Negotiation Transparency and Consistency in Configurable Protocols: An Empirical Investigation

Eman Salem Alashwali

Oriel College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Hilary 2020

بسم الله الرحمن الرحيم

In the name of Allah, the most merciful, the most compassionate.

To my beloved niece, Taliah Bamakhramah (2009-2013).
May her soul rest in peace.

Acknowledgements

First and foremost, all praise and thanks to Allah for granting me this life-changing opportunity. Thanks for giving me the strength and persistence throughout this journey.

I am eternally indebt to Prof. Andrew Martin, my supervisor at Oxford, and to Prof. Pawel Szalachowski, my internship supervisor at SUTD, Singapore, without both of whom, this thesis would not have seen light. I am thankful for their trust, guidance, patience, support, and countless hours of meetings and discussions. It has been a pleasure working with them.

Thanks to my viva examiners: Dr. Joss Wright and Prof. Keith Mayes; my confirmation examiners: Dr. Joss Wright and Prof. Andrew Simpson; and my transfer examiners: Prof. Gavin Lowe and Prof. Kenneth Paterson, for taking the time to read and evaluate my work.

During my work on this thesis, several people and organisations have provided me with help and support. Thanks to the Oxford CS's IT and OxCERT teams for technical support, to the Censys team for granting me researcher access to their data and for their technical support, to the tls-scan developer, Binu Ramakrishnan for technical support, Noura Alashwali for her help in designing an icon for the browser extension, Nicolas Moore for useful hints on javascript and Firefox WebExtensions programming and for feedback on early pieces of my work, Prof. Karthikeyan Bhargavan for feedback on early pieces of my work, and Gill Scrimgeour for proofreading the thesis. Thanks to the CDT in Cyber Security, the CS department, and Oriel college for financially supporting several research trips. Thanks to David Hobbs and Maureen York from the CDT, for their help and support, and to any one who helped and supported me during my time at Oriel college, the CDT, and the CS department. Thanks to the Saudi Ministry of Education and King Abdulaziz University for financially supporting my studies abroad.

Last but not least, no words can express my gratitude to my dear family, who granted me the title "Dr" years before I officially deserve. My dear parents, my beloved brothers Ali and Abdullah, sisters, nieces, nephews, sister in law, and brothers in law. Thanks for your faith in me, for always being there, and for everything.

Abstract

Configurability (also known as agility), is a protocol design framework that allows protocols to support multiple values for parameters such as the protocol version and ciphersuite. At the beginning of a new protocol session, both communicating parties, e.g. client and server, negotiate these parameters to reach a mutual agreement on optimal values for these parameters, which will be used for the rest of the session. The parameters negotiation phase is critical as it defines the security guarantees that the protocol can provide in a particular session. Hence, it has been an attractive target for downgrade attacks. While the literature has looked at the authenticity and integrity of parameters negotiation in configurable protocols to prevent downgrade attacks under the man-in-the-middle attacker model, negotiation transparency and consistency under other attacker models have been largely overlooked.

Are there unexplored attacker models that can result in a downgrade? Can a semi-trusted server discriminate against its clients without being detected? Can two clients' requests to the same server receive inconsistent security guarantees? Can we achieve a better balance between security and backward compatibility?

In this thesis we aim to answer these unexplored interrelated questions, with a focus on the TLS protocol as one of the most important and widely used configurable protocols. To this end, we first introduce a taxonomy of downgrade attacks in the TLS protocol and application protocols using TLS. Second, we define three types of negotiation models based on a new notion we introduce, which we call the "negotiation power". Third, we introduce a novel attacker model which we call the "discriminatory" model. Fourth, through a measurement-based case study on the Forward Secrecy property and the TLS protocol, we find that there are indeed servers that select non-Forward Secrecy, nevertheless they support it, proving that, in the same vein, discrimination downgrade attacks can go unnoticed. Fifth, through two measurement-based case studies in TLS and HTTPS, we quantify inconsistencies in HTTPS and TLS responses to requests that differ in subtle variables that are not expected to affect the received security guarantees. Namely, we quantify inconsistent servers' responses to requests with versus without the `www.` prefix, and to requests from different geographic locations. Finally, we examine the concept of "prior knowledge" to reduce the downgrade attacks' surface. The results of this thesis introduce transparency and consistency as needed properties in configurable protocols, and show that they are not perfectly achieved in widely used protocols today such as TLS and HTTPS.

Contents

List of Figures **xvii**

List of Tables **xix**

List of Abbreviations **xxi**

1 Introduction **1**
- 1.1 Motivation and Research Questions . . . 1
- 1.2 Contributions . . . 5
- 1.3 Published Work . . . 7
  - 1.3.1 Peer-reviewed Publications . . . 7
  - 1.3.2 Co-authorship . . . 8
  - 1.3.3 Non-peer-reviewed Pulications . . . 8
  - 1.3.4 Oral Presentations . . . 9
  - 1.3.5 Poster Presentations . . . 9
- 1.4 Ethical Considerations . . . 9
- 1.5 Organisation . . . 10

2 Background **11**
- 2.1 TLS . . . 11
  - 2.1.1 TLS, an Overview . . . 12
  - 2.1.2 The TLS Handshake Protocol . . . 12
  - 2.1.3 TLS Key Exchange Algorithms . . . 14
    - 2.1.3.1 Non-Forward Secure Key Exchange (RSA) . . . 14
    - 2.1.3.2 Forward Secure Key Exchange (EC)DHE . . . 14
- 2.2 Parameter Negotiation Models . . . 16
  - 2.2.1 Notations . . . 16
  - 2.2.2 The Three Negotiation Models . . . 17
- 2.3 Downgrade Attacks . . . 19
  - 2.3.1 Downgrade Attacks, an Illustrative Example . . . 19
  - 2.3.2 Taxonomy of Downgrade Attacks . . . 21
- 2.4 HTTP . . . 24
  - 2.4.1 HTTP, an Overview . . . 25

2.4.2 HTTP Security Headers . . . 25
2.4.3 HTTP Redirection . . . 26
2.5 DNS . . . 26
2.5.1 DNS, an Overview . . . 27
2.5.2 DNSSEC . . . 27

3 Related Work 29
3.1 Downgrade Attacks, a Survey . . . 30
3.1.1 Algorithm Downgrade . . . 30
3.1.2 Version Downgrade . . . 33
3.1.3 Layer Downgrade . . . 34
3.2 TLS Security Measurements . . . 35
3.2.1 Cryptographic Aspect . . . 36
3.2.2 Certificate Aspect . . . 38
3.2.3 Security Enhancements Aspect . . . 39
3.3 Non-FS Key Compromise . . . 39
3.3.1 Caused by Design . . . 39
3.3.2 Caused by Implementation . . . 40
3.4 Consistency Measurements . . . 40
3.4.1 Client Type Aspect . . . 40
3.4.2 Regional Aspect . . . 41
3.4.3 Vantage Point (VP) Aspect . . . 42
3.5 HTTPS Enhancement Mechanisms . . . 42
3.5.1 DNS-based . . . 43
3.5.2 Header-based . . . 43
3.5.3 Certificate-based . . . 44
3.5.4 Agent-based . . . 44
3.6 Browser Warnings . . . 44
3.6.1 Effectiveness . . . 45
3.6.2 Caveats . . . 45

4 Towards Forward Secure Internet Traffic 47
4.1 Introduction . . . 47
4.2 Dataset . . . 49
4.2.1 Top Domains . . . 49
4.2.2 Random Domains . . . 50
4.2.3 Random IPs . . . 50
4.3 Methodology . . . 51
4.3.1 Scanning Phase . . . 51
4.3.2 Inspection Phase . . . 52

4.3.3 Identifying Device Types . . . 54
4.4 Results . . . 54
4.4.1 Scanning Phase . . . 54
4.4.1.1 Responding servers . . . 55
4.4.1.2 Servers that select non-FS-Ciphersuites . . . 56
4.4.2 Inspection Phase . . . 56
4.4.2.1 Responding servers . . . 57
4.4.2.2 Servers that still select non-FS-Ciphersuites (stable) 57
4.4.2.3 Servers that select non-FS-Ciphersuites, but support FS-Ciphersuites . . . 58
4.4.2.4 Servers that select FS+non-AE-Ciphersuites after enforcing FS-Ciphersuites . . . 59
4.4.2.5 Servers that select FS+non-AE-Ciphersuite after enforcing FS-Ciphersuites, but support FS+AE-ciphersuites . . . 59
4.4.2.6 When enforcing FS-Ciphersuite causes losing the AE property . . . 59
4.4.2.7 Servers that lose the AE property after enforcing FS-Ciphersuites, but support FS+AE-ciphersuites . 60
4.5 Towards FS Internet Traffic . . . 60
4.5.1 Deprecating non-FS-Ciphersuites in TLS Clients . . . 60
4.5.2 Best Effort Forward Secrecy (BEFS) . . . 62
4.5.2.1 Overview . . . 62
4.5.2.2 The Fallback . . . 62
4.5.2.3 BEFS Security Analysis . . . 65
4.5.2.4 Best Effort Forward Secrecy and Authenticated Encryption (BESAFE) . . . 69
4.5.2.5 BEFS and BESAFE Performance . . . 70
4.5.2.6 Improved Performance Through Parallel Attempts 71
4.6 Limitations . . . 71
4.7 Conclusion . . . 72

5 Does "www." Mean Better Transport Layer Security? 73
5.1 Introduction . . . 73
5.2 Dataset . . . 75
5.2.1 Top-Domains . . . 76
5.2.2 Random-Domains . . . 76
5.3 Methodology . . . 76
5.3.1 Data Collection . . . 76

5.3.2 Data Analysis 78
5.4 Results 81
5.4.1 Responding Servers 81
5.4.2 The Difference is in the Detail 82
5.4.3 When "`www.`" Means Better TLS Security 85
5.4.4 Relationship to HTTPS Redirection 87
5.5 Limitations 88
5.6 Conclusion 88

6 Exploring HTTPS Security Inconsistencies: a Cross-Regional Perspective 91
6.1 Introduction 92
6.2 Dataset 92
6.3 Methodology 93
6.3.1 Setup 93
6.3.2 Data Collection 94
6.3.3 URLs Security Weakness Indicators 96
6.3.4 Security Headers Weakness Indicators 97
6.3.5 TLS Weakness Indicators 98
6.4 Results 99
6.4.1 Responses 99
6.4.2 HTTPS Security Inconsistencies 100
6.4.2.1 URLs Security 101
6.4.2.2 Security Headers 107
6.4.2.3 Transport Layer Security (TLS) 108
6.4.2.4 Relationship to URLs and IPs Diversity, and to Redirection Presence 111
6.5 Attack Scenarios 116
6.5.1 Region-Confusion Attack 116
6.5.2 Discrimination Attack 117
6.6 Recommendations 118
6.7 Limitations 120
6.8 Conclusion 121

7 Prior Knowledge as a Means to Defeat Downgrade Attacks 123
7.1 Introduction 124
7.2 TLS Enumeration Scan 125
7.2.1 Dataset 126
7.2.2 Methodology 126
7.2.3 Results 126

7.2.3.1 Responses . . . 126
7.2.3.2 TLS Versions and Ciphersuites . . . 126
7.2.3.3 Concluding Remarks . . . 127
7.3 Preliminaries to Our Proposed Systems . . . 127
7.3.1 Strict vs. Default TLS Policy . . . 127
7.3.2 System Model . . . 129
7.3.3 System Goal . . . 130
7.3.4 Error Handling Mechanisms in Web Browsers . . . 131
7.4 The Proposed Systems . . . 131
7.4.1 Agent-based . . . 132
7.4.1.1 System Overview . . . 132
7.4.1.2 System Details . . . 132
7.4.2 Header-based . . . 139
7.4.2.1 System Overview . . . 139
7.4.2.2 System Details . . . 139
7.4.3 DNS-based . . . 141
7.4.3.1 System Overview . . . 141
7.4.3.2 System Details . . . 142
7.4.3.3 Feasibility . . . 144
7.4.3.4 Performance . . . 145
7.5 Limitations . . . 146
7.6 Summary . . . 148

8 Discussion, Summary, and Future Work 151
8.1 Possible Reasons for Security Misconfigurations and Inconsistency . 151
8.2 Summary of Results . . . 154
8.3 Generalisation . . . 157
8.4 Future Work . . . 157

Appendices

A The TLS Protocol 163
A.1 TLS 1.2 Handshake Protocol . . . 163
A.2 TLS 1.3 Handshake, Major Changes . . . 165
A.3 The Experiments Clients' Ciphersuites . . . 169

B MySQL Queries Examples **171**
B.1 Examples for Chapter 4 MySQL Queries 171
B.1.1 Description of Tables 171
B.1.2 MySQL Queries 173
B.2 Examples for Chapter 5 MySQL Queries 173
B.2.1 Description of Tables 174
B.2.2 MySQL Queries 175
B.3 Examples for Chapter 6 MySQL Queries 176
B.3.1 Description of Tables 176
B.3.2 MySQL Queries 178

References **191**

List of Figures

2.1 Version and ciphersuite negotiation in the TLS Hello messages. . . . 13
2.2 Illustration of the RSA key exchange in pre-TLS 1.3 protocols. . . . 15
2.3 Illustration of the (EC)DHE key exchange in pre-TLS 1.3 protocols. 15
2.4 Server-dominant negotiation model. 18
2.5 Client-dominant negotiation model. 18
2.6 Equilibrium negotiation model. 18
2.7 Illustrative example of a downgrade attack. 19
2.8 A taxonomy of downgrade attacks in the TLS protocol. 21
2.9 Version downgrade attack using the dropping method. 24

4.1 A general overview of our methodology. 51
4.2 A BEFS-enabled client handshake when the server prefers to select a non-FS-ciphersuite while supporting FS-ciphersuites. 63
4.3 A BEFS-enabled client handshake when the server does not support FS-ciphersuites. 63
4.4 Illustration of our newly introduced discriminatory attacker model. 68

6.1 The case of `sendspace.com` in IN vs. the US. 103
6.2 The case of `gannett.com` in the UK vs. the US. 103
6.3 The case of `match.com` in BR vs. the US. 104
6.4 The case of `westelm.com` in the UK vs. the US. 105
6.5 The case of CSP header `gizmodo.com` in BR vs. the US. 109
6.6 The case of HSTS header `creative.com` in IN vs. the UK. 110
6.7 Diverse URLs in inconsistent vs. secure HTTPS domains. 114
6.8 DNS spoofing or poisoning exploiting regional inconsistencies. . . . 117
6.9 A discriminatory attacker model exploiting regional inconsistencies. 118

7.1 Version downgrade attack attack based on ClientHello fragmentation. 125
7.2 Inspection of a TLS 1.3 ClientHello message from a modern Firefox browser (version 71.0). 129
7.3 Illustration of a version downgrade attack prevention through strict client TLS configurations. 131
7.4 System overview of strict TLS policy using agent-based whitelisting. 133

7.5 Our prototypicla Firefox browser extension's list. 134
7.6 Strict vs. default versions in our Firefox browser extension. 136
7.7 Strict vs. default ciphersuites in our Firefox browser extension. . . . 137
7.8 Warning vs. blocking error handling mechanisms in our Firefox browser extension. 138
7.9 System overview of strict TLS policy using header-based whitelisting. 139
7.10 System overview of strict TLS policy using DNS-based whitelisting. 142
7.11 An example of a DSTC record in the DNS for the domain "tls12". . 143
7.12 DSTC prototype with a compliant TLS server (TLS 1.2) vs. a non-compliant TLS server (TLS 1.0). 147

8.1 A high level overview of the possible reasons for servers' security misconfigurations and inconsistency. 152
8.2 An email showing a case of possible reason for inconsistency due to business reasons. 153
8.3 An overview of transparency through public logs. 158

A.1 Message sequence diagram for TLS 1.2 with (EC)DHE key exchange. 166
A.2 Message sequence diagram for TLS 1.2 with RSA key exchange. . . 167
A.3 Message sequence diagram for TLS 1.3 Hello messages with DHE key exchange and HRR. 168

List of Tables

3.1 Classifying the surveyed downgrade attacks using our taxonomy. . . 35

4.1 Summary of the scanning results. 55
4.2 Summary of the inspection results. 56
4.3 The BEFS and BESAFE mechanisms latency in ms. 71

5.1 Responding servers to our TLS clients' handshakes. 82
5.2 Differences between plain-domains and their equivalent www-domain against some TLS configurations. 84
5.3 Breakdown of some weakness indicators *weak* in plain-domains and their equivalent www-domains 86
5.4 Summary of the HTTPS redirection scan results. 87

6.1 Summary of the redirection scan responses. 99
6.2 Summary of the URLs security inconsistencies. 101
6.3 Examples of incompatible final domains and the regions they are found in. 106
6.4 The `tefal.com` case of incompatible final response URLs. 107
6.5 Examples of incompatible intermediate URLs and the regions they are found in. 108
6.6 Summary of the security headers inconsistencies. 108
6.7 Summary of the TLS security inconsistencies. 109
6.8 The percentage of diverse final URLs, diverse IPs, and redirections >0 in two cases: inconsistent HTTPS responses and secure HTTPS responses. 115

7.1 The strict vs. default TLS policies that we define in our mechanisms. 129
7.2 Test case scenarios carried from our Python DSTC-supported client to TLS servers and the effect of DSTC. 146
7.3 The mechanism's computational overhead in milliseconds. 146

A.1 TLS clients' ciphersuits list. 169

B.1 The "top1m" scheme's tables. 172

B.2 The “plain_default” and “plain_fs” tables’ fields. 172
B.3 The “www_vs_plain_v2_scan2” scheme’s tables. 174
B.4 The “tls13_plain_top1m” and “tls13_www_top1m” tables’ fields. . 174
B.5 The “transparency_5_regions_top250k” scheme’s tables. 176
B.6 The “[small_]redirection_[region]_top” tables’ fields. 176
B.7 The “malicious_domains” tables’ fields. 176
B.8 The “tls13_[region]_final_top” tables’ fields. 177

List of Abbreviations

0-RTT . . . . . Zero Round Trip Time.

[2/3/4]G . . . [2^{nd},3^{rd},4^{th}] Generation.

AE . . . . . Authenticated Encryption.

BEFS . . . . . Best Effort Forward Secrecy.

BESAFE . . . Best Effort Forward Secrecy and Authenticated Encryption.

C . . . . . Client.

CA . . . . . Certificate Authority.

CAA . . . . . Certificate Authority Authorization.

CCS . . . . . ChangeCipherSpec.

ccTLD . . . . . Country Code Top Level Domain.

CF . . . . . ClientFinished.

CH . . . . . ClientHello.

Chrome . . . . Google's Chrome Web Browser.

CKE . . . . . ClientKeyExchange.

CN . . . . . Subject Common Name.

CSP . . . . . Content Security Policy.

CT . . . . . Certificate Transparency.

CV . . . . . CertificateVerify.

DANE . . . . . DNS-based Authentication of Named Entities.

DES . . . . . Data Encryption Standard.

DHE . . . . . Ephemeral Diffie-Hellman.

$\mathbf{dhe_{pk_x}}$ . . . . . x's DHE Public Key Parameters.

dlog . . . . . Discrete Log.

DNS . . . . . Domain Name System.

DNSSEC . . . Domain Name System Security Extensions.

DoS . . . . . Denial of Service.

DS . . . . . Delegation Signer.

DSTC . . . . . DNS-based Strict TLS Configurations.

EC . . . . . Elliptic Curve.

ECDHE . . . . Elliptic Curve Ephemeral Diffie-Hellman.

(EC)DHE . . . ECDHE or DHE.

$\mathbf{ecdhe_{pk_x}}$ . . . . x's (EC)DHE Public Key Parameters.

enc . . . . . Encryption Function.

EV . . . . . Extended Validation.

FF . . . . . Finite Field.

FS . . . . . Forward Secrecy/Secure.

FS+AE . . . . FS and AE.

gTLD . . . . . Generic Top Level Domain.

hash . . . . . Hash Function.

HPKP . . . . . HTTP Public Key Pinning.

HRR . . . . . HelloRetryRequest.

HSTS . . . . . HTTP Strict Transport Security.

HTTPS . . . . Secure Hypertext Transfer Protocol.

HTTPSSR . . HTTPS Security Requirements.

IETF . . . . . Internet Engineering Task Force.

IKE . . . . . Internet Key Exchange.

IoT . . . . . Internet of Things.

IP . . . . . Internet Protocol.

IPSec . . . . . IP Security.

$\mathbf{k_x}$ . . . . . x's Session Key.

$\mathbf{kdf_k}$ . . . . . Key Derivation Function for Key k.

KSK . . . . . Key Signing Key.

MAC . . . . . Message Authentication Code.

LTE . . . . . Long Term Evolution.

mac . . . . . Message Authentication Code Function.

main-domain . TLD prefixed by a single label, without any further subdomains.

MitM . . . . . Man-in-the-Middle.

ms . . . . . Master Secret.

$\mathbf{n_x}$ . . . . . x's Nonce.

NAT . . . . . Network Address Translation.

NFS . . . . . Number Field Sieve.

NIST . . . . . National Institute of Standards and Technology.

OS . . . . . Operation System.

$\mathbf{pk_x}$ . . . . . x's Public Key.

PKCS . . . . . Public Key Cryptography Standards.

plain-domain . Domain that does not start with "www.".

pms . . . . . Pre-Master Secret.

pre-TLS 1.3 . TLS Versions Prior to TLS 1.3. Precisely: TLS 1.2, TLS 1.1., TLS 1.0, and SSL 3.0.

PRF . . . . . Pseudo Random Function.

RAM . . . . . Random Access Memory.

RDP . . . . . Remote Desktop Protocol.

RNG . . . . . Random Number Generator.

RR . . . . . Resource Record.

RRSet . . . . . Resource Record Set.

RSA . . . . . Rivest-Shamir-Adleman.

S . . . . . Server.

SAN . . . . . Subject Alternative Name.

SC . . . . . ServerCertificate.

SCSV . . . . . Signaling Cipher Suite Value.

SETUP . . . . Secretly Embedded Trapdoor with Universal Protection.

SF . . . . . ServerFinished.

SH . . . . . ServerHello.

sign . . . . . Sign Function.

$\mathbf{sk_x}$ . . . . . x's Secret/Private Key.

SKE . . . . . ServerKeyExchange.

SKI . . . . . Subject Key Identifiers.

SMTP . . . . . Simple Mail Transfer Protocol.

SNI . . . . . Server Name Indication.

SSH . . . . . Secure SHell.

SSL . . . . . Secure Socket Layer.

TCP . . . . . Transmission Control Protocol.

TLD . . . . . Top Level Domain.

TLS . . . . . Transport Layer Security.

TOFU . . . . . Trust On First Use.

Tor . . . . . The Onion Routing.

URI . . . . . Uniform Resource Identifier.

URL . . . . . Uniform Resource Locator.

VoLTE . . . . Voice Over LTE.

VP . . . . . Vantage Point.

www . . . . . World Wide Web.

www-domain . Domain that start with “www.”.

XSRF . . . . . Cross-Site Request Forgery.

XSS . . . . . Cross-Site Scripting.

ZRTP . . . . . Z Real-Time Protocol.

ZSK . . . . . Zone Signing Key.

1 Introduction

Contents

1.1 Motivation and Research Questions 1
1.2 Contributions 5
1.3 Published Work 7
1.3.1 Peer-reviewed Publications 7
1.3.2 Co-authorship 8
1.3.3 Non-peer-reviewed Pulications 8
1.3.4 Oral Presentations 9
1.3.5 Poster Presentations 9
1.4 Ethical Considerations 9
1.5 Organisation 10

1.1 Motivation and Research Questions

Negotiation, always perceived as a critical phase in politics and business protocols, is just as important in communication security protocols. In the latter, the term "negotiation" usually refers to the process of exchanging security related parameters between the communicating parties (e.g. client and server), in order to reach a mutual agreement on an optimal set of parameters that are supported by both communicating parties. These sensitive parameters may include, but are not limited to, the protocol version, and the ciphersuite that will be used for the key exchange,

encryption, and hash, to secure subsequent messages of the protocol.

Such a negotiation phase is commonly used in protocols that support multiple versions and multiple algorithms, and are widely deployed on various types of platforms that vary in their capabilities, such as personal computers vs. embedded (also known as Internet of Things (IoT)) devices. Transport Layer Security (TLS) and Secure SHell (SSH) are two notable examples of such widely used protocols.

Experience shows that the negotiation of security parameters is an attractive phase for downgrade attacks, where an active man-in-the-middle attacker interferes with the exchanged messages by the communicating parties, leading them to agree on a mode weaker than they prefer, as shown in [1]–[3], to list a few. This allows the attacker to perform subsequent attacks that would not have been possible otherwise.

It has become clear that ensuring the integrity (i.e. the messages have not been tampered with) and authenticity (i.e. the messages are originated from the intended party) of the exchanged parameters is of paramount importance in the negotiation phase, in order to prevent downgrade attacks.

While the literature has looked at negotiation integrity and authenticity in the active man-in-the-middle attacker model, we provide a new perspective to the problem. That is, we introduce ***transparency*** and ***consistency*** as needed properties in parameters negotiation in configurable protocols. This came as a result of introducing a novel attacker model through the following question:

Question 1: Are there unexplored attacker models that result in downgrade attacks?

This question led us to introduce a novel attacker model that we call the "discriminatory" attacker model. This model is applicable to semi-trusted servers running protocols that give the server the power of selecting a security parameter (i.e. running a server-dominant negotiation model), while the client has no means of verifying the server's actual parameters (i.e. justifying the server's decision if it selects a non-preferred parameter such as non-Forward Secure (non-FS) key exchange algorithm). We discuss how this power can be abused by semi-trusted servers to discriminate against their clients, mainly for a powerful third party's

advantage (e.g. government intelligence), with minimal evidence and liabilities (legal, technical, and reputation) of the server's involvement in carrying out the attack. See Section 4.5.2.3 (Item 3 in the list) for further detail about the discriminatory attacker model. This led us to the following question:

Question 2: Can a semi-trusted server discriminate against its clients without being detected?

In answering this question, we show that the discriminatory attacker model is realistic, and server discrimination against clients in terms of security can go unnoticed in today's real world protocols, such as TLS and mainstream clients (e.g. web browsers). We achieve this through an empirical case study on the FS property in the TLS protocol. We quantify servers that select non-FS key exchange algorithms, but support FS ones. Our results reveal the existence of servers that *do not select* FS, nevertheless *do support* FS. These results prove the realism of our attacker model through the following logic: if servers' behavior of selecting a weaker security parameter than what they actually support can go unnoticed by mainstream clients today, then in the same vein, servers' discrimination against clients with respect to the security guarantees that servers provide through the selected parameter, can also go unnoticed.

In the second part of the thesis, we consider assessing negotiation ***consistency***. That is, we test servers' security guarantees in response to clients' requests that differ in subtle variables, which are not expected to affect the security guarantees. We ask the following research question:

Question 3: Can two clients' requests to the same server receive inconsistent security guarantees?

To answer this question, we assess negotiation consistency through two empirical studies on the TLS and the Secure HyperText Transfer Protocol (HTTPS) protocols, respectively. In the first study, we assess servers' TLS security guarantees when requested by their plain-domains (e.g. `example.com`) as opposed to their equivalent www-domains (e.g. `www.example.com`). Our study is motivated by an initial observation from our previous empirical study on FS, combined with evidence of

lack of awareness about such inconsistencies among users, researchers, and developers alike. Our results provide evidence that plain-domains and their equivalent www-domains differ in TLS security configurations and certificates in a non-trivial number of cases. Furthermore, www-domains tend to have stronger security guarantees than their equivalent plain-domains. Further inspection of the HTTPS responses shows that over half of these domains that have stronger configurations for www-domains than those for their equivalent plain-domains tend to perform redirection from plain-domains to their equivalent www-domains. However, nearly a quarter of those redirections are insecure, as they contain plain-HTTP intermediate Uniform Resource Locators (URL)s.

In the second study on negotiation consistency, we assess servers' responses inconsistencies against requests from clients located at different geographic locations (regions). We examine a wide set of HTTPS parameters pertaining to application and transport layers. We find that security inconsistencies between regions do exist. They are strongly related to URLs and Internet Protocol (IP) addresses diversity between regions. We also find that downgraded security in some regions is related to regional blocking. Furthermore, application layer inconsistencies are higher than transport layer inconsistencies. We also provide attack scenarios that show how an attacker can benefit from HTTPS security inconsistencies, and introduce a new attack scenario which we call the "region confusion" attack.

The last part of the thesis examines the concept of ***prior knowledge*** as a means to defeat downgrade attacks through this question:

Question 4: Can we achieve a better balance between security and backward compatibility?

We examine three known methods for policy enforcement. We categorise them based on the domain subscription method: agent-based, header-based, and Domain Name System (DNS)-based. The methods themselves are known, and have been used to enforce certain security policies. However, we study their applicability to defeat TLS version and ciphersuite downgrade attacks in the HTTPS context through strict TLS configurations policy. To the best of our

knowledge, TLS versions and ciphersuites have not been considered previously using these methods. We demonstrate the value that the strict TLS policy for these two parameters can add over the existing "all or nothing" approach in TLS versions and ciphersuites negotiation.

1.2 Contributions

Having stated the research questions with an overview of their answers in Section 1.1, we now list the contributions of this thesis:

1. While Chapter 2 is meant to provide the necessary background to the reader, nevertheless, it contains original contributions. First, we introduce the notion of "negotiation power" in parameters negotiation in configurable protocols, and introduce and define three new notions of parameters negotiation models: the "server-dominant", "client-dominant", and "equilibrium" models. Second, we provide a taxonomoy of downgrade attacks in the TLS protocol and application protocols using TLS, which provides a simple and practical model for reasoning about downgrade attacks.

2. In Chapter 4, first, we introduce a novel attacker model that we call the discriminatory model. Second, we conduct an empirical analysis on FS on over 10 million server addresses from three different datasets that contain top domains, random domains, and random IPs. Our analysis employs a heuristic procedure that allows us to answer a deeper question: *Do servers that select non-FS key exchange algorithms support FS ones?* Our results provide new and useful insights to vendors, policy makers, and decision makers. While the existence of servers that select a weaker parameter than they actually support is not necessarily accidental and can be a deliberate choice for various reasons, it sheds light on the lack of negotiation transparency in widely used protocols such as TLS. Furthermore, it proves that, similar to the observed misconfigurations, discrimination in security guarantees can go unnoticed in today's mainstream clients. Interestingly, the discriminatory

attacker model is generalisable. While we examined it on the FS property in the TLS protocol, it is applicable to more subtle security parameters (e.g. key-related parameters such as randomness), as well as non-security related parameters (e.g. capabilities such as data rate). To the best of our knowledge, transparency and discrimination in configurable protocols have not been discussed in the existing literature. We are the first to observe and write about them [4]–[6]. Third, we discuss possible paths towards FS Internet traffic. In addition to the main question, there are multiple related side questions which we also address in Chapter 4. Fourth, we propose a novel client-side mechanism that we call "Best Effort Forward Secrecy" (BEFS), and an extension of it that we call "Best Effort Forward Secrecy and Authenticated Encryption" (BESAFE), which aims to guide (force) misconfigured servers to FS key exchange algorithms using a best effort approach. We implement and evaluate a proof of concept of it.

3. In Chapter 5, we conduct an empirical study to assess the consistency of servers' TLS security guarantees when requested by plain-domains as opposed to their equivalent www-domains on around 2M domains. Our results provide the first quantitative analysis of such an issue. It raises awareness about the existence of differences to both practitioners, and researchers who use domain names lists in TLS security measurement studies. Interestingly, two subsequent studies show that such inconsistencies also exist at the TCP conformance level [7], and that the human aspect (web administrators) is a possible reason for the existence of such inconsistencies that we observe [8].

4. In Chapter 6, first, we provide the first assessment of the inconsistencies of servers' HTTPS security guarantees in response to requests for the same domain from different regions: Australia, Brazil, India, the UK, and the US. That is, we quantify servers that exhibit weakness indicators in their responses to ***some but not all*** regions. Second, we study the relationship between HTTPS security inconsistencies and URLs diversity, IPs diversity, and the

presence of redirections. Third, we introduce a novel attack scenario that we call the region confusion attack. In this attack, the attacker exploits HTTPS security inconsistencies in servers' responses, and redirects a client's request, e.g. via DNS spoofing, from a region with strong HTTPS security, to another region with weak HTTPS security, to exploit the weaker region's weaknesses.

5. In Chapter 7, we study the concept of prior knowledge as a means of defeating downgrade attacks. We examine three known mechanisms to achieve this concept in the context of TLS version and ciphersuite negotiation, which has not been proposed before. We validate our proposals though prototypical implementation.

1.3 Published Work

The thesis author led several peer-reviewed conference and workshop publications [6], [9]–[13], as well as non-peer-reviewed publications, posters, and oral presentations. It is worth noting that our paper entitled "Towards Forward Secure Internet Traffic" [6], won the "**Best Paper Award**" in the 15$^{\text{th}}$ Security and Privacy in Communication Networks (SecureComm), 2019, held at Orlando, US. In what follows, we list the published papers in a reverse chronological order (most recent first), along with the relevant chapters highlighted in **Bold**. We also provide a note on authorship (related to the peer-reviewed publications), and a list for the rest of publications, posters, and presentations.

1.3.1 Peer-reviewed Publications

1. Eman Salem Alashwali, Pawel Szalachowski, and Andrew Martin, "Exploring HTTPS Security Inconsistencies: A Cross-Regional Perspective", *Computers & Security,* vol. 97, no. 101975, 2020. **Chapter 6**.

2. Eman Salem Alashwali, Pawel Szalachowski, and Andrew Martin, "Towards Forward Secure Internet Traffic", in *Proceedings of the 15th International Con-*

ference on Security and Privacy in Communication Networks (SecureComm), 2019, pp. 341–364. **Chapter 4**.

3. Eman Salem Alashwali, Pawel Szalachowski, and Andrew Martin, "Does "www." Mean Better Transport Layer Security?", in *Proceedings of the 15th International Conference on Availability, Reliability and Security (ARES)*, 2019, pp. 23:1–23:7. **Chapter 5**

4. Eman Salem Alashwali and Pawel Szalachowski, "DSTC: DNS-based Strict TLS Configurations", in *Proceedings of the 13th International Conference on Risks and Security of Internet and Systems (CRiSIS)*, 2018, pp. 93–109. **Chapter 7**.

5. Eman Salem Alashwali and Kasper Rasmussen, "What's in a Downgrade? A Taxonomy of Downgrade Attacks in the TLS Protocol and Application Protocols Using TLS", in *Proceedings of the 6th International Workshop on Applications and Techniques in Cyber Security (ATCS)*, 2018, pp. 468–487. **Chapter 2** and **Chapter 3**.

6. Eman Salem Alashwali and Kasper Rasmussen, "On the Feasibility of Fine-Grained TLS Security Configurations in Web Browsers Based on the Requested Domain Name", in *Proceedings of the 14th International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2018, pp. 213–228. **Chapter 7**.

1.3.2 Co-authorship

The aforementioned peer-reviewed publications represent the foundation of this thesis. They are proposed, executed, and authored by the thesis author, Eman Alashwali. The co-authors provided supervisory feedbacks, advise, and edits.

1.3.3 Non-peer-reviewed Pulications

1. Eman Salem Alashwali, "Negotiation Transparency in Configurable Protocols", in *CDT in Cyber Security Year Book*, 2019, pp. 30–31.

1.3.4 Oral Presentations

1. Eman Salem Alashwali, "Negotiation Transparency in Configurable Protocols", in Oxbridge Women in CS Annual Conference, Cambridge University, 2020. (Accepted for presentations, but the conference has been cancelled due to the Coronavirus (COVID-19) crisis).

2. Eman Salem Alashwali, "Negotiation Transparency in Configurable Protocols", in Oriel Talks, Oriel College, University of Oxford, 2020.

3. Eman Salem Alashwali, "Negotiation Transparency in Configurable Protocols", in CDT Research Showcase, University of Oxford, 2019.

1.3.5 Poster Presentations

1. Eman Salem Alashwali, "DSTC: DNS-based Strict TLS Configurations", in Network Traffic Measurement and Analysis (TMA) Conference Phd School, 2019.

2. Eman Salem Alashwali, "DSTC: DNS-based Strict TLS Configurations", in CDT Research Showcase, 2019.

3. Eman Salem Alashwali, "DSTC: DNS-based Strict TLS Configurations", in CDT Research Showcase, 2018.

1.4 Ethical Considerations

The empirical studies in this thesis (Chapter 4, Chapter 5, and Chapter 6) are in line with the ethical recommendations in carrying out Internet measurement studies [14]. First, we do not collect private data. All servers' data we collect are public metadata that can be viewed by mainstream clients such as web browsers. Second, we do not perform an exhaustive number of handshakes on any single server. Our clients' handshakes can by no means be classified as a Denial of Service (DoS) attack. Third, we avoid potential disturbance to other users in our institution's network if a server has blocked our scanning or inspection device's IP. To this end, we use a designated

public IPv4 address per scanning device instead of Network Address Translation (NAT). Fourth, we use informative DNS names that contain "TLS probing" to help servers' administrators identify our devices' activity in their logs. Finally, we inform the IT and security teams in our institution where the empirical studies have been conducted so they expect a high volume of outgoing connections from our experiment devices, and to expect some incoming blacklisted certificates from random servers.

1.5 Organisation

The rest of the thesis is organised as follows: in Chapter 2, we provide a background that is necessary to comprehend the rest of the thesis. In Chapter 3, we summarise related work. In Chapter 4, we present our discriminatory attacker model and our empirical case study on FS and TLS. In Chapter 5, we present our first consistency study on plain-domains and their equivalent www-domains. In Chapter 6, we present our second consistency study on HTTPS responses from a cross-regional perspective. In Chapter 7, we examine several techniques to provide clients with prior knowledge to defeat TLS version and ciphersuite downgrade attacks. Finally, in Chapter 8, we discuss possible reasons for security misconfiguration and inconsistency, summarise our research findings, discuss the generalization of our models and work, and present some open questions for future work.

2 Background

Contents

2.1 TLS 11
2.1.1 TLS, an Overview 12
2.1.2 The TLS Handshake Protocol 12
2.1.3 TLS Key Exchange Algorithms 14
2.2 Parameter Negotiation Models 16
2.2.1 Notations 16
2.2.2 The Three Negotiation Models 17
2.3 Downgrade Attacks 19
2.3.1 Downgrade Attacks, an Illustrative Example 19
2.3.2 Taxonomy of Downgrade Attacks 21
2.4 HTTP 24
2.4.1 HTTP, an Overview 25
2.4.2 HTTP Security Headers 25
2.4.3 HTTP Redirection 26
2.5 DNS 26
2.5.1 DNS, an Overview 27
2.5.2 DNSSEC 27

2.1 TLS

In this section, we provide a brief overview of the TLS protocol. A more detailed technical description of the TLS 1.2 full handshake protocol, along with the major changes introduced on TLS 1.3 that are necessary to comprehend the rest of the

thesis, are provided in Appendix A. Additionally, a further detailed description is available in the standard specifications of the last two versions of TLS: TLS 1.2 and TLS 1.3 [15], [16].

2.1.1 TLS, an Overview

TLS [15], [16] is one of the most important and widely used protocols to date. It is the main protocol used to secure Internet communication. TLS aims to provide data confidentiality, integrity, and authentication, between two communicating parties. It has been in use since 1995, and was formerly known as the Secure Socket Layer (SSL) [17]. The sixth and latest version of TLS, TLS 1.3, was standardised in August 2018 [16]. TLS consists of multiple subprotocols including the most critical handshake protocol, which will be explained next.

2.1.2 The TLS Handshake Protocol

In the handshake protocol, both communicating parties exchange keys, authenticate each other, and negotiate security related parameters, including the protocol versions and ciphersuites. The ciphersuite is an identifier that defines the cryptographic algorithms that, upon agreement between the communicating parties (client C and server S), will be used to secure subsequent messages of the protocol. In versions prior to TLS 1.3 (pre-TLS 1.3), the ciphersuite defines the key exchange, authentication, symmetric encryption, including the key length, and hash algorithms. In TLS 1.3, the key exchange is not part of the ciphersuite and is negotiated in separate extensions. Some ciphersuites provide stronger security guarantees than others. For example, FS guarantees that a compromise in the server's long-term private key does not compromise past session keys [18]. Similarly, Authenticated Encryption (AE) provides confidentiality, integrity, and authenticity, simultaneously, which provides resilience against some attacks over the MAC-then-encrypt schemes [19], [20]. Most TLS clients today, such as mainstream web browsers, offer a mixture of ciphersuites (16 ciphersuites in the Chrome browser today) that provide different levels of security guarantees such as FS, AE, both FS

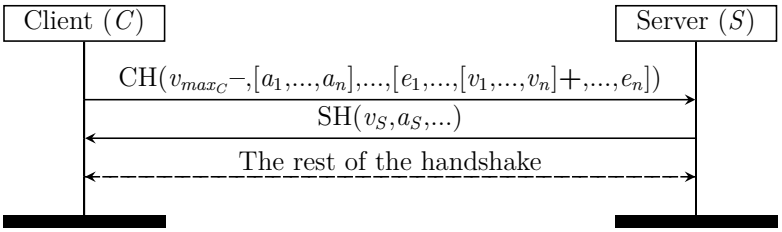


Figure 2.1: Version and ciphersuite negotiation in the TLS Hello messages. Parameters followed by "–" are deprecated in TLS 1.3 while those followed by "+" are newly introduced in TLS 1.3. The unmarked parameters are mutual to both versions.

and AE, or none of them. The same applies to TLS versions, where TLS 1.3 provides stronger security guarantees than older versions that have known design weaknesses.

As depicted in Figure 2.1, at the beginning of a new TLS handshake, both communicating parties negotiate and agree on a protocol version and a ciphersuite. The client sends a ClientHello (CH) message to the server. The CH contains several parameters including the client's supported versions, which are sent as a single value representing the maximum client's supported version (v_{max_C}) for pre-TLS 1.3 versions, and as a list of supported versions ([v_1,...,v_n]) for the TLS 1.3 version (we refer to them as the "client's offered versions"). In addition, the client sends a list of ciphersuites ($[a_1, ..., a_n]$) (we refer to them as the "client's offered ciphersuites"). Upon receiving the CH message, the server selects a single version (v_S) and ciphersuite (a_S) from the client's offer (we refer to them as the "server's selected version" and the "server's selected ciphersuite", respectively). The server then responds with a ServerHello (SH) containing v_S and a_S. If the server does not support the client's offered versions or ciphersuites, i.e. the client's offer is not in the server's supported versions or the server's supported ciphersuites, the server responds with an alert. Similarly, if the server responded with a version or a ciphersuite that is not supported by the client, the client responds with an alert. For further details about the TLS protocol, we refer the reader to [15], [16], which contain detailed specifications for the latest two versions of the protocol.

2.1.3 TLS Key Exchange Algorithms

In pre-TLS 1.3 protocols, there are two main key exchange algorithms: Rivest-Shamir-Adleman (RSA) [21], and the Ephemeral Diffie-Hellman (DHE) [22]. DHE has two variants: the Finite Field (FF) and the Elliptic Curve (EC). The term DHE denotes the FF variant, the term ECDHE denotes the EC variant, while the term (EC)DHE (with brackets) denotes either the FF or the EC variant of DHE. On the other hand, in TLS 1.3, (EC)DHE are the only allowed key exchange algorithms [16].

2.1.3.1 Non-Forward Secure Key Exchange (RSA)

The RSA key exchange algorithm [21] does not guarantee FS. As depicted in Figure 2.2, to generate a session key using RSA, the client generates a random value for the pre-master secret (pms), encrypts it with the server's long-term RSA public key (pk_S) using the (enc) function, then sends the pms in a ClientKeyExchange (CKE) message. After that, both parties derive the master secret (ms) from the pms and their nonces (n_C) and (n_S) using the Key Derivation Function (kdf_{ms}). Then, they compute the session keys (k_C) and (k_S) using another function, the (kdf_{k}). Clearly, the secrecy of the pms relies on the secrecy of the server's long-term private key (sk_S) that is associated with the server's public key pk_S since every pms is encrypted with the same server's long-term key pk_S during the key's lifetime. Therefore, if the server's long-term private key sk_S is compromised at some point in future, a passive attacker who has been collecting encrypted traffic can recover the pms, the ms, k_C and k_S, and hence decrypt past sessions' encrypted data.

2.1.3.2 Forward Secure Key Exchange (EC)DHE

The (EC)DHE [22], [23] key exchange algorithm guarantees FS. As depicted in Figure 2.3, to generate a session key using (EC)DHE, the server sends its (EC)DHE public key parameters ($ecdhe_{pk_S}$), signed with its long-term private key sk_S using the (sign) function in a ServerKeyExchange (SKE) message. The client then sends its (EC)DHE public key parameter ($ecdhe_{pk_C}$) in a CKE message. After that, both parties compute their pms, derive the ms using the kdf_{ms} function. Then, they

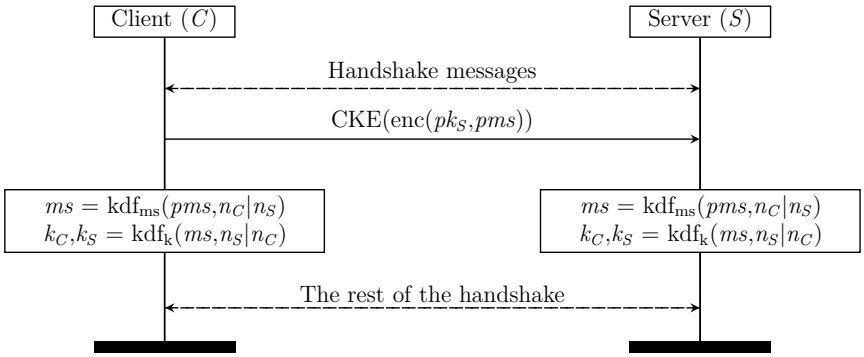


Figure 2.2: Illustration of the RSA key exchange in pre-TLS 1.3 protocols.

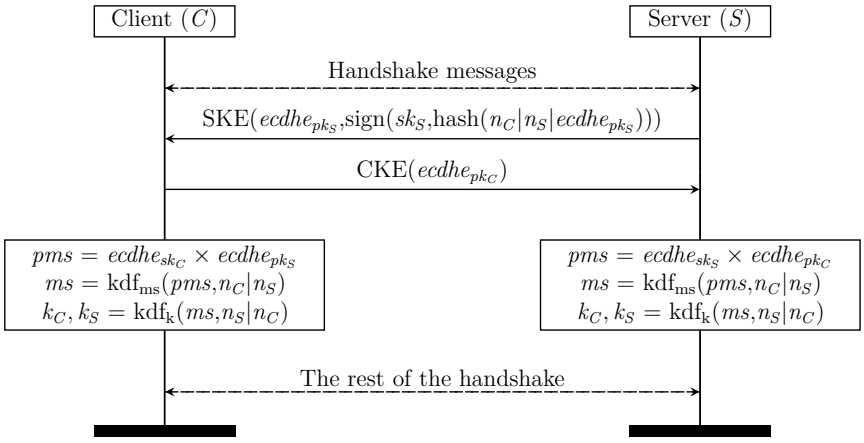


Figure 2.3: Illustration of the (EC)DHE key exchange in pre-TLS 1.3 protocols.

compute the session keys k_C and k_S using the (kdf_k) function. The (EC)DHE key is ephemeral, i.e. a fresh key is generated for each session. Unlike RSA, in (EC)DHE key exchange, the pms is not encrypted with the server's long-term key pk_S. Therefore, the ms, k_C, and k_S, do not rely on the secrecy of the server's long-term private key sk_S. Hence, if the server's long-term private key sk_S is compromised at some point in future, a passive attacker who has been collecting encrypted traffic, cannot recover the ms, k_C, and k_S of past sessions.

2.2 Parameter Negotiation Models

In this section, we provide generic negotiation models, independent of the TLS protocol. We identify three parameter negotiation models, inspired by several case studies of downgrade attacks in key exchange protocols presented in Bhargavan et al. [24] and Dowling and Stebila's work on ciphersuite and version negotiation in the TLS protocol [25]. We classify them in terms of the power of selecting security parameters, which we call the negotiation power, as follows: server-dominant, client-dominant, and client-server-equilibrium (or equilibrium for short). In what follows, we informally define and illustrate each model.

2.2.1 Notations

We assume two communicating parties: client C and server S. In a protocol session, during the negotiation phase, one party offers its parameters such as the protocol versions and ciphersuites, denoted by *params*. Upon receiving them, the recipient must select an optimal single value for each parameter, denoted by *optimal*. These optimal values are computed by running a parameters selection algorithm denoted by (select). The select algorithm takes the offered parameters from the sender as an input, in addition to the local parameters of the party running the select algorithm, such that $optimal = \text{select}(C.params, S.params)$, for C and S communicating parties. The selected parameter value *optimal* is used to set the security *mode*[1] that should be used by both parties. However, who decides the security *mode*? Both parties have a negotiation power denoted by *pow*, where $pow \in \{dominant, recessive, equilibrium\}$. We now describe each possible value of *pow* as follows:

A communicating party is said to be *dominant* with respect to parameters negotiation if it:

- Runs and controls the select algorithm (select).
- The select algorithm result (*optimal*) is sent to, and imposed on, a receiving party.

[1]The term *mode* is adopted from [24].

A communicating party is said to be *recessive* with respect to parameters negotiation if it:

- Does not run or control the select algorithm (select).
- The select algorithm result (*optimal*) is received from a dominant party.
- Does not have the means to verify the select algorithm result.

A communicating party is said to be *equilibrium* with respect to parameters negotiation if it:

- Runs and controls the select algorithm (select).
- The select algorithm result (*optimal*) is local to each party and is not sent to or received from another party.

2.2.2 The Three Negotiation Models

Utilising the aforementioned definitions of dominant, recessive, and equilibrium negotiating parties, we now illustrate the three negotiation models: server-dominant, client-dominant, and equilibrium.

1. The negotiation model is said to be server-dominant if the server is a dominant party and the client is a recessive party, as illustrated in Figure 2.4. The TLS protocol uses a server-dominant negotiation model.
2. The negotiation model is said to be client-dominant if the client is a dominant party and the server is a recessive party, as illustrated in Figure 2.5. The Z Real-Time Protocol (ZRTP) [26] uses a client-dominant model in the algorithms selection.
3. The negotiation model is said to be client-server-equilibrium, or in short, equilibrium, if the client and server are equilibrium parties, as illustrated in Figure 2.6. The SSH protocol [27] uses an equilibrium model in the algorithms selection.

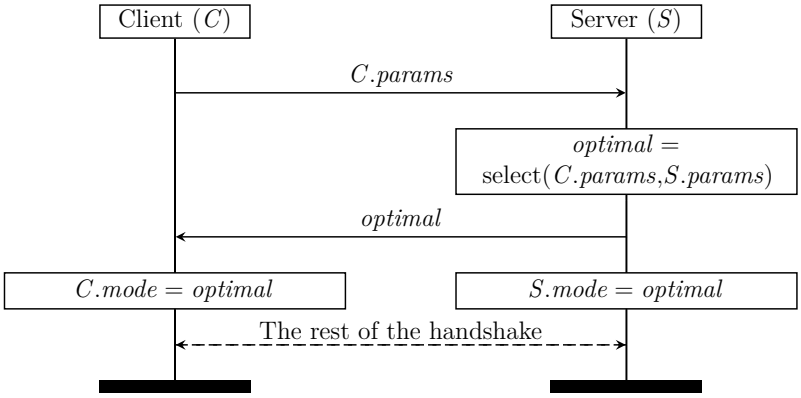


Figure 2.4: Server-dominant negotiation model.

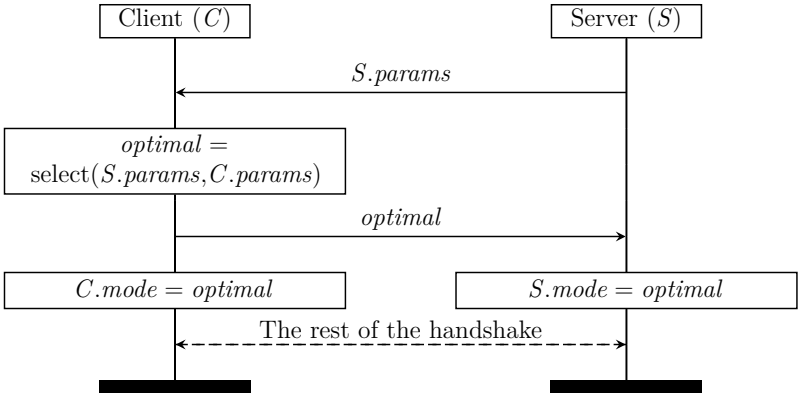


Figure 2.5: Client-dominant negotiation model.

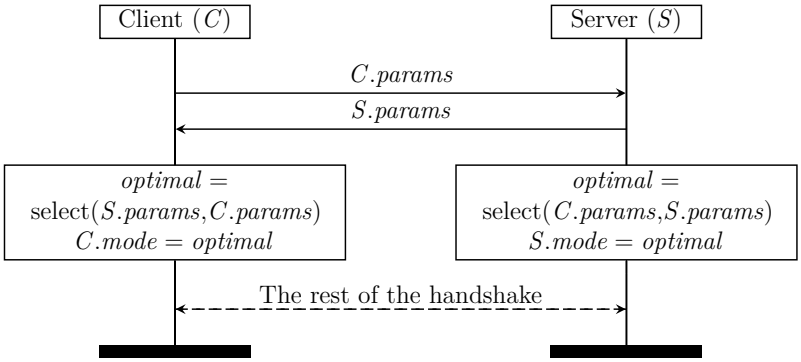


Figure 2.6: Equilibrium negotiation model.

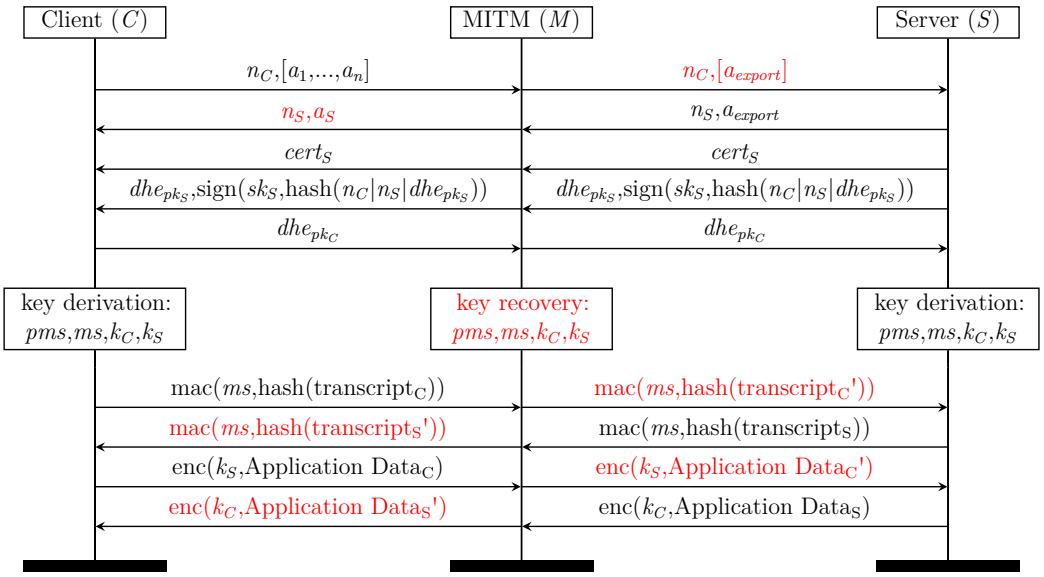


Figure 2.7: Illustrative example of downgrade attack in a simplified version of TLS 1.2.

2.3 Downgrade Attacks

In this section, we provide an illustrative example of downgrade attacks. It should be noted that downgrade attacks are general. They are not limited to the TLS protocol, and can take many forms as we will show next in the taxonomy in Section 2.3.2 and in the related work in Section 3.1. Apart from protocols such as TLS, SSH, ZRTP, another popular area of downgrade attacks is in mobile communication network protocols as shown in Shaik et al. [28], where an attacker can downgrade a 4$^{\text{th}}$ Generation/Long Term Evolution (4G/LTE) connection to non-LTE connection. Hence, the attacker can perform known attacks in its predecessors 2G or 3G.

2.3.1 Downgrade Attacks, an Illustrative Example

Figure 2.7 illustrates a downgrade attack in a simplified version of the TLS 1.2 protocol, inspired by the Logjam attack [2]. A background on the TLS 1.2 protocol that is necessary to comprehend this attack is provided in Appendix A, and in the protocol specifications in [15].

In this example, we assume a certificate-based unilateral server authentication mode using the DHE key exchange algorithm, and a Message Authentication Code (MAC) function (mac) to authenticate the exchanged handshake messages (the transcript). As depicted in Figure 2.7, the client starts the handshake by sending its nonce n_C and a list of offered ciphersuites $[a_1, ..., a_n]$ to the server. In this example, similar to the Logjam attack [2], we assume that the client's ciphersuites list contains only strong ciphersuites. However, the same attack is also applicable if the client's offered ciphersuites list contains weak (export grade) and strong ciphersuites. In both cases, similar to the TLS protocol specifications, the server must select one of the offered ciphersuites to be used in subsequent messages of the protocol. In this attack, a man-in-the-middle (MitM) attacker modifies the client's proposed ciphersuites such that they offer only weak (export grade) ciphersuites a_{export} with 512-bit DHE group. Export grade ciphersuites are weak ciphersuites with a maximum of 512-bit key for asymmetric encryption, and 40-bit key for symmetric encryption [29]. If the server supports export grade ciphersuites, e.g. 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_S and its selected ciphersuite a_{export} to the client. To avoid detection, the man-in-the-middle attacker modifies the server's choice from a_{export} to a non-export ciphersuite a_S to make it acceptable for the client which may not support export grade ciphersuites. Then, the server sends its certificate $cert_S$ which contains the server's public key signed by a trusted Certificate Authority (CA), to allow the client to authenticate the server's messages. Following the certificate, the server sends a message that contains the server's DHE public key parameters dhe_{pk_S}, and a signed hash of the nonces n_C and n_S, along with the server's DHE public key dhe_{pk_S}. The signature is used to authenticate the nonces and the server's selected key parameters. However, in pre-TLS 1.3 versions, the server's signature does not cover the server's selected ciphersuite. Therefore, even if the client supports only strong ciphersuites, if it accepts arbitrary key parameters (e.g. weak DHE groups), it will not distinguish whether the selected ciphersuite is

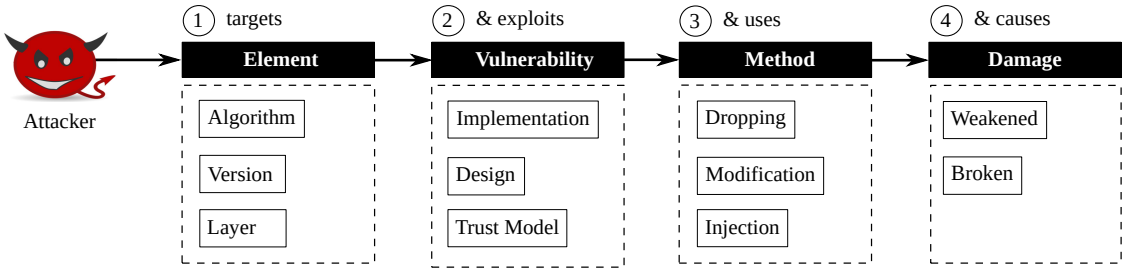


Figure 2.8: A taxonomy of downgrade attacks in the TLS protocol and application protocols using TLS.

an export grade or a strong one, 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 DHE public key parameters dhe_{pk_C}. 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_C and k_S, respectively. The exchanged weak DHE public key parameters enable a man-in-the-middle attacker to recover secret values from the weak DHE public keys, e.g. recover the private exponent from one or both parties' DHE public keys using the Number Field Sieve (NFS) discrete log *dlog* (since we assume DHE key exchange algorithm). Consequently, the attacker will be able to compute the *pms*, *ms*, k_C, and k_S 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 attacker has the session keys, he/she can decrypt the exchanged messages between the client and server as illustrated in Figure 2.7.

2.3.2 Taxonomy of Downgrade Attacks

Based on an analysis of 15 notable downgrade attacks (summarised in Section 3.1), we distill four vectors that characterise downgrade attacks, namely: element, vulnerability, method, and damage. These vectors represent the taxonomy's main categories. We assume a man-in-the-middle attacker model, where the attacker has full control over the communication channel and can drop, modify, inject, or replay messages. In what follows, we define the notions of the categories and subcategories that we use in our taxonomy. Figure 2.8 summarises the taxonomy.

1. Element: this refers to the protocol element that is being negotiated between the communicating parties. The element's value is intrinsic in defining the security mode, i.e. the security level for the rest of the communication. The element is targeted by attackers 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 subcategories as follows:

   (a) Algorithm: refers to the cryptographic algorithms, e.g. key exchange, encryption, hash, signature, and their parameters such as the block cipher modes of operation and key lengths, that are being negotiated to be used in subsequent messages of the protocol. Generally, as in the TLS protocol, the main algorithms are represented by the ciphersuite. However, they can also be represented by other parameters that are not part of the ciphersuite such as the extensions.

   (b) 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: refers to the whole TLS layer, which is negotiated, and optionally added in some legacy protocols. In such protocols such as the Simple Mail Transfer Protocol (SMTP) [30], TLS encapsulation is negotiated through specific upgrade messages, such as the "STARTTLS" [31] in SMTP, in order to upgrade the protocol from an insecure (plaintext and unauthenticated) to a secure (encrypted and/or authenticated) mode over TLS.

2. Vulnerability: similar to any attack performed by an external man-in-the-middle attacker, downgrade attacks require a vulnerability to be exploited. We categorise the vulnerability into three subcategories as follows:

(a) Implementation: refers to a faulty protocol implementation. The existence of implementation vulnerabilities can be due to various reasons, e.g. a programmer's fault, a state machine bug, or a malware that corrupted the code.

(b) Design: 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: 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: this refers to the method used by the attacker to perform the downgrade. We categorise the method into three subcategories as follows:

(a) Modification: 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 or authentication checks for the handshake transcript, the downgrade attack can be trivially performed. Otherwise, the attacker needs to find ways to circumvent the checks, such as breaking the master secret or creating colliding hashes for the transcript.

(b) Dropping: the attacker drops one or more protocol messages, possibly more than once. Figure 2.9 illustrates the dropping method in the POODLE attack [32].

(c) Injection: 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.

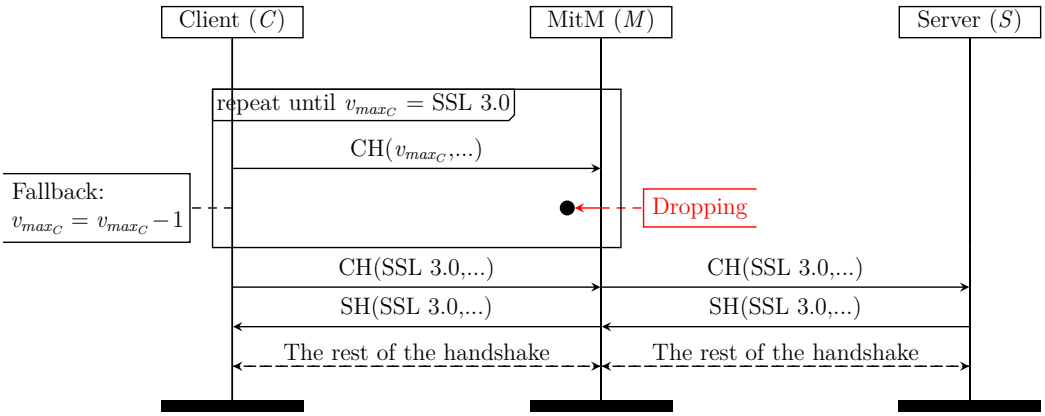


Figure 2.9: Version downgrade attack using the dropping method as in the POODLE attack [32].

4. Damage: this refers to the resulting damage after a successful downgrade attack. We categorise the damage into two subcategories as follows:

   (a) 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 broken security damage, weakened security does not result in an immediate breakage of any of the main security guarantees. Instead, weakened security refers to attacks that result in making the communicating parties choose an unrecommended or less preferred mode, which is not broken yet, but can be broken in future, such as non-FS key exchange.

2.4 HTTP

In this section, we provide a brief overview of the HTTP protocol, HTTP security headers, and HTTP redirection. Further technical details can be found in their specifications in [33]–[39].

2.4.1 HTTP, an Overview

HTTP is a stateless application layer protocol for distributed, and collaborative, hypertext information systems [33]. The HTTP protocol on its own is insecure. Throughout the text, we denote it by plain-HTTP to avoid ambiguity with HTTPS. Indeed, websites running plain-HTTP are prone to eavesdropping, manipulation, and impersonation attacks. To secure HTTP communication, HTTP is encapsulated in TLS (HTTPS). TLS includes the handshake protocol, in which the communicating parties authenticate each other, negotiate security related parameters such as the protocol versions and ciphersuites, and exchange keys. A background on the TLS and the handshake protocols are provided in Section 2.1.

2.4.2 HTTP Security Headers

HTTP headers allow clients and servers to exchange additional information in the HTTP requests and responses. HTTP headers are sent as a key/value pairs, where the key represents the header's name, and the value represents the header's value. There are many types of HTTP headers, including those for signalling and specifying security policies. We refer to the latter by "security headers". In servers' security headers, the server sends one or more security headers to instruct the client to enforce a security policy. Some of the most important security headers that are related to our work are:

1. HTTP Strict Transport Security (HSTS) [39]: to inform the client to always enforce HTTPS, to provide protection against TLS layer downgrade attacks.

2. Content Security Policy (CSP) [34]–[36]: to restrict the sources a client can load content from, to protect against code injection attacks such as Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) attacks, which aim to steal information and inject malware to the client or server.

3. The Secure attribute in the Set-Cookie header [37], [38]: to force sending cookies over a secure HTTPS channel, to protect against leaking users' private data, and identity theft attacks.

2.4.3 HTTP Redirection

HTTP redirection, also known as URL redirection [33], is a technique that allows a resource (e.g. a website) to be reachable from multiple URL addresses. URL redirection also includes domain redirection, where a website is reachable from one or more domain names. For example, when a website can be reached through either its plain-domain (e.g. `example.com`) or its equivalent www-domain (e.g. `www.example.com`), it is because requests to the plain-domain are redirected to its equivalent www-domain, or vice versa. Similarly, domain redirection is used when requests to a domain, e.g. with a generic TLD (gTLD) (`example.com`) are redirected to another domain, e.g. a regional domain with a country code Top Level Domain (ccTLD) (e.g. `example.co.uk`), or to a regional subdomain (e.g. `uk.example.com`).

There are multiple methods to perform HTTP redirection. However, one of the most widely used methods is redirection through the server redirection response, which is the method we consider in our study (mainly in Chapters 5 and 6). The redirection response is identified by the response status code (300-308). In this method, upon receiving the server's redirection response, the client uses the new URL provided in the Location header, and sends a new request to the new URL. The new URL that resulted from the redirection response can also perform a redirection, forming what is known as "redirection chain". The recommended maximum number of redirections that a client can accept is five redirections [33]. However, adhering to the limit depends on the client's vendor.

2.5 DNS

In this section, we provide the necessary background about DNS and DNS Security Extensions (DNSSEC). Further technical details can be found in their specifications [40], [41]

2.5.1 DNS, an Overview

DNS [40], also known as the Internet's phone book, is a hierarchical system that stores and manages domains' data. These data are stored in dedicated Resource Records (RR)s in the domain's zone file. Each DNS holds RRs for the name space that it is authorised for. There are various types of RRs. For example, the A record is used to map a domain name to its IPv4 address, the AAAA record is used for the IPv6 address, while the TXT record is used to hold arbitrary human readable text. The primary function of DNS is resolving domain names to their corresponding IP addresses such that clients can request resources using domain names. This resolution function is important as it eliminates the need for users to memorise long numerical (IPv4) or alphanumerical (IPv6) addresses. DNS resolution precedes communication between hosts. When a user requests a domain name, for example `www.example.com`, the client contacts the DNS infrastructure that resolves this name into its IP address recursively. That is, the client first contacts one of the DNS root servers, which refers the client to the `com`'s server address, which in turn refers the client to the `example.com`'s authoritative server address, which finally returns the address of the targeted domain. To enhance the efficiency of this process, the DNS infrastructure employs different caching strategies. Additionally, a single server (website) can have multiple IPs in multiple A RRs. The IPs can point to multiple physical servers or a single physical server. When the same domain has multiple IPs, the returned IP in the DNS response to the client's request can be managed through load balancing techniques such as the round-robin DNS [42]. Such load balancing techniques may cause security implications if the two IPs (servers) have inconsistent security configurations, where one provides stronger security guarantees than the other, as our empirical studies in Chapters 4 to 6 show this possibility.

2.5.2 DNSSEC

DNS on its own is not secure. DNS responses can be manipulated by man-in-the-middle attackers. To overcome this limitation, DNSSEC [41], has been introduced. DNSSEC is an extension of DNS which aims to provide integrity and

origin authentication for DNS records through digital signatures. In DNSSEC, each DNS zone has a Zone Signing Key (ZSK) pair. The private ZSK is used to sign the DNS RRs, which are grouped into Resource Record Set (RRSet), where there is one RRSet for all the records that share the same RR type (e.g. the A type). The RRSets' signatures are published in the DNS via dedicated RRSIG RRs. Every zone publishes the public ZSK in a special DNSKEY RR. The public ZSK is used by recursive resolvers to verify the zone's signed RRs. To authenticate the public ZSK, the DNSKEY record is signed with the private key of another key pair known as the Key Signing Key (KSK), creating an RRSIG record for the DNSKEY. Resolvers validate the public ZSK using the public KSK. Similar to the ZSK, the public KSK is also published in another DNSKEY RR. Both the public ZSK and the public KSK are published in the DNSKEY RR, forming a DNSKEY RRSet, which is signed by the private KSK. To authenticate the public KSK, for each child zone, the hash of the DNSKEY is published in its parent in a Delegation Signer (DS) RR, which is signed by the parent's private ZSK, forming a trust chain. To validate a zone's public KSK, when the client is referred to a child zone, it is provided with the DS record. Then, the client must compare the hash of the child's DNSKEY with the hash in the DS provided by the parent zone. That is, to validate the authenticity of a DNS response, clients have to follow the trust chain until the root.

3
Related Work

Contents

3.1 Downgrade Attacks, a Survey 30
3.1.1 Algorithm Downgrade 30
3.1.2 Version Downgrade 33
3.1.3 Layer Downgrade 34
3.2 TLS Security Measurements 35
3.2.1 Cryptographic Aspect 36
3.2.2 Certificate Aspect 38
3.2.3 Security Enhancements Aspect 39
3.3 Non-FS Key Compromise 39
3.3.1 Caused by Design 39
3.3.2 Caused by Implementation 40
3.4 Consistency Measurements 40
3.4.1 Client Type Aspect 40
3.4.2 Regional Aspect 41
3.4.3 Vantage Point (VP) Aspect 42
3.5 HTTPS Enhancement Mechanisms 42
3.5.1 DNS-based 43
3.5.2 Header-based 43
3.5.3 Certificate-based 44
3.5.4 Agent-based 44
3.6 Browser Warnings 44
3.6.1 Effectiveness 45
3.6.2 Caveats 45

3.1 Downgrade Attacks, a Survey

Downgrade attacks have existed since the very early versions of TLS: SSL 2.0 [17] and SSL 3.0 [43]. In this section, we provide a survey of notable TLS downgrade attacks, classified by the targeted element: algorithm, version, and layer. We highlight the attack names in **Bold**. All these attacks assume a man-in-the-middle attacker model. We assume the reader's familiarity with the TLS protocol technical details. The unfamiliar reader is advised to read the TLS background in Section 2.1, Appendix A, or the TLS specifications in [15], [16], which provide the necessary background to comprehend the rest of the section. Table 3.1 provides a classification of these attacks based on our taxonomy that we introduced in Section 2.3.2.

3.1.1 Algorithm Downgrade

In [44], Wagner and Schneier showed that 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, 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).

Wagner and Schneier also reported another design flaw in SSL 3.0 which allows a theoretical attack named the **"key exchange rollback"** attack [44], which is a result of the lack of early authentication of the server's selected ciphersuite (which includes the key exchange algorithm) before the Finished MACs. 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 one another, and to decrypt application data.

In [45], Mavrogiannopoulos et al. presented an attack which we call the **"DHE key exchange rollback"**. It can be considered a variant of the aforementioned **"key exchange rollback"** in [44]. 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 (tampered) client's 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 [44], these mismatched views about the key exchange algorithm result in breakable keys, which allow the attacker to recover the pre-master and master secrets. Consequently, the attacker is able to forge the Finished MACs to hide the modifications in the Hello messages, impersonate each party to one another, and decrypt application data.

In Adrian et al. [2], the **Logjam** attack is presented. It uses a method similar to the one we explained in the illustrative example in the background in Section 2.3.1. The Logjam attack is applicable to the DHE key exchange algorithm. It works by modifying the Hello messages to misguide the server into selecting an export grade DHE ciphersuite, which results in weak DHE keys. As stated earlier, pre-TLS 1.3 versions do 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 application data.

Beurdouche et al. [1] presented a similar attack that is called the Factoring RSA Export Keys **(FREAK)** attack. It 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[1]. 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 application data.

In [46], Aviram et al. presented a variant of the Decrypting RSA using Obsolete and Weakened eNcryption (**DROWN**) attack (the "special DROWN") that exploits an OpenSSL server implementation bug [47]. The attack enables a man-in-the-middle attacker 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 makes use of a known flaw that can be exploited if the server's RSA key is shared with an SSL 2.0 server using an attack called the Bleichenbacher attack [48]. This attack enables the attacker to recover the plaintext of an RSA encryption (i.e. the pre-master secret) by using the SSL 2.0 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 application data.

Beurdouche et al. [1] presented another case of algorithm downgrade attacks called the **"Forward Secrecy rollback"** attack [1]. In this attack, the attacker exploits an implementation vulnerability to make the client fall back from Forward Secrecy (FS)[2] mode to non-FS mode by dropping the ServerKeyExchange message. This in turn leads the client to fall back to a non-FS key exchange, and use the server's static key parameters in the certificate, which do not provide FS. However,

[1]Note that the ServerKeyExchange message must not be sent when the key exchange algorithm is non-export grade (i.e. strong) RSA as per the TLS specifications in [15]. However, the ServerKeyExchange is sent in export grade RSA or in (EC)DHE key exchange.

[2]FS is a property that guarantees that a compromise in the long-term key does not compromise past session keys [18].

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.

Finally, Bhargavan et al. [24] reported an algorithm downgrade attack in TLS 1.3 draft-10 [49], which we call the "**HelloRetry downgrade**" attack. It 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 was possible because the transcript hash restarts with every HelloRetryRequest. However, consequent TLS 1.3 drafts mitigated this attack by continuing the hashes over retries [24].

3.1.2 Version Downgrade

In [44], Wagner and Schneier reported that 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. 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 attack undetected.

Version downgrade attacks are not exclusive to SSL 3.0. The Padding Oracle On Downgraded Legacy Encryption **(POODLE)** attack by Moller et al. [32], 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. In the POODLE attack, a man-in-the-middle attacker abuses this mechanism by dropping the ClientHello which leads the client to fall back to SSL 3.0 (see Figure 2.9 for illustration). This in turn brings the specific cryptographic flaw that is associated with the CBC padding in all block ciphers in SSL 3.0. This allows the attacker to decrypt some of the SSL session's data, such as the cookies that may contain sensitive data.

In Bhargavan et al. [50], a downgrade attack in TLS 1.0 and TLS 1.1 is illustrated. The attack comes under a family of attacks called 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 [51], [52]. 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.

Downgrade attacks continued to appear until draft-10 of the latest version of TLS (TLS 1.3 [49]), where Bhargavan et al. reported three possible downgrade attacks in TLS 1.3 draft-10 [50]. 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/her to break the master secret before the Finished MACs as in Adrian et al. [2] and Beurdouche et al.[1], hence circumventing downgrade detection.

The second attack in TLS 1.3 draft-10, which we call the "**downgrade dance version rollback**" attack, is reported by Bhargavan et al. [50]. It employs a method similar to the one employed in the **POODLE** attack [32], 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 circumventing detection due to weak downgrade resilience in TLS 1.2 and lower versions.

3.1.3 Layer Downgrade

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 by Durumeric et al. [3]. They found evidence of corrupted STARTTLS commands which downgrade **SMTPS to SMTP** in more than 41,000 mail servers.

Table 3.1: 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.

No.	Attack	Element			Vuln.			Method			Damage	
		Algorithm	Version	Layer	Implementation	Design	Trust-model	Dropping	Modification	Injection	Weakened	Broken
01	SSL 2.0 Ciphersuite rollback [44]*	✓				✓			✓			✓
02	SSL 3.0 Version rollback [44]*		✓			✓			✓			✓
03	SSL 3.0 key exchange rollback [44]*	✓				✓			✓			✓
04	DHE key exchange rollback [45]	✓				✓			✓			✓
05	TLS 1.0-1.1 SLOTH [50]*		✓			✓			✓			✓
06	POODLE version downgrade [32]		✓		✓			✓				✓
07	FREAK [1]	✓			✓				✓			✓
08	DROWN [46]	✓					✓		✓			✓
09	Forward Secrecy rollback [1]	✓			✓			✓			✓	
10	Logjam [2]	✓				✓			✓			✓
11	SMTPS to SMTP [3]*			✓		✓			✓			✓
12	Proxied HTTPS [53]*			✓			✓			✓		✓
13	TLS 1.3 Version rollback [24]*		✓			✓			✓			✓
14	TLS 1.3 Downgrade-dance version fallback [24]*		✓		✓			✓				✓
15	TLS 1.3 HelloRetry downgrade [24]*	✓				✓				✓	✓	

Similarly, downgraded TLS as a result of **proxied**[3] **HTTPS** connections have been shown to be prevalent. In another study by Durumeric et al. [53], empirical data showed that 10-40% of the proxied TLS connections advertise known broken cryptographic choices (e.g. versions and ciphersuites) [53]. While the empirical results in [53] showed an evidence of downgraded TLS version and algorithm due to proxied HTTPS, a man-in-the-middle attacker can send the client's data to the server in cleartext. Therefore, based on the worst case assumption, we classify the targeted element as a layer.

Table 3.1 summarises the attacks described above and their classification using our taxonomy that we introduced in Section 2.3.2.

3.2 TLS Security Measurements

There are plenty of measurement studies that evaluate the security of TLS handshakes and HTTPS connections from several aspects: cryptographic, certificates,

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

and security enhancement mechanisms. In terms of data gathering techniques for measurement studies, they are usually done through Internet "scans". The scans can be either passive or active. In passive scans, the data are obtained from network traffic, e.g. generated by users while using an organisation's network. On the other hand, in active scans, the data are generated by the researcher through actively initiating connections with a list of servers' addresses, using scanning tools that mimic clients. Passive scans provide a more realistic view of the TLS deployment than active scans. On the other hand, active scans give the researcher the ability to customise the client's configurations, which is useful to probe the servers' supported parameters that are not selected by default, and cannot be measured using passive scans. In terms of servers' addresses format, they can be either domain names or IP addresses. It should be noted that using IP addresses to actively scan certificates can miss a considerable percentage of valid certificates. This is because IP-based scans do not include the Server Name Indication (SNI) [54] in the requests. For this reason, domain-based scans for certificate data is a better choice. Additionally, IP-based scans include a considerably higher percentage of network devices such as embedded (or IoT) devices. In what follows, we summarise the most notable studies in this realm. We classify them into three main categories based on their main focus: cryptographic, certificate, and security enhancements aspects.

3.2.1 Cryptographic Aspect

Lee et al. [55], published one of the earliest measurement studies that evaluated TLS cryptographic strength in real world servers. They actively scanned around 19,000 web server addresses, obtained from multiple top domain lists. Their evaluation included cryptographic parameters such as the key exchange and symmetric encryption algorithms. In [56], Heninger et al. conducted an active scan for the IPv4 address space to assess the strength of certificates' public keys. Their results showed that factorable RSA keys are widespread. They could factor[4] 0.5% of the scanned TLS servers. However, the factorable keys phenomenon is mainly observed in network

[4]i.e. obtain the RSA private key.

devices, due to low entropy in the Random Number Generators (RNG)s during prime generation. Consequently, Alashwali observed a similar behavior in a replicated experiment [57]. In [58], Huan et al. analysed FS deployment on 473,802 Alexa's top domains. They found that the majority (82.9%) of FS deployments use weak key parameters. Additionally, through a performance evaluation experiment, they refuted the performance argument against FS algorithms, where they showed that ECDHE key exchange algorithm can outperform RSA algorithm. In [59], Chang et al. evaluated the security of HTTPS redirections using an active scan for the Alexa top 1 million domains. They found that the majority of HTTPS redirections (83.3%) in TLS servers are insecure. Samarasinghe and Mannan [60], analysed data for 299,858 TLS networked devices and 598,888 Alexa's top domains, through the Censys engine [61], which performs active scans to collect data. They found that weak cryptographic parameters in network devices are widespread. Additionally, a comparison between the cryptographic parameters in network devices against those in top domains, showed that network devices tend to have weaker cryptographic parameters. Around two years later, the same authors of [60], conducted a follow-up study for a larger dataset [62]. In it, they analysed 6,319,951 network devices and 735,638 Alexa's top domains. The study found that, while using TLS among these devices increased, there is no significant progress in terms of cryptographic strength.

More recently, Kotzias et al. [63], conducted a longitudinal analysis to examine the impact of high profile attacks such as Heartbleed, FREAK, and Logjam, on TLS deployment. They analysed data from both passive scans for 319.3 billion connections since 2012, and active scans for the entire IPv4 space since 2015. They correlated changes in the utilised cryptographic algorithms to high profile attacks. They concluded that while clients are fast in adopting new ciphersuites, they are slow in deprecating old ones. Calzavara et al. [64], actively scanned Alexa's top 10,000 domains, and conducted vulnerability analysis. They assessed the vulnerabilities impact on the websites' integrity, authentication credentials, and web tracking. They found that the security of 10% of the homepages are "compromisable". Furthermore, 75% of the compromisable homepages are due to

external or related domains. Frost et al. [65], conducted an active scan on 31 million IPv4 addresses to measure the Data Encryption Standard (DES) cipher support in today's servers. DES is a broken symmetric encryption algorithm since 1997 [66], and has been deprecated by the National Institute of Standards and Technology (NIST) since 2005 [67]. They found over 40% of servers still support at least one variant of DES ciphers. In [68], Holz et al. conducted a large-scale scan to measure TLS 1.3 adoption since its standardisation in 2018. They actively scanned 275 million domains, gathered from multiple sources including three top domains lists: Alexa, Majestic, and Cisco Umbrella. In addition, they analysed large datasets from passive scans. They found that TLS 1.3 adoption is widespread. However, it is mainly dominated by several large players such as CloudFlare and Google.

3.2.2 Certificate Aspect

In [69], Holz et al. conducted active scans for Alexa's top 1 million domains in addition to passive scans from a large research network. The study raised concerns about the security of certificates' ecosystems and identified the main causes of weak certificates. Durumeric et al. [70], conducted large-scale active scans to analyse the TLS certificates' ecosystem for the entire IPv4. They uncovered poor security practices and configurations that may lead to errors or vulnerabilities. In [71], Chung et al. analysed invalid certificates gathered from multiple active scans conducted by the University of Michigan and Rapid7 over a 3 year time span, covering 192 million IPv4 addresses. They found that, on average, 65% of the available certificates are invalid, and are originated from a small set of device types. In [72], Stark et al. evaluated the deployment of Certificate Transparency (CT), a recent standard initiated by Google that aims to monitor certificates to detect misissued certificates [73]. They used passive and active scans from Google Chrome metric and telemetry, in addition to scanning several domains' lists including the Alexa top 10,000 domains list. Overall, they found that most HTTPS traffic is CT-compliant with a low failure rate.

3.2.3 Security Enhancements Aspect

In [74], Kranch and Bonneau analysed the HSTS and PKP deployment using active scans for the preloaded domains in Firefox and Chrome browsers, in addition to Alexa's top one million domains. They found a large number of misconfiguration errors, which undermine the desired security of these mechanisms. They shed light on the importance of the mechanisms' simplicity and clear defaults. In [75], Amann et al. analysed some HTTPS enhancements mechanisms such as: CT, HSTS, HTTP Public Key Pinning (HPKP) security headers, Certificate Authority Authorization (CAA) DNS record, and the DNS-based Authentication of Named Entities (DANE) TLSA[5] DNS record, and the Signaling Cipher Suite Value (SCSV) fallback signalling mechanism. They actively scanned around 193 million domains, in addition to data from passive scans from university networks in three continents. They concluded that only the CT and SCSV mechanisms are successfully widely deployed.

3.3 Non-FS Key Compromise

Since we place emphasis on FS as one of the most important properties we examine in our experiments in Chapters 4 to 6, we now give a brief background on some notable studies that prove that non-FS key compromise is realistic. The compromise can be due to multiple reasons. We present RSA key compromise due to design and implementation reasons. However, non-FS key compromise through social engineering is also possible, but we are not aware of existing studies in this aspect.

3.3.1 Caused by Design

Kleinjung et al. [77] and Cavallar et al. [78], showed that RSA keys can be compromised due to advances in computing power, where they could factor 786-bit and 512-bit RSA keys respectively, using powerful machines. In [48], Bleichenbacher showed that in RSA with Public-Key Cryptography Standards#1 (PKCS#1), an attacker can perform an RSA private key operation if he/she has access to an adaptive chosen ciphertext oracle.

[5] ""TLSA" does not stand for anything; it is just the name of the RRtype." [76].

3.3.2 Caused by Implementation

As stated earlier, Heninger et al. showed that RSA keys can be compromised due to low entropy during prime generation [56]. Additionally, the Heartlbleed bug in the OpenSSL TLS library showed that implementation bugs can cause RSA key compromise [79].

3.4 Consistency Measurements

In recent years, several studies examined various aspects of servers' responses inconsistencies among clients that differ in subtle variables, such as the client type (e.g. desktop vs. mobile), the Vantage Point (VP) (e.g. research university, residential network, The Onion Routing (Tor) gateway proxy, and cloud datacenter), or the geographic location. In this section, we summarise notable studies in this area.

3.4.1 Client Type Aspect

Mendoza et al. [80], analysed the inconsistencies of security headers in servers' responses to mobile vs. desktop users. They conducted active scans for Alexa's top 70,000 domains. Overall, they found 2000 domains with inconsistencies in one or more of the examined security headers, even in very popular domains such as Netflix and Google. Khattak et al. [81], measured the inconsistencies of servers' responses to anonymous (using Tor network) vs. normal users. They conducted active scans, using multiple datasets including the entire IPv4 space, and Alexa's top 1000 domains. They found 1.3 million addresses of the IPv4 address space, and around 3.67% of Alexa's top 1000 domains either block or provide a degraded service to Tor users. Van Goethem et al. [82], actively scanned 10,222 desktop vs. mobile websites for several security defense mechanisms such as the CSP and the Secure attribute in Cookies. They concluded that mobile websites have slightly less security mechanisms than their desktop counterpart, despite the fact that mobile websites were developed nearly seven years after the desktop websites, which indicates that security mechanisms are applied retroactively.

3.4.2 Regional Aspect

Afroz et al. [83], studied server-side blocking of regions. They conducted active scans from eight countries: Botswana, Bulgaria, Kenya, Pakistan, South Africa, UK, Ukraine, and USA, using different network types: institutional, home, VPNs, and cloud, on 7081 domains obtained from Alexa's global and regional top domains. They confirmed the existence of servers that block users based on region. Fruchter et al. [84], analysed cookies and HTTP requests from four countries that differ in privacy regulations: Germany, US, Japan, and Australia. They actively scanned Alexa's top 250 domains, using Amazon Web Services. They found that tracking behaviour differs between countries. In the same vein, Samarasinghe and Mannan [85], examined third party scripts and cookies using active scans for Alexa's 2050 global and regional top domains, from machines located in 56 countries, using the OpenWPM privacy measurement framework. They concluded that the prevalence of web tracking varies between regions. In [86], Eijk et al. studied a similar aspect. They actively scanned a total of 1500 domains, obtained from the Majestic regional top domains list, from 18 locations: Canada, US, Switzerland, and 15 European countries. They found that the inconsistencies in cookie notices are related to Top Level Domains (TLD)s and not users' location. However, Eijk et al. also pointed out that this conclusion does not hold for the `com` domains, where the users' location relates to the inconsistencies of cookie notices. This is because gTLDs such as `com` usually perform redirection based on the location. In [75], Amann et al. conducted TLS scans from various locations. They noticed a small fraction of inconsistencies in the HSTS and HPKP headers in the active scans, but they did not analyse them. However, Amann et al. did not follow HTTP redirections in their scans, where they connect to the "base domain" in all regions. However, they noted that "closer investigation of inconsistent domains may reveal interesting insight", which we try to explore in Chapter 6.

3.4.3 Vantage Point (VP) Aspect

Jueckstock et al. [87], examined the consistency of servers' responses to requests for Alexa's top 5000 domains, initiated from different VP: cloud data centre, research university, residential network, and Tor gateway proxy. They reported poor performance from Tor, and slight differences between others. However, they recommend university VPs over cloud provider VPs in measurement studies, as the university VPs generalises "slightly better" to actual residential browsing experience.

3.5 HTTPS Enhancement Mechanisms

There are several HTTPS enhancement mechanisms that try to assist users and servers' owners to opt in stricter TLS security policies than the default policies. The main motivation behind this line of work is that most mainstream HTTPS clients and servers sacrifice some level of security for backward compatibility. For example, most mainstream web browsers allow users to bypass invalid (e.g. self-signed) certificates, which can be due to a man-in-the-middle attacker who is presenting a forged certificate. We also note that browsers silently (without a warning) fall back to legacy TLS versions or weaker ciphersuites (e.g. non-FS or non-AE), which can be due to a man-in-the-middle attacker performing a downgrade attack, to exploit the legacy version's or the weaker ciphersuite's flaws. In this section, we summarise notable HTTPS enhancement mechanisms that are related to our work. It should be noted that these mechanisms have some requirements to achieve their goals: first, all DNS-based security enhancement mechanisms require DNSSEC, to provide authentication to DNS RRs. Second, header-based mechanisms require Trust On First Use (TOFU), which means a secure first connection when the header is sent by the server. Finally, agent-based mechanisms that require the user's input, assume security-aware users.

3.5.1 DNS-based

One of the earliest proposals in this realm is Schechter's HTTP Security Requirements (HTTPSSR) RR in DNS [88], which aims to prevent TLS layer downgrade (also known as stripping) attacks. It allows domain owners to assert their support for the TLS protocol such that clients refuse to accept plain-HTTP connections from these domains that asserted TLS support. Hallam-Baker [89], proposed the CAA DNS RR, which aims to prevent certificate misissuance. It allows domain owners to whitelist specific CAs that are allowed to issue certificates to their domains. Then, the list must be checked by CAs before issuing a certificate for the listing domain. Dukhovni and Hardaker [90], proposed DANE, which aims to prevent impersonation attacks that use misissued or forged certificates. DANE allows domain owners to bind a domain's certificate or public key through the TLSA DNS RR. During the TLS handshake, the client checks the data of the TLSA RR against the presented certificate's data. Varshney and Szalachowski [91], proposed a general DNS-based metapolicy framework, which represents a general framework to express security policies through the TXT DNS RR. It also provides multi-level failure handling (hard, soft, ignore), to be specified by domain owners. Our proposal for DNS-based strict TLS configurations in Section 7.4.3 can be considered a special case of such a framework at the TLS version and ciphersuite levels.

3.5.2 Header-based

There are several header-based security enhancement mechanisms. They enable servers to send HTTP security headers, which instruct clients to maintain a state about the sending server. Clients can enforce security policies in subsequent connections to the sending server. In [92], Jackson and Barth introduced a header-based policy called "ForceHTTPS". Servers can opt into this policy by advertising a special HTTP response header. The header sets a cookie that instructs the client to enforce strict TLS policy when connecting to the advertising server. Namely, the client converts any plain-HTTP URL to HTTPS, and performs stricter certificate validation than the default one. The mechanism blocks connections to any opted

in website if the policy is violated. In subsequent years, ForceHTTPS has been refined and developed into a standard known as HSTS [39]. In [36], Stamm et al. proposed the CSP, which aims to prevent XSS and XSRF attacks. It is a rule-based policy sent as a server response header, to restrict the sources a client loads scripts from. In [93], the HPKP is introduced. It aims to prevent misissued certificates. It enables web browsers to remember (pin) at least one public key of servers' certificates chain. The client uses the pinned key to validate the received certificate chain in future connections to servers.

3.5.3 Certificate-based

In [94], Szalachowski et al. proposed PoliCert. It introduced the idea of policy certificates. It allows domain owners to specify the CAs who can sign the domain's certificates. In addition, domain owners can specify a minimum TLS security level that the client should enforce in subsequent connections to servers, along with an error handling mechanism.

3.5.4 Agent-based

ForceHTTPS by Jackson and Barth enables strict TLS policy through the user agent [92], in addition to the previously mentioned header-based mechanism in Section 3.5.2. Similarly, both HSTS [39] and HPKP [93], are enabled via user agents' (e.g. browsers) preloaded lists, where opted in domains are preloaded in lists in the user agent, which are maintained by the agent's vendor.

3.6 Browser Warnings

While our work is not focused on usability, it is important to be aware of the usability concerns in deploying security enhancement mechanisms, and the most appropriate methods to make them effective. In what follows, we summarise some notable usability studies on browser warnings.

3.6.1 Effectiveness

Previous work such as the work of Jackson et al. [95] and Schechter et al. [96] showed that users tend to ignore passive security indicators such as the padlock and the Extended Validation (EV)[6] indicators. On the other hand, it has become clear that active warnings that interrupt the user's task and ask for the user's action are more effective than passive indicators as shown by several studies [96]–[99]. However, Reeder et al. [100], found that users' adherence to a warning vary with the context, such as the site reputation. They suggested considering contextual factors to improve warning messages. In [101], Zeng et al. compared notifying server administrators about HTTPS misconfigurations (including legacy TLS versions and ciphersuites) using direct emails alone compared to notification by emails combined with browser warnings. They found that the latter method is more effective than the former and that the browser warnings incentivise administrators to fix misconfigurations [101].

3.6.2 Caveats

Sunshine et al. [102], noted that warning messages should be avoided in benign situations to avoid the "habituation" effect which results from seeing a warning too often such that users underestimate the risk behind it.

[6] EV is a passive browser indicator that appears in the address bar but only for websites which have strongly verified identity.

4

Towards Forward Secure Internet Traffic

Contents

4.1 Introduction 47
4.2 Dataset 49
4.2.1 Top Domains 49
4.2.2 Random Domains 50
4.2.3 Random IPs 50
4.3 Methodology 51
4.3.1 Scanning Phase 51
4.3.2 Inspection Phase 52
4.3.3 Identifying Device Types 54
4.4 Results 54
4.4.1 Scanning Phase 54
4.4.2 Inspection Phase 56
4.5 Towards FS Internet Traffic 60
4.5.1 Deprecating non-FS-Ciphersuites in TLS Clients 60
4.5.2 Best Effort Forward Secrecy (BEFS) 62
4.6 Limitations 71
4.7 Conclusion 72

4.1 Introduction

Forward Secrecy (FS) is a security property in key exchange algorithms which guarantees that a compromise in the secrecy of a long-term private key does not compromise the secrecy of past session keys [18]. With a growing awareness of

long-term mass surveillance programs by governments and others, FS has become widely regarded as a highly desirable property. This is particularly true in the TLS protocol, which is used to secure Internet communication. Experience has shown the possibility of servers' long-term private key compromise. For example, RSA [21] long-term private keys have been compromised through prime factorisation, due to advancement in computing power [77], [78], and due to low entropy during keys generation [56]. Furthermore, long-term private keys can be compromised through implementation bugs as in the Heartbleed bug [79], through social engineering, or other attacks. Due to the increasing importance of FS, the new version of TLS, TLS 1.3, mandates it by design by prohibiting non-FS key exchange algorithms [16]. In recent years, it has been shown that some FS key exchange algorithms (e.g. ECDHE) can outperform non-FS (e.g. RSA) algorithms [58]. Despite recommendations to server administrators to select FS key exchange algorithms, non-FS key exchange algorithms are selected by more than 25% of the servers in our IPs dataset as we will show later. As a result, clients proceed with non-FS key exchange algorithms when connecting to these servers. This puts users' encrypted data at risk of future decryption by attackers who collect traffic today, and decrypt it whenever the targeted servers' private keys are compromised.

Motivated by the importance of FS in Internet security, and to examine negotiation transparency in configurable protocols, in this chapter, we take FS in pre-TLS 1.3 protocols as a case study. We analyse the sate of FS in pre-TLS 1.3 protocols, and introduce the discriminatory attacker model, which exploits the lack of negotiation transparency. We then discuss possible paths towards improving FS adoption, including proposing a new best effort approach, which aims to guide misconfigured servers towards FS. Our analysis aims to answer the following main questions:

1. *What is the percentage of servers that select non-FS-ciphersuites today?*

2. *Do servers that select non-FS-ciphersuites support FS-ciphersuites?*

3. *Do different dataset natures result in different trends in selecting and supporting FS-ciphersuites?*

Whilst addressing these main questions, the following side questions arose:

1. *What is the percentage of servers that select FS+non-AE-ciphersuites after the client's FS-ciphersuites enforcement*[1]*?*

2. *Do servers that select FS+non-AE-ciphersuites after the client's FS-ciphersuites enforcement support FS+AE-ciphersuites?*

3. *Can the client's FS-ciphersuites enforcement lead servers to lose the AE property?*

4. *Do servers that lose the AE property after the client's FS-ciphersuites enforcement support FS+AE-ciphersuites?*

4.2 Dataset

We build three datasets that we name: top-domains, random-domains, and random-ips. We end up with 999,884 distinct domains in the top-domains dataset, 4,960,390 distinct domains in the random-domains dataset, and 4,881,985 distinct IPv4 addresses in the random-ips dataset. The rationale behind choosing these three categories is to represent the real world web as much as possible. In what follows, we explain how we build and preprocess each dataset.

4.2.1 Top Domains

The top-domains dataset initial size is 1 million domains, obtained from the Alexa list of the top 1 million most visited domains globally [103], retrieved on August 22, 2018. We exclude the www-domains because we target plain-domains, which form the majority in the Alexa list. We define www-domains as those that start with "www.", we define plain-domains as those that do not start with "www.", and

[1]The term "client's FS-ciphersuite enforcement" refers to a client that offers FS-ciphersuites exclusively. The same applies for "client's FS+AE-ciphersuite enforcement" but the latter offers FS+AE-ciphersuites exclusively. More details are provided next in the methodology in Section 4.3

main-domains as those with a TLD prefixed by a single label, without any further subdomain (e.g. `example.com`). After excluding the www-domains, we end up with 999,884 domains that are mainly (around 94.81%) classified as main-domains.

4.2.2 Random Domains

The random-domains dataset initial size is 5 million random domains obtained from a large dataset that contains 54,063,220 distinct (alphabetically unordered) domains that successfully completed a TLS handshake in Amann et al. [75], and have been collected from multiple sources. To maintain consistency with the top domains dataset format, we extract 5 million domains from [75] that are classified as both plain-domains and main-domains. We limit our domains' TLDs to gTLDs that do not include "multi-level" TLDs such as ccTLDs (e.g. "ac.uk"). This is to avoid the complexity of distinguishing domains that have subdomains from domains that have ccTLDs. To avoid repeated domains, from the 5 million random domains, we exclude the domains that exist in the top domains dataset, either "as is" or as a main-domain of a subdomain in the top domains dataset (the top domains dataset contains a small percentage of subdomains). We identify the subdomains that appear in the top domains dataset in two steps: first, by using a regular expression, we extract the domains that have more than one dot ".". Second, with the aid of the "tldextract" Python library [104], we distinguish subdomains from domains with ccTLDs such as "example.ac.uk" (the latter is considered a main-domain). The tldextract identifies the second-level domain by maintaining an updated list of all known public TLDs obtained from Mozilla's public suffix list [105]. We end up with 4,960,390 distinct random domains.

4.2.3 Random IPs

The random-ips dataset initial size is 5 million distinct IPv4 addresses that have completed a successful TLS handshake with Censys, a search engine and a database for servers and network devices on the Internet [61], retrieved from the Censys IPv4 dataset on October 20, 2018, through research access to the Censys database. To

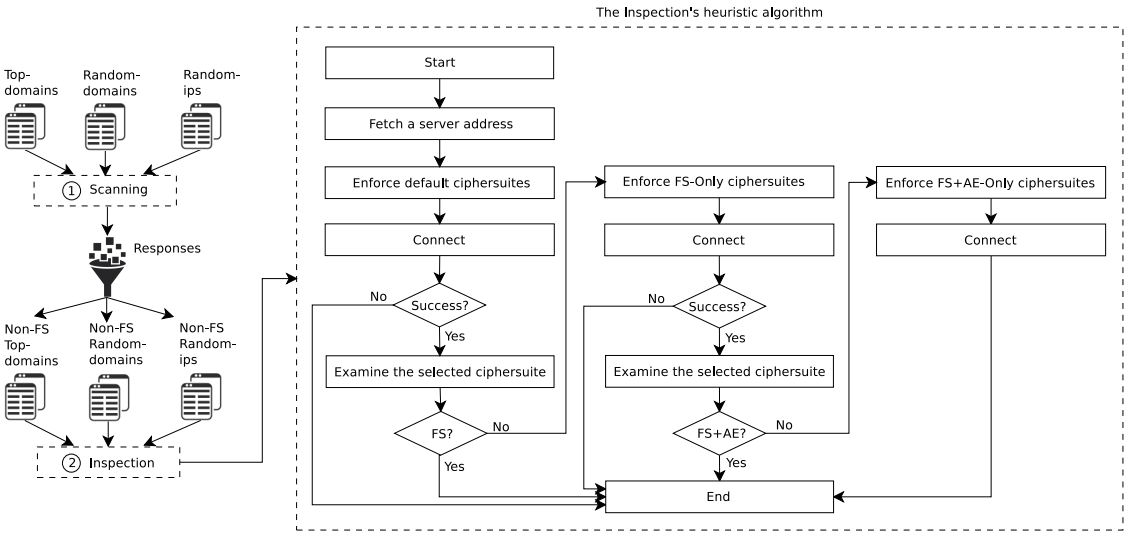


Figure 4.1: A general overview of our methodology showing the two phases and their input.

avoid repeated IPs, from the 5 million IPs, we exclude the IPs that are associated with any domain that has responded to a handshake in the scanning or inspection phases (further details on the scanning and inspection phases will be provided in the methodology in Section 4.3). For this reason, we build the random IPs dataset after we finish the domains' datasets scanning and inspection phases. We end up with 4,881,985 IPs.

4.3 Methodology

As depicted in Figure 4.1, our methodology consists of two main phases: a scanning phase followed by an inspection phase.

4.3.1 Scanning Phase

We consider the scanning phase as an exploration phase. In this phase, for each server address in our datasets, we perform a TLS handshake using the tls-scan tool [106], an open source fast TLS scanner capable of performing concurrent TLS connections. We customise the tls-scan to utilise the OpenSSL 1.1.0g library, and to support Chrome's latest version pre-TLS 1.3 ciphersuites, which support various ciphersuites that provide FS, or AE, or none of them (see the pre-TLS 1.3

ciphersuites in Table A.1 in Appendix A.3 for the client's ciphersuites list). We choose to base the scanning client on Chrome's ciphersuites because Chrome is the most representative TLS client on the Internet. At the time of writing this chapter (February 2019), Chrome's usage is 79.7% [107], and in another source, Chrome's market share in February 2019 is 71.58% [108][2]. The tls-scan includes the SNI extension for domain name based scans by default. We set the timeout argument to 5 seconds, and the concurrency argument to 50 connections. We ran the scans between August 23, 2018 and October 21, 2018 at the University of Oxford in discrete intervals based on the dataset.

4.3.2 Inspection Phase

After the scanning phase is complete, we extract the responding addresses that selected non-FS-ciphersuites in the scanning phase. Each dataset is inspected within a maximum of 48 hours after its scanning phase is complete. For the inspection phase, we develop a TLS client that implements our heuristic procedure (see Figure 4.1), which works as follows, for each server's address:

1. The client performs a TLS handshake based on Chrome's pre-TLS 1.3 default ciphersuites (see the pre-TLS 1.3 default ciphersuites in Table A.1 in Appendix A.3 for the client's ciphersuites list). This first handshake of the inspection phase is similar to the scanning phase handshake[3]. The inspection's first handshake serves as a confirmation of the server's selected ciphersuite. It records the server's selected ciphersuite from a default client's view just before the heuristic procedure starts. If the handshake failed, the client records the error, and the heuristic procedure ends here.

[2]These figures should be taken with cautious: first, they are for the desktop platform only. The figure provided by [108] drops to 62.41% if we choose the statistics for all platforms. Second, they are based on different methodologies. The figure by [107] is based on the W3Schools website visits, which receives over 50 million monthly visits. On the other hand, the figure provided by [108] is based on over 10 billion monthly page views ("and not unique visitors") gathered from over 2 million websites. Third, the figures get updated monthly. Therefore, the figures we report are for February 2019 only.

[3]Except that our inspection client does not support SSL 3.0 while the scanning tls-scan client supports SSL 3.0. However, we analyse FS regardless of the client's supported versions.

2. Upon receiving the server's response to the first handshake (step 1), the client checks the server's selected ciphersuite: if it is an FS-ciphersuite, this means that the server has changed its behaviour after the scanning phase since all the inspection input addresses are for servers that selected non-FS-ciphersuites in the scanning phase. The client records the server's response, and the heuristic procedure for this server ends here. Otherwise, if the server's selected ciphersuite is still a non-FS-ciphersuite, we classify this server as "stable", i.e. consistently selects a non-FS-ciphersuite. The client then updates its TLS context to support FS-ciphersuites *exclusively* (see the pre-TLS 1.3 FS-only ciphersuites in Table A.1 in Appendix A.3 for the client's ciphersuites list). The set of FS-ciphersuites may or may not support AE, i.e. it contains FS+AE-ciphersuites and FS+non-AE-ciphersuites. This context is more restricted than the default one.

3. The client then performs a second handshake utilising the new FS-ciphersuites context. If the handshake failed, the client records the error and the heuristic procedure for this server ends here.

4. Upon receiving the server's response to the second handshake (step 3), the client checks the server's selected ciphersuite: if it is an FS+AE-ciphersuite, this means that the server supports FS+AE-ciphersuite after the client's FS-ciphersuites enforcement. The client then records the server's response, and the heuristic procedure for this server ends here. Otherwise, if the server's selected ciphersuite is an FS+non-AE-ciphersuite, the client updates its context to support FS+AE-ciphersuites *exclusively* (see the pre-TLS 1.3 FS+AE-only ciphersuites in Table A.1 in Appendix A.3 for the client's ciphersuites list). This context is more restricted than the FS-ciphersuites context.

5. The client then performs a third handshake utilising the new FS+AE-ciphersuites context. If the handshake failed, the client records the error and the heuristic procedure ends here.

6. Upon receiving the server's response, the client records the response. The heuristic procedure for this server ends here.

We develop and run the inspection client using Python 3.6.5. Similar to the tls-scan client in the scanning phase, it utilises OpenSSL 1.1.0g. We enable the SNI for the top and random domains inspection (the IPs dataset does not need the SNI), and we set the timeout to 5 seconds. The results are then stored and analysed using MySQL database and queries (see Appendix B.1 for queries examples).

4.3.3 Identifying Device Types

We classify device types into two categories: ordinary web servers and network devices. We use the term "network device" to refer to non-ordinary TLS servers, e.g. embedded web servers in network devices such as routers. To identify the device type, we input the IPs of the dataset in question in a query to Censys database to obtain the IPs metadata. We then produce a breakdown of the responding IPs grouped by the device type. We base our device types queries on the IPs, i.e. in the domains datasets, we first extract the distinct IPs behind the domains, because Censys is mainly an IP based engine. We query the Censys snapshot that dates to the starting date of the scan or inspection (depending on the phase) of the dataset in question. Censys labels the device type of the network devices that it identifies, e.g. "DSL/cable modem". If the device type field is empty, this means that the device is either an ordinary web server, or a network device that Censys cannot identify. Finally, we do not always obtain 100% responses for the IPs whose metadata we query from Censys. However, overall, the percentages of the responses that we receive are between 98.36% to 100% (depending on the dataset) of the IPs we query.

4.4 Results

4.4.1 Scanning Phase

In this phase, we input the servers' addresses in our datasets. The results of the scanning phase are summarised in Table 4.1.

Table 4.1: Summary of the scanning results. Every additional indentation means that the percentages are computed out of the previous level results. The "% Network devices" are computed over the responding IPs to Censys metadata query (exact numbers are provided in text).

	Datasets					
	top-domains		random-domains		random-ips	
Dataset size	999,884		4,960,390		4,881,985	
Responding servers	814,333	(81.44%)	3,221,249	(64.94%)	4,477,279	(91.71%)
Distinct IPs	468,346	(57.51%)	690,912	(21.45%)	4,477,279	(100%)
% Network devices	466	(0.10%)	1208	(0.18%)	518,988	(11.59%)
Select non-FS	43,756	(5.37%)	241,994	(7.51%)	1,171,101	(26.16%)

4.4.1.1 Responding servers

As illustrated in Table 4.1, the highest percentage of responses is in random IPs (91.71%), followed by top domains (81.44%), and finally random domains (64.94%). The response rate is influenced by the dataset category. Both the IPs and top domains datasets are recent. That is, the addresses in the IPs dataset have recently completed a TLS handshake with the Censys engine [61], and TLS adoption in top domains is high. The low response rate in random domains (64.94%) is very likely due to the dataset age. It is obtained from a previous study that was published in 2017 [75]. Hence, many domains could have gone down since then.

In terms of device types, from the responding top domains, there are 468,346 (57.51%) distinct IPs behind all the top domains. We receive metadata responses for 464,191 (99.11%) of them from the Censys database. Of those, only 466 (0.10%) IPs are labelled as network devices. From the responding random domains, there are 690,912 (21.45%) distinct IPs behind them. We receive metadata responses for 686,085 (99.30%) of them. Of those, there are 1208 (0.18%) labelled as network devices. From the responding random IPs, we receive metadata responses for all of them (100%). Of those, there are 518,988 (11.59%) IPs labelled as networked devices. Clearly, the percentage of network devices in the random IPs is higher than that in the top and random domains.

Table 4.2: Summary of the inspection results. Every additional indentation means the percentages are computed out of the previous level results. The input of the inspection phase is the servers that selected non-FS-ciphersuites in the scanning phase. The "% Network devices" are computed over the responding IPs to Censys metadata query (exact numbers are provided in text).

	Datasets					
	non-FS top-domains		non-FS random-domain		non-FS random-ips	
Dataset size	43,756		241,994		1,171,101	
Responding servers	43,374	(99.13%)	240,519	(99.39%)	1,111,802	(94.94%)
Select non-FS (stable)	43,158	(99.50%)	240,274	(99.90%)	1,111,174	(99.94%)
Distinct IPs	33,474	(77.56%)	61,522	(25.60%)	1,111,174	(100%)
% Network devices	76	(0.23%)	361	(0.59%)	434,076	(39.06%)
Support FS	16,916	(39.20%)	58,636	(24.40%)	160,706	(14.46%)
Distinct IPs	12,545	(74.16%)	13,839	(23.60%)	160,706	(100%)
% Network devices	12	(0.10%)	27	(0.20%)	1503	(0.94%)
Select FS+non-AE	10,091	(59.65%)	38,583	(65.80%)	93,566	(58.22%)
Support FS+AE	1629	(16.14%)	1289	(3.34%)	24,128	(25.79%)
Lose AE	2686	(26.62%)	1768	(4.58%)	12,769	(13.65%)
Support FS+AE	45	(1.68%)	91	(5.15%)	4668	(36.56%)

4.4.1.2 Servers that select non-FS-Ciphersuites

From the responding servers, we find 5.37% of the top domains, 7.51% of the random domains, and 26.16% of the random IPs, select non-FS-ciphersuites. The lowest percentage is in the top domains, the highest is in the random IPs, while in the random domains, it is slightly higher than that in the top domains. The fact that the random IPs dataset has the highest percentage of network devices can be correlated to the high percentage of servers that select non-FS-ciphersuites. We can confirm this in the inspection phase when we look closer at the device types of those servers that select non-FS-ciphersuites.

4.4.2 Inspection Phase

In this phase, we input the addresses of servers that select non-FS-ciphersuites in the scanning phase (the highlighted row in Table 4.1). Table 4.2 summarises the inspection phase results.

4.4.2.1 Responding servers

As Table 4.2 illustrates, over 99% of top and random domains, and 94.94% of IPs that select non-FS-ciphersuites in the scanning phase, have responded to our inspection client's handshake. The low response rate in the IPs dataset compared to the top and random domains datasets is very likely attributed to SSL 3.0 devices as our inspection client does not support SSL 3.0, while the scanning client does. It is also very likely that those non-responding IPs are mostly for network devices since using legacy versions in network devices is more common than that in ordinary web servers [60].

4.4.2.2 Servers that still select non-FS-Ciphersuites (stable)

In our work, we use the term "stable" to refer to servers that consistently select non-FS-ciphersuites in both the inspection's first handshake and the scanning handshake. As shown in Table 4.2, clearly, the stability in selecting non-FS-ciphersuites among all datasets is high (over 99% in all datasets), despite the difference in the supported protocol versions in the scanning and inspection clients (the scanning client supports SSL 3.0 while the inspection does not). This suggests that, to some extent, servers' selected ciphersuites are not affected by the negotiated versions. See Listing B.1 in Appendix B.1 for the MySQL query.

In terms of device types, out of the top domains that select non-FS-ciphersuites, there are 33,474 (77.56%) distinct IPs behind all these domains. Of those, we receive metadata responses for 33,079 (98.82%) IPs from the Censys database. Of those, there are 76 (0.23%) IPs labelled as networked devices. Of the random domains that select non-FS-ciphersuites, there are 61,522 (25.60%) distinct IPs behind them. We receive metadata for 61,176 (99.44%) IPs from Censys. Of those, there are 361 (0.59%) IPs labelled as networked devices. Of the random IPs that select non-FS, we receive metadata for 1,111,174 (100%). Of those, there are 434,076 (39.06%) IPs labelled as networked devices. Network devices represent no more than 0.59% of top and random domains that select non-FS-ciphersuites. However, more than a third of servers that select non-FS-ciphersuites in the random IPs dataset are labelled

as network devices. The high percentage of network devices in the random IPs is likely the reason for the high percentage of servers that select non-FS-ciphersuites.

4.4.2.3 Servers that select non-FS-Ciphersuites, but support FS-Ciphersuites

Of the top domains, random domains, and random IPs that select non-FS-ciphersuites in the inspection phase, we find that 39.20% of top domains, 24.40% of random domains, 14.46% of random IPs, do support FS-ciphersuites. The top-domains are the highest, followed by the random domains, and finally, the random IPs are the lowest. Interestingly, this is a shifted paradigm for the percentages of servers that select non-FS-ciphersuites that is shown in Table 4.1, where the random IPs have the highest percentage and the top domains have the lowest percentage. The results reflect that the lack of FS-ciphersuite selection in the top and random domains is to a large extent due to misconfiguration, while in the random IPs, it is mostly due to lack of support. See Listing B.2 in Appendix B.1 for the MySQL query.

In terms of device types, out of the top domains that select non-FS-ciphersuites but support FS-ciphersuites, there are 12,545 (74.16%) distinct IPs behind them. We receive metadata responses for 12,339 (98.36%) IPs. We find 12 (0.10%) of them are labelled as networked devices. Of the random domains that select non-FS-ciphersuites but support FS-ciphersuites, there are 13,839 (23.60%) distinct IPs behind them. We receive metadata responses for 13,744 (99.31%) IPs. Of those, 27 (0.20%) are labelled as network devices. Of the random IPs that select non-FS-ciphersuites but support FS-ciphersuites, we receive metadata responses for 160,706 (100%) IPs. Of those, 1503 (0.94%) are labelled as network devices. The results show that the majority of those devices are not identified as network devices, even in the random IPs dataset that shows the highest percentage of network devices. Those servers that select non-FS-ciphersuites and turned to support FS-ciphersuites are not network devices. Therefore, most of the network devices that select non-FS-ciphersuites, do not support FS-ciphersuites.

4.4.2.4 Servers that select FS+non-AE-Ciphersuites after enforcing FS-Ciphersuites

Out of the top domains, random domains, and random IPs that support FS-ciphersuites after enforcement (row label "Support FS" in Table 4.2), there are 59.65% top domains, 65.80% random domains, and 58.22% random IPs that select FS+non-AE-ciphersuites. The reason for selecting non-AE can be attributed to the fact that TLS 1.0 and TLS 1.1 do not support AE-ciphersuites [109], and these devices might be running legacy versions of TLS. Otherwise, this is attributed to misconfiguration. To better understand this situation, we next check whether those servers support FS+AE-ciphersuites or not.

4.4.2.5 Servers that select FS+non-AE-Ciphersuite after enforcing FS-Ciphersuites, but support FS+AE-ciphersuites

Of the top domains, random domains, and random IPs that select FS+non-AE-ciphersuites after FS-ciphersuites enforcement (row label "Select FS+non-AE" in Table 4.2), there are 16.14% top domains, 3.34% random domains, and 25.79% random IPs, that support FS+AE-ciphersuite. At this point of the heuristic procedure, the majority of the IPs do not belong to network devices. The majority of the top and random domains that select FS+non-AE-ciphersuites do not support FS+AE-ciphersuites. However, selecting FS+non-AE-ciphersuites while supporting FS+AE-ciphersuites in the IPs dataset is the highest, which we classify as misconfiguration.

4.4.2.6 When enforcing FS-Ciphersuite causes losing the AE property

Of the top domains, random domains, and random IPs that select FS+non-AE-ciphersuites after enforcing FS-ciphersuites (row label "Select FS+non-AE" in Table 4.2), 26.62% of top domains, 4.58% of random domains, and 13.65% of random IPs, were selecting AE before enforcing FS-ciphersuites, i.e. were selecting non-FS+AE-ciphersuites. This can be either because they do not support any FS+AE-ciphersuites, or due to misconfiguration. This will be clarified next.

4.4.2.7 Servers that lose the AE property after enforcing FS-Ciphersuites, but support FS+AE-ciphersuites

Out of the top domains, random domains, and random IPs that lose the AE property after enforcing FS (row label "Lose AE" in Table 4.2), we find 1.68% of top domains, 5.15% of random domains, and 36.56% of random IPs, do support FS+AE-ciphersuites. The results reflect that losing the AE property after enforcing the FS in the top and random domains is to a large extent due to a lack of support for FS+AE-ciphersuites, but in the random IPs, it is mostly due to misconfiguration.

4.5 Towards FS Internet Traffic

In this section, we discuss possible paths towards FS Internet traffic from a client's perspective. Then, we propose and evaluate a novel client-side mechanism that we call Best Effort Forward Secrecy (BEFS), and an extension of it that we call Best Effort Forward Secrecy and Authenticated Encryption (BESAFE). We choose to focus our discussion and solutions on clients because unlike servers, clients are controlled by few players, e.g. browser vendors. Client-side security enhancement mechanisms are easier to adopt, as shown by recent adoptions of client-side mechanisms such as Google's CT [72].

4.5.1 Deprecating non-FS-Ciphersuites in TLS Clients

The most straight forward approach towards FS Internet traffic is deprecating non-FS-ciphersuites from TLS clients. As a result, these clients will not be able to establish TLS connections with servers that do not support FS-ciphersuites. This is a conservative approach that has been taken by browser vendors and standardisation bodies in the past with some protocol versions and algorithms such as SSL 3.0 [110] and RC4 [111], after their insecurity has become clear. However, deprecating non-FS-ciphersuites now can be more problematic than the case of deprecating SSL 3.0 and RC4 in 2014 and 2016, respectively. By way of comparison, Lee

et al.[4] conducted a survey in 2006 to assess the cryptographic strength of TLS servers [55]. It shows that 98.36% of the surveyed servers support TLS 1.0, the latest version at the time of the study, and 57.17% of the servers support AES encryption, which was in its early years as it was standardised in 2001 [112]. In light of these figures, we speculate that SSL 3.0 and AES adoption when they were deprecated by most browsers in 2014 and 2016 respectively was over 99%. On the other hand, in our results, we calculate an approximation of the servers that support FS-ciphersuites in each dataset. To this end, we first calculate the number of servers that select non-FS-ciphersuites and do not support FS-ciphersuite which can be derived from Table 4.2 by calculating ("Select non-FS (stable)"−"Support FS"), and then subtracting those results from the overall responses in Table 4.1's row label "Responding servers", which gives: 788,091 (96.78%) top domains, 3,039,611 (94.36%) random domains, and 3,526,811 (78.77%) random IPs. Our results are in line with Censys's October 26, 2018 snapshot figures that show 97.44% of Alexa's top domains, and 77.94% of IPs in the IPv4 space, support FS-ciphersuite. However, our results are more accurate as our client not only counts servers that select FS, but also servers that support FS but select non-FS, which can be guided through client's enforcement as we will explain next. In addition, our client utilises modern client ciphersuites. On the other hand, Censys only measures servers that select FS, and utilises somewhat legacy ciphersuites. We conclude that the percentages of servers that support FS-ciphersuites are less than that in RC4 and SSL 3.0 cases, especially in the IPs datasets. The lack of supporting FS-ciphersuites by those servers can be explained by the fact that until recent years, (EC)DHE key exchange algorithms have been viewed as resource exhaustive compared to RSA key exchange, despite the fact that this argument is no longer true with the ECDHE variant as shown in [58].

[4]Despite the study's age (conducted in 2006), to the best of our knowledge, [55] is the only study that tried to assess servers' supported ciphersuites prior to deprecating RC4 and SSL 3.0. Note that identifying the supported ciphersuites for a server is different from identifying the selected ciphersuite. The former requires multiple handshakes, while the latter requires a single handshake, for each server.

4.5.2 Best Effort Forward Secrecy (BEFS)

4.5.2.1 Overview

The gist of our BEFS mechanism is guiding (forcing) misconfigured servers towards FS-ciphersuites. As explained in Section 2.1.2, in ordinary TLS clients such as web browsers, the client offers default ciphersuites, which includes FS-ciphersuites and non-FS-ciphersuites. Upon receiving the client's offer, a server that does not support or does not prefer to select an FS-ciphersuite will select a non-FS-ciphersuite, and sends its selected ciphersuite to the client. The client accepts the server's choice, and the rest of the communication proceeds with a non-FS-ciphersuite. On the other hand, in BEFS, we exploit the TLS ciphersuite negotiation dynamics to influence (bias) the server's choice towards FS-ciphersuites. That is, a BEFS-enabled client first offers FS-ciphersuites exclusively $[a_{1_{fs}}, ..., a_{n_{fs}}]$. Upon receiving the client's offered ciphersuites, a server that supports FS-ciphersuites will be guided (forced) to select an FS-ciphersuite $a_{S_{fs}}$, even if it prefers to select a non-FS-ciphersuite, since FS-ciphersuites are the only offered ciphersuites as illustrated in Figure 4.2. As shown in Table 4.2, of the servers that select non-FS-ciphersuites, there are between 14.46% to 39.20% that do support FS-ciphersuites, which can benefit from the BEFS enforcement mechanism. If the server indeed does not support FS-ciphersuites, it will return a failure alert (see Section 2.1.2 for a background on TLS version and ciphersuite negotiation). In this case, the BEFS-enabled client makes a second handshake utilising default ciphersuites $[a_{1_{fs}}, ..., a_{n_{nonfs}}]$, which includes non-FS-ciphersuites in addition to the previously offered FS-ciphersuites as Figure 4.3 illustrates. Hence, a server that does not support FS-ciphersuites can still select a non-FS-ciphersuite $a_{S_{nonfs}}$ after the client falls back. BEFS can be viewed as a form of the "Opportunistic Security" concept [113], but at the FS property level. That is, it guides servers to select FS whenever they support it.

4.5.2.2 The Fallback

We now address the fallback aspect. We define three categories of client-side fallbacks: silent fallback, interactive fallback, and signalled fallback. In what follows,

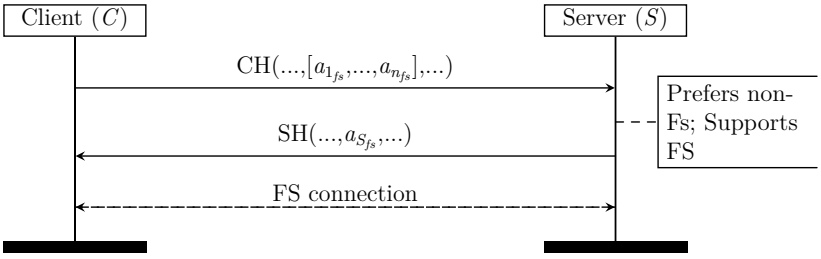


Figure 4.2: A BEFS-enabled client handshake when the server prefers to select a non-FS-ciphersuite while supporting FS-ciphersuites. The server is forced to select FS-ciphersuite through client FS-ciphersuite enforcement.

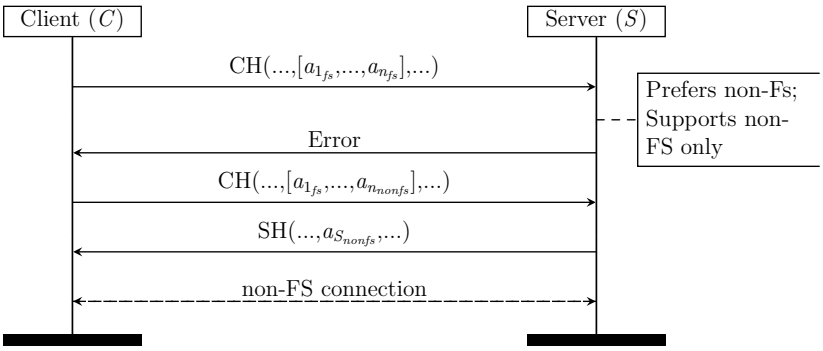


Figure 4.3: A BEFS-enabled client handshake when the server does not support FS-ciphersuites. The client falls back to non-FS-ciphersuites only when the server does not support FS.

we explain them in light of the BEFS mechanism.

1. **Silent fallback:** Silent fallbacks do not involve the user or the server. If used in BEFS, if the FS-ciphersuites handshake failed, the client falls back to default ciphersuites (which include non-FS-ciphersuites), in the background, and performs a second handshake utilising default ciphersuites. Silent fallbacks remove the security decision making overhead from the user at the cost of security. Silent fallbacks do not provide security against active attackers who can perform downgrade attacks. BEFS with silent fallback is secure against passive attackers, which adds significant value in the case of FS. It makes mass

surveillance more difficult to achieve as the attacker has to actively perform downgrade attacks for each session.

2. **Interactive fallback:** Interactive fallbacks involve the user. If used in BEFS, when the FS-ciphersuites handshake fails, the client (e.g. web browser) presents an interrupting warning message and asks the user whether to proceed or not. If the user chooses to proceed, the client falls back from FS-ciphersuites to default ciphersuites and performs a second handshake. Otherwise, if the user chooses not to proceed, the client does not fall back, and aborts the TLS handshake. Interactive fallbacks provide security against active attackers. Interactive fallbacks are similar to the widely known self-signed certificate active warnings [97]. Active security warnings have been shown to be more effective than passive ones such as passive indicators that do not interrupt the user's task [97]. While active security warnings can be viewed as disturbance for users, they have been shown to be more effective in incentivising server administrators to fix their servers' TLS security related configurations [101]. However, active security warnings have to be used with caution in order to not cause the habituation effect, where users ignore them because they see them too often [102]. Therefore, if the majority of servers that do not support FS-ciphersuites (i.e. those that require fallback) are network devices, interactive fallback can be acceptable, as these devices are normally visited infrequently by a limited number of users, such as the device's owner.

3. **Signalled fallback:** Signalled fallbacks involve the server. Therefore, if they are not incorporated in the protocol by design, they require modifications or updates to the server, e.g. a patch to the TLS implementation, to enable the server from interpreting the client's signal. In signalled fallbacks, the client sends a signal, i.e. a special value, to inform the server that the client has performed a fallback. The server aborts the handshake if it is not expecting a fallback, e.g. in BEFS case, if the server supports FS-ciphersuites. Signalled fallbacks provide security against active attackers, if we assume

authenticated messages. Signalled fallbacks have been proposed in the TLS fallback SCSV [114]. It has been used to mitigate TLS version downgrade attacks, mainly the POODLE attack [32], and has been widely adopted as shown in [75]. In BEFS, our problem deals with misconfigured servers and less security-aware server administrators. Had they been security-aware, they would have configured their servers to select FS-ciphersuites. Therefore, in BEFS case, we do not consider signaled fallbacks as a solution that can be adopted quickly. Therefore, we do not include it in our analysis in the coming section.

4.5.2.3 BEFS Security Analysis

We now analyse the security of BEFS against three attacker models: passive network attacker, active network attacker, and our newly introduced discriminatory attacker.

1. **Passive Network Attacker:** Passive attackers can collect network traffic, but cannot interfere (e.g. modify, inject, replay, or drop) with protocol messages. They may obtain access to the server's long-term private key at some point in the future. Once the server's long-term private key is compromised, a passive attacker who has been collecting non-FS network traffic can now decrypt it. BEFS aims to ensure the selection of FS-ciphersuites whenever the server supports FS-ciphersuites. In FS-ciphersuites, an ephemeral key is generated for each session, and this key is not encrypted with the server's long-term private key. By selecting FS-ciphersuites, if the server's long-term private key is compromised, the attacker cannot compromise past session keys. In TLS, the (EC)DHE key exchange algorithms are provably secure against passive attackers [115]. Therefore, BEFS with all types of fallback mechanisms is secure against passive attackers.

2. **Active Network Attacker:** Unlike passive attackers, active attackers can interfere with protocol messages, e.g. by modifying, injecting, replaying, or dropping messages. Similar to passive attackers, they may obtain access to

the server's long-term private key at some point in the future. Hence, they may be able to decrypt non-FS-ciphersuite traffic. BEFS security against active attackers can be analysed with the two fallback mechanisms explained earlier in Section 4.5.2.2. First, in terms of BEFS with silent fallback, since the user of a BEFS-enabled client with silent fallback is not aware of the fallback, an active attacker can perform a downgrade attack by dropping the initial FS-ciphersuite's handshake message to lead the client to fall back and perform a default handshake. Hence, misconfigured servers that select non-FS-ciphersuites but support FS-ciphersuite will not be guided, i.e. will select non-FS-ciphersuite, while with BEFS, they will be guided (forced) to select FS-ciphersuites instead. BEFS with silent fallback does not provide security against active attackers. Second, we analyse BEFS with interactive fallback against an active attacker. This moves the security decision to the user. Users can be classified into two categories: security-aware users, who read the warning message and reject the fallback when they care about FS. The second category of users is less security-aware users, who will not do so. BEFS with interactive fallback and security-aware users is secure against active attackers. The warning message content and the users' reactions to it are beyond the scope of this chapter. BEFS with interactive fallbacks can find its application in special browser modes for sensitive communications, in the same vein as Chrome's incognito and Firefox private modes, which are available for privacy-aware users.

3. **Discriminatory Attacker:** The discriminatory attacker is located at the server and discriminates against its clients in terms of the security level it provides to them (FS-ciphersuite vs. non-FS-ciphersuite in our case). See Figure 4.4 for an abstract overview. The discriminatory attacker model is applicable to semi-trusted servers running protocols such as TLS, which gives the server the power of selecting some parameters that define the security level of a particular session, exemplified by the ciphersuite in our case, while the client has no means of verifying the server's actual capabilities, i.e. justifying

the server's decision if it selects a non-preferred ciphersuite such as a non-FS-ciphersuite. This power can be abused by semi-trusted servers to discriminate against their users, for a powerful third party's advantage. The discriminatory attacker can be compelled by, or collude with the third party, such as government intelligence, to weaken the security of *some* connections, e.g. those coming from specific geographic locations. In our case, the discriminatory attacker denies the FS property to some users, whilst enabling it for others. The discriminatory attacker (server) can then provide its long-term private key that is used for digital signatures and non-FS session keys (*pms*) encryption to the powerful third party, after the key's expiration, when it is no longer used by the server. This allows the third party to decrypt the data of those users who have been discriminated against, but not the data of other users who have been provided with strong security, i.e. FS-ciphersuite in our case. To illustrate, it is useful to present use case scenarios where discrimination in the FS property can be harmful. Imagine a global service provider for Internet search engine, email system, or critical infrastructure system, colluding with a powerful third party attacker (e.g. an intelligence agency), to provide non-FS traffic to some of its clients, e.g. from certain regions, to enable mass surveillance against those clients.

This attacker model gives the semi-trusted server several advantages compared to giving every session key or the decrypted data itself to the third party, which is impractical for servers to carry out, especially in the case of large-scale surveillance. Another advantage to the semi-trusted server lies in the minimal liabilities (e.g. legal) in being directly involved in leaking their users' data, or in giving their private key to the attacker during the key's lifetime. Such an attacker model is not far from the export grade cryptography law that was mandated until the late 90s, where software vendors, for example, were compelled to weaken the security of software exported outside the United States (US), to enable the US intelligence to break their security. Furthermore, leaked confidential documents by Edward Snowden suggest similar scenarios,

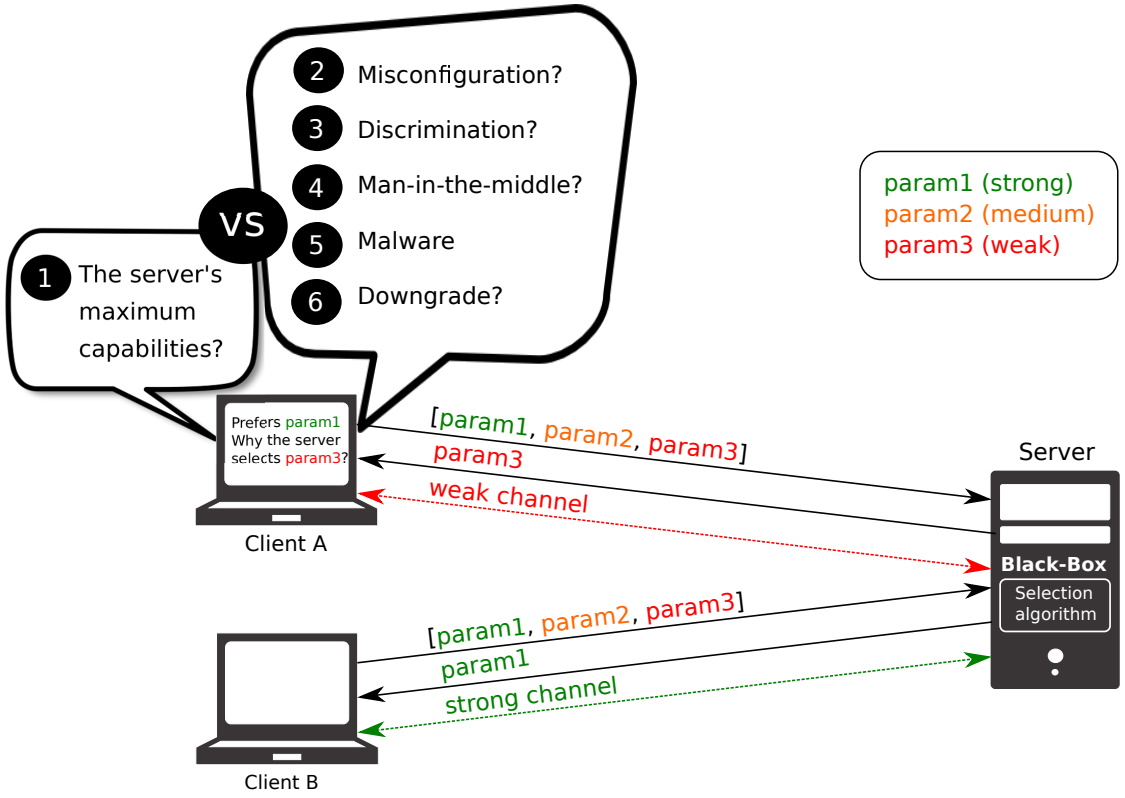


Figure 4.4: Illustration of our newly introduced discriminatory attacker model in parameters negotiation in security protocols with a server-dominant negotiation model such as the TLS protocol. The term "Param" denotes parameter.

where giant companies collude with government intelligence by introducing backdoors that are known to, and can be exploited by, those powerful attackers (e.g. the "PRISM" program) [116]. Figure 4.4 gives an abstract overview of the discriminatory server attacker model in parameters negotiation.

Our discriminatory attacker model is inspired by the "malicious-but-cautious" [117], and the Secretly Embedded Trapdoor with Universal Protection (SETUP) [118] attacker models. The malicious-but-cautious model assumes a cloud service provider (server) can act maliciously but is cautious not to leave a verifiable trace of its malicious behaviour. However, it does not assume that the malicious server is willing to enable a third party to access some users' data. On the other hand, the SETUP model assumes a cryptographic system (server in our case) can enable a third party to secretly obtain secret information such as the private key that decrypts the encrypted data from the system's encrypted output. Our discriminatory attacker weakens the security against

some users for a third party's advantage, and is also cautious not to leave a verifiable trace of its malicious behaviour, e.g. by selecting a supported but non-preferred ciphersuite (non-FS-ciphersuite), as it is still accepted by most clients for backward compatibility.

To better analyse BEFS against the discriminatory attacker (server), we further classify this attacker into two variants: weak discriminatory and strong discriminatory. In the weak variant, the attacker submits to the client's offer (ciphersuites in our case). That is, if the client offers strong choices exclusively, the weak discriminatory has no choice but to select from them, mainly to avoid detection. In the strong variant, the attacker refuses to select strong choices, which forces the client to fallback in order to connect to the server. BEFS with silent fallback is secure against the weak discriminatory attackers. However, strong discriminatory attackers require interactive fallback and security-aware users. In today's real world settings, the weak variant is more realistic. However, the strong variant can be detected through BEFS and security-aware users. Note that this analysis of BEFS against a discriminatory attacker is independent of considerations about the communication channel. That is, if an active attacker is present in the communication channel, interactive fallback is required with both variants of the discriminatory attacker, in order for BEFS to meet its security goal.

4.5.2.4 Best Effort Forward Secrecy and Authenticated Encryption (BESAFE)

Given the fact that more than 50% of the servers select FS+non-AE-ciphersuite after enforcing FS-ciphersuites and that between 16.14% to 25.79% of them support FS+AE-ciphersuites, as an extension to BEFS, we propose BESAFE which adds an additional step to enforce not only FS-ciphersuites, but also FS+AE-ciphersuites. This improvement adds an additional restriction: the client offers FS+AE-ciphersuites exclusively at the first handshake attempt. If it failed, the client falls back to BEFS: it tries FS-ciphersuites exclusively, and if it failed, it falls back

to default ciphersuites. The BESAFE mechanism guides servers towards FS+AE-ciphersuites. Similar to BEFS, BESAFE is secure against passive attackers, or weak discriminatory attackers with all types of fallbacks, and against active attackers, or strong discriminatory attackers with interactive fallback and security-aware users.

4.5.2.5 BEFS and BESAFE Performance

We measure the latency that BEFS and its extension BESAFE incur into a TLS connection establishment with domains that do not support FS-ciphersuites, i.e. when more than one attempt is performed to complete a TLS handshake (otherwise, in BEFS, if the server supports FS-ciphersuites, and in BESAFE, if the server supports FS+AE-ciphersuites, there will be a single handshake as normal and no additional latency is incurred). To this end, we extract 5000 top domains that do not support FS-ciphersuites from our results. We implement a TLS client that supports Chrome's pre-TLS 1.3 default ciphersuites using Python 3.6.5 and utilising OpenSSL 1.1.0g. We disable TLS certificate validation and session tickets (resumption), and enable the SNI. Our client performs three consecutive handshakes for each domain: default, BEFS-enabled, and BESAFE-enabled handshakes. We run the client on a machine equipped with a 3.2 GHz Intel Core i5 processor, 8 GB of RAM, and a 1000 Mbps wired Ethernet card that has a public IPv4 address at the University of Oxford. We measure the time to complete a TLS handshake in a socket connection in milliseconds using the "process_time()", a process-wide timer in Python's "time" module. We count the domains that triggered BEFS and BESAFE to resort to default ciphersuites (i.e. do not support FS-ciphersuites), and also responded to the default TLS handshake. Then we extract the maximum, minimum, and average time they take for each of the three handshake types. There are 4501 domains that do not support FS-ciphersuites and responded to the three types of handshakes we examine. The results based on these responses are summarised in Table 4.3. We can also infer the latency that BESAFE incurs into a connection to a server that does not support FS+AE-ciphersuites but supports FS+non-AE-ciphersuites from the BEFS latency (since both require two attempts).

Table 4.3: The BEFS and BESAFE mechanisms latency in ms compared to the default one when connecting to servers that do not support FS-ciphersuites.

TLS Client	Max.	Min.	Avg.
Default	4.10	0.64	1.69
BEFS-Enabled	5.34	1.79	3.47
BESAFE-Enable	8.60	3.27	5.19

4.5.2.6 Improved Performance Through Parallel Attempts

As shown in the previous section, BEFS introduces a latency on the default TLS connection establishment, but only if the server does not support FS-ciphersuites. To minimise this latency, the client can implement BEFS attempts in parallel instead of consecutively. That is, the client sends two CHs, one with default ciphersuites and the second with FS-ciphersuites, in parallel to the server. For each TLS session establishment, the client waits for all the CH attempts' responses to return. If there is a valid response to the FS-ciphersuites attempt, the client proceeds with the FS-ciphersuites response. Otherwise, if the FS-ciphersuites attempt has failed, the client proceeds with the default ciphersuite response. The same applies to BESAFE but the client sends three handshakes and first checks the FS+AE-ciphersuites, then the FS-ciphersuites attempts' response, before deciding to proceed with default ciphersuites. Sending multiple handshakes with different configurations aims to improve the latency incurred on the client (if a fallback is needed). However, we acknowledge that this may add load to the server.

4.6 Limitations

While there are several reasons that can cause the observed results (detailed in Section 8.1), we cannot firmly specify them. This is because we do not perform servers' fingerprinting apart from what we learn from the Censys engine about the device types (network devices vs. ordinary servers). Additionally, we do not conduct a user study to ask the affected server administrators why they have made this choice. However, our study is mainly concerned with identifying whether there are

servers that support FS but select non-FS ciphersuites. We are confident about the phenomenon's existence, but the exact reasons behind it are left to the speculations that we list in Section 8.1. Finally, our analysis of ciphersuites is at the level of algorithm names to identify FS vs. non-FS, and AE vs. non-AE. We do not check further key properties such as length or randomness.

4.7 Conclusion

In this chapter, we analysed the state of FS on over 10 million servers on the Internet. Using modern TLS client handshake algorithms, our results show 5.37% of top domains, 7.51% of random domains, and 26.16% of random IPs, do not select FS key exchange algorithms. Surprisingly, we found that 39.20% of the top domains, 24.40% of the random domains, and 14.46% of the random IPs that do not select FS, do support FS. Despite the reasons behind this phenomenon, and whether it be due to server administrators' accidental or deliberate choice, the existence of servers that select non-FS, nevertheless support it, sheds light on the lack of negotiation transparency on today's real world protocols. We introduced the discriminatory attacker model, which exploits lack of transparency. We then discussed possible paths towards FS. Finally, we showed that a best effort approach can add a value over the "all or nothing" approach and can increase FS, or FS and AE adoption in misconfigured servers.

5

Does "www." Mean Better Transport Layer Security?

Contents

5.1 Introduction 73
5.2 Dataset 75
5.2.1 Top-Domains 76
5.2.2 Random-Domains 76
5.3 Methodology 76
5.3.1 Data Collection 76
5.3.2 Data Analysis 78
5.4 Results 81
5.4.1 Responding Servers 81
5.4.2 The Difference is in the Detail 82
5.4.3 When "`www.`" Means Better TLS Security 85
5.4.4 Relationship to HTTPS Redirection 87
5.5 Limitations 88
5.6 Conclusion 88

5.1 Introduction

The `www.` (world wide web) prefix has become a de facto naming standard for domains running websites. Plain-domains (those without the `www.` prefix, e.g. `example.com`) are usually treated as synonyms for their equivalent www-domains

(those that are prefixed with `www.`, e.g. `www.example.com`).

As a concrete example, from an application development perspective, recently, in August 2018, Google's Chrome[1] version 69.0.3497.81 decided to hide the `www` and `m` (`m` stands for mobile) subdomains from the steady state URL in Chrome's address bar by default [119]. Google describes these subdomains as "trivial" in Chrome's custom settings, where users can enable displaying these subdomains through the following Uniform Resource Identifier (URI): "`chrome://flags/#omnibox-ui-hide-steady state-url-scheme-and-subdomains`". This change by Google has received criticism and media attention from the security community, as shown in [119]–[121]. One of the reasons for not welcoming this change is that it can cause confusion [119]. For example, two addresses, one with a `www` subdomain and another without a `www` subdomain, can point to completely different websites [119]. One user reported Chrome's new behaviour of hiding the `www` and `m` subdomains as a bug [122]. In the same report, another user pointed out that the Safari mobile browser also hides the `www` subdomains from its address bar [122]. Subsequently, other users reported that Google search engine sometimes hides the `www` subdomains in search results and uses plain-domains format [123]. However, we are unable to reproduce the issue (bug) reported in [122] regarding Chrome's new behaviour since Chrome does not provide an archive for old versions. Nevertheless, the reported bug includes several confirming responses [122], in addition to media reports, such as [119]–[121], which provide sufficient evidence that the issue has existed in that particular version, either in the desktop, or the mobile version, or both. Our test of Chrome's latest version shows that subsequent versions (version 73.0.3683.86 at the time of writing this chapter in March 2019) have reverted this change from a default to a custom setting.

From a research perspective, in particular, on Internet measurement studies which are concerned with examining domains' TLS security configurations and certificates, we observe different treatments for the examined domain names. A considerable number of these studies make use of Alexa's top domains list [103]. Alexa represents domains as plain-domains, except for a small percentage (around

[1] As a shorthand, we use the term Chrome for the rest of the chapter to refer to Google's Chrome.

4.99%) of domains that include subdomains including `www` subdomains. Some studies, such as [124], add the `www` subdomain to the examined domains, and report the results based on TLS handshakes with www-domains. Other studies, such as [58], first try to establish a TLS handshake with the examined domain "as is", and if the handshake with the domain "as is" has failed, they make a second handshake with the domain's equivalent www-domain, and report the results. The third line of Internet measurement studies such as [70], [125], do not mention adding any `www` subdomains to the examined domains, hence, we assume that they treat the list's domains (e.g. the Alexa list in these studies) as is. However, it remains unclear whether plain-domains differ from their equivalent www-domains in terms of TLS configurations and certificates. If one type of domains, e.g. www-domains, provides better TLS security configurations than their equivalent plain-domains, or vice versa, to what extent does taking one approach (for domain treatment) over another affects the overall Internet measurement studies' results, particularly in regards to the domains' adoption of TLS security configurations, e.g. protocol versions, or the domains' certificates status results.

In this chapter, we examine consistency through the following question: *Do plain-domains and their equivalent www-domains differ in TLS security configurations and certificates? If so, to what extent?*

5.2 Dataset

Our study includes two datasets: top-domains, which contains 829,873 distinct most visited domains globally, and random-domains, which contains 992,422 distinct random domains. Our scope is limited to gTLDs, (e.g. `com` and `net` TLDs), and does not include "multi-level" TLDs such as ccTLDs, e.g. "`ac.uk`". This is to avoid the complexity of distinguishing domains that have subdomains from domains that have ccTLDs, which is somewhat difficult to achieve with 100% accuracy. In what follows, we describe how we build and preprocess each dataset.

5.2.1 Top-Domains

The top-domains dataset is derived from Alexa's top one million most visited domains globally [103], retrieved on December 11, 2018. Since our study targets plain-domains initially, from the one million domains, we extract the domains that are classified as plain-domains and as main-domains (for definitions of plain-domains and main-domains, see the list of abbreviations at the beginning of this thesis). We end up with 829,873 distinct top domains. To create their equivalent www-domains, we make a replica of the extracted plain-domains, then we prefix each domain with "`www.`".

5.2.2 Random-Domains

Our random-domains dataset initial size is one million domains, extracted from a large dataset of 54,063,220 domains that have successfully completed TLS handshakes in Amann et al. [75]. The domains in [75] are collected from multiple sources including other lists and previous work. We preprocess our random-domains dataset in two steps: first, to maintain consistency with the top-domains dataset format, from Amann et al.'s list [75], we extract one million random domains that are classified as main-domains and as plain-domains. Second, from the just extracted domains, we exclude the domains that already exist in the top-domains dataset, to avoid repetition. We finish with 992,422 distinct random domains. Finally, to create their equivalent www-domains, we create a replica of the dataset, then we prefix each domain with "`www.`".

5.3 Methodology

5.3.1 Data Collection

To collect data, we run a TLS client that takes the domain names from our datasets as input, performs a TLS handshake with each domain, and outputs the handshake's response data. For each dataset, namely the top-domains and random-domains datasets, the client performs TLS handshakes with plain-domains and their equivalent www-domains concurrently, using two separate TLS client instances. We

perform two types of TLS handshakes per domain: one utilising TLS 1.2 client configurations (for the ciphersuites list, see the pre-TLS 1.3 default ciphersuites in Table A.1 in Appendix A.3), and the second utilising TLS 1.3 client configurations (for the ciphersuites list, see the TLS 1.3 in addition to the pre-TLS 1.3 default ciphersuites in Table A.1 in Appendix A.3). Our clients' configurations (mainly, the supported TLS versions and ciphersuites) are based on Chrome's latest version configurations. Note that, similar to the Chrome's configurations, TLS 1.3 clients also support pre-TLS 1.3 configurations for backward compatibility. We choose to base our clients' TLS configurations on Chrome because it is the most representative TLS client on the Internet today. At the time of writing this chapter (February 2019), Chrome's usage is 79.7% [107], and in another source, Chrome's market share in February 2019 is 71.58% [108][2]. To increase our confidence in the obtained results, we run the above described experiment twice. The first experiment ran from the 11th to 12th December 2018, and the second ran from the 5th to 6th March 2019.

The TLS handshakes' responses, which include the servers' TLS versions, ciphersuites, and certificates, are first stored in json format, parsed, then stored and analysed using MySQL database and queries. In terms of counting the servers' responses, we follow a best effort approach. That is, we count the responding domains but we do not investigate the reasons for the non-responding ones. However, we eliminate the possibility of servers' rate limit[3] as a reason for the non-responding domains, as our client performs only two handshakes (one supports up to TLS 1.2, and the second supports up to TLS 1.3) per domain type (plain-domains and www-domains) per dataset (top-domains and random-domains). We also eliminate the time gap[4] as a reason for the non-responding domains, as our clients' handshakes with both plain-domains and their equivalent www-domains are performed concurrently. In our analysis, we only consider the domains that responded to both plain-domains and their equivalent www-domains handshakes.

[2]Check the footnote in Section 4.3.1 for some notes related to these figures.

[3]Rate limit is a technique used to mitigate DoS attacks by specifying a threshold for the incoming requests by a particular source address.

[4]By time gap we refer to a situation where a domain can be alive in one handshake (e.g. with plain-domain), but goes down in the second handshake (e.g. with www-domain).

In terms of the TLS client that performs the TLS handshakes, our TLS client is based on tls-scan, an open source fast TLS scanner capable of performing concurrent TLS connections [106]. We then customise the tls-scan to utilise OpenSSL 1.1.0g TLS library for our TLS 1.2 client and OpenSSL 1.1.1a for our TLS 1.3 client. We customise our clients' offered TLS versions and ciphersuites to be equivalent to those offered by Chrome's latest version for our TLS 1.3 client, and Chrome's latest version pre-TLS 1.3 configurations for our TLS 1.2 client. The SNI extension is included by default. We set the concurrency argument to 50 and 25 concurrent connections (these vary between the first and second experiment), and the timeout argument is set to 5 seconds. We run the experiments at the University of Oxford, on computers equipped with 1000 Mbps wired Ethernet cards.

After the TLS scan is complete, to better understand the results, we conduct an HTTPS redirection scan. We build a multi-threaded redirection scanner based on the Python's Requests library, which provides HTTPS redirection data [126]. The scanner takes as input the domains that have provided responses for both plain-domains and their equivalent www-domains in the TLS 1.3 handshakes in the second TLS scan experiment. The scanner then performs an HTTPS request with each domain without certificate validation, and collects the HTTPS redirection chain from the initial until the final URL for each request. We then store and analyse the data using the MySQL database and queries. The redirection scan ran from the 15^{th} to 17^{th} March 2019 at the University of Oxford. The short time span between the last TLS scan experiment and the redirection scan (around 8 days) allows us to correlate the redirection behaviour to the TLS configurations and certificate behaviour with high confidence.

5.3.2 Data Analysis

To answer the first part of our research question (*do plain-domains and their equivalent www-domains differ in TLS security configurations and certificates?*), we count the number of domains that:

1. Responded to both plain-domains and their equivalent www-domains TLS handshakes.

2. Provide different values for one or more of the following TLS configurations:

    - TLS versions.
    - Ciphersuites.
    - Versions *and* ciphersuites.
    - Versions *or* ciphersuites.
    - Leaf certificate Subject Key Identifiers (SKI), which is an X509 extension that provides the means for identifying certificates with a particular public key [127].
    - Leaf certificate fingerprints (SHA-1), which provide the means for identifying certificates in general (including all the certificate's fields).
    - IPv4, although not directly a TLS security configuration, we include it as a useful general result.

To answer the second part of our research question (*If so, to what extent?*), we first define some weakness indicators denoted by *weak*. Then, we analyse plain-domains and their equivalent www-domains against *weak*. In what follows, we list these weakness indicators, and define when they are "satisfied" by domains. We use the term satisfied to denote true or exists. That is, a domain satisfying a weakness indicator is indeed weak (or weaker):

1. v<TLS 1.3: is satisfied by domains that select TLS versions less than TLS 1.3 (the latest version).

2. v<TLS 1.2: is satisfied by domains that select TLS versions less than TLS 1.2. Versions less than TLS 1.2 are officially weak and should not be used today.

3. non-FS: is satisfied by domains that select non-FS-ciphersuites (despite the AE or any other properties). We define non-FS-ciphersuites[5] as those that do not support the ECDHE, and are also not negotiated with version TLS 1.3 since TLS 1.3 enforces FS by design in a separate extension. Non-FS ciphersuites provide fewer security guarantees than FS ciphersuites.

4. non-FS+non-AE: is satisfied by domains that select non-FS+non-AE-ciphersuites, i.e. neither provide FS nor AE. We define non-FS+non-AE-ciphersuites as those that do not support ECDHE, are not negotiated with version TLS 1.3, do not support the ChaCha20 symmetric encryption, and do not support the GCM symmetric encryption mode. This indicator is not satisfied by domains that select FS+AE, i.e. those that either support ECDHE or negotiated with version TLS 1.3, and support either the ChaCha20 symmetric cipher or the GCM mode. Non-FS+non-AE-ciphersuites provide fewer security guarantees than those that provide both FS and AE (FS+AE).

5. Exp. Cert: is satisfied by domains that provide expired certificates. The expiration check is performed by the tls-scan client against the scan date [106].

6. Invalid Cert: is satisfied by domains that provide invalid certificates. The certificate validation is performed by the tls-scan client using the "SSL_-get_verify_result" function in the OpenSSL library against our updated Ubuntu 18.04 certificates store. It validates the certificate's signature, trust chain, and other verification steps specified in [128].

7. Key<2048: is satisfied by domains that use RSA certificates with a key length of less than 2048-bit. The minimum recommended RSA key length today is 2048-bit [129].

Then, we count the domains' responses under the following conditions, where *weak* denotes a weakness indicator, *plain* denotes plain-domains that have responses

[5]Our definitions for FS and AE are based on Chrome's configurations.

for their equivalent www-domains, and *www* denotes www-domains that have responses for their equivalent plain-domains:

1. $weak_{plain}$: *weak* is satisfied by *plain*.
2. $weak_{plain} \wedge \neg weak_{www}$: *weak* is satisfied by *plain* and *weak* is *not* satisfied by *www*.
3. $weak_{www}$: *weak* is satisfied by *www*.
4. $weak_{www} \wedge \neg weak_{plain}$: *weak* is satisfied by *www* and *weak* is *not* satisfied by *plain*.

5.4 Results

As described earlier in the methodology, we have conducted two experiments over a 3 month time span, to increase our confidence in the obtained results. In this section, we only report the results from the latter experiment (March 2019). However, they are consistent with the former experiment (December 2018).

5.4.1 Responding Servers

Table 5.1 summarises the number of successful TLS handshake responses from plain-domains and their equivalent www-domains in the top-domains and random-domains datasets, in both types of client handshakes, TLS 1.2 and TLS 1.3. Responses from top domains are higher than those from random domains. This is mostly due to the fact that the TLS adoption rate in top domains is higher than that in random domains. Furthermore, the top-domains dataset is built from a recent Alexa list which contains active domains, while the random-domains dataset is built from a less recent list from Amann et al. [75], which may contain many inactive domains. We notice that in the top-domains dataset, www-domains responses are higher than plain-domains responses by 2.3%. This means that top domains tend to have more active www-domains than plain-domains. On the other hand, in the random-domains dataset, both plain-domains and www-domains responses are nearly equal, with slightly more responses from plain-domains than www-domains.

Table 5.1: Responding servers to our TLS clients' handshakes.

Client	Dataset	Size	Type	Responses
TLS 1.2	top-domains	829,873	plain	691,200 (83.29%)
			www	710,509 (85.62%)
	random-domains	992,422	plain	623,869 (62.86%)
			www	620,884 (62.56%)
TLS 1.3	top-domains	829,873	plain	691,145 (83.28%)
			www	710,496 (85.62%)
	random-domains	992,422	plain	622,904 (62.77%)
			www	620,062 (62.48%)

5.4.2 The Difference is in the Detail

As depicted in Table 5.2, our results show that the client type, i.e. TLS 1.2 vs. TLS 1.3, does not have a noticeable effect in terms of the exhibited differences in TLS configurations and certificates between plain-domains and their equivalent www-domains. However, in the "version" difference, TLS 1.3 client shows higher percentages (1.35% and 0.54%) than TLS 1.2 client (0.22% and 0.12%) in the top-domains and random-domains datasets, respectively (see Listing B.3 in Appendix B.2 for the MySQL query to find the version differences). This can be explained by the fact that TLS 1.3 was standardised in 2018, which suggests that domain administrators may have updated the TLS version of one domain, e.g. the plain-domain, but not its equivalent, e.g. the www-domain, or vice versa. In terms of other TLS configurations and certificate differences at both clients TLS 1.3 and TLS 1.2, in general, top domains show higher percentages of differences in the TLS configurations and certificates between plain-domains and their equivalent www-domains than random domains. In general, the most significant differences are in the leaf certificate fingerprints ("Cert"), certificate "SKI", and in the selected "version or ciphersuite". For example, as illustrated in Table 5.2, in the top-domains dataset, more than 11% of plain-domains provide a different certificate fingerprint than their equivalent www-domains. Over 10% of plain-domains provide a different SKI than their equivalent www-domains. Over 3.4% select a different version or ciphersuite

than their equivalent www-domains. On the other hand, the random-domains dataset shows lower percentages of differences (see Table 5.2).

Table 5.2: Differences between plain-domains and their equivalent www-domain against some TLS configurations. The "Responses" column represents the number of domains that responded to *both* plain-domain and their equivalent www-domain TLS handshakes.

Client	Dataset	Responses	Different						
			Version	Ciphersuite	Version ∧ Ciphersuite	Version ∨ Ciphersuite	SKI	Cert.	IPv4
TLS 1.2	top-domains	678,337	1472 (0.22%)	23,600 (3.48%)	1437 (0.21%)	23,635 (3.48%)	74,433 (10.97%)	75,359 (11.11%)	125,767 (18.54%)
	random-domains	614,184	748 (0.12%)	7883 (1.28%)	737 (0.12%)	7894 (1.29%)	52,465 (8.54%)	52,844 (8.60%)	57,037 (9.29%)
TLS 1.3	top-domains	678,214	9187 (1.35%)	24,819 (3.66%)	9151 (1.35%)	24,855 (3.66%)	74,375 (10.97%)	75,311 (11.10%)	125,861 (18.56%)
	random-domains	612,839	3292 (0.54%)	8386 (1.37%)	3282 (0.54%)	8396 (1.37%)	52,318 (8.54%)	52,718 (8.60%)	57,063 (9.31%)

5.4.3 When "www." Means Better TLS Security

For further analysis, we compare plain-domains and their equivalent www-domains against several defined weakness indicators denoted by *weak* (see Section 5.3.2 for further details about the methodology). In this section, we limit our analysis to TLS 1.3 client handshake results as TLS 1.3 handshake is more representative of an updated TLS client today, such as web browsers (recall that in our TLS scan experiments, we perform two types of handshakes for each domain, one utilising TLS 1.3 configurations, and the second utilising TLS 1.2 configurations). As shown in Table 5.3, it is always the case that there are fewer www-domains that satisfy weakness indicators compared to plain-domains that do so (see Table 5.3, under the "Type/Condition" column, compare the results of row "$weak_{plain}$" to those of "$weak_{www}$" in each dataset). Also, it is always the case that the number of plain-domains that satisfy a weakness indicator while their equivalent www-domains do not, is higher than the number of www-domains that satisfy a weakness indicator while their equivalent plain-domains do not (see Table 5.3, under the "Type/Condition" column, compare the results of row "$weak_{plain} \wedge \neg weak_{www}$" to those of "$weak_{www} \wedge \neg weak_{plain}$" in each dataset). For example, as depicted in Table 5.3, in the top-domains dataset, of the 3.62% plain-domains that provide expired certificates, there are 10.24% that provide non-expired certificates by their equivalent www-domains (see Listing B.4 in Appendix B.2 for the MySQL query to find this result). However, in the same dataset (top-domains), of the 3.42% www-domains that provide expired certificates, there are only 5.18% that provide non-expired certificates by their equivalent plain-domains. The same trend appears in all of the weakness indicators we study (see Table 5.3), which suggests that www-domains tend to have better TLS security configurations and certificates than their equivalent plain-domains.

Table 5.3: Breakdown of some weakness indicators *weak* in plain-domains and their equivalent www-domains based on the TLS 1.3 client handshake responses. The indentation in the "Type/Condition" column indicates that the percentages of the indented row's results are computed over the previous row's results. The percentages of the non-indented row's results are computed over the "Responses" values. Recall: *weak* denotes weakness indicator; $weak_{plain}$ denotes *weak* is satisfied by *plain*; $weak_{plain} \wedge \neg weak_{www}$ denotes *weak* is satisfied by *plain* and *weak* is not satisfied by *www*; $weak_{www}$ denotes *weak* is satisfied by *www*; and $weak_{www} \wedge \neg weak_{plain}$ denotes *weak* is satisfied by *www* and *weak* is not satisfied by *plain*.

Dataset	Responses	Type/Condition	Weakness Indicator (*weak*)						
			v.<TLS 1.3	v.<TLS 1.2	non-FS	non-FS+non-AE	Exp. Cert.	Invalid Cert.	Key<2048
top-domains	678,214	$weak_{plain}$	542,267 (79.96%)	15,731 (2.32%)	27,777 (4.10%)	17,368 (2.56%)	24,521 (3.62%)	55,932 (8.25%)	6306 (0.93%)
		$weak_{plain} \wedge \neg weak_{www}$	5280 (0.97%)	1179 (7.49%)	3151 (11.34%)	1907 (10.98%)	2510 (10.24%)	5281 (9.44%)	278 (4.41%)
		$weak_{www}$	539,543 (79.55%)	14,846 (2.19%)	25,184 (3.71%)	15,710 (2.32%)	23,214 (3.42%)	52,773 (7.78%)	6116 (0.90%)
		$weak_{www} \wedge \neg weak_{plain}$	2556 (0.47%)	294 (1.98%)	558 (2.22%)	302 (1.92%)	1203 (5.18%)	2122 (4.02%)	135 (2.21%)
random-domains	612,839	$weak_{plain}$	545,924 (89.08%)	27,774 (4.53%)	38,653 (6.31%)	30,621 (5.00%)	55,734 (9.09%)	107,521 (17.54%)	18,781 (3.06%)
		$weak_{plain} \wedge \neg weak_{www}$	1667 (0.31%)	536 (1.93%)	770 (1.99%)	496 (1.62%)	911 (1.63%)	1684 (1.57%)	191 (1.02%)
		$weak_{www}$	545,171 (88.96%)	27,448 (4.48%)	38,152 (6.23%)	30,280 (4.94%)	55,350 (9.03%)	106,663 (17.40%)	18,628 (3.04%)
		$weak_{www} \wedge \neg weak_{plain}$	914 (0.17)%	210 (0.77%)	269 (0.71%)	209 (0.69%)	527 (0.95%)	826 (0.77%)	46 (0.25%)

Table 5.4: Summary of the HTTPS redirection scan results.

Dataset	Size	Type	Responses	Redirection
top-domains	678,214	plain	661,596 (97.55%)	225,839 (34.14%)
		www	662,474 (97.68%)	155,921 (23.54%)
random-domains	612,839	plain	587,850 (95.92%)	78,449 (13.35%)
		www	588,380 (96.01%)	59,611 (10.13%)

5.4.4 Relationship to HTTPS Redirection

To better understand the reasons behind the observed phenomenon, we analyse the HTTPS redirection behaviour of those domains. We input 678,214 top domains and 612,839 random domains that have provided responses for both plain-domains and their equivalent www-domains in the TLS 1.3 scan experiment (the latest scan in March 2019). As depicted in Table 5.4, from the top-domains dataset, 97.55% of plain-domains and 97.68% of www-domains responded to our redirection scan. From the random-domains dataset, 95.92% of plain-domains, and 96.01% of www-domains have responded to our redirection scan. Of the responding domains to our redirection scan, in the top-domains dataset, 34.14% of HTTPS plain-domains redirected to (i.e. land on) their equivalent HTTPS www-domains, while 23.54% of HTTPS www-domains redirected to their equivalent HTTPS plain-domains. In the random-domains dataset, 13.35% of HTTPS plain-domains redirected to their equivalent HTTPS www-domains, while 10.13% of HTTPS www-domains redirected to their equivalent HTTPS plain-domains. From these results, we conclude that plain-domains tend to redirect to their equivalent www-domains more than www-domains that redirect to their equivalent plain-domains, and HTTPS redirection from plain-domains to their equivalent www-domains is more utilised in top-domains than in random-domains.

We conduct further analysis on the set of domains that showed at least one weakness indicator in plain-domains, but not in their equivalent www-domains in the TLS scan (see Table 5.3, domains in row "$weak_{plain} \wedge \neg weak_{www}$" in both datasets). In the top-domains dataset, out of 11,893 top domains that fall into this set and

responded to our plain-domains redirection scan, 6345 (53.35%) HTTPS plain-domains redirected to (land on) their equivalent HTTPS www-domains. Of those, 1568 (24.71%) have one or more HTTP (plaintext, unencrypted and unauthenticated) intermediate URLs in the redirection chain. In the random-domains dataset, out of the 3369 random domains that fall into that set and responded to our redirection scan, 664 (19.71%) HTTPS plain-domains redirected to their equivalent HTTPS www-domains. Of those, there are 252 (37.95%) HTTP intermediate URLs. The redirection from HTTPS plain-domains to HTTPS www-domains in these domains that show one or more weakness indicators in plain-domains, but not in www-domains, is higher than the overall redirection rate that is presented in Table 5.4 earlier. Apparently, in this set of domains, domain administrators pay more attention to www-domains security as HTTPS plain-domains are redirected to their equivalent HTTPS www-domains. However, secure redirection should maintain secure TLS configurations and certificates at every point in the redirection chain.

5.5 Limitations

First, it should be noted that our analysis is limited to TLS configurations and certificates. We do not check the content of the pages. Second, while using popular top domain lists such as the Alexa top 1 million is common in Internet measurement studies, most of these lists have some limitations. For example, Scheitle et al. reported low stability in lower ranked domains in Alexa's list [130].

5.6 Conclusion

In this chapter, we presented the results of an experiment that aims to examine consistency by exploring whether plain-domains and their equivalent www-domains differ in TLS security configurations and certificates and if so, to what extent. Our results inform the Internet measurement-based research community, domains (servers) administrators, developers, and users alike. First, we provided evidence that there is a difference between plain-domains and their equivalent www-domains

in terms of TLS security configurations and certificates in a non-trivial number of cases. The difference is more notable in top domains than in random domains. Second, by defining some weakness indicators and examining when plain-domains and their equivalent www-domains satisfy these indicators, we showed that www-domains tend to have better TLS security than their equivalent plain-domains. Third, we showed that HTTPS redirection from plain-domains to their equivalent www-domains is widely utilised, especially in the top domains, and more significantly in the set of domains that show one or more weakness indicator in plain-domains but not in their equivalent www-domains ($weak_{plain} \wedge \neg weak_{www}$). In the latter case, users see the final www-domains (URL) with strong TLS security configurations and certificates, but in fact, the HTTPS request has actually passed through plain-domains which have less secure TLS configurations and certificates at previous points in the redirection chain. Even worse, many intermediate URLs in these redirection chains are plaintext HTTP. This introduces a weak link in the system, which may be dangerous for users.

6

Exploring HTTPS Security Inconsistencies: a Cross-Regional Perspective

Contents

6.1 Introduction 92
6.2 Dataset 92
6.3 Methodology 93
6.3.1 Setup 93
6.3.2 Data Collection 94
6.3.3 URLs Security Weakness Indicators 96
6.3.4 Security Headers Weakness Indicators 97
6.3.5 TLS Weakness Indicators 98
6.4 Results 99
6.4.1 Responses 99
6.4.2 HTTPS Security Inconsistencies 100
6.5 Attack Scenarios 116
6.5.1 Region-Confusion Attack 116
6.5.2 Discrimination Attack 117
6.6 Recommendations 118
6.7 Limitations 120
6.8 Conclusion 121

6.1 Introduction

Since its early versions, TLS, especially in the context of HTTPS, has been under scrutiny from the security research community. This, in turn, revealed a large number of vulnerabilities. As a result, new TLS versions, ciphersuites, HTTP security headers, and guidelines, have been introduced to achieve the desired security of HTTPS. With the existence of multiple TLS versions, ciphersuites, and HTTP security headers, it is important to continuously assess the real world deployment of HTTPS security configurations. Several studies have assessed some aspects of HTTPS deployment in the real world, such as [70], [75], [125]. However, the quest for assessing the consistency of servers' HTTPS security guarantees in response to requests from different geographic regions for the same domain remains unexplored.

In this chapter, we examine negotiation consistency through the following question: *If two or more identical HTTPS clients, located at different regions, make an HTTPS request to the same domain, at approximately the same time, will they receive the same HTTPS security guarantees in response?*

6.2 Dataset

Our dataset consists of the top 250,000 most visited domains on the Internet, obtained from the Tranco list [131], [132], a configurable research oriented top domains list, hardened against manipulation attacks. We first extract the top 1 million list, then we extract the top 250,000 from it. We choose to use the "standard" configurations of the list, which aggregates the ranks of main-domains ("pay-level"[1] domains in the list's terminology), which do not contain subdomains, from four widely used top domains lists: Alexa [103], Umbrella [133], Majestic [134], and Quantcast [135], over the past 30 days. We retrieved the list on the 10$^{\text{th}}$ of April, 2019. In our top 250,000 domains list, the majority of domains have gTLDs in the form of "domain.TLD". However, our dataset contains 9.19% domains with "multi-level" TLDs, e.g. ccTLDs, which we identify by searching for domains that

[1] According to the list's authors, "pay-level" domain refers to "a domain name that a consumer or business can directly register." [131]

have more than one dot ".". We also check our list against Google's Safe Browsing database by setting up a local server using Google's open source Safe Browsing Go implementation [136], which receives updates frequently. We query Google's Safe Browsing database using the API v4 threatMatches using an HTTP POST request to the local server. We made the query on the 27th of June, 2019. We found a total of 250 malicious distinct domains in our dataset. The threat types, based on Google classifier, are: Malware (25 domains), unwanted_software (29), and social_engineering (197) (note: there is one domain that appeared in two threat types). In our results in the coming sections, we exclude known malicious domains.

6.3 Methodology

6.3.1 Setup

To conduct our experiment, we create five machines (clients), configured to be identical from all aspects (hardware, software, and configurations), physically located at five different geographic locations, rented from a cloud provider (Amazon Elastic Compute Cloud (EC2) [137]). We considered choosing locations that are geographically, economically, and politically distant. Therefore, we select a country from 5 different continents. The locations we choose are (continent, country): Australia, Australia (AU); South America, Brazil (BR); Asia, India (IN); Europe, United Kingdom (UK); North America, United States (US). Note that our choices are bound by the region's availability and requirements at Amazon EC2. For example, China would be an interesting region to have, however, it has different requirements such as a valid business license from the Chinese government, which we do not have. We first launch the first client in the first region using Ubuntu 18.04 Operating System (OS) and OpenSSL 1.1.0g and an updated Certificate Authority (CA) store, and we install and configure the software we need in our experiment including the scanners and their libraries. We then create an image of the first client's disk. After that, we create the remaining four clients with the same hardware specifications, running a copy of the first client's image.

6.3.2 Data Collection

To collect data, we run two scans: a redirection scan followed by a TLS scan. The redirection scan collects data at the application layer such as the servers' response headers and the URL redirection chains. The TLS scan collects data at the transport layer such as the servers' selected protocol versions, ciphersuites, and TLS certificates. To run the scans, we use a mixture of existing open source and those developed in house tools.

First, for the redirection scan, we develop our own redirection scanner using the Python's Requests 2.21.0 library [126], which allows us to automate sending HTTPS requests, collect response URLs, redirection chains (if exist), and the full content of response headers. Our scanner utilises a Transport Adapter which we configure with Chrome's latest version pre-TLS 1.3 ciphersuites (for the ciphersuites list, see the pre-TLS 1.3 default ciphersuites in Table A.1 in Appendix A.3). The ciphersuites list at this stage is not sensitive as we do not collect transport layer data at the redirection scan. However, the default ciphersuites list that is shipped with the Requests library is much longer than Chrome's list. Therefore, using Chrome's ciphersuites list has a performance advantage and can maximise the response rate as it contains the most widely supported ciphersuites. We disable the certificate validation as we use the redirection scanner to collect application layer data regardless of the certificate status.

For the TLS scan, we use the tls-scan tool, an open source fast TLS scanner [106]. We customise the tls-scan client to utilise the OpenSSL 1.1.1g which supports versions from TLS 1.0 to TLS 1.3, and to offer Google Chrome's latest version[2] ciphersuites, which adds support for three new TLS 1.3 ciphersuites besides Chrome's pre-TLS 1.3 ciphersuites (for the ciphersuites list, see the TLS 1.3 and pre-TLS 1.3 default ciphersuites in Table A.1 in Appendix A.3). Note that, similar to the Chrome browser, our TLS 1.3 clients also support pre-TLS 1.3 ciphersuites for backward compatibility.

[2]At the time of writing this chapter (April 2019), version 74.0.3729.108.

After setting up the experiment's clients, from each region, on the same day, we first run the redirection scan which gathers application layer data. The redirection scanner takes the domain names from our dataset as input. After that, for each domain, the scanner conducts an HTTPS GET request, collects the final response URL, and the URL redirection chain (if this exists) starting from the requested URL until the one before the final response URL. In addition, we collect the response status codes and the full content of the response headers. After finishing the redirection scan, in each region, for each request made, we extract the domain (including the TLD and subdomains if these exist) of the HTTPS final response, to use it as an input for the TLS scan phase. Note the difference in the input domains between the redirection scan and the TLS scan. The redirection scanner makes requests for the same domains in all regions. For example, if the dataset contains `example.com`, clients in all regions make requests to `example.com`. However, since our redirection scanner follows redirections, the response domain in the redirection scan may differ based on the region, e.g. the US-based client receives `us.example.com` while the UK-based client receives `uk.example.com`. The TLS scanner takes the final response distinct domains as input, and does not perform redirection as these are the final response domains based on the redirection scan results. The TLS scan results reflect the state of the final domain. We conduct both scans between the 11^{th} and 20^{th} April, 2019. However, requests for any set of domains from the five regions are sent on the same day. We scan the 250K domains in two stages: first we scan the top 10K domains. After we gain confidence about the existence of inconsistencies through initial analysis of the results, we proceed to scan the remaining 240K domains. Finally, after the data is collected, we combine both stages' data, load, and analyse them using MySQL database and queries. We describe the data analysis methodology in more detail next.

To analyse data, we define HTTPS weakness indicators based on well-known security guidelines and recommendations that are adopted by secure HTTPS clients and servers. For example, using HTTPS not HTTP, using security headers such as HSTS and CSP. In addition, we define some TLS weakness indicators based on

recommendations such as NIST's recommendations on using the latest TLS version and the secure ciphersuites that provide strong guarantees such as FS and AE [138]. We consider the security of servers' response from three vectors:

1. URLs security.
2. Security headers.
3. Transport Layer Security (TLS).

Then, to find inconsistent HTTPS cases, we count domains that satisfy a weakness indicator in ***some but not all*** of their responses to our clients in the five regions. We use the term satisfied to denote true or exists. That is, a domain satisfying a weakness indicator is indeed weak (or weaker). We now define the weakness indicators in each vector.

6.3.3 URLs Security Weakness Indicators

1. **plain-HTTP (final/intermediate) URL:** this weakness indicator is satisfied if the server's response URL is using the plain-HTTP protocol. Needless to say, plain-HTTP does not give the security guarantees of HTTPS. We analyse the server's response URL from two perspectives: the response's final (landing) URL, and the response's intermediate URL(s) that appear in the redirection chain, if any. Our clients' requests are initiated using the HTTPS protocol, but may receive plain-HTTP in intermediate or final responses, depending on the server's response.

2. **Incompatible (final/intermediate) domain:** this weakness indicator is satisfied if the server's response URL contains a domain that is different from the requested domain. By "incompatible" we mean unequal, different, or inconsistent. However, we avoid reusing the term "inconsistent" to avoid ambiguity as we already use the term "inconsistent" to describe unequal response for requests from the five regions with respect to the defined weakness indicators including the "incompatible (final/intermediate) domain" indicator.

Similar to the previous indicator, we analyse the server's response domain from two perspectives: the response's final (landing) URL, and the response's intermediate URL(s) that appear in the redirection chain, if any. To examine the requested domains against the final/intermediate response's domain, we extract the main-domain (also known as "pay-level" domain or base-domain) of the request and response domains using the tldextract Python library [104], which identifies the main-domains by maintaining an updated list of all known public TLDs obtained from Mozilla's public suffix list [105]. Then, we compare them using a case-insensitive equality check. If they are not equal, they are classified as incompatible. For the intermediate domains check, since a response may contain multiple intermediate URLs, this weakness indicator is satisfied if one or more intermediate domains are incompatible with the requested domain.

The compatibility check is implemented in Python. The results are then stored and queried using MySQL.

6.3.4 Security Headers Weakness Indicators

1. **No-HSTS:** this weakness indicator is satisfied if the Strict-Transport-Security header is absent from the server's response.

2. **No-CSP:** similar to the previous indicator but for the CSP header.

3. **No-"Secure" in Set-Cookie:** this indicator is satisfied if the Set-Cookie header is sent in the servers' responses to the five regions' client requests to a particular domain, and one or more cookie values in the Set-Cookie header do not contain the Secure attribute value.

The headers' presence check is implemented in Python by parsing the headers which are loaded as a dictionary of key/value objects, where the header name is the key. The results are then stored and queried using MySQL.

6.3.5 TLS Weakness Indicators

1. **Version <TLS 1.3:** this indicator is satisfied if the server selects a TLS version less than TLS 1.3. TLS 1.3 is the latest version of the TLS protocol. Since it was standardised in August 2018 and our scan is in April 2019, the resulting inconsistencies against TLS 1.3 can be influenced by its recency. For this reason, we also examine the "Version <TLS 1.2" weakness indicator, to have a balanced perspective.

2. **Version <TLS 1.2:** this indicator is satisfied if the server selects a TLS version less than TLS 1.2. Versions less than TLS 1.2 are officially weak and should not be used today. This indicator is negated if the server selects version TLS 1.2 or TLS 1.3.

3. **Non-FS:** this indicator is satisfied if the server selects a non-FS-ciphersuite. We define[3] non-FS-ciphersuites as those that do not support the ECDHE key-exchange, and also are not negotiated with version TLS 1.3 since TLS 1.3 enforces FS by design in separate extensions.

4. **Non-FS+Non-AE:** this indicator is satisfied if the server selects a non-FS and a non-AE ciphersuite (non-FS+non-AE-ciphersuite), i.e. neither provide FS nor AE. We define non-FS+non-AE ciphersuites as those that do not support ECDHE, are not negotiated with version TLS 1.3, do not support the ChaCha20 symmetric encryption, and do not support the GCM symmetric encryption mode. This indicator is negated if the server selects either an FS-ciphersuite, or AE-ciphersuite, or FS+AE-ciphersuite.

5. **Expired Certificate:** this indicator is satisfied if the server provides an expired certificate. The certificate expiration date is checked against the scan date using the tls-scan client [106].

[3]In this chapter, our ciphersuites definitions are based on Chrome's TLS configurations.

Table 6.1: Summary of the redirection scan responses. The percentages of the responses "(Any/Valid) HTTP(S) Final Responses" are computed over the 250,000 input domains. Each indentation in a row means the percentages of that row are computed from the previous indentation level.

Response Type	Status Code	Joint Responses
Any HTTP(S) Final Response	Any	187,902 (75.16%)
Any Consistent plain-HTTP Final Response	Any	10,320 (5.49%)
Any Consistent HTTPS Final Response	Any	176,936 (94.16%)
Any Consistent HTTPS with Redirections > 0	Any	86,123 (48.67%)
Any Consistent HTTPS with plain-HTTP Inter. Response	Any	12,198 (14.16%)
Valid HTTP(S) Final Response	200	163,235 (65.29%)
Valid Consistent plain-HTTP Final Response	200	9242 (5.66%)
Valid Consistent HTTPS Final Response	200	153,761 (94.2%)
Valid Consistent HTTPS with Redirections > 0	200	81,019 (52.69%)
Valid Consistent HTTPS with plain-HTTP Inter. Response	200	11,873 (14.65%)

6. **Invalid Host Name:** this indicator is satisfied if the server provides an invalid host name in the certificate. The validation is based on the tls-scan client using the "X509_check_host" function in the OpenSSL library [139]. It checks if the certificate's Subject Common Name (CN) or Subject Alternative Name (SAN) matches the specified host name.

6.4 Results

In this section, we first provide a general overview of the response data. Then, we provide the results of our inconsistency analysis against each weakness indicator, which we defined in Section 6.3.3. In this section, we are concerned about inconsistencies. That is, we quantify domains that satisfy a weakness indicator in ***some but not all*** of their responses to our clients' requests from the five regions. We do not report domains that consistently do not satisfy a weakness indicator in their responses to the clients in the five regions.

6.4.1 Responses

Table 6.1 provides a summary of the "joint" response data. A joint response is counted if we receive responses for requests from the five regions. We follow a best effort approach in data collections. That is, we do not investigate the reasons for failed responses in one or more regions, which can be for various reasons, and we

count as a joint fail. However, the failure rate affects the collected data size, but not the validity of the collected data. We compile the data twice: first, without considering the response status code, i.e. any code (see the first half of Table 6.1), which we prefix by "Any". Second, we consider only those responses that provide the "200 OK" status code, which indicates that the HTTP request has succeeded (see the second half of Table 6.1), which we prefix by "Valid". We then dissect the overall HTTP(S) responses in both categories, i.e. the (Any/Valid) categories, into two sub-categories: "(Any/Valid) Consistent plain-HTTP Final response", which denotes responses with the insecure plaintext HTTP protocol (without TLS), and "(Any/Valid) Consistent HTTPS Final Response", which denotes responses with the secure HTTPS protocol. From the "(Any/Valid) Consistent HTTPS Final Responses", we count those with "(Any/Valid) Consistent HTTPS with Redirections >0". From those, we count the redirections that contain one or more "(Any/Valid) Consistent HTTPS with plain-HTTP Intermediate Response". As depicted in Table 6.1, the "Joint Response" column shows the number of cases that received responses from the five regions. Finally, the percentages of the overall responses "(Any/Valid) HTTP(S) Final Responses" are computed over the dataset size (250,000). However, each indented row in Table 6.1 means that the percentages of that row are computed over the previous indentation level.

In the remaining sections, we base our analysis on the valid HTTP(S) and the valid HTTPS final responses (see the highlighted rows in the second half of Table 6.1).

6.4.2 HTTPS Security Inconsistencies

We now summarise the HTTPS security inconsistent cases that we find from three vectors:

1. URLs security.

2. Security headers.

3. Transport Layer Security (TLS).

Table 6.2: Summary of the URLs security inconsistencies. The "Inconsistent" column shows the overall inconsistent cases against the weakness indicator. The percentages are computed over the 163,235 valid joint HTTP(S) responses.

Weakness Indicator	Inconsistent
plain-HTTP Final URL	232 (0.14%)
plain-HTTP Inter. URL	501 (0.31%)
Incompatible Final URL	280 (0.17%)
Incompatible Inter. URL	205 (0.13%)

Overall, the inconsistencies are not so prevalent phenomena, nevertheless, it is important to identify them. Experience shows that many serious security issues tend to be related to the least obvious phenomena. This is the first study to identify and report HTTPS security inconsistencies from a multi-regional perspective.

6.4.2.1 URLs Security

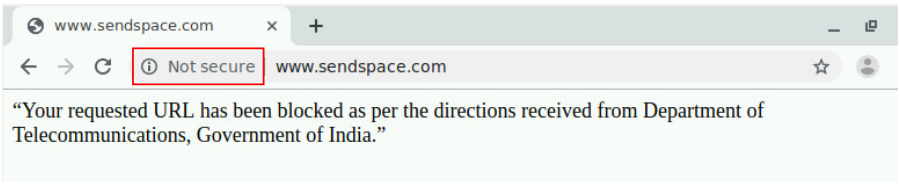
In this section, we consider the valid HTTP(S) final responses (the highlighted row labelled "Valid HTTP(S) Final Response" in Table 6.1) as we require both plain-HTTP as well as HTTPS responses to be analysed. The results of our inconsistency analysis of URLs security are summarised in Table 6.2.

First, regarding the **plain-HTTP final URL** indicator, we find 232 cases for domains that send their final responses in plain-HTTP URL to some but not all of the five regions (see Listing B.5 in Appendix B.3 for the MySQL query used to find this result). Needless to say, domains that send plain-HTTP responses are prone to impersonation attacks due to lack of authentication, data manipulation during transmission due to lack of integrity, and espionage, e.g. to steal users' credentials, due to a lack of secrecy (encryption). To understand the reasons for inconsistent plain-HTTP responses, we manually inspect the plain-HTTP final URLs in each region, within around two weeks after completing the scans. Through the Remote Desktop Protocol (RDP), we connect to the remote client, and from there we

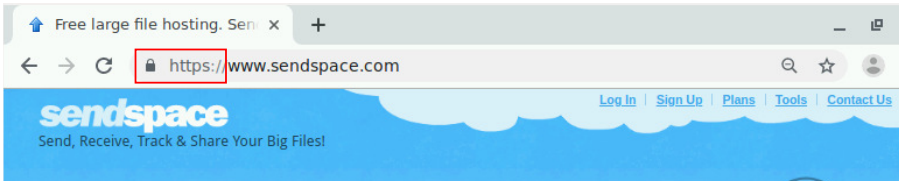
manually visit each website that resulted in plain-HTTP response, using the Firefox[4] web browser. We identify several reasons for inconsistent plain-HTTP responses. We find that blocking pages plays a considerable role in such inconsistencies, where these domains provide a valid plain-HTTP response (200 OK) status code, but display a plain-HTTP blocking page to inform the user that the requested domain (website) is not accessible in the client's region. See Figure 6.1 for `sendspace.com` case, and Figure 6.2 for `gannett.com` case. The latter is an American media company which owns the "US Today" newspaper. Out of the active websites at the time of the manual inspection, we identify blocking pages in (note: the total numbers provided here exclude domains that did not respond to the manual inspection): 19/80 (23.75%) in AU, 20/98 (20.41%) in BR, 42/132 (31.82%) in IN, 28/93 (30.11%) in the UK, and 4/71 (5.63%) in the US. Clearly, out of the five regions, IN has the highest percentage of blocking pages, followed by the UK, while the US has the lowest percentage. There are various reasons for regional blocking as also noted in [83], including, but not limited to, government orders (observed mostly in IN. See Figure 6.1 for example), GDPR compliance (mostly in the UK), business unavailability (e.g. a company does not offer its products to some regions).

Apart from regional blocking, there are indeed inconsistent cases where the same domain is deployed in plain-HTTP in some but not all regions. For example, the case of `match.com`, a high profile dating website, if visited from BR, the client is redirected to the plain-HTTP domain `br.match.com`, while all the other examined regions, such as the US, are provided with HTTPS. See Figure 6.3 for illustration. Interestingly, we revisited `match.com` from BR around four months after our initial observation. We find that it stopped redirecting to `br.match.com`, and when we manually visit `br.match.com` directly, we notice that it is using HTTPS, with a certificate starting from March 8, 2019, around one month before our scan. This indicates that the insecurity of `br.match.com` that we initially observed is due to insecure redirection,

[4]Although the scans are based on Chrome's configurations, we use Firefox in the manual inspection to view plain-HTTP pages to understand the reasons for plaint-HTTP final URLs. We used Firefox because it was the default browser installed in the remote machines. However, the browser choice at this task (viewing pages to identify if they are blocking pages or not) does not make a difference. Browsers differ at a more subtle level.

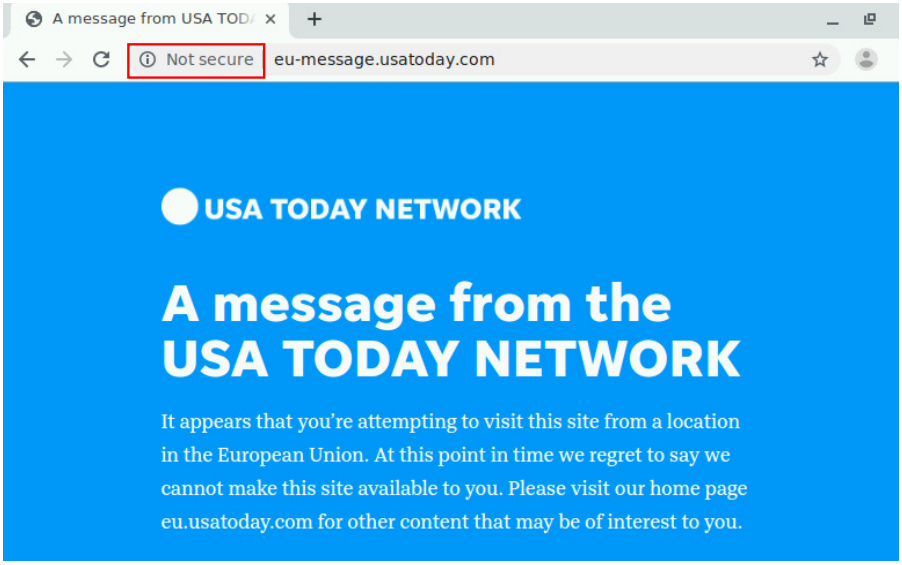


(a) `sendspace.com` over plain-HTTP in the IN due to blocking.

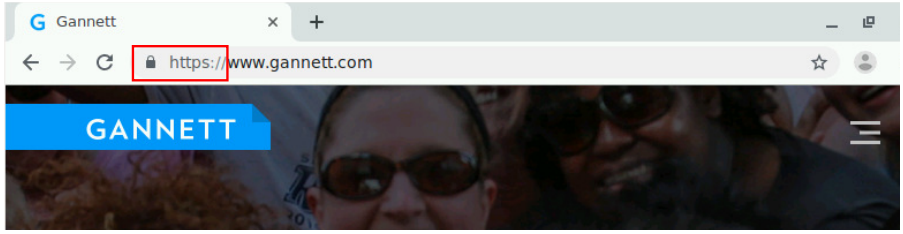


(b) `sendspace.com` over HTTPS in the US.

Figure 6.1: The case of `sendspace.com` in IN vs. the US.



(a) `gannett.com` over plain-HTTP in the UK due to blocking.



(b) `gannett.com` over HTTPS in the US.

Figure 6.2: The case of `gannett.com` in the UK vs. the US.

(a) `match.com` over HTTP in BR.

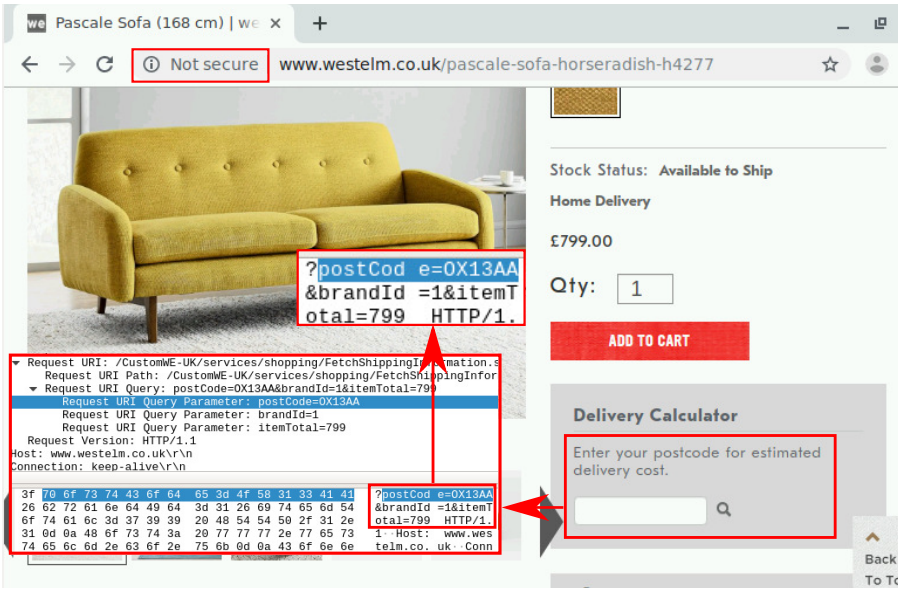
(b) `match.com` over HTTPS in the US.

Figure 6.3: The case of `match.com` in BR vs. the US.

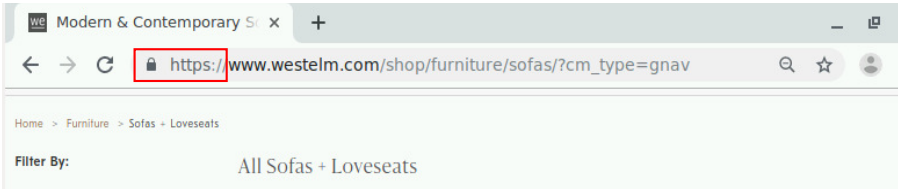
which redirects to the insecure plain-HTTP version of the website. From this case we find that visiting the regional domains manually, e.g. `br.match.com` can be more secure than visiting the generic domain, e.g. `match.com` that redirects to the regional domain, to avoid potential redirection misconfigurations.

We also identify inconsistent cases of plain-HTTP due to partial HTTPS in some but not all regions. The term partial HTTPS refers to websites deploying HTTPS partially, e.g. in some pages such as login pages, while leaving the rest of the pages sent over plain-HTTP, which endangers users' privacy. This is the case we find in `westelm.com` which deploys partial HTTPS in the UK, while deploying full HTTPS in other regions such as the US. See Figure 6.4, which shows that some data, such as users' postcode and item prices, are sent in plaintext for the UK clients. We have emailed their customer service about this weakness, and we offered clarification and help, but we have not heard back yet.

Despite the fact that the user credentials of the weak and strong domains in the aforementioned two examples: the US's `match.com` vs. the BR's `br.match.com`, and the US's `westelm.com` vs. the UK's `westelm.co.uk`, are not shared, and each region requires a separate account (we tested this manually), plain-HTTP final URL inconsistencies are still dangerous for several reasons. First, it provides degraded security for some regions' users. Second, high profile domains provide a false sense of security. For example, a user in one region that used to receive strong



(a) `westelm.com` browsing over HTTP in the UK. The image in the border shows some personal data such as postcode and product prices are sent in clear-text.



(b) `westelm.com` browsing over HTTPS in the US.

Figure 6.4: The case of `westelm.com` in the UK vs. the US.

HTTPS security guarantees, will expect the same security guarantees if the user moved or traveled to another region. Third, the scenario of shared user credentials between different regions is not unusual, and we have found it used in `amazon.com` for example, where we can use a single credential among various regional domains such as `amazon.com` for the US users and `amazon.co.uk` for the UK users. This can enable an attacker who succeeds in redirecting a user from a secure version of the website in one region to an insecure one in another region, to compromise the user's credentials while using the insecure domain.

Second, in terms of **plain-HTTP intermediate URLs**, we find 501 inconsistent cases, where one or more plain-HTTP intermediate URLs are found in some but not all of the responses to the five regions. Note that the results in this indicator are computed out of the HTTP(S) responses. However, if we compile the plain-HTTP

Table 6.3: Examples of incompatible final domains and the regions they are found in.

Region	Initial Domain	Final Domain
AU	https://bbcchannels.com	https://www.bbcaustralia.com
UK	https://aa.com	https://www.americanairlines.co.uk/homePage.do?locale=en_GB
UK	https://live2all.com	https://www.livetotal.net
UK	https://cartoonnetworkasia.com	https://www.cartoonnetworkeurope.com
US	https://hoteis.com	https://www.hotels.com/?pos=HCOM_US&locale=en_US

intermediate URLs inconsistent cases out of the HTTPS final responses only, the number of inconsistent plain-HTTP intermediate URLs drops to 344, which means that 31.34% of plain-HTTP inconsistent intermediate URLs cases contain plain-HTTP final URLs in one or more regions. Moreover, there are 105 (20.96%) domains of the plain-HTTP intermediate URLs inconsistent cases that intersect with the domains of plain-HTTP final URLs inconsistent cases. Unlike final URLs which are normally visible to the user, e.g. through the URL bar in web browsers, intermediate URLs in redirection chains are invisible to the user, which makes inconsistencies in plain-HTTP intermediate URLs worse than those in final URLs. plain-HTTP intermediate URLs can enable a man-in-the-middle attacker to impersonate an intermediate domain, and redirect the user to a malicious, e.g. a phishing website.

Third, in terms of **Incompatible Final URLs**, we find 280 inconsistent cases. We consider incompatible final URLs as a weakness indicator as they contradict the common security advice to users which recommends visually checking the compatibility of the requested domain (e.g. in e-mail links) against the displayed response domain, and suspect response domains that are incompatible with the requested domain. Although HTTP(S) response domains can be incompatible with the requested domain for benign reasons (e.g. a different brand name for a company), having incompatible response domains to a particular requested domain in some but not all of the regions is particularly suspicious. Table 6.3 provides several examples which we also cross-checked with the Chrome browser and found the same behavior that our dataset indicates. The examples also illustrate the difficulty of judging whether an incompatible final domain name is benign or not. For example, in the case of `hoteis.com`, the response domain `hotels.com` differs from the requested domain only in one letter, which is a common phishing technique.

Table 6.4: The `tefal.com` case of incompatible final response URLs found in BR and US regions.

Region	Requested Domain	Final Domain
AU	`https://tefal.com`	`https://www.tefal.com.au`
BR	`https://tefal.com`	`https://www.arno.com.br`
IN	`https://tefal.com`	`https://www.tefal.in`
UK	`https://tefal.com`	`https://www.tefal.co.uk`
US	`https://tefal.com`	`https://www.t-falusa.com`

It also shows the lack of standard domain name format for regional domain names. While many domains tend to change either the subdomain or the TLD for regional domains, we found several cases of completely different domain names with regional indication. For example, requesting `bbcchannels.com` from the AU is redirected to `www.bbcaustralia.com`, a whole different domain name. The lack of standard regional domain names hardens the verifiability of the requested domain, and opens a door for phishing attacks when users benignly believe that the new domain name is a result of regional redirection. Table 6.4 shows a case of incompatible final URLs for `tefal.com` in the BR and US regions.

Finally, in terms of **Incompatible Intermediate URLs**, we find 205 inconsistent cases. Note that we count the incompatible intermediate URLs inconsistencies regardless of the status of the final URLs. However, there are 100 (48.78%) of the domains in the incompatible intermediate URLs inconsistent cases that intersect with the domains of incompatible final URLs inconsistent cases. Unlike final URLs, intermediate URLs are requested silently in the background and are invisible to the user, which makes incompatible intermediate URLs even more suspicious than incompatible final URLs. Table 6.5 shows examples of the incompatible intermediate URLs we found.

6.4.2.2 Security Headers

The results of our analysis of security headers' inconsistencies are computed over the valid HTTPS responses only (the highlighted row "Valid Consistent HTTPS Final Responses" in Table 6.1), since measuring security headers' inconsistencies is

Table 6.5: Examples of incompatible intermediate URls and the regions they are found in. We use the sign ">" to denote redirection.

Region	Initial Domain	Intermediate Domain	Final Domain
AU	https://hobiecat.com	https://hobiecat.com/ > https://www.hobie.com/> https://www.hobie.com/internationalize/ > http://www.hobie.com/au/en/	https://www.hobie.com/au/en/
IN	https://esmas.com	https://esmas.com/ > http://www.televisa.com/	https://www.televisa.com/
UK	https://cpp.com	https://cpp.com/ > https://www.themyersbriggs.com/	https://eu.themyersbriggs.com/
UK	https://centrum.cz	https://centrum.cz/ > https://www.centrum.cz/ > https://id-eco.cz/?redirecturl= https%3A%2F%2Fwww.centrum.cz%2F%3Fredirected%3D1555006695 > https://uid.centrum.cz/?tracking-uid=ftI8-agI8f&redirecturl= https%3A%2F%2Fwww.centrum.cz%2F%3Fredirected%3D1555006695	https://www.centrum.cz/?redirected=1555006695

Table 6.6: Summary of the security headers inconsistencies. The "Inconsistent" column shows the overall inconsistent cases against the weakness indicator. The No-HSTS and No-CSP percentages are computed over the 153,761 valid joint HTTPS responses. The No-Secure in Set-Cookie percentage is computed over the 77,568 valid joint HTTPS responses that contain Set-Cookie headers.

Weakness Indicator	Inconsistent
No-HSTS	517 (0.34%)
No-CSP	481 (0.31%)
No-Secure in Set-Cookie	223 (0.29%)

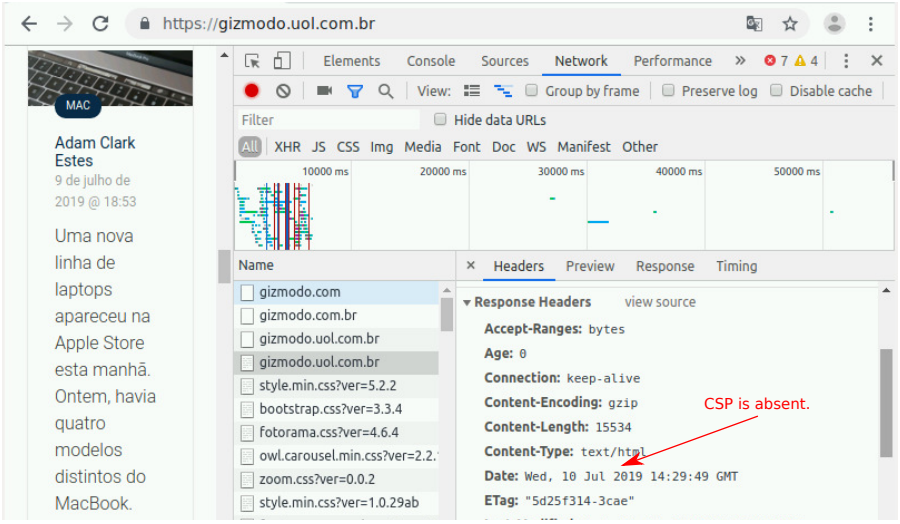
more meaningful in HTTPS connections. For the cookies, we check the consistency of the Secure attribute among HTTPS responses that have the Set-Cookie header in all the five responses to our clients' requests in the five regions.

As depicted in Table 6.6, we find 517 inconsistent cases against the No-HSTS, 481 against No-CSP, and finally, 223 against the No-Secure in Set-Cookie indicator. The HSTS is the highest inconsistent security header (see Listing B.6 in Appendix B.3 for the MySQL query used to find this result), followed by the CSP then the Secure attribute in the Set-Cookie headers. Figure 6.5 and Figure 6.6 show two cases of inconsistent headers (CSP and HSTS).

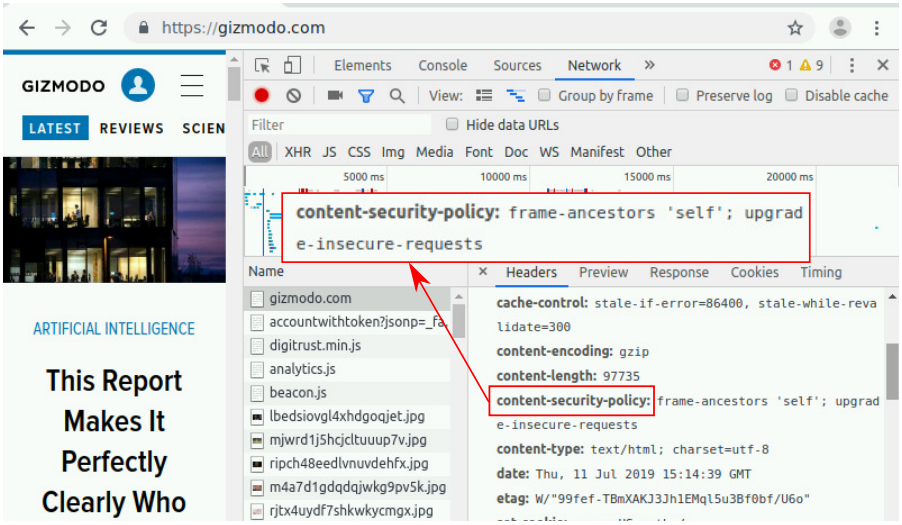
6.4.2.3 Transport Layer Security (TLS)

The results of our analysis of TLS inconsistencies are computed over the valid HTTPS responses that also responded to the TLS scan. The results of our analysis of TLS inconsistencies are summarised in Table 6.7.

Inconsistencies in the "Version <TLS 1.3" indicator are the highest. Versions <TLS 1.3 are prone to various attacks including several cases of downgrade attacks.



(a) `gizmodo.com` in BR.

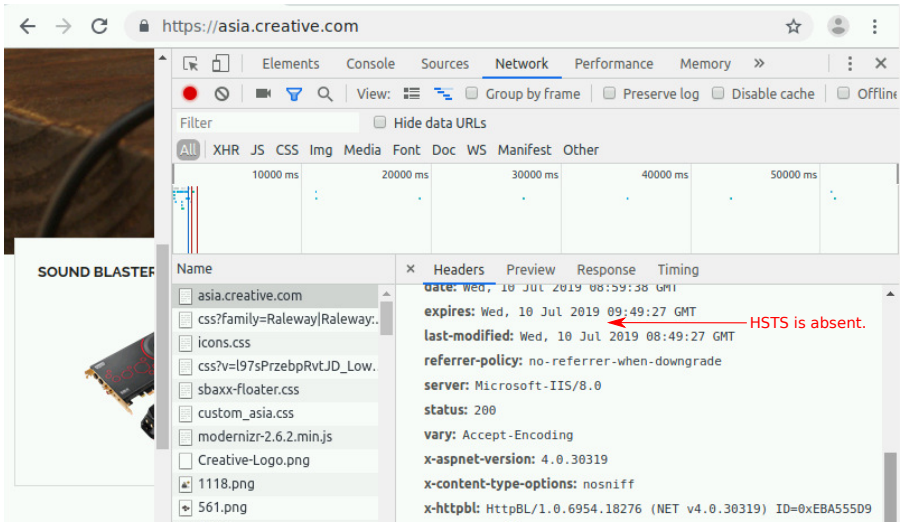


(b) `gizmodo.com` in the US.

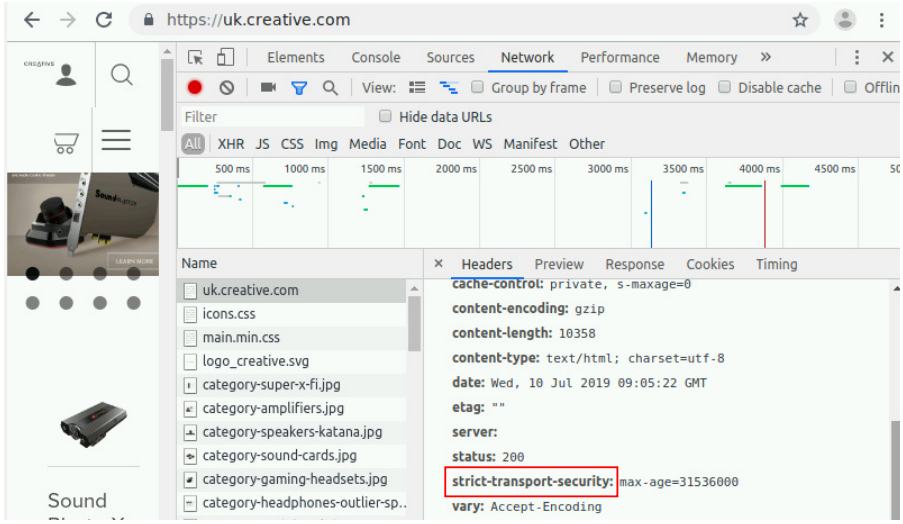
Figure 6.5: The case of CSP header `gizmodo.com` in BR vs. the US.

Table 6.7: Summary of the TLS security inconsistencies. The "Inconsistent" column shows the overall inconsistency cases against the weakness indicator. The percentages are computed over the 152,836 valid joint HTTPS responses that also have responses in the TLS scan.

Weakness Indicator	Inconsistent
Version <TLS 1.3	579 (0.38%)
Version <TLS 1.2	54 (0.04%)
Non-FS	100 (0.07%)
Non-FS+Non-AE	71 (0.05%)
Expired Cert.	21 (0.01%)
Invalid Cert. Host Name	54 (0.04%)



(a) `creative.com` in IN.



(b) `creative.com` in the UK.

Figure 6.6: The case of HSTS header `creative.com` in IN vs. the UK.

TLS versions <TLS 1.2 do not support AE, hence, are prone to attacks over the MAC-then-encrypt schemes. In terms of "Non-FS", we find 100 inconsistent cases (see Listing B.7 in Appendix B.3 for the MySQL query used to find this result). Websites that select Non-FS ciphersuites endanger users' data for future decryption as non-FS ciphersuites do not provide protection to past session keys if the servers' long-term key is compromised at some point in time. We also find 71 inconsistent cases in the "Non-FS+Non-AE" indicator. Expired certificate and invalid host name in certificates have the lowest number of inconsistent cases. We exclude six

inconsistent cases in the "Expired Cert." weakness indicator. In these cases, the inconsistencies arose from the fact that these certificates expired during the scan, where some clients connected to the server before the certificate's expiration while others connected to it after the certificate's expiration. Since the reason is related to the scan time, we decided to exclude them.

6.4.2.4 Relationship to URLs and IPs Diversity, and to Redirection Presence

We also check whether HTTPS security inconsistencies have a relationship to URLs and IPs diversity, and to the presence of redirections. To this end, we define three new criteria:

1. Diverse URLs: is satisfied if the final URLs of a server's responses to the five regions' clients' requests to a particular domain are not equal. We compare the final responses' URLs as is, including the protocol scheme part (i.e. http(s)://).

2. Diverse IPs: is satisfied if the IPs of a server's responses to the clients' requests to a particular domain are not equal.

3. Redirections >0: is satisfied if a server's responses to the clients' requests to a particular domain contain one or more redirections.

After that, for each criterion, we compute the number of domains that satisfy the criterion under two conditions:

1. **Inconsistent** HTTPS responses that have some but not all responses satisfy a weakness indicator denoted by "Inconsistent" in Table 6.8. For example, the "expired cert." weakness indicator is satisfied if some but not all responses have expired certificate, and "version <TLS 1.2" if some but not all responses have version <TLS 1.2.

2. **Secure** HTTPS responses that *do not* satisfy the weakness indicator (i.e. satisfy the negation of the weakness indicator that is specified in the second column in Table 6.8) denoted by "Secure" in Table 6.8. For example, in the

"expired cert." weakness indicator, the secure cases are those that receive responses with valid certificates for the five regions' clients' requests, and for "Version <TLS 1.2" indicator, the **Secure** cases are those that receive responses with versions ≥ TLS 1.2 to the five regions' clients' requests.

The higher the number we obtain from this calculation, the stronger the relationship to the criterion (divers URLs, diverse IPs, or redirection >0). To compute the percentages, as depicted in Table 6.8, for each of the three criteria we examine, we divide the number of "Inconsistent" cases that also satisfy the examined criterion by the total number of inconsistent cases. The same is applied to the "Secure" cases. We divide the number of "Secure" cases that also satisfy the examined criterion by the total number of secure cases. It should be noted that the total number of inconsistent cases, i.e. the divisor of the "Inconsistent" columns in Table 6.8 should be equal to the results we obtained in our inconsistency analysis in Table 6.2, Table 6.6, and Table 6.7, expect in the "Divers IPs" criterion where this does not hold because we compute the total inconsistent cases that responded to the TLS scan, as we only collect IPs in the TLS scan, while the inconsistent cases in Table 6.2 and Table 6.6 are computed from the redirection scan. To give an overview of the queries we perform to build the relationships in Table 6.8, we include the MySQL queries for the "Inconsistent" and "Secure" responses with respect to "incompatible inte. URL" weakness indicator in relation to "Diverse URLs" (i.e. row 4 under the "Diverse URLs" title in Table 6.8). See Listing B.8 in Appendix B.3 for the MySQL script used to compute the "Inconsistent" responses with respect to incompatible intermediate URLs weakness indicator and have diverse URLs, and Listing B.9 in Appendix B.3 for the total number of "Inconsistent" responses with respect to incompatible intermediate URLs weakness indicator. See Listing B.10 in Appendix B.3 for the compatible "Secure" responses with respect to intermediate URLs with diverse URLs, and Listing B.11 in Appendix B.3 for the total number of compatible "Secure" responses with respect to intermediate URLs.

The results show that domains with inconsistent HTTPS security tend to have higher percentages of diverse URLs and IPs, and to a lesser extent redirections,

than secure domains. There are a few exceptions (highlighted in grey) in the TLS weakness indicators, where inconsistent cases with respect to "Non-FS+Non-AE" indicator have less diverse IPs than those in secure cases. Similarly, inconsistent cases with respect to "Version <TLS 1.2" and "Expired Cert." have less redirection >0 than secure cases. Table 6.8 summarises these results, and Figure 6.7 demonstrates the percentages of URL diversity in inconsistent HTTPS responses compared to the secure ones (the results in column labeled "Diverse URLs" in Table 6.8). Note that inconsistent HTTPS against the "plain-HTTP Final URL" and "Incompatible Final URL" weakness indicators imply diverse URLs. This is because it is always the case that some URLs differ in the URL scheme ("https" vs. "http") in inconsistent HTTPS cases against the "plain-HTTP Final URL" weakness indicator, which implies diverse URLs. Similarly, it is always the case that some URLs differ in the domain names in inconsistent HTTPS cases against the "Incompatible Final URL" weakness indicator. This explains the 100% diverse URLs in the inconsistent cases against these two weakness indicators in Table 6.8. The same applies to the HTTPS inconsistent cases against the following weakness indicators: "plain-HTTP Final URL", "plain-HTTP Inter. URL", "Incompatible Final URL", and "Incompatible Inter. URL": they imply redirection, therefore the percentage of diverse URLs with them is 100% as illustrated in Table 6.8.

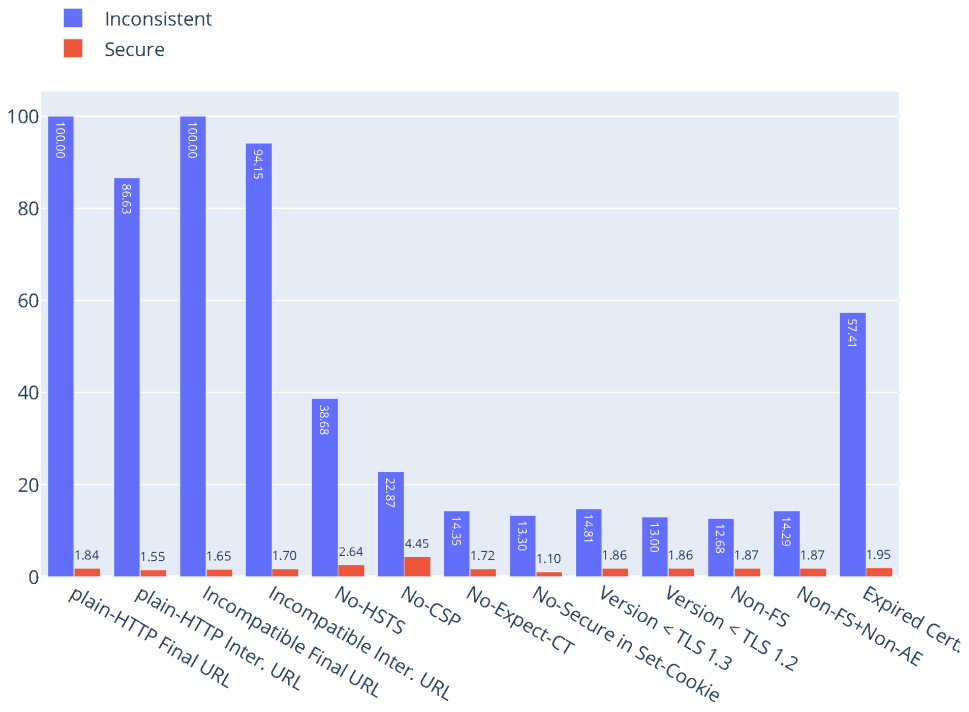


Figure 6.7: A chart illustrating the percentages of diverse URLs in inconsistent vs. secure HTTPS domains.

Table 6.8: The percentage of diverse final URLs, diverse IPs, and redirections >0 in two cases: inconsistent HTTPS responses that have some but not all responses satisfy a weakness indicator (labelled "Inconsistent"), and consistent HTTPS responses that do not satisfy the weakness indicator (labelled "Secure"). The "Diverse IPs" column results are computed over domains that responded to the TLS scan as we retrieve the IPs in the TLS scan only.

Vector	Weakness Indicator	Diverse URLs		Diverse IPs		Redirection >0	
		Inconsistent	Secure	Inconsistent	Secure	Inconsistent	Secure
URLs	plain-HTTP Final URL	232 / 232 (100%)	2822 / 153761 (1.84%)	148 / 211 (70.14%)	56384 / 152836 (36.89%)	232 / 232 (100%)	81405 / 153761 (52.94%)
	plain-HTTP Inter. URL	434 / 501 (86.63%)	2317 / 149012 (1.55%)	376 / 492 (76.42%)	53254 / 147729 (36.05%)	501 / 501 (100%)	76656 / 149012 (51.44%)
	Incompatible Final URL	280 / 280 (100%)	2576 / 155689 (1.65%)	249 / 263 (94.68%)	55818 / 154501 (36.13%)	280 / 280 (100%)	83333 / 155689 (53.53%)
	Incompatible Inter. URL	193 / 205 (94.15%)	2724 / 160378 (1.7%)	185 / 201 (92.04%)	58082 / 159026 (36.52%)	205 / 205 (100%)	88022 / 160378 (54.88%)
Headers	No-HSTS	200 / 517 (38.68%)	820 / 31062 (2.64%)	462 / 516 (89.53%)	13245 / 30945 (42.8%)	494 / 517 (95.55%)	19842 / 31062 (63.88%)
	No-CSP	110 / 481 (22.87%)	463 / 10403 (4.45%)	442 / 481 (91.89%)	4111 / 10382 (39.6%)	449 / 481 (93.35%)	7246 / 10403 (69.65%)
	No-Secure in Set-Cookie	32 / 223 (14.35%)	236 / 13684 (1.72%)	164 / 222 (73.87%)	7036 / 13637 (51.59%)	142 / 223 (63.68%)	6979 / 13684 (51%)
TLS	Version <TLS 1.3	77 / 579 (13.3%)	389 / 35399 (1.1%)	547 / 579 (94.47%)	27867 / 35399 (78.72%)	532 / 579 (91.88%)	17038 / 35399 (48.13%)
	Version <TLS 1.2	8 / 54 (14.81%)	2793 / 149939 (1.86%)	26 / 54 (48.15%)	56283 / 149939 (37.54%)	21 / 54 (38.89%)	80153 / 149939 (53.46%)
	Non-FS	13 / 100 (13%)	2714 / 146180 (1.86%)	43 / 100 (43%)	55964 / 146180 (38.28%)	57 / 100 (57%)	77903 / 146180 (53.29%)
	Non-FS+Non-AE	9 / 71 (12.68%)	2615 / 139558 (1.87%)	25 / 71 (35.21%)	55662 / 139558 (39.88%)	38 / 71 (53.52%)	74241 / 139558 (53.2%)
	Expired Cert.	3 / 21 (14.29%)	2804 / 149945 (1.87%)	15 / 21 (71.43%)	56325 / 149945 (37.56%)	6 / 21 (28.57%)	80499 / 149945 (53.69%)
	Invalid Cert. Host Name	31/54 (57.41%)	2767 / 142126 (1.95%)	39 / 54 (72.22%)	55703 / 142126 (39.19%)	37/54 (68.52%)	79950 / 142126 (56.25%)

6.5 Attack Scenarios

We now provide two attack scenarios that can benefit from the regional HTTPS security inconsistencies.

6.5.1 Region-Confusion Attack

In this attack, the attacker is located on the communication channel and has control over it. It can passively eavesdrop, or actively inject, modify, drop, replay, or redirect messages, sent from, or to, the client or server. We assume that the domain has regional HTTPS security inconsistencies. That is, if two clients at different regions request the same domain (e.g. `example.com`), some regions receive weak security guarantees, while other regions receive strong guarantees. The requested domain may perform URL redirection based on the region ("regional redirection") to a different domain that may have a different IP, contains a new or different subdomain, different TLD, or different documents. We assume the redirection occurs from gTLDs to ccTLDs, and not between ccTLDs. For example, requesting "example.com.sg" from the UK does not redirect to any other domain, while requesting "example.com" redirects to "example.co.uk". We also assume that the requested domain uses the same user credentials (e.g. username and password) such that users can login to multiple domains with the same credentials. Our attacker aims to force the user, e.g. via DNS spoofing if the targeted domains share the certificate, to connect to the weaker domain that has weaker security guarantees. For example, the attacker redirects a UK-based client's request for `example.com` from going to `example.co.uk` to another regional domain that belongs to the same domain, but provides weaker HTTPS security guarantees, e.g. `example.co.br`. This allows the attacker to exploit the weaknesses that exist in the weak region's domain, e.g. perform an XSS attack, due to the absence of a security configuration, e.g. the CSP header, or perform TLS stripping attack, due to the absence of the HSTS header combined with the absence of the includeSubDomains (if the HSTS is present in the parent domain and the regional domain is a subdomain of the parent domain). This attacker model allows more persistent, and harder

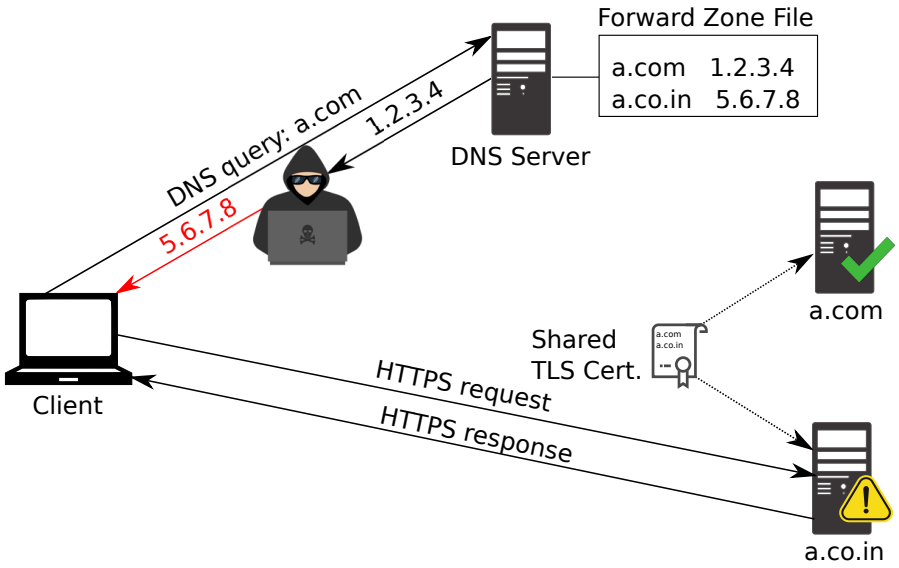


Figure 6.8: DNS spoofing or poisoning exploiting regional inconsistencies.

to detect attacks than in the classical phishing attacks, e.g. via a faked website. This is because, in this attack, the user is redirected to a legitimate website that accepts the user's credentials. The attacker can obtain and abuse, in the long-term, the user's credentials, as opposed to a faked phishing website, where the user is normally more likely to find out that the website is faked, due to the absence of real content. Figure 6.8 demonstrates this attack.

6.5.2 Discrimination Attack

This attack scenario is based on the "discriminatory" attacker model, which we introduced earlier in Section 4.5.2.3 (Item 3) and illustrated in Figure 4.4. This attacker is located on the server side. It can be represented by an insider attacker (e.g. a dishonest system administrator), a dishonest organisation (e.g. a giant cloud provider), or a malware that hit the server. It weakens the security guarantees provided to clients in some regions, for a powerful third party's advantage, e.g. a state level attacker, who has the capabilities to exploit this weakness. For example, the discriminatory attacker can deny some regions from some security guarantees such as the FS property, to enable the powerful third party attacker to decrypt collected traffic either at present, due to the third party's powerful capabilities of breaking non-FS keys, or in the future once the server's long-term key is broken

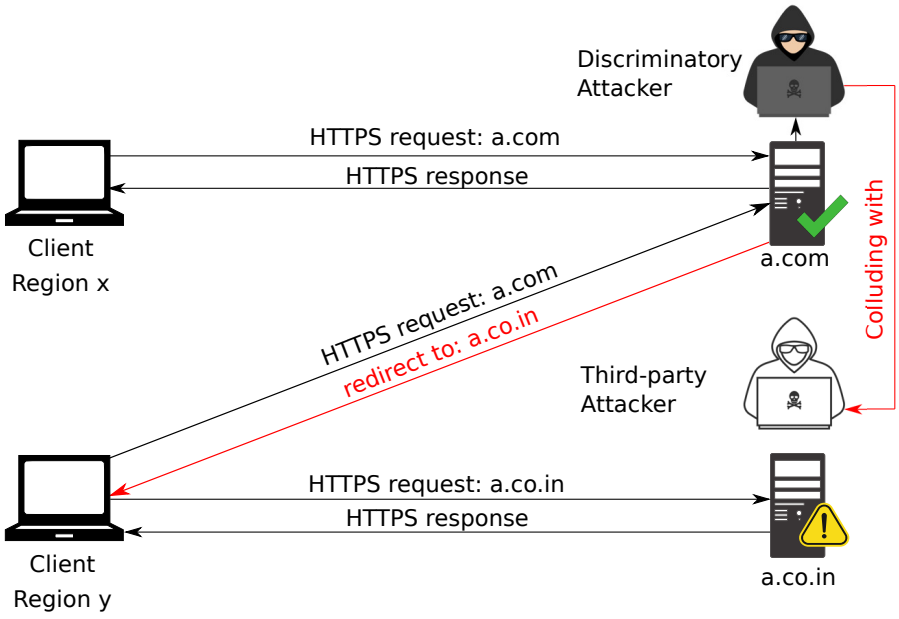


Figure 6.9: A discriminatory attacker model exploiting regional inconsistencies.

due to advancement in computing power, or if the key is given to the powerful third party attacker, after the key is expired when it is no longer used by the server. The semi-trusted server colludes with, or is compelled by, a powerful third party attacker. This model gives the server a financial, legal, and reputational advantage over giving every session key or the decrypted traffic to the third party powerful attacker. This attacker model is inspired by real world incidents such as the export grade cryptography, a depreciated US law that mandated weaker cryptography to products, including software, exported outside the US [29], in addition to the alleged "PRISM" program in which giant service providers open backdoors that are known to third parties (e.g. intelligence) [140]. Figure 6.9 demonstrates this attack.

6.6 Recommendations

Based on our analysis and observations in this experiment, we make the following recommendations:

1. Security favours simplicity: as shown in Section 6.4.2.4, domains with inconsistent HTTPS security tend to have higher percentages of URLs and IPs diversity, and to a lesser extent redirections, compared to domains with secure

HTTPS. This suggests that the more complex the domain is (e.g. diverse IPs and URLs), the more inconsistent HTTPS security is.

2. Secure redirection, and avoiding redirection if possible: the security of intermediate URLs in redirections is no less important than the security of final URLs. Therefore, all intermediate connections need to be over TLS and need to deploy the same security measures as final URLs. This is also noted in [59], where they suggest strict redirection policy in the same vein as HSTS. We also recommend users to use the regional domain directly in their requests if known, as this can provide better security than requesting the generic domain that redirects to the regional domain, to avoid insecure redirections, as shown in the case of `match.com` in Section 6.4.2.1. Additionally, avoiding redirections could be automated by a browser extension that buffers the URL chain and makes the request to the final URL, and by prioritizing regional domains in search engines, based on the region of the searching client.

3. TLS is not only about secrecy: while the content of blocking pages is public and may not require secrecy, the authenticity and integrity of such pages are still important to avoid DoS attacks for example. Therefore, domain owners and governments who present blocking pages in plain-HTTP need to be aware of the TLS goals and the possible harm that can result from unauthenticated blocking pages, and that users' habituation to plain-HTTP can make them more susceptible to DoS attacks that abuse this issue.

4. HTTPS security inconsistencies and the potential effect on Tor network users: the Tor network aims to provide anonymity to its users. It hides users' locations and the websites they visit online, by routing traffic through multiple relay servers run by volunteers all over the world [141]. In Tor, if the domain names are resolved using "SOCKS 4a", which passes hostnames to relays, or through "Tor-resolve", which uses Tor network to resolve host names remotely [141], this means that Tor users may be redirected to regional domain names that are different from their region. As our results show, there

is evidence of HTTPS security inconsistencies in servers' responses to requests from different regions. Therefore, a user may end up connecting to a weaker version of the intended website, mainly because the generic domains, e.g. `example.com` are resolved in another region, e.g. `in.example.com`, which may have weaker HTTPS security.

5. Regional domain format: a single standard format for regional domains, with the same-origin policy (of domains) in mind, can help in reducing phishing attacks' surface and help users identify malicious domains and redirections. The domain same-origin policy can be achieved in brand TLDs, where brands are used as a TLD instead of the traditional TLDs, and in regional subdomains, but not in ccTLDs format.

6. Tools for consistency and redirection testing: tools for consistency assessment may help administrators and users in mitigating and detecting HTTPS security inconsistencies.

6.7 Limitations

In terms of limitations, first, while conducting multiple scans is desirable, we opt for a two-stage single scan (see the methodology Section 6.3.2). The multi-regional experiment setup imposes more challenges over a single region setup in terms of data storage, time, and cost. However, to gain more confidence on our scan results, we manually inspect dozens of inconsistent cases using a web browser in the remote machines. However, manual inspection has its limitations: first, it is not practical to inspect all cases. Second, the longer the time span between the scan and inspection, the less useful it becomes, as domains can change over time. Second, due to the dynamic and rapidly changing nature of the web, the domains and examples included in this study may have changed their configurations over time. This is a well understood limitation in Internet measurement studies in general. However, even if some of those websites have changed, this does not diminish the value of the insights we gained from analysing those data. Additionally, this does

not eliminate the existence of inconsistencies in another set of websites. Third, in order to scale, the HTTP and TLS connections to the domains were automated through programmable HTTP and TLS clients. These tools simulate an HTTPS client such as a web browser. However, we do not advertise a specific browser vendor in the requests' headers. Nevertheless, the domains we present in the tables or figures in this chapter are cross-validated against the latest version of the Chrome browser at the time of the study. Additionally, we measure the consistency of servers' responses between different regions regardless of the client's vendor. We use the same client in all regions. Thus, our results should not be affected if a server provides different responses to different vendors. However, considering improved scanning methodologies based on browser automation frameworks is worth considering in future work, to bridge the gap between automated and browser requests. Fourth, in headers and TLS security inconsistencies, we check the values provided in the final response only and we do not check the intermediate URLs. Analysing each URL in the redirection chain is time and resource prohibitive for this project. Fifth, we analyse security headers in terms of headers' presence/absence only. Checking the headers' values such as syntax and configurations correctness is a further level in the analysis depth that is outside our chapter's scope, and we leave it to future work. Sixth, we do not analyse the responses' page content except the manual analysis we performed in Section 6.4.2.1 to identify blocking home pages. Finally, we check our dataset domains (requests' domains) against known malicious domains by Google's safe browsing, and we exclude the responses of requests to malicious domains from our analysis. However, we do not check intermediate or final response domains against malicious domains nor exclude them, if any.

6.8 Conclusion

In this chapter, we examined negotiation consistency, and demonstrated the existence of a previously unexplored phenomenon. That is, the existence of HTTPS security inconsistencies in servers' responses to clients located in five different regions. We quantified the HTTPS security inconsistencies we identified. These inconsistencies

can provide a false sense of security among users from different locations, and can enable attacks that redirect the user's request to the weaker region to exploit the weaker region's weaknesses. We make the recommendations from our experimental observations, which suggest standardising regional domain format and securing redirection, in addition to the need for testing tools for domain administrators and users that help to mitigate and detect regional domains' inconsistencies.

7

Prior Knowledge as a Means to Defeat Downgrade Attacks

Contents

7.1 Introduction 124
7.2 TLS Enumeration Scan 125
7.2.1 Dataset 126
7.2.2 Methodology 126
7.2.3 Results 126
7.3 Preliminaries to Our Proposed Systems 127
7.3.1 Strict vs. Default TLS Policy 127
7.3.2 System Model 129
7.3.3 System Goal 130
7.3.4 Error Handling Mechanisms in Web Browsers 131
7.4 The Proposed Systems 131
7.4.1 Agent-based 132
7.4.2 Header-based 139
7.4.3 DNS-based 141
7.5 Limitations 146
7.6 Summary 148

7.1 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 such as web browsers treat these differences reveals that they enforce coarse-grained TLS security configurations. In other words, a "one size fits all" policy. They[2] support legacy versions of the protocol that have known design weaknesses, and weak ciphersuites that provide fewer security guarantees, e.g. non-FS, or non-AE, mainly for backward compatibility.

Supporting legacy versions or weak ciphersuites provides backward compatibility, but opens doors to downgrade attacks. In a typical downgrade attack, an active man-in-the-middle attacker forces the communicating parties to operate in a mode weaker than they both prefer. Several studies illustrate the practicality of downgrade attacks in the TLS protocol [1], [2], [24], [32], [46], [50], [142]. Despite numerous efforts to mitigate them, new downgrade attacks continue to appear. For example, in a draft for the latest version of TLS (draft-10), TLS 1.3, Bhargavan et al. reported a downgrade attack [24]. Previous attacks have exploited not only design vulnerabilities, but also implementation and trust model vulnerabilities that bypass design level mitigation techniques such as the handshake messages (transcript) authentication. For example, the POODLE [32] (Figure 2.9), DROWN [46], and ClientHello fragmentation [142] (Figure 7.1) 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

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

[2] At the time of writing this chapter (July 2018), 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.

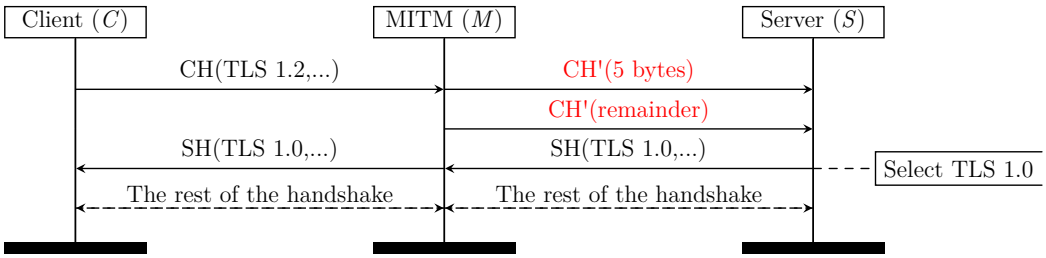


Figure 7.1: Illustration of a version downgrade attack attempt based on the "Version rollback by ClientHello fragmentation" attack scenario [142].

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 to another website.

However, we observe that if the client has prior knowledge about servers' TLS configurations, a better balance between security and backward compatibility can be achieved, which reduces the downgrade attack 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 accordingly, and enforce a strict TLS configurations policy when connecting to these advertising servers, while enforcing default policy for the rest of the servers.

In this chapter, we try to answer the following question: *How to enable clients to make an informed decision on whether to enforce a strict or default TLS configurations policy before connecting to a server?*

In what follows, we first present a TLS versions and ciphersuites enumeration scan to quantify real world setting. Then, we provide preliminaries to our proposed systems. After that, we present the three systems that we examine.

7.2 TLS Enumeration Scan

This section aims to answer questions such as: what is the average number of supported versions and ciphersuites in TLS servers? What level of security

guarantees (strong vs. weak) do these ciphersuites provide? How many TLS servers support a single version exclusively? In what follows, we summarise our experiment.

7.2.1 Dataset

We retrieve the top 10,000 domains list[3] from Alexa Internet [103] on May 5, 2018.

7.2.2 Methodology

To run the scan, we use the sslscan 1.11.11 [143], an open source TLS scanning tool that can perform TLS versions and ciphersuites enumeration through multiple TLS handshakes. The tool supports TLS versions from SSL 2.0 up to TLS 1.2, and 175 ciphersuites. We run the scan from the Singapore University of Technology and Design (SUTD) wired network between the 6^{th} and 12^{th} of May 2018. Our scan is in conformance with the ethical considerations in measurement studies (outlined in Section 1.4).

7.2.3 Results

7.2.3.1 Responses

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 for handshake failure as this is outside our scope. However, one possible contributing factor to the handshake failure in our scan could be due to the SUTD's Internet censorship system that blocks some website categories such as porn and gambling websites.

7.2.3.2 TLS Versions and Ciphersuites

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 versions, both TLS 1.2 and TLS 1.1,

[3]The list gets updated daily, according to Alexa's support.

either exclusively or with other lower versions, is 6462 (91.27%). 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 ciphersuites, which is the case in our findings with 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+non-AE, non-FS+AE, or non-FS+non-AE. 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 non-FS+AE, FS+non-AE, or non-FS+non-AE) in addition to one or more FS+AE ciphersuite.

7.2.3.3 Concluding Remarks

The results show that top domain servers support strong TLS configurations (the strong configurations is defined next in Section 7.3.1). At the same time, they maintain support for weak configurations that have known weaknesses and provide fewer security guarantees. Ideally, clients' configurations influence the servers' selected configurations. Asserting servers' strong configurations to clients adds 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.

7.3 Preliminaries to Our Proposed Systems

7.3.1 Strict vs. Default TLS Policy

Our mechanisms affect the client's fine-grained TLS configurations, namely, the protocol versions and ciphersuites. In addition, it affects the client's fallback

mechanism. In our proposed mechanisms, 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 the 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 the 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 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 7.1 summarises the strict vs. 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)[4]. Finally, we note that in TLS 1.3, FS and AE ciphersuites are enforced by design [16], i.e. strong ciphersuites are implied by TLS 1.3 version. Therefore, in TLS 1.3, the strict configurations policy boils down to the protocol version and the fallback mechanism. However, there is still value in our mechanism's ciphersuites policy even in clients that support and offer TLS 1.3 such as modern web browsers. Unlike most TLS 1.3 clients, where weak and strong ciphersuites are sent in the ClientHello (see Figure 7.2 for the list of versions and ciphersuites that are sent from a modern Firefox browser that support TLS 1.3), relying on the server to select the right version and ciphersuite, our policy forces the client to refine its versions and ciphersuites before the ClientHello message is sent, which provides stronger downgrade resilience. For example, if the server has been tricked, e.g. due to an implementation flaw, to select a weak version or ciphersuite as in the ClientHello fragmentation attack [142], in our proposed systems, the client

[4]At the time of implementing our prototypes, TLS 1.3 version has not been standardised yet. Therefore, we consider TLS 1.2 as the latest version in our systems prototypical implementation. However, in theory, TLS 1.3 is the latest version as shown in Table 7.1.

Table 7.1: The strict vs. default TLS policies that we define in our mechanisms (✓denotes enabled and ✗denotes disabled).

Policy	TLS Version	TLS Ciphersuites	Fallback
Strict	TLS 1.3	FS and AE	disabled
Default	TLS 1.3; TLS 1.2; TLS 1.1; TLS 1.0	FS; AE; non-FS; non-AE	enabled

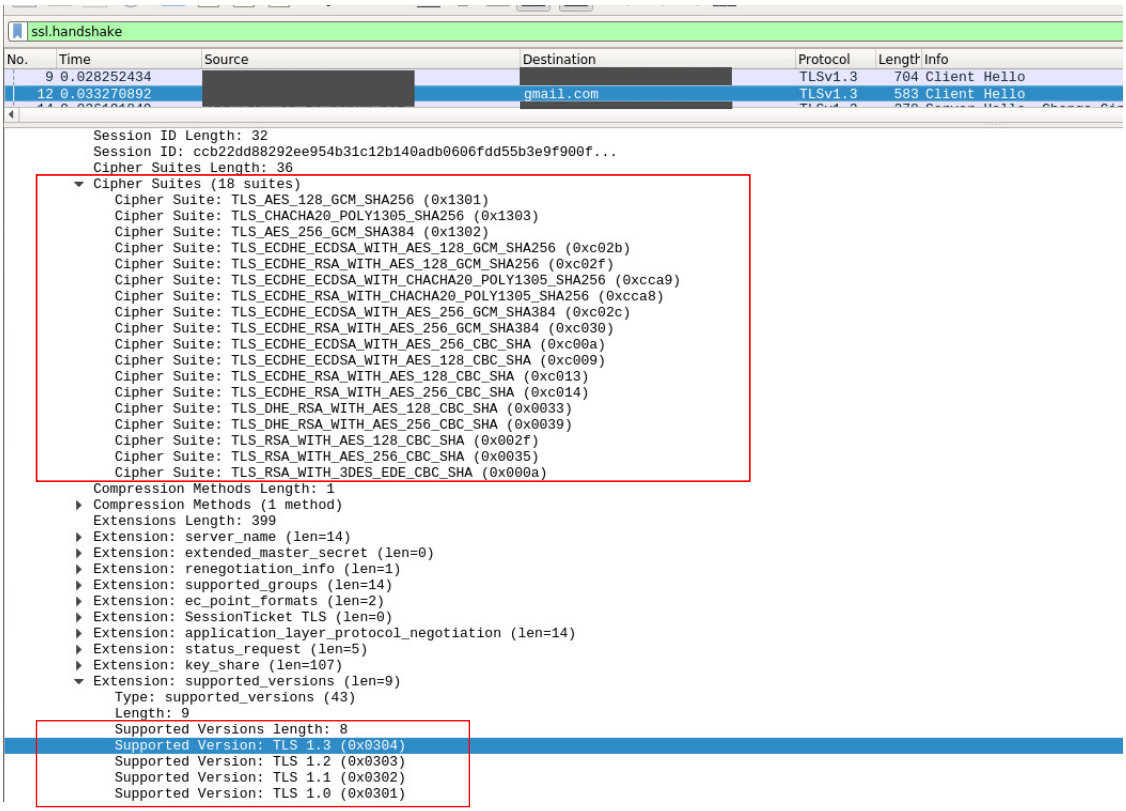


Figure 7.2: Inspection of a TLS 1.3 ClientHello message from a modern Firefox browser (version 71.0) to `gmail.com` using the Wireshark[5] network analyser. Some parts of the source and destination addresses are deliberately redacted.

will not accept to continue the handshake as the enforced policy for that connection has been violated. Experience shows that servers' flaws can be exploited to make the server select the wrong version as in the ClientHello fragmentation attack [142].

7.3.2 System Model

Our system model considers the following parties: a TLS client, a TLS server, and a DNS server. The TLS server is identified by its domain name, and the domain owner controls its DNS zone. These parties are standard for TLS connections. 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, then both parties aim to use it. However, as is the case with most real world systems, the client and server support legacy (weak) versions and ciphersuites. Clients silently fall back to the legacy configurations if the server selected to. Servers silently fall back to legacy configurations if the client offered only legacy configurations. However, legacy configurations must be selected, if and only if, the other party is indeed a legacy one, and does not support the strong configurations. The assumption that the client and server support weak, unrecommended, or plausibly broken configurations such as weak ciphersuites, is realistic. Additionally, classical cryptographic algorithms do not last forever. Algorithm design flaws can be found, and advances in computation powers enable solving hard problems such as prime factorisation. For example, several algorithms were supported through years of speculation about their insecurity until they were officially deprecated such as the RC4 algorithm [111]. Finally, we assume that 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.

7.3.3 System Goal

As explained earlier, several downgrade attacks have been shown to be practical. For example, the version downgrade in the POODLE attack [32], the "Version rollback by ClientHello fragmentation" [142], and the ciphersuite downgrade (from RSA to non-RSA) in a variant of the DROWN attack [46]. These attacks exploit the client's silent fallback to legacy versions or algorithms. Then, they circumvent the handshake transcript authentication mechanism (in the Finished MACs) that is placed to detect any modifications in the protocol messages (including the version or ciphersuite).

Such attacks could have been prevented if the client does not support legacy versions or weak ciphersuites. In these cases, the client will refuse to proceed with the handshake with a version or ciphersuite that it does not support, as illustrated in Figure 7.3.

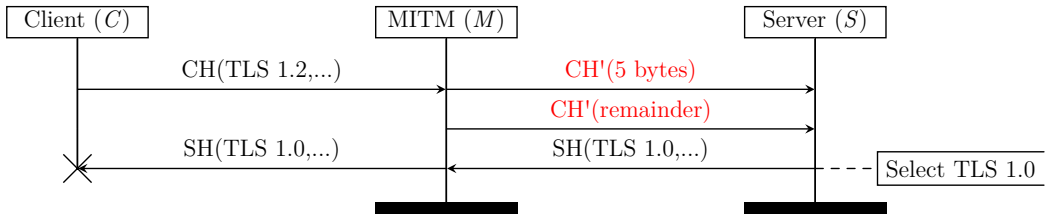


Figure 7.3: Illustration of a version downgrade attack prevention through strict client TLS configurations (when the client does not support legacy TLS versions). The attack scenario is based on the "Version rollback by ClientHello fragmentation" attack in [142].

As stated earlier, our proposals try to tackle the challenge of providing a high level of security for connections going to sensitive websites while maintaining backward compatibility with ordinary websites in TLS clients such as web browsers.

7.3.4 Error Handling Mechanisms in Web Browsers

In general, there are three main strategies for error handling in web browsers: blocking, active warning, and passive warning. Blocking is a conservative approach that prevents the user from proceeding to the website. It should be used when the attack is certain. On the other hand, the active warning strategy is less conservative. It temporarily blocks the users to warn them, but it allows them to click through (bypass) the error through one or more clicks. This strategy is used in self signed certificate warnings in most browsers today. Finally, the passive warning strategy shows an indicator which can be a negative or a positive indicator, without interrupting the user's task, e.g. the padlock icon.

7.4 The Proposed Systems

As stated earlier, our proposals try to tackle the challenge of providing a high level of security for connections going to sensitive websites while maintaining backward compatibility with ordinary websites in TLS clients such as web browsers. To this end, we propose three mechanisms for fine-grained TLS configurations. That is, optimal TLS configurations are enforced for connections to sensitive domains, while

default configurations are enforced for the rest of the connections. To do this, the client needs guidance to distinguish between different contexts. In what follows we describe three mechanisms. The methods themselves are known and have been used with other types of policies such as HSTS [39], [144], HPKP [93], CSP [34], [36], and DANE [90]. However, to the best of our knowledge, applying them to the context of TLS versions and ciphersuites is new and has not been discussed before. We categorise our proposed mechanisms (systems) based on the method of domains' subscription to the policy: agent-based, header-based, and DNS-based.

7.4.1 Agent-based

The first method we examine to enforce fine-grained TLS configurations (versions and ciphersuites) is the agent-based technique. In what follows we describe the system.

7.4.1.1 System Overview

As depicted in Figure 7.4, in this method, the user first needs to prepare a pre-defined list of domains on which the strict TLS policy should be enforced. The system has pre-defined TLS configurations policies: strict and default (see Section 7.3.1 for further details). Then, the system observe the user's HTTPS requests in the URL bar, and checks the domain name of each request. The system examines the requested domain name against the pre-defined list. If found, it enforces the strict policy. Otherwise, the default system policy is enforced. There is a usability concern in this method as it requires motivated and security-aware users. To overcome such a concern, having a pre-defined domains list through some service is desirable.

7.4.1.2 System Details

We implement our proposal's proof of concept as a Firefox browser extension. We now describe the system (i.e. the extension) components:

1. Pre-defined TLS configuration policies: the policies govern the TLS configurations that will be enforced before an HTTPS request is sent. The TLS configurations space for our policies is the set of versions and ciphersuites

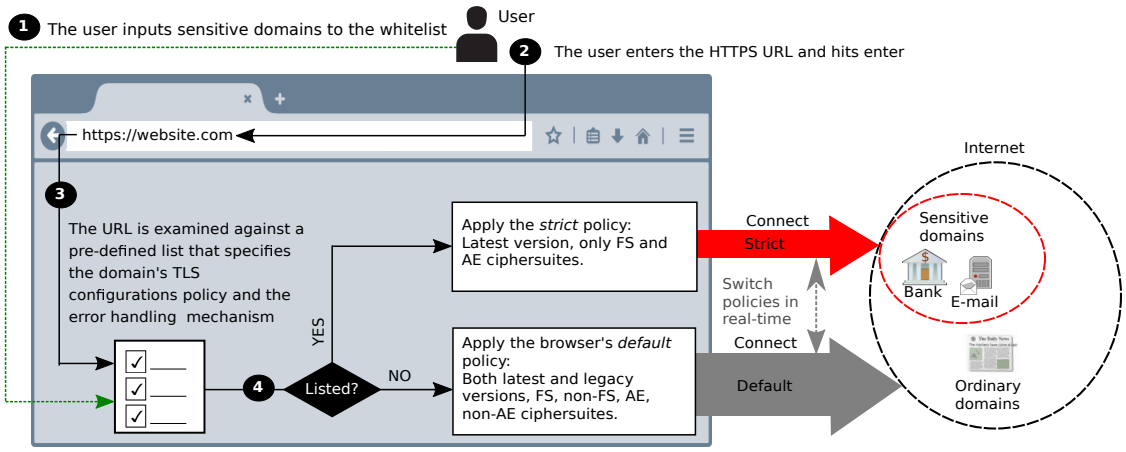


Figure 7.4: System overview of strict TLS policy using agent-based whitelisting.

that exist in Firefox browser developer edition version 55.0b9. That is, in terms of versions: TLS 1.3, TLS 1.2, TLS 1.1, and TLS 1.0, and in terms of ciphersuites: 15 ciphersuites that provide different levels of security guarantees which include FS, non-FS, AE, and non-AE ciphersuites. For our proof of concept, we define two TLS configuration policies: strict and default (as specified in Table 7.1 in Section 7.3.1 except that we consider TLS 1.2 in the strict mode as TLS 1.2 was the highest version at the time of implementing the prototype of this proposal) and hard code them in the extension. The number of policy levels is a design decision. We could have defined more levels if desired. For example, we could have defined a level for each version of TLS. The more levels, the more granularity is achieved. However, more granularity means more warning messages (a warning before falling back to a lower level). We try to maintain a balance between security and usability and decided to define two levels policy where there is a significant shift in the provided security guarantees, and present only one warning message in case of policy violation, to warn the user about falling back from the strict to the default policy. See Figure 7.5 for an illustration of the policy levels.

2. Pre-defined domain names list: the domain names list combined with the TLS policy that is assigned to each domain name is used to provide the browser with the prior knowledge that guides it into which TLS policy should be

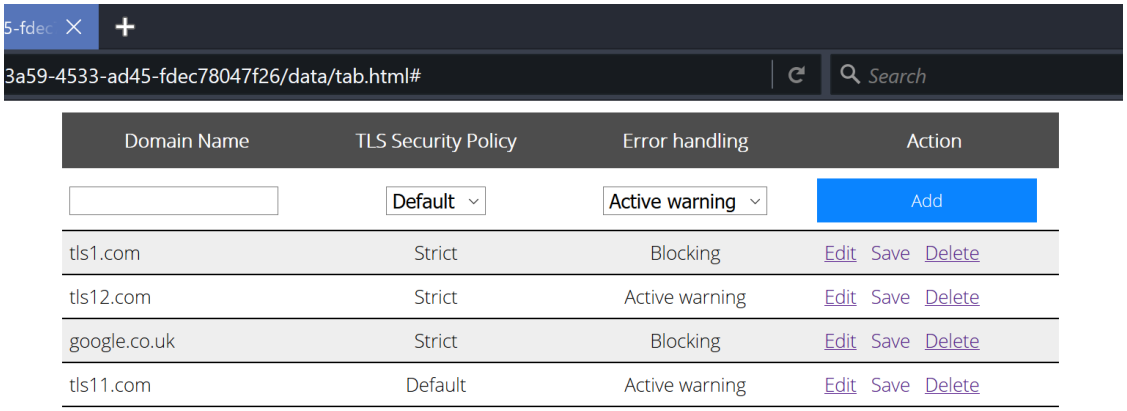


Figure 7.5: Our prototypical Firefox browser extension's list. Users input the whitelisted domains. There are two policies: strict and default, and two error handling mechanisms: active warning and blocking.

enforced for each examined HTTPS request. In our proof of concept, the domain names list takes the domain names manually as an input from the user. The domains are entered in the form of "`example.com`". See Figure 7.5 for an illustration of the domain names list. We do not allow duplicate domain names. Therefore, a domain's policy cannot be overwritten unless after removing the existing record. By default, the domain names are added to the strictest TLS policy that enforces optimal TLS configurations which is the strict policy. However, if the connection using the optimal TLS configurations could not be established due to lack of server support for the requested configurations, the user is presented with a warning message. Depending on the error handling mechanism (which will be explained next) that is assigned to that particular domain, the user will be either blocked from proceeding to the website, or warned and allowed to relax the domain's TLS policy to a lower one (from strict to default in our case).

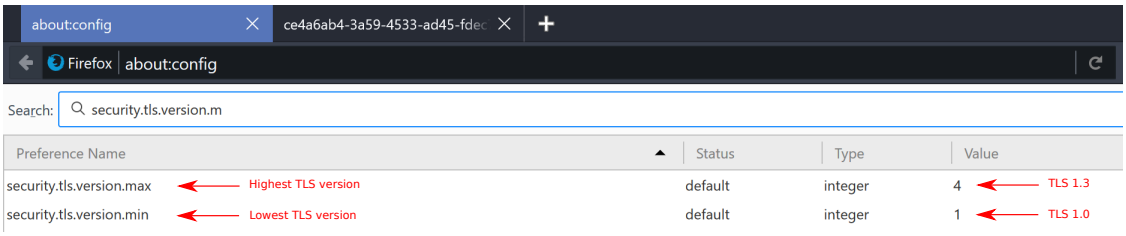
3. Pre-defined error handling mechanism: the error handling mechanism specifies the type of error message that will be presented to the user in case of TLS policy violation. Each whitelisted domain has a TLS configuration policy and an error handling mechanism assigned to it. We define two error handling mechanisms: blocking and active warning. The error mechanism implies the

level of confidence in the server's ability to meet optimal TLS configurations. By default, the error handling mechanism for an agent-based subscription is set to active warning. However, if the user (e.g. IT administrator) has a high level of confidence that the added domain should be able to meet the strict TLS policy (e.g. bank server), he/she can select the blocking error handling mechanism to block the user from proceeding to the website if the TLS server response violates the policy. See Figure 7.5 for illustration.

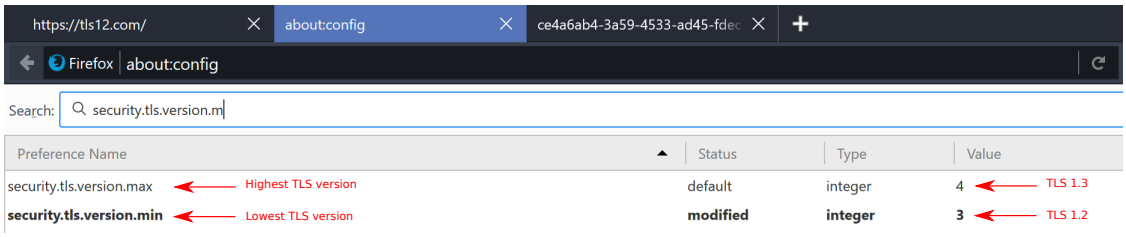
4. HTTP observers: the extension employs three observers (listeners) running in the background, i.e. as long as the extension is running:

   (a) HTTP before send request observer: this observer examines every HTTPS request that goes through the main address bar. The URL is either manually entered in the address bar by the user, or automatically through URL redirections, or through clicking on links. The examination occurs before the request is sent, against the pre-defined domain names list. If the examined URL (e.g. `mail.example.com/etc`) belongs to any of the whitelisted domains (e.g. `example.com`), the extension enforces the TLS policy that is assigned to that domain. If the requested URL does not belong to any of the whitelisted domains, the extension enforces the browser's default policy. After the policy is enforced, the request is sent. Note that the browser rewrites the configurations in real-time. Therefore, if the next URL does not belong to a whitelisted domain, the default policy will be re-enforced again. See Figure 7.6 and Figure 7.7 for an illustration of default versus strict policy configurations in our Firefox extension.

   (b) HTTP error observer: this observer is triggered when the request cannot be processed due to lack of common TLS version or ciphersuite between the client and server. Our extension customises the browser's built-in error detection mechanism if the error occurred for one of the whitelisted domains. Our extension detects the version or ciphersuites



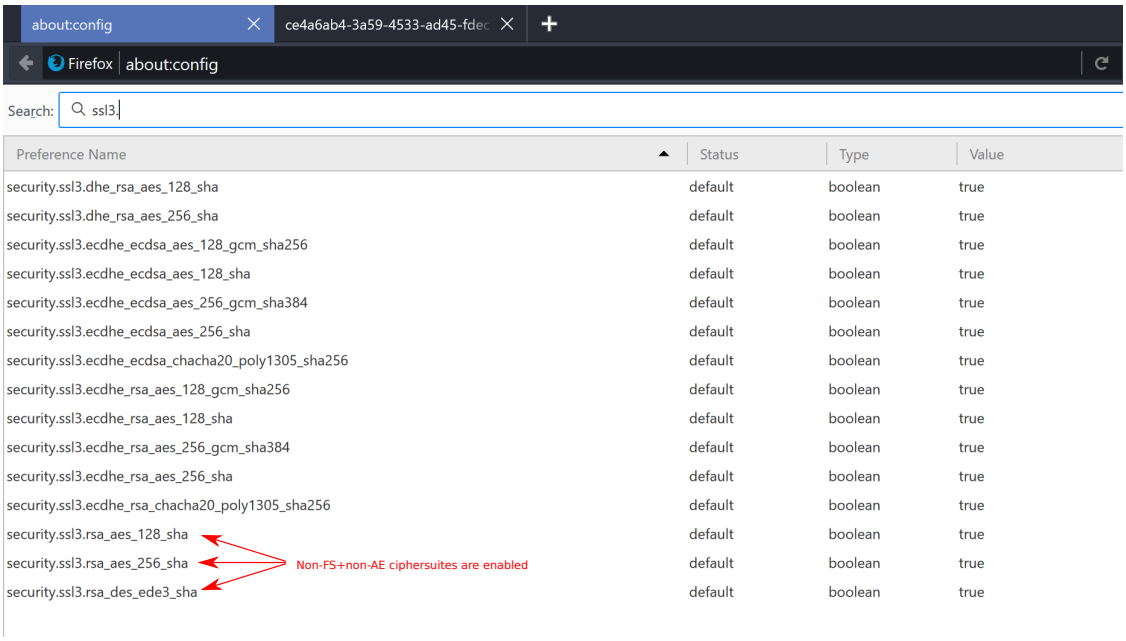
(a) TLS versions in the default policy.



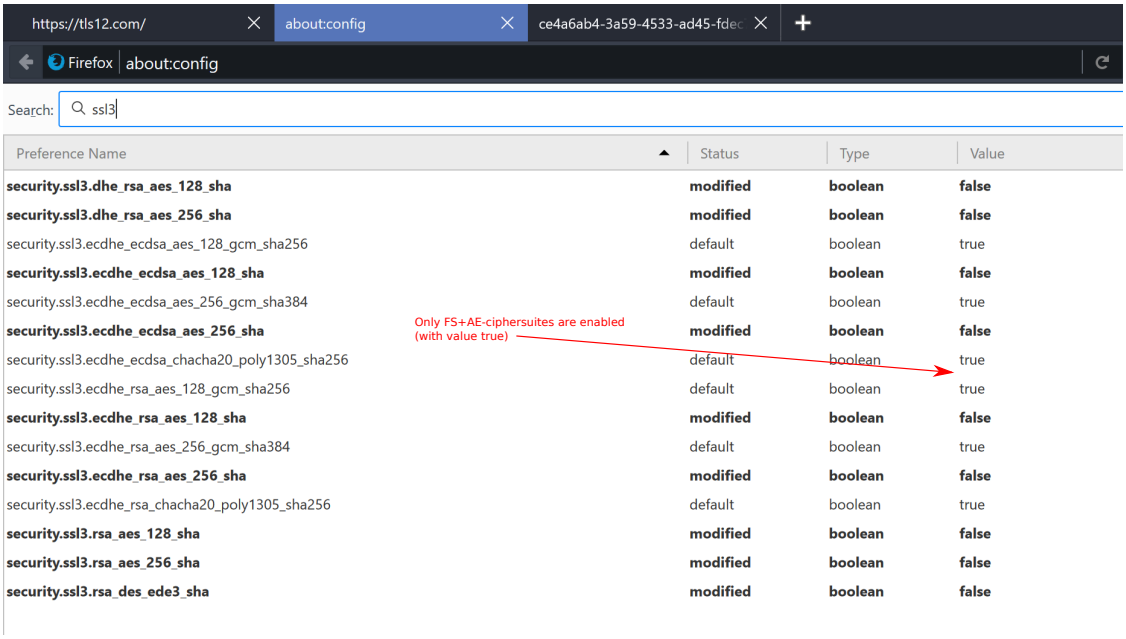
(b) TLS versions in the strict policy.

Figure 7.6: Strict vs. default versions in our Firefox browser extension. The strict versions policy is enforced in real-time when an HTTPS connection is being made to a whitelisted domain.

mismatch errors by observing the loaded document's (i.e. tab's) URI. Then, it matches every loaded tab's URI against defined patterns that represent the Firefox's version and ciphersuites mismatch errors URIs. In particular, we check if the loaded tab URI starts with "about:neterror" and contains either "SSL_ERROR_UNSUPPORTED_VERSION" or "SSL_ERROR_NO_CYPHER_OVERLAP". Note that these patterns are vendor specific. If a match is found, the extension extracts the URL that caused the error from the tab URI. Then, it examines the just extracted URL against the whitelist. If the URL belongs to a whitelisted domain name, the extension updates the tab with our extension's customised error page according to the error handling mechanism that is assigned to the domain name. Note that we borrowed the text of error messages used by Firefox for similar errors, but the error messages' content and design are outside our scope. Figure 7.8 shows the two types of error handling mechanisms in our prototype.

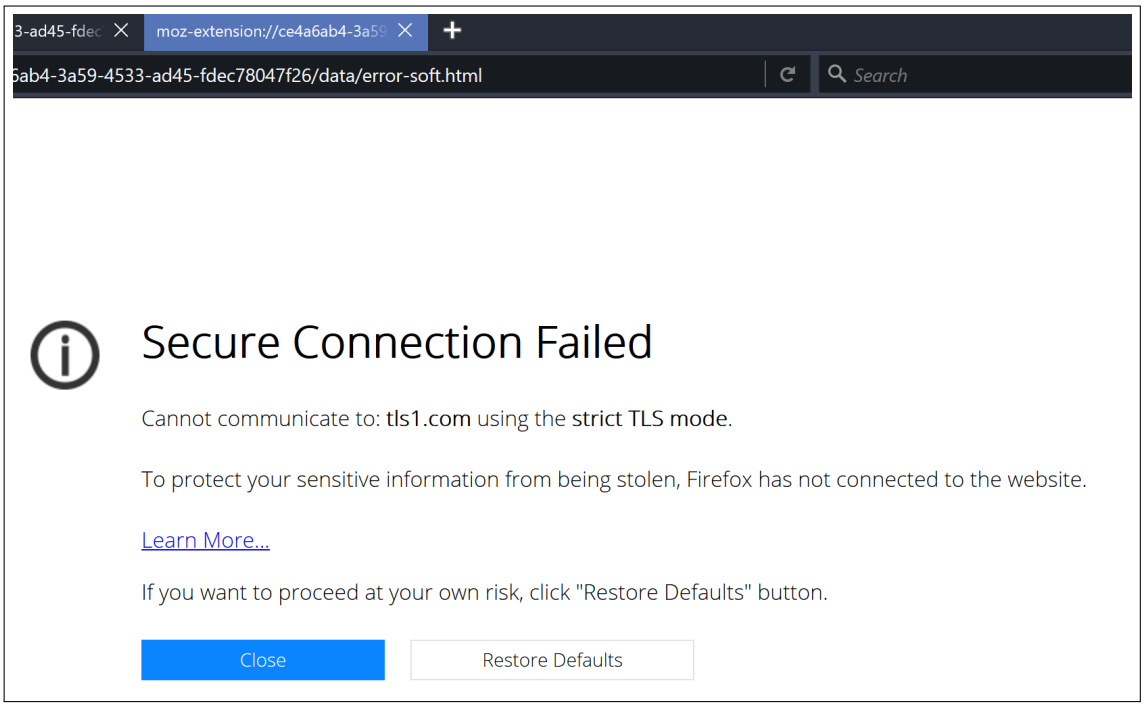


(a) TLS ciphersuites in the default policy.

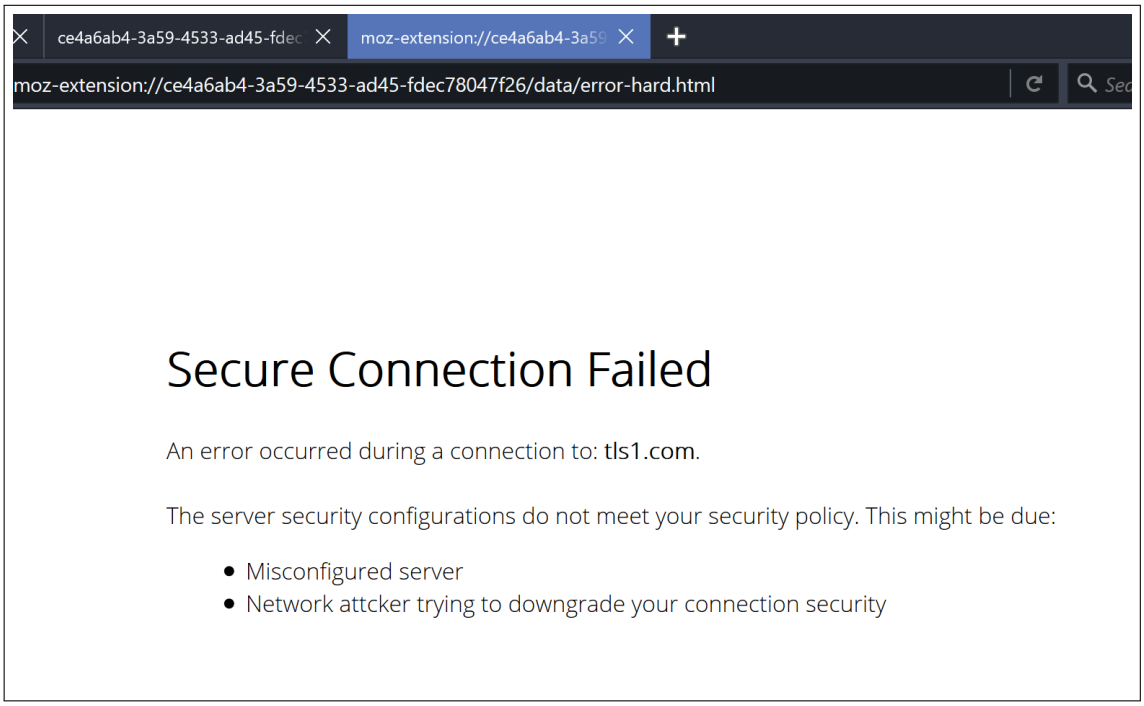


(b) TLS ciphersuites in the strict policy.

Figure 7.7: Strict vs. default ciphersuites in our Firefox browser extension. The strict ciphersuites policy is enforced in real-time when an HTTPS connection is being made to a whitelisted domain.



(a) The warning error message (contains a button to bypass the error and restore defaults) in our Firefox browser extension.



(b) The blocking error message in our Firefox browser extension.

Figure 7.8: Warning vs. blocking error handling mechanisms in our Firefox browser extension.

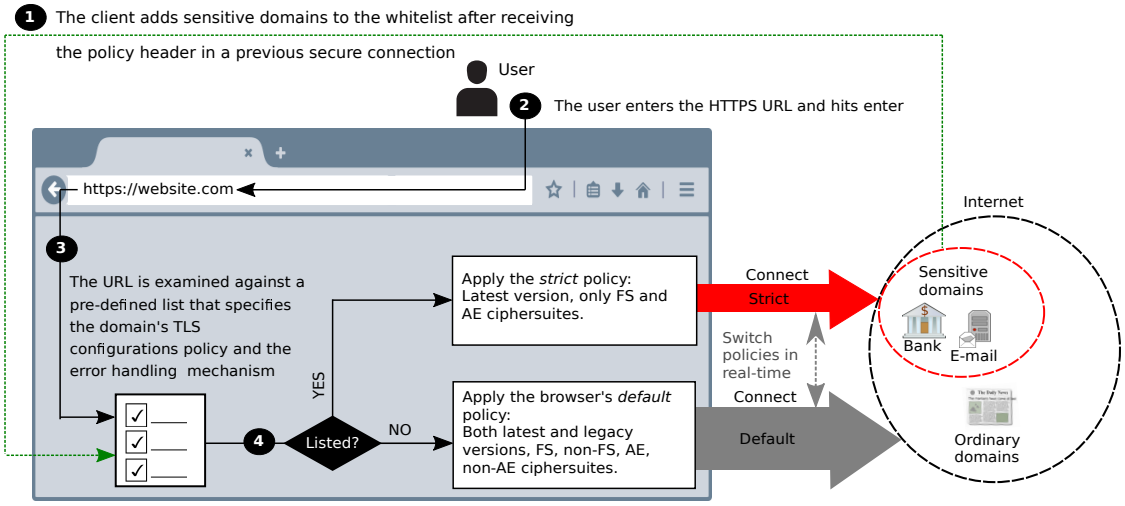


Figure 7.9: System overview of strict TLS policy using header-based whitelisting.

7.4.2 Header-based

The second method we examine to enforce fine-grained TLS configurations (versions and ciphersuites) is the header-based technique. In what follows, we describe the system.

7.4.2.1 System Overview

As depicted in Figure 7.9, in this method, unlike the agent-based method, the domains are added automatically by the server through a specific HTTP header that is sent to the client in an initial secure connection. Once received, the client should store this state. In subsequent connections, the client's HTTPS requests in the URL bar will be checked against these domains that sent the header (i.e. subscribed to the strict policy). If found, the client enforces the strict policy with blocking error handling mechanism. Otherwise, the default system policy is enforced.

7.4.2.2 System Details

The header-based system is similar to the agent-based system which we explained in Section 7.4.1.2 except in the domains' subscription method and the error handling mechanism. Instead of repeating the whole system description, we highlight only the points where the header-based system differs from the previously explained system in Section 7.4.1.2.

Compared to the agent-based system in Section 7.4.1.2, the header-based system works as follows:

1. Pre-defined TLS configurations policies: similar to the agent-based method. There are strict and default policies. When the domain sends the header, the policy becomes strict.

2. Pre-defined domain names list: in the header-based method, the domain subscription to the policy is done by the server through an HTTP response header. This method does not require user intervention to add the domains. However, it assumes an authentic first connection that contains the HTTP header, also known as TOFU. This is a well understood prerequisite for all header-based policies such as HSTS [39], CSP [34]–[36] and HPKP [93]. Therefore, the user must make the first connection to these websites that advertise the mechanism's header using a trusted network.

3. Pre-defined error handling mechanism: in the header-based method, we automatically assign the blocking error handling mechanism for the advertising domains. Since the subscription request is coming from the server, the client's confidence in the server's TLS capabilities is high. Servers that advertise the mechanism's header must first ensure that they are capable of meeting the strict TLS policy requirements. They must be aware that their users will be blocked from reaching the server if the strict TLS configurations policy has been violated. This is a conservative approach towards highly secure connections and reduced decision making effort on users, in the same direction of the HSTS policy [39].

4. HTTP observers: the header-based mechanism does not need the "HTTP before send request observer" that is explained in Section 7.4.1.2. Instead, it employs the following observer.

(a) HTTP response header observer: this observer examines every HTTP response header against a pre-defined header that we name the "strict-transport-security-config". A server that wishes to subscribe to our strict TLS policy must send this header in its HTTP response. Upon receiving this, the browser interprets this as a request to add the domain to the strict configurations policy whitelist, with a blocking error handling mechanism. Ideally, such headers also contain a maximum age parameter that specifies an age after which the header is expired, and the server needs to resubscribe through the next header (in our mechanism, header expiration implies removing the domain from the whitelist). For simplicity, in our proof of concept, our header consists of a name field only, without any fine-grained header parameters such as the maximum age.

(b) HTTP error observer: the HTTP error observer is similar to what we explained in Section 7.4.1.2.

We validate the header-based system through a proof of concept implementation, such as a Firefox browser extensions.

7.4.3 DNS-based

The third method we examine to enforce fine-grained TLS configurations (versions and ciphersuites) is DNS-based technique. In what follows, we describe the system.

7.4.3.1 System Overview

As depicted in Figure 7.10, in this method, first, the domains subscribe to the policy by the domain owner from the DNS through a specific DNS RR which we call the "DNS-based Strict TLS Configurations" (DSTC) record. Once added, the client retrieves the record when it queries the DNS to retrieve the IP address of the domain before the connection is established. After the client receives and validates the DSTC record, it enforces the strict TLS policy. Otherwise, if the record is not present or invalid, the default system policy is enforced.

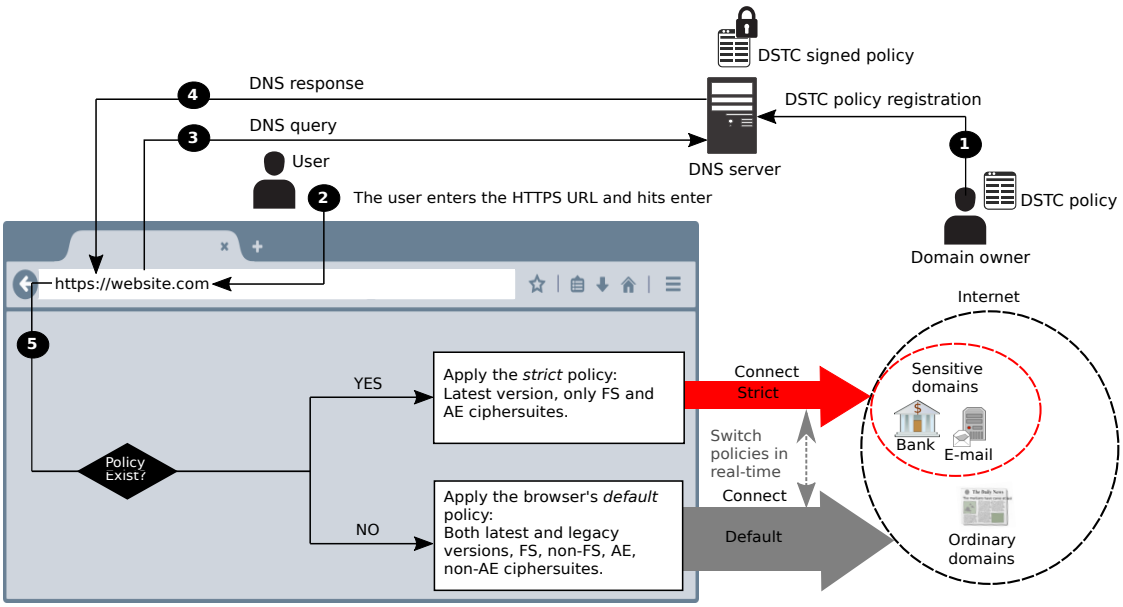


Figure 7.10: System overview of strict TLS policy using DNS-based whitelisting.

7.4.3.2 System Details

1. DSTC policy syntax: in what follows, we describe each directive used in the DSTC policy syntax. Figure 7.11 shows an example of an ideal DSTC record in a DNS zone file. We now describe the DSTC records in more detail:

   (a) `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 as a DSTC policy record by the receiving client. This directive value must be set to "DSTC".

   (b) `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 record must be the effective one. This directive value takes a date in a dd-mm-yyyy format.

   (c) `validTo:` specifies the DSTC policy expiration date. It indicates the validity of the policy. This directive value takes a date in a dd-mm-yyyy format.

```
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"
```

Figure 7.11: An example of a DSTC record in the DNS for the domain "tls12".

(d) `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.

(e) `includeSubDomain:` specifies whether the policy should be enforced to subdomains or not. It takes either 0 to disable the option or 1 to enable it.

(f) `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 it exists.

(g) `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 to the advertised policy to the domain owner.

2. Policy registration:

(a) The policy must be defined by the domain owner according to the policy syntax that is explained above.

(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.

3. 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 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 the verification passes. Otherwise, the policy remains default.

4. 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 offered TLS versions and ciphersuites parameters in the ClientHello message, the client connects to the server. Figure 7.10 illustrates the DSTC system and the actors involved.

7.4.3.3 Feasibility

To test the feasibility of our DSTC concept, we implement a proof of concept for the mechanism. On a machine equipped with 16 GB Random Access Memory (RAM) and Intel Core i7 2.6 GHz processor, that runs Windows 10 (64-bit) Operation System (OS), we build a virtual private network with a virtual host-only Ethernet adapter using VirtualBox [145]. 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 [146] on Ubuntu 16.04 (64-bit) 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 [147]. 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 [148] and Python's TLS/SSL library [149] 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 proof of concept we assume the highest version of TLS is TLS 1.2 (TLS 1.3 was not standardised yet at the time of the prototype implementation). 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 7.2, 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.

7.4.3.4 Performance

To gain an insight into the computational cost that our mechanism adds over an ordinary TLS connection, based on scenario 1 in Table 7.2 (assuming no cached

Table 7.2: Test case scenarios carried from our Python DSTC-supported client to TLS servers and the effect of DSTC (✓denotes DSTC registered domain and ✗denotes an unregistered domain) on the TLS handshake (✓denotes successful and ✗denotes failed).

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

Table 7.3: The mechanism's computational overhead in milliseconds.

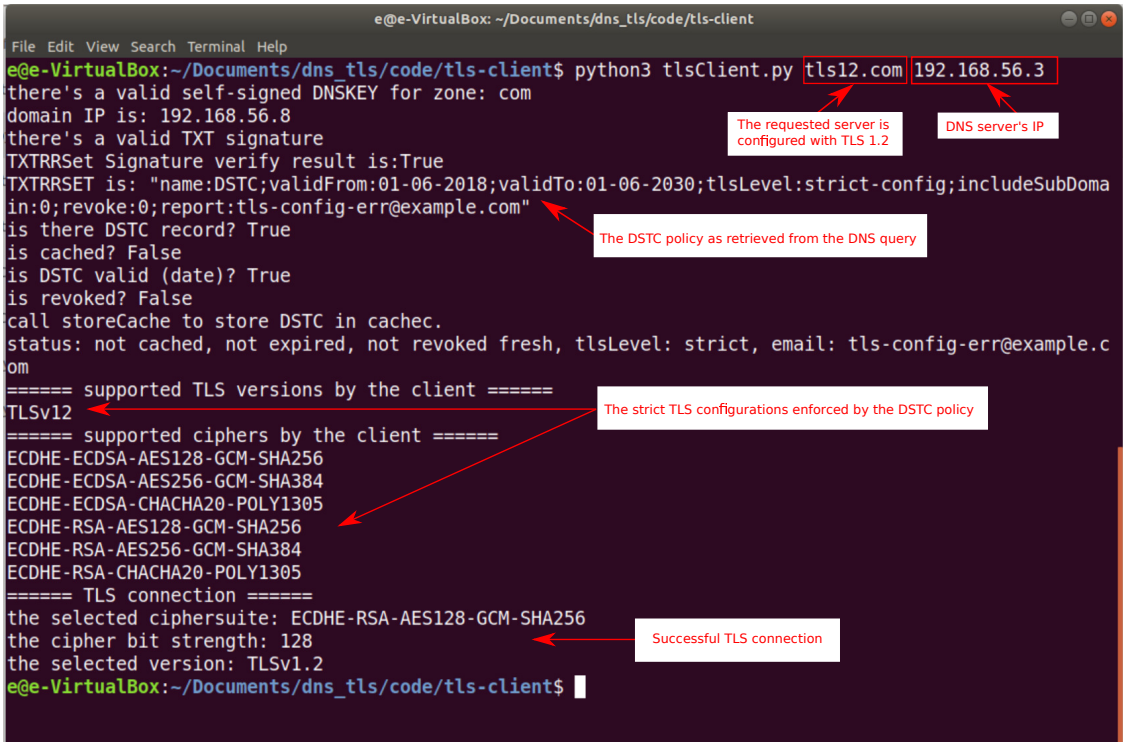
No.	Function	Max.	Min.	Avg.
1	`SigVerify`	1.40	0.63	0.72
2	`QueryVerify`	4.99	2.74	3.09
3	`Enforce`	0.86	0.38	0.41
4	`All 3 functions`	6.10	3.23	3.58

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 7.3 presents the measurements using the processor timer in Python's 3.6 "time" module [150], which is a 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. The computational overhead is about 43.87% additional overhead on the TLS socket connection. We acknowledge this additional overhead, and that it can be a limitation in limited resource devices such as mobile devices.

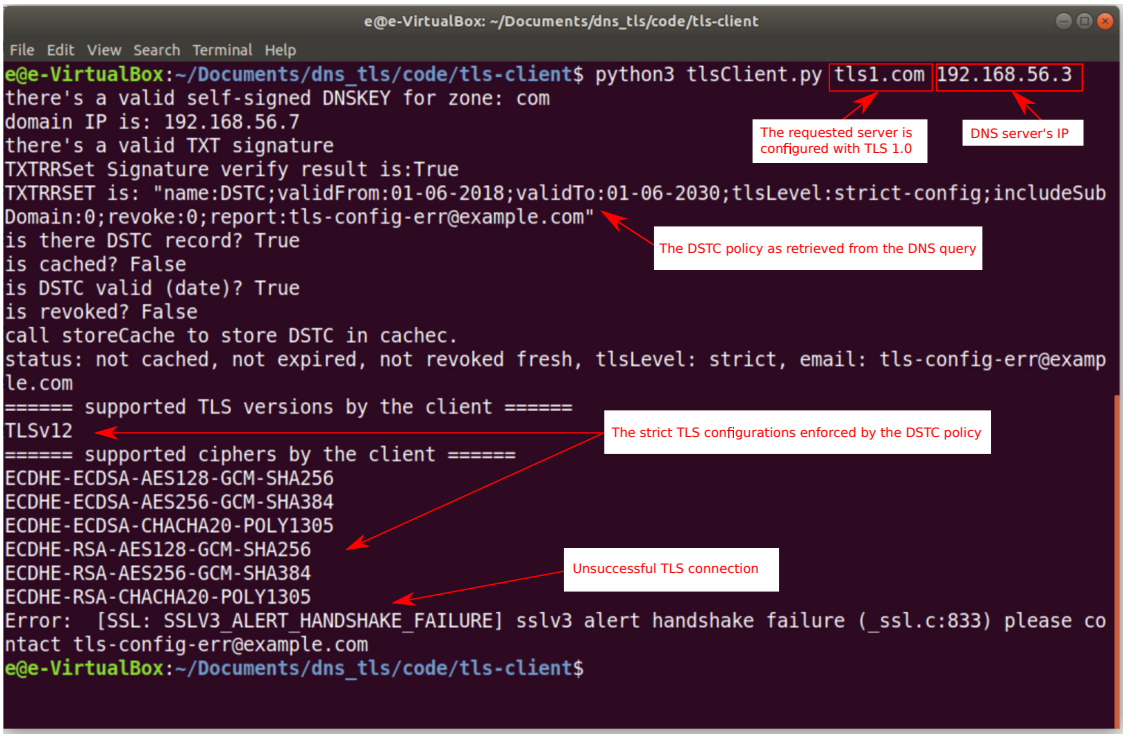
Figure 7.12 shows two screenshots for DSTC prototype with a compliant TLS server (TLS 1.2) vs. a non-compliant TLS server (TLS 1.0).

7.5 Limitations

We now list the limitations of the three presented techniques. In Section 7.4.1 and Section 7.4.2, in our proof of concept implementation, there are some limitations:



(a) DSTC prototype with a compliant TLS server (TLS 1.2).



(b) DSTC prototype with a a non-compliant TLS server (TLS 1.0).

Figure 7.12: DSTC prototype with a compliant TLS server (TLS 1.2) vs. a non-compliant TLS server (TLS 1.0).

first, we used Add-on SDK (which is deprecated starting from Firefox 57.0) to perform the configurations rewriting which is not supported in the newer development framework, Webextensions API. However, our present purpose is to demonstrate the feasibility of the concept. The configurations rewriting will not represent an issue if the mechanism is implemented at the browser source code level. Second, we do not consider measuring the performance at this stage. It can be measured if the mechanism is implemented at the browser source code level. Third, we do not conduct a user study to measure the usability as this is out of our scope. As stated earlier, our scope in this chapter is to propose and test the feasibility of the concept.

7.6 Summary

In this chapter we examined three methods to provide fine-grained TLS configurations, namely the protocol versions and ciphersuites. The main idea is based on a form of whitelisting of domain names through three different subscription methods: agent-based, header-based, and DNS-based. We validated our concepts through prototypical implementations. The agent-based and DNS-based methods enforce the policy before the connection. However, the former requires user involvement while the latter requires DNSSEC and domain administrators' involvement to set it up. The header-based does not require the user's involvement, but requires TOFU and domain administrators' involvement to set it up. Finally, the agent-based method provides protection against the discriminatory server attacker model (in addition to the man-in-the-middle attacker model), since the knowledge is provided by the agent or the user, and not the server.

It is worth noting that the solutions examined in this chapter are meant to be general solutions for version and ciphersuite downgrade attacks. Chapter 4 somewhat anticipates the results of this chapter. Chapter 4's BEFS solution tries to avoid the server administrators' involvement (as in the DNS-based and the header-based solutions) because our analysis in Chapter 4 considers semi-trusted servers in the discriminatory attacker model. It also avoids the overhead imposed on users to

build a whitelist as in the agent-based solution presented in this chapter. BEFS is meant to be deployed by agent vendors such as web browsers (potentially as an optional feature such as the Firefox's "private" and the Google's Chrome "incognito" modes) without server administrators' or users' involvement in its deployment.

Prior knowledge is a powerful concept as it provides clients with confidence to make strict security decisions. However, delivering knowledge about servers' actual configurations to clients in an authentic, usable, and privacy-preserving fashion is not trivial, especially if we add resilience against discriminatory attackers to our aims. Each method presented in this chapter has its advantages and disadvantages. One promising solution which we leave for future work is prior knowledge about servers' configurations through a public log or a trusted third party (see Section 8.4). This method eliminates the need to trust servers in every negotiation, which provides resilience against the discriminatory attacker model. It also gives clients more confidence on their decisions than in the agent-based and the BEFS solutions (discussed here and in Chapter 4 respectively), since the knowledge is based on the servers' actual configurations. This in turn reduces the overhead imposed on users both in the deployment process, and in the decision making process on warning messages. That is, with more confidence, clients can behave more strictly, with minimal user involvement. However, privacy and other third parties' issues are challenges that needs to be overcome.

8

Discussion, Summary, and Future Work

Contents

8.1 Possible Reasons for Security Misconfigurations and Inconsistency 151
8.2 Summary of Results 154
8.3 Generalisation 157
8.4 Future Work 157

8.1 Possible Reasons for Security Misconfigurations and Inconsistency

In Chapter 4 we found servers that select non-FS key exchange despite their support for FS key exchange. In Chapter 5 we found inconsistency between plain-domains as opposed to their equivalent www-domains. In Chapter 6 we found HTTPS security inconsistencies, especially at the application layer, among requests from different regions to the same domain names. In this section, we discuss possible reasons that may be behind these observed phenomena. Figure 8.1 summarises them at a high level with further levels at the server administrators' motives.

The first reason for servers' security misconfigurations and inconsistency can be linked to servers' administrators choice. The choice could be either accidental or

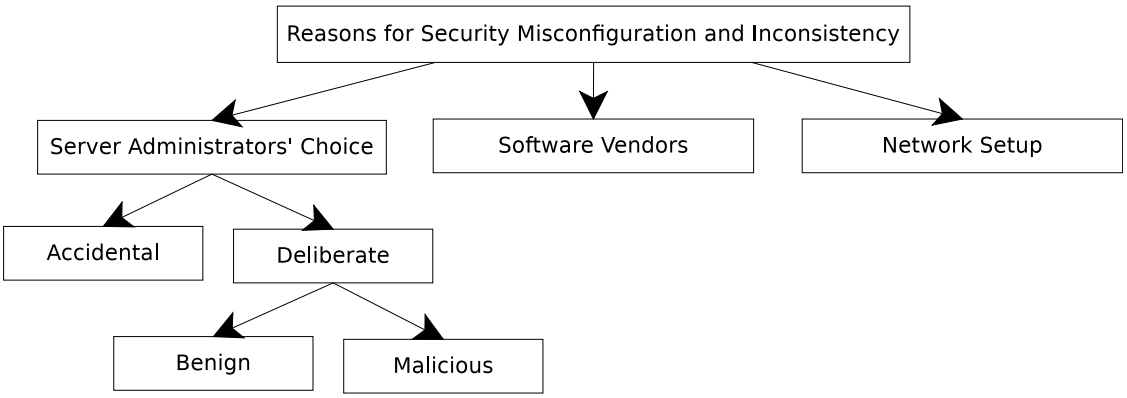


Figure 8.1: A high level overview of the possible reasons for servers' security misconfigurations and inconsistency with further levels at the server administrators' motives.

deliberate. To elaborate using examples from previous studies, in terms of accidental choice, a user study by Tiefenau et al. [8] on TLS and certificates' configurations showed that 3 out of 16 participants in the traditional CA setup group who have been asked to enable FS have misconfigured or missed FS configurations. In terms of deliberate choice, we divide this into two further possibilities: benign (justifiable reasons) or malicious reasons (e.g. discrimination). In what follows, we explain them further:

1. In terms of benign reasons, a user study by Fahl et al. [151] on misconfigured "non-validating" certificates found that in two thirds of the cases, web administrators deliberately chose to configure their servers with non-validating certificates, and provided reasons for deliberately using non-validating certificates, while in one third of the cases, this was due to accidental misconfiguration. Looking at the FS misconfigurations that we found in Chapter 4, deliberate choice of non-FS can be justified by old, but no longer true, advice regarding lower performance overhead by non-FS key exchange algorithms compared to FS ones. However, Huan et al. refuted this assumption where they showed that using ECDHE outperforms RSA [58]. Besides performance arguments, another reason for benign deliberate misconfigurations choice could be non-technical, e.g. for financial or business reasons, not to lose online customers with legacy devices to a competitor for example. Another business reason is related to

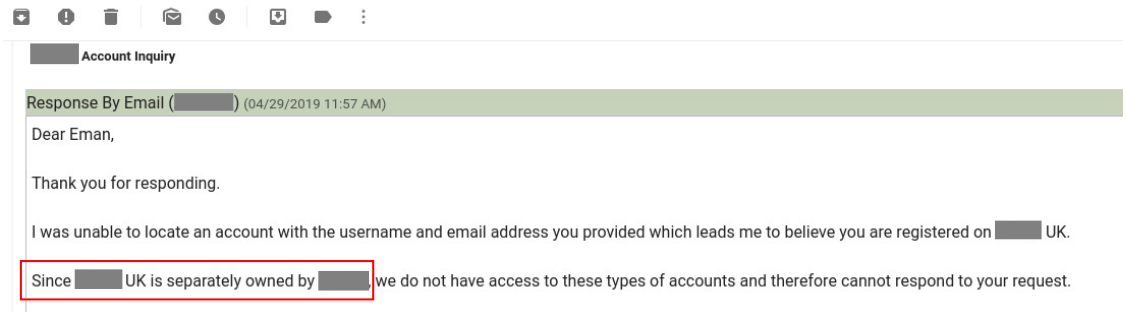


Figure 8.2: An email showing a case of possible reason for inconsistency due to business reasons.

the organisational structure, for example, in what is known as "franchising", where a branch of a big company is operated by another smaller company in another country, using the big company's name (see Figure 8.2 for a real example we found in our consistency study in Chapter 6). In such cases, with a lack of standards that govern all branches, they may follow different security standards. Additionally, different countries may have different standards which may result in inconsistent security among distributed content of the same domain.

2. On the other hand, in terms of administrators' deliberate choice for malicious reasons, this could be for discrimination as we described in the discriminatory attacker model in Item 3 in Section 4.5.2.3 (Figure 4.4) and Section 6.5.2. That is, a malicious administrator or an organisation wants to provide lower security guarantees to some users, e.g. based on their regions, client vendors, and network type.

The second reason for servers' security misconfigurations and inconsistency could be linked to a buggy software implementation for servers' software, load balancers, or security protocol libraries (TLS in our case). This assumption is supported by a response from a user (with a username "adrianmsmith") to a news post in [152] discussing our paper [10] (presented in Chapter 5). In particular, the response is regarding the insecure redirection issue we pointed in our paper, where the user highlighted his/her observations of insecure plain-HTTP redirection injection

without the administrator's knowledge in some cases, seemingly due to a buggy software implementation of some load balancers. The user posted:

> One problem I've found is that if you have a redirect from HTTP to HTTPS, then you don't notice an extra redirect from HTTPS to HTTP (e.g. redirect from HTTPS plain domain to HTTP www, which then in the next request get upgraded from HTTP www to HTTPS www) [152].

The user had previously posted a blog post (see [153]) describing the buggy software issue in more detail. More dangerously, such plain-HTTP redirection happens in the background and cannot be easily detected.

Finally, the third reason for security misconfigurations and inconsistency could be due to network setups. For example, content deliver networks (CDN) and load balancing techniques.

Having speculated on several possible realistic reasons for the observed misconfigurations and inconsistency, it should be noted that confirming any of them in our findings requires further studies and data, to identify the exact reasons which are outside our scope.

8.2 Summary of Results

This thesis aims to introduce and investigate negotiation transparency and consistency as needed properties in configurable protocols. We focus our study on one of the most widely used protocols to date: the TLS protocol. In what follows, we list the key contributions.

- We began our research by creating a taxonomy of downgrade attacks in the TLS protocol and application protocols using TLS. The taxonomy is presented in Chapter 2.

- We introduced three negotiation models based on a new notion which we call the negotiation power: the server-dominant, client-dominant, and equillibrium models. The models are presented in Chapter 2.

- In Chapter 4, we investigated negotiation transparency. We introduced a novel attacker model which we call the discriminatory model. We then conducted an empirical study on the FS property in the TLS protocol on over 10 million server addresses. Our results confirm the existence of servers that support FS but do not select it. There are many possible reasons for this behavior, which we discussed in more detail in the discussion in Section 8.1. Most importantly, the results prove the realism of our attacker model. That is, if misconfigurations can go unnoticed in mainstream clients today, without further enhancement mechanisms, discrimination can go unnoticed too. We also discussed possible paths towards FS. While obviously the ultimate solution to version and ciphersuite downgrades is deprecating older versions and ciphersuites, experience shows that this is a difficult choice in real world setting, especially in widely used protocols such as TLS. Therefore, we proposed mechanisms to enforce FS and AE properties through best effort approaches.

- We investigated negotiation consistency through two case studies. In each study, we examined servers' responses to multiple connections that differ in subtle variables that are not expected to affect the security guarantees. In both cases, we prove the existence of inconsistencies.

  - In the first study in Chapter 5, we showed that www-domains tend to have better TLS security than their equivalent plain-domains. This phenomenon is more prevalent in top domains than in random domains. We also found that most of these websites tend to utilise redirections from plain-domains to www-domains. However, a considerable percentage of these redirections contain an insecure, i.e. one or more plain-HTTP intermediate links. These results inform the Internet measurement-based research community, domains (servers) administrators, developers, and users alike.

  - In the second study in Chapter 6, we examined servers' responses to requests from different regions. We found that HTTPS security

inconsistencies at the application layer are higher than those at the transport layer. We also found that HTTPS security inconsistencies are strongly related to URLs and IPs diversity among regions, and to a lesser extent to the presence of redirections. Further manual inspection showed that there are several reasons behind URLs diversity among regions such as downgrading to the plain-HTTP protocol, using different subdomains, different TLDs, or different home page documents. Furthermore, we found that downgrading to plain-HTTP is related to websites' regional blocking. We also provided attack scenarios that showed how an attacker can benefit from HTTPS security inconsistencies, and introduced a new attack scenario which we call the region confusion attack. Finally, based on our observations, we proposed some recommendations including the need for testing tools for domain administrators and users that help to mitigate and detect regional domains' inconsistencies, standardising regional domains format with the same-origin policy (of domains) in mind, standardising secure URL redirections, and avoiding redirections whenever possible.

- In Chapter 7, we studied the concept of prior knowledge as a means to defeat downgrade attacks. We proposed three different mechanisms to achieve this concept in the context of TLS versions and ciphersuites downgrade. We validated our proposals though proof of concept prototypical implementations.

Our work investigated transparency and consistency in configurable protocols, which remains largely overlooked. Considering methods for enabling verifiability to clients receiving security related decisions from servers, whether it be a protocol version, algorithm, security policy, or subtle key-related parameters, is a worthwhile endeavour, and can help increase trust between communicating parties.

8.3 Generalisation

We focused our thesis on the TLS protocol as it is one of the most important and widely used configurable protocols. Indeed, some of the attacks and concepts we discussed in this thesis, namely: downgrade attacks, negotiation models, the discriminatory attacker model, and prior knowledge as a means to defeat downgrade attacks, are generic concepts that are applicable to a wide variety of protocols that include parameters negotiations. For example, in Bhargavan et al. [24], downgrade attacks have been found in the SSH, IPSec IKE, and ZRTP protocols. Additionally, downgrade attacks have been found in mobile communication network protocols [28]. However, with regards to generalising the prior knowledge concept to other protocols, the methods of populating prior knowledge depend on the protocol details. While we examined the agent-based (e.g. web browser), header-based (HTTP headers), and DNS-based methods, other protocols might need different methods based on the parties involved.

In terms of parameters, while we have focused on TLS security related parameters such as the protocol version and ciphersuites, downgrade attacks can target more subtle security parameters. For example, a discriminatory attacker in the server can produce deliberately weak (e.g. predictable) keys for a third party's advantage. Additionally, the parameters can be non-security parameters such as performance or service level agreement parameters. For example, in [154] Shaik et al. identified what they call the "bidding down attacks" in 4G and 5G mobile communications protocols. This downgrade attack targets the device negotiated capabilities, and downgrades the device's data rate from 27 Mbps to 3.7 Mbps and denies Voice Over LTE (VoLTE) services from LTE subscribers.

8.4 Future Work

There are some promising research questions that resulted from this thesis work which we leave for future work. In what follows, we list the most relevant ones:

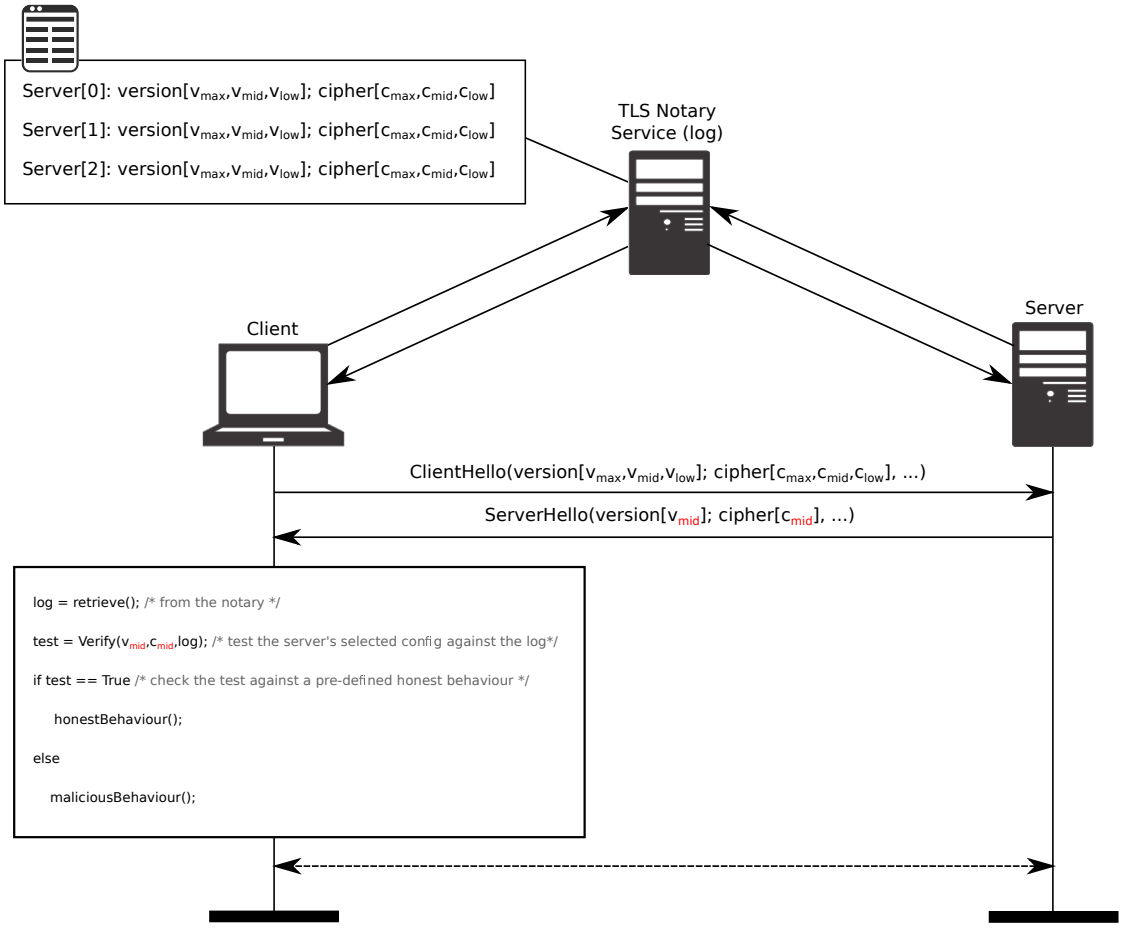


Figure 8.3: An overview of transparency through public logs.

1. Negotiation transparency through public logs: the idea of providing negotiation transparency through public logs came at the very early stages of this thesis. However, public logs raise some concerns. First, there is a privacy concern. Since clients' queries to websites or domain names are sent to the log (see Figure 8.3), the log knows the clients' requested websites. This can enable client profiling. Second, there is latency, especially for websites that are not already in the log from previous requests. Third, the availability of the log is another issue. These concerns have been examined and noted in [155]. Having said that, the concept is still intriguing and can provide transparency as a property. In future, we can study the methods used to overcome such challenges in related problems such as certificate revocation lists and the CT project. Additionally, it could be possible that TLS configurations such as the

protocol version and ciphersuites are incorporated in servers' TLS certificates.

2. Negotiation transparency through new logic: can we think of a better logic to perform parameters negotiation in protocols? It is a question that needs analysis to be answered and we leave it for future work. A game theoretic approach could be considered to conduct this analysis.

3. Comparing negotiation models: in Chapter 2, we introduced three negotiation models in protocols. Is there a model that provides better security than others? To answer this question we need careful analysis to compare the three models, and we leave this to future work. Again, a game theoretic approach could be considered to conduct this comparison.

Appendices

A

The TLS Protocol

Contents

A.1 TLS 1.2 Handshake Protocol 163
A.2 TLS 1.3 Handshake, Major Changes 165
A.3 The Experiments Clients' Ciphersuites 169

A.1 TLS 1.2 Handshake Protocol

We briefly describe the TLS 1.2 handshake protocol in the certificate-based unilateral server authentication mode based on the Internet Engineering Task Force (IETF) standard's specifications [15]. A detailed description of the protocol can be found in [15]. As depicted in Figure A.1, the handshake protocol works as follows:

1. 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 (v_{max_C}); the client's random value (n_C); 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 ($[e_1, ..., e_n]$).

2. Second, the server responds with a ServerHello (SH) message. This message contains: the server's selected TLS version (v_S); the server's nonce (n_S); optionally, a session identifier in case of session resumption ($session_{ID}$); the selected ciphersuite based on the client's proposed list (a_S); the selected compression method from the client's proposed list (c_S); 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 its certificate ($cert_S$), if server authentication is required. Then, if the key exchange algorithm is (EC)DHE (see [22] 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 [21] for details about the RSA algorithm). The SKE contains the server's (EC)DHE public key parameters and a signature over a hash of the nonces (n_C and n_S) 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 [156] 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.

3. Third, upon receiving the SHD, the client should verify the server's certificate and the compatibility of the server's selected parameters in the SH. After that, the client sends a ClientKeyExchange (CKE) to set the pre-master secret. The content of the CKE 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 ($\text{enc}(pk_S, pms)$) as illustrated in Figure A.2. 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 A.1. After that, both parties compute the master secret (ms) and the session keys: (k_C) for the client, and (k_S) for the server, using Pseudo Random Functions (PRF)s as follows: (kdf_{ms}) takes the pms and the nonces as input and produces the ms, while (kdf_{k}) takes the ms and nonces

as input and produces the session keys k_C and k_S. There is 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_C and k_S. 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 CF verifies the integrity of the handshake transcript (i.e. the *log*)[1]. The CF content is computed using a MAC function, which we denote it by (mac), over a hash of the handshake transcript starting from the CH up to, but not including, the CF (i.e. a MAC of *log*1 as shown in Figure A.1 and Figure A.2), using the *ms* as a key. This mac needs to be verified by the server.

4. Fourth, similar to the client, the server sends its CCS followed by a ServerFinished (SF) that consists of a MAC over a hash of the server's transcript up to this point (*log*2), which also needs to be verified by the client.

5. 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_C and k_S. If "False Start" [157] is enabled, the client can send data just after its CF, and before it verifies the SF.

A.2 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 a certificate-based unilateral server-authentication mode. A full description of the of the standard specifications of TLS 1.3 can be found in [16]. Figure A.3 illustrates the Hello messages in TLS 1.3, where the TLS version and algorithms are negotiated. In what follows we list these major changes:

[1]The term *log* is adopted from [24]

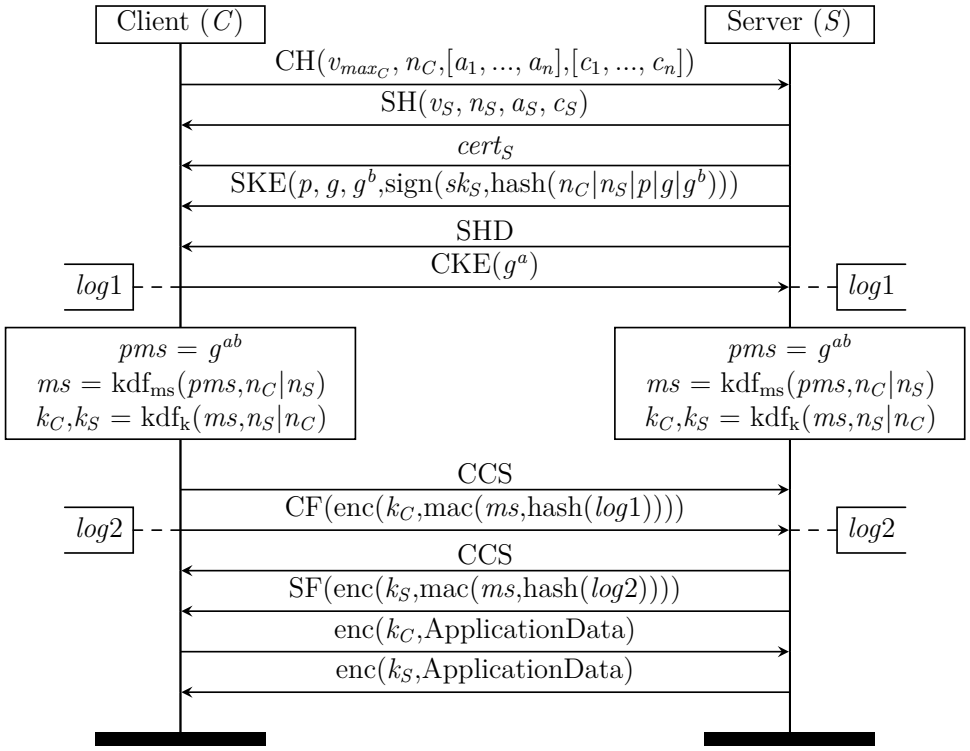


Figure A.1: Message sequence diagram for TLS 1.2 with (EC)DHE key exchange.

1. One of the first changes in TLS 1.3 is prohibiting all known weak and unrecommended cryptographic algorithms such as RC4 for symmetric encryption, and RSA and static DH for key exchange. In addition, TLS 1.3 enforces 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) 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 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.

2. The CH message in TLS 1.3 has major changes. First, in terms of parameters,

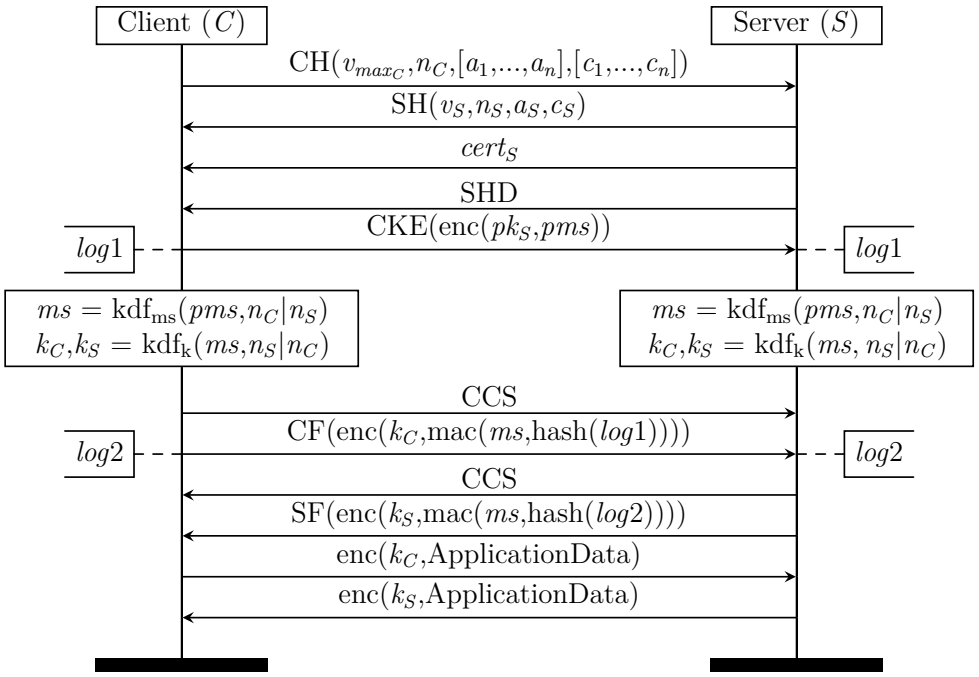


Figure A.2: Message sequence diagram for TLS 1.2 with RSA key exchange.

the following parameters have been deprecated (but still included for backward compatibility): the maximum supported TLS version ($vmax_C$) 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 is sent as a single byte set to zero (c_C). In addition, unlike TLS 1.2 where extensions are optional, in TLS 1.3, the CH extensions are mandatory and must at least include the "supported_versions" extension. Second, in terms of behaviour, the server can optionally respond to a CH 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_C, g^c),...]) which is a list of "key_share" entries ("KeyShareEntry") ordered by preference, but is supported in the client's "supported_groups" extension ([...,G_S,...]). The HRR can also be sent if the

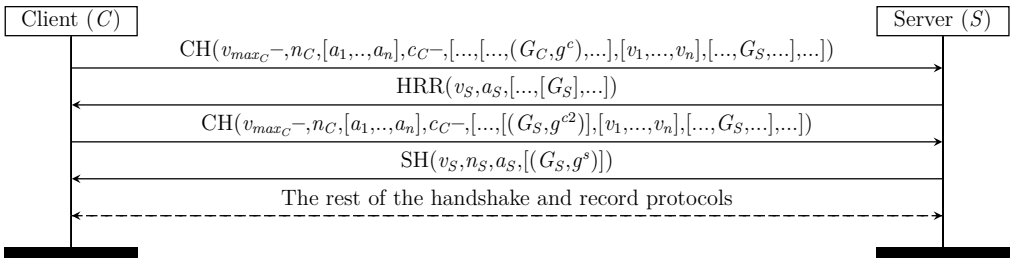


Figure A.3: Message sequence diagram for TLS 1.3 Hello messages with DHE key exchange and HRR. Deprecated parameters that are included for backward compatibility are are followed by the – symbol.

client has not sent any "key_share". After the HRR, the client sends a second CH with the server's requested "key_share" ($[G_S, g^{c2}]$).

3. Upon receiving a CH, if the client's offered parameters are supported by the server, the server responds with an SH message. The SH has two major changes: First, unlike TLS 1.2 where the extensions field is optional, in TLS 1.3, the SH 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 scope). Second, as a version downgrade attack defence mechanism (in addition to other mechanisms), the last eight bytes of the server's nonce n_S 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 CH have been received correctly by the server. This is because the nonces are signed in the TLS 1.3 server's CertificateVerify (CV) and in the TLS 1.2 SKE as well.

4. Finally, the TLS 1.2 SKE 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 SKE in TLS 1.2 to authenticate the server's key parameters is now sent in a new message, namely the server's CV which is sent after the $cert_S$ message. Most importantly, the signature in the server's CV is computed over

a hash of the full transcript from the Hello messages up to the $cert_S$, and not only over the key parameters as in TLS 1.2 SKE. The signature over the full transcript provides protection against downgrade attacks that exploit the lack of ciphersuite authentication in the SKE as demonstrated in [1] and [2].

A.3 The Experiments Clients' Ciphersuites

In this section we list the ciphersuites we used to configure our experiments' clients that have been used in Chapters 4 to 6. In Chapter 4 we used pre-TLS 1.3 clients in three different settings: default, FS-only, and FS+AE-only. In Chapter 5 we used two clients, one was configured with pre-TLS 1.3 (default ciphersuites) and the other with TLS 1.3 (default ciphersuites). In Chapter 6, we used TLS 1.3 (default ciphersuites) client configurations. Note that, similar to the Chrome browser, our TLS 1.3 clients also support pre-TLS 1.3 ciphersuites for backward compatibility.

Table A.1: The TLS clients' ciphersuits list that we used in our experiments. Note the client type and client version since we configured various client types and versions throughout our experiments. Note that, similar to the Chrome browser, our TLS 1.3 clients also support pre-TLS 1.3 ciphersuites for backward compatibility.

Ciphersuite	Client Type	Client Version
TLS_AES_128_GCM_SHA256	Default; FS+AE-only	TLS 1.3
TLS_AES_256_GCM_SHA384	Default; FS+AE-only	TLS 1.3
TLS_CHACHA20_POLY1305_SHA256	Default; FS+AE-only	TLS 1.3
ECDHE_ECDSA_AES128_GCM_SHA256	Default; FS-only; FS+AE-only	pre-TLS 1.3
ECDHE_RSA_AES128_GCM_SHA256	Default; FS-only; FS+AE-only	pre-TLS 1.3
ECDHE_ECDSA_AES256_GCM_SHA384	Default; FS-only; FS+AE-only	pre-TLS 1.3
ECDHE_RSA_AES256_GCM_SHA384	Default; FS-only; FS+AE-only	pre-TLS 1.3
ECDHE_ECDSA_CHACHA20_POLY1305	Default; FS-only; FS+AE-only	pre-TLS 1.3
ECDHE_RSA_CHACHA20_POLY1305	Default; FS-only; FS+AE-only	pre-TLS 1.3
ECDHE_RSA_AES128_SHA	Default; FS-only	pre-TLS 1.3
ECDHE_RSA_AES256_SHA	Default; FS-only	pre-TLS 1.3
AES128_GCM_SHA256	Default	pre-TLS 1.3
AES256_GCM_SHA384	Default	pre-TLS 1.3
AES128_SHA	Default	pre-TLS 1.3
AES256_SHA	Default	pre-TLS 1.3
DES_CBC3_SHA	Default	pre-TLS 1.3

B

MySQL Queries Examples

Contents

B.1 Examples for Chapter 4 MySQL Queries 171
B.1.1 Description of Tables 171
B.1.2 MySQL Queries 173
B.2 Examples for Chapter 5 MySQL Queries 173
B.2.1 Description of Tables 174
B.2.2 MySQL Queries 175
B.3 Examples for Chapter 6 MySQL Queries 176
B.3.1 Description of Tables 176
B.3.2 MySQL Queries 178

B.1 Examples for Chapter 4 MySQL Queries

We first describe the tables and the fields included in the queries. Notes that these are few examples, and there are more tables and queries used in the chapters' experiments.

B.1.1 Description of Tables

In this section, Table B.1 and Table B.2 describe the scheme's tables and their fields that are used in the examples in Appendix B.1.2.

Table B.1: The "top1m" scheme's tables.

Table name	Description
plain_default	The results of the default ciphersuites handshake in the inspection phase
plain_fs	The results of the FS-only ciphersuites handshake in the inspection phase

Table B.2: The "plain_default" and "plain_fs" tables' fields.

Column name	Type	Description
ciphersuite	varchar(500)	Ciphersuite name

B.1.2 MySQL Queries

In this section, we provide MySQL queries examples.

Listing B.1: MySQL script to query servers that select non-FS (stable) in the default ciphersuites handshake from the inspection results.

```
/* This MySQL query is to find servers that prefer non-FS (stable)
    from the inspection results */
/*================================================================*/
SELECT `plain_default`.`ciphersuite` as 'Ciphersuite'
FROM `top1m`.`plain_default`
/* The condition (do not start with "ecdhe").
Note that our scan is for pre-TLS~1.3. No TLS 1.3 ciphersuites
   here. */
WHERE LOWER(`plain_default`.`ciphersuite`) NOT LIKE 'ecdhe%';
```

Listing B.2: MySQL script to query servers that select non-FS (stable) in the default handshake, but support FS in the FS-only ciphersuites handshake from the inspection results.

```
/* This MySQL query is to find servers that prefer non-FS (stable)
    from the inspection results BUT turned to support FS */
/*================================================================*/
/* Simply query those that provided non-FS in the first default
   handshake,
and responded with FS in the FS-only ciphersuites handshake, which
    are stored in the `plain_fs` table */
SELECT `plain_fs`.`ciphersuite` as 'Ciphersuite'
FROM `top1m`.`plain_fs`;

/* Or, another longer query that gives same results. Query two
   tables: the default handshake and FS-only handshake */
/* SELECT  `plain_default`.`ciphersuite` as 'Default ciphersuite',
  `plain_fs`.`ciphersuite` as 'After FS-only ciphersuite
   enforcement' */
/* From joint tables */
/* FROM `top1m`.`plain_default` JOIN `top1m`.`plain_fs`
  ON `plain_default`.`domain` = `plain_fs`.`domain`  */
/* The conditions are stated here. Select non-FS but support FS */
/* WHERE LOWER(`plain_default`.`ciphersuite`) NOT LIKE 'ecdhe%'
  AND LOWER(`plain_fs`.`ciphersuite`) LIKE 'ecdhe%'; */
```

B.2 Examples for Chapter 5 MySQL Queries

We first describe the tables and the fields included in the queries. Notes that these are few examples, and there are more tables and queries used in the chapter's experiments.

B.2.1 Description of Tables

In this section, Table B.3 and Table B.4 describe the scheme's tables and their fields that are used in the examples in Appendix B.2.2.

Table B.3: The "www_vs_plain_v2_scan2" scheme's tables.

Table name	Description
tls13_plain_top1m	The results of the TLS 1.3 handshake for plain-domains in the top domains dataset
tls13_www_top1m	The results of the TLS 1.3 handshake for www-domains in the top domains dataset

Table B.4: The "tls13_plain_top1m" and "tls13_www_top1m" tables' fields.

Column name	Type	Description
domain	varchar(500)	The request's domain name
version	varchar(10)	TLS version
certExpired	varchar(10)	The result of certificate expiration check Either "true" (expired) or "false" (non-expired)

B.2.2 MySQL Queries

In this section, we provide MySQL queries examples.

Listing B.3: MySQL script to query plain-domains and their equivalent www-domains that have different TLS versions.

```
/* This MySQL query is to find plain-domains and their equivalent
   www-domains that have different TLS versions */
/*=============================================================*/
SELECT `tls13_plain_top1m`.`version` as 'Plain TLS version',
`tls13_www_top1m`.`version` as 'www TLS version'
/* Query the two tables: plain-domain scan results and www-domain
   scan results */
FROM `www_vs_plain_v2_scan2`.`tls13_plain_top1m`,
  `www_vs_plain_v2_scan2`.`tls13_www_top1m`
/* The condition is stated here
The www-domain equals its corresponding plain-domain,
and has different TLS version */
WHERE CONCAT('www.',`tls13_plain_top1m`.`domain`) =
  `tls13_www_top1m`.`domain`
AND `tls13_plain_top1m`.`version` != `tls13_www_top1m`.`version`;
```

Listing B.4: MySQL script to query plain-domains that have expired certificates but non-expired certificates in their equivalent www-domains.

```
/* This MySQL query is to find plain-domains that have expired
   certificates but non-expired certificates in their equivalent
   www-domains */
/*=============================================================*/
SELECT `tls13_plain_top1m`.`certExpired`
  as 'Plain expired cert. check',
`tls13_www_top1m`.`certExpired`
  as 'Www expired cert. check'

/* Query the two tables: plain-domain scan results and www-domain
   scan results */
FROM `www_vs_plain_v2_scan2`.`tls13_plain_top1m`,
  `www_vs_plain_v2_scan2`.`tls13_www_top1m`

/* The condition is stated here
The www-domain equals its corresponding plain-domain,
and the plain-domain has expired cert,
and www-domain has non-expired cert. */
WHERE CONCAT('www.',`tls13_plain_top1m`.`domain`) = `
   tls13_www_top1m`.`domain`
  AND `tls13_plain_top1m`.`certExpired` = 'true'
  AND `tls13_www_top1m`.`certExpired` = 'false';
```

B.3 Examples for Chapter 6 MySQL Queries

We first describe the tables and the fields included in the queries. Notes that these are few examples, and there are more tables and queries used in the chapter's experiments.

B.3.1 Description of Tables

In this section, Tables B.5 to B.8 describe the scheme's tables and their fields that are used in the examples in Appendix B.3.2.

Table B.5: The "transparency_5_regions_top250k" scheme's tables. Tables prefixed by "small_" are smaller versions of the non-prefixed tables (i.e. with fewer fields, but contains the same values of the non-prefixed ones), created for performance issues in complex queries.

Table name	Description
[small_]redirection_london_top	The results of the redirection scan from the UK.
[small_]redirection_mumbai_top	The results of the redirection scan from IN.
[small_]redirection_saopaulo_top	The results of the redirection scan from BR.
[small_]redirection_sydney_top	The results of the redirection scan from AU.
[small_]redirection_us_top	The results of the redirection scan from the US.
malicious_domains	The results of malicious domains check against Google safe browsing.
tls13_london_final_top	The results of the TLS 1.3 scan from the UK
tls13_mumbai_final_top	The results of the TLS 1.3 scan from IN
tls13_saopaulo_final_top	The results of the TLS 1.3 scan from BR
tls13_sydney_final_top	The results of the TLS 1.3 scan from AU
tls13_us_final_top	The results of the TLS 1.3 scan from the US

Table B.6: The "[small_]redirection_[region]_top" tables' fields.

Column name	Type	Description
responseURL	mediumtext	The response URL (including the protocol scheme)
host	varchar(500)	The request's URL (including https://)
responseStatusCode	varchar(10)	The response status code
hstsPresence	int(11)	The HSTS header presence. Either 1 (present) or 0 (absent)
responseDomain	varchar(500)	The response's domain (not the full URL)
compatibleInterURL	int(11)	The result of compatibility check of intermediate URLs' domain with the requested domain Either 1 (compatible) or 0 (incompatible)

Table B.7: The "malicious_domains" tables' fields.

Column name	Type	Description
domain	varchar(500)	Malicious domain name.

Table B.8: The "tls13_[region]_final_top" tables' fields.

Column name	Type	Description
version	varchar(10)	The TLS verion.
cipherShort	varchar(500)	The ciphersuite (the word "short" because this is a shortened string).
domain	varchar(500)	The requested domain name.

B.3.2 MySQL Queries

Listing B.5: MySQL script to query inconsistent responses with respect to the plain-http final URL weakness indicator.

```
/* This MySQL query is to find inconsistent responses with respect
    to plain-http final URL weakness indicator in the redirection
   scan results */
/*================================================================*/
/* Select response columns */
SELECT
`redirection_london_top`.`responseURL` as 'Final UK',
`redirection_mumbai_top`.`responseURL` as 'Final IN',
`redirection_saopaulo_top`.`responseURL` as 'Final BR',
`redirection_sydney_top`.`responseURL` as 'Final AU',
`redirection_us_top`.`responseURL` as 'Final US'

/* From joint tables */
FROM `transparency_5_regions_top250k`.`redirection_london_top`
JOIN `transparency_5_regions_top250k`.`redirection_mumbai_top`
ON `redirection_london_top`.`host` =
  `redirection_mumbai_top`.`host`
JOIN `transparency_5_regions_top250k`.`redirection_saopaulo_top`
ON `redirection_mumbai_top`.`host` =
    `redirection_saopaulo_top`.`host`
JOIN `transparency_5_regions_top250k`.`redirection_sydney_top`
ON `redirection_saopaulo_top`.`host` =
    `redirection_sydney_top`.`host`
JOIN `transparency_5_regions_top250k`.`redirection_us_top`
ON `redirection_sydney_top`.`host` =
    `redirection_us_top`.`host`

/* The conditions are stated here */
WHERE(
  /* All have valid responses (200 code) */
  (`redirection_london_top`.`responseStatusCode` LIKE '200'
    AND `redirection_mumbai_top`.`responseStatusCode` LIKE '200'
    AND `redirection_saopaulo_top`.`responseStatusCode` LIKE '200'
    AND `redirection_sydney_top`.`responseStatusCode` LIKE '200'
    AND `redirection_us_top`.`responseStatusCode` LIKE '200')

  /* One or more response have plain-http */
  AND
  (`redirection_london_top`.`responseURL` LIKE 'http://%'
    OR `redirection_mumbai_top`.`responseURL` LIKE 'http://%'
    OR `redirection_saopaulo_top`.`responseURL` LIKE 'http://%'
    OR `redirection_sydney_top`.`responseURL` LIKE 'http://%'
    OR `redirection_us_top`.`responseURL` LIKE 'http://%')

  /* But not all responses have plain-http */
  AND NOT
  (`redirection_london_top`.`responseURL` LIKE 'http://%'
    AND `redirection_mumbai_top`.`responseURL` LIKE 'http://%'
    AND `redirection_saopaulo_top`.`responseURL` LIKE 'http://%'
    AND `redirection_sydney_top`.`responseURL` LIKE 'http://%'
```

```
49      AND `redirection_us_top`.`responseURL` LIKE 'http://%')
50
51    /* And requested domains (hosts) are not in the malicious
       domains table. Note: we can query one region here because we
       query joint responses which are for the same request */
52    AND
53    (`redirection_london_top`.`host` NOT IN
54      (SELECT `malicious_domains`.`domain` FROM `
       transparency_5_regions_top250k`.`malicious_domains`)
55      AND `redirection_mumbai_top`.`host` NOT IN
56        (SELECT `malicious_domains`.`domain` from `
       transparency_5_regions_top250k`.`malicious_domains`)
57      AND `redirection_saopaulo_top`.`host` NOT IN
58        (SELECT `malicious_domains`.`domain` FROM `
       transparency_5_regions_top250k`.`malicious_domains`)
59      AND `redirection_sydney_top`.`host` NOT IN
60        (SELECT `malicious_domains`.`domain` FROM `
       transparency_5_regions_top250k`.`malicious_domains`)
61      AND `redirection_us_top`.`host` NOT IN
62        (SELECT `malicious_domains`.`domain` FROM `
       transparency_5_regions_top250k`.`malicious_domains`))
63  ); /* End WHERE clause */
```

Listing B.6: MySQL script to query inconsistent responses with respect to the No-HSTS weakness indicator.

```
1  /* This MySQL query is to find inconsistent responses with respect
       to the No-HSTS weakness indicator in the redirection scan
      results */
2  /* The header presence of each response is checked using python
      script. The results are inserted back in the DB as 0 (absent)
      or 1 (present) values */
3  /*================================================================*/
4  /* Select response columns */
5  SELECT
6  `redirection_london_top`.`responseHeader` as 'Header UK',
7  `redirection_london_top`.`hstsPresence` as 'HSTS presence UK',
8  `redirection_mumbai_top`.`responseHeader` as 'Header IN',
9  `redirection_mumbai_top`.`hstsPresence` as 'HSTS presence IN',
10 `redirection_saopaulo_top`.`responseHeader` as 'Header BR',
11 `redirection_saopaulo_top`.`hstsPresence` as 'HSTS presence BR',
12 `redirection_sydney_top`.`responseHeader` as 'Header AU',
13 `redirection_sydney_top`.`hstsPresence` as 'HSTS presence AU',
14 `redirection_us_top`.`responseHeader` as 'Header US',
15 `redirection_us_top`.`hstsPresence` as 'HSTS presence US'
16
17 /* From joint tables */
18 FROM `transparency_5_regions_top250k`.`redirection_london_top`
19 JOIN `transparency_5_regions_top250k`.`redirection_mumbai_top`
20 ON `redirection_london_top`.`host` =
21   `redirection_mumbai_top`.`host`
22 JOIN `transparency_5_regions_top250k`.`redirection_saopaulo_top`
23 ON `redirection_mumbai_top`.`host` =
24   `redirection_saopaulo_top`.`host`
25 JOIN `transparency_5_regions_top250k`.`redirection_sydney_top`
```

```
ON `redirection_saopaulo_top`.`host` =
  `redirection_sydney_top`.`host`
JOIN `transparency_5_regions_top250k`.`redirection_us_top`
ON `redirection_sydney_top`.`host` =
  `redirection_us_top`.`host`

/* The Conditions are stated here */
WHERE (
  /* All have valid responses (200 code) */
  (`redirection_london_top`.`responseStatusCode` LIKE '200'
    AND `redirection_mumbai_top`.`responseStatusCode` LIKE '200'
    AND `redirection_saopaulo_top`.`responseStatusCode` LIKE '200'
    AND `redirection_sydney_top`.`responseStatusCode` LIKE '200'
    AND `redirection_us_top`.`responseStatusCode` LIKE '200')

  /* All have https responses because security headers only make
   sense with https */
  AND (`redirection_london_top`.`responseURL` LIKE 'https://%'
    AND `redirection_mumbai_top`.`responseURL` LIKE 'https://%'
    AND `redirection_saopaulo_top`.`responseURL` LIKE 'https://%'
    AND `redirection_sydney_top`.`responseURL` LIKE 'https://%'
    AND `redirection_us_top`.`responseURL` LIKE 'https://%')

  /* One or more response have No-HSTS */
  AND
  (`redirection_london_top`.`hstsPresence` = 0
    OR `redirection_mumbai_top`.`hstsPresence` = 0
    OR `redirection_saopaulo_top`.`hstsPresence` = 0
    OR `redirection_sydney_top`.`hstsPresence` = 0
    OR `redirection_us_top`.`hstsPresence` = 0)

  /* But not all responses have No-HSTS */
  AND NOT
  (`redirection_london_top`.`hstsPresence` = 0
    AND `redirection_mumbai_top`.`hstsPresence` = 0
    AND `redirection_saopaulo_top`.`hstsPresence` = 0
    AND `redirection_sydney_top`.`hstsPresence` = 0
    AND `redirection_us_top`.`hstsPresence` = 0)

  /* And requested domains (hosts) are not in the malicious
   domains table. Note: we can query one region here because we
   query joint responses which are for the same request */
  AND
  (`redirection_london_top`.`host` NOT IN
    (SELECT `malicious_domains`.`domain` FROM `
   transparency_5_regions_top250k`.`malicious_domains`)
    AND `redirection_mumbai_top`.`host` NOT IN
      (SELECT `malicious_domains`.`domain` FROM `
   transparency_5_regions_top250k`.`malicious_domains`)
    AND `redirection_saopaulo_top`.`host` NOT IN
      (SELECT `malicious_domains`.`domain` FROM `
   transparency_5_regions_top250k`.`malicious_domains`)
    AND `redirection_sydney_top`.`host` NOT IN
      (SELECT `malicious_domains`.`domain` FROM `
   transparency_5_regions_top250k`.`malicious_domains`)
```

```
    AND `redirection_us_top`.`host` NOT IN
      (SELECT `malicious_domains`.`domain` FROM `
   transparency_5_regions_top250k`.`malicious_domains`))
); /* End WHERE clause */
```

Listing B.7: MySQL script to query inconsistent responses with respect to the non-FS weakness indicator.

```
/* This MySQL query is to find inconsistent responses with respect
    to the Non-FS weakness indicator in TLS scan results */
/* We join each redirection table with its TLS scan table.
   Therefore, we have nested joins */
/* Due to the size of tables and the number of joins,  we made a
   replica of the redirection tables (prefixed with "small_") but
   smaller (less columns) to be able to run the query. */
/*================================================================*/
/* Select response columns */
SELECT
`tls13_london_final_top`.`version` as 'TLS Version UK',
`tls13_london_final_top`.`cipherShort` as 'TLS Cipher UK',
`tls13_mumbai_final_top`.`version` as 'TLS Version IN',
`tls13_mumbai_final_top`.`cipherShort` as 'TLS Cipher IN',
`tls13_saopaulo_final_top`.`version` as 'TLS Version BR',
`tls13_saopaulo_final_top`.`cipherShort` as 'TLS Cipher BR',
`tls13_sydney_final_top`.`version` as 'TLS Version AU',
`tls13_sydney_final_top`.`cipherShort`  as 'TLS Cipher AU',
`tls13_us_final_top`.`version` as 'TLS Version US',
`tls13_us_final_top`.`cipherShort` as 'TLS Cipher US'

/* From nested joint tables (each redirection table is joint with
   its corresponding TLS scan table) */
FROM
(`transparency_5_regions_top250k`.`tls13_london_final_top`
  JOIN `transparency_5_regions_top250k`.`
   small_redirection_london_top`
  ON `small_redirection_london_top`.`responseDomain` =
  `tls13_london_final_top`.`domain`)
JOIN
(`transparency_5_regions_top250k`.`tls13_mumbai_final_top`
  JOIN `transparency_5_regions_top250k`.`
   small_redirection_mumbai_top`
  ON `small_redirection_mumbai_top`.`responseDomain` =
  `tls13_mumbai_final_top`.`domain`)
ON `small_redirection_london_top`.`host` =
  `small_redirection_mumbai_top`.`host`
JOIN
(`transparency_5_regions_top250k`.`tls13_saopaulo_final_top`
  JOIN `transparency_5_regions_top250k`.`
   small_redirection_saopaulo_top`
  ON `small_redirection_saopaulo_top`.`responseDomain` =
  `tls13_saopaulo_final_top`.`domain`)
ON `small_redirection_mumbai_top`.`host` =
  `small_redirection_saopaulo_top`.`host`
JOIN
(`transparency_5_regions_top250k`.`tls13_sydney_final_top`
```

```
  JOIN `transparency_5_regions_top250k`.`
    small_redirection_sydney_top`
  ON `small_redirection_sydney_top`.`responseDomain` =
  `tls13_sydney_final_top`.`domain`)
ON `small_redirection_saopaulo_top`.`host` =
  `small_redirection_sydney_top`.`host`
JOIN (`transparency_5_regions_top250k`.`tls13_us_final_top`
  JOIN `transparency_5_regions_top250k`.`small_redirection_us_top`
  ON `small_redirection_us_top`.`responseDomain` =
  `tls13_us_final_top`.`domain`)
ON `small_redirection_sydney_top`.`host` =
  `small_redirection_us_top`.`host`

/* The Conditions are stated here */
WHERE (
/* All have valid responses (200 code) */
(`small_redirection_london_top`.`responseStatusCode`
    LIKE '200'
  AND `small_redirection_mumbai_top`.`responseStatusCode`
    LIKE '200'
  AND `small_redirection_saopaulo_top`.`responseStatusCode`
    LIKE '200'
  AND `small_redirection_sydney_top`.`responseStatusCode`
    LIKE '200'
  AND `small_redirection_us_top`.`responseStatusCode`
    LIKE '200')

/* All have https responses */
AND (`small_redirection_london_top`.`responseURL`
    LIKE 'https://%'
  AND `small_redirection_mumbai_top`.`responseURL`
    LIKE 'https://%'
  AND `small_redirection_saopaulo_top`.`responseURL`
    LIKE 'https://%'
  AND `small_redirection_sydney_top`.`responseURL`
    LIKE 'https://%'
  AND `small_redirection_us_top`.`responseURL`
    LIKE 'https://%')

/* One or more response have non-FS (i.e. not TLS 1.3 and not
    ECDHE ciphersuite) */
AND(
((`tls13_london_final_top`.`cipherShort` NOT LIKE 'ecdhe%')
    AND (`tls13_london_final_top`.`version` NOT LIKE 'TLSv1.3'))
  OR ((`tls13_mumbai_final_top`.`cipherShort` NOT LIKE 'ecdhe%')
    AND (`tls13_mumbai_final_top`.`version` NOT LIKE 'TLSv1.3'))
  OR ((`tls13_saopaulo_final_top`.`cipherShort` NOT LIKE 'ecdhe%')
    AND (`tls13_saopaulo_final_top`.`version` NOT LIKE 'TLSv1.3'))
  OR ((`tls13_sydney_final_top`.`cipherShort` NOT LIKE 'ecdhe%')
    AND (`tls13_sydney_final_top`.`version` NOT LIKE 'TLSv1.3'))
  OR ((`tls13_us_final_top`.`cipherShort` NOT LIKE 'ecdhe%')
    AND (`tls13_us_final_top`.`version` NOT LIKE 'TLSv1.3')))

/* But not all responses have non-FS */
AND NOT (
```

```
(((`tls13_london_final_top`.`cipherShort` NOT LIKE 'ecdhe%')
    AND (`tls13_london_final_top`.`version` NOT LIKE 'TLSv1.3'))
  AND (((`tls13_mumbai_final_top`.`cipherShort` NOT LIKE 'ecdhe%')
    AND (`tls13_mumbai_final_top`.`version` NOT LIKE 'TLSv1.3'))
  AND (((`tls13_saopaulo_final_top`.`cipherShort` NOT LIKE 'ecdhe%'
   )
    AND (`tls13_saopaulo_final_top`.`version` NOT LIKE 'TLSv1.3'))
  AND (((`tls13_sydney_final_top`.`cipherShort` NOT LIKE 'ecdhe%')
    AND (`tls13_sydney_final_top`.`version` NOT LIKE 'TLSv1.3'))
  AND (((`tls13_us_final_top`.`cipherShort` NOT LIKE 'ecdhe%')
    AND (`tls13_us_final_top`.`version` NOT LIKE 'TLSv1.3')))

/* And the requested domains (hosts) are not in the malicious
   domains table. Note: we can query one region here because we
   query joint responses which are for the same request */
AND (`small_redirection_london_top`.`host` NOT IN
    (SELECT `malicious_domains`.`domain` FROM `
   transparency_5_regions_top250k`.`malicious_domains`)
  AND `small_redirection_mumbai_top`.`host` NOT IN
    (SELECT `malicious_domains`.`domain` FROM `
   transparency_5_regions_top250k`.`malicious_domains`)
  AND `small_redirection_saopaulo_top`.`host` NOT IN
    (SELECT `malicious_domains`.`domain` FROM `
   transparency_5_regions_top250k`.`malicious_domains`)
  AND `small_redirection_sydney_top`.`host` NOT IN
    (SELECT `malicious_domains`.`domain` FROM `
   transparency_5_regions_top250k`.`malicious_domains`)
  AND `small_redirection_us_top`.`host` NOT IN
    (SELECT `malicious_domains`.`domain` FROM `
   transparency_5_regions_top250k`.`malicious_domains`))
);
```

Listing B.8: MySQL script to query inconsistent responses with respect to the incompatible intermediate URLs weakness indicator that have diverse final URLs.

```
/* This MySQL query is to find inconsistent responses with respect
    to incompatible intermediate URLs weakness indicator
 that also have diverse URLs.*/
/*================================================================*/
/* Select response columns */
SELECT `redirection_london_top`.`compatibleInterURL`
  as 'inter. compatible URL UK',
`redirection_mumbai_top`.`compatibleInterURL`
  as 'inter. compatible URL IN',
`redirection_saopaulo_top`.`compatibleInterURL`
  as 'inter. compatible URL BR',
`redirection_sydney_top`.`compatibleInterURL`
  as 'inter. compatible URL AU',
`redirection_us_top`.`compatibleInterURL`
  as 'inter. compatible URL US'

/* From nested joint tables */
FROM `transparency_5_regions_top250k`.`redirection_london_top`
JOIN `transparency_5_regions_top250k`.`redirection_mumbai_top`
ON `redirection_london_top`.`host` =
```

```
  `redirection_mumbai_top`.`host`
JOIN `transparency_5_regions_top250k`.`redirection_saopaulo_top`
ON `redirection_mumbai_top`.`host` =
  `redirection_saopaulo_top`.`host`
JOIN `transparency_5_regions_top250k`.`redirection_sydney_top`
ON `redirection_saopaulo_top`.`host` =
  `redirection_sydney_top`.`host`
JOIN `transparency_5_regions_top250k`.`redirection_us_top`
ON `redirection_sydney_top`.`host` =
  `redirection_us_top`.`host`

/* The Conditions are stated here */
WHERE (
  /* All have valid responses (200 code) */
  (`redirection_london_top`.`responseStatusCode`
      LIKE '200'
    AND `redirection_mumbai_top`.`responseStatusCode`
      LIKE '200'
    AND `redirection_saopaulo_top`.`responseStatusCode`
      LIKE '200'
    AND `redirection_sydney_top`.`responseStatusCode`
      LIKE '200'
    AND `redirection_us_top`.`responseStatusCode`
      LIKE '200')

  /* One or more response have incompatible inter. URL*/
  AND
  (`redirection_london_top`.`compatibleInterURL` = 0
    OR `redirection_mumbai_top`.`compatibleInterURL` = 0
    OR `redirection_saopaulo_top`.`compatibleInterURL` = 0
    OR `redirection_sydney_top`.`compatibleInterURL`  = 0
    OR `redirection_us_top`.`compatibleInterURL`  = 0)

  /* But not all responses have incompatible inter. URL */
  AND NOT
  (`redirection_london_top`.`compatibleInterURL`  = 0
    AND `redirection_mumbai_top`.`compatibleInterURL`  = 0
    AND `redirection_saopaulo_top`.`compatibleInterURL`  = 0
    AND `redirection_sydney_top`.`compatibleInterURL`  = 0
    AND `redirection_us_top`.`compatibleInterURL`  = 0)

  /* And requested domains (hosts) are not in the malicious
   domains table. Note: we can query one region here because we
   query joint responses which are for the same request */
    AND
  (`redirection_london_top`.`host` NOT IN
    (SELECT `malicious_domains`.`domain` FROM `
   transparency_5_regions_top250k`.`malicious_domains`)
    AND `redirection_mumbai_top`.`host` NOT IN
        (SELECT `malicious_domains`.`domain` FROM `
   transparency_5_regions_top250k`.`malicious_domains`)
    AND `redirection_saopaulo_top`.`host` NOT IN
        (SELECT `malicious_domains`.`domain` FROM `
   transparency_5_regions_top250k`.`malicious_domains`)
    AND `redirection_sydney_top`.`host` NOT IN
```

```
70          (SELECT `malicious_domains`.`domain` FROM `
    transparency_5_regions_top250k`.`malicious_domains`)
71      AND `redirection_us_top`.`host` NOT IN
72          (SELECT `malicious_domains`.`domain` FROM `
    transparency_5_regions_top250k`.`malicious_domains`))
73
74    /* And have diverse final URLs */
75    AND (GREATEST(LOWER(`redirection_london_top`.`responseURL`),
76        LOWER(`redirection_mumbai_top`.`responseURL`),
77        LOWER(`redirection_saopaulo_top`.`responseURL`),
78        LOWER(`redirection_sydney_top`.`responseURL`),
79        LOWER(`redirection_us_top`.`responseURL`))
80      !=
81      LEAST(LOWER(`redirection_london_top`.`responseURL`),
82        LOWER(`redirection_mumbai_top`.`responseURL`),
83        LOWER(`redirection_saopaulo_top`.`responseURL`),
84        LOWER(`redirection_sydney_top`.`responseURL`),
85        LOWER(`redirection_us_top`.`responseURL`)))
86
87    /* Another method to query diverse final URLs. It gives the same
        results as the above condition */
88    /* AND NOT((`redirection_london_top`.`responseURL` = `
      redirection_mumbai_top`.`responseURL`)
89      AND (`redirection_mumbai_top`.`responseURL` = `
      redirection_saopaulo_top`.`responseURL`)
90      AND (`redirection_saopaulo_top`.`responseURL` = `
      redirection_sydney_top`.`responseURL`)
91      AND (`redirection_sydney_top`.`responseURL` = `
      redirection_us_top`.`responseURL`))*/
92  ) /* End the WHERE clause */;
```

Listing B.9: MySQL script to query the total number of inconsistent responses with respect to incompatible intermediate URLs weakness indicator (same as the previous Listing B.8 but without the diverse URLs condition).

```
1  /* This MySQL query is to find the total number of inconsistent
       intermediate URLs.
2  /*================================================================*/
3  /* Select response columns */
4  SELECT `redirection_london_top`.`compatibleInterURL`
5    as 'inter. compatible URL UK',
6  `redirection_mumbai_top`.`compatibleInterURL`
7    as 'inter. compatible URL IN',
8  `redirection_saopaulo_top`.`compatibleInterURL`
9    as 'inter. compatible URL BR',
10 `redirection_sydney_top`.`compatibleInterURL`
11   as 'inter. compatible URL AU',
12 `redirection_us_top`.`compatibleInterURL`
13   as 'inter. compatible URL US'
14
15 /* From nested joint tables */
16 FROM `transparency_5_regions_top250k`.`redirection_london_top`
17 JOIN `transparency_5_regions_top250k`.`redirection_mumbai_top`
18 ON `redirection_london_top`.`host` =
19   `redirection_mumbai_top`.`host`
```

```
20 JOIN `transparency_5_regions_top250k`.`redirection_saopaulo_top`
21 ON `redirection_mumbai_top`.`host` =
22   `redirection_saopaulo_top`.`host`
23 JOIN `transparency_5_regions_top250k`.`redirection_sydney_top`
24 ON `redirection_saopaulo_top`.`host` =
25   `redirection_sydney_top`.`host`
26 JOIN `transparency_5_regions_top250k`.`redirection_us_top`
27 ON `redirection_sydney_top`.`host` =
28   `redirection_us_top`.`host`
29
30 /* The Conditions are stated here */
31 WHERE (
32   /* All have valid responses (200 code) */
33   (`redirection_london_top`.`responseStatusCode`
34       LIKE '200'
35     AND `redirection_mumbai_top`.`responseStatusCode`
36       LIKE '200'
37     AND `redirection_saopaulo_top`.`responseStatusCode`
38       LIKE '200'
39     AND `redirection_sydney_top`.`responseStatusCode`
40       LIKE '200'
41     AND `redirection_us_top`.`responseStatusCode`
42       LIKE '200')
43
44   /* One or more response have incompatible inter. URL*/
45   AND
46   (`redirection_london_top`.`compatibleInterURL` = 0
47     OR `redirection_mumbai_top`.`compatibleInterURL` = 0
48     OR `redirection_saopaulo_top`.`compatibleInterURL` = 0
49     OR `redirection_sydney_top`.`compatibleInterURL`  = 0
50     OR `redirection_us_top`.`compatibleInterURL`  = 0)
51
52   /* But not all responses have incompatible inter. URL */
53   AND NOT
54   (`redirection_london_top`.`compatibleInterURL`  = 0
55     AND `redirection_mumbai_top`.`compatibleInterURL`  = 0
56     AND `redirection_saopaulo_top`.`compatibleInterURL`  = 0
57     AND `redirection_sydney_top`.`compatibleInterURL`  = 0
58     AND `redirection_us_top`.`compatibleInterURL`  = 0)
59
60   /* And requested domains (hosts) are not in the malicious
      domains table. Note: we can query one region here because we
      query joint responses which are for the same request */
61     AND
62   (`redirection_london_top`.`host` NOT IN
63     (SELECT `malicious_domains`.`domain` FROM `
     transparency_5_regions_top250k`.`malicious_domains`)
64     AND `redirection_mumbai_top`.`host` NOT IN
65         (SELECT `malicious_domains`.`domain` FROM `
     transparency_5_regions_top250k`.`malicious_domains`)
66     AND `redirection_saopaulo_top`.`host` NOT IN
67         (SELECT `malicious_domains`.`domain` FROM `
     transparency_5_regions_top250k`.`malicious_domains`)
68     AND `redirection_sydney_top`.`host` NOT IN
69         (SELECT `malicious_domains`.`domain` FROM `
```

```
    transparency_5_regions_top250k`.`malicious_domains`)
70   AND `redirection_us_top`.`host` NOT IN
71       (SELECT `malicious_domains`.`domain` FROM `
    transparency_5_regions_top250k`.`malicious_domains`))
72 ) /* End the WHERE clause */;
```

Listing B.10: MySQL script to query consistent compatible intermediate URLs (secure) that have diverse final URLs.

```
1  /* This MySQL query is to find consistent compatible intermediate
       URLs
2   that are also have diverse URLs.*/
3  /*================================================================*/
4  /* Select response columns */
5  SELECT `redirection_london_top`.`compatibleInterURL`
6    as 'inter. compatible URL UK',
7  `redirection_mumbai_top`.`compatibleInterURL`
8    as 'inter. compatible URL IN',
9  `redirection_saopaulo_top`.`compatibleInterURL`
10   as 'inter. compatible URL BR',
11 `redirection_sydney_top`.`compatibleInterURL`
12   as 'inter. compatible URL AU',
13 `redirection_us_top`.`compatibleInterURL`
14   as 'inter. compatible URL US'
15
16 /* From nested joint tables */
17 FROM `transparency_5_regions_top250k`.`redirection_london_top`
18 JOIN `transparency_5_regions_top250k`.`redirection_mumbai_top`
19 ON `redirection_london_top`.`host` =
20   `redirection_mumbai_top`.`host`
21
22 JOIN `transparency_5_regions_top250k`.`redirection_saopaulo_top`
23 ON `redirection_mumbai_top`.`host` =
24   `redirection_saopaulo_top`.`host`
25
26 JOIN `transparency_5_regions_top250k`.`redirection_sydney_top`
27 ON `redirection_saopaulo_top`.`host` =
28   `redirection_sydney_top`.`host`
29
30 JOIN `transparency_5_regions_top250k`.`redirection_us_top`
31 ON `redirection_sydney_top`.`host` =
32   `redirection_us_top`.`host`
33
34 /* The Conditions are stated here */
35 WHERE (
36   /* All have valid responses (200 code) */
37   (`redirection_london_top`.`responseStatusCode`
38       LIKE '200'
39     AND `redirection_mumbai_top`.`responseStatusCode`
40       LIKE '200'
41         AND `redirection_saopaulo_top`.`responseStatusCode`
42       LIKE '200'
43         AND `redirection_sydney_top`.`responseStatusCode`
44       LIKE '200'
45         AND `redirection_us_top`.`responseStatusCode`
```

```
46        LIKE '200')
47
48    /* All responses have compatible inter. URL or does not have
       inter. URL which is also secure (i.e. the compatibility check
       result is 1 (comptaible) or -1 (no inter. URL) but not 0 (
       incompatible)) */
49     AND
50     (`redirection_london_top`.`compatibleInterURL` != 0
51          AND `redirection_mumbai_top`.`compatibleInterURL` != 0
52          AND `redirection_saopaulo_top`.`compatibleInterURL` != 0
53          AND `redirection_sydney_top`.`compatibleInterURL` != 0
54          AND `redirection_us_top`.`compatibleInterURL` != 0)
55
56    /* And requested domains (hosts) are not in the malicious
       domains table. Note: we can query one region here because we
       query joint responses which are for the same request */
57     AND
58    (`redirection_london_top`.`host` NOT IN
59      (SELECT `malicious_domains`.`domain` FROM `
       transparency_5_regions_top250k`.`malicious_domains`)
60     AND `redirection_mumbai_top`.`host` NOT IN
61         (SELECT `malicious_domains`.`domain` FROM `
       transparency_5_regions_top250k`.`malicious_domains`)
62     AND `redirection_saopaulo_top`.`host` NOT IN
63         (SELECT `malicious_domains`.`domain` FROM `
       transparency_5_regions_top250k`.`malicious_domains`)
64     AND `redirection_sydney_top`.`host` NOT IN
65         (SELECT `malicious_domains`.`domain` FROM `
       transparency_5_regions_top250k`.`malicious_domains`)
66     AND `redirection_us_top`.`host` NOT IN
67         (SELECT `malicious_domains`.`domain` FROM `
       transparency_5_regions_top250k`.`malicious_domains`))
68
69    /* And have diverse final URLs */
70    AND (GREATEST(LOWER(`redirection_london_top`.`responseURL`),
71        LOWER(`redirection_mumbai_top`.`responseURL`),
72        LOWER(`redirection_saopaulo_top`.`responseURL`),
73        LOWER(`redirection_sydney_top`.`responseURL`),
74        LOWER(`redirection_us_top`.`responseURL`))
75      !=
76      LEAST(LOWER(`redirection_london_top`.`responseURL`),
77        LOWER(`redirection_mumbai_top`.`responseURL`),
78        LOWER(`redirection_saopaulo_top`.`responseURL`),
79        LOWER(`redirection_sydney_top`.`responseURL`),
80        LOWER(`redirection_us_top`.`responseURL`)))
81  ) /* End the WHERE clause */;
```

Listing B.11: MySQL script to query the total number of consistent compatible intermediate URLs (secure) (same as the previous Listing B.10 but without the diverse URLs condition).

```
1  /* This MySQL query is to find the total number of consistent
       compatible intermediate URLs.*/
2  /*============================================================*/
3  /* Select response columns */
```

```
SELECT `redirection_london_top`.`compatibleInterURL`
  as 'inter. compatible URL UK',
`redirection_mumbai_top`.`compatibleInterURL`
  as 'inter. compatible URL IN',
`redirection_saopaulo_top`.`compatibleInterURL`
  as 'inter. compatible URL BR',
`redirection_sydney_top`.`compatibleInterURL`
  as 'inter. compatible URL AU',
`redirection_us_top`.`compatibleInterURL`
  as 'inter. compatible URL US'

/* From nested joint tables */
FROM `transparency_5_regions_top250k`.`redirection_london_top`
JOIN `transparency_5_regions_top250k`.`redirection_mumbai_top`
ON `redirection_london_top`.`host` =
  `redirection_mumbai_top`.`host`

JOIN `transparency_5_regions_top250k`.`redirection_saopaulo_top`
ON `redirection_mumbai_top`.`host` =
  `redirection_saopaulo_top`.`host`

JOIN `transparency_5_regions_top250k`.`redirection_sydney_top`
ON `redirection_saopaulo_top`.`host` =
  `redirection_sydney_top`.`host`

JOIN `transparency_5_regions_top250k`.`redirection_us_top`
ON `redirection_sydney_top`.`host` =
  `redirection_us_top`.`host`

/* The Conditions are stated here */
WHERE (
  /* All have valid responses (200 code) */
  (`redirection_london_top`.`responseStatusCode`
      LIKE '200'
    AND `redirection_mumbai_top`.`responseStatusCode`
      LIKE '200'
        AND `redirection_saopaulo_top`.`responseStatusCode`
      LIKE '200'
        AND `redirection_sydney_top`.`responseStatusCode`
      LIKE '200'
        AND `redirection_us_top`.`responseStatusCode`
      LIKE '200')

  /* All responses have compatible inter. URL or does not have
    inter. URL which is also secure (i.e. the compatibility check
    result is 1 (comptaible) or -1 (no inter. URL) but not 0 (
    incompatible)) */
    AND
    (`redirection_london_top`.`compatibleInterURL` != 0
         AND `redirection_mumbai_top`.`compatibleInterURL` != 0
         AND `redirection_saopaulo_top`.`compatibleInterURL` != 0
         AND `redirection_sydney_top`.`compatibleInterURL` != 0
         AND `redirection_us_top`.`compatibleInterURL` != 0)

  /* And requested domains (hosts) are not in the malicious
```

```
    domains table. Note: we can query one region here because we
    query joint responses which are for the same request */
56    AND
57  (`redirection_london_top`.`host` NOT IN
58    (SELECT `malicious_domains`.`domain` FROM `
    transparency_5_regions_top250k`.`malicious_domains`)
59    AND `redirection_mumbai_top`.`host` NOT IN
60        (SELECT `malicious_domains`.`domain` FROM `
    transparency_5_regions_top250k`.`malicious_domains`)
61    AND `redirection_saopaulo_top`.`host` NOT IN
62        (SELECT `malicious_domains`.`domain` FROM `
    transparency_5_regions_top250k`.`malicious_domains`)
63    AND `redirection_sydney_top`.`host` NOT IN
64        (SELECT `malicious_domains`.`domain` FROM `
    transparency_5_regions_top250k`.`malicious_domains`)
65    AND `redirection_us_top`.`host` NOT IN
66        (SELECT `malicious_domains`.`domain` FROM `
    transparency_5_regions_top250k`.`malicious_domains`))
67) /* End the WHERE clause */;
```

References

[1] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A Messy State of the Union: Taming the Composite State Machines of TLS", in *Proceedings of IEEE Symposium on Security and Privacy (SP)*, 2015, pp. 535–552.

[2] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice", in *Proceedings of Conference on Computer and Communications Security (CCS)*, 2015, pp. 5–17.

[3] Z. Durumeric, D. Adrian, A. Mirian, J. Kasten, E. Bursztein, N. Lidzborski, K. Thomas, V. Eranti, M. Bailey, and J. A. Halderman, "Neither Snow Nor Rain Nor MITM...: An Empirical Analysis of Email Delivery Security", in *Proceedings of Internet Measurement Conference (IMC)*, 2015, pp. 27–39.

[4] E. S. Alashwali, "CDT Mini-project-2 Research Proposal: On Downgrade Attacks", University of Oxford, Tech. Rep., 2016.

[5] ——, "'"Negotiation-Transparency" as a Property in Configurable Protocols. DPhil Transfer of Status Report.", University of Oxford, Tech. Rep., 2017.

[6] E. S. Alashwali, P. Szalachowski, and A. Martin, "Towards Forward Secure Internet Traffic", in *Proceedings of Security and Privacy in Communication Networks (SecureComm)*, 2019, pp. 341–364.

[7] M. Kosek, L. Blöcher, J. Rüth, T. Zimmermann, and O. Hohlfeld, "MUST, SHOULD, DON'T CARE: TCP Conformance in the Wild", in *Proceedings of Passive and Active Measurement (PAM)*, 2020, pp. 122–138.

[8] C. Tiefenau, E. von Zezschwitz, M. Häring, K. Krombholz, and M. Smith, "A Usability Evaluation of Let's Encrypt and Certbot: Usable Security Done Right", in *Proceedings of Conference on Computer and Communications Security (CCS)*, 2019, pp. 1971–1988.

[9] E. S. Alashwali, P. Szalachowski, and A. Martin, "Exploring HTTPS Security Inconsistencies: A Cross-Regional Perspective", *Computers & Security*, vol. 97, no. 101975, 2020.

[10] ——, "Does "www." Mean Better Transport Layer Security?", in *Proceedings of Conference on Availability, Reliability and Security (ARES)*, 2019, 23:1–23:7.

[11] E. S. Alashwali and P. Szalachowski, "DSTC: DNS-based Strict TLS Configurations", in *Proceedings of Risks and Security of Internet and Systems (CRiSIS)*, 2018, pp. 93–109.

[12] E. S. Alashwali and K. Rasmussen, "On the Feasibility of Fine-Grained TLS Security Configurations in Web Browsers Based on the Requested Domain Name", in *Proceedings of Security and Privacy in Communication Networks (SecureComm)*, 2018, pp. 213–228.

[13] ——, "What's in a Downgrade? A Taxonomy of Downgrade Attacks in the TLS Protocol and Application Protocols Using TLS", in *Proceedings of Security and Privacy in Communication Networks (SecureComm)*, 2018, pp. 468–487.

[14] C. Partridge and M. Allman, "Ethical Considerations in Network Measurement Papers", *Commun. ACM*, vol. 59, no. 10, pp. 58–64, 2016.

[15] T. Dierks and E. Rescorla. (2008). The Transport Layer Security (TLS) Protocol Version 1.2. Accessed Jul. 6, 2018, [Online]. Available: `https://www.ietf.org/rfc/rfc5246.txt`.

[16] E. Rescorla. (2018). The Transport Layer Security (TLS) Protocol Version 1.3. Accessed Dec. 2, 2019, [Online]. Available: `https://tools.ietf.org/html/rfc8446`.

[17] K. E. Hickman. (1995). The SSL Protocol. Accessed Jan. 3, 2020, [Online]. Available: `https://www-archive.mozilla.org/projects/security/pki/nss/ssl/draft02.html`.

[18] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC press, 1996.

[19] S. Vaudenay, "Security Flaws Induced by CBC Padding-Applications to SSL, IPSEC, WTLS....", in *Advances in Cryptology (EUROCRYPT)*, 2002, pp. 534–546.

[20] N. J. AlFardan and K. G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols", in *Proceedings of IEEE Symposium on Security and Privacy (SP)*, 2013, pp. 526–540.

[21] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[22] W. Diffie and M. Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[23] Y. Nir, S. Josefsson, and M. Pegourie-Gonnard. (2018). Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier. Accessed Jun. 21, 2019, [Online]. Available: `https://tools.ietf.org/html/rfc8422`.

[24] K. Bhargavan, C. Brzuska, C. Fournet, M. Green, M. Kohlweiss, and S. Zanella-Béguelin, "Downgrade Resilience in Key-Exchange Protocols", in *Proceedings of IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 506–525.

[25] B. Dowling and D. Stebila, "Modelling Ciphersuite and Version Negotiation in the TLS Protocol", in *Proceedings of Information Security and Privacy (ACISP)*, 2015, pp. 270–288.

[26] P. Zimmermann, A. Johnston, and J. Callas. (2011). ZRTP: Media Path Key Agreement for Unicast Secure RTP. Accessed Mar. 11, 2020, [Online]. Available: `https://tools.ietf.org/html/rfc6189#page-11`.

[27] T. Ylonen and C. Lonvick. (2006). The Secure Shell (SSH) Transport Layer Protocol. Accessed Mar. 11, 2020, [Online]. Available: https://tools.ietf.org/html/rfc4253#page-17.

[28] A. Shaik, R. Borgaonkar, N Asokan, V. Niemi, and J.-P. Seifert, "Practical Attacks Against Privacy and Availability in 4G/LTE Mobile Communication Systems", in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2016.

[29] Wikipedia. (2017). Export of Cryptography from the United States. Accessed Apr. 19, 2020, [Online]. Available: https://en.wikipedia.org/wiki/Export_of_cryptography_from_the_United_States.

[30] J. Klensin. (2001). Simple Mail Transfer Protocol. Accessed Feb. 25, 2018, [Online]. Available: https://www.ietf.org/rfc/rfc2821.txt.

[31] P. Hoffman. (2002). SMTP Service Extension for Secure SMTP over Transport Layer Security. Accessed Feb. 25, 2018, [Online]. Available: https://tools.ietf.org/html/rfc3207.

[32] B. Möller, T. Duong, and K. Kotowicz. (2014). This POODLE Bites: Exploiting the SSL 3.0 Fallback. Accessed Apr. 19, 2020, [Online]. Available: https://www.openssl.org/~bodo/ssl-poodle.pdf.

[33] R. Fielding and J. Reschke. (2014). Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. Accessed Jul. 2, 2019, [Online]. Available: https://tools.ietf.org/html/rfc7231.

[34] M. West, A. Barth, and D. Veditz. (2016). Content Security Policy Level 2. Accessed Feb. 26, 2020, [Online]. Available: https://www.w3.org/TR/CSP2/.

[35] Mozilla. (2019). Content Security Policy (CSP). Accessed Jun. 3, 2019, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP.

[36] S. Stamm, B. Sterne, and G. Markham, "Reining in the Web with Content Security Policy", in *Proceedings of Conference on World Wide Web (WWW)*, 2010, pp. 921–930.

[37] A. Barth. (2011). HTTP State Management Mechanism. Accessed Feb. 26, 2020, [Online]. Available: https://tools.ietf.org/html/rfc6265#page-5.

[38] Mozilla. (2019). HTTP Cookies. Accessed Jun. 3, 2019, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies.

[39] J. Hodges, C. Jackson, and A. Barth. (2012). HTTP Strict Transport Security (HSTS). Accessed Apr. 19, 2020, [Online]. Available: https://tools.ietf.org/html/rfc6797.

[40] P. Mockapetris. (1987). Domain Names - Implementation and Specification. Accessed Jul. 6, 2018, [Online]. Available: https://tools.ietf.org/html/rfc1035.

[41] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. (2005). DNS Security Introduction and Requirements. Accessed Jul. 6, 2018, [Online]. Available: https://tools.ietf.org/html/rfc4033.

[42] Cloudflare. (2020). What Is Round-Robin DNS? Accessed Mar. 12, 2020, [Online]. Available: https://www.cloudflare.com/learning/dns/glossary/round-robin-dns/.

[43] A. Freier, P. Karlton, and P. Kocher. (2011). The Secure Sockets Layer (SSL) Protocol Version 3.0. Accessed Feb. 25, 2018, [Online]. Available: https://tools.ietf.org/html/rfc6101.

[44] D. Wagner and B. Schneier, "Analysis of the SSL 3.0 Protocol", in *Proceedings of USENIX Workshop on Electronic Commerce (EC)*, 1996, pp. 29–40.

[45] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel, "A Cross-Protocol Attack on the TLS Protocol", in *Proceedings of Conference on Computer and Communications Security (CCS)*, 2012, pp. 62–72.

[46] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt, "DROWN: Breaking TLS Using SSLv2", in *Proceedings of USENIX Security Symposium (USENIX)*, 2016, pp. 689–706.

[47] The MITRE Corporation. (2015). CVE-2015-3197. Accessed Apr. 19, 2020, [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3197.

[48] D. Bleichenbacher, "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS-1", in *Proceedings of Advances in Cryptology (CRYPTO)*, 1998, pp. 1–12.

[49] E. Rescorla. (2015). The Transport Layer Security (TLS) Protocol Version 1.3 draft-ietf-tls-tls13-10. Accessed Feb. 25, 2020, [Online]. Available: https://tools.ietf.org/html/draft-ietf-tls-tls13-10.

[50] K. Bhargavan and G. Leurent, "Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH", in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2016.

[51] T. Dierks and C. Allen. (1999). The TLS Protocol Version 1.0. Accessed Feb. 25, 2020, [Online]. Available: https://www.ietf.org/rfc/rfc2246.txt.

[52] T. Dierks and E. Rescorla. (2006). The Transport Layer Security (TLS) Protocol Version 1.1. Accessed Feb. 25, 2020, [Online]. Available: https://tools.ietf.org/html/rfc4346.

[53] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. Halderman, and V. Paxson, "The Security Impact of HTTPS Interception", in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2017.

[54] B. VanderSloot, J. Amann, M. Bernhard, Z. Durumeric, M. Bailey, and J. A. Halderman, "Towards a Complete View of the Certificate Ecosystem", in *Proceedings of Internet Measurement Conference (IMC)*, 2016, pp. 543–549.

[55] H. K. Lee, T. Malkin, and E. Nahum, "Cryptographic Strength of SSL/TLS Servers: Current and Recent Practices", in *Proceedings of Internet Measurement Conference (IMC)*, 2007, pp. 83–92.

[56] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices", in *Proceedings of USENIX Security Symposium (USENIX)*, 2012.

[57] E. S. Alashwali, "Cryptographic Vulnerabilities in Real-Life Web Servers", in *Proceedings of International Conference on Communications and Information Technology (ICCIT)*, 2013, pp. 6–11.

[58] L. Huang, S. Adhikarla, D. Boneh, and C. Jackson, "An Experimental Study of TLS Forward Secrecy Deployments", *IEEE Internet Computing*, vol. 18, no. 6, pp. 43–51, 2014.

[59] L. Chang, H.-C. Hsiao, W. Jeng, T. H.-J. Kim, and W.-H. Lin, "Security Implications of Redirection Trail in Popular Websites Worldwide", in *Proceedings of Conference on World Wide Web (WWW)*, 2017, pp. 1491–1500.

[60] N. Samarasinghe and M. Mannan, "Short Paper: TLS Ecosystems in Networked Devices vs. Web Servers", in *Proceedings of Financial Cryptography and Data Security (FC)*, 2017, pp. 533–541.

[61] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman, "A Search Engine Backed by Internet-Wide Scanning", in *Proceedings of Conference on Computer and Communications Security (CCS)*, 2015, pp. 542–553.

[62] N. Samarasinghe and M. Mannan, "Another Look at TLS Ecosystems in Networked Devices vs. Web Servers", *Computers & Security*, vol. 80, pp. 1–13, 2019.

[63] P. Kotzias, A. Razaghpanah, J. Amann, K. G. Paterson, N. Vallina-Rodriguez, and J. Caballero, "Coming of Age: A Longitudinal Study of TLS Deployment", in *Proceedings of Internet Measurement Conference (IMC)*, 2018, pp. 415–428.

[64] S. Calzavara, R. Focardi, M. Nemec, A. Rabitti, and M. Squarcina, "Postcards from the Post-HTTP World: Amplification of HTTPS Vulnerabilities in the Web Ecosystem", in *Proceedings of IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 281–298.

[65] V. Frost, D. J. Tian, C. Ruales, V. Prakash, P. Traynor, and K. R. B. Butler, "Examining DES-based Cipher Suite Support Within the TLS Ecosystem", in *Proceedings of Asia Conference on Computer and Communications Security (Asia CCS)*, 2019, pp. 539–546.

[66] M. Curtin and J. Dolske. (1998). A Brute Force Search of DES Keyspace. Accessed Dec. 5, 2019, [Online]. Available: `http://web.interhack.com/publications/des-key-crack`.

[67] NIST. (1999). DATA ENCRYPTION STANDARD (DES). Accessed Dec. 5, 2019, [Online]. Available: `https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf`.

[68] R. Holz, J. Amann, A. Razaghpanah, and N. Vallina-Rodriguez. (2019). The Era of TLS 1.3: Measuring Deployment and Use with Active and Passive Methods. Accessed Apr. 19, 2020, [Online]. Available: `https://arxiv.org/pdf/1907.12762.pdf`.

[69] R. Holz, L. Braun, N. Kammenhuber, and G. Carle, "The SSL Landscape: A Thorough Analysis of the X.509 PKI Using Active and Passive Measurements", in *Proceedings of Internet Measurement Conference (IMC)*, 2011, pp. 427–444.

[70] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, "Analysis of the HTTPS Certificate Ecosystem", in *Proceedings of Internet Measurement Conference (IMC)*, 2013, pp. 291–304.

[71] T. Chung, Y. Liu, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson, "Measuring and Applying Invalid SSL Certificates: The Silent Majority", in *Proceedings of Internet Measurement Conference (IMC)*, 2016, pp. 527–541.

[72] E. Stark, R. Sleevi, R. Muminovic, D. O'Brien, E. Messeri, A. Porter Felt, B. McMillion, and P. Tabriz, "Does Certificate Transparency Break the Web? Measuring Adoption and Error Rate", in *Proceedings of IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 211–226.

[73] B. Laurie, A. Langley, and E. Kasper. (2013). Certificate Transparency. Accessed Feb. 25, 2019, [Online]. Available: `https://tools.ietf.org/html/rfc6962`.

[74] M. Kranch and J. Bonneau, "Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning", in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2015.

[75] J. Amann, O. Gasser, Q. Scheitle, L. Brent, G. Carle, and R. Holz, "Mission Accomplished?: HTTPS Security After DigiNotar", in *Proceedings of Internet Measurement Conference (IMC)*, 2017, pp. 325–340.

[76] P. Hoffman and J. Schlyter. (2012). The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. Accessed Apr. 19, 2020, [Online]. Available: `https://tools.ietf.org/html/rfc6698`.

[77] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann, "Factorization of a 768-bit RSA Modulus", in *Proceedings of Advances in Cryptology (CRYPTO)*, 2010, pp. 333–350.

[78] S. Cavallar, B. Dodson, A. K. Lenstra, W. Lioen, P. L. Montgomery, B. Murphy, H. te Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. Leyland, J. Marchand, F. Morain, A. Muffett, C. Putnam, Craig, and P. Zimmermann, "Factorization of a 512-bit RSA Modulus", in *Proceedings of Advances in Cryptology (EUROCRYPT)*, 2000, pp. 1–18.

[79] Synopsys Inc. (2014). The Heartbleed Bug. Accessed Sep. 17, 2018, [Online]. Available: `http://heartbleed.com`.

[80] A. Mendoza, P. Chinprutthiwong, and G. Gu, "Uncovering HTTP Header Inconsistencies and the Impact on Desktop/Mobile Websites", in *Proceedings of Conference on World Wide Web (WWW)*, 2018, pp. 247–256.

[81] S. Khattak, D. Fifield, S. Afroz, M. Javed, S. Sundaresan, V. Paxson, S. J. Murdoch, and D. McCoy, "Do You See What I See? Differential Treatment of Anonymous Users", in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2016.

[82] T. Van Goethem, V. Le Pochat, and W. Joosen, "Mobile Friendly or Attacker Friendly? A Large-scale Security Evaluation of Mobile-first Websites", in *Proceedings of Asia Conference on Computer and Communications Security (Asia CCS)*, 2019, pp. 206–213.

[83] S. Afroz, M. C. Tschantz, S. Sajid, S. A. Qazi, M. Javed, and V. Paxson. (2018). Exploring Server-side Blocking of Regions. Accessed Feb. 25, 2020, [Online]. Available: `https://arxiv.org/pdf/1805.11606.pdf`.

[84] N. Fruchter, H. Miao, and S. Stevenson, "Variations in Tracking in Relation to Geographic Location", in *Proceedings of Web 2.0 Security and Privacy (W2SP)*, 2015.

[85] N. Samarasinghe and M. Mannan, "Towards a Global Perspective on Web Tracking", *Computers & Security*, vol. 87, no. 101569, 2019.

[86] R. Van Eijk, H. Asghari, P. Winter, and A. Narayanan, "The Impact of User Location on Cookie Notices (Inside and Outside of the European Union)", in *Proceedings of Workshop on Technology and Consumer Protection (ConPro)*, 2019.

[87] J. Jueckstock, S. Sarker, P. Snyder, P. Papadopoulos, M. Varvello, B. Livshits, and A. Kapravelos. (2019). The Blind Men and the Internet: Multi-Vantage Point Web Measurements. Accessed Apr. 16, 2020, [Online]. Available: `https://arxiv.org/pdf/1905.08767.pdf`.

[88] S. Schechter. (2007). Storing HTTP Security Requirements in the Domain Name System. Accessed Feb. 25, 2018, [Online]. Available: `https://lists.w3.org/Archives/Public/public-wsc-wg/2007Apr/att-0332/http-ssr.html`.

[89] P. Hallam-Baker. (2013). DNS Certification Authority Authorization (CAA) Resource Record. Accessed Jul. 6, 2018, [Online]. Available: `https://tools.ietf.org/html/rfc6844`.

[90] V. Dukhovni and W. Hardaker. (2015). The DNS-Based Authentication of Named Entities (DANE) Protocol: Updates and Operational Guidance. Accessed Jul. 6, 2018, [Online]. Available: `https://tools.ietf.org/html/rfc7671`.

[91] G. Varshney and P. Szalachowski, "A Metapolicy Framework for Enhancing Domain Expressiveness on the Internet", in *Proceedings of Security and Privacy in Communication Networks (SecureComm)*, 2018, pp. 155–170.

[92] C. Jackson and A. Barth, "ForceHTTPS: Protecting High-Security Web Sites From Network Attacks", in *Proceedings of Conference on World Wide Web (WWW)*, 2008, pp. 525–534.

[93] C. Evans, C. Palmer, and R. Sleevi. (2015). Public Key Pinning Extension for HTTP. Accessed Jan. 6, 2019, [Online]. Available: `https://tools.ietf.org/html/rfc7469`.

[94] P. Szalachowski, S. Matsumoto, and A. Perrig, "PoliCert: Secure and Flexible TLS Certificate Management", in *Proceedings of Conference on Computer and Communications Security (CCS)*, 2014, pp. 406–417.

[95] C. Jackson, D. R. Simon, D. S. Tan, and A. Barth, "An Evaluation of Extended Validation and Picture-In-Picture Phishing Attacks", in *Proceedings Financial Cryptography and Data Security (FC)*, 2007, pp. 281–293.

[96] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer, "The Emperor's New Security Indicators", in *Proceedings of IEEE Symposium on Security and Privacy (SP)*, 2007, pp. 51–65.

[97] D. Akhawe and A. P. Felt, "Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness", in *Proceedings of USENIX Security Symposium (USENIX)*, 2013, pp. 257–272.

[98] S. Egelman, L. F. Cranor, and J. Hong, "You've Been Warned: An Empirical Study of the Effectiveness of Web Browser Phishing Warnings", in *Proceedings of Conference on Human Factors in Computing Systems (CHI)*, 2008, pp. 1065–1074.

[99] M. Wu, R. C. Miller, and S. L. Garfinkel, "Do Security Toolbars Actually Prevent Phishing Attacks?", in *Proceedings of Conference on Human Factors in Computing Systems (CHI)*, 2006, pp. 601–610.

[100] R. Reeder, A. P. Felt, S. Consolvo, N. Malkin, C. Thompson, and S. Egelman, "An Experience Sampling Study of User Reactions to Browser Warnings in the Field", in *Proceedings of Conference on Human Factors in Computing Systems (CHI)*, 2018, pp. 1–13.

[101] E. Zeng, F. Li, E. Stark, A. P. Felt, and P. Tabriz, "Fixing HTTPS Misconfigurations at Scale: An Experiment with Security Notifications", in *Proceedings of Workshop on the Economics of Information Security (WEIS)*, 2019.

[102] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor, "Crying Wolf: An Empirical Study of SSL Warning Effectiveness", in *Proceedings of USENIX Security Symposium (USENIX)*, 2009, pp. 399–416.

[103] Alexa Internet, Inc. (2018). Alexa Top Sites. Accessed Aug. 22, 2018, [Online]. Available: `http://s3.amazonaws.com/alexa-static/top-1m.csv.zip`.

[104] J. Kurkowski. (2017). tldextract. Accessed Oct. 30, 2018, [Online]. Available: `https://github.com/john-kurkowski/tldextract`.

[105] Mozilla. (2019). Public Suffix List. Accessed Oct. 14, 2019, [Online]. Available: `https://www.publicsuffix.org/`.

[106] Yahoo Inc. (2016). tls-scan. Accessed Sep. 8, 2018, [Online]. Available: `https://github.com/prbinu/tls-scan`.

[107] W3Schools. (2019). Browser Statistics. Accessed Feb. 27, 2019, [Online]. Available: `https://www.w3schools.com/browsers`.

[108] StatCounter. (2020). Desktop Browser Market Share Worldwide. Accessed Feb. 21, 2020, [Online]. Available: `https://gs.statcounter.com/browser-market-share/desktop/worldwide#monthly-201902-202006`.

[109] J. Salowey, A. Choudhury, and D. McGrew. (2008). AES Galois Counter Mode (GCM) Cipher Suites for TLS. Accessed Nov. 12, 2018, [Online]. Available: `https://tools.ietf.org/html/rfc5288#page-3`.

[110] R. Barnes, M. Thomson, A. Pironti, and A. Langley. (2015). Deprecating Secure Sockets Layer Version 3.0. Accessed Sep. 30, 2018, [Online]. Available: `https://tools.ietf.org/html/rfc7568`.

[111] A. Popov. (2015). Prohibiting RC4 Cipher Suites. Accessed Sep. 30, 2018, [Online]. Available: `https://tools.ietf.org/html/rfc7465`.

[112] FIPS. (2001). Advanced Encryption Standard (AES). Accessed Sep. 30, 2018, [Online]. Available: `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf`.

[113] V. Dukhovni. (2014). Opportunistic Security: Some Protection Most of the Time. Accessed Oct. 1, 2018, [Online]. Available: https://tools.ietf.org/html/rfc7435.html.

[114] B. Moeller and A. Langley. (2014). TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks. Accessed Oct. 1, 2018, [Online]. Available: https://tools.ietf.org/html/draft-ietf-tls-downgrade-scsv-00.

[115] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk, "On the Security of TLS-DHE in the Standard Model", in *Proceedings of Advances in Cryptology (CRYPTO)*, 2012, pp. 273–293.

[116] Wikipedia. (2018). PRISM (Surveillance Program). Accessed Oct. 3, 2018, [Online]. Available: https://en.wikipedia.org/wiki/PRISM_(surveillance_program).

[117] M. D. Ryan, "Enhanced Certificate Transparency and End-to-End Encrypted Mail", in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2018.

[118] A. Young and M. Yung, "The Dark Side of Black-Box Cryptography or: Should We Trust Capstone?", in *Proceedings of Advances in Cryptology (CRYPTO)*, 1996, pp. 89–103.

[119] S. Nichols. (2018). Official: Google Chrome 69 Kills off the World Wide Web (in URLs). Accessed Dec. 1, 2018, [Online]. Available: https://www.theregister.co.uk/2018/09/07/google_kills_www.

[120] L. Tung. (2018). Chrome 69 Kills Off www in URLs: Here's Why Google's Move Has Made People Angry. Accessed Dec. 1, 2018, [Online]. Available: https://www.zdnet.com/article/chrome-69-kills-off-www-in-urls-heres-why-googles-move-has-made-people-angry.

[121] S. Brink. (2018). Hide or Show Scheme and WWW Subdomains of URLs in Google Chrome Address Bar. Accessed Dec. 1, 2018, [Online]. Available: https://www.tenforums.com/tutorials/117616-hide-show-www-subdomains-urls-address-bar-google-chrome.html.

[122] hugues.a...@gmail.com. (2018). Incorrect Transforms When Stripping Subdomains. Accessed Dec. 1, 2018, [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=881410.

[123] L. Abrams. (2018). Google Testing Removal of WWW Subdomain from Search Results. Accessed Dec. 17, 2018, [Online]. Available: https://www.bleepingcomputer.com/news/google/google-testing-removal-of-www-subdomain-from-search-results/.

[124] X. Li, C. Wu, S. Ji, Q. Gu, and R. Beyah, "HSTS Measurement and an Enhanced Stripping Attack Against HTTPS", in *Proceedings of Security and Privacy in Communication Networks (SecureComm)*, 2017, pp. 489–509.

[125] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz, "Measuring HTTPS Adoption on the Web", in *Proceedings of USENIX Security Symposium (USENIX)*, 2017, pp. 1323–1338.

[126] A. K. R. Project. (2019). Requests: HTTP for Humans. Accessed Apr. 3, 2019, [Online]. Available: http://docs.python-requests.org/en/master.

[127] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. (2008). Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Accessed Dec. 28, 2018, [Online]. Available: https://tools.ietf.org/html/rfc5280#page-28.

[128] OpenSSL Software Foundation. (2018). SSL_get_verify_result. Accessed Dec. 24, 2018, [Online]. Available: https://www.openssl.org/docs/man1.1.1/man3/SSL_get_verify_result.html.

[129] D. Giry. (2020). Keylength - Cryptographic Key Length Recommendation. Accessed Mar. 25, 2020, [Online]. Available: https://www.keylength.com.

[130] Q. Scheitle, O. Hohlfeld, J. Gamba, J. Jelten, T. Zimmermann, S. D. Strowes, and N. Vallina-Rodriguez, "A Long Way to the Top: Significance, Structure, and Stability of Internet Top Lists", in *Proceedings of the Internet Measurement Conference (IMC)*, 2018, 478–493.

[131] V. L. Pochat, T. V. Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation", in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2019.

[132] ——, (2020). A Research-Oriented Top Sites Ranking Hardened Against Manipulation. Accessed Mar. 25, 2020, [Online]. Available: https://tranco-list.eu/.

[133] Cisco Umbrella. (2019). Umbrella Popularity List. Accessed Jun. 30, 2019, [Online]. Available: http://s3-us-west-1.amazonaws.com/umbrella-static/index.html.

[134] Majestic. (2019). The Majestic Million. Accessed Jun. 30, 2019, [Online]. Available: https://majestic.com/reports/majestic-million.

[135] Quantcast. (2019). Top International Websites. Accessed Jun. 30, 2019, [Online]. Available: https://www.quantcast.com/top-sites/.

[136] Google. (2019). Safe Browsing API Go Client. Accessed Jun. 27, 2019, [Online]. Available: https://github.com/google/safebrowsing/.

[137] Amazon Web Services, Inc. (2019). Amazon EC2. Accessed Jun. 29, 2019, [Online]. Available: https://aws.amazon.com/ec2/.

[138] K. McKay and D. Cooper. (2019). Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations. Accessed Oct. 8, 2019, [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r2.pdf.

[139] OpenSSL Software Foundation. (2019). X509_check_host. Accessed Oct. 14, 2019, [Online]. Available: https://www.openssl.org/docs/man1.1.1/man3/X509_check_host.html.

[140] Wikipedia. (2018). PRISM (Surveillance Program). Accessed Oct. 3, 2018, [Online]. Available: https://en.wikipedia.org/wiki/PRISM_(surveillance_program).

[141] Tor. (2019). Tor FAQ. Accessed Jul. 15, 2019, [Online]. Available: https://2019.www.torproject.org/docs/faq.html.en.

[142] B. Beurdouche, A. Delignat-Lavaud, N. Kobeissi, A. Pironti, and K. Bhargavan, "FLEXTLS A Tool for Testing TLS Implementations", in *Proceedings of USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[143] rbsec. (2018). sslscan Tests SSL/TLS Enabled Services to Discover Supported Cipher Suites. Accessed Jul. 6, 2018, [Online]. Available: https://github.com/rbsec/sslscan.

[144] Chromium. (2017). HSTS Preloaded List Submission. Accessed Dec. 16, 2019, [Online]. Available: https://hstspreload.org.

[145] Oracle. (2018). VirtualBox. Accessed Jul. 6, 2018, [Online]. Available: https://www.virtualbox.org/wiki/VirtualBox.

[146] Apache. (2018). Apache HTTP Server Project. Accessed Jul. 6, 2018, [Online]. Available: https://httpd.apache.org.

[147] Internet Systems Consortium. (2018). Bind Open Source DNS Server. Accessed Jul. 6, 2018, [Online]. Available: https://www.isc.org/downloads/bind.

[148] Python. (2018). Python. Accessed Jul. 6, 2018, [Online]. Available: https://www.python.org.

[149] ——, (2018). ssl – TLS/SSL Wrapper for Socket Objects. Accessed Jul. 6, 2018, [Online]. Available: https://docs.python.org/3.6/library/ssl.html.

[150] ——, (2018). time – Time Access and Conversions. Accessed Jul. 6, 2018, [Online]. Available: https://docs.python.org/3/library/time.html.

[151] S. Fahl, Y. Acar, H. Perl, and M. Smith, "Why Eve and Mallory (Also) Love Webmasters: A Study on the Root Causes of SSL Misconfigurations", in *Proceedings of Asia Conference on Computer and Communications Security (Asia CCS)*, 2014, pp. 507–512.

[152] MrXOR. (2019). Does "www." Mean Better Transport Layer Security? Accessed Mar. 25, 2020, [Online]. Available: https://news.ycombinator.com/item?id=20735220.

[153] A. Smith. (2018). How to Prevent Jetty, Deployed via Docker on AWS ECS, Behind an AWS ELB, Redirecting from Secure HTTPS to Insecure HTTP. Accessed Mar. 17, 2020, [Online]. Available: https://www.databasesandlife.com/jetty-redirect-keep-https.

[154] A. Shaik, R. Borgaonkar, S. Park, and J.-P. Seifert, "New Vulnerabilities in 4G and 5G Cellular Access Network Protocols: Exposing Device Capabilities", in *Proceedings of Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2019, pp. 221–231.

[155] G. Merzdovnik, K. Falb, M. Schmiedecker, A. G. Voyiatzis, and E. Weippl, "Whom You Gonna Trust? A Longitudinal Study on TLS Notary Services", in *Proceedings of Data and Applications Security and Privacy (DBSec)*, 2016, pp. 331–346.

[156] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. (2006). Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). Accessed Feb. 28, 2020, [Online]. Available: `https://tools.ietf.org/html/rfc4492`.

[157] A. Langley, N. Modadugu, and B. Moeller. (2016). Transport Layer Security (TLS) False Start. Accessed Feb. 28, 2020, [Online]. Available: `https://tools.ietf.org/html/rfc7918`.