграфе 4.1.3. Серверное сообщение Hello, позволяет обнаружить понижение до одной из предыдущих версий TLS. В работе [BBFGKZ16] рассмотрены дополнительные детали TLS 1.3 и понижения версии.

Как только клиент и сервер обменялись достаточно большим объёмом информации для создания общих ключей, оставшаяся часть согласования шифруется для защиты от пассивных атак, даже если рассчитанные общие ключи не аутентифицированы. Поскольку сервер аутентифицируется раньше клиента, клиент может быть уверен, что при аутентификации раскрывает своё отождествление подлинному серверу. Отметим, что реализации должны использовать представленный механизм дополнения записей в процессе согласования для предотвращения утечки информации об идентификаторах по их размеру. Предложенные клиентом отождествления PSK не шифруются и не выбираются сервером.

E.1.1. Вывод ключей и HKDF

Вывод ключей в TLS 1.3 использует функцию HKDF, как определено в [RFC5869], и две её компоненты - HKDF-Extract и HKDF-Expand. Полное обоснование конструкции HKDF приведено в [Kraw10], а обоснование способа применения в TLS 1.3 - в [KW16]. В этом документе каждое применение HKDF-Extract сопровождается одним или несколькими вызовами HKDF-Expand. Этот порядок следует соблюдать всегда (включая будущие пересмотры этого документа), в частности **не следует** использовать вывод HKDF-Extract в качестве входных данных при другом вызове HKDF-Extract без применения HKDF-Expand между ними. Допускается многократное применение HKDF-Expand к одним и тем же входным данным, если их можно различить по ключу или меткам.

Отметим, что HKDF-Expand реализует псевдослучайную функцию со входными и выходными значениями переменного размера. В некоторых вариантах применения HKDF в этом документе (например, для генерации экспортёров и resumption_master_secret) требуется обеспечить стойкость HKDF-Expand к конфликтам, а именно невозможность найти два разных входных значения HKDF-Expand, дающих одинаковый результат на выходе. Это требует стойкости базовой хэш-функции к конфликтам, а размер выходного значения HKDF-Expand должен быть не меньше 256 битов (или столько, сколько нужно хэш-функции для предотвращения конфликтов).

E.1.2. Аутентификация клиента

Клиент, отправивший данные аутентификации на сервер в процессе согласования или аутентификации после согласования, не может быть уверен в том, что сервер в дальнейшем будет считать клиента аутентифицированным. Если клиенту нужно определить, считает сервер соединение аутентифицированным с одной стороны или взаимно, это должно выполняться на прикладном уровне (см. [CHHSV17]). Кроме того, анализ аутентификации после согласования в работе [Kraw16] показывает, что клиент, идентифицированный сертификатом, отправленным после аутентификации, владеет ключом трафика. Следовательно, эта сторона является клиентом, участвовавшем в исходном согласовании, или клиентом, которому исходный клиент передал ключ трафика (в предположении, что этот ключ не скомпрометирован).

E.1.3. 0-RTT

Режим 0-RTT в общем случае обеспечивает защитные свойства, аналогичные режиму данных 1-RTT с двумя исключениями, - ключи шифрования 0-RTT надёжной секретности (forward secrecy) и сервер не способен гарантировать уникальность согласования (без возможности воспроизведения) без сохранения потенциально ненадёжного избыточного числа состояний. Механизмы ограничения раскрытия для повторного использования описаны в разделе 8. 0-RTT и Anti-Replay.

E.1.4. Независимость экспортёров

Значения exporter_master_secret и early_exporter_master_secret выводятся независимо от ключей трафика и поэтому не представляют угрозы безопасности трафика, зашифрованного с этими ключами. Однако эти значения можно использовать для расчёта значения любого экспортёра, их **следует** уничтожать как можно быстрей. Если известен полный набор меток экспортёра, реализации **следует** предварительно вычислить внутреннюю стадию Derive-Secret расчёта экспортёров для всех этих меток, а затем уничтожить [early_]exporter_master_secret, за которым следует каждое внутреннее значение,как только станет ясно, что больше секрет не потребуется.

E.1.5. Безопасность после компрометации

TLS не обеспечивает защиты для согласований, которые происходят после раскрытия долгосрочного секрета партнёра (ключ подписи или внешний PSK). Поэтому не обеспечивается защиты после компрометации [CCG16], которую иногда называют обратной или будущей секретностью (backward or future secrecy). Это отличается от стойкости KCI, где описываются гарантии защиты, которые сторона имеет после раскрытия её долгосрочного секрета.

E.1.6. Дополнительная литература

Дополнительный анализ согласования TLS можно найти в работах [DFGS15], [CHSV16], [DFGS16], [KW16], [Kraw16], [FGSW16], [LXZFH16], [FG17], [BBK17].

E.2. Уровень Record

Уровень записей зависит от согласования, создающего стойкие секреты трафика, из которых можно вывести двухсторонние ключи шифрования и nonce. В предположении, что это выполняется и ключи применяются для объёма данных, не превышающего заданный в параграфе 5.5 предел, уровень записей должен обеспечивать указанные ниже гарантии.

Конфиденциальность

Атакующий не сможет определить открытое содержимое данной записи.

Целостность

Атакующий не способен создать новую запись, которая отличается от существующей записи, но будет воспринята получателем.

Защита порядка и невоспроизводимость

Атакующий не сможет вынудить получателя к восприятию записи, которая уже воспринята или заставить его воспринять запись N+1, не обработав до этого запись N.

Сокрытие размера

На основе записи с данным внешним размером атакующий не сможет определить объем содержимого и заполнения в этой записи.

Прогрессивная защита после смены ключа

Если применяется механизм смены ключей, описанный в параграфе 4.6.3, и прежняя версия ключа удалена, атакующий, взломавший конечную точку, не сможет расшифровать трафик, использовавший старый ключ.

Неформально TLS 1.3 обеспечивает эти свойства защитой AEAD для открытых данных со стойким ключом. Шифрование AEAD [RFC5116] обеспечивает конфиденциальность и целостность данных. Невозможность воспроизведения обеспечивается с помощью своего значения nonce в каждой записи и выводом nonce из порядковых номеров записей (5.3. Nonce для отдельной записи) с независимой нумерацией на каждой стороне. Таким образом, записи доставленные с нарушением порядка, столкнутся с отказом при снятии защиты AEAD. Для предотвращения массового криптоанализа, когда одни и те же открытые данные шифруются разными пользователями с одним ключом (как это обычно происходит в HTTP) значения nonce формируются путём смешивания порядкового номера с секретным вектором инициализации (свой вектор для соединения), создаваемым вместе с ключами трафика. Анализ этой конструкции приведён в [BT16].

Метод смены ключей в TLS 1.3 (7.2. Обновление секретов трафика) использует конструкцию последовательного генератора, описанную в [REKEY], которая показала, что смена ключей позволяет применять кличи для большего числа операций шифрования, нежели возможно без смены. Это зависит от безопасности функции HKDF-Expand-Label в качестве PRF. Кроме того, пока эта функция действительно необратима, нет возможности рассчитать ключи трафика по использованным до смены (прогрессивная секретность - forward secrecy).

TLS не обеспечивает защиты данных, которые передаются через соединение после компрометации секрета трафика для этого соединения, т. е. TLS не обеспечивает защиты после компрометации (обратной или будущей защиты) применительно к секретам трафика. Действительно, атакующий, которому известен секрет трафика, может рассчитать все будущие секреты трафика для этого соединения. Системы, которым нужны гарантии, должны выполнить новое согласование и организовать новое соединение с обменом (EC)DHE.

E.2.1. Дополнительная литература

Дополнительный анализ уровня записи TLS приведён в [BMMRT15], [BT16], [BDFKPPRSZZ16], [BBK17], [PS18].

E.3. Анализ трафика

TLS подвергается множеству атак с анализом трафика, основанных на наблюдении размера и времени передачи шифрованных пакетов [CLINIC] [HCJC16]. Это особенно просто при наличии небольшого числа возможных сообщений, которые следует различать, например, для видеосервера с фиксированным содержимым, но даёт полезную информацию и в более сложных случаях.

TLS не обеспечивает какой-либо конкретной формы защиты от этого типа атак, но включает механизм заполнения, который могут использовать приложения. Открытые данные, защищённые функцией AEAD, включают содержимое и заполнение переменного размера, что позволяет приложениям создавать шифрованные записи произвольного размера, а также передавать трафик, содержащий только заполнение, чтобы скрыть различие между периодами передачи данных и «молчания». Поскольку заполнение шифруется вместе с реальным содержимым, атакующий не может напрямую определить размер заполнения, но может иметь возможность косвенно оценить его, используя каналы синхронизации, раскрытые в процессе обработки записи (т. е. видеть время обработки записи или отслеживать

записи, вызывающие отклик сервера). В общем случае неизвестно, как удалить все такие каналы, потому что даже функция удаления заполнения с постоянным временем, скорее всего будет передавать содержимое с зависимым от его размера временем. Как минимум, сервер или клиент с постоянным временем обработки будут требовать тесного взаимодействия с реализацией протокола прикладного уровня, включая постоянное время такого взаимодействия.

Примечание. Надёжная защита от анализа трафика будет с очевидностью снижать производительность работы приложений в результате вносимых задержек и роста объёма трафика.

E.4. Атаки по побочным каналам

В общем случае TLS не обеспечивает конкретной защиты против атак по побочным каналам (т. е. тех, где атака организуется через вторичный канал, например, канал синхронизации), оставляя эти меры для реализации соответствующих криптографических примитивов. Однако некоторые возможности TLS облегчают создание кода, устойчивого к побочным каналам.

- В отличие от прежних версий TLS, где применялась составная структура «MAC, затем шифрование», TLS 1.3 использует только алгоритмы AEAD, позволяя реализациям применять самодостаточные реализации примитивов с постоянным временем.
- TLS использует сигнал bad_record_mac для всех ошибок расшифровки, что позволяет не дать атакующему возможности получить сведения об отдельных частях сообщения. Дополнительная стойкость обеспечивается за счёт разрыва соединения при таких ошибках. Новое соединение будет использовать другой криптографический материал, предотвращая атаки на криптографические примитивы, которые требуют множества проверок.

Утечка информации через побочные каналы может происходить на уровнях выше TLS, в прикладных протоколах и использующих эти протоколы приложениях. Стойкость к таким утечкам зависит от приложений и прикладных протоколов, каждый из которых отвечает по отдельности за предотвращение утечки конфиденциальной информации.

E.5. Атаки на 0-RTT с повторным использованием

Воспроизводимые данные 0-RTT представляют множество угроз для использующих TLS приложений, если эти приложения не включают своей защиты от повторного использования (как минимум идемпотентность, но зачастую могут требоваться более жёсткие условия, такие как постоянное время отклика). Возможные атаки указаны ниже.

- Дублирование действий, вызывающее побочные эффекты (например, покупка или перевод денег) и наносящие вред сайту или пользователю.
- Атакующий может сохранить и воспроизвести сообщения 0-RTT для нарушения порядка среди других сообщений (например, удаляя их после создания).
- Использование поведения синхронизации кэша для раскрытия содержимого сообщений 0-RTT путём воспроизведения сообщения 0-RTT на другом узле кэша и использование отдельного соединения для измерения задержки запроса с целью проверки принадлежности обоих запросов к одному ресурсу.

Если данные могут быть воспроизведены много раз, это открывает дополнительные возможности для атак, таких как периодические измерения скорости криптографических операций. Кроме того, они могут перегружать системы, ограничивающие скорость. Дополнительная информация о таких атаках представлена в [Mac17].

В конечном счёте, серверы несут ответственность за защиту самих себя от атак с использованием репликации данных 0-RTT. Описанные в разделе 8 механизмы предназначены для предотвращения повторного использования на уровне TLS, но не обеспечивают полной защиты от получения множества копий данных клиента. TLS 1.3 возвращается к согласованию 1-RTT, когда у сервера нет никакой информации о клиенте, например, по причине размещения в другом кластере, который не имеет общего состояния, или в результате удаления квитанции, как описано в параграфе 8.1. Если протокол прикладного уровня повторно передаёт данные в такой системе, атакующий может вызвать дублирование сообщения путём передачи ClientHello в исходный кластер (который сразу обработает данные) и другой кластер, который вернётся к 1-RTT и обработает данные повторно на прикладном уровне. Масштаб такой атаки ограничен желанием клиента повторять транзакции и поэтому допускает лишь ограниченное число дубликатов, при этом каждый из них появляется на сервере как новое соединение.

При корректной реализации механизмы, описанные в параграфах 8.1 и 8.2, предотвращают многократное восприятие воспроизведённых ClientHello и связанных с ними данных 0-RTT любым кластером с согласованием состояний. Для серверов, ограничивающих использование 0-RTT одним кластером для одной квитанции, данное сообщение ClientHello и связанные с ним данные 0-RTT будут восприняты лишь один раз. Однако при неполной согласованности состояний атакующий сможет получить несколько копий данных, воспринятых в окне репликации. Поскольку клиент не знает точных деталей поведения сервера, ему **недопустимо** передавать в ранних данных сообщения, воспроизведение которых небезопасно и которые они не хотят повторять через множество соединений 1-RTT.

Прикладным протоколам **недопустимо** использовать данные 0-RTT без определяющего их применение профиля. Такой профиль должен указывать, какие сообщения и взаимодействия безопасны для использования с 0-RTT и как обрабатывать ситуации с отклонением сервером 0-RTT и возвратом к 1-RTT.

Кроме того, для предотвращения случайного некорректного использования реализациям TLS **недопустимо** разрешать 0-RTT (передачу или восприятие) без специального запроса приложения, а также **недопустимо** автоматически повторять данные 0-RTT, отвергнутые сервером, без явного указания приложения. Серверные приложения могут реализовать специальную обработку данных 0-RTT для некоторых типов трафика приложений (например, разрыв соединения, запрос повтора передачи данных на прикладном уровне, задержка обработки до завершения согласования). Чтобы позволить приложениям реализовать этот вид обработки, реализации TLS **должны** обеспечивать приложениям способ фиксации завершения согласования.

E.5.1. Воспроизведение и экспортёры

Воспроизведение ClientHello создаёт такой же ранний экспортёр, поэтому требуется дополнительное внимание со стороны приложений, использующих экспортёры. В частности, при использовании экспортёров в качестве привязки канала аутентификации (например, путём подписи вывода экспортёра), атакующий, который скомпрометировал PSK, может поменять аутентификаторы между соединениями без компрометации ключа проверки подлинности.

Кроме того, ранние экспортёры **не следует** применять для генерации ключей шифрования «от сервера к клиенту», поскольку это повлечёт повторное использование таких ключей. Использование ключей шифрования раннего трафика допустимо только в направлении от клиента к серверу.

E.6. Раскрытие отождествления PSK

Поскольку реализации реагируют на непригодную привязку PSK прерыванием согласования, атакующий может проверить пригодность отождествления данного PSK. В частности, если сервер воспринимает согласование как с внешним PSK, так и на основе сертификатов, действительное отождествление PSK приведёт к отказу согласования, тогда как непригодное отождествление будет просто пропущено и приведёт к успешному согласованию по сертификату. Серверы, поддерживающие только PSK-согласование, могут противостоять таким атакам, считая идентичными случаи отсутствия пригодного отождествления PSK и наличия отождествления с непригодной привязкой.

E.7. Общее использование PSK

TLS 1.3 использует осторожный подход к PSK, связывая их с конкретной функцией KDF. В отличие от этого, TLS 1.2 разрешает применять PSK с любой хэш-функцией и TLS 1.2 PRF. Таким образом, любой PSK, применяемый с TLS 1.2 и TLS 1.3, должен использоваться только с одним хэшем в TLS 1.3, что не совсем оптимально, если пользователи хотят предоставить один PSK. Конструкции в TLS 1.2 и TLS 1.3 различаются, хотя обе основаны на HMAC. Пока не известно, каким образом один PSK мог бы давать связанный вывод в обеих версиях, анализ будет ограниченным. Реализации могут обеспечить безопасность кросс-протокольного вывода, не применяя один PSK в TLS 1.3 и TLS 1.2.

E.8. Атаки на статический шифр RSA

Хотя TLS 1.3 не использует транспортировку ключей RSA и в результате не подвержен атакам Bleichenbacher [Blei98], при поддержке сервером TLS 1.3 статического RSA в контексте прежних версий TLS можно выдать себя за сервер для соединений TLS 1.3 [JSS15]. Реализации TLS 1.3 могут предотвратить такие атаки путём запрета поддержки статического RSA для всех версий TLS. В принципе, реализации также могут разделять сертификаты с разными битами keyUsage для статической расшифровки и подписи RSA, но этот метод основан на отказе клиентов воспринимать подписи, использующие ключи из сертификатов, в которых не установлен бит digitalSignature, а многие клиенты не применяют это ограничение.

Участники работы

Martin Abadi
University of California, Santa Cruz
abadi@cs.ucsc.edu

Christopher Allen (соредактор TLS 1.0)
Alacrity Ventures
ChristopherA@AlacrityManagement.com

Richard Barnes
Cisco
rlb@ipv.sx

Steven M. Bellovin
Columbia University
smb@cs.columbia.edu

David Benjamin
Google
davidben@google.com

Benjamin Beurdouche
INRIA & Microsoft Research
benjamin.beurdouche@ens.fr

Karthikeyan Bhargavan (редактор [RFC7627])
INRIA
karthikeyan.bhargavan@inria.fr

Simon Blake-Wilson (соавтор [RFC4492])
BCI
sblakewilson@bcisse.com

Nelson Bolyard (соавтор [RFC4492])
Sun Microsystems, Inc.
nelson@bolyard.com

Ran Canetti
IBM
canetti@watson.ibm.com

Matt Caswell
OpenSSL
matt@openssl.org

Stephen Checkoway
University of Illinois at Chicago
sfc@uic.edu

Pete Chown
Skygate Technology Ltd
pc@skygate.co.uk

Katriel Cohn-Gordon
University of Oxford
me@katriel.co.uk

Cas Cremers
University of Oxford
cas.cremers@cs.ox.ac.uk

Antoine Delignat-Lavaud (соавтор [RFC7627])
INRIA
antdl@microsoft.com

Tim Dierks (соавтор TLS 1.0, редактор TLS 1.1, 1.2)
Independent
tim@dierks.org

Roelof DuToit
Symantec Corporation
roelof_dutoit@symantec.com

Taher Elgamal
Securify
taher@securify.com

Pasi Eronen

Nokia
pasi.eronen@nokia.com

Cedric Fournet
Microsoft
fournet@microsoft.com

Anil Gangolli
anil@busybuddha.org

David M. Garrett
dave@nulldereference.com

Illya Gerasymchuk
Independent
illya@iluxonchik.me

Alessandro Ghedini
Cloudflare Inc.
alessandro@cloudflare.com

Daniel Kahn Gillmor
ACLU
dkg@fifthhorseman.net

Matthew Green
Johns Hopkins University
mgreen@cs.jhu.edu

Jens Guballa
ETAS
jens.guballa@etas.com

Felix Guenther
TU Darmstadt
mail@felixguenther.info

Vipul Gupta (соавтор [RFC4492])
Sun Microsystems Laboratories
vipul.gupta@sun.com

Chris Hawk (соавтор [RFC4492])
Corriente Networks LLC
chris@corriente.net

Kipp Hickman

Alfred Hoenes

David Hopwood
Independent Consultant
david.hopwood@blueyonder.co.uk

Marko Horvat
MPI-SWS
mhorvat@mpi-sws.org

Jonathan Hoyland
Royal Holloway, University of London
jonathan.hoyland@gmail.com

Subodh Iyengar
Facebook
subodh@fb.com

Benjamin Kaduk
Akamai Technologies
kaduk@mit.edu

Hubert Kario
Red Hat Inc.
hkario@redhat.com

Phil Karlton (соавтор SSL 3.0)

Leon Klingele
Independent
mail@leonklingele.de

Paul Kocher (соавтор SSL 3.0)
Cryptography Research
paul@cryptography.com

Hugo Krawczyk
IBM
hugokraw@us.ibm.com

Adam Langley (соавтор [RFC7627])
Google
agl@google.com

Olivier Levillain
ANSSI
olivier.levillain@ssi.gouv.fr

Xiaoyin Liu
University of North Carolina at Chapel Hill
xiaoyin.l@outlook.com

Ilari Liusvaara
Independent
ilariliusvaara@welho.com

Atul Luykx
K.U. Leuven
atul.luykx@kuleuven.be

Colm MacCarthaigh
Amazon Web Services
colm@allcosts.net

Carl Mehner
USAA
carl.mehner@usaa.com

Jan Mikkelsen
Transactionware
janm@transactionware.com

Bodo Moeller (соавтор [RFC4492])
Google
bodo@acm.org

Kyle Nekritz
Facebook
knekritz@fb.com

Erik Nygren
Akamai Technologies
erik+ietf@nygren.org

Magnus Nystrom
Microsoft
mnystrom@microsoft.com

Kazuho Oku
DeNA Co., Ltd.
kazuhooku@gmail.com

Kenny Paterson
Royal Holloway, University of London
kenny.paterson@rhul.ac.uk

Christopher Patton
University of Florida
cjpatton@ufl.edu

Alfredo Pironti (соавтор [RFC7627])
INRIA
alfredo.pironti@inria.fr

Andrei Popov
Microsoft
andrei.popov@microsoft.com

Marsh Ray (соавтор [RFC7627])
Microsoft
maray@microsoft.com

Robert Relyea
Netscape Communications
relyea@netscape.com

Kyle Rose
Akamai Technologies
krose@krose.org

Jim Roskind
Amazon
jroskind@amazon.com

Michael Sabin

Joe Salowey
Tableau Software
joe@salowey.net

Rich Salz
Akamai
rsalz@akamai.com

David Schinazi
Apple Inc.
dschinazi@apple.com

Sam Scott
Royal Holloway, University of London
me@samjs.co.uk

Thomas Shrimpton
University of Florida
teshrim@ufl.edu

Dan Simon
Microsoft, Inc.
dansimon@microsoft.com

Brian Smith
Independent
brian@briansmith.org

Brian Sniffen
Akamai Technologies
ietf@bts.evenmere.org

Nick Sullivan
Cloudflare Inc.
nick@cloudflare.com

Bjoern Tackmann
University of California, San Diego
btackmann@eng.ucsd.edu

Tim Taubert
Mozilla
ttaubert@mozilla.com

Martin Thomson
Mozilla
mt@mozilla.com

Hannes Tschofenig
Arm Limited
Hannes.Tschofenig@arm.com

Sean Turner
sn3rd
sean@sn3rd.com

Steven Valdez
Google
svaldez@google.com

Filippo Valsorda
Cloudflare Inc.
filippo@cloudflare.com

Thyla van der Merwe
Royal Holloway, University of London
tjvdmerwe@gmail.com

Victor Vasiliev
Google
vasilvv@google.com

Hoeteck Wee
Ecole Normale Superieure, Paris
hoeteck@alum.mit.edu

Tom Weinstein

David Wong
NCC Group
david.wong@nccgroup.trust

Christopher A. Wood
Apple Inc.
cawood@apple.com

Tim Wright
Vodafone
timothy.wright@vodafone.com

Peter Wu
Independent
peter@lekensteyn.nl

Kazu Yamamoto
Internet Initiative Japan Inc.
kazu@iij.ad.jp

Адрес автора

Eric Rescorla

Mozilla

Email: ekr@rtfm.com

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru