# DSTC: DNS-based Strict TLS Configurations

Eman Salem Alashwali[1,2(✉)] and Pawel Szalachowski[3]

[1] University of Oxford, United Kingdom
[2] King Abdulaziz University (KAU), Saudi Arabia
eman.alashwali@cs.ox.ac.uk
[3] Singapore University of Technology and Design (SUTD), Singapore
pawel@sutd.edu.sg

**Abstract.** Most TLS clients such as modern web browsers enforce coarse-grained TLS security configurations. They support legacy versions of the protocol that have known design weaknesses, and weak ciphersuites that provide fewer security guarantees (e.g. non Forward-Secrecy), mainly to provide backward compatibility. This opens doors to downgrade attacks, as is the case of the POODLE attack [18], which exploits the client's silent fallback to downgrade the protocol version to exploit the legacy version's flaws. To achieve a better balance between security and backward compatibility, we propose a DNS-based mechanism that enables TLS servers to advertise their support for the latest version of the protocol and strong ciphersuites (that provide Forward-Secrecy and Authenticated-Encryption simultaneously). This enables clients to consider prior knowledge about the servers' TLS configurations to enforce a fine-grained TLS configurations policy. That is, the client enforces *strict* TLS configurations for connections going to the advertising servers, while enforcing *default* configurations for the rest of the connections. We implement and evaluate the proposed mechanism and show that it is feasible, and incurs minimal overhead. Furthermore, we conduct a TLS scan for the top 10,000 most visited websites globally, and show that most of the websites can benefit from our mechanism.

## 1 Introduction

Websites[1] vary in the sensitivity of the content they serve and in the level of communication security they require. For example, a connection to an e-banking website to make a financial transaction carries more sensitive data than a connection to an ordinary website to view public news. A close look at how mainstream TLS clients (e.g. web browsers) treat these differences reveals that they enforce coarse-grained TLS security configurations, i.e. a "one-size-fits-all" policy. They[2] support legacy versions of the protocol that have known design

[1] Throughout the paper we use the terms website, server, and domain, interchangeably to refer to an entity that offers a service or content on the Internet.

[2] We tested the following browsers: `Google Chrome` version 67.0.3396.87, `Mozilla Firefox` version 60.0.2, `Microsoft Internet Explorer` version 11.112.17134.0, `Microsoft Edge` version 42.17134.1.0, and `Opera` version 53.0.2907.99.

weaknesses and weak ciphersuites that provide fewer security guarantees, e.g. non Forward-Secrecy (non-FS), and non Authenticated-Encryption (non-AE), mainly for backward compatibility.

Supporting legacy versions or weak ciphersuites provides backward compatibility, but opens doors to downgrade attacks. In downgrade attacks, an active Man-in-the-Middle (MitM) attacker forces the communicating parties to operate in a mode weaker than they both support and prefer. Several studies illustrate the practicality of downgrade attacks in TLS [1,8,9,10,11,12,18]. Despite numerous efforts to mitigate them, they continue to appear up until 2016 in a draft for the latest version of TLS, TLS 1.3 [11]. Previous attacks have exploited not only design vulnerabilities, but also implementation and trust model vulnerabilities that bypass design-level mitigations such as the handshake messages (transcript) authentication. For example, the POODLE [18], DROWN [8], and `ClientHello` fragmentation [10] downgrade attacks.

Clearly, disabling legacy TLS versions and weak ciphersuites at both ends prevents downgrade attacks: There is no choice but the latest version and strong ciphersuites. However, the global and heterogeneous nature of the Internet have led both parties (TLS client vendors and server administrators) to compromise some level of security for backward compatibility. Furthermore, from a website perspective, supporting legacy TLS versions and weak ciphersuites may not only be a technical decision, but also a business decision not to lose customers for another website.

However, we observe that if the client has prior knowledge about the servers' TLS configurations, a better balance between security and backward compatibility can be achieved, which reduces the downgrade attack's surface. Given prior knowledge about the servers' ability to meet the latest version of the protocol and strong ciphersuites, the client can change its behaviour and enforce a *strict* TLS configurations policy when connecting to these advertising servers.

In this paper, we try to answer the following question: ***How to enable domain owners to advertise their support for the latest version of the TLS protocol and strong ciphersuites to clients in a usable and authenticated manner? This is in order to enable clients to make an informed decision on whether to enforce a strict or default TLS configurations policy before connecting to a server.***

Our contributions are as follows: First, we propose a mechanism that enables domain owners to advertise their support for the latest version of the TLS protocol and strong ciphersuites. This enables clients to enforce *strict* TLS configurations when connecting to the advertising domains while enforcing *default* configurations for the rest of the domains. We show how our mechanism augments clients' security to detect certain types of downgrade attacks and server misconfiguration. Second, we implement and evaluate a proof-of-concept for the proposed mechanism. Finally, we examine the applicability of our mechanism in real-world deployment by conducting a TLS scan for the top 10,000 most visited websites globally on the Internet.

## 2 Background

### 2.1 Domain Name System (DNS)

Domain Name System (DNS) [17] is a decentralized and hierarchical naming system that stores and manages information about domains. DNS introduces different types of information which are stored in dedicated resource records. For example, the `A` resource records are used to point a domain name to an IPv4 address, while `TXT` records are introduced for storing arbitrary human-readable textual information. DNS is primarily used for resolving domain names to IP addresses, and usually this process precedes the communication between hosts. Whenever a client wants to find an IP address of a domain, for example "www.example.com", it contacts the DNS infrastructure that resolves this name recursively. Namely, first, a DNS root server is contacted to localize an authoritative server for "com", then this server helps to localize "example.com"'s authoritative server, which at the end returns the address of the target domain. To make this process more efficient, the DNS infrastructure employs different caching strategies.

### 2.2 Domain Name System Security Extension (DNSSEC)

DNS itself does not provide (and was never designed to provide) any protection of the resource records returned to clients. DNS responses can be freely manipulated by MitM attackers. DNS Security Extensions (DNSSEC) [7] is an extension of DNS which aims to improve this state. DNSSEC protects DNS records by adding cryptographic signatures to assert their origin authentication. In DNSSEC, each DNS zone has its Zone Signing Key (`ZSK`) pair. The `ZSK`'s private-key is used to sign the DNS records. Signatures are published in DNS via dedicated `RRSIG` resource records. The `ZSK` public-key is also published in DNS in the special `DNSKEY` record. The `DNSKEY` record is also signed with the private-key of a Key Signing Key (`KSK`) pair, which is signed by an upper-level `ZSK` (forming a trust chain). To validate authentication of the DNS received responses, clients have to follow the trust chain till the root.

### 2.3 Transport Layer Security (TLS)

Transport Layer Security (TLS) is one of the most important and widely-deployed client-server protocols that provides confidentiality and data integrity on the Internet. It was formerly known as the Secure Socket Layer (SSL). TLS consists of multiple sub-protocols including the TLS handshake protocol that is used for establishing TLS connections. A particularly important and security-sensitive aspect of the handshake is the selection of the protocol version and the cryptographic algorithms with their parameters (i.e. ciphersuites). Every new version of TLS prevents security attacks in previous versions. Some ciphersuites provide more security guarantees than others. For example, Forward Secrecy (FS) is a property that guarantees that a compromised long-term private key does not

compromise past session keys [16]. Both finite-field Ephemeral Diffie-Hellman (`DHE`) and Elliptic-Curve Diffie-Hellman (`ECDHE`) key-exchange algorithms provide the FS property. On the other hand, RSA does not provide this property. Similarly, Authenticated Encryption (AE) provides confidentiality, integrity, and authenticity simultaneously such that they are resilient against padding oracle attacks [27][29]. `GCM`, `CCM`, and `ChaCha-Poly1305` ciphers provide the AE property while the `CBC` MAC-then-Encrypt ciphers do not provide authentication and encryption simultaneously, and hence do not provide the AE property.

### 2.4 TLS Version and Ciphersuite Negotiation

We base our description on TLS 1.2 [24]. The coming version TLS 1.3 is still a draft [25]. At the beginning of a new TLS handshake, the client sends a `ClientHello` (`CH`) message to the server. The `ClientHello` contains several parameters including the supported versions and ciphersuites. In TLS 1.2 the client sends its supported versions as a single value which is the maximum supported version by the client $vmax_C$, while in TLS 1.3, they are sent as a list of supported versions $[v_1, ..., v_n]$ in the `supported_versions` extension. The $vmax_C$ is still included in TLS 1.3 `ClientHello` for backward compatibility and its value is set to TLS 1.2. The `supported_versions` extension is not for pre TLS 1.3 versions [25]. The client's supported ciphersuites are sent as a list $[a_1, ..., a_n]$. Upon receiving a `ClientHello`, the server selects the version and ciphersuite that will be used in that session, and responds with a `ServerHello` (`SH`) containing the selected version $v_S$ and the selected ciphersuite $a_S$. Ideally, these two values are influenced by the client's offered versions and ciphersuites. If the server selected a version lower than the client's maximum version, most TLS clients fall back silently to the lower versions (up to TLS 1.0 in all mainstream browsers today). The silent fallback mechanism can be abused by attackers to perform downgrade attacks as shown in the POODLE [18], a variant of DROWN [8], and ClientHello fragmentation [10] downgrade attacks.

### 2.5 TLS Downgrade Attacks

In a typical downgrade attack, an active MitM attacker interferes with the protocol messages leading the communicating parties to operate in a mode weaker than they both support and prefer. Downgrade attacks have existed since the very early versions of TLS, SSL v2 [30]. They can exploit various types of vulnerabilities (design, implementation, or trust-model), and target various elements of the protocol (algorithm, version, or layer) [3]. In the absence of handshake transcript authentication, downgrade attacks can be trivially performed. Starting from SSL v3, the handshake transcript is authenticated at the end of the handshake to prevent downgrade attacks. However, experience has shown a series of downgrade attacks that circumvent the handshake transcript authentication. For example, [9][1][18][10]. Figure 1 shows version downgrade as in the POODLE [18] attack.

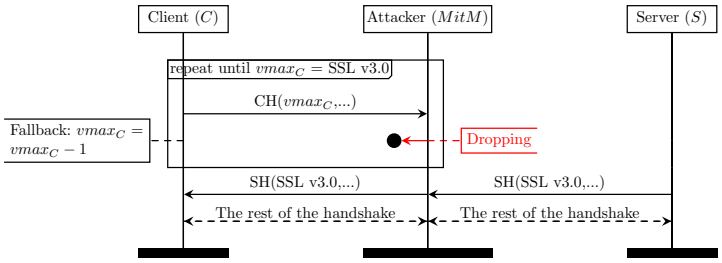


Fig. 1: Version downgrade in the POODLE attack [18].

# 3 Preliminaries

## 3.1 Strict versus Default TLS Policy

Our mechanism affects the client's fine-grained TLS configurations. Namely, the protocol version and ciphersuites. In addition, it affects the client's fallback mechanism. In our proposed mechanism, there are two pre-defined policies (or contexts) for the TLS client configurations: *strict* and *default*. The *strict* policy enforces strong TLS configurations and disables the fallback. We define strong TLS configurations as those that support only the latest version of the protocol and only strong ciphersuites. We define strong ciphersuites as those that support both FS and AE properties simultaneously. The fallback is a mechanism that instructs the client to retry the handshake with weak configurations if the handshake with strong configurations has failed. On the other hand, the *default* policy enforces both strong and weak TLS configurations, and enables the fallback. Weak configurations are defined as those that support both the latest and the legacy versions of the protocol, and both strong and weak ciphersuites. Weak ciphersuites are defined as those that support non-FS or non-AE. Table 1 summarises the *strict* versus *default* policies that we define in our mechanism. Our prototypical TLS client implementation supports TLS versions: 1.0, 1.1, and 1.2, and 14 ciphersuites (similar to those supported in `Firefox` browser version 60.0.2 except that our client does not support the DES ciphersuite). Although TLS 1.3 is present, in our implementation and evaluation (section 6) we consider TLS 1.2 as the latest TLS version. The reason is that TLS is currently in a transition state from version TLS 1.2 to TLS 1.3. TLS 1.3 has not been officially approved as a standard (is still a draft [25]), and is still in its beta version in most mainstream implementations such as `OpenSSL`. However, this does not affect our concept in general as it is applicable to the current deployment where TLS 1.2 is the latest version. Finally we note that in TLS 1.3, FS and AE ciphersuites are enforced by design [25], i.e. strong ciphersuites are implied by TLS 1.3 as a version. Therefore, in TLS 1.3, the *strict* configurations policy boils down to the protocol version and the fallback mechanism. However, there is still a value in our mechanism's ciphersuites policy even in TLS 1.3. Our policy enforces the

Table 1: The *strict* versus *default* TLS policies that we define in our DSTC mechanism (✓denotes enabled and ✗denotes disabled).

| Policy | TLS Version | TLS Ciphersuites | Fallback |
|---|---|---|---|
| *Strict* | TLS 1.3 | FS and AE | ✗ |
| *Default* | TLS 1.3; TLS 1.2; TLS 1.1; TLS 1.0 | FS; AE; non-FS; non-AE | ✓ |

client to refine its ciphersuites before the `ClientHello` is sent which provides downgrade resilience even when the server is flawed. This is unlike most TLS 1.3 clients, weak and strong ciphersuites are sent in the `ClientHello`, relying on the server to select the right version and ciphersuite. Experience shows that servers' flaws can be exploited to make the server select the wrong version as in `ClientHello` fragmentation [10].

### 3.2 Problem Statement

Achieving both security and backward compatibility is challenging. A *strict* TLS client configurations policy provides stronger downgrade resilience than the *default* one. However, the *strict* policy may render many ordinary legacy servers unnecessarily unreachable, which results in a difficult user experience. On the other hand, the *default* policy (such as mainstream web browsers today), provide backward compatibility but this is achieved at the cost of security. Experience shows that the *default* policy can be abused by attackers to perform downgrade attacks as shown in the POODLE attack [18]. ***Can we achieve a better balance between the two extremes? Can we enable clients to enforce fine-grained TLS configurations based on prior knowledge about the servers' TLS configurations? Can we design a usable and authenticated mechanism that allows servers to advertise their support for strong TLS configurations so that clients can enforce a strict TLS configurations policy for connections going to these servers while enforcing a default configurations policy for the rest of the connections?***

### 3.3 System and Threat Models

Our system model considers the following parties: a TLS client, a TLS server, and a DNS server. A TLS server is identified by its domain name, and the domain owner controls its DNS zone. These parties are standard for TLS connections and are assumed to be honest. As is the case of most real-world systems, the client and server support multiple protocol versions and ciphersuites that vary in the security guarantees they provide. Some of the versions and ciphersuites that the client and server support are weak, and are supported by both parties to be used *if and only if* their peer is indeed a legacy one that does not support the strong configurations. The client and server aim to establish a TLS session using strong configurations. For example, if both parties support the latest version of the protocol (as of this writing, TLS 1.3), then both parties aim to use TLS 1.3.

The DNS supports DNSSEC and uses strong signature algorithms and strong keys to sign the zone file which contains all the DNS records. The DNS keys are authenticated keys through a chain of trust in the DNS hierarchy.

In terms of threat model, we consider a MitM attacker who can passively eavesdrop on the transmitted messages, as well as actively modify, inject, drop, and replay messages during transmission. The attacker cannot break sufficiently strong cryptographic primitives (e.g. RSA signatures with 2048 bit ore more) that are properly deployed. The attacker does not have access to the DNS private-key that is used to sign the DNS zone file. We also assume the absence of MitM attackers in the first connection from the client to the DNS server for each domain. However, the MitM can exist in subsequent connections from the client to the DNS server.

### 3.4 System Goals

Our system goals can be summarised as follows:

- Authentication: TLS clients should be able to verify that the statement advertising the domain's support for the strong TLS configurations in the DNS is genuinely produced by the domain owner.
- Usability: The mechanism should be usable to the clients' end users. It should not incur additional manual configurations on the users.
- Compatibility: The mechanism should be compatible with existing Internet infrastructure. It should not require additional infrastructure or trusted third parties above those in a typical TLS connection.
- Performance: The mechanism should be lightweight. It should incur minimal overhead on the clients' performance.

## 4 The DSTC Mechanism

### 4.1 Overview

Our mechanism aims to provide a usable and authenticated method that allows domain owners to advertise their support for strong TLS configurations to TLS clients. This provides the clients with prior knowledge that enables them to take an informed decision on whether to enforce a *strict* or *default* TLS configurations policy, before connecting to a domain. Throughout the paper, we refer to the DSTC record in the DNS as the DSTC policy record.

### 4.2 DSTC Policy Syntax

In what follows, we describe each directive used in the DSTC policy syntax. Figure 2 shows an example of an ideal DSTC record in a DNS `zone` file.

- `name:` Specifies an identifier for the DSTC records. Our mechanism uses a general purpose DNS record (`TXT`). Therefore, the record must be identified as a DSTC to be interpreted by clients as a DSTC policy record. This directive value must be set to `DSTC`.

```
tls12  IN TXT
"name:DSTC;validFrom:01-06-2018;validTo:01-06-2019;tlsLevel:strict-config;
includeSubDomain:0;revoke:0;report:config-errors@tls12.com"
```

Fig. 2: An example of a DSTC record in the DNS for the domain "tls12".

- `validFrom:` Specifies the DSTC policy issuance date. It indicates the recency of the policy. It acts as a version number for the policy when there are multiple issued policies. The most recent must be the effective one. This directive value takes a date in a `dd-mm-yyy` format.
- `validTo:` Specifies the DSTC policy expiry date. It indicates the validity of the policy. This directive value takes a date in a `dd-mm-yyy` format.
- `tlsLevel:` Specifies the TLS level that the server advertises. This directive value must be set to `strict-config` for the *strict* TLS configurations policy to be enforced by the client.
- `includeSubDomain:` Specifies whether the policy should be enforced to sub-domains or not. It takes either `0` to disable the option or `1` to enable it.
- `revoke:` Specifies whether the domain wants to opt-out from the DSTC policy or not. It takes either `0` to disable the option or `1` to enable it. If enabled, it acts as a poisoning flag. When a server wants to opt-out from the DSTC, it should keep advertising a `revoke` with value `1` until the expiry date of any previously published DSTC policy. This instructs clients to delete the revoked DSTC from their storage if exists.
- `report:` Specifies the email address of the domain owner. It takes a string in an email address format. The email can be used by TLS clients to allow the user to report a domain's failure of complying with the advertised policy to the domain owner.

### 4.3 Details

The mechanism can be summarised in three main phases as follows:

1. **Policy Registration:**
   (a) The policy must be defined by the domain owner according to the policy syntax in section section 4.2.
   (b) The policy needs to be published as a `TXT` record in the DNS by the domain owner.
   (c) The policy needs to be signed by the domain owner using the private-key of the `ZSK`. By the end of this step, the signed DSTC policy is publicized in the DNS in the domain's `TXT` record.

2. **Policy Query and Verification:**
   (a) When a client wants to connect to a website, the client queries the DNS to retrieve the domain's DNS records. The DSTC is returned in a signed `TXT` record.
   (b) The client verifies the signature using an authenticated public-key of the `ZSK`. If the signature is valid, the client verifies the rest of the DSTC

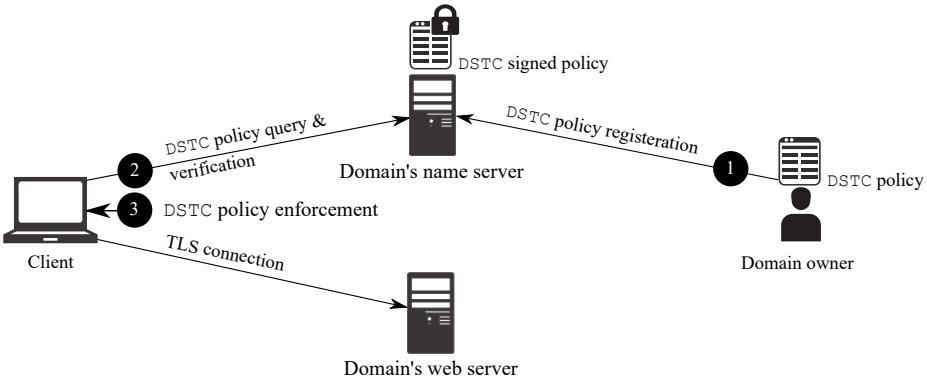


Fig. 3: A high-level overview of the DSTC mechanism.

policy directives. Based on the verification result, this step returns a value that signals the TLS configuration policy to be enforced: either *strict* or *default* along with a message to clarify the status (e.g. invalid signature) and the reporting email. The *strict* policy is returned only when all the verifications pass. Otherwise, the policy remains *default*.

3. **Policy Enforcement:**
   (a) The client receives the TLS configuration policy from the previous step (Query and Verification).
   (b) The client enforces the policy according to the policy received: either *strict* or *default*.

After the TLS configurations policy is enforced, which affects the TLS `ClientHello` offered versions and ciphersuites parameters, the client connects to the server. Figure 3 illustrates the DSTC system and the actors involved. The TLS connection is not part of our mechanism phases, but we include it in Figure 3 to provide a complete view of the system.

## 5 Security Analysis

In our system, the attacker wins under two conditions: First, if he can forge a DSTC policy and present it to a DSTC-supported TLS client as a valid policy. Second, if he can perform an undetectable TLS version or ciphersuite downgrade attack that makes a DSTC-supported TLS client accept weak TLS configurations despite the downgrade-resilience that the DSTC policy provides.

### 5.1 DSTC Forgery

An active MitM attacker can achieve DSTC forgery if he can add, modify, delete, drop, or replay a DSTC policy record for a particular domain. The attacker's gain from each method can be summarised as follows: First, adding a policy for

a domain that did not register a DSTC policy can cause a Denial of Service (DoS) attack for that domain. When DSTC-supported clients enforce a *strict* configurations policy for a domain that actually did not register a DSTC record and does not comply with the policy's requirements (e.g. uses a legacy protocol version), this will result in aborted handshake by the client. Second, modifying a DSTC policy record's directives can cause either DoS or Denial of Policy (DoP) for the concerned domain, depending on the modified directive. DoP prevents a policy from being enforced despite the domain's registration, which results in *default* client configurations which in turn provides weaker downgrade-resilience than desired. For example, modifying the `validTo` directive to an earlier date than it actually is, results in DoP since the policy will be marked as expired by the client at some point of time, and will not be enforced, while it is expected to be enforced by the domain. On the other hand, modifying the `validTo` directive to a later date results in DoS since the policy will be enforced for a domain that is not advertising the policy and may no longer complying with it. Third, deleting a DSTC policy record will result in DoP since the client does not get the DSTC record and enforces the *default* TLS configurations, which provides weaker downgrade-resilience. Fourth, replaying a non recent or revoked policy that has a valid signature can cause a DoS or DoP attacks as explained above.

In our system, adding, modifying, or deleting a DSTC policy record for a domain is defeated by the digital signature. The DNSSEC is a mandatory component of the system where DSTC records are signed by the domain owner using the private-key of the `ZSK`. The attacker does not have access to the DNS private-key and does not have the power to break it or break the signature algorithm. Regarding replay attacks, the client stores the policy locally and updates or revokes (deletes) it when a signed, more recent (i.e. more recent `validFrom` date), and non-expired policy is received. A replayed outdated or revoked policy will have a less recent issuance date than the stored one, and hence will be detected even if it has a valid signature. Finally, dropping attacks are also defeated by the stored policy from the first connection which is received under the assumption of the absence of MitM in the first connection from client to DNS. If the client has a non-expired stored policy, and the client has not received any new `revoke`-enabled policy to instruct the client to delete it, the absence of the DSTC record in subsequent DNS queries signals a DSTC dropping attack. Note that connections after the stored DSTC policy expires are considered a first connection and assumed to be in a MitM-free connection.

### 5.2 TLS Downgrade Attacks

We now show how the DSTC mechanism prevents a class of downgrade attacks that abuse the client's support for legacy configurations and silent fallback. We demonstrate it on real-world downgrade attack scenarios.

The first scenario is inspired by the `ClientHello` fragmentation version downgrade attack [10]. In this attack, due to a flawed TLS server implementation, if an attacker fragments the `ClientHello`, the server falls back to TLS 1.0. A *default* client will silently fall back to TLS 1.0 under the assumption that

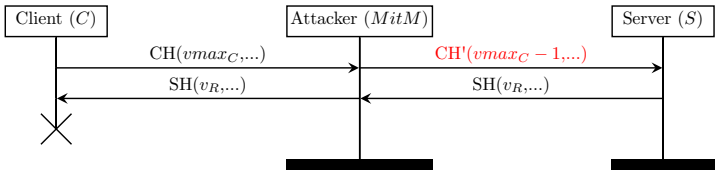


Fig. 4: Illustration of a version downgrade attack with a DSTC-supported client.

it is connecting to a legacy server. However, with a DSTC-supported client and registered server, this attack is defeated as the client enforces a *strict* TLS policy and does not fallback, hence the attack will be detected and the handshake will be aborted.

The second scenario is inspired by the POODLE version downgrade attack [18]. In this attack, the attacker drops the `ClientHello` message one or more times. Some TLS clients interpret this as a server compatibility issue and retry to send the `ClientHello` using a lower version. With a DSTC-supported client, the client does not fallback since it has prior knowledge about the server's support for strong configurations, hence the attack will be detected and the handshake will be aborted.

## 6 Implementation and Evaluation

### 6.1 Applicability

To get an insight into the applicability of our proposed mechanism, we conduct a TLS scan (IPv4 space) for the top 10,000 most visited Internet domains globally. The scan provides quantitative data about the supported and preferred TLS versions and ciphersuites in real-world servers. We retrieve the top 10,000 domains list[3] from Alexa Internet [5] on the 5$^{th}$ of May 2018. To run the scan, we use `sslscan 1.11.11` [23], a state-of-the-art open source TLS scanning tool that can perform TLS versions and ciphersuites enumeration through multiple TLS handshakes. The tool supports SSLv2 up to TLS 1.2, and 175 ciphersuites. We run the scan from the SUTD university's campus wired network between the 6$^{th}$ and 12$^{th}$ of May 2018. In terms of ethical considerations, our scan does not collect any private or personal data. The TLS versions and ciphersuites are public data which can be viewed by TLS clients through TLS handshakes. The number of handshakes the tool performs does not represent a danger of DoS.

The total number of servers that completed a successful TLS handshake with one or more TLS versions and ciphersuites is 7080 (70.80%). We do not investigate the reasons of handshake failure as this is outside our scope. However,

[3] The list gets updated daily, according to Alexa's support (in a private communication).

a recent study that performed domain name-based TLS scans for various domains [4], reports 55.7 million and 58.0 million successful TLS handshakes out of 192.9 million input domains (29.48% on average). Given the fact that our scan is for top domains, our TLS response rate sounds normal. However, one possible contributing factor to the handshake failure in our scan can be due to SUTD university's Internet censorship system that blocks some website categories such as porn and gambling.

In terms of TLS versions, of the responding servers in our results, there are 6888 (97.29%) servers that support TLS 1.2. TLS 1.2 is the preferred version in all the servers that support it. However, there are only 373 (5.27%) servers that support TLS 1.2 exclusively (without any other versions). On the other hand, the number of servers that support at least two version, both TLS 1.2 and TLS 1.1, either exclusively or with other lower versions, is 6462 (91.27%). And the number of servers that support at least three versions, TLS 1.2, TLS 1.1 and TLS 1.0, either exclusively or with other lower versions, is 6202 (87.60%).

In terms of ciphersuites, we examine the servers' ciphersuites in version TLS 1.2 only. The most frequent number of supported ciphersuites (the norm) is 20 ciphersuite, which appeared in 938 servers (13.62%). To count the servers that support FS and/or AE, in each domain in our results, we labeled each supported ciphersuite by one of the following labels: FS+AE, FS+nonAE, nonFS+AE, or nonFS+nonAE. The four labels are based on the two properties: FS and AE. FS is identified by checking if the ciphersuite starts with `ECDHE` or `DHE`, while AE is identified by checking if the ciphersuite contains `GCM`, `CCM`, `CCM8`, or `ChaCha20` strings. There are 6500 (94.37%) TLS 1.2 servers containing at least one FS+AE ciphersuite, either exclusively or with other labels. We find 6483 (94.12%) TLS 1.2 servers that support non-FS or non-AE (i.e. labeled with nonFS+AE, FS+nonAE, or nonFS+nonAE) in addition to one or more FS+AE ciphersuite.

The results show that top domain servers support the strong TLS configurations. At the same time, they maintain support for weak configurations that have known weaknesses and provide fewer security guarantees. Ideally, the clients' configurations influence the servers' selected configurations. Asserting servers' strong configurations to clients adds a value by providing clients with the confidence to enforce a *strict* TLS configurations policy for connections to these servers, which reduces the downgrade attack surface as we showed in section 5.2.

### 6.2 Feasibility

To test the feasibility of our concept, we implement a Proof-of-Concept (PoC) for the mechanism. On a machine equipped with 16 GB Random Access Memory (RAM) and `Intel Core i7` 2.6 GHz processor, and runs `Windows 10` (64-bit) OS, we build a virtual private network with a virtual host-only Ethernet adapter using `VirtualBox` [19]. It includes four virtual machines: Three TLS web servers, a DNS server, and a TLS client. The web servers are equipped with 2 GB of RAM, `Intel Core i7` CPU 2.60 GHz processor, and 1000 Mbps wired network card. They run `Apache 2.4.18` [6] on `Ubuntu 16.04` (64-bit) Operating System

Table 2: Test-case scenarios carried from our `python` DSTC-supported client to TLS servers and the effect of DSTC (✓denotes DSTC registered domain and ✗denotes unregistered) on the TLS handshake (✓denotes successful and ✗denotes failed).

| No. | TLS Server Configurations | | | Successful |
|---|---|---|---|---|
| | Version | Ciphersuites Feature | DSTC | Handshake |
| 1 | TLS 1.2 | FS and AE | ✓ | ✓ |
| 2 | TLS 1.0 | non-AE | ✓ | ✗ |
| 3 | TLS 1.1 | non-AE | ✗ | ✓ |

(OS). The DNS server is similar to the web servers in specifications except that it has 4 GB RAM and runs `BIND 9.10.3` [15]. The DNS server supports DNSSEC and the zone file is signed with a 2048 RSA `ZSK`. The `ZSK` is signed with a 2048 RSA `KSK`. We assume the `KSK` is validated through a chain of trust. To evaluate a DSTC-supported client, we implement a TLS client using `Python 3.6.5` [20] and `python`'s TLS/SSL library [21] on a Linux `Ubuntu 18.04` (64-bit) OS on a device equipped with 4 GB of RAM, `Intel Core i7` CPU 2.60 GHz processor, and 1000 Mbps wired network card. The client uses `OpenSSL 1.1.0g` that is shipped with `Ubuntu 18.04`. In our PoC we assume the highest version of TLS is TLS 1.2. Therefore a DSTC-compliant server should comply to TLS 1.2 and strong ciphersuites. Our client initiates a handshake with the three TLS web servers. The servers are configured as follows: First, to represent a DSTC compliant server that has registered a DSTC record, we configure a TLS 1.2 server with strong ciphersuites, and register a DSTC policy record for it in the DNS. Second, to represent a downgrade attack or misconfigured server, we use a straight-forward method to make the server's version lower than the DSTC requirements, we configure a TLS 1.0 server and add a DSTC policy record for it. Third, to represent a server that has not registered a DSTC record which should not be affected, we configure a TLS 1.1 which does not comply with the DSTC requirements and we do not register a DSTC policy record for it.

As depicted in Table 2, the handshake with the first server succeeds as the server complies with the DSTC requirements. The handshake with the second server fails as the server fails to comply with the DSTC requirements. The handshake succeeds with the third server as the server did not register a DSTC policy record. Our experiment confirms that the concept is technically feasible.

### 6.3 Performance

To get an insight into the computational cost that our mechanism adds over an ordinary TLS connection, based on scenario 1 in Table 2 (assuming no cached policy in the client) we measure the execution time for the following functions: `SigVerify` for the DNS `TXTRRset` records signature verification, `QueryVerify` for the DNS records query and verification (which includes `SigVerify`), `Enforce` for the TLS policy enforcement based on the `QueryVerify` output, and finally, the time for the three functions together. Table 3 presents the measurements using the processor timer in `python`'s `3.6 time` module [22], which is

Table 3: The mechanism's computational overhead in milliseconds.

| No. | Function | Max. | Min. | Avg. |
|---|---|---|---|---|
| 1 | `SigVerify` | 1.40 | 0.63 | 0.72 |
| 2 | `QueryVerify` | 4.99 | 2.74 | 3.09 |
| 3 | `Enforce` | 0.86 | 0.38 | 0.41 |
| 4 | `All 3 functions` | 6.10 | 3.23 | 3.58 |

processor-wide timer. Each measurement is repeated 500 times. A TLS socket connection establishment in our client takes 8.16 ms on average (without certificate validation). The mechanism's overall average overhead costs 3.58 ms. We conclude that the computational overhead is affordable which is about 43.87% additional overhead on the TLS socket connection. Our mechanism's overhead can be considered an upper-bound as there is a room for improvements through code optimisation.

## 7 Related work

Schechter [26] proposes the HTTP Security Requirements in the Domain Name System (HTTPSSR DNS). It allows domain owners to assert their support for the TLS protocol to prevent TLS layer downgrade (a.k.a. stripping) attacks. However, experience shows that asserting TLS (as a layer only) is not sufficient. Several downgrade attacks that target TLS configurations such as the protocol version or ciphersuite as in the POODLE version downgrade [18] have been shown successful. Dukhovni and Hardaker [13] propose the DNS-based Authentication of Named Entities (DANE). It allows domain owners to bind their own CA public keys or certificates to detect faked TLS certificates to prevent domain impersonation attacks. Hallam-Baker [14] proposes the Certificate Authority Authorisation (CAA). It allows domain owners to whitelist specific Certificate Authorities (CAs) for their domains to prevent mis-issued certificates. Alashwali and Rasmusssen [2] propose client *strict* TLS configurations against whitelisted domains as a downgrade attacks defense. The domains are added either by the client's users or through servers' HTTP headers. While adding domains through the servers' headers is usable, the *strict* policy can only be enforced starting from the second connections (the first connection is configured before the headers are fetched and hence uses *default* configurations). Our scheme extends this work by leveraging DNS which allows the *strict* policy enforcement before the first connection in a usable and authenticated manner without extra effort from clients' users. Finally, Varshney and Szalachowski [28] propose a general DNS-based meta-policy framework. Overall, none of the previous work have looked at using DNS to enable domain owners to assert strong TLS configurations.

## 8 Conclusion

We propose a mechanism that allows domain owners to advertise their support for strong TLS configurations through a signed DNS record. The client interprets this record and changes its behaviour to the *strict* policy which affects the TLS version, ciphersuite, and the fallback mechanism. Our prototype implementation and its evaluation show the feasibility of our mechanism. Furthermore, our Internet scan results depict that the majority of servers are ready to benefit from the proposed mechanism.

## Acknowledgement

We thank Prof. Andrew Martin for feedback and Monica Kaminska for proofreading. Pawel's work was supported by the SUTD SRG ISTD 2017 128 grant.

## References

1. 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: Computer and Communications Security (CCS). pp. 5–17 (2015)
2. Alashwali, E., Rasmussen, K.: On the Feasibility of Fine-Grained TLS Security Configurations in Web Browsers Based on the Requested Domain Name. In: Security and Privacy in Communication Networks (SecureComm) (2018)
3. Alashwali, E., Rasmussen, K.: What's in a Downgrade? A Taxonomy of Downgrade Attacks in the TLS Protocol and Application Protocols Using TLS. In: Applications and Techniques in Cyber Security (ATCS) (2018)
4. Amann, J., Gasser, O., Scheitle, Q., Brent, L., Carle, G., Holz, R.: Mission Accomplished? HTTPS Security after Diginotar. In: Internet Measurement Conference (IMC). pp. 325–340 (2017)
5. AmazonWS: Alexa Top 1M Global Sites. `http://s3.amazonaws.com/alexa-static/top-1m.csv.zip` (2018), Accessed May 5, 2018
6. Apache: Apache HTTP Server Project. `https://httpd.apache.org` (2018), Accessed Jul. 6, 2018
7. Arends, R., Austein, R., Larson, M., Massey, D., Rose, S.: DNS Security Introduction and Requirements. `https://tools.ietf.org/html/rfc4033` (2005), Accessed Jul. 6, 2018
8. Aviram, N., Schinzel, S., Somorovsky, J., Heninger, N., Dankel, M., Steube, J., Valenta, L., Adrian, D., Halderman, J.A., Dukhovni, V., Kasper, E., Cohney, S., Engels, S., Paar, C., Shavitt, Y.: DROWN: Breaking TLS Using SSLv2. In: USENIX Security Symposium. pp. 689–706 (2016)
9. 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: Security and Privacy (SP). pp. 535–552 (2015)

10. Beurdouche, B., Delignat-Lavaud, A., Kobeissi, N., Pironti, A., Bhargavan, K.: FLEXTLS A Tool for Testing TLS Implementations. In: USENIX Workshop on Offensive Technologies (WOOT) (2014)
11. Bhargavan, K., Brzuska, C., Fournet, C., Green, M., Kohlweiss, M., Zanella-Béguelin, S.: Downgrade Resilience in Key-Exchange Protocols. In: Security and Privacy (SP). pp. 506–525 (2016)
12. Bhargavan, K., Leurent, G.: Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH. In: Network and Distributed System Security Symposium (NDSS) (2016)
13. Dukhovni, V., Hardaker, W.: The DNS-Based Authentication of Named Entities (DANE) Protocol: Updates and Operational Guidance. `https://tools.ietf.org/html/rfc7671` (2015), Accessed Jul. 6, 2018
14. Hallam-Baker, P.: DNS Certification Authority Authorization (CAA) Resource Record. `https://tools.ietf.org/html/rfc6844` (2013), Accessed Jul. 6, 2018
15. Internet Systems Consortium: Bind Open Source DNS Server. `https://www.isc.org/downloads/bind` (2018), Accessed Jul. 6, 2018
16. Menezes, A.J., Van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC press (1996)
17. Mockapetris, P.: Domain Names - Implementation and Specification. `https://tools.ietf.org/html/rfc1035` (1987), Accessed Jul. 6, 2018
18. Möller, B., Duong, T., Kotowicz, K.: This POODLE Bites: Exploiting the SSL 3.0 Fallback. `https://www.openssl.org/~bodo/ssl-poodle.pdf` (2014), Accessed Jul. 6, 2018
19. Oracle: Virtualbox. `https://www.virtualbox.org/wiki/VirtualBox` (2018), Accessed Jul. 6, 2018
20. Python: Python. `https://www.python.org` (2018), Accessed Jul. 6, 2018
21. Python: ssl - TLS/SSL Wrapper for Socket Objects. `https://docs.python.org/3.6/library/ssl.html` (2018), Accessed Jul. 6, 2018
22. Python: time-Time Access and Conversions. `https://docs.python.org/3/library/time.html` (2018), Accessed Jul. 6, 2018
23. rbsec: sslscan Tests SSL/TLS Enabled Services to Discover Supported Cipher Suites. `https://github.com/rbsec/sslscan` (2018), Accessed Jul. 6, 2018
24. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. `https://www.ietf.org/rfc/rfc5246.txt` (2008), Accessed Jul. 6, 2018
25. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3 draft-ietf-tls-tls13-28. `https://tools.ietf.org/html/draft-ietf-tls-tls13-28` (2018), Accessed Jul. 6, 2018
26. Schechter, S.: Storing HTTP Security Requirements in the Domain Name System. `https://lists.w3.org/Archives/Public/public-wsc-wg/2007Apr/att-0332/http-ssr.html` (2007), Accessed Jul. 6, 2018
27. Sullivan, N.: Padding Oracles and the Decline of CBC-mode Cipher Suites. `https://blog.cloudflare.com/padding-oracles-and-the-decline-of-cbc-mode-ciphersuites/` (2016), Accessed Jul. 6, 2018
28. Varshney, G., Szalachowski, P.: A Metapolicy Framework for Enhancing Domain Expressiveness on the Internet. In: Security and Privacy in Communication Networks (SecureComm) (2018)
29. Vaudenay, S.: Security Flaws Induced by CBC Padding-Applications to SSL, IPSEC, WTLS.... In: Theory and Applications of Cryptographic Techniques (2002)
30. Wagner, D., Schneier, B.: Analysis of the SSL 3.0 Protocol. In: USENIX Workshop on Electronic Commerce (EC). pp. 29–40 (1996)