# What's in a Downgrade? A Taxonomy of Downgrade Attacks in the TLS Protocol and Application Protocols Using TLS

Eman Salem Alashwali[1,2] (✉) and Kasper Rasmussen[1]

[1] University of Oxford, Oxford, United Kingdom
{eman.alashwali,kasper.rasmussen}@cs.ox.ac.uk
[2] King Abdulaziz University (KAU), Jeddah, Saudi Arabia
ealashwali@kau.edu.sa

**Abstract.** A number of important real-world protocols including the Transport Layer Security (TLS) protocol have the ability to negotiate various security-related choices such as the protocol version and the cryptographic algorithms to be used in a particular session. Furthermore, some insecure application-layer protocols such as the Simple Mail Transfer Protocol (SMTP) negotiate the use of TLS itself on top of the application protocol to secure the communication channel. These protocols are often vulnerable to a class of attacks known as *downgrade attacks* which targets this negotiation mechanism. In this paper we create the first taxonomy of TLS downgrade attacks. Our taxonomy classifies possible attacks with respect to four different vectors: the protocol element that is targeted, the type of vulnerability that enables the attack, the attack method, and the level of damage that the attack causes. We base our taxonomy on a thorough analysis of fifteen notable published attacks. Our taxonomy highlights clear and concrete aspects that many downgrade attacks have in common, and allows for a common language, classification, and comparison of downgrade attacks. We demonstrate the application of our taxonomy by classifying the surveyed attacks.

## 1 Introduction

A number of important real-world protocols, such as the Transport Layer Security protocol (TLS) [13] [29], which is used by billions of people everyday to secure internet communications, support multiple protocol versions and algorithms, and allow the communicating parties to negotiate them during the handshake. Furthermore, some important legacy application-layer protocols that are *not* secure by design such as the Simple Mail Transfer Protocol (SMTP) [22] allow the communicating parties to negotiate upgrading the communication channel to a secure channel over a TLS layer. However, experience has shown that protocol developers tend to maintain support for weak protocol versions and algorithms, mainly to provide backward compatibility. In addition, empirical analysis of real-world deployment shows that a high percentage of SMTP servers that support

TLS and are capable of upgrading SMTP to SMTP-Secure (SMTPS) are configured in the "opportunistic security" mode [15], meaning that they "fail open", and operate in an unauthenticated plaintext mode if the upgrade failed for any reason, favoring functionality over security [9] [17].

In a typical downgrade attack, an active network adversary[1] interferes with the protocol messages, leading the communicating parties to operate in a mode that is weaker than they prefer and support. In recent years, several studies illustrated the practicality of downgrade attacks in widely used protocols such as TLS. More dangerously, downgrade attacks can succeed even when only one of the communicating parties supports weak choices as in [2] [4].

There are plenty of reported downgrade attacks in the literature that pertain to TLS such as [17] [2] [4] [6] [27] [16] [33] [5] [24] [3]. A close look at these attacks reveals that they are not all identical: they target various elements of the protocol, exploit various types of vulnerabilities, use various methods, and result in various levels of damage.

The existing literature lacks a taxonomy that shows the big picture outlook of downgrade attacks that allows classifying and comparing them. To bridge this gap, this paper presents a taxonomy of downgrade attacks with a focus on the TLS protocol based on an analysis of fifteen notable published attacks. The taxonomy helps in deriving answers to the following questions that arise in any downgrade attack:

1. *What has been downgraded?*
2. *How is it downgraded?*
3. *What is the impact of the downgrade?*

Our downgrade attack taxonomy classifies downgrade attacks with respect to four vectors: element (to answer: What has been downgraded?), vulnerability and method (to answer: How is it downgraded?), and damage (to answer: What is the impact of the downgrade?).

The aim of this paper is to provide a reference for researchers, protocol designers, analysts, and developers that contributes to a better understanding of downgrade attacks and its anatomy. Although our focus in this paper is on the TLS protocol and the application protocols that use it, this does not limit the paper's benefit to TLS. The paper can benefit the design, analysis, and implementation of any protocol that has common aspects of TLS.

Our contribution is twofold: First, we provide the first taxonomy of TLS downgrade attacks based on a thorough analysis of fifteen surveyed attacks. Our taxonomy dissects complex downgrade attacks into clear categories and provides a clean framework for reasoning about them. Second, although our paper is not meant to provide a comprehensive survey, however, as a necessary background, we provide a brief survey of all notable published TLS downgrade attacks. Unlike the existing general surveys on TLS attacks, our survey is focused on a particular family of attacks that are on the rise, and covers some important

[1] Throughout the paper we will use the terms: active network attacker, active network adversary, and man-in-the-middle interchangeably

recent downgrade attacks that none of the existing surveys [26] [10] (which date back to 2013) have covered.

The rest of the paper is organised as follows: in section 2, we summarise related work. In section 3, we provide an illustrative example of downgrade attacks. In section 4, we describe the attacker model that we consider in our taxonomy. In section 5, we describe the methodology we use to devise the taxonomy. In section 6, we briefly survey fifteen cases of downgrade attacks in TLS. In section 7, we present our taxonomy. In section 8, we provide a discussion. In section 9, we conclude. Finally, Appendix A provides a background in the TLS protocol.

## 2 Related Work

Bhargavan et al. [5] provide a formal treatment of downgrade resilience in cryptographic protocols and define downgrade security. In our work, we look at downgrade attacks from an informal and pragmatic point of view. We also consider downgrade attacks in a context beyond the key-exchange, e.g. in negotiating the use of TLS layer in multi-layers protocols such as SMTP.

The work of [10] and [26] provide surveys on TLS attacks in general. Their surveys cover some of the TLS downgrade attacks that we cover. However, our work is not meant to survey TLS downgrade attacks, but to analyse them to create a taxonomy of downgrade attacks and to provide a framework to reason about them. Furthermore, our work covers state-of-the-art TLS downgrade attacks that have not been covered in previous surveys such as downgrade attacks in draft-10 of the coming version of TLS (TLS 1.3) [5], the SLOTH attack [6], the DROWN attack [3], among others.

Howard and Longstaff [21] present a general taxonomy of computer and network attacks. Our approach is similar to the one taken in [21] in terms of presenting the taxonomy in logically connected steps. We have some common categories such as the vulnerability, but we also introduce our own novel categories such as the element and damage which classifies downgrade attacks at a lower level.

In [31] a taxonomy of man-in-the-middle attacks is provided. It is based on four tiers: “state”, “target”, “behaviour”, and “vulnerability”. Our taxonomy is particularly focused on downgrade attacks, thus provides further insights over the general man-in-the-middle taxonomy. We also have different perspectives. For example, although we share the vulnerability category, [31] present it in an exhaustive list of vulnerabilities such as “cipher block chaining”, “compression”, “export key”, etc. while our approach is to focus on the source of the flaw that allows the attack. We end up with three vulnerability sub-categories: implementation, design, and trust-model, which are more likely to capture future attacks.

## 3 Downgrade Attacks, an Illustrative Example

Figure 1 shows an illustrative example of downgrade attacks in a simplified version of the TLS 1.2 protocol inspired by the Logjam attack [2]. Throughout the paper in the message sequence diagrams, we denote the communicating parties

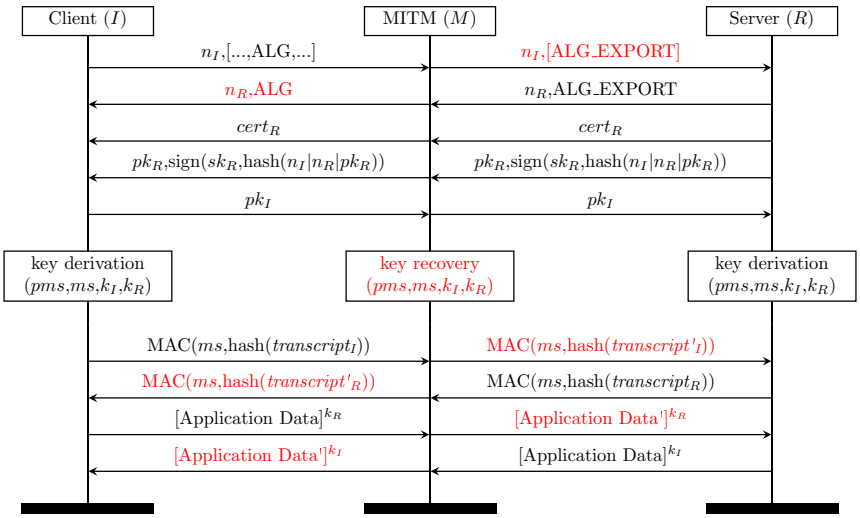


**Fig. 1:** Illustrative example of downgrade attack in a simplified version of TLS.

by client (initiator $I$) and server (responder $R$). We denote the man-in-the-middle by (MITM $M$). A background on the TLS protocol that is necessary to comprehend the example is provided in Appendix A.

In this example, we assume certificate-based unilateral server-authentication mode using ephemeral Diffie-Hellman (DHE) key-exchange algorithm, and Message Authentication Code (MAC) to authenticate the exchanged handshake messages (the transcript). As depicted in Figure 1 the client starts the handshake by sending its nonce ($n_I$) and a list of ciphersuites ([...,ALG,...]) to the server. The ciphersuite is a string (ALG) that defines the algorithms to be used in a particular session. In this example, we assume that the client's ciphersuites list contains only strong ciphersuites. The server must select one of the offered ciphersuites to be used in subsequent messages of the protocol. A man-in-the-middle modifies the client's proposed ciphersuites such that they offer only export-grade[2] ciphersuite ([ALG_EXPORT]), e.g. key-exchange with 512-bit DHE group. If the server supports export-grade ciphersuites, for example, to provide backward compatibility to legacy clients, it will select an export-grade one, misguided by the modified client message that offered only export-grade ciphersuites. Then, the server sends its nonce ($n_R$) and its selected ciphersuite ALG_EXPORT to the client. To avoid detection, the man-in-the-middle modifies the server's choice from ALG_EXPORT to ALG to make it acceptable for the client that may not

[2] Export-grade ciphers are weak ciphers with a maximum of 512-bit key for asymmetric encryption, and 40-bit key for symmetric encryption [34].

support export-grade ciphersuites as is the case in most updated web browsers today. Then, the server sends its certificate ($cert_R$), followed by a message that contains the server's public-key parameter $pk_R$, and a signed hash of the nonces ($n_I$ and $n_R$) and the server's public-key parameters $pk_R$. The signature is used to authenticate the nonces and the server's selected key parameters. However, in TLS 1.2 and below, the server's signature does not cover the server's selected ciphersuite (ALG_EXPORT in our example). Therefore, even if the client supports only strong ciphersuites, if it accepts arbitrary key parameters (e.g. non standard DHE groups), it will not distinguish whether the selected ciphersuite is export-grade or strong, and will generate weak keys based on the server's weak key parameters, despite the client's support for only strong ciphersuites. After that, the client sends its key parameter ($pk_I$). Then, both parties should be able to compute the pre-master secret ($pms$), the master secret ($ms$), and the client and server session keys, ($k_I$) and ($k_R$), respectively. The exchanged weak public-key parameters enable a man-in-the-middle to recover secret values from the weak public-keys, e.g. recover the private exponent from one or both parties' public-keys using Number Field Sieve (NFS) discrete log ($dlog$) (since we assume DHE key). Consequently, be able to compute the $pms$, $ms$, $k_I$, and $k_R$ in real-time. As a result of breaking the $ms$, the attacker can forge the MACs that are used to provide transcript integrity and authentication, hence, circumvent downgrade detection. Since the man-in-the-middle has the session keys, he can decrypt messages between the client and server as illustrated in Figure 1. This general example is similar to the Logjam [2] attack. This example is not the only form of TLS downgrade attacks as the paper will elaborate in the coming sections.

## 4 Attacker Model

In our taxonomy, we assume an external man-in-the-middle attacker who can passively eavesdrop on, as well as actively inject, modify, or drop messages between the communicating parties. The attacker can also connect to multiple servers in parallel. Furthermore, the attacker has access to bounded computational resources that allow him to break weak cryptographic primitives.

## 5 Methodology

First, to devise the taxonomy, we analyse fifteen published cases of downgrade attacks that relate to TLS from: [17] [2] [4] [6] [27] [16] [33] [5] [24] [3] (some papers have more than one attack). These attacks represent all the notable published downgrade attacks that we are aware of, starting from the first version of TLS (SSL 2.0) until draft-10 of the upcoming version (TLS 1.3). We summarise them in section 6. Second, we extract the features that characterise each attack (which we refer to as vectors), namely: the attacker targets an element that defines the mode of the protocol which can be the protocol algorithms, version, or the TLS layer, in order to modify or remove. The attacker also needs to exploit

a vulnerability, which can be due to implementation, design, or trust-model. The downgrade is achieved by using a method which can be message modification, dropping, or injection. Finally, the attack results in a damage which can be either broken security or weakened security. These four main vectors are intrinsic to any downgrade attack under the specified attacker model and can therefore be used to characterise each attack in that model. Third, after identifying the vectors, we devise the taxonomy. We define the notions of the taxonomy's categories and sub-categories in section 7. Finally, we show the taxonomy's application in classifying known TLS downgrade attacks.

## 6 Downgrade Attacks in TLS, a Brief Survey

In this section, we briefly survey the TLS downgrade attacks that we have analysed in order to devise the taxonomy. We highlight the attack names in **Bold** and we use these names throughout the paper. We assume the reader's familiarity with the TLS technical details. The unfamiliar reader is advised to read Appendix A, which provides the required background to comprehend the rest of the paper.

Downgrade attacks have existed since the very early versions of TLS: SSL 2.0 [19] and SSL 3.0 [18]. SSL 2.0 suffers from the **"ciphersuite rollback"** attack, where the attacker limits SSL 2.0 strength to the "least common denominator", i.e. the weakest ciphersuite, by modifying the ciphersuites list in one or both of the `Hello` messages that both parties exchange so that they offer the weakest ciphersuite [33] [32], e.g. export-grade or "NULL" encryption ciphersuites. To mitigate such attacks, SSL 3.0 mandated a MAC of the protocol's transcript in the `Finished` messages which needs to be verified by both parties to ensure identical views of the transcript (i.e. unmodified messages).

However, SSL 3.0 is vulnerable to the **"version rollback"** attack that works by modifying the client's proposed version from SSL 3.0 to SSL 2.0 [33]. This in turn leads SSL 3.0 servers that support SSL 2.0 to fall back to SSL 2.0. Hence, all SSL 2.0 weaknesses will be inherited in that handshake including the lack of integrity and authentication checks for the protocol's transcript as we described above, which render the downgrade undetected.

Another design flaw in SSL 3.0 allows a theoretical attack named the **"key-exchange rollback"** attack, which is a result of lack of authentication for the server's selected ciphersuite (which includes the name of the key-exchange algorithm) before the `Finished` MACs [33]. In this attack, the attacker modifies the client's proposed key-exchange algorithm from RSA to DHE, which makes the communicating parties have different views about the key-exchange algorithm. That is, the server sends DHE key parameters in the `ServerKeyExchange` message while the client treats them according to export-grade RSA algorithm. These mismatched views about the key-exchange result in generating breakable keys which are then used by the attacker to forge the `Finished` MACs to hide the attack, impersonate each party to the other, and to decrypt the application data.

In [24], an attack which we call the **"DHE key-exchange rollback"** is presented. It can be considered a variant of the **"key-exchange rollback"** in [33]. In this attack the attacker modifies the client's proposed key-exchange algorithm from DHE to ECDHE. As a result, the server sends a `ServerKeyExchange` that contains ECDHE parameters based on the client offer while the client treats them as DHE parameters. The client does not know the selected key-exchange algorithm by the server since the selected ciphersuite (which includes the key-exchange algorithm) is not authenticated in the `ServerKeyExchange`. Similar to the **"key-exchange rollback"** attack in [33], these mismatched views about the key-exchange algorithm result in breakable keys, which allow the attacker to recover the pre-master and master secretes. Consequently, be able to forge the `Finished` MACs to hide the modifications in the `Hello` messages, impersonate each party to the other, and decrypt the application data.

Version downgrade is not exclusive to SSL 3.0. The Padding Oracle On Downgraded Legacy Encryption **(POODLE)** attack [27] shows the possibility of version downgrade in recent versions of TLS (up to TLS 1.2) by exploiting the "downgrade dance", a client-side implementation technique that is used by some TLS clients (e.g. web browsers). It makes the client fall back to a lower version and retries the handshake if the initial handshake failed for any reason [27]. In the POODLE attack, a man-in-the-middle abuses this mechanism by dropping the `ClientHello` to lead the client to fall back to SSL 3.0. This in turn brings the specific flaw that is in the CBC padding in all block ciphers in SSL 3.0, which allows the attacker to decrypt some of the SSL session's data such as the cookies that may contain login passwords.

In [2], the **Logjam** attack is presented. It uses a method similar to the one we explained in the illustrative example in section 3. The Logjam attack is applicable to DHE key-exchange. It works by modifying the `Hello` messages to misguide the server into selecting an export-grade DHE ciphersuite which result in weak DHE keys. As stated earlier, TLS up to version 1.2 does not authenticate the server's selected ciphersuite (which includes the key-exchange algorithm) until the `Finished` MACs. As a result, the client receives weak key parameters and generates weak keys based on the server's weak parameters. The lack of early authentication of the server's selected ciphersuite gives the attacker a window of time to recover the master secret from the weakly generated keys in real-time, before the `Finished` MACs. Consequently, the attacker can forge the `Finished` MACs to hide the modifications in the `Hello` messages, and decrypt the the application data.

A similar attack called the Factoring RSA Export Keys **(FREAK)** attack [4] is performed using a method similar to the one used in the Logjam attack [2], which leads the server into selecting an export-grade ciphersuite. However, FREAK is applicable to RSA key-exchange and requires a client implementation vulnerability that makes a client that does not support export-grade ciphersuites accept a `ServerKeyExchange` message with weak ephemeral export-grade RSA key parameters, while the key-exchange algorithm is RSA (note that the `ServerKeyExchange` message must not be sent when the key-exchange algo-

rithm is non-export-grade RSA [13]). However, the `ServerKeyExchange` is sent in export-grade RSA or in (EC)DHE key-exchange. This implementation vulnerability leads the client to use the export-grade RSA key parameters that are provided in the `ServerKeyExchange` to encrypt the pre-master secret instead of encrypting it with the long-term (presumably strong) RSA key that is provided in the server's `Certificate`. This results in breakable keys that can be used to forge the `Finished` MACs and decrypt the application data.

In [3], a variant of the Decrypting RSA using Obsolete and Weakened eNcryption (**DROWN**) attack that exploits an OpenSSL server implementation bug [1] is presented. The attack enables a man-in-the-middle to force a client and server into choosing RSA key-exchange algorithm despite their preference for non-RSA (e.g. (EC)DHE) by modifying the `Hello` messages. The attacker then make use of a known flaw that can be exploited if the server's RSA key is shared with an SSLv2 server using an attack called Bleichenbacher attack [8] which enables the attacker to recover the plaintext of an RSA encryption (i.e. the pre-master secret) by using the SSLv2 server as a decryption oracle. If the attacker can break the pre-master secret, he can break the master secret and forge the `Finished` MACs to hide the attack, and be able to decrypt the application data. This downgrade attack is similar to the FREAK attack but the main difference is that it does not require the server to support export-grade RSA. This attack breaks even strong RSA keys (2048-bits) if the RSA key is shared with an SSLv2 server.

Another case of downgrade attack is the **"Forward Secrecy rollback"** attack [4], in which the attacker exploits an implementation vulnerability to make the client fall back from Forward Secrecy (FS)[3] mode to non-FS mode by dropping the `ServerKeyExchange` message. However, non-FS mode does not result in immediate breakage of any security guarantee such as secrecy unless the long-term key that encrypts the session keys got broken after the session keys have been used to encrypt application data.

In [6], a downgrade attack in TLS 1.0 and TLS 1.1 is illustrated. The attack comes under a family of attacks named Security Losses from Obsolete and Truncated Transcript Hashes (**SLOTH**). This attack is possible due to the use of non collision resistant hash functions (MD5 and SHA-1) in the `Finished` MACs. The use of MD5 and SHA-1 is mandated by the TLS 1.0-1.1 specifications [11] [12]. Non collision resistant hash functions allow the attacker to modify the `Hello` messages without being detected in the `Finished` MACs by creating a prefix-collision in the transcript hashes [6].

Downgrade attacks in multi-layered protocols that negotiate upgrading the connection to operate over TLS have been shown to be prevalent based on an empirical analysis of SMTP deployment in the IPv4 internet space [17]. In [17] they found evidence for corrupted `STARTTLS` commands which downgrade **SMTPS to SMTP** in more than 41,000 mail servers.

[3] Forward Secrecy (FS) is a property that guarantees that a compromised long-term key does not compromise past session keys [25])

Similarly, downgraded TLS as a result of **proxied HTTPS** connections[4] has been shown to be prevalent. In [16], empirical data show that 10-40% of the proxied TLS connections advertise known broken cryptographic choices [16].

Downgrade attacks continued to appear until draft-10 of the coming version of TLS (TLS 1.3 [28]), where [5] report three possible downgrade attacks in TLS 1.3 draft-10. The first attack is similar in spirit to SSL 3.0 **"version rollback"** attack that we explained earlier in this section. In this attack, the attacker modifies the proposed version to TLS 1.2 and enjoys the vulnerabilities in TLS 1.2 that (in the presence of export-grade ciphersuites either on the server side or in both sides) enable him to break the master secret before the `Finished` MACs as in [2] [4], hence circumventing downgrade detection.

The second attack in TLS 1.3 draft-10, which we call the "**downgrade dance version rollback**" attack, employs a method similar to the one employed in the POODLE attack [27], i.e. the attacker drops the initial handshake message one or more times to lead the clients that implement the "downgrade dance" mechanism to fall back to a lower version such as TLS 1.2, hence circumvent detection due to downgrade security weaknesses in TLS 1.2 and lower versions.

Finally, the third reported downgrade attack in TLS 1.3 draft-10, which we call the "`HelloRetry` **downgrade**" attack, occurs when an attacker injects a `HelloRetryRequest` message to downgrade the (EC)DHE group to a less preferred group despite the client and server preference to use another group. This attack can circumvent detection because the transcript hash restarts with every `HelloRetryRequest` [5]. However, consequent TLS 1.3 drafts mitigated this attack by continuing the hashes over retries [5].

## 7 Taxonomy of Downgrade Attacks

Based on the surveyed attacks in section 6, we distill four vectors that characterise the surveyed downgrade attacks, namely: element, vulnerability, method, and damage. These vectors represent the taxonomy's main categories. We define the notions of the categories and sub-categories that we use in our taxonomy. Figure 2 summarises the taxonomy.

1. **Element:** The element refers to the protocol element that is being negotiated between the communicating parties. The element's value is intrinsic in defining the protocol mode, i.e. the security level of the protocol run. The element is targeted by the attacker because either modifying or removing it will result in either a less secure, non secure, or less preferred mode of the protocol. We categorise the element into three sub-categories as follows:
   (a) Algorithm: The algorithm refers to the cryptographic algorithms, e.g. key-exchange, encryption, hash, signature, etc. and their parameters such

[4] A proxy refers to an entity that is located between the client and server that splits the TLS session into two separate sessions. As a result, the client encrypts the data using the proxy's public-key.

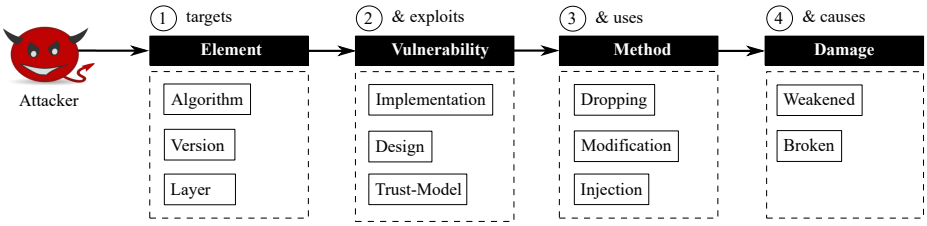


**Fig. 2:** A taxonomy of downgrade attacks in the TLS protocol and application protocols using TLS.

as block cipher modes of operation and key lengths, that are being negotiated to be used in subsequent messages of the protocol. Generally, in TLS, the main algorithms are represented by the ciphersuite, but they can also be represented by other parameters that are not part of the ciphersuite such as the extensions.

(b) Version: The version refers to the protocol version. A number of protocols including TLS allow their communicating parties to support multiple versions, negotiate the protocol version that both communicating parties will run, and allow them to fall back to a lower version to match the other party's version if the versions at both ends do not match.

(c) Layer: The layer refers to the whole TLS layer which is negotiated and optionally added in some legacy protocols. In such protocols like SMTP [22] for example, TLS encapsulation is negotiated through specific upgrade messages, e.g. STARTTLS [20], in order to upgrade the protocol from an insecure (plaintext and unauthenticated) to a secure (encrypted and/or authenticated) mode.

2. **Vulnerability:** Like any attack performed by an external man-in-the-middle, downgrade attacks require a vulnerability to be exploited. We categorise the vulnerability into three sub-categories as follows:
   (a) Implementation: An implementation vulnerability refers to a faulty protocol implementation. The existence of implementation vulnerabilities can be due to various reasons, for example, a programmer's fault, a state-machine bug, or a malware that corrupted the code.
   (b) Design: A design vulnerability refers to a flaw in the protocol design (i.e. the specifications). The protocol design is independent of the implementation. That is, even if the protocol was perfectly implemented, an attacker can exploit a design flaw to perform a downgrade attack.
   (c) Trust-Model: A trust-model vulnerability refers to a flaw in the architectural aspect (the TLS ecosystem in our case) and the trusted parties involved in this architecture which is independent of the protocol design and implementation.
3. **Method:** The method refers to the method used by the attacker to perform the downgrade. We categorise the method into three sub-categories as follows:

(a) Modification: In the modification method, the attacker modifies the content of one or more protocol messages that negotiate the element (i.e. algorithm, version, layer). If the protocol does not employ any integrity nor authentication checks for the handshake transcript, the downgrade attack can be trivially performed. Otherwise, the attacker needs to find ways to circumvent the checks, for example, break the master secret or create colliding hashes for the transcript.
(b) Dropping: In the dropping method, the attacker drops one or more protocol messages (possibly more than once).
(c) Injection: In the injection method, the attacker sends a new message to one of the communicating parties by impersonating the party's peer, for example to request a different algorithm or version than what is initially offered by the communicating party. The injection method is trivial in the absence of transcript integrity and authentication checks. Otherwise, it requires circumventing the integrity and authentication checks.

4. **Damage:** The damage refers to the resulted damage after a successful downgrade attack. We categorise the damage into two sub-categories as follows:
(a) Broken Security: Broken security refers to downgrade attacks that result in allowing the attacker to break one or more main security goals that the protocol claims to guarantee. In TLS the guarantees are: secrecy, authentication, and integrity.
(b) Weakened Security: Unlike the broken security damage, weakened security does not result in immediate breakage of any of the main security guarantees. Instead, weakened security refers to attacks that result in making the communicating parties choose a non-recommended or less preferred mode, which is not broken yet.

## 8 Discussion

In Table 1, we show the taxonomy's application in classifying the surveyed TLS downgrade attacks. Then we discuss our reasoning in some of the noteworthy cases (we will refer to the attacks by their reference number according to the numbers in Table 1).

It should be noted that classifying attacks that have implementation is straightforward as is the case in the attacks: **04**, **06**, **07**, **08**, **09**, and **10** where their classifications in Table 1 are self-explanatory based on mapping the surveyed attacks description in section 6 with the categories in Table 1. On the other hand, classifying either theoretical attacks such as **01**, **02**, **03**, **05**, **13**, **14**, and **15**, or attacks that have been reported based on evidence from empirical data such as **11** and **12**, is less straightforward and requires making some assumptions.

Ideally the taxonomy helps in classifying concrete attacks that have implementation. However, for the sake of illustration, we make some assumptions (mostly worst case assumptions) to mimic a concrete attack case from the general attack that does not have an implementation. In the following, we elaborate more on these cases.

| No. | Attack | Element | | | Vuln. | | | Method | | | Damage | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Algorithm | Version | Layer | Implementation | Design | Trust-model | Dropping | Modification | Injection | Weakened | Broken |
| 01 | SSL 2.0 Ciphersuite rollback [33]* | ✓ | | | | ✓ | | | ✓ | | | ✓ |
| 02 | SSL 3.0 Version rollback [33]* | | ✓ | | | ✓ | | | ✓ | | | ✓ |
| 03 | SSL 3.0 key-exchange rollback [33]* | ✓ | | | | ✓ | | | ✓ | | | ✓ |
| 04 | DHE key-exchange rollback [24] | ✓ | | | | ✓ | | | ✓ | | | ✓ |
| 05 | TLS 1.0-1.1 SLOTH [6]* | | ✓ | | | ✓ | | | ✓ | | | ✓ |
| 06 | POODLE version downgrade [27] | | ✓ | | ✓ | | | ✓ | | | | ✓ |
| 07 | FREAK [4] | ✓ | | | ✓ | | | | ✓ | | | ✓ |
| 08 | DROWN [3] | ✓ | | | | | ✓ | | ✓ | | | ✓ |
| 09 | Forward Secrecy rollback [4] | ✓ | | | ✓ | | | ✓ | | | ✓ | |
| 10 | Logjam [2] | ✓ | | | | ✓ | | | ✓ | | | ✓ |
| 11 | SMTPS to SMTP [17]* | | | ✓ | | ✓ | | | ✓ | | | ✓ |
| 12 | Proxied HTTPS [16]* | | | ✓ | | | ✓ | | | ✓ | | ✓ |
| 13 | TLS 1.3 Version rollback [5]* | | ✓ | | | ✓ | | | ✓ | | | ✓ |
| 14 | TLS 1.3 Downgrade-dance version fallback [5]* | | ✓ | | ✓ | | | ✓ | | | | ✓ |
| 15 | TLS 1.3 `HelloRetry` downgrade [5]* | ✓ | | | | ✓ | | | | ✓ | ✓ | |

**Table 1:** Classifying the surveyed downgrade attacks using our taxonomy. Attacks that are followed by "*" do not have an implementation and are either theoretical or based on evidence from measurement studies.

Attacks **01**, **02**, and **03** are theoretical. We classify the damage on these attacks based on the worst case assumption as follows: In **01**, we assume that the attacker can select export-grade or "NULL" encryption ciphersuites, which breaks a main security guarantees of TLS. In **02**, once the attacker downgrades SSL 3.0 to SSL 2.0, he can perform attack **01** without being detected due to lack of downgrade security in SSL 2.0. In **03**, we assume that the attacker can break the master secret. Similar to the FREAK [4] and Logjam [2] attacks, this allows the attacker to forge the `Finished` MACs which enables him to impersonate the client and/or the server and decrypt the application data, and this breaks main security guarantees.

Attack **05** is a theoretical attack. Based on the worst case assumption, we classify the downgraded element under the version element. The attacker can modify the version as well as the algorithms and hide the attack by producing prefix collision in the transcript hashes which are computed using non collision resistant hashes (MD-5 and SHA1 based on the protocol design and specifications) that go into the `Finished` MACs. If the attacker succeeded in downgrading the version to a broken version such as SSL 3.0, he can break main security guarantees (e.g. the CBC flaw in the symmetric encryption in SSL 3.0), hence the damage in attack **05** is classified under broken category.

Although attack **08** has an implementation but it is quite complex attack and its vulnerability classification is noteworthy. We classified its vulnerability under the trust model. By contemplating the main cause that allows this downgrade attack to succeed we find the main reason lies in breaking the RSA key that is then used to forge the `Finished` MACs, otherwise the attack will be detected. In this attack, the attacker can break even a strong 2048-bit RSA if the key is shared with an SSLv2 server (e.g. both servers uses the same certificate). Sharing RSA keys among servers is a trust-model vulnerability that allows the key sharing, rather than a protocol design nor implementation.

Attack **11** is based on evidence from real-world deployment. Based on the reported evidence described in [17], the method is classified under modification. However, dropping can also work as another method based on the STARTTLS specifications [20]. Since forcing TLS is not mandated by the SMTP protocol design and specifications, we do not consider the "fail open" local policy as an implementation vulnerability but a design one.

Attack **12** is widely known as HTTPS interception, where a man-in-the-middle (represented by a proxy) has full control over the TLS channel, which gives him the ability to downgrade TLS (algorithm, version, or layer). The empirical results in [16] shows an evidence of downgraded TLS version and algorithm due to proxied HTTPS. However, in fact, the man-in-the-middle can send the client's data to the server in cleartext. Therefore, based on the worst case assumption, the targeted element is classified under layer. The method is classified under injection since the man-in-the-middle injects a new message to the server by impersonating the client.

Attack **13** is similar in spirit to **02** that occurs in SSL 3.0 which is due to a design vulnerability. In TLS 1.3, the attack has been mitigated by redesigning the server's nonce to signal the received client's version [29].

Attack **14** is similar in spirit to **06** which targets the protocol version. If the attacker succeed in downgrading the version to a flawed version that has downgrade security weaknesses (as is the case in TLS 1.2 and below), the attacker can break main security guarantees based on the worst case assumption.

Attack **15** damage is classified under weakened security because as of this writing, no known broken (EC)DHE group elements are allowed in TLS 1.3 by design. Therefore, under the worst case assumption, the resulted damage leads both parties to agree on the least preferred DHE group.

Finally, as Table 1 shows, in most of the cases the resulted damage is broken security except in two cases.

## 9 Conclusion and Future Work

In conclusion, we introduce the first taxonomy of downgrade attacks in the TLS protocol and application protocols using TLS. Our taxonomy classifies downgrade attacks with respect to four vectors: element, vulnerability, method, and damage. It is based on a through analysis of fifteen TLS downgrade attack cases under the assumption of an external man-in-the-middle attacker model. In

addition, we provided a brief survey of all notable published TLS downgrade attacks to date. Finally, we demonstrate our taxonomy's application in classifying known TLS downgrade attacks. For future work, we plan to test the taxonomy on downgrade attacks in protocols other than TLS for potential generalisation of the taxonomy. Furthermore, we believe that the taxonomy has the potential of serving as a useful tool in devising downgrade attack severity assessment model, which can enable ranking the attack severity, which can help in identifying the attacks that require more research efforts to mitigate them.

## 10 Acknowledgment

The authors would like to thank Prof. Kenny Paterson, Prof. Andrew Martin, and Nicholas Moore for their feedback, and Mary Bispham, Ilias Giechaskiel, Jacqueline Eggenschwiler, and John Gallacher for proofreading earlier versions of this paper.

## References

1. CVE-2015-3197 (2015), https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3197
2. Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J.A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S., Zimmermann, P.: Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In: Proceedings of Conference on Computer and Communications Security (CCS). pp. 5–17 (2015)
3. Aviram, N., Schinzel, S., Somorovsky, J., Heninger, N., Dankel, M., Steube, J., Valenta, L., Adrian, D., Halderman, J.A., Dukhovni, V., Käsper, E., Cohney, S., Engels, S., Paar, C., Shavitt, Y.: DROWN: Breaking TLS Using SSLv2. In: Proceedings of USENIX Security Symposium. pp. 689–706 (2016)
4. Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y., Zinzindohoue, J.K.: A Messy State of the Union: Taming the Composite State Machines of TLS. In: Proceedings of IEEE Symposium on Security and Privacy (SP). pp. 535–552 (2015)
5. Bhargavan, K., Brzuska, C., Fournet, C., Green, M., Kohlweiss, M., Zanella-Béguelin, S.: Downgrade Resilience in Key-Exchange Protocols. In: Proceedings of IEEE Symposium on Security and Privacy (SP). pp. 506–525 (2016)
6. Bhargavan, K., Leurent, G.: Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH. In: Proceedings of Network and Distributed System Security Symposium (NDSS) (2016)
7. Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., Moeller, B.: Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) (2006), https://tools.ietf.org/html/rfc4492
8. Bleichenbacher, D.: Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS-1. In: Proceedings of Advances in Cryptology (CRYPTO 98). pp. 1–12. Springer (1998)
9. Bursztein, E.: Understanding How TLS Downgrade Attacks Prevent Email Encryption (2015), https://www.elie.net/blog/understanding-how-tls-downgrade-attacks-prevent-email-encryption

10. Clark, J., van Oorschot, P.C.: SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements. In: Proceedings of IEEE Symposium on Security and Privacy (SP). pp. 511–525 (2013)
11. Dierks, T., Allen, C.: The TLS Protocol Version 1.0 (1999), https://www.ietf.org/rfc/rfc2246.txt
12. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.1 (2006), https://tools.ietf.org/html/rfc4346
13. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2 (2008), https://tools.ietf.org/html/rfc5246
14. Diffie, W., Hellman, M.: New Directions in Cryptography. IEEE Transactions on Information Theory 22(6), 644–654 (1976)
15. Dukhovni, V.: Opportunistic Security: Some Protection Most of the Time (2014), https://tools.ietf.org/html/rfc7435
16. Durumeric, Z., Ma, Z., Springall, D., Barnes, R., Sullivan, N., Bursztein, E., Bailey, M., Halderman, J., Paxson, V.: The Security Impact of HTTPS Interception. In: Proceedings of Network and Distributed Systems Symposium (NDSS) (2017)
17. Durumeric, Z., Adrian, D., Mirian, A., Kasten, J., Bursztein, E., Lidzborski, N., Thomas, K., Eranti, V., Bailey, M., Halderman, J.A.: Neither Snow Nor Rain Nor MITM...: An Empirical Analysis of Email Delivery Security. In: Proceedings of Internet Measurement Conference (IMC). pp. 27–39 (2015)
18. Freier, A., Karlton, P., Kocher, P.: The Secure Sockets Layer (SSL) Protocol Version 3.0 (2011), https://tools.ietf.org/html/rfc6101
19. Hickman, K.: SSL 0.2 Protocol Specification (2008), http://www-archive.mozilla.org/projects/security/pki/nss/ssl/draft02.html
20. Hoffman, P.: SMTP Service Extension for Secure SMTP over Transport Layer Security (2002), https://tools.ietf.org/html/rfc3207
21. Howard, J.D., Longstaff, T.A.: A Common Language for Computer Security Incidents. Sandia National Laboratories 10 (1998)
22. Klensin, J.: Simple Mail Transfer Protocol (2001), https://www.ietf.org/rfc/rfc2821.txt
23. Langley, A., Modadugu, N., Moeller, B.: Transport Layer Security (TLS) False Start (2016), https://tools.ietf.org/html/rfc7918
24. Mavrogiannopoulos, N., Vercauteren, F., Velichkov, V., Preneel, B.: A Cross-protocol Attack on the TLS Protocol. In: Proceedings of Conference on Computer and Communications Security (CCS). pp. 62–72 (2012)
25. Menezes, A.J., Van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC press (1996)
26. Meyer, C., Schwenk, J.: SoK: Lessons Learned from SSL/TLS Attacks. In: Proceedings of International Workshop on Information Security Applications. pp. 189–209 (2013)
27. Möller, B., Duong, T., Kotowicz, K.: This POODLE Bites: Exploiting the SSL 3.0 Fallback (2014), https://www.openssl.org/~bodo/ssl-poodle.pdf
28. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3 draft-ietf-tls-tls13-10 (2015), https://tools.ietf.org/html/draft-ietf-tls-tls13-10
29. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3 draft-ietf-tls-tls13-25 (2018), https://tools.ietf.org/html/draft-ietf-tls-tls13-25
30. Rivest, R.L., Shamir, A., Adleman, L.: A Method for Obtaining Digital Signatures and Public-key Cryptosystems. Commun. ACM 21(2), 120–126 (1978)
31. Stricot-Tarboton, S., Chaisiri, S., Ko, R.K.: Taxonomy of Man-In-The-Middle Attacks on HTTPS. In: Proceedings of IEEE Trustcom/BigDataSE/ISPA. pp. 527–534 (2016)

32. Turner, S., Polk, T.: Prohibiting Secure Sockets Layer (SSL) Version 2.0 (2011), https://tools.ietf.org/html/rfc6176
33. Wagner, D., Schneier, B.: Analysis of the SSL 3.0 Protocol. In: Proceedings of USENIX Workshop on Electronic Commerce (EC 96). pp. 29–40 (1996)
34. Wikipedia: Export of Cryptography from the United States (2017), https://en.wikipedia.org/wiki/Export_of_cryptography_from_the_United_States

# Appendix A The TLS Protocol

## A.1 TLS, a General Overview

The main goal of TLS is to provide a secure communication channel between two communicating parties [13], ideally client (initiator $I$) and server (responder $R$). TLS consists of two sub-protocols: the handshake protocol and the record protocol [13]. Briefly, the handshake protocol is responsible for version and ciphersuite negotiation, client and server authentication, and key exchange. On the other hand, the record protocol is responsible for carrying the protected application data, encrypted with the just negotiated keys in the handshake. As of this writing, TLS 1.2 [13] is the currently deployed standard. The coming version of TLS, TLS 1.3 [29], is still work in progress. Figure 3 shows the message sequence diagram for TLS 1.2 using Ephemeral Diffie-Hellman (EC)DHE[5] key-exchange [14], Figure 4 shows TLS 1.2 using Rivest-Shamir-Adleman (RSA) key-exchange [30], and Figure 5 illustrates the changes in the `Hello` messages in TLS 1.3 based on the latest draft (draft-25 as of this writing) [29]. Our scope in this paper is TLS in certificate-based unilateral server-authentication mode. In the diagrams, the messages are represented by their initials (e.g. `CH` refers to `ClientHello`). Throughout the paper, the protocol messages are distinguished by a `TypeWriter` font.

## A.2 TLS 1.2 Handshake Protocol

We briefly describe the TLS 1.2 handshake protocol in certificate-based unilateral server-authentication mode based on the Internet Engineering Task Force (IETF) standard's specifications [13]. A detailed description of the protocol can be found in [13]. As depicted in Figure 3, the handshake protocol works as follows: First, the client sends a `ClientHello` (`CH`) message to initiate a connection with the server. This message contains: the maximum version of TLS that the client supports ($vmax_I$); the client's random value ($n_I$); optionally, a session identifier if the session is resumed ($session_{ID}$); a list of ciphersuites that the client supports ordered by preference ($[a_1, ..., a_n]$); a list of compression methods that the client supports ordered by preference ($[c_1, ..., c_n]$); and finally, an optional list of extensions ($[e_1, ..., e_n]$).

Second, the server responds with a `ServerHello` (`SH`) message. This message contains: the server's selected TLS version ($v_R$); the server's nonce ($n_R$);

[5] We use (EC)DHE as an abbreviation for: Elliptic-Curve Ephemeral Diffie-Hellman (ECDHE) or Ephemeral Diffie-Hellman (DHE).

optionally, a session identifier in case of session resumption ($session_{ID}$); the selected ciphersuite based on the client's proposed list ($a_R$); the selected compression method from the client's proposed list ($c_R$); and optionally, a list of the extensions that are requested by the client and supported by the server ($[e_1, ..., e_n]$). After that, the server sends a `ServerCertificate` (`SC`), which contains the server's certificate ($cert_R$) if server authentication is required. Then, if the key-exchange algorithm is (EC)DHE (see [14] for details about the DH algorithm), the server sends a `ServerKeyExchange` (`SKE`) message. This message must not be sent when the key-exchange algorithm is RSA (see [30] for details about the RSA algorithm). The `ServerKeyExchange` contains the server's (EC)DHE public-key parameters and a signature over a hash of the nonces ($n_I$ and $n_R$) and the (EC)DHE key parameters. In case of DHE (i.e. Finite Field DHE), the key parameters are: the prime ($p$), the generator ($g$), and the server's public value ($g^b$). We omit describing the ECDHE parameters and we refer the reader to [7] for details about ECDHE key parameters. Finally, the server sends a `ServerHelloDone` (`SHD`) to indicate to the client that it finished its part of the key-exchange.

Third, upon receiving the `ServerHelloDone` the client should verify the server's certificate and the compatibility of the server's selected parameters in the `ServerHello`. After that, the client sends a `ClientKeyExchange` (`CKE`) to set the pre-master secret. The content of the `ClientKeyExchange` depends on the key-exchange algorithm. If the key-exchange algorithm is RSA, the client sends the pre-master secret encrypted with the server's long-term RSA public-key ($[pms]^{pk_R}$) as illustrated in Figure 4. 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 ($g^{ab}$) as illustrated in Figure 3. After that, both parties compute the master secret ($ms$) and the session keys: ($k_I$) for the client, and ($k_R$) for the server, using Pseudo Random Functions PRFs as follows: ($\text{kdf}_{\text{ms}}$) takes the $pms$ and nonces as input and produces the $ms$, while ($\text{kdf}_{\text{k}}$) takes the $ms$ and nonces as input and produces the session keys $k_I$ and $k_R$. There are more than a pair for the session keys, i.e. separate key pairs for encryption and authentication, but we abstract away from these details and refer to the session keys in general by the key pair $k_I$ and $k_R$. Finally, the client sends `ChangeCipherSpec` (`CCS`) (this message is not considered part of the handshake and is not included in the transcript hash), followed by a `ClientFinished` (`CF`) which is encrypted by the just negotiated algorithms and keys. The `ClientFinished` verifies the integrity of the handshake transcript (i.e. the *log* (We adopted the term *log* from [5]). The `ClientFinished` content is computed using a PRF which serves as a Message Authentication Code (MAC) that we denote it by (mac) over a hash of the handshake transcript starting from the `ClientHello` up to, but not including, the `ClientFinished` (i.e. mac of $log_1$ as shown in Figure 3 and Figure 4), using the $ms$ as a key. This mac needs to be verified by the server.

Fourth, similar to the client, the server sends its `ChangeCipherSpec` (`CCS`) followed by a `ServerFinished` (`SF`) that consists of a mac over a hash of the

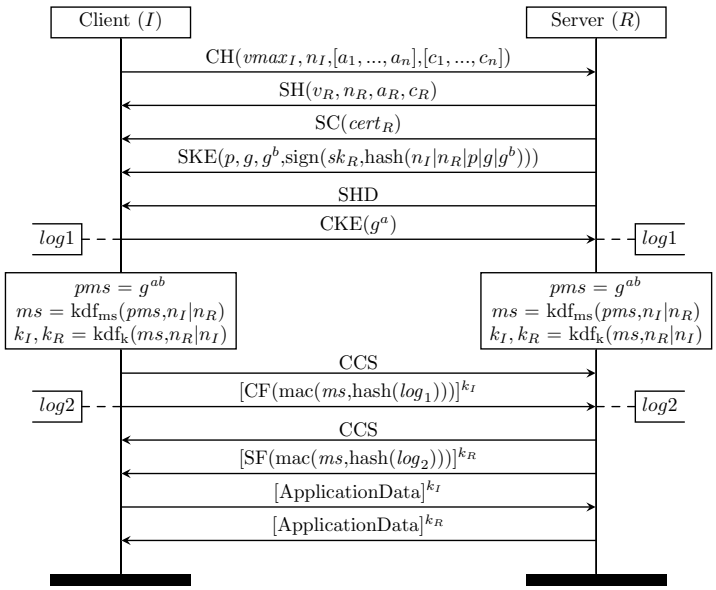


**Fig. 3:** Message sequence diagram for TLS 1.2 with (EC)DHE key-exchange.

server's transcript up to this point ($log_2$), which also needs to be verified by the client.

Once each communicating party has verified its peer's `Finished` message, they can now send and receive encrypted data using the established session keys $k_I$ and $k_R$. If "False Start" [23] is enabled, the client can send data just after its `ClientFinished`, and before it verifies the `ServerFinished`.

### A.3 TLS 1.3 Handshake, Major Changes

This section is not meant to provide a comprehensive description of TLS 1.3, but to highlight some major changes in TLS 1.3 over its predecessor TLS 1.2. Similar to the previous section, we assume certificate-based unilateral server-authentication mode. A full description of the latest draft of TLS 1.3 (as of this writing) can be found in [29]. Figure 5 illustrates the `Hello` messages in TLS 1.3, where the TLS version and algorithms are negotiated.

One of the first changes in TLS 1.3 is prohibiting all known weak and unrecommended cryptographic algorithms such as RC4 for symmetric encryption, RSA and static DH for key-exchange, etc. In addition, TLS 1.3 enforces Forward Secrecy (FS) in both modes: the full handshake mode and the session resumption mode (with the exception of the early data in the Zero Round Trip Time (0-RTT)

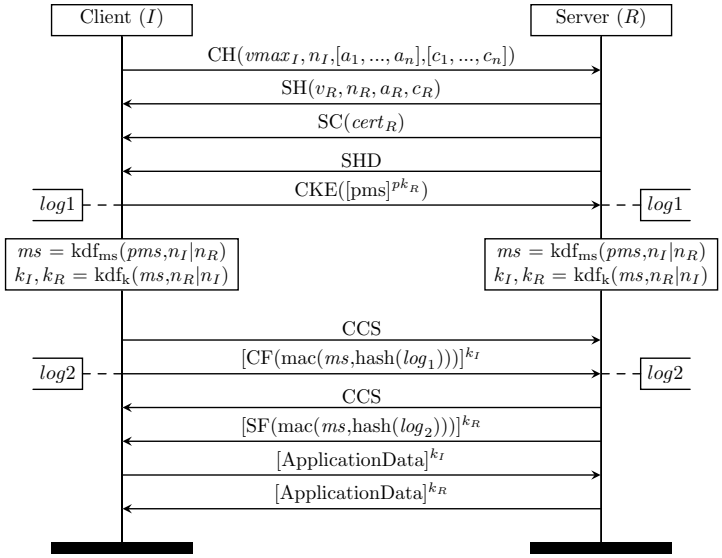


**Fig. 4:** Message sequence diagram for TLS 1.2 with RSA key-exchange.

mode that is always sent in non-FS mode), compared to TLS 1.2, where FS is optional in the full handshake mode, and not possible in the session resumption mode. It also enforces Authenticated Encryption (AE) and standard (i.e. non arbitrary) DH groups and curves. Furthermore, unlike TLS 1.2 where all handshake messages before the `Finished` messages are sent in cleartext, all TLS 1.3 handshake messages are encrypted as soon as both parties have computed shared keys, i.e. after the `ServerHello` message.

The `ClientHello` message in TLS 1.3 has major changes. First, in terms of parameters, the following parameters have been deprecated (but still included for backward compatibility): the maximum supported TLS version ($vmax_I$) has been substituted by the "supported_versions" extension ($[v_1, ..., v_n]$); the session ID ($session_{ID}$) has been substituted by the "pre_shared_key" extension; the compression methods list [$c_1$,...,$c_n$] are not used any more and sent as a single byte set to zero ($c_I$). In addition, unlike TLS 1.2 where extensions are optional, in TLS 1.3, the `ClientHello` extensions are mandatory and must at least include the "supported_versions" extension. Second, in terms of behaviour, the server can optionally respond to a `ClientHello` with a `HelloRetryRequest` (`HRR`), a newly introduced message in TLS 1.3 that can be sent from server to client to request a new (EC)DHE group that has not been offered in the client's "key_share" extension ([...,($G_I, g^i$),...]) which is a list of "key_share" entries ("KeyShareEntry") ordered by preference, but is supported in the client's "supported_groups"

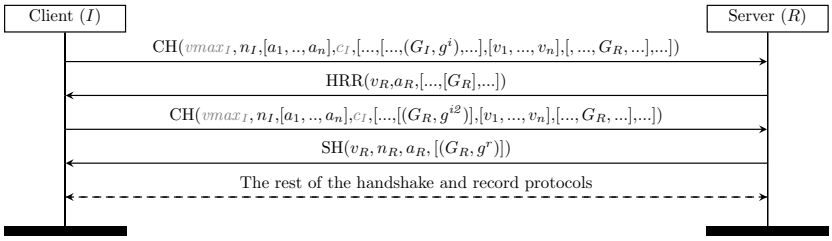


**Fig. 5:** Message sequence diagram for TLS 1.3 `Hello` messages with DHE key-exchange and `HelloRetryRequest`. Deprecated parameters that are included for backward compatibility are marked with gray color.

extension ([...,$G_R$,...]). The `HelloRetryRequest` can also be sent if the client has not sent any "key_share". After the `HelloRetryRequest`, the client sends a second `ClientHello` with the server's requested "key_share" ($[G_R, g^{i2}]$).

Upon receiving a `ClientHello`, if the client's offered parameters are supported by the server, the server responds with a `ServerHello` message. The `ServerHello` has two major changes: First, unlike TLS 1.2 where the extensions field is optional, in TLS 1.3, the `ServerHello` must contain at least the "key_share" or "pre_shared_key" extensions (the latter is sent in case of session resumption which is beyond our paper's scope). Second, as a version downgrade attack defence mechanism (in addition to other mechanisms), the last eight bytes of the server's nonce $n_R$ are set to a fixed value that signals the TLS version that the server has received from the client. This allows the client to verify that the versions that were sent in the `ClientHello` have been received correctly by the server. This is because the nonces are signed in the TLS 1.3 `CertificateVerify` and in the TLS 1.2 `ServerKeyExchnage` as well.

Finally, the TLS 1.2 `ServerKeyExchange` is not used in TLS 1.3. This is a result of shifting the key-exchange to the `Hello` messages, namely to the "key_share" and "pre_shared_key" extensions. The signature over the key parameters that is sent in the `ServerKeyExchange` in TLS 1.2 to authenticate the server's key parameters is now sent in a new message, namely the `ServerCertificateVerify` which is sent after the server's `Certificate` message. Most importantly, the signature in the `ServerCertificateVerify` is computed over a hash of the full transcript from the `Hello` messages up to the `Certificate`, and not only over the key parameters as in TLS 1.2 `ServerKeyExchange`. The signature over the full transcript provides protection against downgrade attacks that exploit the lack of ciphersuite authentication in the `ServerKeyExchange` as demonstrated in [2] and [4].