

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 21, 2014

E. Rescorla
Mozilla
February 17, 2014

New Handshake Flows for TLS 1.3
draft-rescorla-tls13-new-flows-01

Abstract

This document sketches some potential new handshake flows for TLS 1.3.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 21, 2014.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November

Internet-Draft

TLS 1.3 Flows

February 2014

10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1.	Introduction	4
2.	Background	4
2.1.	Handshake Flows	4
2.2.	Handshake Latency	5
2.3.	Plaintext Data in the Handshake	6
3.	Basic Assumptions	7
4.	Challenging Issues	8
5.	Design Principles	9
5.1.	Backward Compatibility is Required	9
5.2.	Remove Static Key Exchange	9
5.3.	Protect SNI But Require Common Crypto Parameters	10
6.	New Handshake Modes	10
6.1.	Overview	10
6.2.	New Behaviors	12
6.2.1.	Client Behavior for First Flight	12
6.2.1.1.	No Knowledge	12
6.2.1.2.	Server Keying Material	13
6.2.1.3.	Anti-Replay Token	13
6.2.2.	Server Behavior For First Flight	14
6.2.2.1.	Non-Optimistic Handshakes	15
6.2.2.2.	Predicted Parameters Correct	15
6.2.3.	Client Processing of First Flight	17
6.2.3.1.	Successful first Flight But No Anti-Replay	17
6.2.3.2.	Complete 0-RTT Handshake	18
6.2.4.	Session Resumption	18
6.3.	Example Flows	18
6.4.	New/Modified Messages	20
6.4.1.	EarlyData Extension	20
6.4.2.	EncryptedExtensions	21
6.4.3.	PredictedParameters	21

6.4.4.	ServerKeyExchange	22
6.4.5.	ServerParameters	23
6.4.6.	Anti-Replay Token	23
7.	Backward Compatibility	25
8.	Security Considerations	25

8.1.	Limits Of Identity Hiding	25
8.2.	Partial PFS	25
9.	Acknowledgements	26
10.	References	26
10.1.	Normative References	26
10.2.	Informative References	26
Appendix A.	Design Rationale	27
A.1.	EarlyData Extension	27
A.2.	Always Encrypt Client Parameters	27
A.3.	Always Restarting Non-Optimistic Handshakes	28
A.4.	Still TODO...	28
Appendix B.	Summary of Existing Extensions	28
Appendix C.	Non-recommended Flows	29
Author's Address	30

1. Introduction

DISCLAIMER: THIS IS A ROUGH DRAFT. EVERYTHING HERE IS SOMEWHAT HANDWAVY AND HASN'T REALLY HAD ANY SECURITY ANALYSIS.

The TLS WG is specifying TLS 1.3, a revision to the TLS protocol. The two major design goals for TLS 1.3 are:

- o Reduce the number of round trips in the handshake, providing at least "zero round-trip" mode where the client can send its first data message immediately without waiting for any response from the server.
- o Encrypt as much of the handshake as possible in order to protect against monitoring of the handshake contents.

This document proposes revisions to the handshake to achieve these objectives. They are being described in a separate document and for ease of analysis and discussion. If they are considered acceptable, some of them may be integrated into the main TLS document.

2. Background

In this section we briefly review the properties of TLS 1.2 [[RFC5246](#)]

2.1. Handshake Flows

As a reminder, this section reproduces the two major TLS handshake variants, from [[RFC5246](#)]. For clarity, data which is

cryptographically protected by the record protocol (i.e., encrypted and integrity protected) are shown in braces, as in {Finished}.

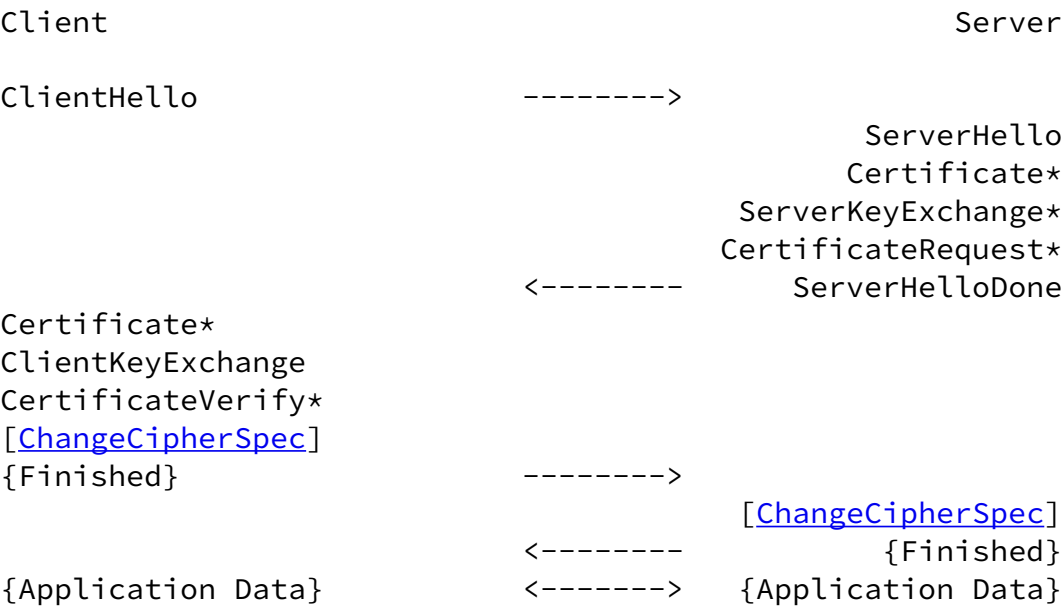


Figure 1: TLS 1.2 Full Handshake

Client

Server

```

ClientHello                ----->
                                ServerHello
                                [ChangeCipherSpec]
                                <-----
                                {Finished}

[ChangeCipherSpec]
{Finished}                ----->
{Application Data}        <-----> {Application Data}

```

Figure 2: TLS 1.2 Resumed Handshake

2.2. Handshake Latency

The TLS "Full Handshake" shown above incurs 2RTT of latency: the client waits for the server Finished prior to sending his first application data record. The purpose of the Finished is to allow the client to verify that the handshake has not been tampered with, for instance that the server has not mounted a downgrade attack on the cipher suite negotiation. However, if the client is satisfied with the handshake results (e.g., the server has selected the strongest parameters offered by the client), then the client can safely send its first application data immediately after its own Finished (this is often called either "False Start" or "Cut Through" [[I-D.bmoeller-tls-falsestart](#)]), thus reducing the handshake latency to 1RTT for a full handshake.

```

Client                                Server

ClientHello                ----->
                                ServerHello
                                Certificate*
                                ServerKeyExchange*
                                CertificateRequest*
                                <-----
                                ServerHelloDone

Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
{Finished}
{Application Data}        ----->

```

		[ChangeCipherSpec]
	<-----	{Finished}
{Application Data}	<----->	{Application Data}

TLS 1.2 with False Start

This technique is not explicitly authorized by the TLS specification but neither is it explicitly forbidden. A number of client implementations (e.g., Chrome, Firefox, and IE) already do False Start. However, because some servers fail if they receive the application data early, it is common to use some kind of heuristic to determine whether a server is likely to fail and therefore whether this optimization can be used.

The abbreviated handshake already succeeds in 1RTT from the client's perspective.

There have been proposals to take advantage of cached state between the client and server to reduce the handshake latency to 0RTT [[I-D.agl-tls-snapstart](#)]. However, they have not been widely adopted.

2.3. Plaintext Data in the Handshake

As shown in the figures above, essentially the entire handshake is in the clear. Some of these values are potentially sensitive, including:

- o The client certificate.
- o The server name indication (i.e., which server the client is trying to contact).
- o The server certificate (this is only interesting when the server supports name-based virtual hosting via SNI)

- o The next protocol in use [[I-D.ietf-tls-applayerprotoneg](#)].
- o The channel ID [REF: Channel ID]
- o The client cipher suite list (potentially usable for client fingerprinting.)

There have been proposals to address this just for extensions [[I-D.agl-tls-nextprotoneg](#)] as well as for the handshake as a whole [[I-D.ray-tls-encrypted-handshake](#)]. In general, the amount of privacy

protection which can be provided is somewhat limited by four factors:

- o A fair amount of information can be gathered from traffic analysis based on message size and the like.
- o Because the existing mechanisms do not encrypt these values, an active attacker can generally simulate being a server which does not accept whatever new handshake protection mechanisms are offered and force the client back to the old, unprotected mechanism. This form of active attack can be mitigated by refusing to use the old mechanism, however that is not always possible if one wishes to retain backward compatibility.
- o Many inspection devices mount a man-in-the-middle attack on the connection and therefore will be able to inspect information even if it is encrypted.
- o It's very hard to avoid attackers learning the server's capabilities because they generally fall into an easily probable/enumerable set and in most cases the clients are anonymous (and thus indistinguishable from attackers). Probably the best we can do is prevent attackers from learning which of a server's capabilities a given client is exercising.

However, there are still advantages to providing protection against passive inspection. The flows in this document attempt to provide this service to the extent possible.

3. Basic Assumptions

This section lays out the basic assumptions that motivate the designs in this document (aside from the objectives in [Section 1](#).

Retain Basic TLS Structure: The intent of this document is to retain the basic TLS structure and messages, tweaking them as minimally as necessary to accomplish the objectives in [Section 1](#).
Conservative design is good when working with security protocols.

implementations to interoperate with TLS 1.2 and below.

Minimize Variation: TLS already has a very large number of variant handshakes which makes it confusing to analyze. We would like to avoid multiplying this unnecessarily. We will probably deprecate some of the old flows in TLS 1.3.

0-RTT modes require server-side state: The existing TLS anti-replay mechanism involves the server and client jointly contributing nonces and therefore can be stateless on the server (as long as a fresh nonce can be generated.) Any 0-RTT mode in which the client sends data along with his initial handshake message must use some other mechanism to prevent replay, and this involves the server keeping some state.

Latency is often more important than bandwidth: Because networks are getting faster but the speed of light is not, it is often more important to minimize the number of round trips than the number of bits on the wire.

Client-side computation is often cheap: In many (but not all) cases, clients can afford to do a lot of cryptographic operations very cheaply.

Clients can and should be optimistic: When we put together the previous points, we come to the conclusion that it's OK for clients to be optimistic that things will succeed. So, for instance, it's OK to send messages to the server that might need to be retransmitted or recomputed if the server's state is not as expected. This is a key element of a number of round-trip reducing strategies.

[4.](#) Challenging Issues

This section previews some known challenging issues to keep in mind in the text below.

SNI privacy versus round-trips and variant server configuration: SNI is intended to alter the behavior of the server, but it also leaks information about the intended server identity. These demands are in tension. In situations where the server uses incompatible cryptography for the different SNIs (e.g., the servers use only RSA and use different keys) it is not possible to hide the SNI. In other cases, e.g., where DHE is used but authenticated with different keys, it is possible to have distinct configurations but at the cost of multiple RTs in order to first exchange keys and

then to send the encrypted SNI and then respond to the server's response.

Round-trips required for PFS: It's clearly not possible to do a 0-RTT handshake while also providing PFS. The two basic alternatives are (1) abandon PFS for 0-RTT handshake (maybe using re-handshakes on the same connection to get PFS) (2) Have a two-stage crypto exchange where the client initially uses a key generated using a cached server DH share and then and then later uses a key generated from a fresh server DH share. The latter approach seems too expensive to do on a regular basis.

[5.](#) Design Principles

[5.1.](#) Backward Compatibility is Required

It must be possible for TLS 1.3 implementations to interoperate with TLS 1.2 and below. In the vast majority of cases this has to happen cleanly and without a whole pile of extra round trips. This means that, for instance, you can't just unconditionally send a ClientHello that no TLS 1.2 server can accept and then do application layer fallback. It may be acceptable to do so with a server you have good reason to believe knows TLS 1.3, but that must not happen frequently and there must be a fast way to fall back that never (or almost never) fails.

[5.2.](#) Remove Static Key Exchange

We propose to eliminate all the static key exchange modes; this principally means RSA since that is widely used (whereas static DH and ECDH is not). The two major arguments for this change are:

- o Static cipher suites are inherently non-forward secure. Modern practice favors forward-secure algorithms.
- o They add protocol complexity because we need to handle both ephemeral and static modes.

In addition, RSA has a much worse performance/nominal security profile than the ECDHE modes which we are likely to use for ephemeral keying.

The major argument for continuing to allow static RSA is performance, but ECDHE is fast enough that RSA + ECDHE is only marginally slower than static RSA. Note that ephemeral keying can be used with the existing RSA certificates, so there is no new deployment cost for

servers.

[5.3.](#) Protect SNI But Require Common Crypto Parameters

If you want to encrypt SNI data, then the server needs to be willing to do encryption with the same keying material for all the virtual servers; since we have banned static RSA above, that means effectively the same DHE/ECDHE group. Also, encrypted SNI is inherently incompatible with TLS 1.2 and below, since those servers need to be able to see the SNI in order to know what certificate to choose. Thus, any use of encrypted SNI will either need to come with some sort of at least semi-graceful fallback to non-encrypted SNI or accept that encrypted SNI can only be used with known TLS 1.3 servers. This does not mean, however, that the server cannot use a different certificate; it merely needs to be willing to advertise and use the same initial ephemeral key for each virtual server.

[6.](#) New Handshake Modes

This document includes a number of strategies for improving latency and privacy, including:

- o Move the CCS up in the handshake with respect to other messages so that more of the handshake can be encrypted, thus improving protection against passive handshake inspection.
- o Allow the client to send at least some of his second flight of messages (ClientKeyExchange, CCS, Finished, etc.) together with the first flight in the full handshake, thus improving latency.
- o Allow the client to send data prior to receiving the server's messages, thus improving latency.

In addition, where prior versions of TLS generally assume that the client is totally ignorant of the server's capabilities (e.g., certificate and supported cipher suites) we assume that the client has prior information about the server, either from prior contact or some discovery mechanism such as DNS.

[6.1.](#) Overview

Our basic target is a 1-RTT handshake predicated on the assumption

that the client has a semi-static DHE/ECDHE key for the server. This key can have been acquired in a number of possible ways, including a prior interaction, a DNS lookup, or via an extra round trip in the same handshake. Assuming that the client knows this key, you end up with the handshake shown in Figure 3.

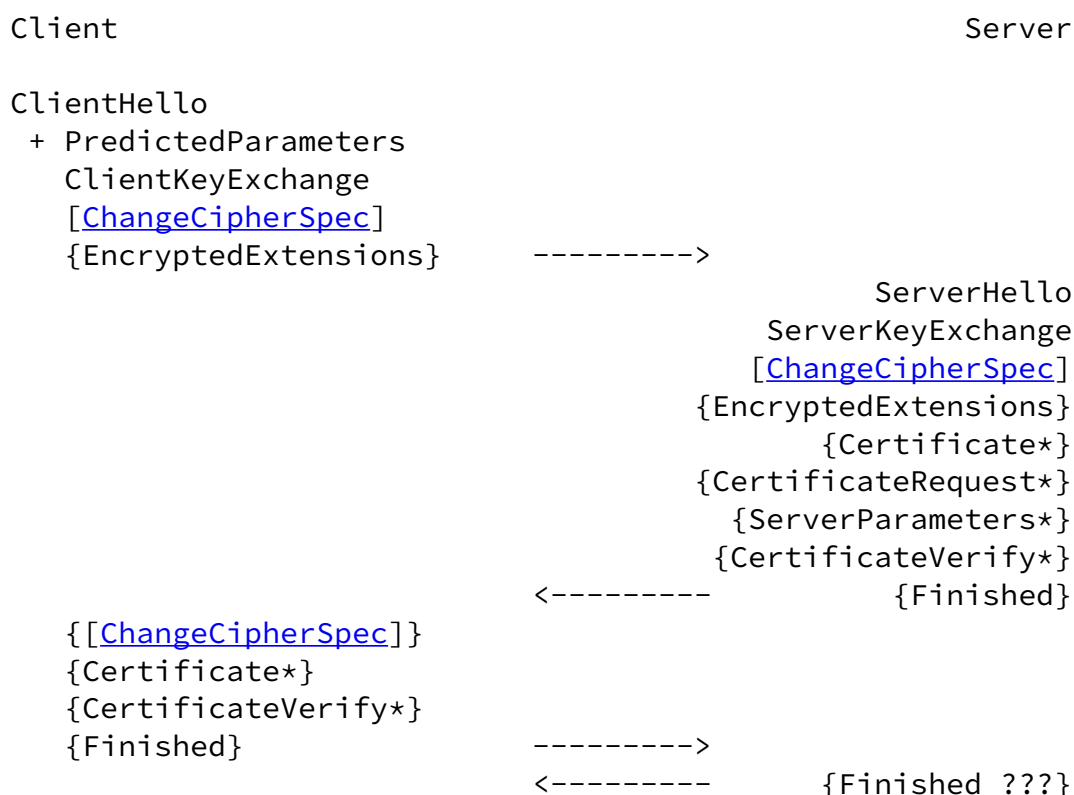


Figure 3: Basic 1-RTT Handshake

The most important thing to note about this handshake is that because the client has initial knowledge of the server's key, the client sends data to the server using two different cryptographic contexts. The first is established by pairing the server's previously-known key pair with the client's key and is used only to encrypt the client's in the first flight. These messages do not get PFS. The second context is based on the server's ephemeral key found in the **ServerKeyExchange** and the client's ephemeral key share and is used to

encrypt the rest of the handshake messages and any subsequent data. These messages can have PFS. Note that the client sends two ChangeCipherSpec messages in order to delineate the transition between cleartext and the first context and then between the first and second contexts.

This basic handshake can be extended with two major variants:

- o If the client has no knowledge of the server's parameters, it sends an initial ClientHello to solicit the server's DHE/ECDHE key. Once it has the server's key, it simply sends the same handshake messages as shown above. This is shown in Figure 4
- o Once the client and server have communicated once, they can establish an anti-replay token which can be used by the client to do a 0-RTT handshake with the server in future. This is shown in Figure 5.

[[OPEN ISSUE: This document does not currently discuss renegotiation. It is an open question whether we should do so.]]

[6.2.](#) New Behaviors

[6.2.1.](#) Client Behavior for First Flight

The contents of the client's initial flight (what would be the ClientHello in TLS 1.2 and below) depend on the client's knowledge -- or at least beliefs -- about the server. The remainder of this section describes appropriate client behavior in each of the major cases. Note that these cases are described in sequence of increasing client knowledge. The client may of course be wrong about the server's state.

[6.2.1.1.](#) No Knowledge

In the simplest case, the client has no knowledge of the server's state and does not wish to do anything speculative. In that case, the client simply generates a standard TLS ClientHello with essentially the same contents as would be in TLS 1.2 (though with a TLS 1.3 version number). This ClientHello is backward compatible with existing TLS 1.2 servers, modulo any issues with version negotiation.

The client has two options here vis-a-vis sensitive information in the extensions (principally SNI and/or ALPN).

- o Send a message without SNI and ALPN. This will be compatible with any TLS 1.3 server because that server will respond with its DHE key, but may have compatibility issues with TLS 1.2 servers which have differential behavior for SNI and ALPN.
- o Send a message with SNI and/or ALPN. This will be compatible with TLS 1.2 servers as well as TLS 1.3 but leaks some potentially sensitive information.

If the client has strong reason to believe that the server supports TLS 1.3, it SHOULD use the first option. [[OPEN ISSUE: Is there a way to have clean fallback if the client doesn't provide SNI? Need to determine how TLS 1.2 and below servers behave if you (a) offer a high version of TLS and (b) don't offer SNI. I suspect they offer a default certificate in which case you could detect this case. This is not an issue for ALPN, since you just won't negotiate the next protocol.]]

In either case, a TLS 1.3 server will respond to this message with a ServerKeyExchange, as in [Section 6.2.2.1](#). The client then knows the server's keying material and so can restart the handshake as

described in the next section.

[6.2.1.2](#). Server Keying Material

Once the client knows a valid DHE/ECDHE share for the server, it can start to send encrypted handshake data (but not application data) immediately. In order to do so, it SHOULD send along with its ClientHello:

- o A PredictedParameters ([Section 6.4.3](#)) message containing the expected server selected parameters based on previous negotiations or other out of band information. Note that the client is not selecting out of server preferences; it is attempting to predict the cipher suites the server will select out of the client's advertised preferences.
- o A ClientKeyExchange message containing an appropriate DHE/ECDHE share.
- o A ChangeCipherSpec message to indicate that it is now sending

- encrypted data.
- o An EncryptedExtensions ([Section 6.4.2](#)) message containing any extensions which should be transmitted confidentially such as SNI or ALPN.

In order to preserve compatibility with pre TLS 1.3 intermediaries all of this data is packed into an EarlyData extension ([Section 6.4.1](#)).

The encryption keys for the encrypted messages are computed using the ordinary PRF construction but with an all-zero ServerRandom value. Thus implies that there is no guarantee of freshness from the server's side but that the client knows that the keying material is fresh provided it generated the ClientRandom correctly. [[OPEN ISSUE: If the client and server have previously exchanged messages (see [Section 6.2.1.1](#)) then we could use that ServerRandom, but this just makes the security properties more confusing.]]

[6.2.1.3](#). Anti-Replay Token

If the client also has an anti-replay token (see [Section 6.4.6](#) for details) it can act as in the previous section but also include the anti-replay information in the EncryptedExtensions message (the information is encrypted here to avoid linkage between multiple handshakes by the same client).

If previous interactions with the same server indicate that client authentication is required [[TODO: provide an explicit signal for this in either ServerParameters or CertificateRequest]], the client MUST also provide Certificate and CertificateVerify messages.

In addition, a client MAY include one or more encrypted application_data records containing data for the server to process. These records MUST follow the Finished message.

[[TODO: We need to tie the CertificateVerify to the server certificate. In the other handshakes, this binding is a side effect of having the server certificate supplied in the handshake. Here, the certificate is implicit, so the client's signature doesn't cover it automatically. One option is to have the client replay the server's Certificate message after the AntiReplayToken. The cached_information extension could allow this to be replaced with a

hash of the same.]]

[6.2.2.](#) Server Behavior For First Flight

Upon receiving the client's first flight, the server must examine both the ClientHello information and the EarlyData information to determine the extent to which the client has optimistically added information and the extent to which the client's optimism is warranted. The server SHOULD follow the following algorithm or an equivalent one:

1. If the client's maximum version number is less than 1.3, then proceed as defined in [[RFC5246](#)].
2. Perform cipher suite negotiation and extension negotiation as specified in [[RFC5246](#)] and remember the results and the resultant ServerHello. NOTE: This is why SNI requires the cryptographic parameters to be identical for each virtual host.
3. If the client has not included a ClientKeyExchange in the ClientHello, this is a non-optimistic handshake, proceed as described [Section 6.2.2.1](#). Send a ServerHello and ServerKeyExchange to give the client the server's parameters.
4. If the ClientKeyExchange message is incompatible with the negotiated parameters, then this is a failed optimistic handshake. Proceed as in [Section 6.2.2.1](#).
5. If a PredictedParameters message is present, check that it matches the negotiated parameters from step 2. Note that this means that the client must predict *exactly* the cipher suite and compression parameters that the server selects. If there is a failed match, this is a failed optimistic handshake so proceed as if it were not optimistic (See [Section 6.2.2.1](#).) Otherwise, this is a successful optimistic handshake, so proceed as in [Section 6.2.2.2](#).
6. Anything else is an error.

[[TODO: rewrite the logic here for more clarity??]]

[6.2.2.1.](#) Non-Optimistic Handshakes

In the case where the client has not been optimistic or has been optimistic but wrong, the server simply sets the client's

expectations so that the client can try again. Specifically, this means that the server responds with the following messages:

- o ServerHello indicating the negotiated parameters.
- o ServerKeyExchange ([Section 6.4.4](#))
- o ServerHelloDone

[[OPEN ISSUE: This looks a lot like the of the messages described in [Section 6.2.2.2](#), so the client needs to infer that he needs to restart based on the ServerHelloDone being present and Finished being absent. This works, but it might be better to make it more explicit by adding a new message such as HelloRequest.]]

The server MUST ignore any client extensions which are not necessary to negotiate the cryptographic parameters. This does not mean that they will not be used, merely that they will be negotiated in a subsequent exchange. Note that TLS 1.2-compatible clients generally will need to put SNI and perhaps ALPN in their ClientHello. Because TLS 1.3 servers MUST use common cryptographic parameters regardless of these extensions (see [Section 5.3](#)), the server MUST ignore these values. The client responds to these messages by sending a new ClientHello that conforms to the servers known expectations, as in [Section 6.2.1.2](#). The original round-trip messages MUST be included in the handshake hashes for the Finished to tie them to the rest of the handshake.

[[OPEN ISSUE: This is a deliberately missed opportunity for a modest optimization. If the client has already provided a full ClientHello as would be needed for TLS 1.2, then the server could do a TLS 1.2-style handshake including sending its Certificate, CertificateRequest, etc., whereas we now have to have a full round trip. However, in the name of reducing protocol complexity, we are eschewing this. An alternate choice would be to simply fall back to the TLS 1.2 behavior and do a TLS 1.2-style handshake.]]

[6.2.2.2](#). Predicted Parameters Correct

If the client has correctly predicted the server parameters (i.e., the cipher suite, compression, etc. that the server would have selected based on the ClientHello), then the client and server now share a PreMaster Secret based on the client's ClientKeyExchange and the previously provided ServerParameters. The server computes the PMS, Master Secret, and traffic keys and decrypts the rest of the client's handshake messages. If any non-handshake messages are

present, this is an error and the server MUST fail with a fatal "unexpected_message" alert.

Once the server has generated the keys, it MUST process the client's EncryptedExtensions, which contains any extensions it wants protected. The EncryptedExtensions MAY contain an Anti-Replay Token (ART) ([Section 6.4.6](#)), which may either be valid or invalid.

[6.2.2.2.1](#). Missing or Invalid ART

If the ART is missing or invalid, then this is a basic 1-RTT handshake. The server ignores any application_data records as well as client handshake messages other than those specified in [Section 6.2.1.2](#) and sends his first flight of messages in the following order:

- o ServerHello
- o ServerKeyExchange ([Section 6.4.4](#))
- o ChangeCipherSpec
- o EncryptedExtensions ([Section 6.4.2](#))
- o Certificate*
- o CertificateRequest*
- o ServerParameters* [Section 6.4.5](#)
- o CertificateVerify* (if Certificate provided)
- o Finished [[TODO:AlmostFinished?]]

The use of the CertificateVerify is new in this context. In prior versions of TLS, the CertificateVerify was only used to authenticate the client. Here it is also used to authenticate the server. This usage has the side benefit that it authenticates the entire handshake up to this point, not just the server's key.

Note that everything after the ServerKeyExchange and ChangeCipherSpec is encrypted, thus this mode provides limited privacy. All extensions other than those required to establish the cryptographic parameters MUST be in the EncryptedExtensions, not the ServerHello. Specifically, it protects the server's certificate (but not SNI) and the selected ALPN data.

[6.2.2.2.2](#). Valid ART

If the ART is valid, then this is a 0-RTT handshake. The server MUST verify that the client sent the appropriate handshake messages, including Certificate and CertificateVerify if required by server policy, as well as a valid Finished message. The server then generates and sends its own first flight which is exactly the same as above but MUST omit the CertificateRequest, since the client MUST

already have provided its certificate if required.

The EncryptedExtensions MUST include an ART indicator that matches the client's ART so that the client knows that the 0-RTT handshake was successful and that the client's application_data was accepted. The details of how this works are TBD. (See [Section 6.4.6.](#))

The server MUST also process any application_data records in the client's initial flight.

[[TODO: Open issue: should we require Certificate and CertificateVerify for 0-RTT handshakes? The client in principle has the server's parameters, but it would make life more consistent to have less options and the signature isn't that big a deal any more. The cost here is the computation and the message size.]]

[6.2.3.](#) Client Processing of First Flight

Upon receiving the server's first flight, the client must examine the server's messages to determine what happened. The client SHOULD follow an algorithm equivalent to the following.

1. If the server version is 1.2 or below, then follow the processing rules for [\[RFC5246\]](#).
2. If the server has provided only a ServerHello, ServerKeyExchange, and ServerHelloDone, then all optimistic key exchange has failed. Re-send a ClientHello with the provided server key and negotiated parameters as in [Section 6.2.1.2](#)
3. If the server provided a full flight of messages but no anti-replay token in the EncryptedExtensions then the client needs to process the messages and send his second flight as described in [Section 6.2.3.1.](#)
4. If the server responds with an anti-replay token as well as a full flight of messages the handshake is finished and the client can assume that any data it sent was processed. See [Section 6.2.3.2](#)
5. If the server supplies a ServerParameters message, the client SHOULD remember those parameters for use with future handshakes and forget any previous ServerParameters for this server. The ServerParameters are not used for this connection.

[6.2.3.1.](#) Successful first Flight But No Anti-Replay

If the server's first flight is complete but has no anti-replay token, then the handshake is not quite complete: the client processes the messages and generates its second flight, consisting of:

- o ChangeCipherSpec (to indicate either the key change)
- o Certificate and CertificateVerify (if client authentication was requested)
- o Finished

At this point, the client can start sending `application_data`. If any application data was sent with the original ClientHello, the server will have discarded it and it must be retransmitted.

[[OPEN ISSUE: Do we need a final Finished from the server. The only thing it does is confirm the server's receipt of the client certificate, but at the cost of an RT if the client actually checks it.]]

[6.2.3.2](#). Complete 0-RTT Handshake

If the server has provided an anti-replay token that matches the client's, the handshake is complete. The client MUST then send a ChangeCipherSpec and Finished to acknowledge the new key provided by the server in the ServerKeyExchange. [[OPEN ISSUE: Can we remove Finished here?]]

[6.2.4](#). Session Resumption

While resumption adds significant complexity, there are going to be low-end devices which still need to support resumption. This is particularly relevant for the Internet-Of-Things type devices. Note that this question is orthogonal to the non-resumed handshake flows.

[TODO: This section still needs to be written depending on if people want to do resumption. It should be straightforward.]

[6.3](#). Example Flows

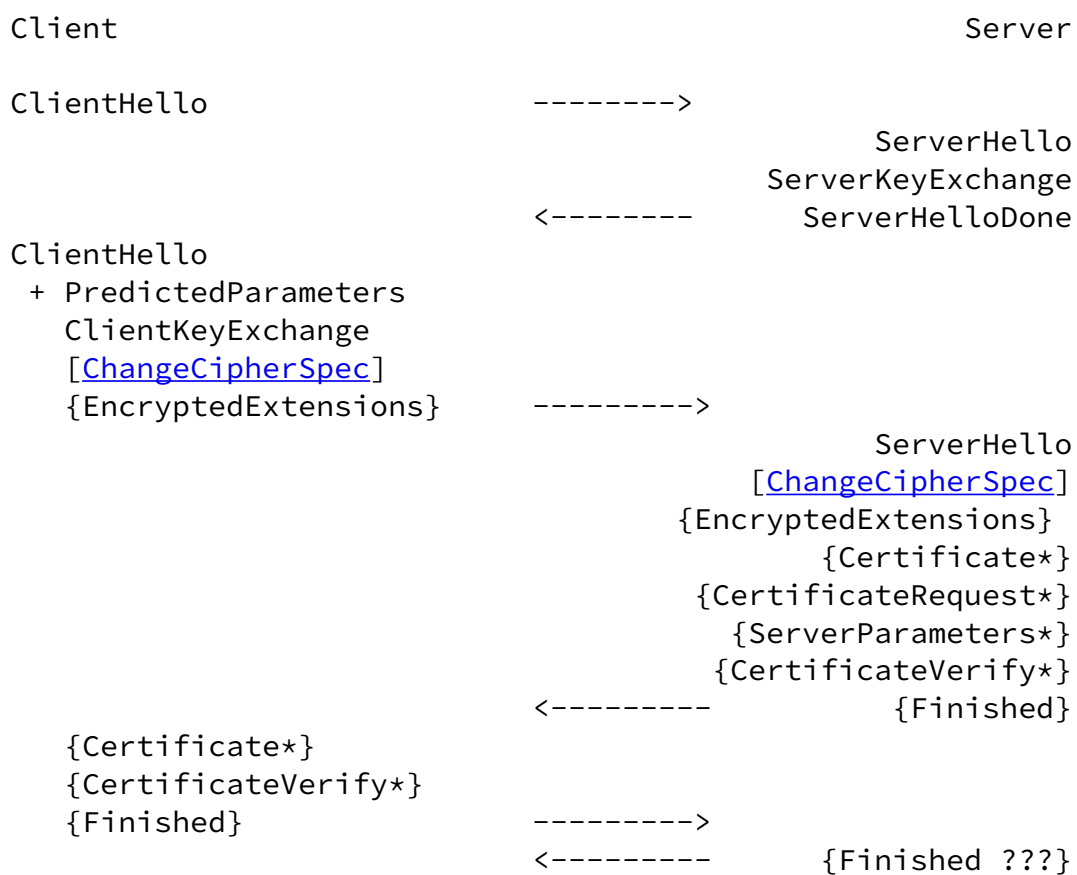


Figure 4: Non-Optimistic Handshake

[TODO: Can we remove the first finished rather than the second finished. Could we remove a RT.]]

Rescorla

Expires August 21, 2014

[Page 19]

Internet-Draft

TLS 1.3 Flows

February 2014

Client

Server

ClientHello

- + PredictedParameters
- ClientKeyExchange
- [\[ChangeCipherSpec\]](#)
- {EncryptedExtensions
- + AntiReplayToken}
- {Certificate*}
- {CertificateVerify*}
- {Finished}
- {ApplicationData}

----->

ServerHello

- ServerKeyExchange
- [\[ChangeCipherSpec\]](#)
- {EncryptedExtensions
- + AntiReplayToken}
- {Certificate*}
- {CertificateRequest*}

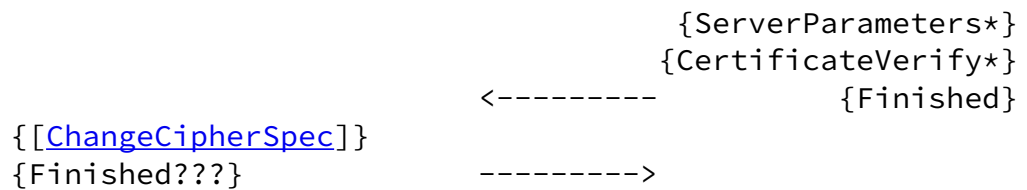


Figure 5: Handshake With Anti-Replay

6.4. New/Modified Messages

6.4.1. EarlyData Extension

In order to comply with TLS 1.2, any messages which we wish to be in the client's first flight must be packaged as extensions to the ClientHello. The EarlyData extension is usable for this purpose.

```

struct {
    TLSCipherText messages<5 .. 2^24-1>;
} EarlyDataExtension;
  
```

The client may include no more than one EarlyData extension in its ClientHello. The extension simply contains the TLS records which would otherwise have been included in the client's first flight. Any data included in EarlyData is not integrated into the handshake hashes directly. Instead, it is hashed in as marshalled into the extension. Note that this may include application_data traffic. Because this is an extension, the server is explicitly permitted to ignore these messages and the client must be prepared to continue properly. However, TLS 1.3 servers SHOULD process any messages in

the EarlyData extension, though it may not ultimately be able to use them.

[[Open Issue: in this version, we never send EarlyData unless we have evidence that the server is TLS 1.3. In principle we might be able to just send the messages directly, but what about middleboxes which have appeared in the path since we discovered that. In that case, we would need a ClientHelloDone]]

6.4.2. EncryptedExtensions

```

struct {
  
```

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

The EncryptedExtensions message simply contains any extensions which should be protected, i.e., any which are not needed to establish the cryptographic context. It **MUST** only be sent after the switch to protected operations. See [Appendix B](#) for guidance on which extensions go where.

[6.4.3.](#) PredictedParameters

```
struct {
    ProtocolVersion version;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
    opaque server_key_label<0..2^16-1>;
    Extension extensions<0..2^16-1>;
} PredictedParameters;
```

The PredictedParameters message contains the clients prediction of the parameters that the server will select and therefore which the client is optimistically using for the connection. The values here are:

version
The negotiated TLS version.

cipher_suite
The selected cipher suite.

compression_method

The negotiated TLS compression method. **[[OPEN ISSUE: Should we just remove compression.]]**.

server_key_label
The label for the server key (provided in a [Section 6.4.5](#)

message).

extensions

Any extensions which form part of the SecurityParameters, such as "truncated_hmac" or "max_fragment_length".

[6.4.4.](#) ServerKeyExchange

In the existing TLS handshake, if the server is authenticated (the most common case) the ServerKeyExchange contains a digital signature over the server's ephemeral public keying material. However, this obviously leaks the identity that the server is using; even if the server certificate is encrypted an attacker can simply iterate over the server's certificates until it finds one that allows verification of the signature (at least in the case where each SNI uses a different certificate).

In order to remove this issue, we remove the signature from the ServerKeyExchange and use the CertificateVerify message (which was previously only used for client certificate verification).

```
struct {
    select (KeyExchangeAlgorithm) {
        case dhe_dss:
        case dhe_rsa:
            ServerDHParams params;

        case ec_diffie_hellman:
            ServerECDHParams params;
    }
} ServerKeyExchange;
```

[6.4.5.](#) ServerParameters

```
struct {
    opaque label<0..2^16-1>;
    uint32 lifetime;
    select (KeyExchangeAlgorithm) {
        case dhe_dss:
        case dhe_rsa:
            ServerDHPParams params;

        case ec_diffie_hellman:
            ServerECDHPParams params;
    };
} ServerParameters;
```

The ServerParameters message is used to indicate the server's "long-term" (really medium-term) parameters, specifically a DHE/ECDHE key pair which can potentially be used by the client for future connections, thus shortcutting the round-trip to discover the server's DH key. [[OPEN ISSUE: Should we add other policy-like stuff like whether we expect client auth? This would make the behavior expected for 0-RTT handshakes explicit rather than implicit. I am leaning towards this.]] The params are the same fields as in the ServerKeyExchange [[OPEN ISSUE: Merge these?]]. The other fields are as follows.

label

A label for these parameters which can be re-sent by the client in the PredictedParameters messages. Note that this is left potentially quite large to allow the server to pack data into it, though this may have an impact on performance.

lifetime

The expected useful lifetime of these parameters in seconds from now.

[6.4.6.](#) Anti-Replay Token

There are a number of potential mechanisms for detecting replay on the server side. Generally all of them require the server to give the client some identifier which is later replayed to the server to help it recover state and detect replays.

This section lists the main ones I know of:

- o Memorize arbitrary client values, typically within some scope, (as in [[I-D.agl-tls-snapstart](#)])
- o Force the client to use an identifier + values in some ordered way (e.g., by using a sliding anti-replay bitmask)
- o Give the client some finite number of tokens that you remember until they are used (as an enhancement, these could be DH public keys)

Each of these approaches has costs and benefits; generally the exact form of the anti-replay mechanism is orthogonal to the design of 0-RTT handshake.

As an example, consider the mechanism used by Snap Start, which assumes roughly synchronized clocks between the client and the server. The server maintains a list of all the ClientRandom values offered within a given time window (recall that the TLS ClientRandom value contains a timestamp) and rejects any ClientRandoms which appear to be outside that window. The server also provides the client with an 8-byte "orbit" value (which would serve here as the Anti-Replay Token) which can be used to separate the anti-replay lists from distinct servers. If a server sees a duplicate ClientRandom, one from a different orbit, or one from a time outside the window, it rejects it and forces the client to complete the handshake with a fresh server nonce.

If we were to follow this model, we would have something like the following. [[OPEN ISSUE: The following is extremely handwavy and presented as one possibility. It needs to be gone over more carefully.]]

```
struct {
    opaque orbit[8];
    uint64 time;

    select (ConnectionEnd) {
        case Client:
            opaque certificate_hash;
        case Server:
            ; // Empty
    };
} AntiReplayToken;
```

These fields would be something like:

`orbit`

An arbitrary 64-bit quantity selected by the server.

`time`

The current time in microseconds since the UNIX epoch.

`certificate_hash`

A digest of the server's expected Certificate message using the Hash from the PRF. This would need to be checked by the server to verify that the client is trying to talk to the right server.

[[OPEN ISSUE: This still seems like kind of a mess.]]

[7.](#) Backward Compatibility

The high-order compatibility issue for any change to TLS is whether the ClientHello is compatible with old servers. To a first order, we deal with this by putting any new information in extensions. This is ugly, but mostly works.

[8.](#) Security Considerations

Everything in this document needs further analysis to determine if it is OK. Comments on some known issues below.

[8.1.](#) Limits Of Identity Hiding

If the SNI and ALPN information are only sent encrypted (i.e., are not included in an initial unencrypted ClientHello) then they are secure from a passive attacker, and thus so is the server's identity. An active attacker can, however, force the client to reveal them by simulating a non-optimistic handshake (even if the client has a previous ServerParameters value) and sending his own DHE/ECDHE share,

thus eliciting a version of the extensions encrypted to him. This cannot be done transparently to the client, however, since it will generate a handshake failure.

By contrast, since the server authenticates first and the client only encrypts its certificate under keying material which has been authenticated by the server, the attacker does not learn the client's identity.

[8.2.](#) Partial PFS

If the client opts to use an optimistic handshake, any messages the client sends before the ServerKeyExchange will not have PFS, as they

Rescorla

Expires August 21, 2014

[Page 25]

Internet-Draft

TLS 1.3 Flows

February 2014

will be encrypted with a key which is necessarily somewhat long-term. The server can of course limit the exposure of the data by limiting the lifetime of keys, but at the cost of reducing the success probability of optimistic handshakes.

Servers can opt to always require PFS by never supplying ServerParameters, thus forcing the client to go through a full handshake. Clients can opt to always require PFS by never offering an optimistic handshake.

[9.](#) Acknowledgements

This document borrows ideas and/or has benefitted from feedback from a large number of people, including Dan Boneh, Wan-Teh Chang, Steve Checkoway, Matt Green, Cullen Jennings, Adam Langley, Nagendra Modadugu, Bodo Moeller, Andrei Popov, Marsh Ray, Hovav Shacham, Martin Thomson, and Brian Smith.

[10.](#) References

[10.1.](#) Normative References

[I-D.ietf-tls-cached-info]

Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension",
[draft-ietf-tls-cached-info-16](#) (work in progress),

February 2014.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), March 1997.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", [RFC 5246](#), August 2008.

[10.2](#). Informative References

[I-D.agl-tls-nextprotoneg]

Langley, A., "Transport Layer Security (TLS) Next Protocol Negotiation Extension", [draft-agl-tls-nextprotoneg-04](#) (work in progress), May 2012.

[I-D.agl-tls-snapstart]

Langley, A., "Transport Layer Security (TLS) Snap Start", [draft-agl-tls-snapstart-00](#) (work in progress), June 2010.

[I-D.bmoeller-tls-falsestart]

Rescorla

Expires August 21, 2014

[Page 26]

Internet-Draft

TLS 1.3 Flows

February 2014

Langley, A., Modadugu, N., and B. Moeller, "Transport Layer Security (TLS) False Start", [draft-bmoeller-tls-falsestart-00](#) (work in progress), June 2010.

[I-D.ietf-tls-applayerprotoneg]

Friedl, S., Popov, A., Langley, A., and S. Emile, "Transport Layer Security (TLS) Application Layer Protocol Negotiation Extension", [draft-ietf-tls-applayerprotoneg-04](#) (work in progress), January 2014.

[I-D.ray-tls-encrypted-handshake]

Ray, M., "Transport Layer Security (TLS) Encrypted Handshake Extension", [draft-ray-tls-encrypted-handshake-00](#) (work in progress), May 2012.

[Appendix A](#). Design Rationale

This section attempts to explain some of the design decisions in this document.

[A.1.](#) EarlyData Extension

In TLS 1.2, the client's first flight consists solely of the ClientHello message. In TLS 1.3, we allow other handshake messages, but these are likely to cause incompatibilities. Specifically, if the client sends other messages besides the ClientHello it will cause handshake failures. In prior versions of this document, this was necessary because the client could optimistically send messages even if it did not know that the server supported TLS 1.3. In the current version of the document, this is not possible, but there may still be intermediaries in the path which enforce pre TLS 1.3 semantics. These intermediaries may have been added after the client's last contact with the server. For that reason, we continue to keep these extra messages in an extension. This may change if research indicates it is unnecessary.

[A.2.](#) Always Encrypt Client Parameters

If there is a commonly-recognized DHE/ECDHE group, the client can just offer its DHE/ECDHE key in the ClientHello and hope that the server supports it, as shown in Figure 6. This mode would protect only the server's response to the client but not the client's extensions, etc. We want to have a mode which does protect the client's extensions (principally SNI) and supporting both modes adds complexity, so we opted not to provide the partially-encrypted mode. If the WG wanted to add it, we could do so.

[A.3.](#) Always Restarting Non-Optimistic Handshakes

The non-optimistic handshake shown in Figure 4 requires the client to restart with a new ClientHello even if the server could actually have sent a full first flight as in TLS 1.2. This case could occur either because the client sent SNI and ALPN for backwards compatibility or because the server doesn't have behavior contingent on SNI/ALPN. The result is that the handshake takes an extra round trip when compared to TLS 1.2 with False Start.

The alternative would be to allow a server to do a TLS 1.2-style handshake if the client had supplied enough information to do so (though we could encrypt the client's Certificate if we chose). This would shorten the round trip time in the case where the client and

server had no previous contact (though with the known security restrictions of The major argument against this alternative is that it is yet another mode in an already complex protocol.

[A.4.](#) Still TODO...

[[TODO: Explain the following]]

- o Server-side parameter selection versus client-side parameter selection.
- o Public semi-static server parameters versus client-specific parameters.
- o Correction instead of negotiation

[Appendix B.](#) Summary of Existing Extensions

Many of the flows in this document have Extensions appear both both in the clear and encrypted. This section attempts to provide a first-cut proposal (in table form) of which should appear where.

Extension	Privacy level	Proposed Location
server_name	Medium	Server chooses (*)
max_fragment_length	Low	Clear
client_certificate_url	High	Encrypted
trusted_ca_keys	Low	Server chooses (*)
truncated_hmac	Low	Clear

status_request	Low	Encrypted
user_mapping	?	?
client_authz	?	?
server_authz	?	?
cert_type	Low	Server chooses (*)
elliptic_curves	Low	Clear
ec_point_formats	Low	Clear
srp	High	????
signature_algorithms	Low	Clear
use_srtp	Low	Encrypted
heartbeat	Low	Encrypted
alpn	Medium	Encrypted
status_request_v2	Low	Encrypted
signed_certificate_timestamp	?	?
SessionTicket TLS	Medium	Encrypted/Clear ***
renegotiation_info	Low	Encrypted

* The server may need these in the clear but the client would prefer them encrypted.

*** The SessionTicket must appear in the clear when resuming but can be encrypted when being set up.

The general principle here is that things should be encrypted where possible; extensions generally are proposed to be in the Clear part of handshake only if it seems they must be there to make the rest of the handshake work. The things that seem problematic are those which leak information about the client's dynamic state (as opposed to implementation fingerprinting) but are potentially needed by the server for ciphersuite selection. These are labelled "Server Chooses" in this table.

It's possible we can make life somewhat simpler by deprecating some unused extensions, but based on the table above, it looks like the extensions that make life complicated are not the ones that can be easily removed.

[Appendix C](#). Non-recommended Flows

The flow below is the only one in which we don't protect client

extensions. It complicates things and doesn't seem to meet our goals in terms of protecting that sort of information from traffic analysis.

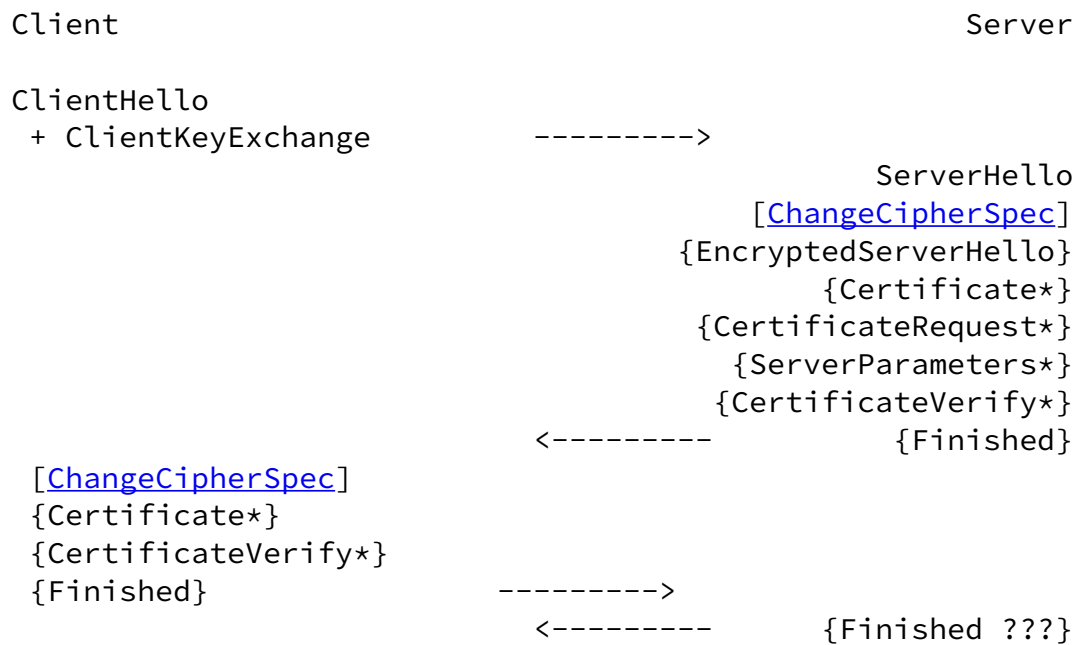


Figure 6: Handshake With Known Group

Author's Address

Eric Rescorla
Mozilla
2064 Edgewood Drive
Palo Alto, CA 94303
USA

Phone: +1 650 678 2350
Email: ekr@rtfm.com

