CENTRE *for* DOCTORAL TRAINING *in*
CYBER SECURITY

CDT Technical Paper

05/16

On Downgrade Attacks in the TLS Protocol

Eman Alashwali

On Downgrade Attacks in the TLS Protocol

Eman Alashwali
Supervised by: Prof. Cas Cremers

University of Oxford
eman.alashwali@cs.ox.ac.uk

Abstract. The TLS protocol was designed to support various versions and ciphersuites. This provides a high level of agility and backward compatibility. At the same time, it opens doors for so-called downgrade attacks. Downgrade attack is a specific type of attacks that result in forcing two communicating parties to use weaker configurations than the ones they wished to use. In this paper, we explore downgrade attacks in TLS. We revisit TLS 1.2, and summarise the major changes in the upcoming standard TLS 1.3 based on the latest draft (revision 14)[1]. We summarise notable examples of TLS downgrade attacks. Finally, we analyse TLS 1.3 protection mechanisms when a cryptographic primitive is broken. Our analysis adds clarity over the existing literature and shows the following: first, weak DHE parameters alone can not allow an attacker to complete a handshake with downgraded ciphersuite, due to a signature over the handshake transcript. Second, compromised digital signatures would allow successful version downgrade that leverages TLS 1.2 attacks (e.g. Logjam) but would not help the attacker to make both ends compute the same keys, due to disparity in the server's nonce that is used as input in the Key Derivation Function (KDF). Finally, weak hash functions would allow successful completion of the handshake despite different transcripts at each end-point, without the attacker knowing the key.

Keywords: tls; downgrade attacks; protocols; security.

1 Introduction

Transport Layer Security (TLS) is probably the most important security protocol used to date, because it is the main security protocol for the internet. The primary goal of TLS is to provide a secure channel between two communicating parties [1], typically, client and server. TLS aims to provide confidentiality, integrity and (peer entity) authentication [1][2]. Confidentiality means that any data that travels over the secure channel is invisible to observers. Integrity means that the data can not be modified and received as it was sent. Finally, peer entity authentication means that at least one of the communicating parties can verify that the other party's identity is the intended one [2]. In TLS, typically, the client

[1] IETF released revision 15 at the end of August 2016. However, this paper is based on revision 14 as it was the latest one when we started the project in July.

authenticates the server, e.g. through a public-key certificate, and optionally, the server authenticates the client. Without authenticating one of the parties, neither integrity nor confidentiality can provide much security [2]. To illustrate, if a client is communicating with an impersonated server using TLS, the client will create a shared secret with the impersonated server (i.e. the attacker), and will encrypt the data using the attacker's key, hence, the adversary will be able to decrypt the encrypted data. Confidentiality is meaningless here without authentication.

Due to the impact of TLS almost on every internet user, TLS has received a lot of attention from both academia and industry. In the last couple of years, researchers showed several concrete examples of a specific type of attacks known as "downgrade attacks". In TLS, these attacks are a result of supporting old and deprecated TLS versions or cryptographic primitives (such as export-grade[2] ciphers) either by clients, or servers, or both. In this report, we explore downgrade attacks and downgrade protection mechanisms.

This report is organised as follows: in section 2, we revisit TLS 1.2, then, we summarise the major changes in TLS 1.3 based on the latest draft (revision 14), and we summarise how different versions of TLS will interact with each other. In section 3, we define downgrade attacks and summarise two notable recent examples: FREAK [4] and Logjam [5]. In section 4, we present the latest protection mechanisms for the current version, TLS 1.2, and the upcoming TLS 1.3. In section 5, we analyse downgrade protection mechanisms under broken or weakened cryptographic primitives assumptions. We propose solutions in section 6. Finally, in section 7, we conclude.

2 SSL/TLS

2.1 Historical Overview

TLS was originally designed by Netscape with the name Secure Socket Layer (SSL). The first version, SSL 1.0, completed in 1994 [6]. However, TLS 1.0 was never released. By the end of 1994, Netscape shipped SSL 2.0 with Netscape Navigator (a browser developed by Netscape) [6]. In 1995, SSL released SSL 3.0 which addressed weaknesses in SSL 2.0 [6]. Although SSL belongs to Netscape, the web community have contributed to the development of the first three versions [6]. In 1996, the responsibility of SSL development has been shifted to the Internet Engineering Task Force (IETF). After that, it has been decided to change the protocol name from SSL to Transport Layer Security (TLS). In 1999, IETF released the final version of TLS 1.0 specifications, based on its predecessor SSL 3.0 with a few changes [7]. Since then, almost every browser such as Netscape Navigator, Microsoft Internet Explorer, etc. has adopted the

[2] Export-grade ciphers are weak ciphers such as 512-bit RSA. They were required to be used in products (including software) shipped to countries outside the United States (US). This was due to a law mandated by US government that was effective until the late 90s, because US considered cryptography as an "Auxiliary Military Equipment"[3].

protocol [6]. TLS versions continued to evolve. In 2006, TLS 1.1 was released [8], followed by TLS 1.2 in 2008 [1]. In 2014, IETF published the first draft of the coming version, TLS 1.3 [9]. As of July 2016, IETF has published revision 14 of the draft [10]. The standard TLS 1.3 has not been released yet.

TLS 1.3 has major changes over its predecessor TLS 1.2. Interestingly, it went through academic scrutiny and many academic published results have contributed to the development of TLS drafts, for example, the work of [11] and [12]. See Figure 1 for SSL/TLS evolutionary time-line.

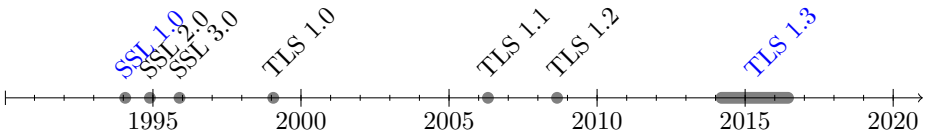

Fig. 1. SSL/TLS timeline. Remark: SSL 1.0 was never released and TLS 1.3 is still work-in-progress and has not been released yet (both marked in blue).

2.2 TLS 1.2

In this section, we will describe the currently deployed version of TLS, TLS 1.2. We will briefly explain the content and purpose of every handshake message exchanged between two communicating parties based on the IETF TLS 1.2 specifications [1]. The communicating parties are client (initiator I) and server (responder R). We adopt the message name conventions from TLS 1.2 standard [1]. The message name will be preceded by the message direction as follows: from $\rightarrow$ to. Finally, optional messages will be appended by (*) symbol. Figure 2 illustrates TLS (Elliptic Curve) Diffie-Hellman ((EC)DHE) full handshake and Figure 3 shows TLS RSA full handshake. The Figures include mandatory and optional messages but not necessarily all optional parameters.

1. **client $\rightarrow$ server: ClientHello.** This message is sent when a client wants to initiate a connection with a server. It can also be sent by a client either during a connection to renegotiate the session's security parameters or upon receiving a HelloRequest message from a server at any time. ClientHello message contains the following parameters:
   - Client version ($vmax_I$): the maximum version of TLS that the client wants to use for the session. For TLS 1.2, this value is: {3,3}.
   - Client's random value (n_I): a random value generated by the client.
   - Session identifier ($session_{\mathrm{ID}}$): a number that identifies an already established session between the client and a server so that the client can request a previous session's security parameters from a server's cache using the $session_{\mathrm{ID}}$ without performing a full handshake. This is known as "session resumption". If the client's $session_{\mathrm{ID}}$ value is empty, this

means that either there is no $session_{\text{ID}}$ available or the client wants to use new security parameters for this session.
- Ciphersuite ($[a_1, ..., a_n]$): a list of ciphersuites that the client supports ordered by preference (the most preferred first). Ciphersuite is a string (see Appendix A for full description of ciphersuite name convention) that defines cryptographic parameters including key-exchange algorithm, symmetric-key encryption algorithm and its key length, and MAC algorithm. In case of session resumption (i.e. $session_{\text{ID}}$ is not empty), the ciphersuite of the resumed session must be used in this field.
- Compression methods ($[c_1, ..., c_n]$): a list of compression methods that the client supports ordered by preference (the most preferred first). In case of session resumption, the compression algorithms of the resumed session must be used in this field. The client and server must agree on a compression method. However, the client can offer a "CompressionMethod.null".
- Extensions* ($[e_1, ..., e_n]$): contains any additional functionalities that the client can request. For example, there is a "signature_algorithms" extension which allows the client to specify hash and digital signature algorithms that the client is welling to use for verifying the server's certificate and the ServerKeyExchange message that will be described next.

2. **server → client: ServerHello.** This message is sent in response to a ClientHello message when the server finds a matching set of algorithms to the client's offered ones in the ClientHello. If no match was found, the server will respond to the ClientHello with a "handshake failure" alert. The ServerHello message consists of the following parameters:
   - Server version (v_R): the highest commonly supported version by the client and server.
   - Server random value (n_R): a random value generated by the server.
   - Session id ($session_{ID}$): represents identity of the session of this connection. If the client specified a $session_{ID}$ in the ClientHello, and the server finds a matching ID in its sessions cache and is willing to resume the session, the server will use the client's $session_{ID}$. This means that the same ciphersuite and compression method of the cached session will be used and the two parties will proceed to the Finished message without performing a complete handshake. This is known as "session resumption". Otherwise, if the server decided to start a new session or can not resume the cached session, it will send a new $session_{ID}$.
   - Ciphersuite (a_R): the selected cipher suite that the server selected from the client's proposed list. In case of session resumption, this field contains the resumed session's value for this field.
   - Compression method (c_R): the selected compression method that the server selected from the client's proposed list. In case of session resumption, this field contains the resumed session's value for this field.

- Extensions* ($[e_1, ..., e_n]$): a list of extensions requested by the client in the ClientHello and supported by the server. The server can only include extensions that it supports and that were requested in the ClientHello extensions field. If the server included extensions that were not in the ClientHello, the client will send "unsupported_extension" fatal alert and abort the handshake.

3. **server → client: ServerCertificate.** This message contains the server's certificate. It is sent when the chosen key-exchange algorithm in the ServerHello requires certificate-based authentication (i.e. all key-exchange algorithms that are supported by TLS 1.2 except DH_anon). The server's certificate must be compatible with the chosen key-exchange algorithm as well as with the signature and hash algorithms in the "signature_algorithms" extension provided by the client.

4. **server → client: ServerKeyExchange*.** This message is sent if the ServerCertificate does not contain all the data that allow the client to compute the pre-master secret. In other words, it is needed with the following key-exchange algorithms (DHE_DSS, DHE_RSA, DH_anon) and is illegal with the following key-exchange algorithms (RSA, DH_DSS, DH_RSA). ServerKeyExchange contains the server's key exchange parameters. In DHE key-exchange algorithm, these parameters are: the prime p, the generator g, the server's DHE public value $g^b \bmod p$ (for short, in the rest of the paper, we denote the DHE client and server public values by g^a, g^b without $mod\ p$ but it is implicitly there). In non-anonymous DHE, these parameters are hashed along with the client's and server's nonces (n_I, n_R, p, g, g^b) and sent in a digitally signed format. If the client specified a signature and hash algorithms in the "signature_algorithms" extension, then the specified algorithms must be used. The "signature_algorithms" must be compatible with the key type in the server's certificate.

5. **server → client: CertificateRequest*.** This message is optionally sent from a "non-anonymous" server to the client to request the client's certificate. It contains the following parameters:
   - Certificate types $[cert_1, ..., cert_n]$: list of types of certificates that are acceptable by the server. e.g. rsa_sign, dss_sign, etc.
   - Signature algorithms $[hs_1, ..., hs_n]$: list of hash and signature algorithms that are acceptable by the server.
   - Certificate authorities $[ca_1, ..., ca_n]$: list of certificate authorities that are acceptable by the server.

6. **server → client: ServerHelloDone.** It indicates that the server finished its part of the key-exchange and the client can proceed its part. Upon receiving this message, the client should verify the server's certificate and also verify that the the server's selected parameters that are provided in the ServerHello are supported by the client.

7. **client → server: ClientCertificate*.** This message contains the client's certificate $cert_I$. It is sent if the server requested a client certificate. If there is no certificate available, the client must respond with an empty certificate message. ClientCertificate message is used by the server in the CertificateVerify for optional signature-based client authentication or to compute the pre-master secret if the key-exchange is static DH.

8. **client → server: ClientKeyExchange.** This message is used to set the pre-master secret. The content depends on the key-exchange algorithm as follows: first, if the key-exchange algorithm is RSA, the client encrypts the pre-master secret pms using the server's long-term RSA public-key pk_R. The pre-master secret includes (in addition to random bytes) the TLS version offered by the client (not the selected version by the server). The version number is included to be checked by the server as a roll-back attack detection technique. Second, if the key-exchange algorithm is DHE, the client sends its DHE public value g^a to allow the server to compute the shared DHE secret-key g^{ab}. In the case of fixed DH key, the client sends its DH public value in the client's certificate and the ClientKeyExchange value must be empty.

9. **client → server: ClientCertificateVerify*.** This message is sent if a ClientCertificate message with signing capabilities was sent. It is used to provide verification of the client's Certificate. It includes a signature over a hash of all handshake messages sent and received (log_1) starting from the ClientHello and up to, but not including this message (ClientCertificateVerify). The hash and signature algorithms must be specified by the server in the CertificateRequest message and must be compatible with the client's certificate key.

10. **client → server: ClientFinished.** This message is sent after a ChangeCipherSpec message which must be sent before sending the Finished message. (Remark: ChangeCipherSpec is not a handshake message and is not included in the Finished hash computations). The Finished message is protected with the just negotiated algorithms and keys. It is an encrypted message. It verifies the integrity of the key-exchange messages. This message needs to be verified by the server (i.e. verify the MAC). It contains a Keyed Message Authentication Code (HMAC) over a hash of all handshake messages sent and received [3] ($logs$) starting from the ClientHello message up to, but not including, the Finished message (log_3). The HMAC is computed using the negotiated master-secret.

[3] Throughout the paper, we will use the term *log* or transcript interchangeably to refer to the concatenation of handshake messages sent or received up to and including the current message where the *log* is marked in the illustration figures. We adopted the term *log* from [12].

11. **server → client: ServerFinished.** This message is similar to ClientFinished but sent from the server to the client after the server receives and verifies the ClientFinished message. This message (i.e. the MAC) must be verified by the client too.

After both parties have received and verified the finished messages, they can start exchanging data encrypted and authenticated with the negotiated keys.

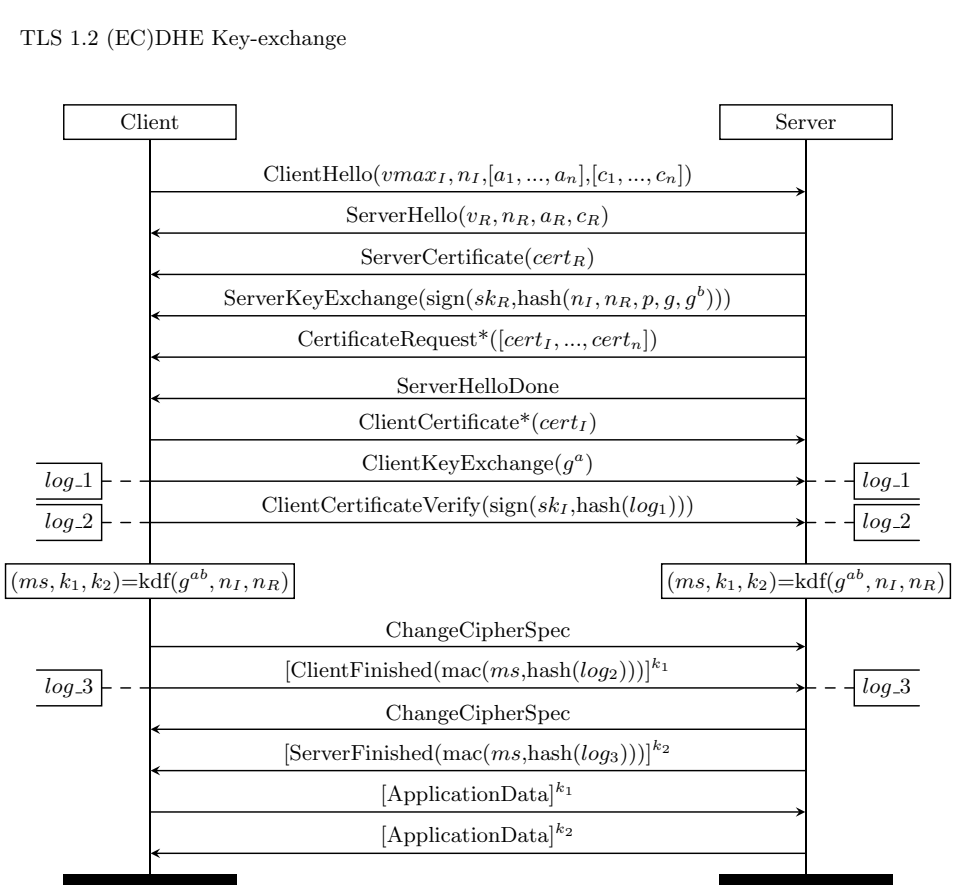

Fig. 2. Message sequence diagram for TLS 1.2 (EC)DHE full handshake

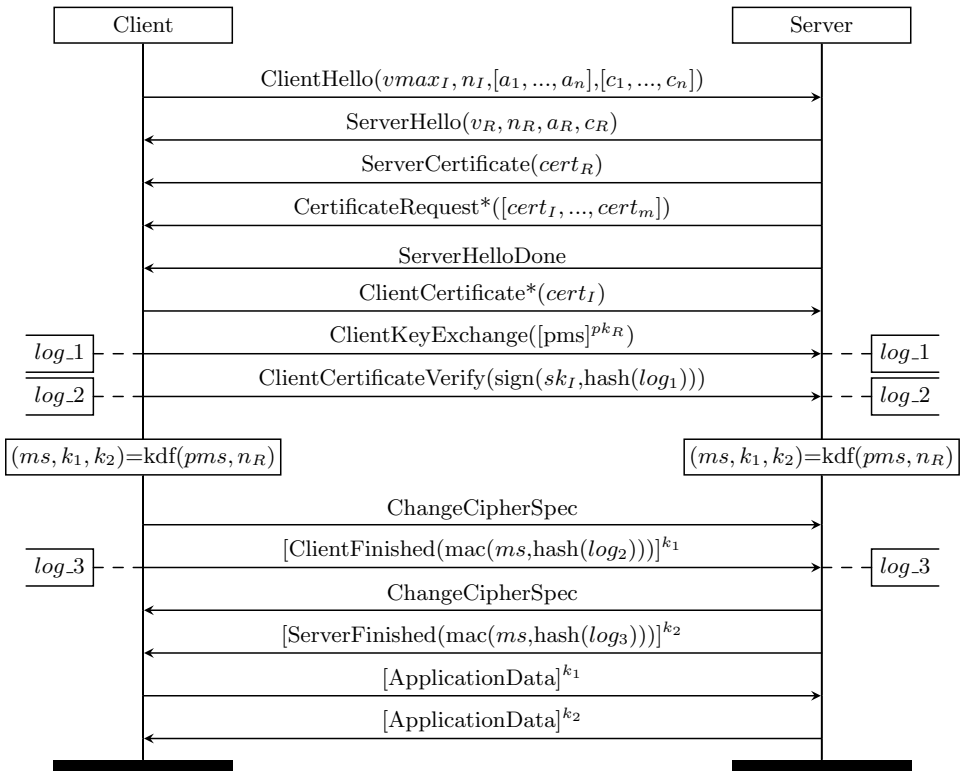

Fig. 3. Message sequence diagram for TLS 1.2 RSA full handshake

2.3 TLS 1.3, Major Changes

In this section we list the major changes in TLS 1.3 based on the latest draft (as of July 2016, the latest draft was revision 14 [10]).

Changes to the Protocol Messages. We list the messages that have major changes or are newly introduced and briefly describe them (see Figure 4 for the TLS 1.3 server-authenticated handshake message sequence diagram):

– **client → server: ClientHello.** The major changes in this message can be summarised as follows: first, unlike TLS 1.2 where the extensions field is optional and the server must accept ClientHello messages with or without extensions, in TLS 1.3, the extensions field must at least contain a value for

the *KeyShare* or the Pre-Shared Key (*PSK*) extensions. Second, in TLS 1.2, ClientHello can be sent by the client to initiate a connection, or in an existing session to renegotiate the session's security parameters, or in response to HelloRequest coming from the server at any time. In TLS 1.3, the draft specifies two cases in which ClientHello is sent. The first case is similar to previous versions of TLS, i.e. when a client wants to connect to a server. The second case is newly introduced to TLS 1.3 where ClientHello can be sent in response to a HelloRetryRequest. The latter is sent from the server and will be explained below. Upon receiving HelloRetryRequest, the client re-sends the ClientHello with the same values except for a new $KeyShare$ value and removed *EarlyDataIndication* extension value. Third, in TLS 1.3 ClientHello, $session_{ID}$ field is added for backward compatibility only. This field was used for session resumption in TLS 1.2, but in TLS 1.3 session resumption is achieved through PSK using $PSK_{IDENTITY}$ which is derived from a completed handshake's PSK. PSK_{IDENTITY}, similar to $session_{ID}$ in session resumption, allows a client to request a server to use a previous session's security parameters. Servers negotiating TLS 1.3 must ignore $session_{ID}$ field and clients that do not have cached $session_{ID}$ set by pre-TLS 1.3 servers should set the $session_{ID}$ value to zero. Fourth, in TLS 1.3, compression methods field is also included for backward compatibility reasons and a TLS 1.3 client should set this field value to zero (i.e. "null" compression method).

– **server → client: HelloRetryRequest.** This message is newly introduced in TLS 1.3. It is sent when the server does not accept the client's offered groups in the *KeyShare* extension that is included in the ClientHello. HelloRetryRequest contains the following fields: the selected version v_R, the selected ciphersuite a_R, the requested group G_R, and finally the extensions $[e_1, ..., e_n]$. The selected version, selected ciphersuite, and extensions have similar purpose and content as in a normal ServerHello in TLS 1.2. However, there is a fourth field that specifies the server's requested group G_R. The server's requested group must not be already listed in the client's *KeyShare* but it must be supported by the client in the client's *SupportedGroups* extension. Furthermore, HelloRetryRequest must not be sent more than once in a single connection. If any of the aforementioned conditions occurred, the client responds with "handshake_failure" alert. Otherwise, the client will resend the ClientHello after appending the server's requested group G_R in the *KeyShare*.

– **server → client: ServerHello.** There are four major differences in TLS 1.3 ServerHello over the one in TLS 1.2. First, in TLS 1.3, similar to TLS 1.2, ServerHello is sent as a response for ClientHello after the server finds a matching set of algorithms to the client's proposal. But, in TLS 1.3, in addition to that, the server must also agree on the client's proposed groups in the *KeyShare* extension. Second, in TLS 1.3, ServerHello can be sent after a second ClientHello (which was sent as a response to HelloRetryRequest).

Third, similar to the ClinetHello in TLS 1.3, the extensions field in the ServerHello must contain at least one extension, the *KeyShare* or *PSK* while the extensions field was optional in TLS 1.2. Fourth, the last eight bytes of the server's random number n_R are set to a fixed value that indicates the version of TLS that the server received from the client in the ClientHello. This mechanism is added as a version downgrade protection mechanism because the server's random number will get signed by the server in the ServerKeyExchange and this allows the client to verify that the ClientHello version was received correctly and has not been modified. However, this requires a patch for TLS 1.2 client and server implementation.

- **server → client: ServerCertificateVerify.** In versions prior to TLS 1.3, server's certificate is not followed by ServerCertificateVerify. In TLS 1.2, there is only a ClientCertificateVerify which is sent only after an optional client's certificate while in TLS 1.3, the server's certificate is followed by ServerCertificateVerify and it contains a signature over a hash of the hello messages full transcript (log_1 in Figure 4).

Encrypted Handshake Messages. In TLS 1.3, the protocol messages are encrypted as soon as both parties have computed pre-shared-keys, i.e. after the ServerHello. At the server side, the following messages from the server to the client were sent in plaintext in TLS 1.2 while in TLS 1.3 they have become encrypted: CertificateRequest*, Certificate, and CertificateVerify*. On the client side, the following messages have become encrypted in TLS 1.3: Certificate*, and CertificateVerify*.

Disabled Weak Cryptography. In TLS 1.3, the number of supported ciphersuites has been reduced. In draft 14, in the standard track of "server-authenticated (and optionally client-authenticated) cipher suites" only 13 ciphersuites are supported (See Appendix A.4) [10]. More importantly, weak cryptographic algorithms are not supported in TLS 1.3. For example, since draft 2, static RSA and DH are not supported. Since draft 8, MD5 and SHA-224 hashes with signatures are not supported any more. Then in draft 9, SHA-1 with signatures has been deprecated. Since draft 6, the use of RC4 for backward compatibility has been prohibited. In draft 9, DSA is not supported.

Removed Messages From the Handshake. Some messages used in TLS 1.2 do not exist any more in TLS 1.3. These messages are:

- **server → client: ServerKeyExchange.** This message has been removed in TLS 1.3. The DHE group and the server's DHE public value are negotiated in the hello messages.

- **client → sever: ClientKeyExchange.** Similar to ServerKeyExchange above. The client's DHE public value is sent in the hello message.

– **client → sever and server → client: ChangeCipherSpec.** In TLS 1.3, ChangeCipherSpec messages at both directions (from client to server and vice versa) have been removed. In TLS 1.2, ChangeCipherSpec messages were not a handshake message but they were required before the Finished messages.

Including Handshake Transcript (*logs*) in KDFs. In TLS 1.3, the number of hashes computed over the handshake message transcripts i.e. hash(*logs*) are more than that in previous versions. In TLS 1.3, hashes over *logs* are used as input for various functions at various stages. For example, a hash of handshake messages transcript is used as input in the Key Derivation Functions KDF that are used to compute the session keys and the master-key (see Figure 4). This method of including the hash of handshake transcripts in the KDFs is newly introduced in TLS 1.3 and was not used in the KDF of TLS 1.2.

2.4 Version Update Handling in TLS

When a new standard is released, such as the upcoming TLS 1.3, it may take a substantial amount of time until all clients and servers get updated. For this reason, TLS designers have maintained compatibility between versions. SSL 3 and TLS 1.x have compatible ClientHello messages [10]. Basically, TLS version is negotiated in the handshake as follows: the client's maximum supported version $vmax_I$ is proposed in the ClientHello. Then, the server selects the maximum version that is commonly supported by both the client and server. After that, the server sends the selected version v_R in the ServerHello.

We summarise how different TLS versions will interact with the introduction of TLS 1.3. This is especially important for understanding version downgrade attacks. The following summary is based on IETF TLS 1.3 draft 14 (Appendix C. Backward Compatibility) [10].

Old Client Communicating with New Server. When the client's proposed version is less than the server's maximum supported version but higher than the minimum server's supported version, the server will select the highest common version between the client and server. For example, a client that offers TLS 1.2 to a server that supports TLS 1.3; TLS 1.2; and TLS 1.1, the server will select TLS 1.2. However, if the client's proposed version is below the server's minimum supported version, e.g., a client offers TLS 1.0 to a server that supports the following version: TLS 1.3; TLS 1.2; and TLS 1.1, the server will respond with the "protocol_version" alert message and close the connection.

New Client Communicating with Old Server. When the client's proposed version is higher than the server's maximum supported version, the server selects the maximum version it supports (which is older than the client's version). If the client supports and accepts the server's version, the connection will proceeds.

TLS 1.3 Server-authenticated

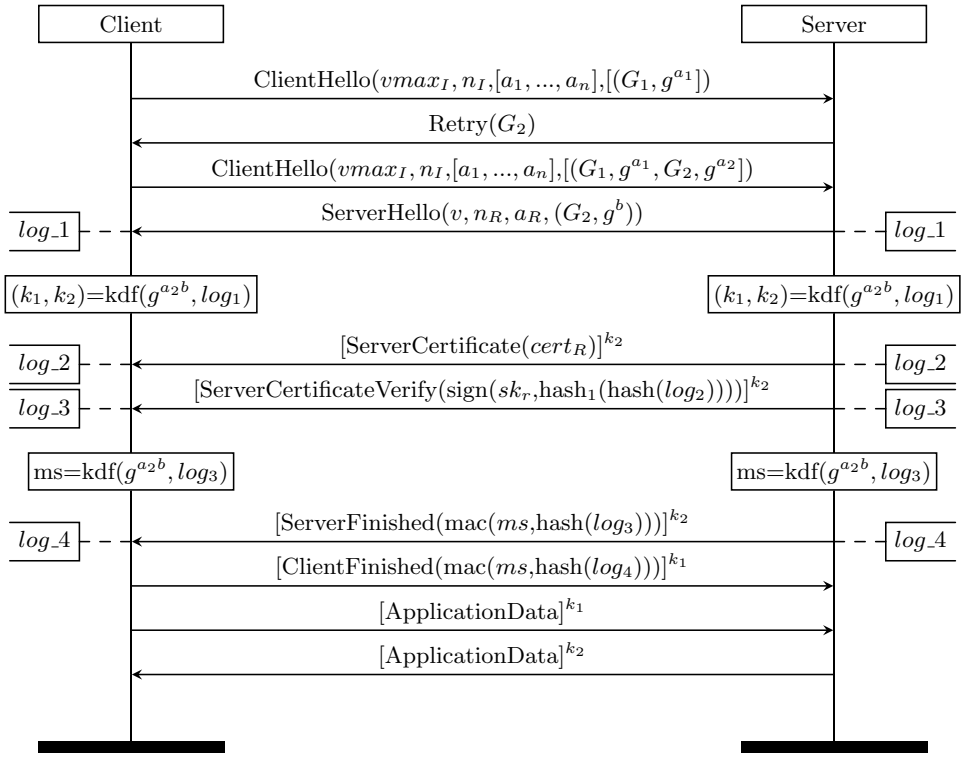

Fig. 4. Message sequence diagram for TLS 1.3 server-authenticated handshake. Reference [12]

Otherwise, if the client does not support the server's version, the client will respond with a "protocol_version" alert and the connection will be closed. For example, when a client that offers TLS 1.3 while the server's maximum supported version is TLS 1.2, the server will respond with TLS 1.2. Then, the client decides whether to proceed or not. It is noted that some improper server implementations will abort the connections if it encountered a client version that is unknown to it (e.g. newer than the server's maximum supported version). Furthermore, "0-RTT data is not compatible with older servers" [10].

3 Downgrade Attacks

In this section, we provide the Bhargavan et al. [12] definitions for downgrade attack and downgrade security. Then, we provide anatomy of downgrade attacks through notable examples, namely, FREAK [4] and Logjam [5]. Finally, we summarise the lessons learned from these two attacks.

3.1 Definitions

Informal Definition. We infer the following informal definition of downgrade attack from [12]:

"[D]owngrade attacks, where an active network adversary interferes with the negotiation, causing honest peers to complete a key exchange, albeit using a mode that is weaker than the one they would have used on their own."

Formal Definition. Before we provide the formal definition, we need to define the notations and terms that are necessary for understanding the downgrade security definition.

- π: a session. Each session holds variables that are initialized to $\perp$ and assigned values only once [12]. The following are examples for session variables:
  - $\pi.cfg$: "initial configuration (including the session role)", i.e. $\pi.cfg.role$ (for short, $\pi.role$);
  - $\pi.uid$: "unique identifier of the session";
  - $\pi.mode$: "negotiated mode (including long-term identities)";
  - $\pi.complete$: "flag set when the session completes successfully".
- $Nego$ [12]: a function that:

  > *maps two configurations with opposite roles picked from the universe of all supported configurations* $\boldsymbol{C}$ *to the protocol mode negotiated (if any) in the absence of active adversaries. Formally, if a session* π *talking to a session* π' *completes, it must be the case that* $\pi.mode = Nego_r(\pi.cfg; \pi'.cfg)$*, where* $Nego_r$ *is an abbreviation defined by case:*
  > $$Nego_r(cfg_r, cfg_{\overline{r}}) \triangleq \begin{cases} Nego(cfg_r, cfg_{\overline{r}}) & when\ r = I \\ Nego(cfg_{\overline{r}}, cfg_r) & when\ r = R. \end{cases}$$

  Example 1 (Example for a function Nego [12]). In TLS, *Nego* would determine the resulting configurations after the peers negotiation. For example, it determines that the protocol version is TLS 1.2; the ciphersuite is DHE-RSA-AES256-GCM-SHA384 with a 2048-bit DH key.

- **Definition 1 (Partnering [12]).** *"Sessions* π *and* π' *are partnered if* $\pi'.role = \overline{\pi.role}$ *(they have opposite roles) and* $\pi'.uid = \pi.uid$*. A session* π *is unpartnered when there is no such* π'*".*

– **Definition 2 (Downgrade-Protection Predicate (DP) [12]).** *"[O]perates on pairs of configurations (...), and that identifies pairs of configurations from which we expect downgrade resilience."*

Example 2 (Example: Downgrade-Protection Predicate (DP) [12]). In SSH, DP defines that the configurations of π must require peer authentication, honest keys, and strong signature algorithms.

Definition 3 (Downgrade Security [12]).

A session π is downgraded when $\pi.complete = true$ and there is a partnered session π' such that $DP(\pi.cfg, \pi'.cfg)$, and $\pi.mode \neq Nego_{\pi.role}(\pi.cfg, \pi'.cfg)$.
The advantage $\boldsymbol{Adv}^{downgrade}_{\Pi,DP,X}(A)$ of A against downgrade security with pre-agreement on X is the probability that, when A terminates after interacting with Π, there exists a session π that either is downgraded or does not pre-agree with a partnered session π' on X. We write $\boldsymbol{Adv}^{downgrade}_{\Pi,\ DP}(A)$ when $X = \{\}$.

3.2 Anatomy of Notable Examples of Downgrade Attacks

In this section, we chose two recently discovered notable examples of downgrade attacks, namely: FREAK [4] and Logjam [5]. We summarise them to illustrate the possible causes and mechanisms by which a successful downgrade attack can take place.

FREAK [4]. FREAK is a result of faulty implementation discovered in OpenSSL, SecureTransport, and other TLS libraries that include code to handle export-grade ciphersuites such as RSA_EXPORT. FREAK can occur when the key-exchange algorithm is RSA and the server supports RSA_EXPORT ciphersuites. At the time of the vulnerability discovery, there were about 25% of HTTPS servers that supported RSA_EXPORT. The attack (illustrated in Figure 5) works as follows: when the client sends a ClientHello with RSA ciphersuites, a MITM modifies the ClientHello to request RSA_EXPORT. The ciphersuite list in the ClientHello should be engineered in such a way that makes the server selects the RSA_EXPORT. A straightforward method is that the attacker makes RSA_-EXPORT the only ciphersuite in the client's ciphersuite list. The server receives the modified ClientHello and believes that RSA_EXPORT is the client's wish. Therefore, the server selects RSA_EXPORT and sends the selected ciphersuite in a ServerHello to the client. The attacker modifies the ServerHello and changes the ciphersuite from RSA_EXPORT to RSA to disguise the modification that he made in the ciphersuite. Then, the attacker forwards the message to the client. After that, the server sends its certificate with strong RSA key (e.g. 2048-bit) and the attacker forwards the certificate as it is to the client. The server sends a ServerKeyExchange containing RSA_EXPORT key (i.e. 512-bit or below) followed by a ServerHelloDone. A key point here is that ServerKeyExchange must

not be sent when the key-exchange algorithm is RSA. The attacker forwards both messages (ServerKeyExchange and ServerHelloDone) sequentially to the client. Upon receiving, the client sends ClientKeyExchange which contains a pre-master secret (pms) encrypted with the the 512-bit RSA key instead of the 2048-bit RSA key that is in the server's certificate. The attacker factors the 512-bit RSA modulus (achievable in hours) and computes the private-key of the 512-bit RSA key. This would allow him to decrypt the pre-master secret that is in the ClientKeyExchange, and as a result, derives the master-secret, sends server's ChangeCipherSpec and forge the ServerFinished MAC. After that, the handshake will complete successfully and the client is not aware of the ciphersuite disparity due to the server's forged Finished MAC. From now on, MITM can read data sent from the client to the server not only for this session but also for any future sessions that use the factored RSA-key and can impersonate the server as well. Many RSA servers use the same RSA key (normally, servers change the key with every restart), so the attacker does not have to factor the key for every session. Figure 5 illustrates the attack.

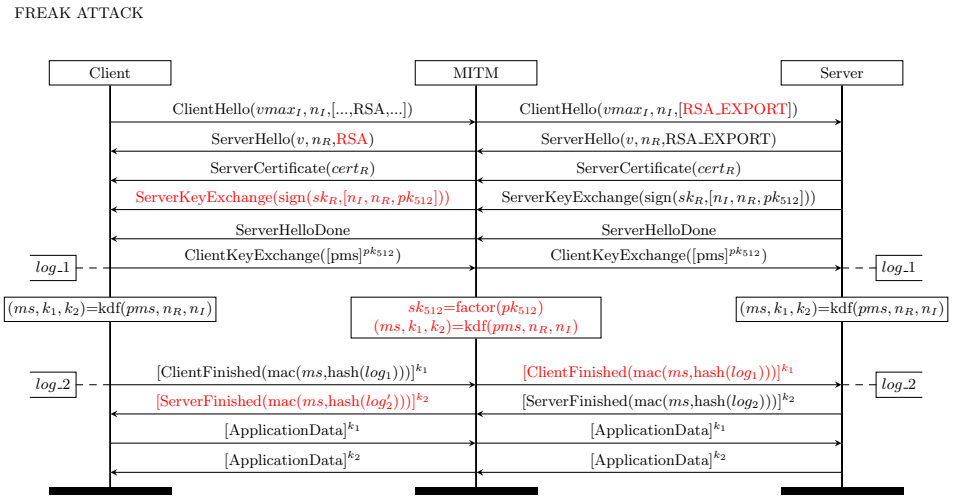

Fig. 5. Illustration for FREAK attack in TLS 1.2.

Logjam [5]. Logjam exploits a protocol design flaw found in TLS 1.2 or below. Logjam can occur when the key-exchange algorithm is DHE and the server supports DH_EXPORT ciphersuites (e.g. 512-bit DHE primes). An internet survey conducted by the authors at the time of the attack discovery shows that 8.4% of HTTPS websites in Alexa's top 1M websites support DHE_EXPORT

while 4.9% of IPv4 HTTPS with "browser trusted certificate" servers also do so. The survey also revealed about high percentage of using the same primes where over 92% of the servers in both datasets (Alexa's top 1M and IPv4 HTTPS sites) use only 2 primes. This will save time for the attacker in computing the key. The attack works as follows (illustrated in Figure 6): the client sends a ClientHello message with DHE ciphersuites. A MITM modifies the ClientHello to request DHE_EXPORT. The server selects a DHE_EXPORT ciphersuite and responds with ServerHello that contains the selected ciphersuite. The attacker modifies the ciphersuite in the ServerHello from DHE_EXPORT to DHE to disguise the modification from the client and forwards the ServerHello to the client. The server sends its Certificate, followed by ServerKeyExchange message and the attacker forwards them to the client. The ServerKeyExchange message contains a signature that includes the server's chosen DHE parameters including the client's and server's nonces n_I, n_R, the weak prime, p_{512}, the group g, and the server's public DHE value, g^b. However, the signature does not include enough parameters, e.g. the full ciphersuite. After receiving this ServerKeyExchange, the client sends its ClientKeyExchange to the attacker. After that, because the keys were computed using a weak prime, the attacker can compute the server's secret key (the exponent b) in about an hour using Number Field Sieve (NFS). Consequently, this allows the attacker to compute the shared secret g^{ab}, and as a result, computes the master secret and forge the MAC in the server's Finished message. Thus, successfully completing the handshake with different transcripts at both ends. The main challenge in this attack is computing the shared secret g^{ab} before the handshake connection time-out. The computation takes about 70 seconds. It may not work with some browsers but the authors of [5] suggested several cases where this attack can still work such as non-browser clients. In case of normal browsers that employ TLS false start, this attack allows the attacker to decrypt the false start data even if the handshake timed out. See Figure 6 for illustration.

Lessons Learned. The main two lessons we learned from these two attacks are the following: first, clients and servers security both are integral. From the aforementioned attacks, we see that, even when clients are updated and do not support export-grade ciphers, by just having servers that support export-grade ciphersuite, the security of TLS session falls apart. Second, there is a need for integrity check for the handshake message transcripts at both communicating parties before the finished messages. Although the finished MACs placed to provide downgrade protection to detect any disparity in the transcripts between the two parties, however, the MACs check seems too late. The selected ciphersuite needs to be included in the server's signature so that the client can verify it. This point has already been considered in TLS 1.3 as we will elaborate more on TLS 1.3 downgrade protection mechanisms next.

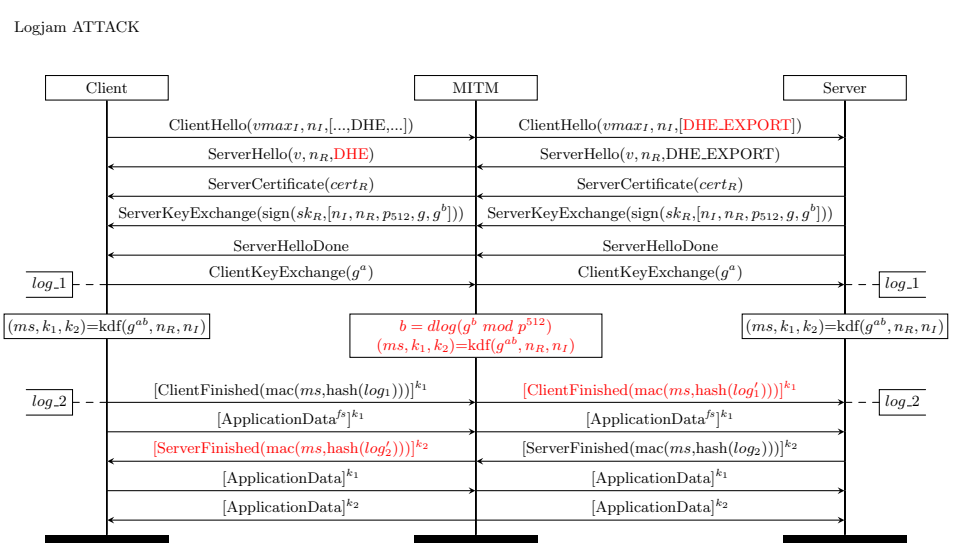

Fig. 6. Illustration for Logjam attack in TLS 1.2 (we complemented the message sequence diagram in [5] to show how MITM can get both parties to have the same key). For the original diagram, see [5]

4 Downgrade Protection in TLS

In this section, we summarise the available downgrade protection mechanisms in TLS 1.2. Then, we explain the reasons of their failure to protect against downgrade attacks. Finally, we will summarise the proposed downgrade protection mechanisms for TLS 1.3 based on the latest draft (revision 14).

4.1 Downgrade Protection Mechanisms in TLS 1.2

In TLS 1.2, downgrade protection is meant to be achieved mainly by two countermeasures:

Digital Signature. In TLS 1.2, after both parties exchange the Hello messages, and after the server sends its public-key certificate, the server sends ServerKeyExchange message which contains a digital signature over a hash of the following parameters: the client's nonce (n_I), the server's nonces (n_R), the selected group parameters (p,g) and the server's DH public value (g^b). The signature allows the client to verify that the nonces have not been modified and authenticates the server's selected DHE parameters and its DHE public value. However, the signature does not include the full transcript of the Hello messages. In particular, it does not include the selected protocol version v, nor

the selected ciphersuite a_R to allow the client to verify the integrity of the client's offer and the server's selected version and ciphersuite. Excluding these two parameters in the signature delays verifying the Hello messages' integrity till the very end of the handshake in the Finished MACs, which we will explain next. However, relying on the Finished MACs for downgrade protection turned to be insufficient. As illustrated previously in Logjam [5] and FREAK [4] attacks, a MITM attacker can downgrade the ciphersuite to export-grade ciphersuite, then, use the export-grade ciphersuite weak algorithm to break the master-key, and then forge the Finished MACs. As a result, the disparity between the client's offered ciphersuite and the server's selected one remains undiscovered.

Finished MACs. At the end of TLS 1.2 handshake, each party sends a Finished message. In this message, a MAC over a hash of the handshake messages is computed using the master-secret. Upon receiving, each party must verify the MAC to ensure the integrity and authenticity of the messages exchanged. After receiving and verifying the MACs, each party can use the negotiated keys and ciphers to send data. The Finished MACs are supposed to protect against downgrade attacks as any modification in the handshake messages by the attacker will result in disparity in the MACs, therefore, any downgrade will be detected. However, the main problem in this protection mechanism is that it is computed at the very end of the handshake. Therefore, an active MITM attacker can downgrade the ciphersuite and reconstruct the master-secret, then, forge the MACs to disguise the downgrade. This attack scenario used in Logjam [5] and FREAK [4].

4.2 Downgrade Protection Mechanisms in TLS 1.3

From the previous section, it is obvious that the downgrade protection mechanisms in TLS 1.2 are not sufficient to prevent downgrade attacks. Therefore, TLS 1.3 draft 14 included the following mechanisms (some of them already exist in TLS 1.2 but have been improved):

Signature Over the Full Handshake Transcript. One of the main problems that allowed attacks such as Logjam [5] and FREAK [4] to succeed is that the signature in the ServerKeyExchange does not include the ciphersuite which allows an attacker to alter the ciphersuite to a weaker one and use the weak ciphersuite to recover the master-secret before the Finished MAC checks the transcript integrity at both ends. To overcome this problem, TLS 1.3 proposes a new message ServerCertificateVerify (improved ServerKeyExchange). ServerCertificateVerify contains a signature over a hash of the full transcript (up to and including the server's certificate, i.e. log_2 in Figure 4) using the server's long-term key. This message is sent from the server to the client after the ServerCertificate message.

Fixed Number in the Server's Nonce. As a version downgrade protection mechanism above the signature, [12] proposed including the maximum version number supported by the server in the server's nonce in all versions. A somewhat modified technique included in draft 14 as follows: the last eight bytes of the server's nonce is set to a fixed number. This fixed number will indicate to the client the protocol version that the server has received from the client. To illustrate, if the server received TLS 1.2 or below, it sets the last eight bytes of the server's nonce n_R to the following value: 44 4F 57 4E 47 52 44 01, and if it received TLS 1.1 or below, it sets the bytes to the following value: 44 4F 57 4E 47 52 44 00. Upon receiving the ServerHello, the client gets to check this number. If TLS 1.3 client received any of the previously mentioned numbers, this indicates that the version has been downgraded from TLS 1.3 to a lower version and the client must close the handshake. The fixed number can get modified by an attacker to disguise a downgrade attack, however, the nonce is included in the ServerKeyExchange messages which is signed by the server and this provides guarantee that the client will detect any change occurred in the server's nonce. Also, n_R is included in the KDF. Therefore, if n_R got modified, both ends will fail in generating shared session keys.

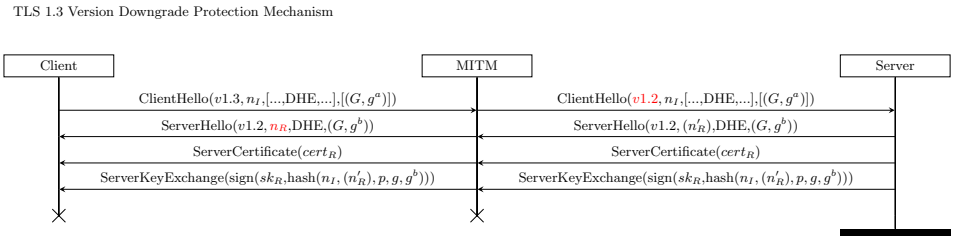

Fig. 7. Illustration for version downgrade protection mechanism. The last 8 bytes of n_R are set to a fixed number. If a MITM modified the new server's random value after the eight bytes have been set, i.e. modified it from n'_R to n_R, the verification of the server's signature will fail and the client will detect the modification unless the attacker is capable of forging the signature.

Hash Handshake Messages (i.e. Hash the Transcript or Hash the *log*). In TLS 1.3, the handshake transcripts are used as input to functions at various stages of the protocol run. This helps in detecting any disparity in the messages sent or received at stage earlier than the Finished MAC. We use the term *log* that was originally introduced by [12] which referes to the concatenation of messages up to and including a certain point [13]. In the following, we describe the *logs* used in TLS 1.3 (based on draft 10):

- Log_1: includes the Hello messages. It is used as input for the KDF as follows: kdf(g^{ab}, log_1). KDF is computed at both ends. Therefore, if there is disparity in log_1, the client and server will not succeed in agreeing in session keys.
- Log_2: includes all earlier handshake messages including the ServerCertificate. Log_2 is double hashed and used as input for the digital signature in the ServerCertificateVerify as follows: [ServerCertificateVerify(sign(sk_r,hash$_1$(hash(log_2)))]^{k_2}, where the outer hash (hash$_1$) uses one of the "signature_algorithms" hashes and the inner hash uses the ciphersuite hash.
- Log_3: includes all earlier handshake messages including the ServerCertificate and the ServerCertificateVerify. Log_3 is used as input for two functions. First, as input for KDF to compute the master-secret. As stated earlier, if the clinet and server have different log_3, they will not succeed in computing shared master-secret. Second, log_3 is used in the ServerFinished as follows: [ServerFinished(mac(ms,hash(log_3)))]^{k_2}.
- Log_4: includes all earlier handshake messages including the ServerFinished. Log_4 is used as input for the ClientFinished as follows: [ClientFinished(mac(ms,hash(log_4)))]^{k_1}

  See Figure 4 for illustration of *logs* points.

Finished MACs. Similar to TLS 1.2, TLS 1.3 still compute the Finished MAC. The main difference here is that, in TLS 1.3 there are several downgrade protection mechanisms above the Finished MACs while in TLS 1.2, Finished MACs are almost the only downgrade protection mechanism.

5 Downgrade Protection With Broken or Weakened Primitives

Downgrade protection relies on strong cryptographic primitives. In this section, our aim is to add clarity over the existing literature. We want to articulate the effect of broken primitives on downgrade protection. Our approach is to present an example for a trace of TLS 1.3 protocol run under the assumption of broken primitive and an attacker trying to perform downgrade attack. Of course, we assume at least client or server supporting weaker versions and ciphersuites for reasons such as backward compatibility or for agility. This represents what happens normally in real-world or might happen in the future for TLS 1.3 ciphersuites when the currently strong primitives get broken due to advances in cryptanalysis. Precise assumptions for each case will be provided in every example.

5.1 Downgrade Attack Levels

First, it is convenient to classify downgrade attacks in terms of the levels they can take place in:

- **Version Downgrade.** In this class of downgrade, the attacker forces both parties to use an older TLS version than the one they would choose on their own with the aim of using the older version weaknesses to achieve his goals.
- **Ciphersuite Downgrade.** This class of downgrade does not downgrade the version. The attacker forces both parties to negotiate a specific cipher that has some weaknesses (e.g. RC4 cipher, export-grade cipher, RSA key-exchange, etc.) with the aim of using their weaknesses to achieve his goals like decrypting traffic, decrypting the master secret, or preventing forward-secrecy.
- **Protection Mechanism Downgrade.** In this class of downgrade, the attacker downgrades the cryptographic primitive that is used in a downgrade protection mechanism. For example, an attacker forces the client to offer a weak hash that has collision in the "signature_algorithms". In this example, the attacker did not downgrade the version nor the ciphersuite, but the digital signature.

5.2 Downgrade Attacker Motivation

We can classify downgrade attacker motivations as follows:

1. When downgrade allows specific types of attacks that can not be performed in the version and ciphersuite that both parties would negotiate without interference. For example, FREAK [4] and Logjam [5] can not be performed in TLS 1.3 as the flaws has been fixed (at least the protocol flaws has been fixed, and hopefully the implementation flaws too). To illustrate, FREAK is specific to RSA ciphersuites which are not supported anymore in TLS 1.3 and Logjam needs to exploit a protocol design flaw that has been fixed in TLS 1.3 (mainly, the server signature).
2. When downgrade helps the attacker to reduce the security level to a weaker but not broken cipher to achieve future attacks when a cipher gets broken. For example, forcing non-forward secrecy ciphersuites which are not supported in TLS 1.3 but supported in TLS 1.2. With non-forward secrecy, e.g., with RSA key-exchange, an attacker can collect encrypted traffic and once the currently deployed keys get broken (e.g. RSA 2048), the attacker can decrypt all the previously collected traffic. This would not be possible with forward-secrecy key-exchange such as (EC)DHE because every session key is unique for that session.
3. When downgrade is just an instance of an attack that can be performed in the newer version as well. For example, in the case of compromised signature key, the attacker can impersonate the client to the server and vice versa by exchanging his own DHE keys as if they are the other parties' keys. This impersonation attack can be performed in the newer version TLS 1.3 under some assumptions e.g. broken digital signature scheme. Similarly, he can perform version downgrade to TLS 1.2 and perform the impersonation next.

5.3 Cryptographic Primitives in Downgrade Protection

In protocols, cryptographic primitives are the cryptographic algorithms that can be plugged into the protocol to provide some security goals. In TLS 1.3, there are various primitives that play vital role in providing downgrade protection:

- Digital Signatures Scheme
- Key-exchange Algorithm
- Hash Functions
- Message Authentication Codes (MACs)
- Encryption Algorithms
- Other Primitives, e.g. KDF.

In the next sections, we will analyse three major primitives in relation to downgrade protection, namely, digital signature, key-exchange algorithm, and hash function. We leave a complete analysis for all the primitives to future work due to time limitations. We will provide a brief description of the primitive and its security. Then, we will specify its usage in TLS 1.3 in relation to downgrade protection. After that, we will provide an example of a downgrade attack that illustrates the consequences of a broken primitive on downgrade protection. Our analysis assume a certificate-based server-authenticated setting.

5.4 Digital Signatures Scheme

Digital signatures are based on public-key cryptography. They are used to provide message integrity and authenticity. A signer signs a message using his private-key to generate a signature. The message is sent along with its signature to the receiver. The receiver who has the right public-key that corresponds to the private-key that was used to sign the message can verify the message. If the verification succeeded, this means two things: first, the message was produced by the intended signer whose identity is associated with the public-key. Second, the signed message has not been changed. Otherwise, if the verification failed, this means that either the message was signed with the wrong key that does not corresponds to the public-key, or that the message has been modified after the signer has signed it.

In TLS 1.3, RSA and ECDSA are the two supported signature schemes [10]. It is worth mentioning that, currently, in TLS 1.2, the dominant digital signature scheme is RSA.

The Security of Digital Signature. A digital signature scheme is secure if the signature is unforgeable. That is, given the public-key only, it should be infeasible for an adversary to generate a valid signature for a message.

Digital Signature Usage in TLS 1.3. In TLS 1.3, digital signatures are used in CertificateVerify messages when certificate-based authentication is used. There is no CertificateVerify with PSK or (EC)DHE-PSK ciphersuites. CertificateVerify messages can be sent by the server or the client or both after they send their certificates and before the Finished messages. The server CertificateVerify is a newly introduced message in TLS 1.3 which did not exist in previous versions. It is a replacement for the ServerKeyExchange message which was in TLS 1.2. The signature in CertificateVerify is used as a proof of possession of the private-key by the party that sent the certificate [10]. It also provides integrity for the handshake messages transcript up to the CertificateVerify point [10]. It is worth noting that if CertificateVerify was sent by the server, the signature algorithm that is used in the server's CertificateVerify must be compatible with the client's offered signature algorithms in "signature_-algorithms" extension [10]. And if the CertificateVerify was sent by the client, the signature algorithm used in CertificateVerify must be compatible with the "supported_signature_algorithms" field in CertificateRequest that is sent by the server [10]. In both cases, the signature algorithm must be compatible with the key type in the certificate [10]. In CertificateVerify, the signer uses its private key to sign.

For the rest of our analysis, we will consider an authenticated-server case only, i.e. when CertificateVerify is sent by the server. This is because server authentication (i.e. unilateral authentication) is the most commonly deployed method currently. In the server's CertificateVerify, the signature includes a double hash for log_2 (the handshake messages transcript) which covers the ClientHello(s), HelloRetryRequest (if it was sent), ServerHello and ServerCertificate messages. The outer hash ($hash_1$) is the hash specified in the "signature_algorithms" and the inner hash is specified in the ciphersuite. For illustration, the following shows the ServerCertificateVerify message (see Figure 4): [ServerCertificateVerify(sign(sk_r,hash_1(hash(log_2)))]^{k_2}

Assumptions. We assume a client and a server that both support TLS 1.3 and the patched version of TLS 1.2 that implements the version downgrade protection mechanism but also support export-grade ciphers for backward compatibility. The server's long-term signature key is compromised, e.g. factorable RSA key. The attacker is an active MITM that has full control over the network traffic. The attacker can also perform the discrete log computation for weak DHE primes (e.g. 512-bit primes) in real-time. The attacker goal is to downgrade the TLS version form TLS 1.3 to TLS 1.2 to leverage an attack that can be performed in the older version only.

The Consequences of Compromised Signature Key on TLS Downgrade Protection. A compromised signature key would allow the attacker to modify the transcript and issue a valid signature over the modified transcript (log_2) to hide the modification. This leads to a successful version downgrade attack as we will illustrate in the coming example.

In this example, the attacker will succeed in forcing the client and server to negotiate TLS 1.2 to perform an attack specific to TLS 1.2, though, both of the client and server support TLS 1.3. As depicted in Figure 6, the client offers TLS 1.3 and strong DHE groups. The attacker modifies the ClientHello to a crafted ClientHello that appears as TLS 1.2 with DHE_EXPORT cipher. In particular, the attacker changes the proposed version $vmax_I$ in the ClientHello from TLS 1.3 to TLS 1.2 and changes the client's proposed ciphersuite list $[a1, ..., a_n]$ to [DHE_EXPORT]. In addition, the attacker removes the *KeyShare* extension in the ClientHello which contains the originally proposed strong groups to make the ClientHello format similar to a TLS 1.2 one. Based on the modified ClientHello, the server responds with the highest common TLS version, in this case, TLS 1.2. Since the server received DHE_EXPORT, it will respond with DHE_EXPORT. In addition, based on the latest proposed downgrade protection mechanism in draft 14, the server sets a fixed number in the last eight bytes of the server's nonce to indicate to the client that the server has received version 1.2 or below. The nonce n'_R will get signed next in the ServerKeyExchange, therefore, ideally, such downgrade attack that modifies the version should be detected by the client when the client receives the signed nonce in the ServerKeyExchange.

However, if the attacker holds the signature key (the server's long-term private-key), he can sign a message of his choice. Therefore, the attacker signs a modified ServerKeyExchange after he replaces the fixed number from the server's nonce to disguise the version downgrade and successfully downgrade the negotiated TLS version from TLS 1.3 to TLS 1.2. As a result, all TLS 1.2 attacks are inherited. The attacker can now perform the Logjam [5] attack that mainly exploits the fact that the server signature in the ServerKeyExchange does not include the ciphersuite name, hence, the client will not detect the ciphersuite downgrade. Consequently, once the attacker receives the client's DHE public value, g^a, he can perform the discrete log to recover the server's exponent b, and then, computes the shared DHE secret g^{ab}. However, since the client's and server's nonces are input to the KDF, and since the attacker has modified the server's nonce, the attacker needs to compute two different KDFs, one with the server using the server's nonce (with the fixed number in the last eight bytes) n'_R and the second one with the client using the modified server's nonce (without the fixed number) n_R. The attacker will fail in making the client and server compute shared keys that he can compute but he can cerate tuple of shared keys (ms_I, k_{1I}, k_{2I}) with the client and another tuple with the server (ms_R, k_{1R}, k_{2R}) with his own keys with each party. Therefore, he can impersonate the client to the server and vice versa. Now, the attacker can decrypt messages from the server, re-encrypts them to the client, then, forge the Finished MACs. Finally, the attacker can read and modify the application data. See Figure 8 for illustration.

Another more interesting attack if the attacker performs version downgrade attack to force the use of RSA key-exchange (which is not supported in TLS 1.3) instead of (EC)DHE. This will provide him with the ability to decrypt any past or future traffic encrypted using the same key.

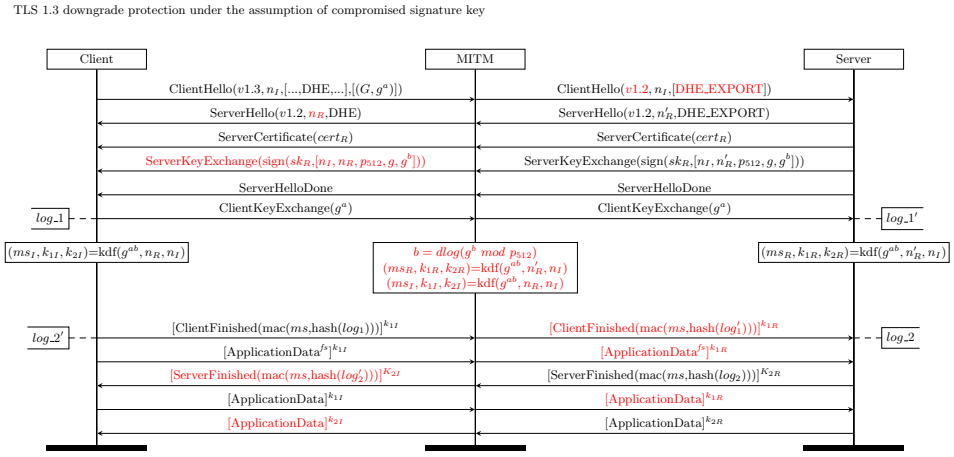

Fig. 8. TLS 1.3 downgrade protection under the assumption of compromised signature key. The *fs* above the [ApplicationData] refers to *false start*, i.e. the client starts sending data before the server sends its Finished message.

5.5 Key-exchange Algorithms.

Key-exchange algorithms are used to define the method that two communicating parties are going to use to create and exchange a master-secret. In TLS 1.3, there are four supported key-exchange algorithms: DHE, ECDHE, DHE-PSK (PSK-DHE in some ciphersuites), and ECDHE-PSK (PSK refers to Pre-Shared Key. PSK is suitable for resource constrained devices such as embedded devices where both parties agree on a shared key in advance rather than during the TLS handshake, to save the cost of the key computation).

The Security of DHE Key. The security of the DHE key-exchange algorithm relies on the difficulty of solving the discrete logarithm problem. That is, given $x = g^a \ mod \ p$ it is hard to find a.

DHE Usage in TLS 1.3. In TLS 1.3, DHE is used in the key-exchange for non-PSK key establishment. The client offers (EC)DHE cryptographic parameters in the *KeyShare* extension. The server either accepts the client's offer or offers other DHE parameters that must be supported by the client. The *KeyShare* extension contains a *KeyShareEntry* which contains the *NamedGroup* and *key_exchange*. In TLS 1.3, clients must offer at least one *NamedGroup* value listed as follows (section 4.2.3. - Negotiated Groups [10]): secp256r1, secp384r1, secp521r1, x25519, and x448. The supported DHE groups in TLS 1.3 are: ffdhe2048, ffdhe3072, ffdhe4096, ffdhe6144, and ffdhe8192 [10].

Assumptions. In this model, we assume a client and a server that both have weak DHE groups due to advancement in cryptanalysis (e.g. similar to the current situation regarding 512-bit and 1024-bit keys). The client and server are offering the latest version (TLS 1.3) and support several ciphersuites. Some of these ciphersuites are least preferred than others due to some weaknesses (e.g. similar to the current situation regarding RC4 in TLS 1.2) but they are supported by both the client and server for backward compatibility or due to negligence. The attacker is an active MITM who is capable of computing the discrete log for weak DHE primes, e.g. 512-bit primes. The attacker goal is to downgrade the negotiated ciphersuite between TLS 1.3 client and server (e.g. removes all ciphersuites and keeps the least preferred cipher that has weaknesses, e.g. weak encryption algorithm).

The Consequences of Weak DHE Parameters in TLS Downgrade Protection. A weak DHE key (e.g. 512-bit prime) means that the attacker can compute the private exponent. However, we will show that weak DHE key alone does not always mean successful downgrade attack.

In this attack, as shown in Figure 9, the client sends a ClientHello with weak DHE group. The attacker modifies the ciphersuite list with the intention to use a weaker ciphersuite. The attacker removes all ciphersuites in the list and leaves the least preferred ciphersuite. Because there is only one ciphersuite in the list, the server is forced to choose that ciphersuite which is neither the client nor the server preferred cipher but was just supported for backward compatibility. After the client receives the ServerHello, both parties compute the session keys. In the same time, the attacker computes the discrete log to get the private exponent b for the server and use that to compute the shared secret g^{ab}. The client and server have different transcripts log_1 and log'_1 due to the modification made by the attacker in the ciphersuite list in the ClientHello. However, the attacker has the right transcript with each party and has the shared key g^{ab}. Therefore, he can compute shared keys (the session and master keys) with each party. In particular, he computes (ms_I, k_{1I},k_{2I}) with the client and (ms_R, k_{1R},k_{2R}) with the server where k_{1I} is the attacker's shared key with the client and k_{2R} is the attacker's shared key with the server (see Figure 9 for illustration). The attacker now can act in a similar fashion to the classical unauthenticated DH MITM attack. The server encrypts the client's messages using k_{2R} believing it is the client's session key while it is the attacker's key impersonating the client. Then, the attacker decrypts the message (and may modify it), then re-encrypts it again using the client's session key k_{2I} and sends it to the client pretending that the message is coming from the server. Next, the server sends its encrypted certificate, the attacker forwards it to the client. Then, the server sends ServerCertificateVerify which contains a signature over a hash of the transcript (log_2). To proceed with the downgrade attack, the attacker needs to produce a valid signature over the modified transcript (log'_2) but he can not do so as he does not have the server's long-term signature key. Once the client receives the ServerCertificateVerify and verifies the signature, the disparity in the transcripts will be discovered. The

downgrade attack will fail to continue unless if the attacker issued a forged certificate that impersonates the server which he can use to sign the transcript to hide the logs disparity. However, this section analyses the consequences in the presence of weak DHE keys only without considering other factors such as forged certificate or signature as the assumptions clarify.

Having said that, weak DH alone can not allow an attacker to successfully complete a ciphersuite downgraded attack. We will reach similar conclusion if the example starts with a version downgrade with the aim of downgrading the ciphersuite there. In that case, the disparity in the server's nonces between n_R and n'_R will be discovered once the client receives the ServerKeyExchange that contains a signature covering the server nonce. Similarly, if the attacker does not have additional capabilities over the weak DHE parameters, he will fail in producing a valid signature.

However, if we added another weak primitive (e.g. weak hash functions) to the weak DHE parameters assumption, i.e., if the hashes in the downgraded ciphersuite and the "signature_algorithms" has collision, this will help the attacker to complete the handshake by producing identical hashes for different transcripts. In addition, the attacker needs to forge the Finished MACs later. However, if he performed version downgrade, he still needs to decrypt and re-encrypt the traffic because the version downgrade protection mechanism in the server's nonce which is included in the KDF. With the version downgrade protection mechanism, it is not possible for the attacker to correctly create shared secret between the client and the server. However, if the downgraded ciphersuite has strong hash, despite any other weak primitives (e.g. broken symmetric encryption algorithm) in the ciphersuite, the attacker may not succeed in the downgrade attack because the disparity in the transcripts will always reveal the downgrade attack.

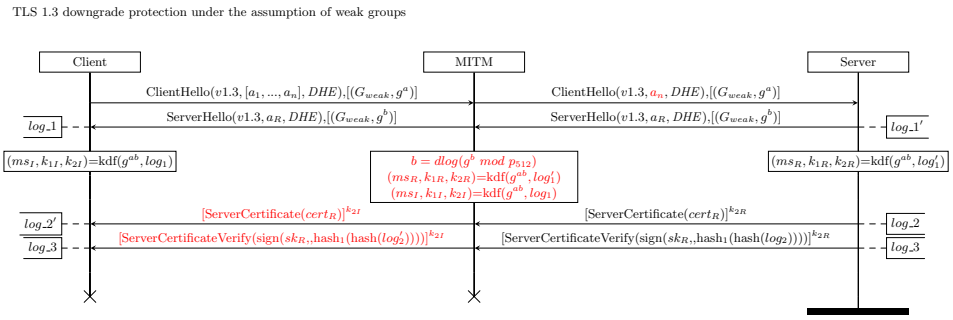

Fig. 9. TLS 1.3 downgrade protection under the assumption of weak DHE parameters.

5.6 Hash Functions

Secure cryptographic hash functions are functions that map arbitrary length strings into a fixed size string called the hash value or hash. Hash functions are often used to achieve integrity. If two hashes are equal, this entails with high probability that both input strings are identical (refer to the next section for secure cryptographic hash function requirements). Otherwise, this means that the two strings are not equal.

TLS 1.3 uses hash functions in the HKDF which are specified in the ciphersuite and chosen from SHA256 and SHA384. Hash functions are also used in the signature scheme chosen from: SHA256, SHA384, SHA1 and SHA512 (see section 4.2.2 [10]).

The Security of Hash Functions. A secure hash function should meet the following requirements:

- Collision-resistance: it should be hard to find two different strings x and x' such that $x \neq x'$ and $H(x) = H(x')$.
- Pre-image resistance: given a hash value y, it should be hard to find a string x such that $H(x) = y$.
- Second pre-image resistance: given a string x, it should be hard to find another string x' such that $x \neq x'$ and $H(x) = H(x')$

Hash Functions Usage in TLS 1.3. In TLS 1.3, hash functions are used in various places. First, in the ServerCertificateVerify there is a double hash over the handshake transcript (log_2) as follows: the inner hash uses one of the two hash algorithms of the ciphersuite and the outer hash uses one of the "signature_algorithms" hashes. Second, hash functions are used in the Finished MACs over the handshake transcripts, log_3 in the ServerFinished and log_4 in the ClientFinished. Third, there is a hash function used in two KDFs. The first one is over handshake transcript (log_1) to compute session keys k_1 and k_2. The second one is computed over the handshake transcript (log_3) to compute the master-secret. Finally, hash functions are also used in the heart of the construction of other function, namely, in the finish HMAC and in the HKDF.

Assumptions. The client and server are using TLS 1.3 and both supports TLS 1.2 that implements the version downgrade protection mechanism for backward compatibility. We also assume that the hash functions used in the protocol has collision. The client and server are offering the latest version (TLS 1.3) and support several ciphersuites. Some of these ciphersuites are least preferred than others due to some weaknesses (e.g. similar to the current situation regarding RC4 in TLS 1.2) but they are supported by both the client and server for backward compatibility or due to negligence. The attacker is an active MITM who is capable of computing the discrete log for weak DHE primes e.g. 512-bit primes. The attacker goal is to downgrade the negotiated ciphersuite between TLS 1.3 client and server (e.g. removes all ciphersuites and keeps the least preferred cipher that has weaknesses e.g. weak encryption algorithm).

The Consequences of Collision in Hash Functions in TLS Downgrade Protection. A collision in the hash function means that an attacker can hide modifications in the transcript and consequently, can forge a MAC and a signature. We will show an attack example that illustrates how a collision in the hash function allows an attacker to successfully make the client and server completes that handshake with the least preferred cipher and agree on a shared secret though the transcripts at both ends are different.

In essence, this attack example have same goals as the previous example with weak DHE parameters. The attacker's goal is to force both ends to negotiate the least preferred ciphersuite. The client sends a ClientHello with list of ciphersuites. The attacker removes all ciphersuites in the list and leaves the least preferred ciphersuite. Because there is only one ciphersuite in the list, the server is forced to choose that ciphersuite which is neither the client nor the server preferred cipher but was just supported for backward compatibility. After the client receives the ServerHello, both parties compute the session keys. As stated earlier, there is a hash function that is computed over log_1 which is used as input for the KDF that computes the session keys. Due to collision in the hash function, the attacker can succeed in producing identical hashes for different log_1 at each end. The attacker continue in producing collision at each step where hashes are computed at both ends until the Finished MACs.

Collision in the hash function can be made up during the handshake as shown in [13]. In our example, collision in the transcript will allow the attacker to produce identical transcript at both ends even without knowledge of the keys.

6 Proposed Solutions

In this section, we propose solutions that can contribute to reducing downgrade attacks in protocols. We provide general description of the ideas we suggest. A precise evaluation, analysis and message sequence diagram is still needed to judge these proposals.

Select the Ciphersuite by the Client One possible solution is to select the ciphersuite by the client instead the server. Our reasoning is based on the fact that most of downgrade attacks happens by MITM attacker who modifies the ClientHello parameters. The ClientHello message is the first message and no security parameters have been negotiated yet unless the client has a certificate, and in real-world implementations, mostly, clients do not have a certificate. Therefore, we suggest the following sequence: first, the server sends its certificate and sends the supported ciphersuites in a signed format. Second, the client selects the ciphersuite locally. Third, the client needs to inform the server with the selected cipher, therefore, it sends the selected ciphersuite to the server. The server responds with a signed message that contains the received selected cipher. The client verifies the signed message to check whether the server has received the right ciphersuite or not.

This method requires two signed messages from the server: first, the proposed cipher list. Second, a signed message to allow the client to verify whether the server has received the correct ciphersuite. This might add performance overhead.

Select the Ciphersuite Independently Another method works as follows: first, the client offers its ciphersuites in the ClientHello. Second, the server sends its ciphersuites in a signed format. At this stage, we have at least one trusted list (the server's signed list). Now, both ends can compute the selected ciphersuite independently. The client has the server's preferences in signed format and has its preferences locally. Therefore, we can consider that the client's selected ciphersuite will be as intended (untampered). Then, the server sends its chosen ciphersuite in a signed message to the client. The client checks if the server got the same result as the client. If so, this means the client and server had the correct lists. Otherwise, the lists has been modified (mostly, the client list that is sent to the server in cleartext).

This method requires a precisely defined method for selecting the ciphersuites at both ends which currently does not exist in TLS standards and it may be difficult to reach a consensus towards a single algorithm (but not impossible because other protocols such as SSH has defined method for ciphersuite selection).

Sign the Server's Ciphersuites in the Certificate. Since we already rely on the Certificate system, we can can get rid of the negotiation problems by treating the server's ciphersuites in a similar fashion to the server's public-key (the server's public-key gets signed by a trusted third party). The client gets the server's signed ciphersuite list from the certificate. Then, the client computes the right ciphersuite locally.

Similar to the first solution, the client needs to inform the server with the chosen ciphersuite. Then, the server responds with a signed message that contains the received selected ciphersuite. The client verifies the server's signed message to check whether the received ciphersuite at the server's end is what the client originally sent or not. The server does not need to send the ciphersuites list in a signed message as the signed list is already in the certificate. However, flexibility is reduced. For example, if the server wants to modify the list of ciphersuites, this can't be done directly done but through a certificate authority. This can be tedious especially that normal certificates life-time is between one-two years. However, this draw back can be avoided by assigning expiry-date for the ciphersuites list so that the server can push updated cipehrsuites list frequently during the certificate life-time. The pushing updates overhead is significantly less than negotiating the ciphersuites for every session.

Finally, in addition to the previous proposals, we advocate for defining a method for ciphersuite selection and not leave open interpretation for developers as this leads to mistakes. There might not be exact right way to select the

ciphersuite but one way is to iterate the client's preferences and select the first matched ciphersuite in the server's list. We suggest giving preference to the client because clients are normally represented by web browsers and presumably, web browsers developers are more aware of the advancements in security and will get their list order based on the recommendations and will update their software in a timely manner (whether clients will update their browser's software or not, that's another issue).

7 Conclusion and Future Work

In this report, we explored downgrade attacks in TLS protocol. We studied downgrade protection mechanisms in the currently deployed version, TLS 1.2 and the upcoming version TLS 1.3. Since cryptographic primitives are crucial for providing downgrade protection mechanisms, this motivated us to analysed TLS 1.3 protection mechanisms when a cryptographic primitive is broken. Our analysis added clarity over the existing literature and helped us articulate broken primitives consequences. The analysis showed that weak DHE parameters alone can not allow an attacker to complete a handshake for downgraded parameters while compromised digital signature would allow version downgrade that leverages TLS 1.2 attacks (e.g. Logjam), and weak hash functions allow successful completion of handshake despite different transcripts at each end-point without the need for breaking the key.

Our study for TLS and downgrade attacks led us to reason about new possible negotiation methods that can eliminate downgrade attacks. We provided a brief description for some proposed solutions. They still need further analysis and proof of correctness and we leave this for future work. However, any radical solutions may be faced with compatibility issues but we still believe that new design ideas worth analysing and comparing with the existing negotiation methods.

8 Acknowledgment

I would like to express my gratitude to: Dr. Markulf Kohlweiss from Microsoft research for clarifying several questions related to his paper [12], Katriel Cohn-Gordon for proof-reading the report.

References

1. T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," 2008. [Online]. Available: https://tools.ietf.org/html/rfc5246
2. E. Rescorla, "Guidelines for Writing RFC Text on Security Considerations," 2003.
3. epic.org, "International Traffic in Arms Regulations," 1992. [Online]. Available: https://epic.org/crypto/export_controls/itar.html

4. B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Y. Strub, and J. K. Zinzindohoue, "A Messy State of the Union: Taming the Composite State Machines of TLS," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 535–552.
5. D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, 2015, pp. 5–17. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813707
6. S. Thomas, *SSL and TLS Essentials: Securing the Web*. John Wiley and Sons, Inc., 2000.
7. T. Dierks and C. Allen, "The TLS Protocol Version 1.0," 2016. [Online]. Available: https://tools.ietf.org/html/rfc2246
8. T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," 2006. [Online]. Available: https://tools.ietf.org/html/rfc4346
9. E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3 - draft-ietf-tls-tls13-00," 2014. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tls-tls13-00
10. E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3 - draft-ietf-tls-tls13-14," 2016. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tls-tls13-14
11. C. Cremers, M. Horvat, S. Scott, and T. van der Merwe, "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication," in *Proceedings of the IEEE Symposium on Security and Privacy (SP 2016)*, 2016.
12. K. Bhargavan, C. Brzuska, C. Fournet, M. Green, M. Kohlweiss, and S. Zanella-Béguelin, "Downgrade Resilience in Key-Exchange Protocols," Cryptology ePrint Archive, Report 2016/072, 2016. https://eprint. iacr. org/2016/072. 4, Tech. Rep., 2016.
13. K. Bhargavan and G. Leurent, "Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH," in *Proceedings of the Network and Distributed System Security Symposium 2016 (NDSS 16)*, 2016.
14. IANA, "Transport Layer Security (TLS) Parameters," 2016. [Online]. Available: http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml

Appendix A. TLS Ciphersuite

Ciphersuite name has the following name convention:
"TLS_KEA_AUTH_WITH_CIPHER_HASH = VALUE" [14]. In the following we explain each part of the name convention [14]:

- TLS: the protocol TLS
- KEA: the key exchange algorithm, e.g. RSA or Diffie-Hellman (DH). DH can be static (DH), Ephemeral (DHE), or Elliptic-curve (ECDHE).
- AUTH: the authentication mechanism, e.g. RSA which means, RSA digital signature using RSA long term key in that is associated with RSA certificate.
- WITH: the conjuncture string "WITH"
- CIPHER: the symmetric encryption algorithm name which includes the key length, e.g. AES_128, CHACHA20.

– HASH: the hash algorithm to be used in HKDF.
– VALUE: the two byte id for the ciphersuite. Ciphersuites IDs are assigned by the Internet Assigned Numbers Authority (IANA) [14]