An Analysis of the Transport Layer Security Protocol

Thyla van der Merwe

Thesis submitted to the University of London
for the degree of Doctor of Philosophy

Information Security Group
School of Mathematics and Information Security
Royal Holloway, University of London

2018

Declaration

These doctoral studies were conducted under the supervision of Professor Kenneth G. Paterson.

The work presented in this thesis is the result of original research I conducted, in collaboration with others, whilst enrolled in the School of Mathematics and Information Security as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Thyla van der Merwe

May, 2018

Dedication

To my niece, Emma.
May you always believe in your abilities, no matter what anybody tells you, and may you draw on the strength of our family for support, as I have done (especially your Gogo, she's one tough lady).

"If you're going through hell, keep going."
Unknown

Abstract

The Transport Layer Security (TLS) protocol is the *de facto* means for securing communications on the World Wide Web. Originally developed by Netscape Communications, the protocol came under the auspices of the Internet Engineering Task Force (IETF) in the mid 1990s and today serves millions, if not billions, of users on a daily basis. The ubiquitous nature of the protocol has, especially in recent years, made the protocol an attractive target for security researchers. Since the release of TLS 1.2 in 2008, the protocol has suffered many high-profile, and increasingly practical, attacks. Coupled with pressure to improve the protocol's efficiency, this deluge of identified weaknesses prompted the IETF to develop a new version of the protocol, namely TLS 1.3.

In the development of the new version of the protocol, the IETF TLS Working Group has adopted an "analysis-prior-to-deployment" design philosophy. This is in sharp contrast to all previous versions of the protocol. We present an account of the TLS standardisation narrative, commenting on the differences between the reactive development process for TLS 1.2 and below, and the more proactive design process for TLS 1.3. As part of this account, we present work that falls on both sides of this design transition. We contribute to the large body of work highlighting weaknesses in TLS 1.2 and below by presenting two classes of attacks against the RC4 stream cipher when used in TLS. Our attacks exploit statistical biases in the RC4 keystream to recover TLS-protected user passwords and cookies. Next we present a symbolic analysis of the TLS 1.3 draft specification, using the Tamarin prover, to show that TLS 1.3 meets the desired goals of authenticated key exchange, thus contributing to a concerted effort by the TLS community to ensure the protocol's robustness prior to its official release.

Publications

This thesis is based on the following five publications, to which each author contributed equally:

1. Christina Garman, Kenneth G. Paterson, and Thyla van der Merwe. Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 113-128, 2015.

2. Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016.*, pages 470-485, 2016.

3. Kenneth G. Paterson and Thyla van der Merwe. Reactive and Proactive Standardisation of TLS. In *Security Standardisation Research - Third International Conference, SSR 2016, Gaithersburg, MD, USA, December 5-6, 2016.*, pages 160-186, 2016.

4. Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A Comprehensive Symbolic Analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, pages 1773-1788, 2017.

5. Remi Bricout, Kenneth G. Paterson, Sean Murphy, and Thyla van der Merwe. Analysing and Exploiting the Mantin Biases in RC4. *Designs, Codes and Cryptography,* 86(4):743-770, April 2018.

Acknowledgements

I would like to thank my supervisor, Kenny Paterson, for his guidance and support over the course of my Ph.D., and for facilitating a number of excellent research opportunities. Thank you for always making time for my work. I would also like to thank Cas Cremers for being an incredible mentor, collaborator, and friend. Your constant encouragement has meant the world to me.

Thanks go to Eric Rescorla for hosting me at Mozilla as an intern (twice!), and for teaching me a great deal about the protocol upon which this thesis is based, TLS. Your advice, and help, over the years have been instrumental in dictating the direction of my research. I also thank Christine Swart for sparking my interest in cryptography many moons ago.

Sam Scott and Jonathan Hoyland, I will never forget our Tamarin adventures, and I thank you for your patience and good humour. Sam, I can't believe that we ventured down the Tamarin rabbit hole more than once! Thank you for teaching me so much about the tool, and for introducing me to dulce de leche during our time in California. I have been fortunate to have worked with a host of impressive and inspiring collaborators, I thank you all.

Sheila Cobourne, I thank you for helping me to start the Women In the Security Domain and/Or Mathematics (WISDOM) group, and for instilling in me a love of cake. I also thank Thalia Laing for her calming influence on the group.

I thank my Ph.D. office mates for creating a lively working environment, and my friends, James Hourston, Jayni Shah, Marc Nimmerrichter, Daniel Etcovitch, Sarah Frewen and Marianne Jonassen for making me laugh when the journey seemed long, dark, and arduous. I especially thank my friends Emily Grace Williams and Hugh Pastoll not only for providing excellent proofreading services but for being fellow warriors in the Battle of Science.

Acknowledgements

I am grateful to the Engineering and Physical Sciences Research Council (EPSRC) for funding my work as part of the Centre for Doctoral Training (CDT) at Royal Holloway, University of London. Thanks go to my fellow CDT cohort members for making the ride memorable.

I thank Dave and Sharon Jackson for their faith in me, and I thank Gary Turner for his love and care over the last three hundred and fifty-two yards. Finally, I thank my family for being there, always. You are the foundation upon which my dreams and aspirations rest.

Contents

I Motivation and Background **16**

1 Introduction **17**

1.1 Motivation 17

1.2 Thesis Structure 20

2 Preliminaries **23**

2.1 The TLS Protocol 23

2.2 TLS 1.2 and Below 25

2.2.1 The Handshake Protocol 25

2.2.2 The Record Protocol 28

2.2.3 Security Properties 28

2.3 TLS 1.3 30

2.4 TLS 1.3 `draft-10` 32

2.4.1 The Handshake Protocol 32

2.4.2 The Record Protocol 36

2.4.3 Security Properties 36

2.5 TLS 1.3 `draft-21` 38

2.5.1 The Handshake Protocol 38

2.5.2 The Record Protocol 42

2.5.3 Post-Handshake Mechanisms 42

2.5.4 Security Properties 43

3 Reactive and Proactive Standardisation of TLS **46**

3.1 Post-Deployment Analysis 46

3.1.1 Design, Release, Break, Patch 47

3.1.2 Fixes, Constraints and Time Lags 51

3.1.3 Impact and Incentives 53

3.2 Pre-Deployment Analysis . . . 53
3.2.1 Design, Break, Fix, Release . . . 54
3.2.2 Available Tools . . . 58
3.2.3 Impact and Incentives . . . 61

II Attacking TLS 1.2 and Below 62

4 Password Recovery Attacks Against RC4 63
4.1 Introduction . . . 63
4.2 Preliminaries . . . 69
4.2.1 Bayes' Theorem . . . 69
4.2.2 The RC4 Algorithm . . . 69
4.2.3 Single-byte Biases in the RC4 Keystream . . . 70
4.2.4 Double-byte Biases in the RC4 Keystream . . . 71
4.2.5 RC4 and the TLS Record Protocol . . . 76
4.2.6 Passwords . . . 77
4.3 Plaintext Recovery via Bayesian Analysis . . . 78
4.3.1 Formal Bayesian Analysis . . . 79
4.3.2 Using a Product Distribution . . . 82
4.3.3 Double-byte-based Approximation . . . 83
4.4 Simulation Results . . . 85
4.4.1 Methodology . . . 85
4.4.2 Results . . . 88
4.5 Practical Validation . . . 98
4.5.1 The BasicAuth Protocol . . . 98
4.5.2 Attacking BasicAuth . . . 100
4.6 Conclusion . . . 102

5 Analysing and Exploiting the Mantin Biases in RC4 104
5.1 Introduction . . . 104
5.2 Preliminaries . . . 108
5.2.1 Inferential Form of Bayes' Theorem . . . 109
5.2.2 Order Statistics . . . 109
5.2.3 The Mantin Biases . . . 110
5.2.4 Dynamic Programming Algorithms . . . 110

5.3 Plaintext Recovery using the Mantin Biases . . . 111
5.3.1 Maximum Likelihood Estimation . . . 112
5.3.2 Plaintext Recovery Attack . . . 114
5.3.3 Distribution of the Maximum Likelihood Statistic and Attack Performance . . . 115
5.3.4 Incorporating Prior Information about Plaintext Bytes . . . 120
5.4 Recovering Multiple Plaintext Bytes . . . 123
5.4.1 A Likelihood Analysis for Multiple Plaintext Bytes . . . 124
5.4.2 Algorithms for Recovering Multiple Plaintext Bytes . . . 125
5.5 Simulation Results . . . 127
5.5.1 Methodology . . . 127
5.5.2 Results . . . 128
5.6 Conclusion . . . 130

III Verifying TLS 1.3 134

6 Automated Analysis and Verification of `draft-10` 135
6.1 Introduction . . . 135
6.2 Preliminaries . . . 139
6.2.1 Symbolic Analysis . . . 139
6.2.2 Tamarin Fundamentals . . . 140
6.3 `draft-10` Analysis . . . 154
6.3.1 Building the Model . . . 157
6.3.2 Encoding Security Properties . . . 162
6.3.3 Analysis and Results . . . 166
6.3.4 Attacking Post-handshake Client Authentication . . . 169
6.4 Conclusion . . . 173

7 Automated Analysis and Verification of `draft-21` 175
7.1 Introduction . . . 175
7.2 Preliminaries . . . 178
7.3 `draft-21` Analysis . . . 178
7.3.1 Building the Model . . . 182
7.3.2 Encoding Security Properties . . . 191
7.3.3 Analysis and Results . . . 198

7.4 Conclusion 203

IV Concluding Remarks **204**

8 Conclusion **205**

Bibliography **206**

A STS .spthy File **225**

List of Acronyms

AES	Advanced Encryption Standard
AEAD	Authenticated Encryption with Associated Data
AKE	Authenticated Key Exchange
CBC	Cipher Block Chaining
CORS	Cross Origin Resource Sharing
DDM	Decrypt-then-Decode-then-MAC
DHE	Ephemeral Diffie Hellman
(EC)DHE	(Elliptic Curve) Ephemeral Diffie Hellman
GCM	Galois Counter Mode
HKDF	HMAC-based Key Derivation Function
HMAC	Hash-based Message Authentication Code
HTTP	Hypertext Transfer Protocol
KCI	Key Compromise Impersonation
MAC	Message Authentication Code
MD5	Message Digest 5
MEE	MAC-then-Encode-then-Encrypt
MITM	Man-In-The-Middle
PFS	Perfect Forward Secrecy
PKI	Public Key Infrastructure
PKCS	Public Key Cryptography Standards
PRF	Pseudo-Random Function
PSK	Pre-Shared Key
PSK-DHE	Pre-Shared Key and Ephemeral Diffie-Hellman
RC4	Refers to a stream cipher developed by Ron Rivest.
RSA	Refers to the RSA encryption primitive.
SHA	Secure Hash Algorithm
SSL	Secure Sockets Layer
STS	Station-to-Station
TCP	Transmission Control Protocol
TLS	Transport Layer Security
WG	Working Group
0-RTT	Zero Round-Trip Time

List of Figures

2.1 TCP/IP protocol architecture 24
2.2 TLS 1.2 initial handshake 26
2.3 TLS 1.2 resumption handshake 27
2.4 `draft-10` (EC)DHE handshake 32
2.5 `draft-10` 0-RTT handshake 34
2.6 `draft-10` PSK resumption handshake 35
2.7 Key computation hierarchy for `draft-10` 36
2.8 `draft-21` (EC)DHE handshake 40
2.9 `draft-21` PSK resumption handshake 40
2.10 `draft-21` 0-RTT handshake 41
2.11 Key computation hierarchy for `draft-21` 42

4.1 Measured biases for RC4 keystream byte pair (Z_{16}, Z_{17}) 72
4.2 Measured biases for RC4 keystream byte pair (Z_{384}, Z_{385}) 73
4.3 Measured biases for RC4 keystream byte pair (Z_1, Z_2) 74
4.4 Absolute value of the largest single-byte bias for keystream bytes Z_{240} to Z_{272} 75
4.5 Recovery rate for Singles.org passwords using the RockYou data set 87
4.6 Recovery rates for the single-byte algorithm 89
4.7 Recovery rates for the double-byte algorithm 90
4.8 Performance of the single-byte algorithm versus a naive attack 91
4.9 Recovery rate of the single-byte versus the double-byte algorithm 92
4.10 Recovery rate for uniformly distributed passwords versus known *a priori* distribution 93
4.11 Effect of password length on recovery rate 94
4.12 Effect of try limit T on recovery rate 96
4.13 Value of T required to achieve a given password recovery rate 97
4.14 Recovery rate of a base64 encoded password versus an ASCII password . . 98
4.15 Recovery rate of the shift attack versus the double-byte algorithm 99

5.1 Cumulative distribution function of the plaintext rank . . . 119
5.2 Experimental validation of the cumulative distribution function of the plaintext rank . . . 121
5.3 Success rate of the list Viterbi algorithm using double-sided biases . . . 129
5.4 Success rate of the list Viterbi algorithm using single-sided and double-sided biases . . . 130
5.5 Success rate of the beam search algorithm using double-sided biases . . . 132
5.6 Success rate of the beam search algorithm with first byte known versus first byte unknown . . . 132
5.7 Success rate of the beam search algorithm without final list pruning versus final list pruning . . . 133
5.8 Success rate of list the Viterbi algorithm versus the beam search algorithm 133

6.1 Basic STS protocol . . . 141
6.2 Tamarin PKI rules for the STS protocol . . . 144
6.3 Tamarin client rules for the STS protocol . . . 145
6.4 Tamarin server rules for the STS protocol . . . 146
6.5 Simple state diagram for the STS protocol . . . 149
6.6 Tamarin verification of the STS secrecy lemma . . . 152
6.7 Partial Tamarin graph for the STS secrecy lemma . . . 153
6.8 Excerpt from partial Tamarin graph for the STS secrecy lemma . . . 153
6.9 Handshake modes for `draft-10` . . . 155
6.10 Tamarin C_1 rule for `draft-10` . . . 158
6.11 Partial client state machine for `draft-10` . . . 159
6.12 Partial server state machine for `draft-10` . . . 159
6.13 Client impersonation attack on `draft-10+` . . . 171

7.1 Handshake modes for `draft-21` . . . 179
7.2 Partial state machine for `draft-21` . . . 183
7.3 `draft-21` (EC)DHE handshake in sub-flights . . . 184
7.4 Tamarin `client_hello` rule for `draft-21` . . . 185
7.5 Part 1 of the full state machine for `draft-21` . . . 187
7.6 Part 2 of the full state machine for `draft-21` . . . 188
7.7 Record layer state machine for `draft-21` . . . 189
7.8 Tamarin lemma map . . . 201

List of Tables

4.1 Password recovery attack parameters 67

5.1 Plaintext recovery attack parameters 107
5.2 Median rank of the maximum likelihood estimate 119

7.1 Tamarin results for `draft-21` 199

Part I

Motivation and Background

Chapter 1

Introduction

Contents

3.1 Post-Deployment Analysis 46
3.2 Pre-Deployment Analysis 53

In this chapter we provide an overview of this thesis. We discuss the motivation for our work and present the overall structure of the thesis.

1.1 Motivation

The Transport Layer Security (TLS) protocol is the *de facto* means for securing communications on the World Wide Web. Initially released as Secure Sockets Layer (SSL) by Netscape Communications in 1995, the protocol has been subject to a number of version upgrades over the course of its 23-year lifespan. Rebranded as TLS when it fell under the auspices of the Internet Engineering Task Force (IETF) in the mid-nineties, the protocol has been incrementally modified and extended with the release of TLS 1.0 [48] in 1999, TLS 1.1 [49] in 2006, and TLS 1.2 [50] in 2008. Since then, TLS has received increasing amounts of attention from the security research community. Dozens of research papers on TLS have been published [8, 10, 11, 16, 18, 19, 26, 29, 31, 33, 34, 37, 40, 42, 56, 67, 68, 71, 77, 80, 83, 84, 87, 90, 96, 105, 108, 111, 114, 117, 121, 148, 149, 151], containing both positive and negative results for the protocol. What began as a trickle of papers has, in the last five years or so, become a flood. Arguably, the major triggers for this skyrocketing in interest from the research community were the TLS Renegotiation flaw of Ray and Dispensa in 2009,[1] and the BEAST[2] [57] and CRIME[3] [58] attacks in 2011 and 2012.

[1] This flaw was rediscovered by Martin Rex as part of a discussion on the TLS WG mailing list in November of 2009. Ray and Dispensa discovered the problem in August of the same year. No formal reference for the attack exists but a description can be found at `http://www.educatedguesswork.org/2009/11/understanding_the_tls_renegoti.html`.

[2] Browser Exploit Against SSL/TLS

[3] Compression Ratio Info-leak Made Easy

The many weaknesses identified in TLS 1.2 and below, as well as increasing pressure to improve the protocol's efficiency (by reducing its latency in establishing an initial secure connection), prompted the IETF to start drafting the next version of the protocol, TLS 1.3, in the Spring of 2014. Unlike the development process employed for earlier versions, the TLS Working Group (WG) has adopted an "analysis-prior-to-deployment" design philosophy, making a concerted effort to engage the research community in an attempt to catch and remedy any weaknesses before the protocol is finalised.

Given the critical nature of TLS, the recent shift in the IETF's design methodology for TLS 1.3, and TLS 1.3 now reaching the end of the standardisation process, we think it pertinent that the TLS standardisation story be told. Prior to the standardisation of TLS 1.3, the TLS WG conformed to a reactive standardisation process – attacks would be announced and the WG would respond to these attacks by either updating the next version of the protocol or by releasing patches for the TLS standard. A number of factors contributed to the adoption of such a standardisation process: protocol analysis tools were not mature enough at the time of design, the research community's involvement in the standardisation process was minimal, and until the first wave of attacks in 2009-2012, attacks on TLS were not considered to be of enough practical importance to warrant making changes with urgency. In contrast, the TLS 1.3 standardisation process has been highly proactive. The availability of more mature analysis tools, the threat of practical attacks, the presence of an engaged research community, and a far more open dialogue with that community have, we contend, enabled this shift in the TLS standardisation process.

In this thesis we present work that falls on both sides of this process shift. We describe two attacks against TLS 1.2 and below, thus contributing to the post-2011 TLS attack era and further highlighting the need for a new version of the protocol. Our attacks exploit weaknesses in the RC4 keystream to recover plaintext protected by RC4 in TLS. The first class of attacks extends the attack ideas of AlFardan *et al.* [10] to recover user passwords, the pre-eminent means of user authentication on the Web. While the attacks in [10] break RC4 when used in TLS in an academic sense, they are far from being practical. For instance, the authors' preferred method for recovering secure cookies requires around 2^{34} encryptions of the target cookie. We enhance the statistical techniques of [10] and exploit specific features of the password setting to produce attacks that are of greater practical significance, showing that good-to-excellent password recovery rates can be achieved using $2^{24} - 2^{28}$ ciphertexts. We demonstrate our attacks against the real-world protocol BasicAuth and

help to render the use of RC4 in TLS indefensible.

Our second class of attacks exploits the Mantin biases [101] to recover plaintexts from RC4-encrypted traffic. Our basic attack targets two unknown bytes of plaintext that are located close to sequences of known bytes of plaintext, a situation which commonly arises when RC4 is used in TLS. We go beyond the plaintext recovery attack to develop a statistical framework that enables us to make predictions about the attack performance and its variants. This framework relies on results from *order statistics*, a well-established field of statistical investigation that, prior to our work, does not appear to have been used extensively in cryptanalysis. Using standard dynamic programming techniques, we extend our basic attack to recover longer plaintexts and show that TLS-protected HTTP cookies can successfully be recovered using 2^{31} ciphertexts, a result which again improves on the attacks described in [10].

We note here that usage of RC4 has dropped rapidly due to the high-profile nature of our attacks, and those by others [10, 148], coupled with the deprecation of RC4 by the IETF in 2015, and the decision by major vendors to disable RC4 in their browsers.[4]

As mentioned, we also include work that concerns TLS 1.3, specifically work that contributes to building confidence in the new protocol's design and catching flaws in a timely fashion. We present an analysis of revision 10 (henceforth referred to as `draft-10`) of the TLS 1.3 specification using the Tamarin prover, a state-of-the-art symbolic analysis tool designed for the automated inspection of security protocols. Our work shows that `draft-10` meets the desired goals of authenticated key exchange and provides the first supporting evidence for the security of complex protocol mode interactions in TLS 1.3, thereby ruling out attacks which exploit the interaction of several handshake modes, such as the Renegotiation and the Triple Handshake [29] attacks against TLS 1.2 and below. Our `draft-10` analysis also highlights an attack against the post-handshake client authentication feature as proposed for inclusion in `draft-11` of the specification. The attack was reported to the TLS WG and informed the next revision of the specification, adding valuable insight into the protocol's design.

The TLS 1.3 specification has been a rapidly moving target, with large changes being effectuated on a fairly regular basis. This has often rendered much of the analysis work

[4] See, for example, `http://www.infoworld.com/article/2979527/security/google-mozilla-microsoft-browsers-dump-rc4-encryption.html`.

'out-of-date' within the space of a few months as substantial changes to the specification effectively result in a new protocol, requiring a new wave of analysis. Building on our `draft-10` work, we contribute to what is hopefully the near-final wave of analysis of TLS 1.3 before its official release, analysing `draft-21` of the specification using the Tamarin prover. The many differences between `draft-10` and `draft-21` make for a very different TLS 1.3 protocol, specifically from a symbolic perspective. As a Tamarin model aims to consider the interaction of all possible handshake modes and variants, changes to these modes as well as the inclusion of new post-handshake features, results in a very different set of mode combinations to be considered when proving security properties. Hence, our work on `draft-21` presents a substantially different model to that of `draft-10`, and takes a far more fine-grained and flexible approach to modelling TLS 1.3. Our `draft-21` analysis reveals an unexpected behaviour in the protocol which has the ability to inhibit strong authentication guarantees in some implementations of the protocol.

The standardisation process for TLS 1.3 has arguably been successful, with several research works [14, 27, 32, 45–47, 54, 55, 61, 61, 85, 91, 95] influencing the design of the protocol. The amount of communication between those who implement TLS and those who analyse TLS has probably never been greater. The results presented in this thesis reinforce the need for a new version of TLS, and highlight the value of the more proactive design process employed by the IETF by exposing several flaws in TLS 1.3 before the protocol is finalised.

1.2 Thesis Structure

This thesis is made up of four parts. Chapters 2 and 3 constitute the remainder of Part I, **Motivation and Background**.

Chapter 2. This chapter provides detail concerning the TLS protocol. We describe the general structure of the protocol, introducing its relevant sub-protocols and security properties. We cover the pertinent technical details of TLS 1.2 and below, and TLS 1.3, remarking on the differences between the various protocol versions. We introduce all necessary details concerning the old and new handshake modes of TLS and briefly cover TLS key derivation.

We note that, where necessary, all other chapter-specific preliminaries are introduced in

the relevant chapters.

Chapter 3. In this chapter we explore the TLS standardisation process, examining factors which may have contributed to the different standardisation cycles employed for TLS 1.2 and below, and TLS 1.3, respectively. We comment on the tools available for analysis, the levels of academic involvement, and the incentives driving the agents involved in the standardisation process.

Part II of this thesis, namely **Attacking TLS 1.2 and Below**, covers our work on the earlier versions of TLS and contains Chapters 4 and 5.

Chapter 4. In this chapter we introduce our password recovery attacks against RC4 in TLS. We describe the RC4 algorithm, discuss the relevant single- and double-byte keystream biases, and provide further background on password distributions. We present a formal Bayesian analysis that combines *a priori* password distribution with keystream distribution statistics to produce *a posteriori* password likelihoods, yielding a procedure which is statistically optimal (if the password distribution is known exactly). We demonstrate the effectiveness of our attacks via extensive simulations and present a proof-of-concept implementation against a widely-used application that makes use of passwords over TLS, namely BasicAuth.

Chapter 5. This chapter covers attacks against RC4 that exploit the Mantin biases – patterns of the form $ABSAB$ that occur in the RC4 keystream with higher probability than expected for a random sequence (A and B are byte values, and S is an arbitrary byte string of some length G). We develop a statistical framework for exploiting these biases which leads to an algorithm that recovers adjacent pairs of unknown plaintext bytes, under the assumption that the target plaintext bytes are in the neighbourhood of known plaintext bytes, a valid assumption in an attack against TLS. Our analysis enables us to make predictions about the number of ciphertexts needed to reliably recover target plaintext bytes by using results from order statistics. We extend the algorithm to recover longer sequences of plaintext bytes, as would be needed to attack 16-byte cookies protected by TLS. We rely on the beam-search and list Viterbi algorithms to achieve this and report on a large range of attack simulations, focussing on a 16-byte target plaintext.

In Part III, **Verifying TLS 1.3**, we present our results on TLS 1.3. These are laid out in

Chapters 6 and 7.

Chapter 6. This chapter covers our symbolic analysis of the TLS 1.3 draft specification using the Tamarin prover. We introduce the symbolic setting and cover the necessary Tamarin fundamentals. We formally model `draft-10` of the specification and encode the desired security properties, as laid out in the draft, using the Tamarin specification language. We use a mixture of automated inspection and manual interaction with the tool to analyse these properties. We also extend this model to include the post-handshake authentication mechanism as suggested for `draft-11`. Our results represent some of the first supporting evidence of the security of several handshake mode interactions in TLS 1.3.

Chapter 7. In this chapter we model and analyse `draft-21` of the TLS 1.3 specification, reforming our `draft-10` model and incorporating the changes made to the specification since `draft-10`. Our work on `draft-21` reinforces the security of several handshake mode interactions in TLS 1.3 for a near-final version of the protocol.

Finally, in Part IV, **Concluding Remarks**, we end with Chapter 8.

Chapter 8. In this chapter we conclude and briefly mention avenues for future work.

Chapter 2

Preliminaries

Contents

4.1 Introduction 63

4.2 Preliminaries 69

4.3 Plaintext Recovery via Bayesian Analysis 78

4.4 Simulation Results 85

4.5 Practical Validation 98

4.6 Conclusion 102

This chapter provides detail concerning the TLS protocol. We describe the general structure of the protocol, introducing its relevant sub-protocols and security properties. We cover the pertinent technical details of TLS 1.2 and below, and TLS 1.3, remarking on the differences between the various protocol versions. We introduce all necessary details concerning the old and new handshake modes of TLS, and briefly cover TLS key derivation.

2.1 The TLS Protocol

TLS is a network protocol designed to provide security services for protocols running at the application layer. TLS runs over transport layer protocols, as depicted in Figure 2.1. Specifically, TLS runs over the Transmission Control Protocol (TCP) [124], a *reliable* network protocol that ensures in-order delivery of network packets in the face of accidental deletion and re-ordering.[1] The primary goal of TLS is to facilitate the establishment of a secure channel between two communicating entities, namely the client and the server.

[1] The Datagram Transport Security Layer (DTLS) protocol [130, 131], which is based on TLS, is designed to run over transport protocols which are not necessarily reliable, such as the User Datagram Protocol (UDP) [125].

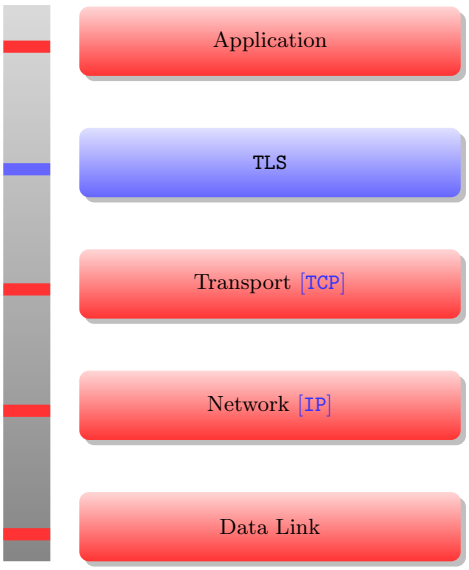


Figure 2.1: Conceptual positioning of TLS within the TCP/IP protocol architecture model [43]. Network layers are represented in red. Network protocols are depicted in blue.

The TLS protocol is made up of a number of sub-protocols, the two most important being the Handshake Protocol and the Record Protocol. The Handshake Protocol negotiates all cryptographically relevant parameters (including the TLS version, the authentication and key exchange method, and what subsequent symmetric key algorithms will be used). It authenticates one (or both) of the communicating entities, and establishes the keys for the symmetric algorithms that will be used in the Record Protocol to protect application data.

For instance, if a client and a server agree on the `TLS_RSA_WITH_AES_128_CBC_SHA256` cipher suite during a TLS 1.2 handshake, then the server will provide an RSA certificate to be used for key exchange and entity authentication purposes. In this example, the Record Protocol will then make use of the Advanced Encryption Standard (AES) in Cipher Block Chaining (CBC) mode for the encryption of application data, and the hash function SHA-256[2] will be used in the Hash-based Message Authentication Code (HMAC) algorithm to provide message authentication.

TLS also includes another sub-protocol, namely, the Alert Protocol. This protocol triggers when errors occur within the operation of the other two sub-protocols. It will provide alert messages to the application layer, indicating the severity of the alert. *Fatal alerts* will result in immediate termination of a TLS connection. *Warning alerts* are informational and do not necessarily result in connection termination. We do not say much more about this

[2]From the Secure Hash Function 2, or SHA-2, family of hash functions.

protocol in the remainder of this chapter given that it is not a focus of the work contained in this thesis.

The TLS Handshake Protocol is intended to negotiate cryptographic keys via the mechanism of Authenticated Key Exchange (AKE). This means that not only are symmetric keys securely established by the client and the server but also that there are guarantees regarding the claimed identities of the communicating peers. The keys established in the Handshake Protocol are then used by the Record Protocol to provide critical security guarantees, including confidentiality and integrity of application data. TLS is intended to provide these guarantees in the presence of an active network attacker,[3] i.e., an attacker that can capture, modify, delete, replay, and otherwise tamper with messages sent over the communication channel. We say more about the desired security properties of TLS in the sections to follow.

2.2 TLS 1.2 and Below

We now describe the structure of TLS 1.2 (and below – the logical structure of these versions is similar), covering the various handshake modes, the Record Protocol, and the intended security goals and properties.

2.2.1 The Handshake Protocol

TLS 1.2 [50] has three types of handshakes, including an initial handshake to set up the TLS session, a renegotiation handshake to update the session's cryptographic parameters, and a resumption handshake for repeated handshakes within the session. We discuss these handshake types now:

Initial Handshake. The message flows for an initial TLS 1.2 handshake are depicted in Figure 2.2.[4] Messages marked with an asterisk are optional or situation-dependent and braces of the type "[. . .]" indicate encryption with the application traffic keys. The client and

[3] We note that throughout this thesis we use the terms 'attacker' and 'adversary' interchangeably.

[4] Our TLS-specific ladder diagrams closely follow the message flows and naming conventions as laid out in the TLS specifications. We include descriptions of the necessary cryptographic parameters in the textual descriptions of the diagrams.

the server exchange `ClientHello` and `ServerHello` messages in order to agree on a cipher suite and to exchange nonce values. The communicating entities also exchange cryptographic parameters (`ServerKeyExchange`, `ClientKeyExchange`) that allow for the derivation of the `pre-master secret`. Certificates and the corresponding verification information (`Certificate`, `CertificateVerify`) are sent for the purposes of entity authentication. A `master secret` is derived from the nonce values and the `pre-master secret`, and in turn used in the derivation of the application traffic keys to be employed by the Record Protocol. The `Finished` message comprises a Message Authentication Code (MAC) over the entire handshake, ensuring that the client and the server share an identical view of the handshake and that an active attacker has not altered any of the handshake messages.

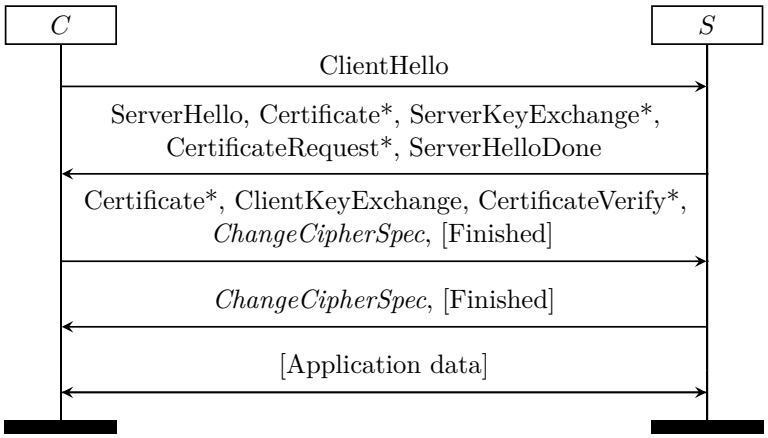


Figure 2.2: TLS 1.2 initial handshake

The Handshake Protocol runs over the Record Protocol, initially with null encryption and MAC algorithms. The `ChangeCipherSpec` messages signal the intent to start using newly negotiated cryptographic algorithms and keys; these messages are not considered part of the handshake but instead are the messages of a peer protocol, the ChangeCipherSpec protocol. As the `Finished` messages come after the `ChangeCipherSpec` messages, they are protected using the application data traffic keys derived in the handshake. These messages, then, are the first to be protected as part of the Record Protocol. They are followed by application data messages, now protected by the Record Protocol.

TLS 1.2 allows for static RSA [135] key exchange, static Diffie-Hellman key exchange [51], as well as ephemeral Diffie-Hellman key exchange. In the case of an RSA key exchange, the client will select a `pre-master secret` value and encrypt it with the server's RSA

public key, received in the server's `Certificate` message. This will form the client's `ClientKeyExchange` message. In the case of (ephemeral) Diffie-Hellman key exchange, the client and the server will exchange Diffie-Hellman key shares in the `ClientKeyExchange` and `SeverKeyExchange` messages, respectively.

Renegotiation. The cryptographic parameters established in the initial handshake constitute a TLS *session*. A session can be updated via a renegotiation handshake. This is a full handshake that runs under the protection of an already established TLS session. This mechanism allows cryptographic parameters to be changed (for example, upgraded), or client authentication to be demanded by a server (in the event that it was not requested previously).

Session Resumption. In order to avoid the expensive public key operations in repeated handshakes, TLS 1.2 also offers a lightweight resumption handshake in which a new `master secret` is derived from the old `pre-master secret` and new nonces, thus forcing fresh application data keys. Each such resumption handshake leads to a new TLS *connection* within the existing session; many connections can exist in parallel for each session. A TLS 1.2 resumption handshake is depicted in Figure 2.3.

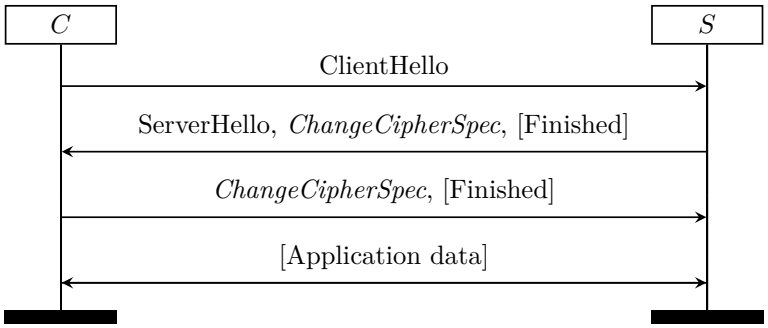


Figure 2.3: TLS 1.2 resumption handshake

Pre-Shared Keys (PSKs), in this case symmetric keys established out-of-band prior to communication, are allowed to be used in TLS 1.2 and below for authentication purposes. Their use is described in TLS extensions RFC 4279 [59] and RFC 5487 [17]. They are intended to avoid the use of expensive public key operations.

2.2.2 The Record Protocol

The Record Protocol provides a secure channel for transmission of application data (as well as Handshake Protocol and Alert messages). In TLS 1.0 and 1.1, it uses a MAC-then-Encode-then-Encrypt (MEE) construction, with the MAC algorithm being HMAC instantiated with a range of hash functions and the encryption algorithm being instantiated with CBC-mode of a block cipher or the RC4 stream cipher. Sequence numbers are included in the cryptographic processing, creating a stateful secure channel in which replays, deletions and re-orderings of TLS records can be detected. TLS 1.2 added support for Authenticated Encryption with Associated Data (AEAD) schemes, with AES in Galois Counter Mode (AES-GCM) being an increasingly popular option [138].

Our work in Chapters 4 and 5 of this thesis describe attacks on the RC4 stream cipher when used in TLS 1.2 and below. We cover details concerning the operation of this cipher in TLS in Chapter 4.

2.2.3 Security Properties

The TLS 1.2 Handshake Protocol is responsible for establishing the symmetric session keys that will be used as part of the Record Protocol to provide confidentiality and integrity for application data messages. The Handshake Protocol also (optionally) authenticates communicating peers whilst establishing these keys. The stated goals and security properties of TLS 1.2 are discussed in the Security Analysis section of the TLS 1.2 specification, namely, Appendix F. According to this appendix, the Handshake Protocol should satisfy the following TLS security properties:

1. **Secrecy of Session Keys.** Upon completion of the handshake, the client and the server should have established a set of session keys which are known to the client and the server only.

2. **Peer (Entity) Authentication.** In the case of unilateral authentication, upon completion of the handshake, if a client `C` believes it is communicating with a server `S`, then it is indeed `S` who is in the server role. An analogous property for the server also holds in the case of mutual authentication, i.e., the server has a guarantee regarding

the identity of the client. Both forms of authentication are optional. It is also possible that neither party authenticates. In this handshake variant, Diffie-Hellman is used as the method of key exchange but, as noted in the specification, it is subject to the well-known Man-In-The-Midde (MITM) attack in which an active attacker exploits the lack of entity authentication to establish distinct session keys with the client and the server, respectively, with the honest parties being oblivious to the MITM's presence.

3. **Integrity of Handshake Messages.** An active attacker should not be able to successfully tamper with the handshake messages, potentially causing the client and the server to adopt weak cipher suites.

Appendix D of the specification also addresses denial of service attacks as well as version rollback attacks. Denial of service attacks against TLS are possible owing to the reliance on the underlying TCP protocol. The opening up of a large number of TCP connections, carrying TLS protocol messages, could overwhelm a server with TLS cryptographic computations, and manipulation of the TCP protocol itself can also cause the termination of TLS connections. TLS does not claim to be able to thwart this class of attacks. In terms of version rollback attacks, the specification includes a countermeasure involving the alteration of RSA encryption padding but this solution is not secure against all attackers (further details can be found [50]).

The Record Protocol should satisfy the following desired property:

4. **Protection of Application Data.** This property is comprised of two sub-properties:

   (i) *Confidentiality of Application Data.* Application data exchanged between a client and a server should be known only to the client and the server.

   (ii) *Integrity of Application Data.* The unauthorised manipulation of application data exchanged between a client and a server should be detectable.

Showing that the Record Protocol confidentiality property is not achieved when RC4 is used as the Record Protocol encryption algorithm in TLS 1.2 and below is the focus of our work in Chapters 4 and 5.

We note here that TLS versions 1.0, 1.1 and 1.2 are all currently in use, with TLS 1.2 being the most popular for servers at the time of writing (according to SSL Pulse[5]).

2.3 TLS 1.3

Owing to the many attacks against TLS 1.2 and below, as well as pressure to improve the protocols efficiency, since the Spring of 2014 the IETF has been working on the next version of the protocol, TLS 1.3. In a large structural departure form TLS 1.2, the main design goals of TLS 1.3 include [146]:

(i) encrypting as much of the handshake as possible,

(ii) re-evaluating the handshake contents,

(iii) reducing handshake latency – introducing a one Round-Trip Time (1-RTT) exchange for full handshakes and a zero Round-Trip Time (0-RTT) mechanism for repeated handshakes, and,

(iv) updating the record protection mechanisms.

We now discuss how TLS 1.3 meets these four requirements:

Handshake Encryption. The motivation behind handshake encryption is to reduce the amount of observable data to both passive and active adversaries [146]. In contrast to TLS 1.2, which only provides communicating entities with session keys to protect application data, TLS 1.3 provides for the establishment of additional session keys to be used for handshake encryption purposes. Handshake encryption begins immediately after the handshake keys have been negotiated via a Diffie-Hellman exchange.

Handshake Contents. As will be discussed in the following section, the handshake structure has been reworked for efficiency purposes. An additional server message has been included to accommodate the event of a parameter mismatch, and compression of application data has been removed. Static Diffie-Hellman and RSA have been removed in favour of the Perfect Forward Secrecy (PFS)-supporting Ephemeral Diffie-Hellman

[5] Available at `https://www.ssllabs.com/ssl-pulse/`.

(DHE) and Elliptic Curve Ephemeral Diffie-Hellman (ECDHE) key exchange modes.[6] RSA certificates are still being used for entity authentication purposes in both the DHE and ECDHE modes. Server-side signatures have been mandated in all handshake modes.

Handshake Latency. The TLS 1.2 handshake required a two Round-Trip Time (2-RTT) exchange prior to communicating entities being able to transmit application data. The handshake has been reworked in TLS 1.3 to require just 1-RTT if no parameter mismatches occur.

TLS 1.3 also includes a 0-RTT option in which the client is able to send application data as part of its first flight of messages, offering a clear efficiency advantage over TLS 1.2. Additionally, the pre-existing mechanism for PSKs has been extended to cover session resumption. This mode also requires a single round trip, and less computation than a full handshake. We describe its details when discussing PSKs and session resumption in Sections 2.4 and 2.5.

Record Protection Mechanisms. The earlier versions of TLS used the MAC-then-Encrypt generic composition scheme as a record protection mechanism. This scheme is not secure in general [22], and while it is still used today in TLS 1.2, there was a proposal to replace it by the Encrypt-Then-MAC paradigm (in RFC 7366 [74]). Similarly, when Krawczyk [89] announced the OPTLS protocol on the TLS mailing list, a protocol intended to serve as a theoretical basis for TLS 1.3, he stated it would use Encrypt-then-MAC for record protection. Ultimately, the TLS WG decided that TLS 1.3 would avoid generic composition schemes and only use block ciphers that can operate in AEAD modes [106]. All non-AEAD ciphers have thus been removed in TLS 1.3.

At the time of writing, TLS 1.3 has been through twenty-six draft iterations. In the sections to follow we discuss two of these drafts, including the protocol details necessary for the understanding of the work presented in Chapters 6 and 7.

[6] Academic works often refer to Perfect Forward Secrecy using lowing case letters. Where suitable, we adopt this convention.

2.4 TLS 1.3 draft-10

We now discuss draft-10 of the TLS 1.3 protocol, covering the draft's Handshake Protocol, its Record Protocol and its claimed security goals and properties.

2.4.1 The Handshake Protocol

Some of the most significant changes in TLS 1.3 are due to the newly introduced handshake mechanisms. Here we provide a brief overview of these different handshake modes, starting with a description of the regular, initial handshake.

Initial (EC)DHE Handshake. The solid message flows in Figure 2.4 represent this handshake. Again, messages marked with an asterisk are optional or situation-dependent, and the CertificateRequest, Certificate and CertificateVerify protocol messages followed by an asterisk can be omitted if only unilateral (server) authentication is required. Braces of the type { } indicate encryption under the handshake traffic keys, whereas braces of the type [ ] indicate encryption under the application traffic keys.

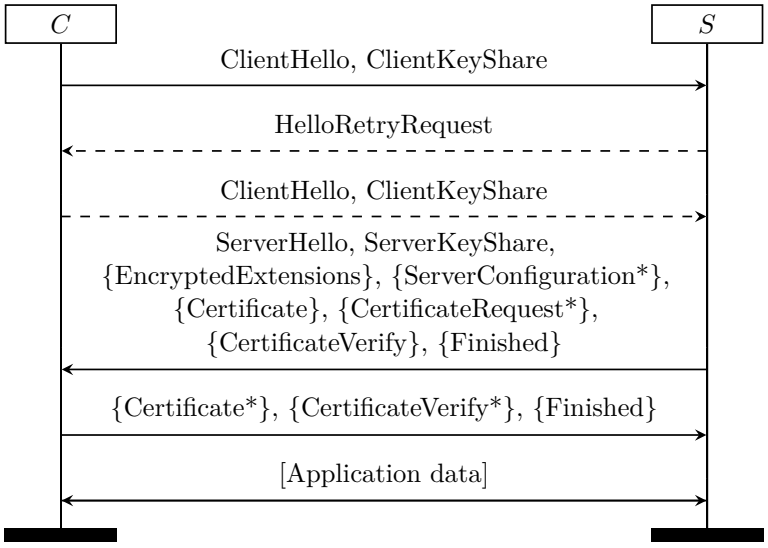


Figure 2.4: draft-10 (EC)DHE handshake

A client sends a server an offer of cryptographic parameters, including a client nonce, that are later used to establish session keys (ClientHello). It also sends freshly generated

Diffie-Hellman key shares along with the associated set of groups (`ClientKeyShare`). The server responds with its choice of cryptographic parameters, including a server nonce and a selected group from among those offered by the client (`ServerHello`). The server also sends its own freshly generated Diffie-Hellman key share (`ServerKeyShare`), extensions not used for key establishment (`EncryptedExtensions`) and an optional semi-static (EC)DH key share to be used in later handshakes (`ServerConfiguration`). Also included in the server's first flight are its public key certificate for authentication purposes (`Certificate`), an optional request for the client's certificate in the case that mutual authentication is desired (`CertificateRequest`), and a signature on all messages exchanged thus far (`CertificateVerify`). The server's `Finished` message comprises a MAC over the entire handshake using a handshake key derived from the Diffie-Hellman key shares. Finally, if the client received a request for authentication, the client either sends its own certificate (`Certificate`) and a signature on the whole handshake thus far (`CertificateVerify`), or a blank certificate representing no authentication. As in the server's case, the client's `Finished` is a MAC over the entire handshake using a handshake key derived from the Diffie Hellman key shares. The purpose of the `Finished` messages is to provide integrity of the handshake.

If the client does not supply an appropriate key share in its first flight (it may suggest groups that are unacceptable to the server, for instance), the server transmits a `HelloRetryRequest` message in order to entice the client to change its key share offer. Upon receipt of this message, the client should send a newly generated key share. These messages are indicated as dashed arrows in Figure 2.4. If no common parameters can be agreed upon, the server will send a `handshake_failure` or `insufficient_security` alert and the session will be aborted.

0-RTT. Following the initial handshake in which the server provides the client with a semi-static (elliptic curve) Diffie-Hellman share, the client is able to use this share to generate a shared key which it then uses to encrypt early data. Figure 2.5 depicts the `draft-10` 0-RTT handshake. The client's `EarlyDataIndication` value signals a 0-RTT handshake, which the server can choose to ignore (the server would then not process the early data and a 1-RTT handshake would ensue). Braces of the type ( ) indicate encryption under the early traffic keys derived from the server's semi-static key share and the client's ephemeral key share.

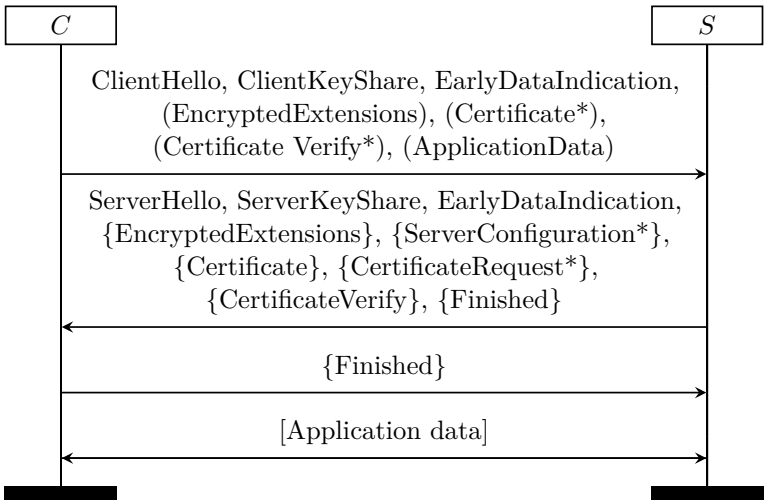


Figure 2.5: `draft-10` 0-RTT handshake

PSKs and Session Resumption. TLS 1.3 effectively merges the PSK and session resumption functionalities of TLS 1.2 into a single handshake mode. There are two possible sources of PSKs: new session tickets (NSTs) and out-of-band mechanisms. While the former are specified in `draft-10`, the latter are not entirely clarified with regards to their intended implementation or assumed security properties (this becomes more explicit in later drafts). Figure 2.6 depicts a PSK handshake following an initial handshake. Note that a new session ticket is sent by the server directly after receiving the client's `Finished` message in the initial handshake.

In this PSK resumption handshake, the client sends a key share in its first flight to allow for the server to decline resumption and fall back to the full (EC)DHE handshake. The `PreSharedKeyExtension` value indicates the identity of the PSK to be used in the exchange. We note that a PSK handshake need not only take the form of a resumption handshake; if a client and a server share an existing secret, a PSK handshake may be an initial handshake. PSKs may also be used in conjunction with an (EC)DHE exchange so as to provide forward secrecy; the corresponding mode is called PSK-DHE.

Key Derivation. In contrast to TLS 1.2, TLS 1.3 employs the use of handshake traffic keys as well as application traffic keys. This keying material is derived from two secrets, namely the ephemeral secret (`es`) and the static secret (`ss`). In the 1-RTT (EC)DHE handshake, `es` and `ss` are identical with the secret being derived from the ephemeral client

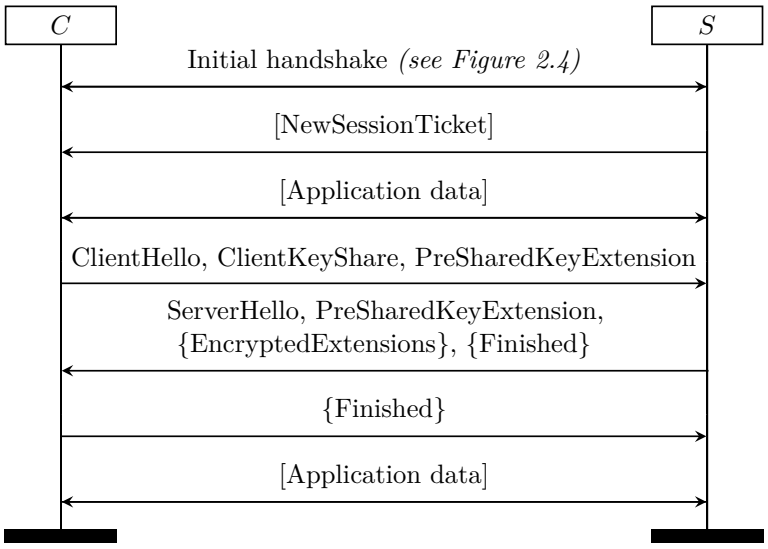


Figure 2.6: draft-10 PSK resumption handshake (after an initial handshake)

and server key shares. In a PSK handshake, these two values are again identical and take on the value of the PSK. In PSK-DHE mode, es is derived from the ephemeral client and server key shares and ss is the PSK. In a 0-RTT handshake, es is again derived from the ephemeral client and server key shares and ss is computed using the server's semi-static key share and the client's ephemeral key share.

The secrets described above are used as inputs to a HMAC-based Key Derivation Fucntion (HKDF) [86,88] in order to derive a master secret ms. This secret, in turn, is used in HKDF computations to derive a resumption secret rs, an exporter secret exs, and application data keys. Handshake traffic keys are derived from es, and early traffic keys, as well as the finished secret fs, are derived from ss. These secrets and keys are derived according to the schematic presented in Figure 2.7.

Another input to HKDF computations is the handshake_hash. This consists of a hash of all the handshake messages, including all client and server messages, up to the present time but excluding the Finished messages. The final value of the handshake_hash is called the session_hash. As such, the session keys established are cryptographically bound to both of the shared secrets negotiated, and rely on both parties having a matching view of the handshake transcript.

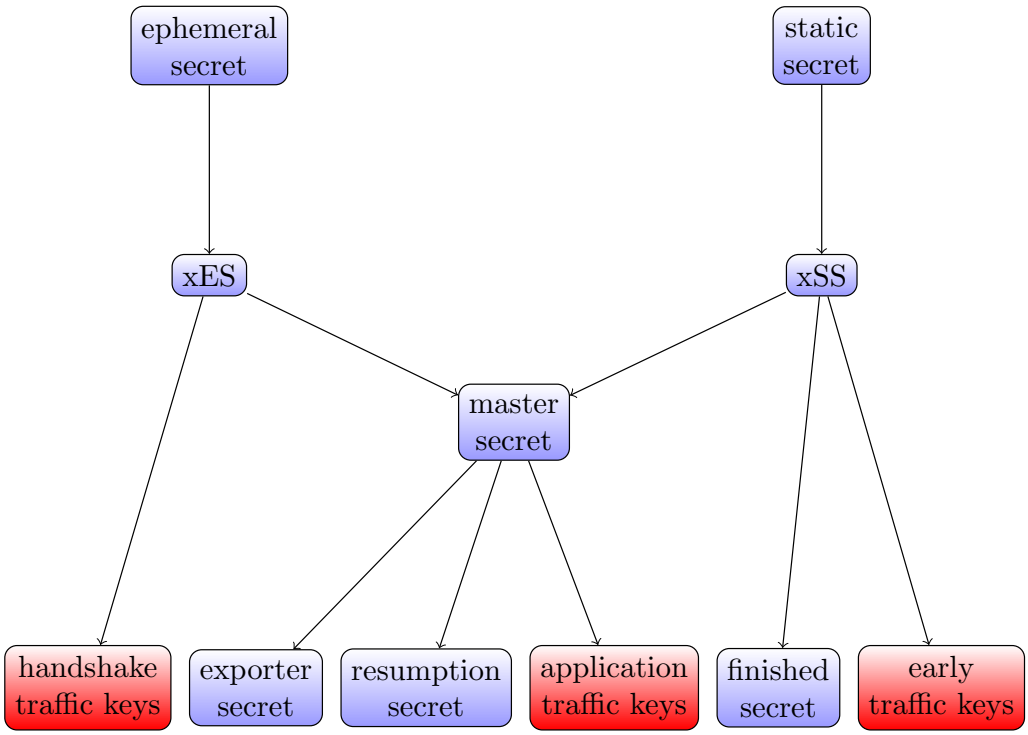


Figure 2.7: Key computation hierarchy for `draft-10`. Key computation secrets are represented in blue and traffic keys are depicted in red. `xES` and `xSS` are HKDF outputs derived using the ephemeral secret and the static secret, respectively. Exporter secrets are created for application layer protocols wishing to leverage TLS keying material for their own purposes, outside of the already provided TLS functionality, as per RFC 5705 [126]. Image adapted from [134].

2.4.2 The Record Protocol

Using keys established in the Handshake Protocol, the TLS 1.3 Record Protocol is responsible for providing confidentiality and integrity of application data messages. It operates on fragments (manageable blocks) of messages, protecting these fragments via the use of AEAD mechanisms. As stated in Section 2.3, the list of supported symmetric algorithms to be used as part of the Record Protocol now *only* includes AEAD algorithms. Upon receipt of protected record fragments, the Record Protocol verifies, decrypts and reassembles application data messages for delivery to the application protocol.

2.4.3 Security Properties

The TLS Record Protocol is claimed to provide confidentiality and integrity of application data. The TLS Handshake Protocol is claimed to allow unilateral or, optionally, mutual entity authentication, as well as establishing shared secrets that are unavailable to eaves-

droppers and adversaries who can place themselves in the middle of the connection. The handshake is claimed to be reliable: no adversary can modify the handshake messages without being detected by the communicating parties.

The security properties for draft-10 are referred to in Appendix D (the security analysis section) of the specification [127]. However, we note that this appendix contains the disclaimer "Todo: Entire security analysis needs a rewrite". As we will see in the next section, this appendix did indeed get rewritten in subsequent drafts. The security properties that the Handshake Protocol is required to satisfy (inferred from Appendix D) include the following:

1. **Secrecy of Session Keys.** Upon completion of the handshake, the client and the server should have established a set of session keys which are known to the client and the server only.

2. **Perfect Forward Secrecy (PFS).** In the case of compromise of either party's long-term key material, sessions completed before the compromise should remain secure. This property is not claimed to hold in the PSK-only handshake mode, nor in the 0-RTT handshake mode.

   In a PSK-only handshake, if compromised, the PSK could be used to decrypt all messages previously protected by the PSK. In the 0-RTT case, the client is the only party to have provided freshness, therefore early data messages may be replayed. In addition, the security of the early data depends on the semi-static (elliptic curve) Diffie-Hellman share, which may have a considerable validity period, and therefore a large attack window. For these reasons, early data cannot be considered to be forward secure.

3. **Peer (Entity) Authentication.** In the case of unilateral authentication, upon completion of the handshake, if a client C believes it is communicating with a server S, then it is indeed S who is in the server role. An analogous property for the server also holds in the mutual authentication case. Authentication of the server is mandatory in all handshake modes. Mutual authentication, i.e., additional authentication of the client, is optional.

4. **Integrity of Handshake Messages.** An active attacker should not be able to successfully tamper with the handshake messages, potentially causing the client and

the server to adopt weak cipher suites.

The above properties form the focus of our work in Chapter 6. Besides the required need for the protection of application data via the Record Protocol and the properties listed above, as in the TLS 1.2 specification, Appendix D of the `draft-10` specification also addresses denial of service attacks as well as version rollback attacks. As stated previously, TLS does not claim to be able to thwart denial of service attacks. Defending against version rollback attacks, i.e., attacks which force the use of previous, potentially weaker versions of TLS (or SSL), is rolled into the *Downgrade Protection* property mentioned in the next section.

2.5 TLS 1.3 `draft-21`

In comparison to `draft-10` of the TLS 1.3 protocol, `draft-21` [128] incorporates an altered 0-RTT handshake mode, a revised key schedule, and new post-handshake mechanisms, i.e., handshake-type mechanisms that can be executed at any time after a TLS 1.3 handshake has been completed.

2.5.1 The Handshake Protocol

Initial (EC)DHE Handshake. In an initial (EC)DHE handshake, as depicted in Figure 2.8, the client sends a `ClientHello` message containing a random nonce, i.e., a freshly generated random value, and a list of symmetric algorithms. The client also sends a set of Diffie-Hellman key shares and the associated groups, `ClientKeyShare`, and potentially some other extensions (such as the preferred set of signature algorithms to be used by the server, for instance).

Upon receipt of a `ClientHello` message, the server selects appropriate cryptographic parameters for the connection and responds with a `ServerHello` message. This message contains a server-generated random nonce, an indication of the selected parameters and potentially some other extensions. The server also sends a `ServerKeyShare` message along with an `EncryptedExtensions` message and optionally a `CertificateRequest` message.

The `ServerKeyShare` contains the server's choice of group and its ephemeral Diffie-Hellman

key share. The client and server key shares are used to compute handshake and application traffic keys. The `EncryptedExtensions` message contains material that is not necessary for determining cryptographic parameters. For instance, the draft specification lists the server name and the maximum TLS fragment length as possible values to be sent in this message. The `CertificateRequest` message indicates that the server requests client authentication in the mutual authentication case.

The server will also send a `Certificate` message, containing the server's certificate and a `CertificateVerify` message, which is a digital signature over the current transcript. These two messages allow the client to authenticate the server. The server also sends a `Finished` message. This message is a MAC over the entire handshake, providing key confirmation and binding the server's identity to the computed traffic keys. As the server is now in a position to establish the application data traffic keys, the server may send protected application data at this point. However, as the client has not (yet) been authenticated, this application data is being sent to an unauthenticated peer.

The client responds with `Certificate` and `CertificateVerify` messages, if requested, and then sends its own `Finished` message. These message flows are depicted in Figure 2.8. It can be the case that the groups sent by a client are not acceptable to the server and the server may respond with a `HelloRetryRequest` message. This indicates to the client which groups the server will accept, and provides the client with the opportunity to respond with an appropriate key share before returning to the main handshake. The associated retry request messages are indicated by dashed message flows in Figure 2.8.

PSKs and Session Resumption. The PSK handshake variant allows a client to use a key established out-of-band to start a new session, or to use a new session ticket (NST) established in a previous handshake to resume the session. In the event that a PSK has been established, a client and a server can begin communicating without a Diffie-Hellman exchange. This is potentially attractive for low-power environments. However, without an ephemeral Diffie-Hellman exchange, the connection loses perfect forward secrecy. In a PSK handshake, the server authenticates via a PSK. By combining a PSK with a Diffie-Hellman exchange this mode maintains perfect forward secrecy. The PSK handshake has the intended aim of avoiding the use of expensive public-key operations and, in the case of a resumption and use of an NST, ties the security context of the new connection to the original connection. In `draft-21`, the sending of NSTs acts as a post-handshake

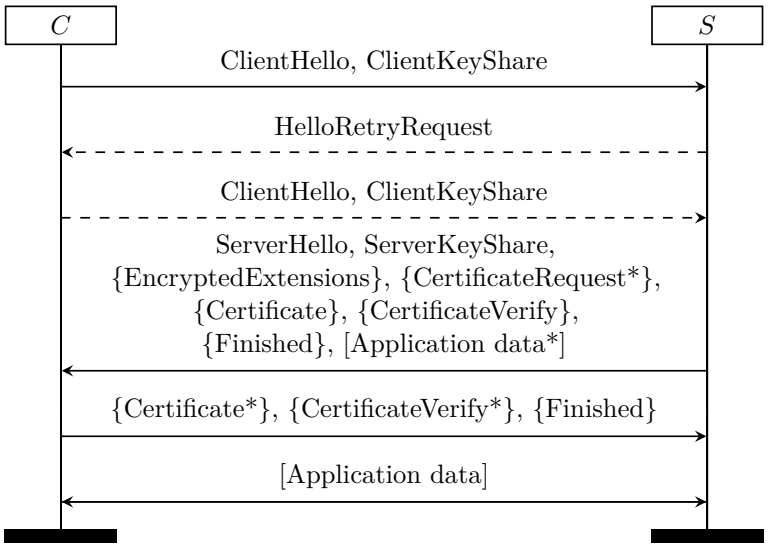


Figure 2.8: `draft-21` (EC)DHE handshake

mechanism which we discuss in more detail in Section 2.5.3. As a server may reject a resumption attempt made by a client, the specification recommends that the client supplies an additional (EC)DHE key share with its pre-shared key (PSK) when trying to resume a session. Figure 2.9 depicts a PSK resumption handshake.

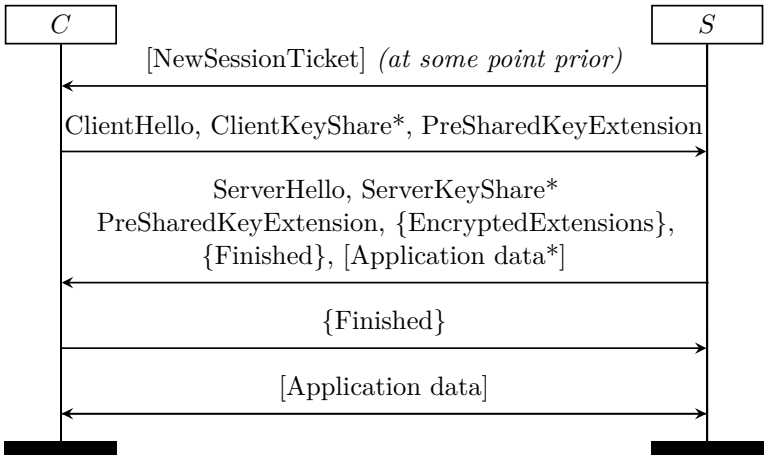


Figure 2.9: `draft-21` PSK resumption handshake

In contrast to `draft-10` of TLS 1.3, `draft-21` includes a *PSK binder* mechanism. A PSK binder is a value that binds a PSK to the handshake where the PSK is offered by a client in the `PreSharedKeyExtension`, and if a PSK was generated by the server in-band, to

the handshake where it was generated. A `PreSharedKeyExtension` can contain multiple binders arranged in a list, where each binder is computed as an HMAC over a hash of the `ClientHello` up to but excluding the binder list.

0-RTT. A client can use a PSK to send application data in its first flight of messages, reducing the latency of the connection. As noted in the `draft-21` specification, this data is not protected against replay attacks. If the communicating entities wish to take advantage of the 0-RTT mechanism, they should provide their own replay protection at the application layer. A 0-RTT handshake is depicted in Figure 2.10. The client's early data is protected by a PSK only, indicated by braces of the type ( ).

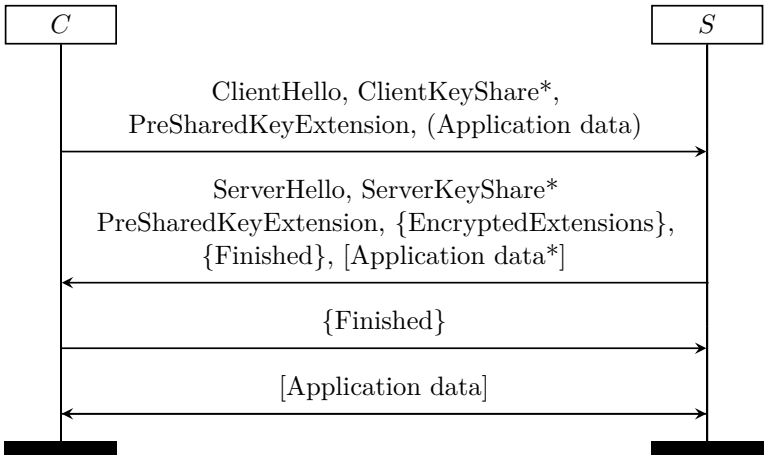


Figure 2.10: `draft-21` 0-RTT handshake

Key Derivation. A TLS 1.3 handshake will generate a set of keys on which both the client and the server agree. The `draft-21` specification defines a key schedule which uses the repeated application of an HKDF to combine the secret inputs with fixed labels so as to generate a set of independent keys.

The key schedule has two secret inputs, the (EC)DHE key share and the PSK. Depending on the handshake mode, either one or both of these will be used. The key schedule also includes the transcript hash in the key derivation. As the transcript includes nonces, even if the secret inputs are repeated, the generated keys are guaranteed to be independent.[7] The respective key derivation secrets are derived according to the schematic presented in Figure 2.11.

[7] Of course, collisions may occur with low probability

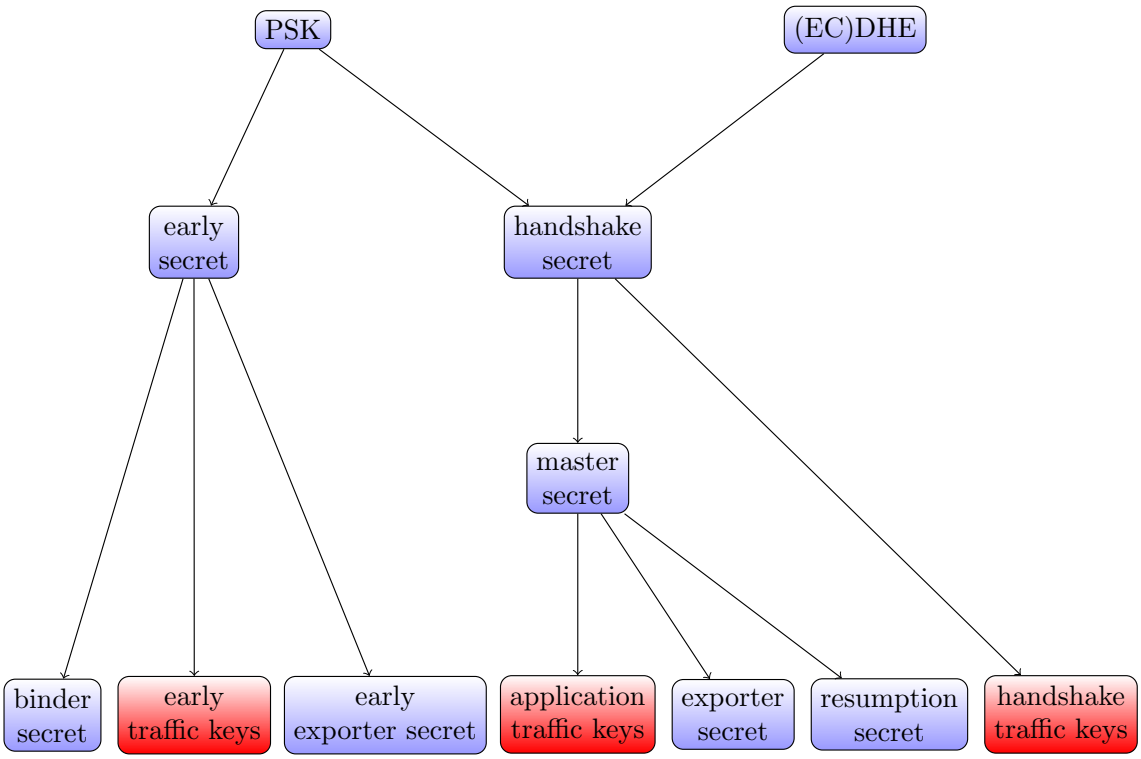


Figure 2.11: Key computation hierarchy for `draft-21`. Key computation secrets are represented in blue and traffic keys are depicted in red. Exporter secrets are created for use in application layer protocols as previously described, and the binder secrets are for computation of PSK binder values.

2.5.2 The Record Protocol

In comparison to `draft-10`, the operation of the Record Protocol has remained largely unchanged. The Record Protocol section in `draft-21` is, however, more precise and explicitly discusses record boundary details. As this is an implementation concern, we do not consider it in our modelling of the protocol in Chapter 7.

2.5.3 Post-Handshake Mechanisms

The TLS 1.3 `draft-21` specification describes three post-handshake mechanisms:

New Session Ticket (NST). After a successful handshake, the server can issue an NST at any time. These tickets specify a binding to a PSK (derived from the resumption master secret) which can be used in subsequent handshakes. This differs to the `draft-10` specification in which NSTs can only be sent immediately after an initial (EC)DHE handshake.

Post-Handshake Client Authentication. After a successful handshake, the server can send a `CertificateRequest` message. If the client responds with an acceptable certificate (and the accompanying digital signature), then the server might authenticate the client. However, because the specification allows certificates to be rejected 'silently', the client cannot be sure of its authentication status in general. We discuss this behaviour in greater detail in Chapter 7.

Key Update. After a successful handshake, either party can request an application data key update. As the read and write keys for application data are independent, either party can immediately update their write key after requesting a key update. Current application data keys are used as inputs to an HKDF function to create the new application data keys.

2.5.4 Security Properties

The TLS 1.3 Handshake Protocol negotiates cryptographic keys which are then used by the Record Protocol to provide critical security guarantees, including confidentiality and integrity of messages. As stated in the sections above, TLS 1.3 makes use of independent keys to protect handshake messages and application data messages. Protection of the handshake messages starts with the server's `EncryptedExtensions` message, and in the majority of handshake modes, protection of application data messages occurs after the transmission of the server and client `Finished` messages, respectively. In `draft-21`, in the case of a zero round trip time (0-RTT) handshake, any application data in the client's first flight of messages is protected with a PSK.

The TLS 1.3 `draft-21` specification [128, Appendix E.1] lists eight properties that the Handshake Protocol is required to satisfy:

1. **Establishing Identical Session Keys**. Upon completion of the handshake, the client and the server should have established a set of session keys on which they both agree.

2. **Secrecy of Session Keys**. Upon completion of the handshake, the client and the server should have established a set of session keys which are known to the client and the server only.

3. **Peer (Entity) Authentication**. In the unilateral case, upon completion of the handshake, if a client C believes it is communicating with a server S, then it is indeed S who is executing the server role. An analogous property for the server also holds in the mutual authentication case. Authentication of the server is mandatory and mutual authentication is optional.

4. **Uniqueness of Session Keys**. Each run of the protocol should produce distinct, independent session keys.

5. **Downgrade Protection**. An active attacker should not be able to force the client and the server to employ weak cipher suites, or older versions of the TLS protocol.

6. **Perfect Forward Secrecy (PFS)**. In the case of compromise of either party's long-term key material, sessions completed before the compromise should remain secure. This property is not claimed to hold in the PSK key exchange mode.

7. **Key Compromise Impersonation (KCI) Resistance**. Should an attacker compromise the long-term key material of party A, the attacker should not be able to use this key material to impersonate an uncompromised party in communication with A.

8. **Protection of Endpoint Identities**. The identity of the server cannot be revealed by a passive attacker that observes the handshake, and the identity of the client cannot be revealed even by an active attacker that is capable of tampering with the communication.

As stated previously, 0-RTT mechanisms allow for replay of early data across sessions. The draft-21 specification recommends addressing this at the application layer. We model and discuss more fully the properties described above in Chapter 7 of this thesis. The draft-21 specification refers to RFC 3552 [129] for an informal description of the TLS 1.3 threat model. This model assumes a Dolev-Yao adversary [53] – an adversary that can perform MITM attacks by being able to replay, insert, delete, and modify messages at will. We discuss this model, as well as our enhancements to it, in Chapter 7.

The material presented in this chapter informs the chapters to follow. Chapter 3 draws on both the TLS 1.2 (and below) and TLS 1.3 information to discuss the different standardisation procedures followed for TLS 1.2 and below, and TLS 1.3, respectively. Chapters 4

and 5 are informed by the material on TLS 1.2 and below, and Chapters 6 and 7 make use of the material provided on TLS 1.3

Chapter 3

Reactive and Proactive Standardisation of TLS

Contents

5.1 Introduction 104
5.2 Preliminaries 108
5.3 Plaintext Recovery using the Mantin Biases 111
5.4 Recovering Multiple Plaintext Bytes 123
5.5 Simulation Results 127
5.6 Conclusion 130

In this chapter we explore the TLS standardisation process, examining factors which may have contributed to the different standardisation cycles employed for TLS 1.2 and below and TLS 1.3, respectively. We comment on the tools available for analysis, the levels of academic involvement, and the incentives driving the agents involved in the standardisation process.

3.1 Post-Deployment Analysis

The standardisation process for TLS 1.2 and below can arguably be described as reactive. Following the announcement of attacks against the protocol, the TLS WG has responded by either making the necessary changes to the next version of the standard or by releasing interim recommendations or extensions. This conforms to what we will term the *design-release-break-patch* cycle of standards development. In what follows, we outline this development process as it pertains to TLS, highlighting high-profile attacks against the protocol and the IETF's responses to these attacks. We note that we provide just enough

technical detail concerning these attacks so as to effectively examine the IETF's reactions. This chapter is not intended to give a full technical account of each of the attacks mentioned; the applicable references are provided for interested readers.

We note that each TLS version builds on the previous version, incorporating changes where necessary. All TLS versions up to and including TLS 1.2 are currently in use, with clients and servers often supporting more than one version. At the time of writing, almost 85% of sites probed in the SSL Pulse[1] survey support TLS 1.1, with support of TLS 1.0 and TLS 1.2 both being in the region of 90%.[2]

3.1.1 Design, Release, Break, Patch

The TLS standard officially sprang to life with a decision by the IETF to standardise a version of the Secure Sockets Layer (SSL) protocol[3] in 1996. The growing need to support e-commerce and hence the growing deployment of the SSL protocol prompted the IETF to this course of action. At this stage, two versions of SSL existed in the public domain, namely SSLv2 and SSLv3 [66]. SSLv2 had a number of weaknesses, in particular offering no defence against downgrade attacks. It was finally deprecated by the IETF in [147], published in 2011.

In 1998, Bleichenbacher published an attack on RSA when encryption used the PKCS #1[4] encoding scheme [81], affecting SSLv3 [37]. The attack targets the RSA-encrypted pre-master secret sent from client to server (see Section 2.2.1) by using the distinctive server-generated PKCS #1-padding error message as an oracle. Successive, adaptive calls to this oracle allow an attacker to narrow in on the value of the pre-master secret, and once this is obtained, the attacker is able to derive the symmetric keys used in the connection. The TLS 1.0 standard [48] briefly addresses this attack in a two-paragraph note that describes the following countermeasure: a server that receives an incorrectly formatted RSA block should use a pre-generated, random 48-byte value as the pre-master secret instead, thereby eliminating the oracle. The Bleichenbacher attack has been re-enabled (in

[1] SSL Pulse serves as a global dashboard, monitoring between 135,000 and 150,000 SSL- and TLS-enabled websites. Websites are monitored over time and are selected based on the Alexa list of the most popular websites worldwide. The dashboard is available at `https://www.trustworthyinternet.org/ssl-pulse/`.

[2] Support statistics for February 2018. Retrieved from `https://www.trustworthyinternet.org/ssl-pulse/`.

[3] Designed by Netscape Communications in the 1990s.

[4] The first family of Public-Key Cryptography Standards (PKCS).

various forms) in several works [78, 83, 108], the most recent case being DROWN [16], a cross-protocol attack targeting all versions of TLS running on servers that also support SSLv2. Surprisingly, a large number of servers still support this legacy version of the protocol.[5]

Following the release of TLS 1.0 [48], the first significant attack against TLS appears to be Vaudenay's padding oracle attack [42, 149]. This attack exploits the specific Cipher Block Chaining (CBC) mode padding format used by TLS in its MAC-then-Encode-then-Encrypt (MEE) construction in the Record Protocol. The TLS WG initially responded to the attack by adding a countermeasure to the attack in the TLS 1.1 specification [49]. This was intended to equalise the running time of the reverse of the MEE processing – decryption, decoding, MAC verification (DDM). This knowingly left a small timing channel, but it was not believed to be exploitable. A decade later, in 2013, AlFardan and Paterson [11], in their Lucky 13 attack, showed that in fact it was exploitable in a sophisticated timing attack. Notably, the definitive patch against this attack required roughly 500 lines of new code in the OpenSSL implementation, illustrating the difficulty of making the DDM operations constant time. Moreover, several follow-up papers [9, 12, 13] have shown that variants of the attack are still mountable in certain circumstances or for certain implementations.

Following the release of TLS 1.2 [50] in 2008, we see more of a "patch" process being adopted by the TLS WG. During this time, we see an explosion of attacks against TLS. We discuss some of these attacks next.

In 2009 Ray, Dispensa and Rex more or less simultaneously discovered the TLS Renegotiation attack against the TLS handshake. By exploiting the lack of a cryptographic binding between an attacker's initial handshake and a subsequent renegotiation handshake between an honest client and an honest server, the attacker is able to convince the server to interpret traffic – both the attacker-injected traffic and the honest client's traffic – as coming from the honest client. The WG's response to this attack was the announcement of a mandatory TLS extension [132] applicable to all versions of TLS. The extension proposed including the respective `Finished` messages in the client and server renegotiation `Hello` messages, thus creating a binding between the two handshakes. Unfortunately, the Triple Handshake attack of Bhargavan *et al.* [29] resurrected the Renegotiation attack by cleverly

[5] Approximately 3.5% of the roughly 150k servers surveyed by SSL Pulse still support SSLv2. Support statistic for February 2018. Retrieved from `https://www.trustworthyinternet.org/ssl-pulse/`.

exploiting the interaction of various TLS resumption and renegotiation handshakes. The attack completely breaks client authentication in TLS 1.2 and below.

In 2011 Duong and Rizzo announced the BEAST[6] attack [57]. The attack affects the TLS 1.0 Record Protocol and makes use of the chained-IV vulnerability in CBC mode observed by Moller [109] and Bard [19], though it has its roots in an observation of Rogaway [136] from as early as 1995. BEAST exploits the fact that in TLS 1.0, the final ciphertext block of a CBC-encrypted record becomes the IV for the next record to be encrypted. This enables an attacker with a chosen plaintext capability to recover low entropy plaintexts. The main significance of the BEAST attack is the clever use of malicious JavaScript running in a victim's browser to realise the low entropy, chosen plaintext requirement and thereby mount an HTTP session cookie recovery attack against TLS. However, it should be noted that the attack required a zero-day vulnerability[7] in the browser in order to obtain the required fine control over chosen plaintexts. The malicious JavaScript techniques were leveraged a year later by the same authors in the CRIME[8] attack (see [139] for a useful description of the attack). Unlike BEAST, however, CRIME exploits the compression side-channel inherent to all versions of TLS, a vulnerability noted in theoretical form by Kelsey in 2002 [82]. Interestingly, whilst the BEAST and CRIME attacks can be seen as having triggered the flood of research that followed, neither came from the academic research community, but instead from the "hacker" community (which partly explains the lack of formal research papers describing the attacks). Both attacks required a strong understanding not only of the cryptographic aspects of the protocol, but also of how the protocol is deployed in the web context.

The widespread response to CRIME was to disable TLS's compression feature. However, this does not completely solve the problem of compression-based attacks because compression can also take place at the application layer and introduce similar side-channels (see the BREACH and TIME attacks). A common response to BEAST was to switch to using RC4 as the encryption method in the Record Protocol, since a stream cipher would not be susceptible to the CBC vulnerabilities. Unfortunately, the RC4 keystream has long been known to be biased [103], and these keystream biases were exploited in a number of plaintext recovery attacks on TLS [10, 40, 68, 113, 148], leading to the IETF deprecating RC4 in March 2015 in [122]. We say more about these attacks in Part II of this thesis.

[6] Browser Exploit Against SSL/TLS
[7] Details surrounding which zero-day was used were never specified.
[8] Compression-Ratio Info-leak Made Easy

Other notable attacks to follow the BEAST, CRIME and RC4 attacks include the FREAK [26] and Logjam [8] attacks of 2015, and the SLOTH attack [34] of 2016. Both FREAK and Logjam exploit the enduring widespread support for weak export-grade cryptographic primitives. Whereas the FREAK attack affects certain TLS implementations, the Logjam vulnerability, in contrast, is the result of a protocol flaw and targets Diffie-Hellman key exchange in TLS. The attack requires a server to support export-grade cryptography, and for the client to be willing to use low security Diffie-Hellman groups. An active attacker can convince the server to provide an export-grade 512-bit group to a client that has requested a non-export DHE cipher suite, and the client will in turn accept this weak group as being valid for the DHE handshake. Clever use of a pre-computation phase for state-of-the-art discrete logarithm algorithms in [8] allowed for the quick computation of individual connections' secrets. An early intimation of these types of cross-cipher-suite attacks can be found in the work of Wagner and Schneier [151] as early as 1996. The warning from this paper seems to have been either forgotten or ignored in subsequent developments of TLS. Moreover, from version 1.1 onwards, export-grade cipher suites were not supported by the TLS standards. However, as already noted, almost all servers do support TLS 1.0 and so become vulnerable to this class of attack.

The change in TLS 1.2 from supporting the MD5/SHA-1 hash function combination to supporting single hash functions for digital signatures meant that stronger hash functions, such as SHA-256, could be used but alas, so could weaker hash functions, such as MD5. Wang and Yu [152] described collision attacks against MD5 in 2005; the SLOTH attack [34] exploits this weakness to break client authentication in TLS 1.2 when MD5-based signatures are employed. The attacks presented are near-practical and falsify the belief of some practitioners that only second-preimage resistance is required of the hash functions used for TLS signatures.

We have described, at a high-level, a number of the most prominent attacks on TLS and the TLS WG's responses to these attacks. We now turn to examining whether or not these attacks were adequately addressed, and indeed, to what extent they could have been addressed by the standardisation process.

3.1.2 Fixes, Constraints and Time Lags

The TLS 1.2 specification provides the following cautionary note with regards to the Bleichenbacher attack:

> "a TLS server MUST NOT generate an alert if processing an RSA-encrypted premaster secret message fails, or the version number is not as expected. Instead, it MUST continue the handshake with a randomly generated premaster secret. It may be useful to log the real cause of failure for troubleshooting purposes; however, care must be taken to avoid leaking the information to an attacker (through, e.g., timing, log files, or other channels.)"

Upon first glance, the countermeasure appears adequate. However, as pointed out by Jager *et al.* [78], the discovery of new side-channels and the development of more sophisticated analysis techniques allow for the implementation of Bleichenbacher-style attacks even though the vulnerability was thought to be successfully patched. The attacks by Meyer *et al.* [108] on implementations of TLS serve as an example of this. One course of action open to the TLS WG was to remove the use of the PKCS#1 v1.5 encoding scheme in favour of the PKCS#1 v2.1 encoding scheme (implementing OAEP padding) [79]. This would have been more secure against the Bleichenbacher attack and all envisionable variants. However, as is explained in the TLS 1.1 and TLS 1.2 RFCs, in order to maintain compatibility with earlier TLS versions, this replacement was not made. We presume that the desire to maintain backwards compatibility and confidence in the *ad hoc* countermeasure trumped the evidently better security available from the use of PKCS#1 v2.1.[9]

A very similar situation pertains to padding oracle attacks and Lucky 13: an implementation patch was put in place in TLS 1.1 and 1.2, but shown to be inadequate by the Lucky 13 attack [11]. With hindsight, it would have been less effort overall, and less damaging to the reputation of the protocol, to reform the MEE construction used in TLS at an earlier stage, replacing it with a modern design fully supported by theoretical analysis (notwithstanding the positive results of [87], whose limitations were pointed out in [117]). A repeated pattern in the development of TLS 1.2 and below is that the TLS community (a larger group of

[9] We note that Manager published an attack on PKCS#1 v2.1 in 2001 [100], however, his work included improvements to address the weaknesses uncovered, and by the release of TLS 1.1 in 2006, precautions against this attack had been incorporated into the PKCS#1 v2.1 standard.

individuals and organisations than the TLS WG) seem to need to see concrete working attacks before addressing a potential vulnerability or adopting an intrinsically more secure solution, rather than applying a patch to each specific vulnerability.

In the case of attacks that exploit the existence of primitives or mechanisms that have long been known to exhibit weaknesses, the simple (but naive) solution is to simply consider removing a primitive or mechanism as soon as it is shown to be weak. However, this might not be straightforward given implementation and interoperability constraints. In the case of FREAK and Logjam, the standardisation process cannot be faulted: the weak export cipher suites were removed from TLS 1.1 and TLS 1.2 and these attacks exist as a result of poor implementation choices by practitioners. Similar remarks apply to the IV-chaining vulnerability, which while already known in 1995, was introduced to TLS 1.0 in 1999, but then removed in TLS 1.1 in 2006. Unfortunately, deployed versions of TLS did not move so quickly, with widespread support for TLS 1.0 in servers even today. On the other hand, all modern browsers will now prefer TLS 1.2 and Authenticated Encryption with Associated Data (AEAD) cipher suites in an initial handshake attempt, thanks to the long line of attacks on TLS's CBC mode and RC4 options. In the case of SLOTH, however, the issue might not be as clear-cut. MD5-based signature schemes should not have been re-introduced in the TLS 1.2 RFC. And RC4 has a very long track-record of weaknesses stretching back more than 15 years, meaning that its phasing out from TLS could arguably have been initiated much sooner than it was, instead of waiting for the attacks to become a threat. In many cases, particularly where hardware support for AES is available, AES-GCM could have served as a better choice for encryption.

With the many research papers professing the security of the TLS Handshake Protocol [31, 71, 77, 90], the existence of attacks exploiting the interaction of various TLS handshakes may have come as a surprise to the TLS community. However, even here, there were early signs that things were amiss with the 1996 cross-cipher-suite attack of Wagner and Schneier [151]. Perhaps the lack of a practical attack in that paper and in later papers such as [105] led to a more relaxed attitude being adopted by the TLS WG here. The subtle interaction of different TLS handshakes was never fully considered in any analysis of TLS prior to the Triple Handshake attack of 2014 [29]. It is therefore not surprising that attacks of this form would have slipped through the standardisation process. Yet it should be remembered that the Triple Handshake attack is a resurrection of the Renegotiation attack from 2009. This is indicative of insufficiently broad or powerful analysis tools having

been available to the TLS WG in the period intervening between the two attacks.

We argue that, in general and in view of the extreme importance of TLS, a much more conservative approach to dealing with attacks on TLS is warranted. We do, however, appreciate that bringing about meaningful change is challenging given (i) the large scale and wide diversity of TLS deployment, (ii) the historical reticence of the major implementations to code newer versions of the protocol (especially TLS 1.2), and (iii) the slowness with which users (particularly on the server side) have tended to update their TLS versions.

3.1.3 Impact and Incentives

In the design-release-break-patch standardisation cycle, maximal reward for researchers has come in the form of producing and promoting high impact attacks against TLS, and engagement of the research community was largely encouraged in a retroactive fashion. The obvious problem with this incentive model is that it leaves users of published standards vulnerable to attack and imposes a potential patch action on the TLS WG. In the next section we show that a shift in the standardisation cycle leaves the opportunity for researchers to have impact (of a different kind) whilst positively benefiting the standardisation process.

3.2 Pre-Deployment Analysis

In contrast to the development of TLS 1.2 and below, the standardisation process for TLS 1.3 has been proactive in nature. It has followed what we describe as the *design-break-fix-release* cycle for standards development. Working more closely with the research community, the TLS WG has released multiple protocol drafts and welcomed analyses of the protocol before its final release. This design philosophy has simultaneously led to the discovery of weaknesses and provided confidence in the WG's design decisions. We explore the factors that have enabled this newer process by considering the improvements in the protocol analysis tools available, as well as the shift in design attitudes and incentives.

3.2.1 Design, Break, Fix, Release

As stated in the previous chapter, the two broad design goals for TLS 1.3 have been (i) to improve efficiency of the Handshake Protocol and (ii) to address the weaknesses identified in TLS 1.2 and below. The initial challenge for the TLS WG was to go about achieving these goals without having to invent an entirely new protocol: in addition to requiring new code libraries, a new protocol might introduce new weaknesses. The development of Google's QUIC Crypto by Langley and Chang [93] in 2013, offering a zero round-trip time (0-RTT) capability for the QUIC protocol [137], put pressure on the TLS WG to consider ways of reducing handshake latency in TLS 1.3. And, after the flurry of attacks in the preceding years, the protocol was due an overhaul to remove weak or broken features.

In comparison to TLS 1.2 and below, the first few drafts of TLS 1.3 (beginning with draft 00 in April 2014) incorporated changes that aimed to fortify the protocol against known attacks, such as the removal of support for compression, as well as the removal of static RSA and Diffie-Hellman key exchange mechanisms, leaving ephemeral Diffie-Hellman as the only method of key exchange. Troublesome record layer cipher suites, such as those allowing for the use of CBC mode and RC4 were also removed, leaving AEAD algorithms as the only supported algorithms for record layer protection. Handshake latency was also reduced by the introduction of a one round-trip time (1-RTT) TLS handshake (previously an initial handshake required two round trips before a client and a server could start exchanging application data).

Two important changes that were introduced in the drafts up to and including `draft-05` are the concept of a *session hash* and the removal of the renegotiation handshake. At the time of release of `draft-04`, the session hash constituted a hash value of all messages in a handshake starting with the `ClientHello`, up to and including the `ClientKeyExchange`. The session hash is included in the key derivation process to prevent an active attacker from synchronising the `master secret` across two different sessions, a trick employed in the Triple Handshake attack [29]. The removal of renegotiation prevents renegotiation-based attacks, the Triple Handshake attack again serving as an example of this class of attack.

In terms of analysis of TLS 1.3, Dowling *et al.* [54] and Kohlweiss *et al.* [85] published works on `draft-05`, the latter set of authors using a constructive-cryptography approach to provide security guarantees for the protocol. Their work highlights that the design

choice in TLS 1.3 to separate out the Handshake and Record protocols helps with their analysis, and indeed with provable security approaches in general. (Recall that in TLS 1.2 and below, the application traffic keys derived in the Handshake Protocol were used to encrypt the `Finished` messages of the Handshake Protocol itself. This interaction adds significant complexity to analyses of TLS 1.2 and below, in particular because it violates the standard indistinguishability security goal for a key exchange protocol.)

Dowling *et al.* [54] used the multi-stage key exchange model of Fischlin and Günther [60] to show that the keys output by the Handshake Protocol could be securely used in the Record Protocol. Their work included several positive comments regarding the soundness of the TLS 1.3 design, thereby explicitly providing useful feedback to the TLS WG.

In `draft-07` we see the most radical shift away from TLS 1.2, with the cryptographic core of the TLS handshake becoming strongly influenced by the OPTLS protocol of Krawczyk and Wee [91], with many OPTLS elements being incorporated into the draft. OPTLS has been expressly designed to be simple and modular, offering a 1-RTT, forward secure TLS handshake that employs ephemeral Diffie-Hellman key exchange. OPTLS also offers 0-RTT support as well as a pre-shared key (PSK) mode, capturing the use case in which a client and a server enter into the protocol having previously shared a key. This particular mode is of relevance from `draft-07` onwards as the TLS 1.2-style resumption mechanism is replaced with a mechanism that makes use of PSKs. This draft included a 0-RTT handshake and key derivation schedule that is similar to that of OPTLS, employing the HKDF primitive designed by Krawczyk [88]. The OPTLS designers provided a detailed analysis of their protocol in [91], again providing the TLS WG with confidence in its design choices.

However, it should be noted that significant changes were made in adapting OPTLS to meet the needs of TLS. For example, OPTLS originally assumed that servers' long term keys would be Diffie-Hellman values, in turn supported by certificates. However, such certificates are not widely used in practice today, potentially hindering deployment of TLS 1.3. Thus, in the "translation" of OPTLS into TLS 1.3, a two-level process was assumed, with the server using a traditional signing key to authenticate its long-term Diffie-Hellman value. But this created yet another real-world security issue: if an attacker can gain access to a server's signing capability just once, then he would be able to forge a credential enabling him to impersonate a server on a long-term basis. Thus it was decided to change

the signature scope to also include client-supplied, session-specific information, limiting the value of any temporary access to the signing capability. This reduces the efficiency of the protocol, since now a fresh signature must be produced by the server in each handshake.

Notable changes in `draft-08` and `draft-09` of the protocol include the removal of support for MD5-based signatures as well as the deprecation of SHA-1-based signatures, partly in response to the SLOTH vulnerability [34] and as a result of pressure from practitioners and researchers to remove these weak primitives, as evidenced on the TLS mailing list [69, 70].

Cremers *et al.* [46] performed an automated analysis of TLS 1.3 using the TAMARIN prover [6]. Their model covers `draft-10` and their analysis showed that this draft meets the goals of authenticated key exchange. They used a symbolic model to analyse the interaction of the various handshake components of `draft-10`. Anticipating the inclusion of the post-handshake client authentication mechanism in the TLS 1.3 series of drafts, they discovered a potential interaction attack which would break client authentication. Their attack was communicated to the TLS WG, and `draft-11`, which officially incorporated the post-handshake client authentication mechanism, included the necessary fix as part of the design. This work will be described in detail in Chapter 6 of this thesis.

In concurrent work to [46], Li *et al.* [95] analysed the interaction of the various TLS 1.3 handshake modes in the computational setting using their *multi-level&stage* security model. They found `draft-10` to be secure in this model. The post-handshake authentication threat was not identified in this work presumably because this mechanism was not officially part of `draft-10`.

In February of 2016, just prior to the release of `draft-12`, the Internet Society hosted a "TLS Ready or Not?" (TRON) workshop. The workshop showcased analyses of TLS 1.3, both published and under development, bringing together members of the TLS Working Group, researchers and industry professionals with the aim of testing the readiness of TLS 1.3 in its then current form. Besides the aforementioned works by Kohlwiess *et al.* [85], Krawczyk and Wee [91], and Cremers *et al.* [46], there were several other presentations highlighting progress in the protocol's development, as well as the challenges still facing the TLS WG. Dowling *et al.* updated their previous analysis to cover `draft-10` [55], showing the full (EC)DHE handshake to be secure in the multi-stage key exchange setting. Bhargavan *et al.* introduced ProScript [32], a JavaScript variant of their verified TLS

implementation, miTLS [3, 30]. Interestingly, ProScript also allows for the extraction of a symbolic model for use within the ProVerif protocol analysis tool [4, 35]. This work highlighted the potential dangers of incorporating certificate-based authentication into PSK handshakes, a potential protocol extension being considered by the TLS WG. Work on the secure of implementation of TLS 1.3 by Berdouche *et al.* [25] considered how to maintain compatibility with current TLS versions whilst protecting against downgrade attacks, and highlighted simplifications to the protocol which could be beneficial from an implementation point of of view.

Importantly, the TRON workshop led to discussions between the WG and the research community regarding potential simplifications and enhancements to the protocol, informing subsequent drafts of the protocol.

At around the same time as the TRON workshop, an analysis by the Cryptographic protocol Evaluation towards Long-Lived Outstanding Security (CELLOS) Consortium, using the ProVerif tool, was announced on the TLS WG mailing list [14, 104]. This work showed the initial (EC)DHE handshake of `draft-11` to be secure in the symbolic setting.

Further publications of relevance to TLS 1.3 include the work on downgrade resilience by Bhargavan *et al.* [28] and the work on key confirmation by Fischlin *et al.* [61]. The first provides suggestions on how to strengthen downgrade security in TLS 1.3 and the second provides assurances regarding the key confirmation mechanisms used.

A smaller *ad hoc* meeting informally called "TRON2" took place in May 2016. At this meeting, the latest changes to the protocol were discussed, further formal analysis was presented, and TLS 1.3 implementations were compared.[10]

Since the TRON2 meeting, more work on TLS 1.3 has been produced, including work by Bhargavan *et al.* [27] on `draft-18` that proposes a methodology for developing verified symbolic and computational models of TLS 1.3 together with a high-assurance reference implementation of the protocol. This work includes a symbolic ProVerif model, as well as a computational CryptoVerif[11] model, of `draft-18`, and an interoperable implementation of TLS 1.0 through 1.3 known as RefTLS which allows for the extraction

[10] See `https://www.mitls.org/tron2/` for details.

[11] An automated tool for the construction of models and the production of the corresponding proofs in the computational setting. Available at [1].

of symbolic ProVerif models from its statically typed JavaScript code. The authors' symbolic and computational analyses of `draft-18` of TLS 1.3 find that the protocol meets its desired security properties, however, they explain a scenario in which an attacker is able to mount an *authenticated replay* attack, i.e., a replay of an authenticated client's data, in the 0-RTT handshake case. The TLS 1.3 draft does not claim to prevent replays of 0-RTT data, however, in response to the scenario presented in [27], it now includes warnings regarding this type of attack. The work also confirmed the downgrade resistance inherent in TLS 1.3, with respect to previous versions of TLS, and presented an important tool for the development of TLS 1.3 implementations – a reference implementation with support for symbolic verification.

Further progress on TLS 1.3 implementations includes the work by Delignat-Lavaud *et al.* [47] which builds and verifies the first reference implementation of the TLS 1.3 Record Protocol (as described in `draft-19`). Plugging their implementation into the miTLS library [3], the authors show interoperability with the Chome and Firefox browsers.

Using their previous `draft-10` analysis as a foundation, Cremers *et al.* developed a symbolic model of `draft-21` of the TLS 1.3 protocol using the Tamarin prover [45]. This work shows that many of the TLS 1.3 handshake modes meet the desired properties, however, the work also uncovers an unexpected authentication behaviour in the post-handshake authentication setting. We discuss this work in more detail in Chapter 7 of this thesis.

Much of the work pertaining to `draft-18` and beyond was presented at the "TLS 1.3: Design, Implementation & Verification Workshop" (TLS:DIV) that was co-located with the IEEE European Symposium on Security and Privacy and the Eurocrypt conferences of 2017. This workshop again showcased the bringing together of the TLS WG and the academic community, highlighting the collaborative nature of the TLS 1.3 design process with an emphasis on finding and remedying flaws prior to the protocol's official release.

3.2.2 Available Tools

Since the release of TLS 1.2 in 2008, cryptographic protocol analysis tools have developed and matured to the extent that they can now effectively serve a proactive standardisation process, thereby contributing to, and perhaps even enabling, a more collaborative design

effort for TLS 1.3. Significant advances have been made across all fronts, from lower-level primitives such as key derivation and authenticated encryption, to higher level primitives such as authenticated key exchange and cryptographic modelling of secure channels.

An early analysis of the TLS protocol itself can be found in the work of Gajek *et al.* [67] in 2008. However, their analysis only covers unauthenticated key exchange. Many refinements and advances in the area of provable security for TLS have since been made. A major on-going challenge has been to provide accurate modelling of the protocol and to capture the complexity of its many interacting components and modes. In 2010, Morrissey *et al.* [111] also analysed the TLS Handshake Protocol. However, their work only considered a truncated version of the protocol (with no encryption of `Finished` messages), assumed that a CCA-secure encryption scheme was used for key transport (which is unrealistic given that TLS implementations employ PKCS#1 v1.5-based RSA encryption), and relied on the random oracle model. In 2012, Jager *et al.* [77] introduced the Authenticated and Confidential Channel Establishment (ACCE) security model in an attempt to handle the unfortunate mixing of key usage in the Handshake and Record protocols; they used the ACCE model to analyse certain Diffie-Hellman-based key exchanges in TLS. Their work built in part on a 2011 work of Paterson *et al.* [117], who introduced the notion of length-hiding Authenticated Encryption, which models desired security goals of the TLS Record Protocol.

Further important works include those by Krawczyk *et al.* [90] and Kohlar *et al.* [84]. The former work analysed multiple, different TLS key exchange methods using a single, uniform set of proof techniques in the ACCE setting, while the latter extended the work of Jager *et al.* to show that the RSA and Diffie-Hellman handshakes can be proven secure in the mutual authentication setting. Li *et al.* [96] performed a similar task for pre-shared key cipher suites. Giesen *et al.* [71] explicitly consider multiple Handshake Protocol runs and their interactions in their formal treatment of the security of TLS renegotiation, while Dowling and Stebila [56] examined cipher suite and version negotiation in TLS. All of these works offer techniques that have been harnessed, and extended, in the analysis of TLS 1.3, prior to its final release. Moreover, they represent a growth in interest in the TLS protocol from the research community, a necessary precursor to their greater involvement in the TLS 1.3 design process.

A major step forward in the domain of program verification for TLS came with the first

release of the miTLS reference implementation in 2013 [3, 30]. The miTLS implementation integrates software security and computational cryptographic security approaches so as to obtain security proofs for TLS source code. This approach aims to eliminate the reliance on the simplifying assumptions employed by the more traditional provable security techniques – those tend to analyse abstract and somewhat high-level models of TLS and tend to ignore many implementation details in order to obtain tractable models (in the form of pseudo-code) suitable for the production of hand-generated proofs; moreover, they tend to focus on "fragments" of the TLS protocol suite rather than the entire system. Using their approach, Bhargavan *et al.* provided a security analysis of the TLS 1.2 handshake as implemented in miTLS [31]. The miTLS implementation provides a reference for the secure implementation of TLS 1.2 and below, and interoperates with all major web browsers and servers. Not only has the miTLS project lead to the discovery of vulnerabilities such as the Triple Handshake attack and FREAK, but it has also left the TLS community with tools such as FlexTLS [2] which allows for the rapid prototyping and testing of TLS implementations. These tools have also been harnessed to assess TLS 1.3, as discussed in the previous section.

The rise of automated protocol analysis tools such as ProVerif [4] and the Tamarin prover [6] can also be counted as a boon for the TLS WG. The more recent Tamarin tool, for instance, offers good support for Diffie-Hellman-based protocols and allows for the instantiation of an unbounded number of protocol participants and sessions, making it a good choice for the modelling and consequent symbolic analysis of TLS 1.3. Once established, this type of model can also be easily adapted in response to protocol changes, making this tool invaluable in an ongoing development process.

The advances in the areas of provable security, program verification and formal methods have contributed to a development environment in which a design-break-fix-release standardisation cycle can thrive. Previously, the absence of these techniques, or the limited experience in applying them to real-world protocols like TLS, would have limited the amount of pre-release analysis that could have been performed, making a design-release-break-patch standardisation cycle understandable, natural even, for TLS 1.2 and below.

3.2.3 Impact and Incentives

In the development of TLS 1.3, the WG has taken many positive steps in aiming to protect the protocol against the various classes of attacks mentioned in Section 3.1. Removal of support for weak hash functions, renegotiation, and non-AEAD encryption modes, as well as the introduction of the session hash mechanism serve as illustrative examples. The WG has also made design choices that have eased the analysis of the protocol, such as making a clean separation of the Handshake and Record Protocols, for instance. This is undoubtedly a positive step by the WG to respond to the research community's needs, marking a shift in the WG's design mindset. The TRON workshop also displays a desire by the IETF to involve the research community in the design of TLS 1.3, and to incorporate its contributions. The research community, on the other hand, has gained a much greater awareness of the complexities of the TLS protocol and its many use cases, and has tried to adapt its analyses accordingly. In view of the rising interest in, and focus on, TLS within the research community over a period of years, and the attendant refinement of its analysis tools, this community has been in a much better position to contribute to the TLS 1.3 design process than it was for former editions of the protocol.

The ability to adapt the protocol in response to potential attacks, such as those identified by Cremers *et al.* [46] and Bhargavan *et al.* [32], makes for a stronger protocol and has allowed the WG to implement changes pre-emptively, hopefully reducing the need to create patches post-release. In comparison to the previous process described in Section 3.1, the design-break-fix-release standardisation cycle appears to leave the incentives for researchers unchanged, with a number of top-tier papers being produced prior to the protocol's finalisation. However, it is notable that these papers provide largely positive security results about TLS 1.3 rather than new attacks. We consider this to be as a result of the research community's stronger appreciation of the importance of TLS and its greater awareness of the value in contributing to its standardisation, in comparison to former development cycles.

Part II

Attacking TLS 1.2 and Below

Chapter 4

Password Recovery Attacks Against RC4

Contents

6.1 Introduction 135
6.2 Preliminaries 139
6.3 `draft-10` Analysis 154
6.4 Conclusion 173

In this chapter we introduce password recovery attacks against RC4 in TLS. We describe the RC4 algorithm, discuss the relevant single- and double-byte keystream biases, and provide further background on password distributions. We present a formal Bayesian analysis that combines an a priori *password distribution with keystream distribution statistics to produce* a posteriori *password likelihoods, yielding a procedure which is truly optimal (if the password distribution is known exactly). We demonstrate the effectiveness of our attacks via extensive simulations and present a proof-of-concept implementation against a widely-used application that makes use of passwords over TLS, namely BasicAuth.*

4.1 Introduction

In 2013 the RC4 stream cipher suffered several blows from the academic community, as evidenced in the works [10], [76] and [113]. The work in [10] by Al Fardan *et al.* specifically targets RC4 when used in TLS, presenting two attacks aimed at recovering TLS-protected HTTP session cookies. The first attack leverages the existence of single-byte biases in the

early positions of the RC4 keystream, i.e., biases of the form

$$\Pr(Z_r = z) = 2^{-8} \cdot (1 + \varepsilon),$$

where Z_r denotes the r-th output byte of the RC4 keystream, z is a byte value, and $\varepsilon \neq 0$. The attack exploits these biases to mount a ciphertext only attack against TLS. The attack requires multiple, independent encryptions of the target plaintext, otherwise known as a *broadcast attack*. The second attack exploits periodically occurring double-byte biases, i.e., biases of the form

$$\Pr((Z_r, Z_{r+1}) = (z_1, z_2)) = 2^{-16} \cdot (1 + \varepsilon),$$

where Z_r and Z_{r+1} denote the r-th and $(r + 1)$-th output bytes of the RC4 keystream, z_1 and z_2 are byte values, and $\varepsilon \neq 0$. This attack also requires repeated encryptions of the target plaintext.

Despite the existence of these attacks, SSL Pulse[1] showed that, in February 2015, 74.5% of the roughly 150,000 sites surveyed still allowed negotiation of RC4. Even worse, a January 2015 survey[2] of about 400,000 of the Alexa top 1 million sites showed that 3712 of them, or 0.79%, supported *only* RC4 cipher suites, and 8.75% forced the use of RC4 in TLS 1.1 and 1.2, where better ciphers (such as AES-GCM) were available. Adding to the affront, March 2015 data from the ICSI Certificate Notary project[3] showed that more than 30% of SSL/TLS connections were still using RC4.

At the time, a major reason for RC4 remaining so popular was that while the attacks of [10] broke RC4 in TLS in an academic sense, the attacks were far from being practical. For example, the preferred cookie-recovery attack in [10] needs around 2^{33} – 2^{34} encryptions of a 16-byte, base64-encoded secure cookie to reliably recover it. The number is so high because, with mainstream browsers and taking into account the verbosity of the HTTP protocol, the target cookie is not located near the start of the RC4 keystream, meaning that the strong, single-byte keystream biases in RC4 observed in [10] cannot be exploited. Rather, the preferred attack from [10] uses the much weaker, long-term Fluhrer-McGrew double-byte biases from [64]. This substantially increases the number of required encryptions before the plaintext cookie can be reliably recovered, to the point where, even with highly-tuned

[1] Available at `https://www.trustworthyinternet.org/ssl-pulse/`.

[2] Available at `https://securitypitfalls.wordpress.com/2015/02/01/january-2015-scan-results/`.

[3] The International Computer Science Institute (ICSI) Certificate Notary project monitors live network traffic, observing public key certificates in real time and recording the SSL/TLS cipher suites negotiated as part of SSL/TLS handshakes.

malicious JavaScript running in the victim's browser generating 6 million cookie-bearing HTTP POST requests per hour, the wall-clock time to execute the attack is on the order of 2000 hours using the experimental setup reported in [10]; moreover the attack generates many terabytes of network traffic. Thus the practical threat posed by the RC4 attacks reported in [10] was arguably quite limited.

In this chapter we present attacks recovering TLS-protected passwords whose ciphertext requirements are significantly reduced compared to those of [10]: we achieve a reduction from 2^{34} ciphertexts down to 2^{26} – 2^{28} ciphertexts. We also describe a proof-of-concept implementation of these attacks against a specific application-layer protocol making use of passwords, namely BasicAuth.

We revisit the statistical methods of [10], refining, extending and applying them to the specific problem of recovering TLS-protected passwords. Our target is to reduce as much as possible the ciphertext requirements of the original RC4 attacks from [10]. Our overall objective when conducting this work was to bring the use of RC4 in TLS closer to the point where it became indefensible and had to be abandoned by practitioners. Passwords served as a good target for our attacks because they are still very widely used on the Internet for providing user authentication, and are frequently protected using TLS to prevent them being passively eavesdropped. It is true that major websites use secure cookies for managing user authentication but the authentication is usually bootstrapped via password entry. However, to build effective attacks, we needed to find and exploit systems in which users' passwords are automatically and repeatedly sent under the protection of TLS, so that sufficiently many ciphertexts could be gathered for our statistical analyses.

Our contributions in this chapter are as follows:

(i) We present a formal Bayesian analysis that combines an *a priori* plaintext distribution with keystream distribution statistics to produce *a posteriori* plaintext likelihoods. This analysis formalises and extends the procedure followed in [10] for single-byte attacks. There, only keystream distribution statistics were used (specifically, biases in the individual bytes in the early portion of the RC4 keystream) and plaintexts were assumed to be uniformly distributed, while here we also exploit (partial) knowledge of the plaintext distribution to produce a more accurate estimate of the *a posteriori* likelihoods. This yields a procedure that is optimal (in the sense of yielding a

maximum *a posteriori* estimate for the plaintext) if the plaintext distribution is known exactly. In the context of password recovery, an *estimate* for the *a priori* plaintext distribution can be empirically formed by using data from password breaches or by synthetically constructing password dictionaries. We will demonstrate, via simulations, that this Bayesian approach improves performance (measured in terms of success rate of plaintext recovery for a given number of ciphertexts) compared to the approach in [10]. We develop two attack algorithms, the first making use of a single-byte-based approximation for a vector of consecutive keystream bytes (necessary for recovering a vector of consecutive plaintext bytes such as a password), and the second making use of a double-byte based approximation for a vector of consecutive keystream bytes. The dominant terms in the running time for both of the resulting algorithms is $\mathcal{O}(nN)$ where n is the length of the target password and N is the size of the dictionary used in the attack.

A major advantage of our new algorithms over the work in [10] is that they output a value for the likelihood of each password candidate, enabling these to be ranked and then tried in order against a user's account. This fits neatly with how password authentication often works in practice: users are given a pre-determined number of tries before their account locks out.

(ii) We evaluate and compare our password recovery algorithms through extensive simulations, exploring the relationships between the main parameters of our attack in Table 4.1. Naturally, given the combinatorial explosion of possible parameter settings (and the cost of performing simulations), we focus on comparing the performance with all but one or two parameters or variables being fixed in each instance.

(iii) Our final contribution is to identify and apply our attacks to a specific and widely-deployed application making use of passwords over TLS: BasicAuth. We introduce the BasicAuth application and describe a proof-of-concept implementation of our attacks against it, giving an indication of the practicality of our attacks.

Our attacks exhibit significant success rates with only $S = 2^{26}$ ciphertexts, in contrast to the 2^{34} ciphertexts required in [10]. This is because we are able to force the target passwords into the first 256 bytes of plaintext, where the strong single-byte keystream biases come into play. For example, with $S = 2^{26}$ ciphertexts, we would expect to recover a length 6 BasicAuth password with a 44.5% success rate with $T = 5$ tries; the rate rises to 64.4% if $T = 100$ tries are made. In practice, many sites

Parameter	Description
n	The length in bytes of the target password.
S	The number of available encryptions of the password.
r	The starting position of the password in the plaintext stream.
N	The size of the password dictionary used in the attack.
T	The number of tries made (meaning that our algorithm is considered successful if it ranks the correct password amongst the top T passwords, i.e., the T passwords with highest likelihoods as computed by the algorithm).
`algorithm choice`	Which of our two algorithms is used (the one computing the keystream statistics using the product distribution or the one using a double-byte-based approximation).
`encoding`	Whether the passwords are base64 encoded before being transmitted, or are sent as raw ASCII/Unicode.

Table 4.1: Attack parameters

do not configure any limit on the number of BasicAuth attempts made by a client; moreover a study [39] showed that 84% of websites surveyed allowed for up to 100 password guesses (though these sites were not necessarily using BasicAuth as their authentication mechanism). As we will show, our result compares very favourably to the previous attacks and to random guessing of passwords without any reference to the ciphertexts.

However, there is a downside too: to make use of the early, single-byte biases in RC4 keystreams, we have to repeatedly cause TLS connections to be closed and new ones to be opened. Due to latency in the TLS Handshake Protocol, this leads to a significant slowdown in the wall clock running time of the attack; for $S = 2^{26}$, a fairly low latency of 100ms, coupled with exploiting browsers' propensity to open multiple parallel connections, we estimate a running time of around 300 hours for the attack. This is still more than 6 times faster than the 2000 hours estimated in [10].

Related Work. The RC4 stream cipher has long been subject to analysis by the academic community and has been shown to exhibit many weaknesess [63, 64, 72, 73, 99, 101, 103, 120, 140]. Many of these weaknesses concern biases in the RC4 keystream, which in turn have been exploited to recover plaintext. Within the context of TLS, besides the two attacks by AlFardan *et al.* [10] against RC4 in TLS (using single-byte biases in the first and double-byte Fluhrer-McGrew biases from [64] in the second), Isobe *et al.* [76] present plaintext recovery attacks for RC4 using single-byte and double-byte biases, although their

attacks are less effective than those of [10] and they do not explore in any great detail the applicability of the attacks to TLS. The work by Ohigashi *et al.* [113] also introduces plaintext recovery attacks against RC4 but instead makes use of the Mantin biases [101], i.e., biases of the form

$$\Pr\left((Z_r, Z_{r+1}) = (Z_{r+G+2}, Z_{r+G+3})\right) = 2^{-16}\left(1 + \frac{e^{(-4-8G)/256}}{256}\right),$$

where $G \geq 0$ is a small integer and Z_r denotes the r-th output byte of the RC4 keystream.

In concurrent work to ours, Vanhoef and Piessens [148] conducted an extensive search for new biases in RC4 keystreams, and settled on exploiting the Mantin biases in combination with the Fluhrer-McGrew biases to target the recovery of HTTP session cookies from TLS sessions. This work, along with the work of Ohigashi *et al.*, is relevant to the material presented in the next chapter and will be discussed in more detail therein (in Section 5.1).

In further concurrent work, Mantin presented the Bar Mitzvah attack against RC4 in TLS [102]. The attack exploits the Invariance Weakness identified by Fluhrer *et al.* in 2001 [63]. Exploiting this weakness allows for a partial plaintext recovery attack against RC4 in TLS, allowing for the recovery of the least significant bits (LSBs) of up to 100 bytes of the encrypted stream. Unfortunately, owing to the infrequent presence of the desired weakness in the RC4 keystream, an attacker needs to observe, on average, roughly 2^{30} TLS connections before the attack is successful. Also, the verbosity of the HTTP protocol ensures that no data of interest to an attacker is located within the first 100 bytes of the encrypted stream, reducing the practical threat of the attack.

We note that several other works present attacks against RC4 outside of the context of TLS, focusing on the stream cipher's usage in WPA [115, 116, 143, 144, 150], and the HIVE hidden volume encryption system [119]. We are currently not aware of any other attacks that exploit biases in the RC4 keystream to specifically target passwords when protected by RC4 in TLS.

4.2 Preliminaries

We now present concepts and definitions that are relevant to the understanding of the sections to follow. We introduce Bayes' Theorem and include a description of the RC4 algorithm, explaining its usage in TLS. We provide more detail on the single- and double-byte biases present in the RC4 keystream, and in particular, provide details concerning our search for double-byte biases in the early positions of the RC4 keystream. We also provide more detail concerning user-selected passwords.

4.2.1 Bayes' Theorem

Named after Reverend Thomas Bayes in tribute to his foundational work on the theory of probability [21], simply put, Bayes' Theorem states that

$$Pr(A|B) = \frac{Pr(B|A) \cdot Pr(A)}{Pr(B)},$$

where A and B are events, i.e., sets of outcomes of experiments to which probabilities are assigned, and $Pr(B) \neq 0$. $Pr(A|B)$ is the likelihood of A occurring given that B is true (a *conditional* probability). $Pr(B|A)$ is the likelihood of B occurring given that A is true. $Pr(A)$ and $Pr(B)$ represent the probabilities of A and B occurring independently of each other (known as *marginal* probabilities). We use this notion in our statistical attacks on stream cipher encrypted plaintexts in Section 4.3.1.

4.2.2 The RC4 Algorithm

Originally a proprietary stream cipher designed by Ron Rivest in 1987, RC4 is remarkably fast when implemented in software and has a very simple description. Details of the cipher were leaked in 1994 and the cipher has been subject to public analysis and study ever since.

RC4 allows for variable-length key sizes, anywhere from 40 to 256 bits, and consists of two algorithms, namely, a *key scheduling algorithm* (KSA) and a *pseudo-random generation algorithm* (PRGA). The KSA takes as input an l-byte key and produces the initial internal state $st_0 = (i, j, \mathcal{S})$ for the PRGA; $\mathcal{S}$ is the canonical representation of a permutation of

Algorithm 1: RC4 key scheduling (KSA)

input : key K of l bytes
output : initial internal state st_0
begin
  for $i = 0$ **to** 255 **do**
    $\mathcal{S}[i] \leftarrow i$
  $j \leftarrow 0$
  for $i = 0$ **to** 255 **do**
    $j \leftarrow j + \mathcal{S}[i] + K[i \bmod l]$
    swap$(\mathcal{S}[i], \mathcal{S}[j])$
  $i, j \leftarrow 0$
  $st_0 \leftarrow (i, j, \mathcal{S})$
  return st_0

Algorithm 2: RC4 keystream generator (PRGA)

input : internal state st_r
output : keystream byte Z_{r+1} updated internal state st_{r+1}
begin
  parse $(i, j, \mathcal{S}) \leftarrow st_r$
  $i \leftarrow i + 1$
  $j \leftarrow j + \mathcal{S}[i]$
  swap$(\mathcal{S}[i], \mathcal{S}[j])$
  $Z_{r+1} \leftarrow \mathcal{S}[\mathcal{S}[i] + \mathcal{S}[j]]$
  $st_{r+1} \leftarrow (i, j, \mathcal{S})$
  return (Z_{r+1}, st_{r+1})

the numbers from 0 to 255 where the permutation is a function of the l-byte key, and i and j are indices for $\mathcal{S}$. The KSA is specified in Algorithm 1 where K represents the l-byte key array and $\mathcal{S}$ the 256-byte state array. Given the internal state st_r, the PRGA will generate a keystream byte Z_{r+1} and updated state st_{r+1} as specified in Algorithm 2.

4.2.3 Single-byte Biases in the RC4 Keystream

RC4 has several cryptographic weaknesses, notably the existence of various biases in the RC4 keystream, see for example [10, 73, 101, 103]. Large single-byte biases are prominent in the early positions of the RC4 keystream. Mantin and Shamir [103] observed the first of these biases, in Z_2 (the second byte of the RC4 keystream), and showed how to exploit it in what they termed a broadcast attack, wherein the same plaintext is repeatedly encrypted under different keys. AlFardan *et al.* [10] performed large-scale computations to estimate these early biases, using 2^{45} keystreams to compute the single-byte keystream

distributions in the first 256 output positions. They also provided a statistical approach to recovering plaintext bytes in the broadcast attack scenario, and explored its exploitation in TLS. Much of the new bias behaviour they observed was subsequently explained in [140]. Unfortunately, from an attacker's perspective, the single-byte biases die away very quickly beyond position 256 in the RC4 keystream. This means that they can only be used in attacks to extract plaintext bytes which are found close to the start of plaintext streams. This was a significant complicating factor in the attacks of [10], where, because of the behaviour of HTTP in modern browsers, the target HTTP secure cookies are not so located.

4.2.4 Double-byte Biases in the RC4 Keystream

Fluhrer and McGrew [64] showed that there are biases in adjacent bytes in RC4 keystreams, and that these so-called double-byte biases are persistent throughout the keystream. The presence of these long-term biases (and the absence of any other similarly-sized double-byte biases) was confirmed computationally in [10]. AlFardan *et al.* [10] also exploited these biases in their double-byte attack to recover HTTP secure cookies.

Owing to the fact that we wish to exploit double-byte biases in early portions of the RC4 keystream and because the analysis of [64] assumes the RC4 permutation $\mathcal{S}$ is uniformly random (which is not the case for early keystream bytes), we carried out extensive computations to estimate the initial double-byte keystream distributions: we used roughly 4800 core-days of computation to generate 2^{44} RC4 keystreams for random 128-bit RC4 keys (as used in TLS); we used these keystreams to estimate the double-byte keystream distributions for RC4 in the first 512 positions.

While the gross behaviour that we observed is dominated by products of the known single-byte biases in the first 256 positions and by the Fluhrer-McGrew biases in the later positions, we did observe some new and interesting double-byte biases. In Figure 4.1, for instance, the influence of the single-byte key-length-dependent bias [72], and the single-byte r-bias [10] are evident. The former can be observed as the strong vertical line at $Z_{16} = \texttt{0xE0}$, while the latter can be seen as the lines at $Z_{16} = \texttt{0x10}$ and $Z_{17} = \texttt{0x11}$. The faint diagonal line appears to be a new double-byte bias (that is not accounted for as a product of single-byte biases). It appears in many early positions. For example, it is at least twice as strong as that arising in the product distribution for at least 64 of the 256

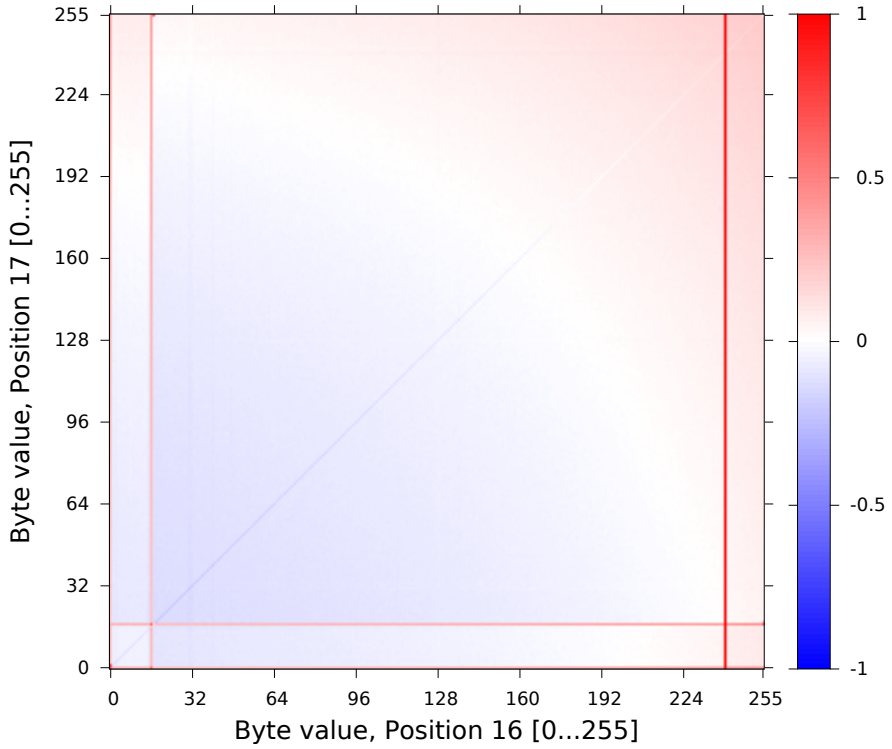


Figure 4.1: Measured biases for RC4 keystream byte pair (Z_{16}, Z_{17}). The colouring scheme encodes the strength of the bias, i.e., the deviation from the expected probability of $1/2^{16}$, scaled by a factor of 2^{22}, capped at a maximum of 1.

possible byte values from positions (Z_3, Z_4) up to positions (Z_{110}, Z_{111}). It then gradually disappears but reappears at around positions (Z_{192}, Z_{193}) (albeit as a positive bias) and persists up to positions (Z_{257}, Z_{258}) (changing sign again at (Z_{255}, Z_{256})).

The presence of horizontal and vertical lines in Figure 4.1 and the absence of other strong biases, which is typical for the early positions, indicates that the adjacent bytes behave largely independently of each other. In other words, there are very few strong conditional biases in the first 256 positions of the RC4 keystream. For later positions in the keystream, Figure 4.2 depicts what is typical in terms of bias behaviour: the presence of Fluhrer-McGrew biases only. These are visible in Figure 4.2 at (Z_{384}, Z_{385})= (`0x00`, `0x01`) and (`0x81`, `0xFF`) for example.

Finally, of particular interest is the distribution of (Z_1, Z_2). Figure 4.3a shows the raw distribution for this position pair, while Figure 4.3b shows the residual biases when the product distribution of Z_1 and Z_2 is removed. Note that the raw distribution is predominantly negatively biased; this is because of the effect of the large Mantin-Shamir positive bias towards `0x00` in position Z_2, and the compensating negative single byte biases

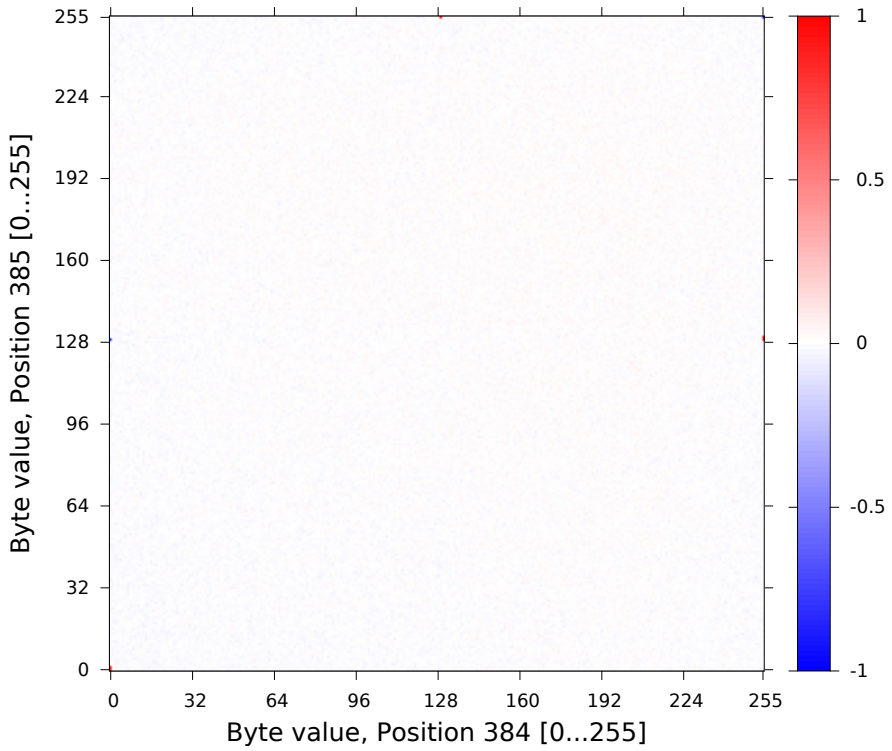


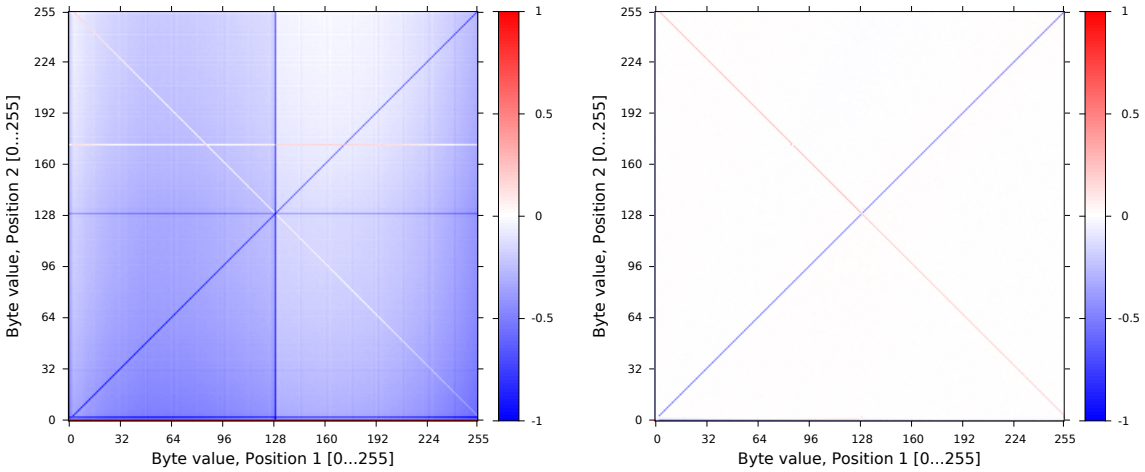
Figure 4.2: Measured biases for RC4 keystream byte pair (Z_{384}, Z_{385}). The colouring scheme encodes the strength of the bias, i.e., the deviation from the expected probability of $1/2^{16}$, scaled by a factor of 2^{24}, capped at a maximum of 1.

for all other values of Z_2. Note also the two diagonal lines in Figure 4.3b. The "positive" (blue-coloured) diagonal here represents a negative bias in (Z_1, Z_2) for all byte pairs (z, z) where $z \in \mathcal{B} \setminus \{\texttt{0x00}\}$ where $\mathcal{B}$ denotes the set of bytes $\{\texttt{0x0x00}, \ldots, \texttt{0x0xFF}\}$; this bias is also evident in the raw distribution in Figure 4.3a. The "negative diagonal" in Figure 4.3b shows that there is a systematic difference between the raw double-byte distribution and the product distribution. It manifests itself as a white-coloured negative diagonal in the raw double-byte distribution shown in Figure 4.3a; thus, in the raw distribution, it forms a structured set of unbiased pairs against a largely negatively-biased background.

The only other previously known bias of this nature in this portion of the keystream is due to Isobe *et al.* [76], who showed that:

$$\Pr(Z_1 = \texttt{0x00} \wedge Z_2 = \texttt{0x00}) = 2^{-16} \cdot (1 + 2^{0.996}).$$

This bias is also evident in Figure 4.3. By contrast, the new diagonal biases are negative,



(a) Colouring scheme encodes the strength of the bias, scaled by a factor of 2^{22}, capped at a maximum of 1.

(b) Colouring scheme encodes the strength of the bias after the product of single-byte biases for positions Z_1 and Z_2 is removed, scaled by a factor of 2^{22}, capped at a maximum of 1.

Figure 4.3: Measured biases for RC4 keystream byte pair (Z_1, Z_2).

sporting magnitudes in the region of 2^{-22}. For example, we empirically observe:

$$\Pr(Z_1 = \texttt{0x14} \wedge Z_2 = \texttt{0x14}) = 2^{-16} \cdot (1 - 2^{-6.097}).$$

We formally define a *large* double-byte bias to be one whose magnitude is at least 2^{-24}. We observed 103,031 such large biases in total. Note that with 2^{44} keystreams, all such biases are statistically significant and highly unlikely to arise from random fluctuations in our empirical analysis. For, in each position pair $(r, r+1)$ we have 2^{16} counters, one for each possible pair (Z_r, Z_{r+1}), so, in the absence of any biases, each counter would be (roughly) normally distributed with mean $2^{44} \cdot 2^{-16} = 2^{28}$ and standard deviation σ of approximately $\sqrt{2^{28}} = 2^{14}$. Then a bias of size 2^{-24} would lead to a counter value of around

$$2^{44} \cdot (2^{-16} + 2^{-24}) = 2^{28} + 2^6 \cdot 2^{14}$$

which is a 64σ event. Using the standard tail bound for the normal distribution, even with 2^{25} counters in total (across 512 positions), we would expect to see only $2^{18} \cdot e^{-2048}/\pi \ll 1$ such events.

We found that 643 (less than 1%) of the large biases that we observed were at least twice the size (in absolute value) of biases resulting from the products of single-byte biases or of the expected Fluhrer-McGrew bias in the same positions. In other words, most of the large biases that we observed arise from the product distribution or are explained by Fluhrer

and McGrew's results. We also note that we did find double-byte biases in all the positions predicted by Fluhrer and McGrew [64] starting from byte pair (Z_4, Z_5) onwards. This is not surprising given that the idealised assumption concerning the internal state of the RC4 algorithm that was used in the analysis of [64] is well approximated after a few invocations of the RC4 keystream generator. However, in many such cases, the magnitude of the bias we observed is greater than is predicted by the Fluhrer-McGrew analysis. For example, in byte pair (Z_6, Z_7) we observed

$$\Pr(Z_6 = \texttt{0x07} \wedge Z_7 = \texttt{0xFF}) = 2^{-16} \cdot (1 - 2^{-6.487}),$$

whereas the corresponding specified Fluhrer-McGrew probability for this byte pair, namely the $(i + 1, \texttt{0xFF})$ byte pair where i is the internal variable of the RC4 keystream generator, is $2^{16}(1 + 2^{-8})$.

We do, however, note a transition to the regular Fluhrer-McGrew double-byte biases from position 257 onwards. We also note the disappearance of the single-byte biases from roughly this point onwards. This is illustrated in Figure 4.4, which shows the absolute value of the largest single-byte bias observed in our data as a function of keystream position r.

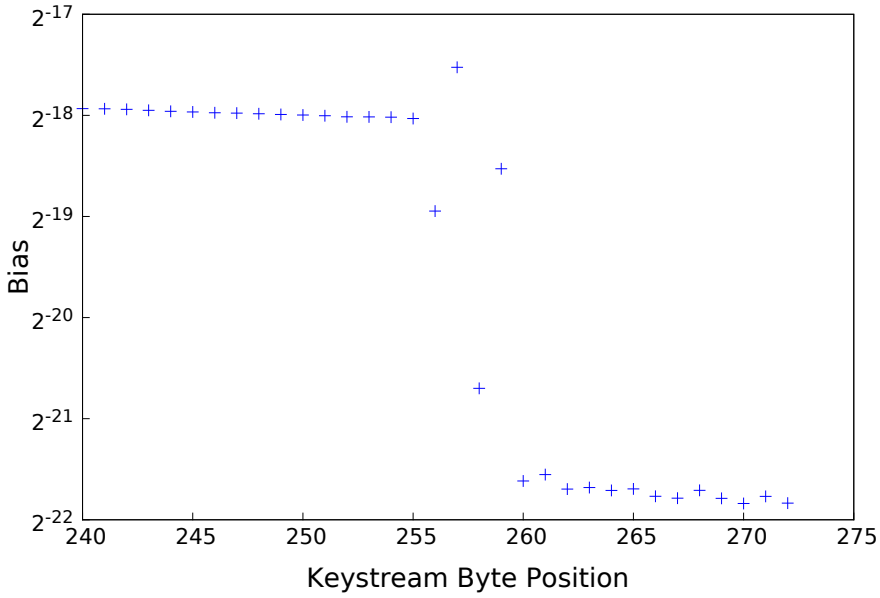


Figure 4.4: Absolute value of the largest single-byte bias for keystream bytes Z_{240} to Z_{272}.

4.2.5 RC4 and the TLS Record Protocol

We provide an overview of the TLS Record Protocol with RC4 selected as the method for encryption (further details for TLS 1.2 and below are given in Chapter 2 of this thesis).

Application data to be protected by TLS, i.e, a sequence of bytes or a record R, is processed as follows: An 8-byte sequence number `SQN`, a 5-byte header `HDR` and R are concatenated to form the input to an HMAC function. We let T denote the resulting output of this function. In the case of RC4 encryption, the plaintext, $P = T||R$, is XORed byte-per-byte with the RC4 keystream. In other words,

$$C_r = P_r \oplus Z_r,$$

for the r^{th} bytes of the ciphertext, plaintext and RC4 keystream respectively (for $r = 1, 2, 3 \dots$). The data that is transmitted has the form $\texttt{HDR}||C$, where C is the concatenation of the individual ciphertext bytes.

The RC4 algorithm is initialised in the standard way at the start of each TLS connection with a 128-bit encryption key. This key, k, is derived from the TLS master secret that is established during the TLS Handshake Protocol; k is either established via the full TLS Handshake Protocol or TLS session resumption. The first few bytes to be protected by RC4 encryption is a `Finished` message of the TLS Handshake Protocol. We do not target this record in our attacks since this message is not constant over multiple sessions. The exact size of this message is important in dictating how far down the keystream our target plaintext will be located; in turn this determines whether or not it can be recovered using only single-byte biases. A common size is 36 bytes, but the exact size depends on the output size of the TLS pseudo-random function (PRF) used in computing the `Finished` message and of the hash function used in the HMAC algorithm in the Record Protocol.

Decryption is the reverse of the process described above. As noted in [10], any error in decryption is treated as fatal – an error message is sent to the sender and all cryptographic material, including the RC4 key, is disposed of. This enables an active attacker to force the use of new encryption and MAC keys: the attacker can induce session termination, followed by a new session being established when the next message is sent over TLS, by simply modifying a TLS Record Protocol message. This could be used to ensure that the

target plaintext in an attack is repeatedly sent under the protection of a fresh RC4 key. However, this approach is relatively expensive since it involves a rerun of the full TLS Handshake Protocol, involving multiple public key operations and, more importantly, the latency involved in an exchange of 4 messages (2 complete round-trips) on the wire. A better approach is to cause the TCP connection carrying the TLS traffic to close, either by injecting sequences of `FIN` and `ACK` messages in both directions, or by injecting a `RST` message in both directions. This causes the TLS connection to be terminated, but not the TLS session (assuming the session is marked as "resumable" which is typically the case). This behaviour is codified in [50, Section 7.2.1]. Now when the next message is sent over TLS, a TLS session resumption instance of the Handshake Protocol is executed to establish a fresh key for RC4. This avoids the expensive public key operations and reduces the TLS latency to 1 round-trip before application data can be sent.

4.2.6 Passwords

Text-based passwords are arguably the dominant mechanism for authenticating users to web-based services and computer systems. As is to be expected of user-selected secrets, passwords do not follow uniform distributions. Various password breaches of recent years, including the Adobe breach of 150 million records in 2013 and the RockYou leak of 32.6 million passwords in 2009, attest to this with passwords such as `123456` and `password` frequently being counted amongst the most popular.[4] For example, our own analysis of the RockYou password data set confirmed this: the number of unique passwords in the RockYou dataset is 14,344,391, meaning that (on average) each password was repeated 2.2 times, and we indeed found the most common password to be `123456` (accounting for about 0.9% of the entire data set). Our later simulations will make extensive use of the RockYou data set as an attack dictionary. A more-fine grained analysis of this data set can be found in [153]. We also make use of data from the Singles.org breach for generating our target passwords. Singles.org is a now-defunct Christian dating website that was breached in 2009; religiously-inspired passwords such as `jesus` and `angel` appear with high frequency in its 12,234 distinct entries, making its frequency distribution quite different from that of the RockYou set.

[4] A comprehensive list of data breaches, including password breaches, can be found at `http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/`.

There is an extensive literature regarding the reasons for poor password selection and usage, including [7, 62, 154, 155]. In [38], Bonneau formalised a number of different metrics for analysing password distributions and studied a corpus of 70M Yahoo! passwords (collected in a privacy-preserving manner). His work highlights the importance of careful validation of password guessing attacks, in particular, the problem of estimating attack complexities in the face of passwords that occur rarely – perhaps uniquely – in a data set, the so-called *hapax legomena* problem. The approach to validation that we adopt benefits from the analysis of [38], as explained further in Section 4.4.

4.3 Plaintext Recovery via Bayesian Analysis

Our Bayesian analysis concerns vectors of consecutive plaintext bytes, which is appropriate given passwords as the plaintext target. This however means that the keystream distribution statistics also need to be for vectors of consecutive keystream bytes. Such statistics do not exist in the prior literature on RC4, except for the Fluher-McGrew biases [64] (which supply the distributions for adjacent byte pairs far down the keystream). Fortunately, in the early bytes of the RC4 keystream, the single-byte biases are dominant enough that a simple product distribution can be used as a reasonable estimate for the distribution on vectors of keystream bytes. We also show how to build a more accurate approximation to the relevant keystream distributions using double-byte distributions. This approximation is not only more accurate but also *necessary* when the target plaintext is located further down the stream, where the single-byte biases disappear and where double-byte biases become dominant. Indeed, our double-byte-based approximation to the keystream distribution on vectors can be used to smoothly interpolate between the region where single-byte biases dominate and where the double-byte biases come into play (which is exhibited as a fairly sharp transition around position 256 in the keystream, see Figure 4.4).

In the end, what we obtain is a formal algorithm that estimates the likelihood of each password in a dictionary based on both the *a priori* password distribution and the observed ciphertexts. This formal algorithm is amenable to efficient implementation using either the single-byte based product distribution for keystreams or the double-byte-based approximation to the distribution on keystreams.

We now present our formal Bayesian analysis of plaintext recovery attacks in the broadcast setting for stream ciphers. We then apply this to the problem of extracting passwords, specialising the formal analysis and making it implementable in practice based only on the single-byte and double-byte keystream distributions.

4.3.1 Formal Bayesian Analysis

Suppose we have a candidate set of N plaintexts, denoted $\mathcal{X}$, with the *a priori* probability of an element $x \in \mathcal{X}$ being denoted p_x. We assume for simplicity that all the candidates consist of byte strings of the same length, n. For example $\mathcal{X}$ might consist of all the passwords of a given length n from some breach data set, and then p_x can be computed as the relative frequency of x in the data set. If the frequency data is not available, then the uniform distribution on $\mathcal{X}$ can be assumed.

Next, suppose that a plaintext from $\mathcal{X}$ is encrypted S times, each time under independent, random keys using a stream cipher such as RC4. Suppose also that the first character of the plaintext always occurs in the same position r in the plaintext stream in each encryption. Let $c = (c_{ij})$ denote the $S \times n$ matrix of bytes in which row i, denoted $c^{(i)}$ for $0 \leq i < S$, is a vector of n bytes corresponding to the values in positions $r, \ldots, r+n-1$ in ciphertext i. Let X be the random variable denoting the (unknown) value of the plaintext.

We wish to form a maximum *a posteriori* estimate for X, given the observed data c and the *a priori* probability distribution p_x, that is, we wish to maximise $\Pr(X = x \mid C = c)$ where C is a random variable corresponding to the matrix of ciphertext bytes.

Using Bayes' theorem, we have

$$\Pr(X = x \mid C = c) = \frac{\Pr(C = c \mid X = x) \cdot \Pr(X = x)}{\Pr(C = c)}.$$

Here the term $\Pr(X = x)$ corresponds to the *a priori* distribution p_x on $\mathcal{X}$. The term $\Pr(C = c)$ is independent of the choice of x (as can be seen by writing $\Pr(C = c) = \sum_{x \in \mathcal{X}} \Pr(C = c \mid X = x) \cdot \Pr(X = x)$). Since we are only interested in maximising $\Pr(X = x \mid C = c)$, we ignore this term henceforth.

Now, since ciphertexts are formed by XORing keystreams z [5] and plaintext x, we can write

$$\Pr(C = c \mid X = x) = \Pr(W = w)$$

where w is the $S \times n$ matrix formed by XORing each row of c with the vector x and W is a corresponding random variable. Then to maximise $\Pr(X = x \mid C = c)$, it suffices to maximise the value of

$$\Pr(X = x) \cdot \Pr(W = w)$$

over $x \in \mathcal{X}$. Let $w^{(i)}$ denote the i-th row of the matrix w, so $w^{(i)} = c^{(i)} \oplus x$. Then $w^{(i)}$ can be thought of as a vector of keystream bytes (coming from positions $r, \ldots r + n - 1$) induced by the candidate x, and we can write

$$\Pr(W = w) = \prod_{i=0}^{S-1} \Pr(Z = w^{(i)})$$

where, on the right-hand side of the above equation, Z denotes a random variable corresponding to a vector of bytes of length n starting from position r in the keystream. We can rewrite this as:

$$\Pr(W = w) = \prod_{z \in \mathcal{B}^n} \Pr(Z = z)^{N_{x,z}}$$

where the product is taken over all possible byte strings of length n and $N_{x,z}$ is defined as:

$$N_{x,z} = |\{i : z = c^{(i)} \oplus x\}_{0 \leq i < S}|,$$

that is, $N_{x,z}$ counts the number of occurrences of vector z in the rows of the matrix formed by XORing each row of c with candidate x. Putting everything together, our objective is to compute for each candidate $x \in \mathcal{X}$ the value:

$$\Pr(X = x) \cdot \prod_{z \in \mathcal{B}^n} \Pr(Z = z)^{N_{x,z}}$$

and then to rank these values in order to determine the most likely candidate(s).

Notice that the expressions here involve terms $\Pr(Z = z)$ which are probabilities of occurrence for n consecutive bytes of keystream. Such estimates are not generally available in the literature, and for the values of n we are interested in (corresponding to putative

[5] Note that we now let z denote n consecutive bytes of keystream. Previously, we used z to denote a byte value.

Algorithm 3: Single-byte attack

input : $c_{i,j} : 0 \le i < S, 0 \le j < n$ – array formed from S independent encryptions of fixed n-byte candidate X
r – starting position of X in plaintext stream
$\mathcal{X}$ – collection of N candidates
p_x – *a priori* probability of candidates $x \in \mathcal{X}$
$p_{r+j,z}$ $(0 \le j < n,\ z \in \mathcal{B})$ – single-byte keystream distribution

output : $\{\gamma_x : x \in \mathcal{X}\}$ – set of (approximate) log likelihoods for candidates in $\mathcal{X}$

begin

- **for** $j = 0$ **to** $n - 1$ **do**
  - **for** $z =$ `0x00` **to** `0xFF` **do**
    - $N'_{z,j} \leftarrow 0$
- **for** $j = 0$ **to** $n - 1$ **do**
  - **for** $i = 0$ **to** $S - 1$ **do**
    - $N'_{c_{i,j},j} \leftarrow N'_{c_{i,j},j} + 1$
- **for** $j = 0$ **to** $n - 1$ **do**
  - **for** $y =$ `0x00` **to** `0xFF` **do**
    - **for** $z =$ `0x00` **to** `0xFF` **do**
      - $N_{y,z,j} \leftarrow N'_{z \oplus y,j}$
    - $L_{y,j} = \sum_{z \in \mathcal{B}} N_{y,z,j} \log(p_{r+j,z})$,
- **for** $x = (x_0, \ldots, x_{n-1}) \in \mathcal{X}$ **do**
  - $\gamma_x \leftarrow \log(p_x) + \sum_{j=0}^{n-1} L_{x_j,j}$
- **return** $\{\gamma_x : x \in \mathcal{X}\}$

password lengths), obtaining accurate estimates for them by sampling many keystreams would be computationally prohibitive. For example, our computation for double-byte probabilities discussed in Section 4.2.4 involved 2^{44} keystreams and, with highly optimised code, consumed roughly 4800 core-days of computation. This yields the required probabilities only for $n = 2$. Moreover, the product $\prod_{z \in \mathcal{B}^n}$ involves 2^{8n} terms and is not amenable to calculation. Thus we must turn to approximate methods to make further progress.

Note also that taking $n = 1$ in the above analysis, we obtain exactly the same approach as was used in the single-byte attack in [10], except that we include the *a priori* probabilities $\Pr(X = x)$ whereas these were (implicitly) assumed to be uniform in [10].

4.3.2 Using a Product Distribution

Our task is to derive simplified ways of computing the expression

$$\Pr(X = x) \cdot \prod_{z \in \mathcal{B}^n} \Pr(Z = z)^{N_{x,z}}$$

and then to apply these to produce efficient algorithms for computing (approximate) likelihoods of candidates $x \in \mathcal{X}$.

The simplest approach is to assume that the n bytes of the keystreams can be treated independently. For RC4, this is actually a very good approximation in the regime where single-byte biases dominate (that is, in the first 256 positions). Thus, writing $Z = (Z_r, \ldots, Z_{r+n-1})$ and $z = (z_r, \ldots, z_{r+n-1})$ (with the subscript r denoting the position of the first keystream byte of interest), we have:

$$\Pr(Z = z) \approx \prod_{j=0}^{n-1} \Pr(Z_{r+j} = z_{r+j}) = \prod_{j=0}^{n-1} p_{r+j,z}$$

where now the probabilities appearing on the right-hand side are single-byte keystream probabilities, as reported in [10] for example. Then writing $x = (x_0, \ldots, x_{n-1})$ and rearranging terms, we obtain:

$$\prod_{z \in \mathcal{B}^n} \Pr(Z = z)^{N_{x,z}} \approx \prod_{j=0}^{n-1} \prod_{z \in \mathcal{B}} {p_{r+j,z}}^{N_{x_j,z,j}}.$$

where $N_{y,z,j} = |\{i : z = c_{i,j} \oplus y\}_{0 \le i < S}|$ counts (now for single bytes instead of length n vectors of bytes) the number of occurrences of byte z in the column vector formed by XORing column j of c with a candidate byte y.

Notice that, as in [10], the counters $N_{y,z,j}$ for $y \in \mathcal{B}$ can all be computed efficiently by permuting the counters $N_{\mathtt{0x00},z,j}$, these being simply counters for the number of occurrences of each byte value z in column j of the ciphertext matrix c.

In practice, it is more convenient to work with logarithms, converting products into sums, so that we evaluate for each candidate $x = (x_0, \ldots, x_{n-1})$ an expression of the form

$$\gamma_x := \log(p_x) + \sum_{j=0}^{n-1} \sum_{z \in \mathcal{B}} N_{x_j,z,j} \log(p_{r+j,z}).$$

Given a large set of candidates $\mathcal{X}$, we can streamline the computation by first computing the counters $N_{y,z,j}$, then, for each possible byte value y, the value of the inner sum $\sum_{z\in\mathcal{B}} N_{y,z,j} \log(p_{r+j,z})$, and then reusing these individual values across all the relevant candidates x for which $x_j = y$. This reduces the evaluation of γ_x for a single candidate x to $n+1$ additions of real numbers.

The above procedure, including the various optimisations, is specified as an attack in Algorithm 3. We refer to it as our single-byte attack because of its reliance on the single-byte keystream probabilities $p_{r+j,z}$. It outputs a collection of approximate log likelihoods $\{\gamma_x : x \in \mathcal{X}\}$ for each candidate $x \in \mathcal{X}$. These can be further processed to extract, for example, the candidate with the highest score, or the top T candidates.

4.3.3 Double-byte-based Approximation

We continue to write $Z = (Z_r, \ldots, Z_{r+n-1})$ and $z = (z_r, \ldots, z_{r+n-1})$ and aim to find an approximation for $\Pr(Z = z)$ which lends itself to efficient computation of approximate log likelihoods as in our first algorithm. Now we rely on the double-byte keystream distribution, writing

$$p_{s,z_1,z_2} := \Pr((Z_s, Z_{s+1}) = (z_1, z_2)), \quad s \geq 1, k_1, k_2 \in \mathcal{B}$$

for the probabilities of observing bytes (z_1, z_2) in the RC4 keystream in positions $(s, s+1)$. We estimated these probabilities for r in the range $1 \leq r \leq 511$ using 2^{44} RC4 keystreams; for larger r, these are well approximated by the Fluhrer-McGrew biases [64] (as was verified in [10]).

We now make the assumption that, for each j,

$$\begin{aligned} \Pr(Z_j = z_j \mid Z_{j-1} = z_{j-1} \wedge \cdots \wedge Z_0 = z_0) \\ \approx \Pr(Z_j = z_j \mid Z_{j-1} = z_{j-1}), \end{aligned} \tag{4.1}$$

meaning that byte j in the keystream can be modelled as depending only on the preceding byte and not on earlier bytes. We can write

$$\Pr(Z_j = z_j \mid Z_{j-1} = z_{j-1}) = \frac{\Pr(Z_j = z_j \wedge Z_{j-1} = z_{j-1})}{\Pr(Z_{j-1} = z_{j-1})}$$

where the numerator can then be replaced by p_{j-1,z_{j-1},z_j} and the denominator by $p_{j-1,z_{j-1}}$, a single-byte keystream probability. Then using an inductive argument and our assumption (4.1), we easily obtain:

$$\Pr(Z = z) \approx \frac{\prod_{j=0}^{n-2} p_{r+j,z_j,z_{j+1}}}{\prod_{j=1}^{n-2} p_{r+j,z_j}}$$

giving an approximate expression for our desired probability in terms of single-byte and double-byte probabilities. Notice that if we assume that the adjacent byte pairs are independent, then we have $p_{r+j,z_j,z_{j+1}} = p_{r+j,z_j} \cdot p_{r+j+1,z_{j+1}}$ and the above expression collapses down to the one we derived in the previous subsection.

For candidate x, we again write $x = (x_0, \ldots, x_{n-1})$ and rearranging terms, we obtain:

$$\prod_{z \in \mathcal{B}^n} \Pr(Z = z)^{N_{x,z}} \approx \frac{\prod_{j=0}^{n-2} \prod_{z_1 \in \mathcal{B}} \prod_{z_2 \in \mathcal{B}} p_{r+j,z_1,z_2}^{N_{x_j,x_{j+1},z_1,z_2,j}}}{\prod_{j=1}^{n-2} \prod_{z \in \mathcal{B}} p_{r+j,z}^{N_{x_j,z,r+j}}}.$$

where $N_{y_1,y_1,z_1,z_2,j} = |\{i : z_1 = c_{i,j} \oplus y_1 \wedge z_2 = c_{i,j+1} \oplus y_2\}_{0 \leq i < S}|$ counts (now for consecutive pairs of bytes) the number of occurrences of bytes (z_1, z_2) in the pair of column vectors formed by XORing columns $(j, j+1)$ of c with candidate bytes (y_1, y_2) (and where $N_{x_j,z,r+j}$ is as in our previous algorithm).

Again, the counters $N_{y_1,y_2,z_1,z_2,j}$ for $y_1, y_2 \in \mathcal{B}$ can all be computed efficiently by permuting the counters $N_{\texttt{0x00},\texttt{0x00},z_1,z_2,j}$, these being simply counters for the number of occurrences of pairs of byte values (z_1, z_2) in column j and $j+1$ of the ciphertext matrix c. As before, we work with logarithms, so that we evaluate for each candidate $x = (x_0, \ldots, x_{n-1})$ an expression of the form

$$\gamma_x := \log(p_x) + \sum_{j=0}^{n-2} \sum_{z_1 \in \mathcal{B}} \sum_{z_2 \in \mathcal{B}} N_{x_j,x_{j+1},z_1,z_2,j} \log(p_{r+j,z_1,z_2}) - \sum_{j=1}^{n-2} \sum_{z \in \mathcal{B}} N_{x_j,z,r+j} \log(p_{r+j,z}).$$

With appropriate pre-computation of the terms $N_{y_1,y_2,z_1,z_2,j} \log(p_{r+j,z_1,z_2})$ and $N_{y,z,r+j} \log(p_{r+j,z})$ for all y_1, y_2 and all y, the computation for each candidate $x \in \mathcal{X}$ can be reduced to roughly $2n$ floating point additions. The pre-computation can be further reduced by computing the terms for only those pairs (y_1, y_2) actually arising in some candidate in $\mathcal{X}$ in positions $(j, j+1)$. We use this further optimisation in our implementation.

The above procedure is specified as an attack in Algorithm 4. We refer to it as our double-byte attack because of its reliance on the double-byte keystream probabilities p_{s,z_1,z_2}. It again outputs a collection of approximate log likelihoods $\{\gamma_x : x \in \mathcal{X}\}$ for each candidate $x \in \mathcal{X}$, suitable for further processing such as extracting the candidate with the highest score, or find the top T candidates. Note that for simplicity of presentation, it involves a quintuply-nested loop to compute the values $N_{y_1,y_2,z_1,z_2,j}$; these values should of course be directly computed from the $(n-1) \cdot 2^{16}$ pre-computed counters $N'_{c_{i,j},c_{i,j+1},j}$ in an in-line fashion using the formula $N_{y_1,y_2,z_1,z_2,j} = N'_{z_1 \oplus y_1, z_2 \oplus y_2,,j}$.

4.4 Simulation Results

We performed extensive simulations of both of our attacks, varying the different parameters to evaluate their effects on success rates.

4.4.1 Methodology

We focus on the problem of password recovery, using the RockYou data set as an attack dictionary and the Singles.org data set as the set of target passwords. Except where noted, in each simulation, we performed 256 independent runs of the relevant attack. In each attack simulation, we select a password of some fixed length n from the Singles.org password data set according to the known *a priori* probability distribution for that data set, encrypt it S times in different starting positions r using random 128-bit keys for RC4, and then attempt to recover the password from the ciphertexts using the set of all passwords of length n from the entire RockYou data set (14 million passwords) as our candidate set $\mathcal{X}$. We declare success if the target password is found within the top T passwords suggested by the algorithm (according to the approximate likelihood measures γ_x). Our default settings, unless otherwise stated, are $n = 6$ and $T = 5$, and we try all values for r between 1 and $256 - n + 1$, where the single-byte biases dominate the behaviour of the RC4 keystreams. Typical values of S are 2^s where $s \in \{20, 22, 24, 26, 28\}$.

Using different data sets for the attack dictionary and the target set from which encrypted passwords are chosen is more realistic than using a single dictionary for both purposes, not least because in a real attack, the exact content and *a priori* distribution of the target set

Algorithm 4: Double-byte attack

input : $c_{i,j} : 0 \le i < S, 0 \le j < n$ – array formed from S independent encryptions of fixed n-byte candidate X
r – starting position of X in plaintext stream
$\mathcal{X}$ – collection of N candidates
p_x – *a priori* probability of candidates $x \in \mathcal{X}$
$p_{r+j,z}$ $(0 \le j < n,\ z \in \mathcal{B})$ – single-byte keystream distribution
p_{r+j,z_1,z_2} $(0 \le j < n-1,\ z_1, z_2 \in \mathcal{B})$ – double-byte keystream distribution

output : $\{\gamma_x : x \in \mathcal{X}\}$ – set of (approximate) log likelihoods for candidates in $\mathcal{X}$

begin

- **for** $j = 0$ **to** $n-2$ **do**
  - **for** $z_1 =$ `0x00` **to** `0xFF` **do**
    - $N'_{z,j} \leftarrow 0$
    - **for** $z_2 =$ `0x00` **to** `0xFF` **do**
      - $N'_{z_1,z_2,j} \leftarrow 0$
- **for** $j = 0$ **to** $n-2$ **do**
  - **for** $i = 0$ **to** $S-1$ **do**
    - $N'_{c_{i,j},j} \leftarrow N'_{c_{i,j},j} + 1$
    - $N'_{c_{i,j},c_{i,j+1},j} \leftarrow N'_{c_{i,j},c_{i,j+1},j} + 1$
- **for** $j = 1$ **to** $n-2$ **do**
  - **for** $y =$ `0x00` **to** `0xFF` **do**
    - **for** $z =$ `0x00` **to** `0xFF` **do**
      - $N_{y,z,j} \leftarrow N'_{z \oplus y,j}$
    - $L_{y,j} = \sum_{z \in \mathcal{B}} N_{y,z,j} \log(p_{r+j,z})$,
- **for** $j = 0$ **to** $n-2$ **do**
  - **for** $y_1 =$ `0x00` **to** `0xFF` **do**
    - **for** $y_2 =$ `0x00` **to** `0xFF` **do**
      - **for** $z_1 =$ `0x00` **to** `0xFF` **do**
        - **for** $z_2 =$ `0x00` **to** `0xFF` **do**
          - $N_{y_1,y_2,z_1,z_2,j} \leftarrow N'_{z_1 \oplus y_1, z_2 \oplus y_2,,j}$
      - $L_{y_1,y_2,j} = \sum_{z_1 \in \mathcal{B}} \sum_{z_2 \in \mathcal{B}} N_{y_1,y_2,z_1,z_2,j} \log(p_{r+j,z_1,z_2})$,
- **for** $x = (x_0, \ldots, x_{n-1}) \in \mathcal{X}$ **do**
  - $\gamma_x \leftarrow \log(p_x) + \sum_{j=0}^{n-2} L_{x_j,x_{j+1},j} - \sum_{j=1}^{n-2} L_{x_j,j}$
- **return** $\{\gamma_x : x \in \mathcal{X}\}$

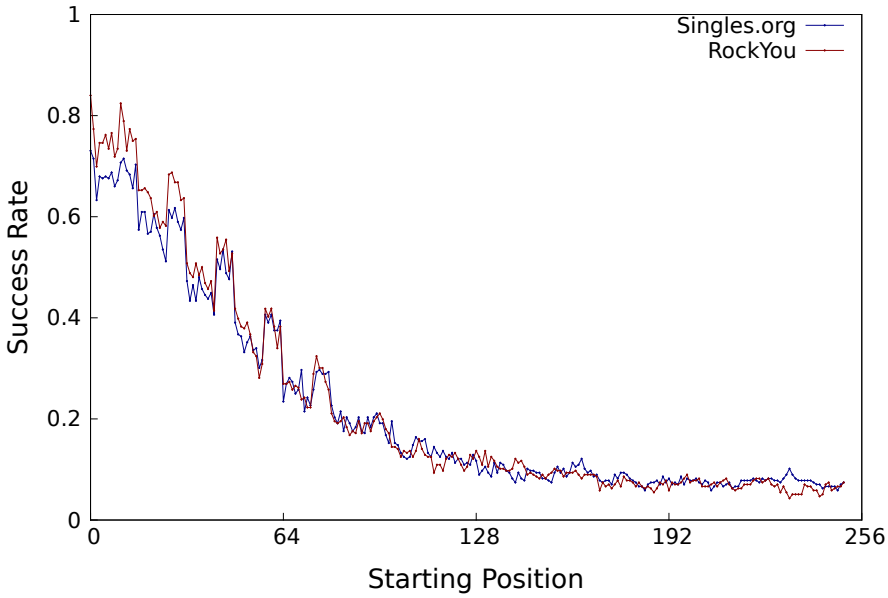


Figure 4.5: Recovery rate for Singles.org passwords using RockYou data set as dictionary, compared to recovery rate for RockYou passwords using RockYou data set as dictionary ($S = 2^{24}$, $n = 6$, $T = 5$, $1 \leq r \leq 251$, double-byte attack).

would not be known. This approach also avoids the problem of *hapax legomena* highlighted in [38]. However, this has the effect of limiting the success rates of our attacks to less than 100%, since there are highly likely passwords in the target set (such as `jesus`) that do not occur at all, or only have very low *a priori* probabilities in the attack dictionary, and conversely. Figure 4.5 compares the use of the RockYou password distribution to attack Singles.org passwords with the less realistic use of the RockYou password distribution to attack RockYou itself. It can be seen that, for the particular choice of attack parameters ($S = 2^{24}$, $n = 6$, $T = 5$, double-byte attack), the effect on success rate is not particularly large. However, for other attack parameters, as we will see below, we observe a maximum success rate of around 80% for our attacks, whereas we would achieve 100% success rates if we used RockYou against itself. The observed maximum success rate could be increased by augmenting the attack dictionary with synthetically generated, site-specific passwords and by removing RockYou-specific passwords from the attack dictionary. Our work did not explore these improvements.

Many data sets are available from password breaches. We settled on using RockYou for the attack dictionary because at the time this work was conducted, it was one of the biggest data sets in which all passwords and their associated frequencies were available, and because the distribution of passwords, while certainly skewed, was less skewed than

for other data sets. We used Singles.org for the target set because the Singles.org breach occurred later than the RockYou breach, so that the former could reasonably used as an attack dictionary for the latter. Moreover, the Singles.org distribution being quite different from that for RockYou makes password recovery against Singles.org using RockYou as a dictionary more challenging for our attacks. A detailed evaluation of the extent to which the success rates of our attacks depend on the choice of attack dictionary and target set is left to future work.

A limitation of our approach is that we assume the password length n to be already known, whereas in reality this may not be the case. At least four potential solutions to this problem exist. Firstly, in specific applications, n may leak via analysis of packet lengths or other forms of traffic analysis. Secondly we can run our attacks for the full range of password lengths, possibly adjusting the likelihood measure γ_x for each password candidate x to scale it appropriately by its length (except for the p_x term). A third approach is to augment the shorter passwords with the known plaintext that typically follows them in a specific targeted application protocol and then run our attacks for a fixed, but now longer, n. A fourth approach applies in protocols which use known delimiters to denote the end of a password (such as the = symbol seen at the end of base64 encodings for certain username/password lengths); here, the idea is to adapt our general attacks to compute the likelihood that such a delimiter appears in each possible position, and generate an estimate for n by selecting the position for which the likelihood is highest. We leave the exploration of these approaches to future work.

4.4.2 Results

Single-Byte Attack. We ran the attack described in Algorithm 3 with our default parameters ($n = 6$, $T = 5$, $1 \leq r \leq 251$) for $S = 2^s$ with $s \in \{20, 22, 24, 26, 28\}$ and evaluated the attack's success rate. We used our default of 256 independent runs per parameter set. The results are shown in Figure 4.6. We observe that:

- The performance of the attack improves markedly as S, the number of ciphertexts, increases, but the success rate is bounded by 75%. We attribute this to the use of one dictionary (RockYou) to recover passwords from another (Singles.org) – for the same attack parameters, we achieved 100% success rates when using RockYou against

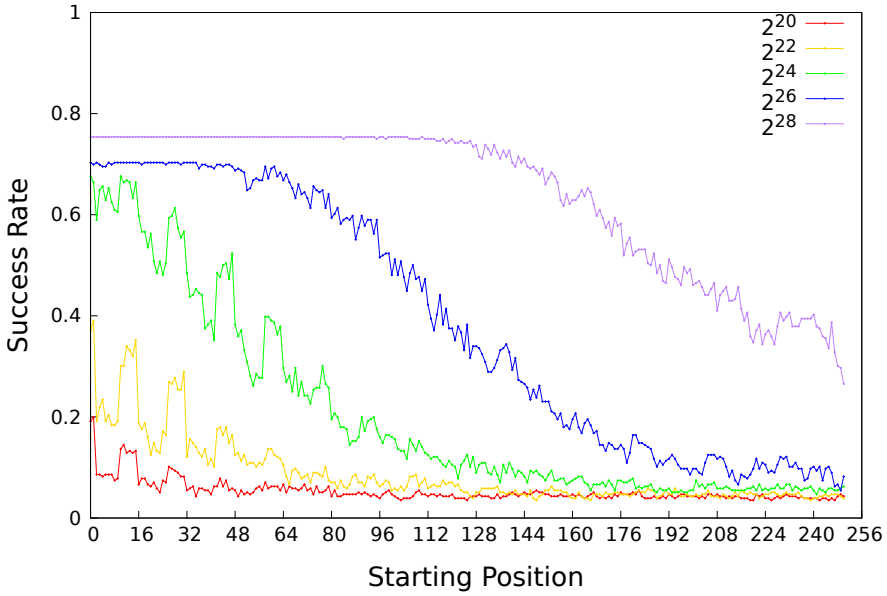


Figure 4.6: Recovery rates for single-byte algorithm for $S = 2^{20}, \ldots, 2^{28}$ ($n = 6$, $T = 5$, $1 \leq r \leq 251$).

RockYou, for example.

- For 2^{24} ciphertexts we see a success rate of greater than 60% for small values of r, the position of the password in the RC4 keystream. We see a drop to below 50% for starting positions greater than 32. We note the effect of the key-length-dependent biases on password recovery; passwords encrypted at starting positions $16\ell - n, 16\ell - n + 1, \ldots, 16\ell - 1, 16\ell$, where $\ell = 1, 2, \ldots, 6$, have a higher probability of being recovered in comparison to neighbouring starting positions.
- For 2^{28} ciphertexts we observe a success rate of more than 75% for $1 \leq r \leq 120$.

Double-Byte Attack. Analogously, we ran the attack of Algorithm 4 for $S = 2^s$ with $s \in \{20, 22, 24, 26, 28\}$ and our defaults of $n = 6$, $T = 5$. The results for these simulations are shown in Figure 4.7. Note that:

- Again, at 2^{24} ciphertexts the effect of key-length-dependent biases is visible.
- For 2^{26} ciphertexts we observe a success rate that is greater than 78% for $r \leq 48$.

Comparing the Single-Byte Attack with a Naive Algorithm. Figure 4.8 provides a comparison between our single-byte algorithm with $T = 1$ and a naive password recovery

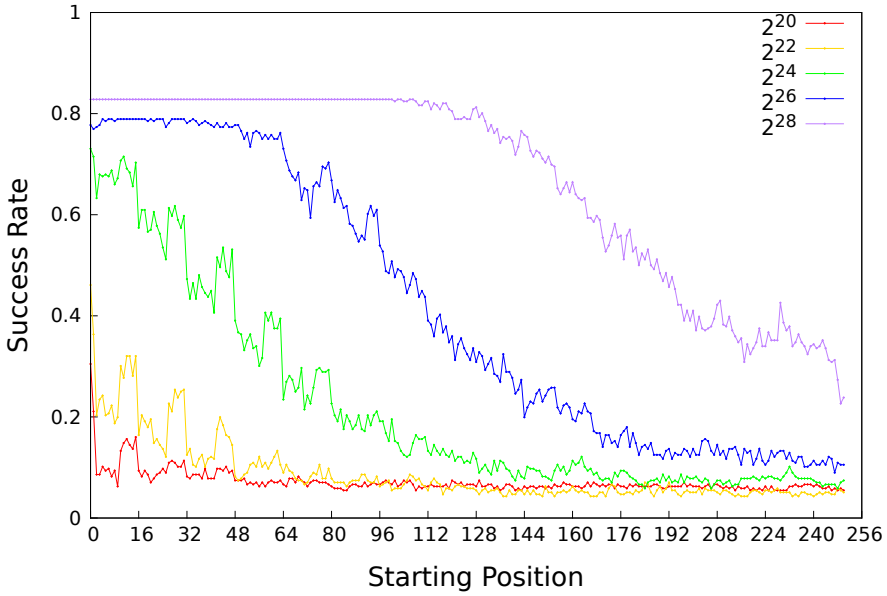


Figure 4.7: Recovery rates for double-byte algorithm for $S = 2^{20}, \ldots, 2^{28}$ ($n = 6$, $T = 5$, $1 \leq r \leq 251$).

attack based on the methods of [10], in which the password bytes are recovered one at a time by selecting the highest likelihood byte value in each position and declaring success if all bytes of the password are recovered correctly. Significant improvement over the naive attack can be observed, particularly for high values of r. For example with $S = 2^{24}$, the naive algorithm essentially has a success rate of zero for every r, whereas our single-byte algorithm has a success rate that exceeds 20% for $1 \leq r \leq 63$. By way of comparison, an attacker knowing the password length and using the obvious guessing strategy would succeed with probability 4.2% with a single guess, this being the *a priori* probability of the password `123456` amongst all length 6 passwords in the Singles.org dataset (and `123456` being the highest ranked password in the RockYou dictionary, so the first one that an attacker using this strategy with the RockYou dictionary would try). As another example, with $S = 2^{28}$ ciphertexts, a viable recovery rate is observed all the way up to $r = 251$ for our single-byte algorithm, whereas the naive algorithm fails miserably beyond $r = 160$ for even this large value of S. Note however that the naive attack can achieve a success rate of 100% for sufficiently large S, whereas our attack cannot. This is because the naive attack directly computes a password candidate rather than evaluating the likelihood of candidates from a list which may not contain the target password. On the other hand, our attack trivially supports larger values of T, whereas the naive attack is not so easily modified to enable this feature.

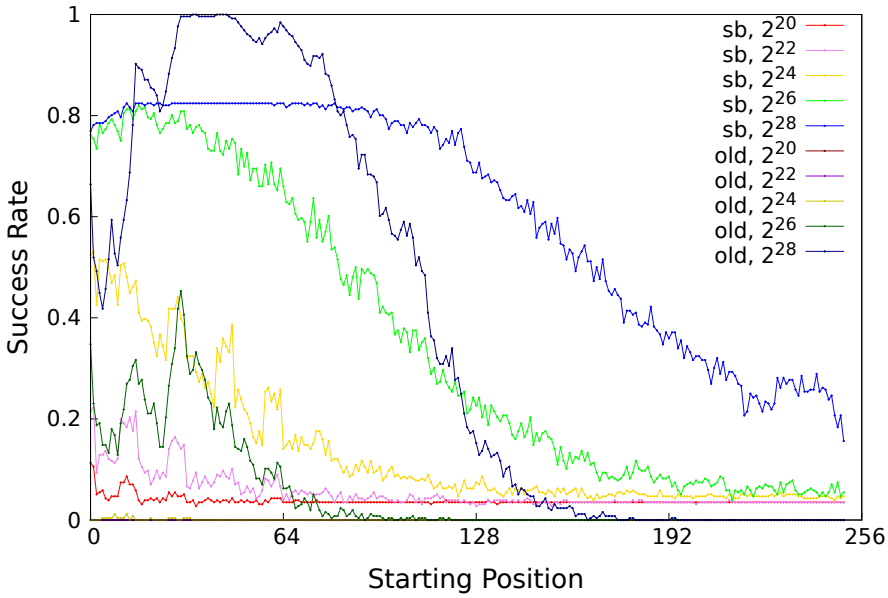


Figure 4.8: Performance of our single-byte algorithm versus a naive single-byte attack based on the methods of AlFardan *et al.* (labelled "old"). ($n = 6$, $T = 1$, $1 \leq r \leq 251$.)

Comparing the Single-Byte and Double-Byte Attacks. Figure 4.9 provides a comparison of our single-byte and double-byte attacks. With all other parameters equal, the success rates are very similar for the initial 256 positions. The reason for this is the absence of many strong double-byte biases that do not arise from products of the known single-byte biases in the early positions of the RC4 keystream.

Effect of the *a priori* Distribution. As a means of testing the extent to which our success rates are influenced by knowledge of the *a priori* probabilities of the candidate passwords, we ran simulations in which we tried to recover passwords sampled correctly from the Singles.org dataset but using a uniform *a priori* distribution for the RockYou-based dictionary used in the attack. Figure 4.10 shows the results ($S = 2^{24}$, $n = 6$, $T = 5$, double-byte attack) of these simulations, compared to the results we obtain by exploiting the *a priori* probabilities in the attack. It can be seen that a significant gain is made by using the *a priori* probabilities, with the uniform attack's success rate rapidly dropping to zero at around $r = 128$.

Effect of Password Length. Figure 4.11 shows the effect of increasing n, the password length, on recovery rates, with the sub-figures showing the performance of our double-byte attack for different numbers of ciphertexts ($S = 2^s$ with $s \in \{24, 26, 28\}$). Other parameters are set to their default values. As intuition suggests, password recovery becomes more

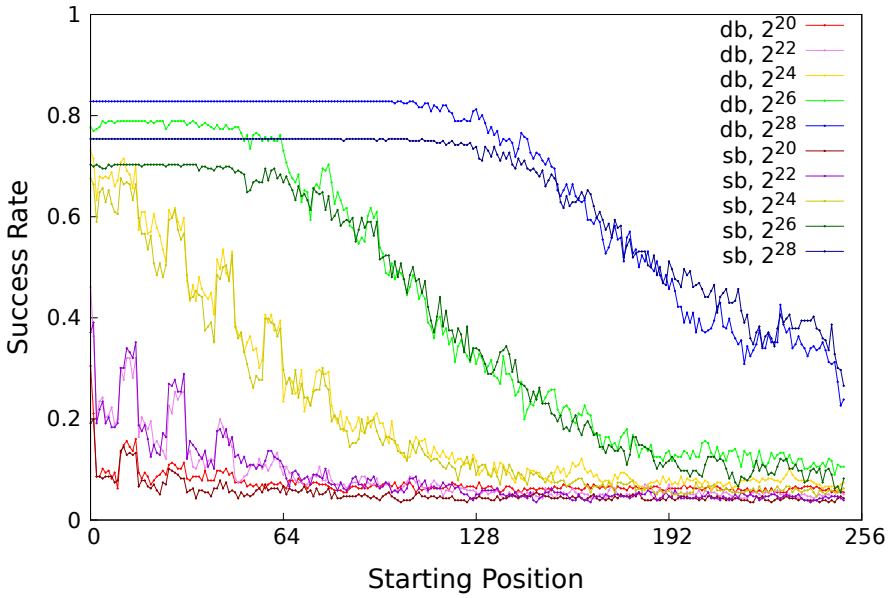


Figure 4.9: Recovery rate of single-byte versus double-byte algorithm for $S = 2^{20}, \ldots, 2^{28}$ ($n = 6$, $T = 5$, $1 \leq r \leq 251$).

difficult as the length increases. Also notable is that the ceiling on success rate of our attack decreases with increasing n, dropping from more than 80% for $n = 5$ to around 50% for $n = 8$. This is due to the fact that only 48% of the length 8 passwords in the Singles.org data set actually occur in the RockYou attack dictionary: our attack is doing as well as it can in this case, and we would expect stronger performance with an attack dictionary that is better matched to the target site.

Effect of Increasing Try Limit T. Recall that the parameter T defines the number of password trials our attacks make. The number of permitted attempts for specific protocols like BasicAuth is server-dependent and not mandated in the relevant specifications. Whilst not specific to our chosen protocols, a 2010 study [39] showed that 84% of websites surveyed allowed at least $T = 100$ attempts; many websites appear to actually allow $T = \infty$. Figure 4.12 shows the effect of varying T in our double-byte algorithm for different numbers of ciphertexts ($S = 2^s$ with $s \in \{24, 26, 28\}$). Other parameters are set to their default values. It is clear that allowing large values of T boosts the success rate of the attacks.

Note however that a careful comparison must be made between our attack with parameter T and the success rate of the obvious password guessing attack given T attempts. Such a guessing attack does not require any ciphertexts but instead uses the *a priori* distribution on passwords in the attack dictionary (RockYou) to make guesses for the target password

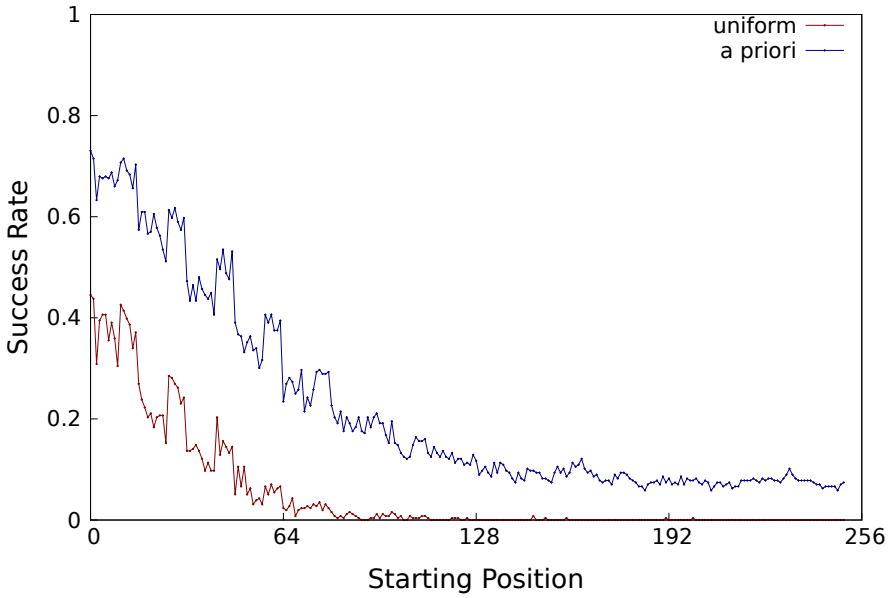
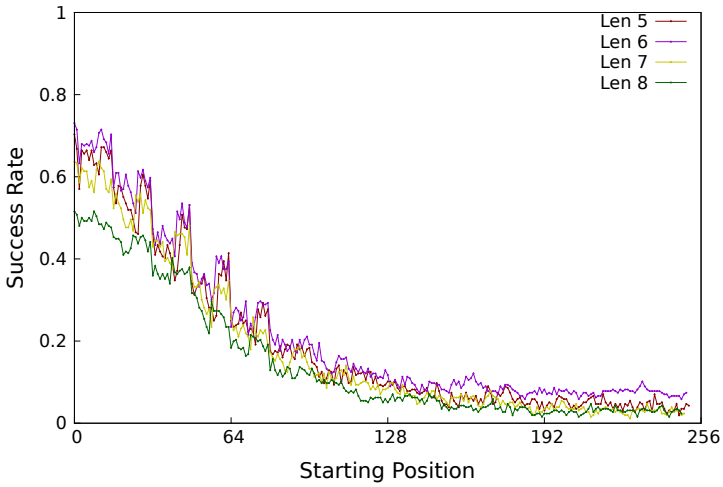


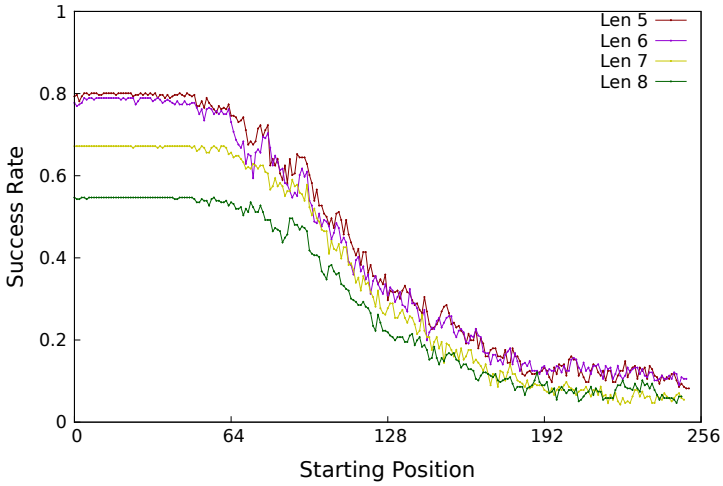
Figure 4.10: Recovery rate for uniformly distributed passwords versus known *a priori* distribution ($S = 2^{24}$, $n = 6$, $T = 5$, $1 \leq r \leq 251$, double-byte algorithm).

in descending order of probability, the success rate being determined by the *a priori* probabilities of the guessed passwords in the target set (Singles.org). Clearly, our attacks are only of value if they significantly out-perform this ciphertext-less attack.

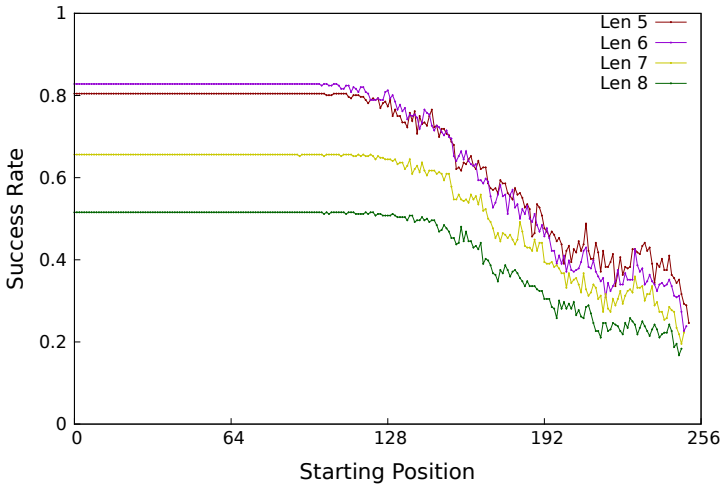
Figure 4.13 shows the results of plotting $\log_2(T)$ against success rate α for $S = 2^s$ with $s \in \{14, 16, \ldots, 28\}$. The figure then illustrates the value of T necessary in our attack to achieve a given password recovery rate α for different values of S. This measure is related to the α-work-factor metric explored in [38] (though with the added novelty of representing a work factor when one set of passwords is used to recover passwords from a different set). To generate this figure, we used 1024 independent runs rather than the usual 256, but using a fixed set of 1024 passwords sampled according to the *a priori* distribution for Singles.org. This was in an attempt to improve the stability of the results (with small numbers of ciphertexts S, the success rate becomes heavily dependent on the particular set of passwords selected and their *a priori* probabilities, while we wished to have comparability across different values of S). The success rates shown are for our double-byte attack with $n = 6$ and $r = 133$, this specific choice of r being motivated by it being the location of passwords for our BasicAuth attack proof-of-concept when the Chrome browser is used (similar results are obtained for other values of r). The graph also shows the corresponding work factor T as a function of α for the guessing attack (labeled "optimal guessing" in the figure).



(a) 2^{24} ciphertexts



(b) 2^{26} ciphertexts



(c) 2^{28} ciphertexts

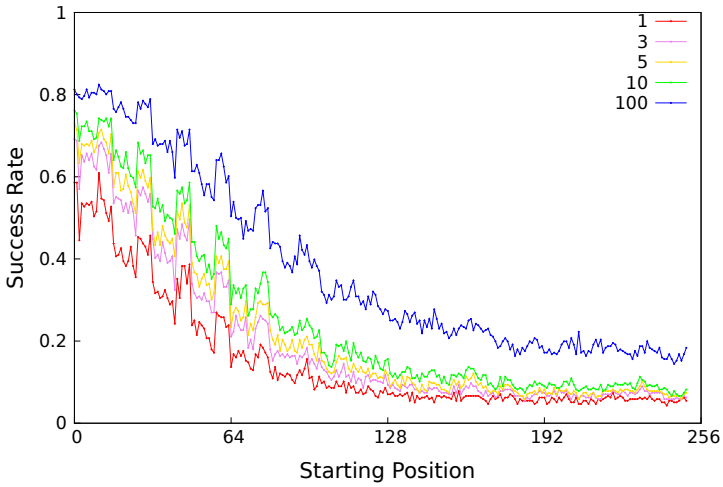
Figure 4.11: Effect of password length on recovery rate ($T = 5$, $1 \leq r \leq 251$, double-byte algorithm).

Figure 4.13a shows that our attack far outperforms the guessing attack for larger values of S, with a significant advantage accruing for $S = 2^{24}$ and above. However, as Figure 4.13b shows, the advantage over the guessing attack for smaller values of S, namely 2^{20} and below, is not significant. This can be attributed to our attack simply not being able to compute stable enough statistics for these small numbers of ciphertexts. In turn, this is because the expected random fluctuations in the keystream distributions overwhelm the small biases; in short, the signal does not sufficiently exceed the noise for these low values of S.

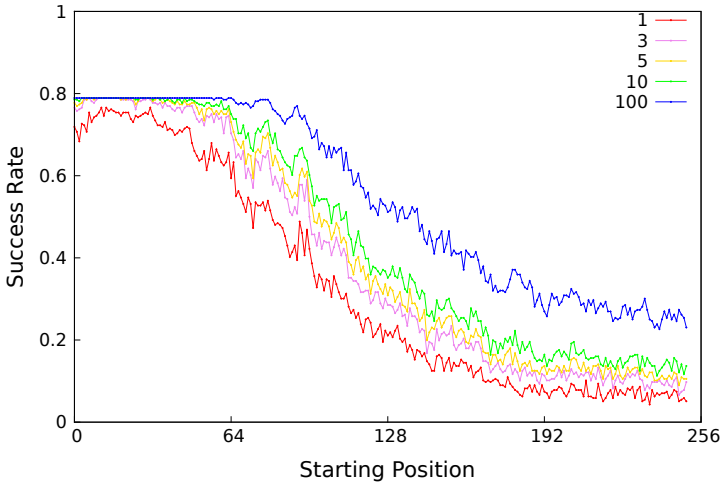
Effect of Base64 Encoding. We investigated the effect of base64 encoding of passwords on recovery rates, since many application layer protocols use such an encoding. The encoding increases the password length, making recovery harder, but also introduces redundancy, potentially helping the recovery process to succeed. Figure 4.14 shows our simulation results comparing the performance of our double-byte algorithm acting on 6-character passwords and on base64 encoded versions of them. It is apparent from the figure that the overall effect of the base64 encoding is to help our attack to succeed. In practice, the start of the target password may not be well-aligned with the base64 encoding process (for example, part of the last character of the username and/or a delimiter such as ":" may be jointly encoded with part of the first character of the password). This can be handled by building a special-purpose set of candidates $\mathcal{X}$ for each possibility. Handling this requires some care when mounting a real attack against a specific protocol; a detailed analysis is not conducted here.

Shifting Attack. It was observed in [10] and elsewhere that for 128-bit keys, RC4 keystreams exhibit particularly large "key-length-dependent" biases at positions $r = 16\ell$, $\ell = 1, \ldots, 7$, with the bias size decreasing with increasing ℓ. These large biases boost recovery rates, as already observed in our discussion of Figure 4.6.

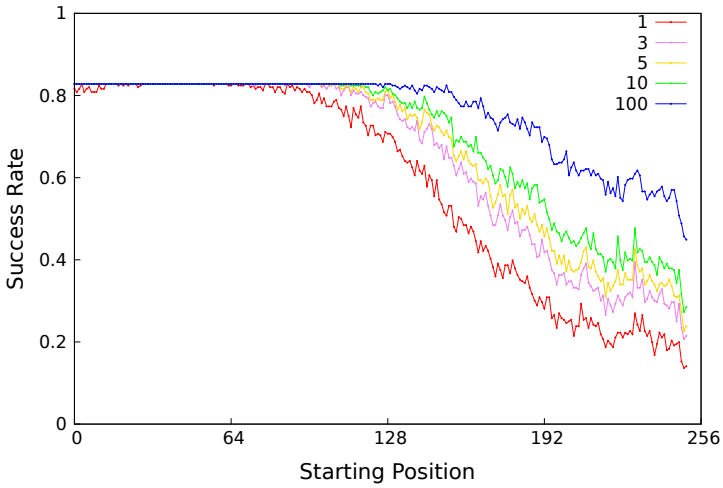
In certain application protocols and attack environments (such as HTTPS) it is possible for the adversary to incrementally pad the plaintext messages so that the unknown bytes are always aligned with positions having large keystream biases. Our algorithm descriptions and code are both easily modified to handle this situation, and we have conducted simulations with the resulting shift attack.



(a) 2^{24} ciphertexts

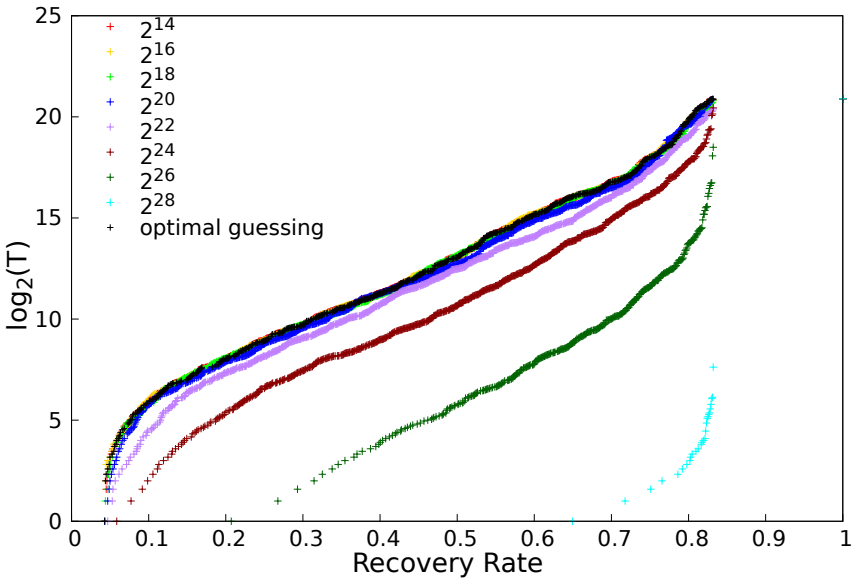


(b) 2^{26} ciphertexts

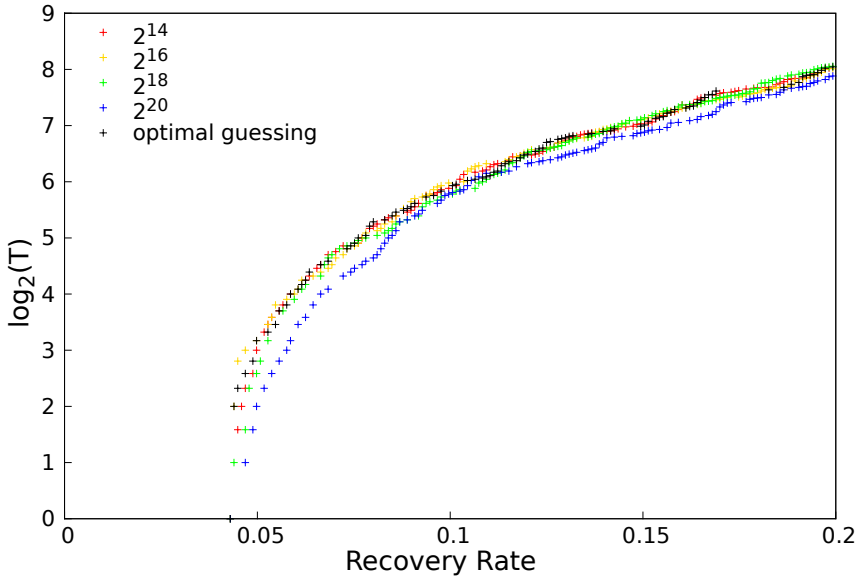


(c) 2^{28} ciphertexts

Figure 4.12: Effect of try limit T on recovery rate ($n = 6$, $1 \leq r \leq 251$, double-byte algorithm).



(a) $\alpha \in [0, 1]$



(b) $\alpha \in [0, 0.2]$

Figure 4.13: Value of T required to achieve a given password recovery rate α for $S = 2^s$ with $s \in \{14, 16, \ldots, 28\}$ ($n = 6$, $r = 133$, double-byte algorithm).

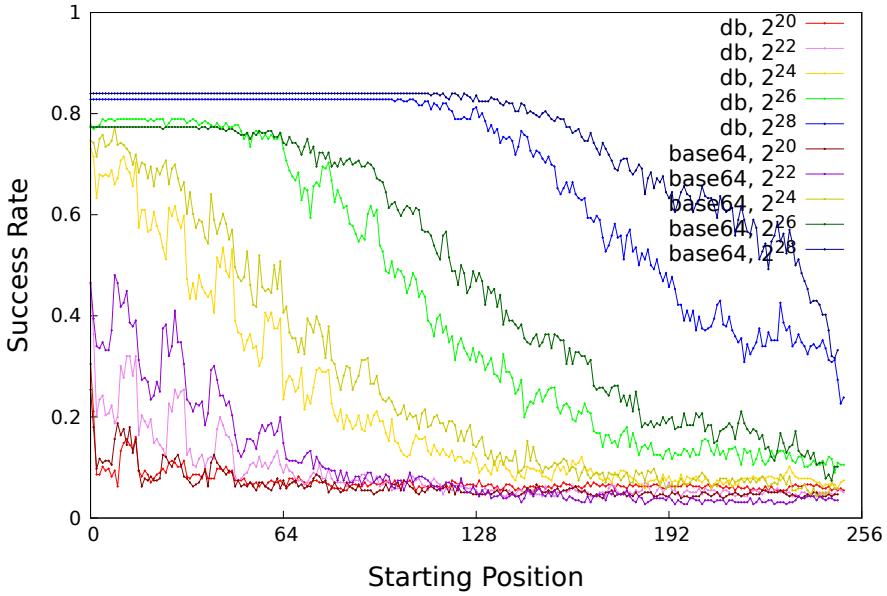


Figure 4.14: Recovery rate of base64 encoded password versus a "normal" password for 6-character passwords ($T = 5$, $1 \leq r \leq 251$, double-byte algorithm).

Figure 4.15 shows the results for the shift version of our double-byte algorithm. In the shift attack, the true number of ciphertexts is equal to $n \times S$, since we now use S ciphertexts at each of n shift positions. So a proper comparison would compare with one of our earlier attacks using an appropriately increased value of S. Making this adjustment, it can be seen that the success rate is significantly improved, particularly for small values of $r = 16\ell$ where the biases are biggest.

4.5 Practical Validation

We now describe a proof-of-concept implementation of our attacks against a specific application-layer protocol running over TLS, namely BasicAuth.

4.5.1 The BasicAuth Protocol

Defined as part of the HTTP/1.0 specification [24], the Basic Access Authentication scheme (BasicAuth) provides a means for controlling access to webpages and other protected resources. Here we provide a high-level overview of BasicAuth and direct the reader to [24] and [65] for further details.

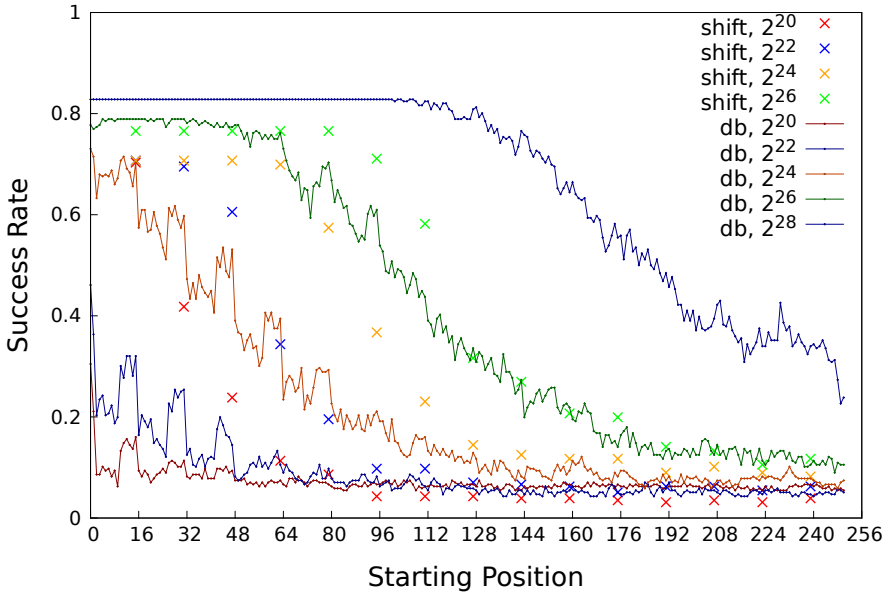


Figure 4.15: Recovery rate of shift attack versus double-byte algorithm ($n = 6$, $T = 5$, $1 \leq r \leq 251$).

BasicAuth is a challenge-response authentication mechanism: a server will present a client with a challenge to which the client must supply the correct response in order to gain access to the resource being requested. In the case of BasicAuth, the challenge takes the form of either a `401 Unauthorized` response message from an origin server, or a `407 Proxy Authentication Required` response message from a proxy server. BasicAuth requires that the client response contain legitimate user credentials – a username and password – in order for access to be granted. Certain web browsers may display a login dialog when the challenge is received and many browsers present users with the option of storing their user credentials in the browser, with the credentials thereafter being automatically presented on behalf of the user.

The client response to the challenge is of the form

`Authorization: Basic base64(userid:password)`

where `base64`$(\cdot)$ denotes the base64 encoding function (which maps 3 characters at a time onto 4 characters of output). Since the username and password are sent over the network as cleartext, BasicAuth needs to be used in conjunction with a protocol such as TLS.

4.5.2 Attacking BasicAuth

In order to obtain a working attack against BasicAuth, we need to ensure that two conditions are met:

- The base64-encoded password included in the BasicAuth client response can be located sufficiently early in the plaintext stream.
- There is a method for forcing a browser to repeatedly send the BasicAuth client response.

We observed that the first condition is met for particular browsers, including Google Chrome 38. For example, we inspected HTTPS traffic sent from Chrome to an iChair server.[6] We observed the user's base64-encoded password being sent with every HTTP(S) request in the same position in the stream, namely position $r = 133$ (this includes 16 bytes consumed by the client's `Finished` message as well as the 20-bytes consumed by the TLS Record Protocol MAC tag). For Mozilla Firefox 34, the value of r was the less useful 349.

For the second condition, we adopt the methods used in the BEAST, CRIME and Lucky 13 attacks on TLS, and also used in attacking RC4 in [10]: we assume that the user visits a site `www.evil.com` which loads JavaScript into the user's browser; the JavaScript makes GET or POST requests to the target website at `https://www.good.com` by using `XMLHttpRequest` objects (this is permitted under Cross Origin Resource Sharing (CORS), a mechanism developed to allow JavaScript to make requests to a domain other than the one from which the script originates). The base64-encoded BasicAuth password is automatically included in each such request. In order to force the password to be repeatedly encrypted at an early position in the RC4 keystream, we use a Man-In-The-Middle (MITM) attacker to break the TLS connection (by injecting sequences of TCP `FIN` and `ACK` messages into the connection). This requires some careful timing on the part of the JavaScript and the MITM attacker.

We built a proof-of-concept demonstration of these components to illustrate the principles. We set up a virtual network with three virtual machines each running Ubuntu 14.04,

[6] At the time this work was conducted, iChair was a popular system for conference reviewing, widely used in the cryptography research community and available from `http://www.baigneres.net/ichair`. It uses BasicAuth as its user authentication mechanism.

kernel version 3.13.0-32. On the first machine, we installed iChair. We configured the iChair web server to use RC4 as its default TLS cipher. The second machine was running the Chrome 38 browser and acted as the client in our attack. We installed the required JavaScript directly on this machine rather than downloading from another site. The third machine acted as the MITM attacker, required to intercept the TLS-protected traffic and to tear-down the TLS connections. We used the Python tool Scapy[7] to run an ARP poisoning attack on the client and server from the MITM so as to be able to intercept packets; with the connection hijacked we were able to force a graceful shutdown of the connection between the client and the server after the password-bearing record had been observed and recorded. We observed that forcing a graceful shutdown of each subsequent connection did allow for TLS resumption (rather than leading to the need for a full TLS Handshake run).

With this setup, the JavaScript running in the client browser sent successive HTTPS GET requests to the iChair server every 80ms. Our choice of 80ms was motivated by the fact that for our particular configuration, we observed a total time of around 80ms for TLS resumption, delivery of the password-bearing record and the induced shutdown of the TCP connection. This choice enabled us to capture 2^{16} encrypted password-bearing records in 1.6 hours (the somewhat greater than expected time here being due to anomalies in network behaviour). Running at this speed, the attack was stable over a period of hours.

We note that the latency involved in our setup is much lower than would be found in a real network in which the server may be many hops away from the client: between 500ms and 1000ms is typical for establishing an initial TLS 1.2 (and below) connection to a remote site, with the latency being roughly half that for session resumptions. Notably, the cost of public key operations is not the issue, but rather the network latency involved in the round-trips required for TCP connection establishment and then running the TLS Handshake. However, browsers also open up multiple TLS connections in parallel when fetching multiple resources from a site, as a means of reducing the latency perceived by users; the maximum number of concurrent connections per server is 6 for both the Chrome and Firefox browsers (though, we only ever saw roughly half this number in practice). This means that, assuming a TLS resumption latency (including the client's TCP `SYN`, delivery of the password-bearing record and the final, induced TCP `ACK`) of 250ms and the JavaScript is running fast enough to induce the browser to maintain 6 connections in

[7] Available at `http://www.secdev.org/projects/scapy/`.

parallel, the amount of time needed to mount an attack with $S = 2^{26}$ would be on the order of 776 hours. If the latency was further reduced to 100ms (because of proximity of the server to the client), the attack execution time would be reduced to 312 hours.

Again setting $n = 6$, $T = 100$, $r = 133$ and using the simulation results displayed in Figure 4.14, we would expect a success rate of 64.4% for this setup (with $S = 2^{26}$). For $T = 5$, the corresponding success rate would be 44.5%.

We emphasise that we did not execute a complete attack on these scales, but merely demonstrated the feasibility of the attack in our laboratory setup.

4.6 Conclusion

We have presented plaintext recovery attacks that derive from a formal Bayesian analysis of the problem of estimating plaintext likelihoods given an *a priori* plaintext distribution, suitable keystream distribution information, and a large number of encryptions of a fixed plaintext under independent keys. We applied these ideas to the specific problem of recovering passwords encrypted by the RC4 algorithm with 128-bit keys as used in TLS 1.2 and below, though they are of course more generally applicable – to uses of RC4 other than in TLS, and to stream ciphers with non-uniform keystream distributions in general. Using large-scale simulations, we have investigated the performance of these attacks under different settings for the main parameters.

We then studied the applicability of these attacks for the application layer protocol BasicAuth. For certain browsers and clients, user passwords were located at a favourable point in the plaintext stream and we could induce a password to be repeatedly encrypted under fresh, random keys. We built a proof-of-concept implementation of this attack. It was difficult to arrange for the rate of generation of encryptions to be as high as desired for a speedy attack. This was mainly due to the latency associated with TLS connection establishment (even with session resumption) rather than any fundamental barrier. We discussed scenarios in which the latency may be reduced.

Good-to-excellent password recovery success rates can be achieved using 2^{24} – 2^{28} ciphertexts in our attacks. We also demonstrated that our single-byte attack for password recovery

significantly outperforms a naive password recovery attack based on the ideas of [10]. We observed an improvement over a guessing strategy even for low numbers (2^{22} or 2^{24}) of ciphertexts. In contrast to these numbers, the preferred double-byte attack of [10] required on the order of 2^{34} encryptions to recover a 16-byte cookie. In view of our results, we feel justified in claiming that we significantly narrowed the gap between the feasibility results of [10] and our goal of achieving practical attacks on RC4 in TLS. In fact, by November 2015, a few months after our research was published, usage of RC4 in TLS had dropped to 8.3%, according to the ICSI Certificate Notary project. This accounts for a decrease in roughly 22% of RC4 usage since the release of our work in March 2015. Of course, other factors such as the deprecation of RC4 in TLS in February 2015 [122], as well as publication of concurrent work, [148] and [102], also contributed to the dramatic reduction of RC4 usage in TLS during this time. However, we would like to think that our work played as big of a role in bringing RC4 to the point where it had to be abandoned by practitioners. In early 2016, mainstream browsers such as Chrome and Firefox shipped without support for RC4[8], and today, usage of RC4 in TLS 1.2 and below doesn't even feature in an independent category on the ICSI Certificate Notary project website.[9] The degree to which it contributes to the 'other cipher suites' category which stands at 6% is unclear. Nevertheless, we determine it safe to conclude that usage of RC4 in TLS has dropped significantly since the onset of our work, and in part because of it.

[8]Google and Mozilla announcements available at `https://security.googleblog.com/2015/09/disabling-sslv3-and-rc4.html` and `https://blog.mozilla.org/security/2015/09/11/deprecating-the-rc4-cipher/`, respectively.

[9]See `https://notary.icsi.berkeley.edu/`.

Chapter 5

Analysing and Exploiting the Mantin Biases in RC4

Contents

7.1 Introduction 175
7.2 Preliminaries 178
7.3 `draft-21` Analysis 178
7.4 Conclusion 203

This chapter covers attacks against RC4 that exploit the Mantin biases – patterns of the form ABSAB that occur in the RC4 keystream with higher probability than expected for a random sequence (A and B are byte values, and S is an arbitrary byte string of some length G). We develop a statistical framework for exploiting these biases which leads to an algorithm that recovers adjacent pairs of unknown plaintext bytes, under the assumption that the target plaintext bytes are in the neighbourhood of known plaintext bytes, a valid assumption in an attack against TLS. Our analysis enables us to make predictions about the number of ciphertexts needed to reliably recover target plaintext bytes by using results from order statistics. We extend the algorithm to recover longer sequences of plaintext bytes, as would be needed to attack 16-byte cookies protected TLS. We rely on the beam-search and list Viterbi algorithms to achieve this and report on a large range of attack simulations, focussing on a 16-byte target plaintext.

5.1 Introduction

As discussed in the previous chapter, the usage of RC4 in protocols such TLS and WPA has come under heavy attack in recent years – see [10, 68, 76, 113, 115, 116, 148]. The main

idea of these attacks is to exploit known and newly discovered biases in RC4 keystreams to recover fixed plaintexts that are repeatedly encrypted under RC4. Such attacks can be realised against applications using RC4, including TLS, and in particular, lead to serious breaks in application layer protocols using TLS.

Mantin [101] showed that patterns of the form $AB\mathcal{S}AB$ occur in RC4 keystreams with higher probability than expected for a random sequence. Here A and B are byte values and $\mathcal{S}$ is an arbitrary byte string of some length G. Mantin's main result can be stated as follows: Let $G \geq 0$ be a small integer and let Z_r denote the r-th output byte produced by RC4. Under the assumption that the RC4 state is a random permutation at step r, then

$$\Pr\left((Z_r, Z_{r+1}) = (Z_{r+G+2}, Z_{r+G+3})\right) = 2^{-16}\left(1 + \frac{e^{(-4-8G)/256}}{256}\right).$$

Note that for a truly random byte string $Z_r, \ldots, Z_{r+G+3}$, the probability that $(Z_r, Z_{r+1}) = (Z_{r+G+2}, Z_{r+G+3})$ is equal to 2^{-16}. The relative bias is therefore equal to $e^{(-4-8G)/256}/256$, which is about $1/256$ for small G.

Mantin's biases are particularly attractive for use in attacks on RC4 because they are (a) relatively large, (b) numerous, and (c) persistent in RC4 keystreams. Their presence was confirmed experimentally in [101] and [119]. Indeed, they have already been exploited in attacks, [113], and in concurrent work to ours, [148]. In this chapter, we make a systematic study of their use in attacking RC4 in the broadcast setting. Our main contributions can be summarised as follows:

(i) We develop a statistical framework for exploiting the Mantin biases in plaintext recovery attacks for the broadcast setting. We provide such a framework which directly leads to an algorithm that recovers adjacent pairs of unknown plaintext bytes, under the assumption (also used in [113] and [148] and valid in practice for attacks against protocols like TLS) that the target plaintext bytes are in the neighbourhood of *known* plaintext bytes.

(ii) Importantly, and in contrast with [113] and [148], our analysis enables us to make predictions about the numbers of ciphertexts needed to reliably recover target plaintext bytes. More precisely, our attack computes the *likelihood* of each possible target plaintext byte pair, and we are able to compute the distribution of the *rank* of the likelihood of the correct byte pair amongst the likelihoods of all possible pairs as

a function of the number of ciphertexts, N, and the number of known plaintext bytes T[1]. In particular, we can compute the values of (N, T) needed to ensure that the median value of the rank is 1, meaning that the correct plaintext is recovered with high probability. Our approach here is to use results from *order statistics*, a well-established field of statistical investigation that does not appear to have been applied extensively before in cryptanalysis.

(iii) Our framework extends smoothly to make predictions in practically interesting cases where, for example, some additional information is known about the plaintexts, or where known plaintext bytes are present on either side of the unknown bytes.

(iv) We extend the algorithm targeting just two unknown plaintext bytes to the situation where the target is a longer sequence of unknown plaintext bytes. This is a situation of practical interest in attacking session cookies [10] and passwords (as dicussed in the previous chapter) that are protected by RC4 in TLS. We formally justify using as a likelihood estimate for a longer sequence of plaintext bytes the *sum* of the likelihoods of the overlapping pairs of adjacent bytes comprising that longer sequence. As a consequence of our summation formula for likelihoods, we are able to make use of standard methods from the literature, namely beam search and the list Viterbi algorithm [145], to find longer plaintext candidates having high likelihoods. The beam search algorithm is memory-efficient but does not provide any guarantees about the quality of its outputs; the list Viterbi algorithm is memory-intensive but is guaranteed to output a list of candidates having the L highest likelihoods, where L is a parameter of the algorithm. In practical attacks involving cookies and passwords, this type of guarantee is sufficient, since large numbers of candidates can be tested for correctness.

(v) We report on a range of experiments with the beam search and list Viterbi algorithms, evaluating their performance for different parameters, as specified in Table 5.1. For example, using $L = 2^{16}$ in the list Viterbi algorithm, $N = 2^{31}$ ciphertexts, and 130 known plaintext bytes split either side of a 16-byte unknown plaintext, we are able to recover the 16-byte target plaintext with a success rate of about 80%. This is a significant improvement on the preferred attack of [10], which required around 2^{33} – 2^{34} ciphertexts, and our result is broadly comparable with the results obtained

[1] In the previous chapter we used T to denote the password rate limiting factor, i.e., the number of password candidates attempted in our password recovery attacks. In this chapter, as stated, T denotes the number of known plaintext bytes.

in [148].

Parameter	Description
N	The number of available encryptions of the target plaintext.
T	The number of known plaintext bytes.
`algorithm choice`	The dynamic programming algorithm selected, either list Viterbi or beam search.
L	The list size of the list Vitberbi algorithm and the beam width of the beam search algorithm.

Table 5.1: Attack parameters

Related Work. As stated in the previous chapter, AlFardan *et al.* [10] presented two attacks against RC4 in TLS, using single-byte biases in the first and double-byte Fluhrer-McGrew biases from [64] in the second. As in our work, their second attack uses a Viterbi algorithm (though only outputting a single plaintext candidate, so not a *list* Viterbi algorithm). Their second attack requires around 2^{34} ciphertexts to reliably recover a 16-byte target plaintext. Isobe *et al.* [76] also give plaintext recovery attacks for RC4 using single-byte and double-byte biases – their attacks are less effective than those of [10] and are not directly applied to TLS.

Ohigashi *et al.* [113] were the first to use the Mantin biases in plaintext recovery attacks against RC4. They present an attack that targets a single unknown plaintext byte and that uses multiple Mantin biases (for different values of G). Roughly speaking, the unknown plaintext byte is aligned with the second "B" in patterns of the form $AB\mathcal{S}AB$ for varying sizes of $\mathcal{S}$, while the plaintext bytes in the other 3 positions are known; a count is made of the number of times in the RC4 output a string $AB\mathcal{S}AB$ is suggested for each unknown plaintext byte. In the analysis of [113], all biases are "weighted" in the same way, while, intuitively, the weaker the bias, the less reliable the information about plaintext bytes it provides. This overweights the known plaintext bytes that are far from the unknown, target bytes, and leads to a statistically sub-optimal attack. Their attack also recovers multiple plaintext bytes in a byte-by-byte fashion, meaning that if the attack goes awry, then it tends to continue unsuccessfully. This in turn means that the success rate of the attack decreases exponentially with the target plaintext length. Ohigashi *et al.* did not provide any rigorous analysis of their attacks, relying instead on simulations to estimate their effectiveness.

In concurrent work to ours, Vanhoef and Piessens [148] conducted an extensive search for

new biases in RC4 keystreams and settled on using the Mantin biases in combination with the Fluhrer-McGrew biases to target HTTP session cookies protected by TLS. Like us, they use a likelihood-based analysis involving Mantin biases but their analysis is only formalised for single values of G, and they simply take the products of likelihoods for different values of G without further formal statistical justification (though this procedure can be rigorously justified, as our work shows). They also include in their product a likelihood term arising from the Fluhrer-McGrew biases. Given the *ad hoc* nature of their approach, they resort to verification of attack performance via simulations. By contrast, we are able to provide an analytical approach which makes predictions about the distribution of the rank of our likelihood statistic for the correct plaintext bytes.

Vanhoef and Piessens [148] extend their attacks to the recovery of multiple plaintext bytes using a list Viterbi algorithm, though without providing a formal justification, as we do. Their impressive headline result is obtained using a list size $L = 2^{23}$ and recovers a 16-byte plaintext with a 94% success rate using $N = 9 \cdot 2^{27}$ ciphertexts, and roughly 256 known plaintext bytes on either side of the unknown bytes. However, it should be noted that this result applies for a restricted plaintext alphabet, which, as our analysis shows, can significantly boost the performance of attacks. Building on the work of [10], they implement their attack in a real network environment, showing that HTTP session cookies could be recovered in around 75 hours. We do not carry out our attack with this level of realism but instead content ourselves with performing extensive simulations to confirm our theoretical analysis.

5.2 Preliminaries

Much of the preliminary material presented in Chapter 4 (Section 4.2) is relevant here, in particular the material on Bayes' Theorem, the RC4 algorithm, and its usage in TLS. We now provide detail on additional notions and concepts that are necessary for the understanding of the material presented in this chapter. We present an alternate form of Bayes' Theorem, introduce a result from order statistics, discuss the Mantin biases, and expound briefly on the dynamic programming algorithms used in our attacks.

5.2.1 Inferential Form of Bayes' Theorem

In the previous chapter, we introduced the traditional statement of Bayes' Theorem, namely,

$$Pr(A|B) = \frac{Pr(B|A) \cdot Pr(A)}{Pr(B)},$$

where A and B are events, $Pr(B) \neq 0$, and $Pr(A|B)$ and $Pr(B|A)$ are likelihoods (conditional probabilities). The denominator, $Pr(B)$, in the above expression acts as a normalising factor for the posterior probability, $Pr(A|B)$, and for work concerned with finding the most likely outcome of event A given event B, the numerator works just as well as the normalised posterior. Hence, in the context of Bayesian updating, Bayes' Theorem can be expressed as a statement about proportionality of the posterior probability and the Bayes numerator:

$$Pr(A|B) \propto Pr(B|A) \cdot Pr(A).$$

In other words,

$$\text{Posterior} \propto \text{Likelihood} \times \text{Prior}.$$

We use this form of Bayes' Theorem in our attacks in Section 5.3. We note that we implicitly applied this form of Bayes' Theorem in Chapter 4.

5.2.2 Order Statistics

Given random variables $X_1, \ldots, X_k$, the order statistics $X_{(1)}, \ldots, X_{(k)}$ are themselves random variables defined by sorting the realisations of $X_1, \ldots, X_k$ in increasing order. Our work makes use of the following result concerning order statistics [15]:

Result 1. *Suppose $X_1, \ldots, X_k$ are independent standard normal $N(0,1)$ random variables and that Φ denotes the distribution function of a standard normal $N(0,1)$ random variable. Then $\Phi(X_1), \ldots, \Phi(X_k)$ are independent uniform $Uni(0,1)$ random variables and the order statistics $X_{(1)}, \ldots, X_{(k)}$ satisfy*

$$\mathbf{E}\left(\Phi(X_{(j)})\right) = \frac{j}{k+1},$$

where **E** denotes the expected value (in this case of the random variable $\Phi(X_{(j)})$).

5.2.3 The Mantin Biases

As stated in Section 5.1, in 2005 Mantin published a result showing that patterns of the form $AB\mathcal{S}AB$ occur in the RC4 keystream with higher probability than expected of a random sequence [101]. His main result is the following:

$$\Pr\left((Z_r, Z_{r+1}) = (Z_{r+G+2}, Z_{r+G+3})\right) = 2^{-16}\left(1 + \frac{e^{(-4-8G)/256}}{256}\right), \tag{5.1}$$

where A and B are byte values and $\mathcal{S}$ is an arbitrary byte string of some length G. In the published paper containing the work presented in this chapter [40], we present results showing that the distribution of patterns of the form $ABSAB$ in RC4 outputs does not conform exactly with Mantin's analysis. However, the deviations from the predicted behaviour are small, in the sense of affecting the probabilities of only a small proportion of the possible patterns. Hence, in our attacks, we make use of Mantin's result as stated above.

5.2.4 Dynamic Programming Algorithms

Our work employs two dynamic programming algorithms, namely, the list Viterbi algorithm and the beam search algorithm.

List Viterbi Algorithm. The list Viterbi algorithm is described in detail in [145] and generalises the usual Viterbi algorithm. In its general form the algorithm finds the L lowest cost state sequences through a complete trellis of some width W on some state space, given an initial state and a final state and where each state transition in the trellis has an associated cost. The algorithm produces a rank ordered list of the L globally *best* candidates after the trellis search.

Beam Search Algorithm. The beam search algorithm is a heuristic algorithm that explores a graph by expanding the most promising nodes within a limited set of nodes. It is

known as a *best-first* search algorithm, meaning that it orders all partial solutions according to a specified heuristic aimed at determining how close a partial (local) solution is to a complete (global) solution. However, the algorithm only keeps the most promising partial solutions as candidates for the next round of expansion, i.e., it keeps a predetermined number, say L, of them. Thus it is a *greedy algorithm* – making the locally optimal choice at each stage. The beam search algorithm gradually builds a search tree, at each level expanding states (nodes) by generating all successors of the current states, sorting them according to the specified heuristic, and then pruning them down to the L best candidates. The beam search operation makes it memory-efficient, however, it cannot guarantee the L globally best candidates upon completion.

We discuss the applicability of these algorithms to our work in Section 5.4.2.

5.3 Plaintext Recovery using the Mantin Biases

We present a plaintext recovery attack that exploits the Mantin biases. The attack is derived by first posing the plaintext recovery problem as one of maximum likelihood estimation. This enables us to also provide a concise analysis of the expected number of ciphertexts required to successfully recover the correct plaintext (and, more generally, to rank the correct plaintext within the top R candidates, for some chosen value of R).

We operate in the broadcast setting, so the same plaintext is assumed to be encrypted many times under different RC4 keystream segments, in known positions. We target the recovery of two unknown, consecutive plaintext bytes that are adjacent to a group of known plaintext bytes. These attack assumptions (partially known plaintext and the broadcast setting) are fully realistic when mounting attacks that target HTTP cookies when protected by RC4 in TLS (see [10], for instance). In the following section (Section 5.4), we explain how to extend our attack targeting two consecutive plaintext bytes so as to recover longer strings of bytes.

5.3.1 Maximum Likelihood Estimation

We consider the problem of plaintext recovery for various situations arising from RC4 encryption as a maximum likelihood problem. In comparison to the analysis presented in the previous chapter, we adopt a far more rigorous statistical approach to the problem of recovering plaintext bytes, largely out of necessity – harnessing the full effect of the Mantin biases requires such rigour.

Notational Setup. Suppose $p_1, \ldots, p_T, P_{T+1}, P_{T+2}$ are $T+2$ successive plaintext bytes which are to be encrypted a number of times under RC4 using a number of different keystreams. We suppose that the first T plaintext bytes $p_1, \ldots, p_T$ are known plaintext bytes, but that the next two plaintext bytes P_{T+1}, P_{T+2} are unknown and we wish to determine them. (Throughout we use lower-case letters for known quantities, and upper-case for unknown quantities, which can be regarded as random variables.)

We let $c_{i,1}, \ldots, c_{i,T}, c_{i,T+1}, c_{i,T+2}$ denote the $T+2$ successive known ciphertext bytes obtained by encrypting the plaintext bytes $p_1, \ldots, p_T, P_{T+1}, P_{T+2}$ using the i^{th} RC4 keystream $z_{i,1}, \ldots, z_{i,T}, Z_{i,T+1}, Z_{i,T+2}$. Thus we have that

$$z_{i,1} = p_1 \oplus c_{i,1},\ \ldots\ , z_{i,T} = p_T \oplus c_{i,T} \text{ are known keystream bytes}$$
$$\text{and}$$
$$Z_{i,T+1} = P_{T+1} \oplus c_{i,T+1},\ Z_{i,T+2} = P_{T+2} \oplus c_{i,T+2} \text{ are unknown keystream bytes.}$$

Now the Mantin bias can be expressed in the following way: We first define a positive decreasing sequence $\delta_0, \delta_1, \ldots, \delta_{T-2}$ by

$$\delta_G = e^{(-4-8G/256)}/256 = 2^{-8} e^{-\frac{1}{64}} e^{-\frac{G}{32}} \qquad [G = 0, 1, \ldots, T-2].$$

Then, from Mantin's result (Equation 5.1), we have:

$$\Pr\left((Z_{i,T+1}, Z_{i,T+2}) = (z_{i,T-G-1}, z_{i,T-G})\right) \approx 2^{-16}(1 + \delta_G).$$

For byte pairs (a_1, a_2) not in the i^{th} RC4 keystream we have

$$\Pr\left((Z_{i,T+1}, Z_{i,T+2}) = (a_1, a_2)\right) \approx 2^{-16} \quad [(a_1, a_2) \neq (z_{i,1}, z_{i,2}), \ldots (z_{i,T-1}, z_{i,T})].$$

A Likelihood Function. We now calculate the probability mass function for $\theta = (P_{T+1}, P_{T+2})$ for the i^{th} encryption based on the above probabilities. This will lead us to a likelihood function for θ.

By a straightforward calculation, we have:

$$\Pr\left((P_{T+1}, P_{T+2}) = (p', p'')\right) \;=\; \Pr\left((Z_{i,T+1}, Z_{i,T+2}) = (p' \oplus c_{i,T+1}, p'' \oplus c_{i,T+2})\right).$$

This probability is therefore different from 2^{-16} if, for some G, there exists a keystream byte pair $(z_{i,T-G-1}, z_{i,T-G})$ such that

$$\begin{aligned}(p' \oplus c_{i,T+1}, p'' \oplus c_{i,T+2}) &= (Z_{i,T+1}, Z_{i,T+2}) = (z_{i,T-G-1}, z_{i,T-G}) \\ &= (p_{T-G-1} \oplus c_{i,T-G-1}, p_{T-G} \oplus c_{i,T-G}),\end{aligned}$$

that is to say if

$$(p', p'') = (p_{T-G-1} \oplus c_{i,T-G-1} \oplus c_{i,T+1}, p_{T-G} \oplus c_{i,T-G} \oplus c_{i,T+2}).$$

We now let $x_{i,G}$ denote the known 2-byte quantity

$$(p_{T-G-1} \oplus c_{i,T-G-1} \oplus c_{i,T+1}, p_{T-G} \oplus c_{i,T-G} \oplus c_{i,T+2})$$

for the i^{th} RC4 encryption, and we let $x_i = (x_{i,0}, \ldots, x_{i,T-2})^T$ denote the vector of such known 2-byte quantities. If we then let θ denote the value of the unknown plaintext bytes (P_{T+1}, P_{T+2}), then the probability mass function of x_i given the parameter θ is

$$f(x_i; \theta) \approx \begin{cases} 2^{-16}(1 + \delta_G) & x_{i,G} = \theta \quad [G = 0, \ldots, T-2] \\ 2^{-16} & \text{otherwise.} \end{cases}$$

This means that the likelihood function of the parameter $\theta = (P_{T+1}, P_{T+2})$ given the data x_i is given by

$$L(\theta; x_i) \approx \begin{cases} 2^{-16}(1 + \delta_G) & \theta = x_{i,G} \quad [G = 0, \ldots, T-2] \\ 2^{-16} & \text{otherwise.} \end{cases}$$

Here the approximations arise from the fact that, for a given i, the equality $\theta = x_{i,G}$ could hold for multiple values of G, while our analysis ignores this eventuality (which is of low

probability).

We now consider the likelihood function of the parameter $\theta = (P_{T+1}, P_{T+2})$ given N such data vectors $x_1, \ldots, x_N$ derived from known plaintext-ciphertext bytes. If we let

$$S_G(\theta; x) = \#\{x_{i,G} = \theta \mid i = 1, \ldots, N\}$$

be a count of the number of times the G^{th} component of $x_1, \ldots, x_N$ is equal to θ, then the joint likelihood function satisfies

$$L(\theta; x_1, \ldots, x_N) \approx 2^{-16N} \prod_{G=0}^{T-2} (1 + \delta_G)^{S_G(\theta;x)}.$$

Thus if we let x denote the data $x_1, \ldots, x_N$, then the log-likelihood function is given by

$$\begin{aligned} \mathcal{L}(\theta; x) = \log L(\theta; x) &= -16N \log 2 + \sum_{G=0}^{T-2} S_G(\theta; x) \log(1 + \delta_G) \\ &\approx -16N \log 2 + \sum_{G=0}^{T-2} \delta_G S_G(\theta; x) \\ &\approx \delta^T S(\theta; x) - 16N \log 2, \end{aligned}$$

where $\delta = (\delta_0, \ldots, \delta_{T-2})^T$ and $S(\theta; x) = (S_0(\theta; x), \ldots, S_{T-2}(\theta; x))^T$. Thus the value of θ which maximises

$$\delta^T S(\theta; x) \approx \mathcal{L}(\theta; x) + 16N \log 2$$

is essentially the maximum likelihood estimate $\widehat{\theta}$ of the plaintext parameter $\theta = (P_{T+1}, P_{T+2})$ given the known data x.

5.3.2 Plaintext Recovery Attack

The preceding analysis leads immediately to an attack recovering the two unknown bytes $\theta = (P_{T+1}, P_{T+2})$ given access to N ciphertexts: for each value of θ, compute $\delta^T S(\theta; x)$ and output the value of θ which maximises this expression.

The attack can be implemented efficiently by processing the i-th ciphertext as it becomes available, using it to compute the quantities $x_{i,G}$ and updating a $(T-1) \times 2^{16}$ array of integer counters by incrementing the array in positions $(G, x_{i,G})$ for each G between 0 and $T-2$. Once all N ciphertexts are processed in this way, the array contains the counts

$S_G(\theta; x)$ from which the log likelihood of each candidate θ can be computed by taking inner products with the vector δ.

Note too that, since the attack produces log likelihood estimates for each of the 2^{16} candidates θ, it is trivially adapted to output a ranked list of plaintext candidates in order of descending likelihood. This feature is important for our extended attacks in the following section.

This basic attack can be extended in several different ways (some of which can be considered in combination):

1. To the situation where the unknown plaintext bytes are not contiguous with the known plaintext bytes. This merely requires adjusting the above analysis to use Mantin biases for the correct values of G (rather than starting from $G = 0$). Note that because the Mantin biases decrease in strength with increasing G, the attack will be rendered less effective.

2. To the case where known plaintext bytes are located on both sides of the unknown plaintext bytes (possibly in a non-contiguous fashion on one or both sides). Again, this only requires the above analysis to be adjusted to use the correct set of values for G. Using more biases in this way results in a stronger attack.

3. To the case where one of two target plaintext bytes, P_{T+1} say, is already known. This is easily done by considering only the log likelihoods of a reduced set of candidates θ in the attack.

4. To the situation where the plaintext space is constrained in some way, for example, where the bytes of θ are known to be ASCII characters or where base64 encoding is used. Again, this can be done by working with a reduced set of candidates θ.

5.3.3 Distribution of the Maximum Likelihood Statistic and Attack Performance

We now proceed to evaluate the effectiveness of the above basic attack, as a function of the number of available ciphertexts, N, and the number of known plaintext bytes, T.

We let θ^* denote the true value of the plaintext parameter θ. The component $S_G(\theta; x)$ has a binomial distribution, and there are two cases depending on whether or not θ is this true value θ^*, so we have

$$\begin{aligned} S_G(\theta^*; x) &\sim \text{Bin}(N, 2^{-16}(1+\delta_G)) \\ \text{and } S_G(\theta; x) &\sim \text{Bin}(N, 2^{-16}) \qquad\qquad [\theta \neq \theta^*]. \end{aligned}$$

If we write $\mu = N2^{-16}$, then $\mathbf{E}(S_G(\theta^*; x)) = 2^{-16}N(1+\delta_G) = \mu(1+\delta_G)$ and $\mathbf{E}(S_G(\theta; x)) = 2^{-16}N = \mu$ for $\theta \neq \theta^*$, with $\text{Var}(S_G(\theta; x)) \approx 2^{-16}N = \mu$ for all θ (to a very good approximation). For the values of N and hence $\mu = 2^{-16}N$ of interest to us, these binomial random variables are very well-approximated by normal random variables, and we essentially have

$$\begin{aligned} S_G(\theta^*; x) &\sim \text{N}(\mu(1+\delta_G), \mu) \\ \text{and } S_G(\theta; x) &\sim \text{N}(\mu, \mu) \qquad\qquad [\theta \neq \theta^*]. \end{aligned}$$

Thus the vector $S(\theta^*; x) = (S_0(\theta^*; x), \ldots, S_{T-2}(\theta^*; x))^T$ corresponding to the true parameter θ^* and the vectors $S(\theta; x) = (S_0(\theta; x), \ldots, S_{T-2}(\theta; x))^T$ (for $\theta \neq \theta^*$) corresponding to other values of the plaintext parameter have a multivariate normal distribution. Furthermore, it is reasonable to assume that the components of these vectors are independent, so we have

$$\begin{aligned} S(\theta^*; x) &\sim \text{N}_{T-1}(\mu(\mathbf{1}+\delta), \mu I_{T-1}) \\ \text{and } S(\theta; x) &\sim \text{N}_{T-1}(\mu\mathbf{1}, \mu I_{T-1}) \qquad\qquad [\theta \neq \theta^*]. \end{aligned}$$

The maximum likelihood statistic is essentially determined by the distributions of $\delta^T S(\theta^*; x)$ and $\delta^T S(\theta; x)$ (for $\theta \neq \theta^*$). However, these are just rank-1 linear mappings of multivariate normal random variables and so have univariate normal distributions given by

$$\begin{aligned} \delta^T S(\theta^*; x) &\sim \text{N}(\mu(\delta^T\mathbf{1} + |\delta|^2), \mu|\delta|^2) \\ \text{and } \delta^T S(\theta; x) &\sim \text{N}(\mu\delta^T\mathbf{1}, \mu|\delta|^2) \qquad\qquad [\theta \neq \theta^*]. \end{aligned}$$

The above distributions suggest that it is convenient to consider the function

$$J(\theta; x) = \frac{\delta^T S(\theta; x) - \mu\mathbf{1}^T\delta}{\mu^{\frac{1}{2}}|\delta|} = \mu^{-\frac{1}{2}}|\delta|^{-1}\left(\delta^T S(\theta; x)\right) - \mu^{\frac{1}{2}}|\delta|^{-\frac{1}{2}}(\mathbf{1}^T\delta)$$

on the parameter space. It is clear that $J(\theta; x)$ is a very good approximation to an affine

transformation of the log-likelihood function, so the value of θ which maximises $J(\theta; x)$ is essentially the maximum likelihood estimate $\widehat{\theta}$ of the plaintext parameter $\theta = (P_{T+1}, P_{T+2})$ given the known data x.

We note that $J(\theta; x)$ has a univariate normal distribution with unit variance in both cases as we have

$$J(\theta^*; x) \sim \mathrm{N}\left(\mu^{\frac{1}{2}}|\delta|, 1\right) \text{ and } J(\theta; x) \sim \mathrm{N}(0, 1) \text{ for } \theta \neq \theta^*.$$

Furthermore, we may essentially regard all of these random variables $J(\theta; x)$ as independent since the random variables $S_G(\theta; x)$ are very close to being independent.

The function $J(\theta; x)$ can be thought of as a "variance-stabilised" form of log-likelihood function $\mathcal{L}(\theta; x)$ of the plaintext parameter θ. Furthermore, the squared length of the vector δ can be calculated as

$$|\delta|^2 = \sum_{G=0}^{T-2} \delta_G^2 = \frac{e^{\frac{1}{32}} - e^{\frac{1}{32}(3-2T)}}{2^{16}(e^{\frac{1}{16}} - 1)}.$$

This means, for instance, that $|\delta| \approx 0.00385$ for $T = 2$ and $|\delta| \approx 0.00930$ for $T = 8$, with $|\delta| \approx 0.0156$ for large T.

Performance of Plaintext Ranking in the Basic Attack. With the above reformulation, finding the maximum likelihood estimate $\widehat{\theta}$ by maximising the function $J(\theta; x)$ can now be seen as essentially comparing a realisation of a normal $\mathrm{N}(\mu^{\frac{1}{2}}|\delta|, 1)$ random variable (corresponding to $J(\theta^*; x)$) with a set $\mathcal{R} = \{J(\theta; x) | \theta \neq \theta^*\}$ of realisations of $2^{16} - 1 = 65535$ independent standard normal $\mathrm{N}(0, 1)$ random variables. Thus the maximum likelihood estimate $\widehat{\theta}$ gives the true plaintext parameter θ^* if a realisation of an $\mathrm{N}(\mu^{\frac{1}{2}}|\delta|, 1)$ random variable exceeds the maximum of the realisations of $2^{16} - 1$ independent standard normal random variables.

This enables the probability that the maximum likelihood estimate is correct (and the basic attack succeeds) to be evaluated as a function of N and T. However, we are able to go further and consider the rank of the correct plaintext θ^* in the ordered list of values $J(\theta; x)$ (from highest to lowest) as a function of N and T, that is to evaluate the performance of the ranking version of the plaintext recovery attack. Such an evaluation makes use of the result concerning order statistics mentioned in Section 5.2, namely:

Result 1. *Suppose $X_1, \ldots, X_k$ are independent standard normal $N(0,1)$ random variables and that Φ denotes the distribution function of a standard normal $N(0,1)$ random variable. Then $\Phi(X_1), \ldots, \Phi(X_k)$ are independent uniform $Uni(0,1)$ random variables and the order statistics $X_{(1)}, \ldots, X_{(k)}$ satisfy*

$$\mathbf{E}\left(\Phi(X_{(j)})\right) = \frac{j}{k+1},$$

where $\mathbf{E}$ denotes the expected value of the random variable $\Phi(X_{(j)})$.

It follows that $\Phi(z)$ is an accurate representation on a linear uniform scale between 0 and 1 of the position of a value z within $X_{(1)}, \ldots, X_{(k)}$. Thus the random variable giving the position (from highest to lowest) or "rank" of $J(\theta^*; x)$ with respect to the set $\mathcal{R}$, and hence the rank of θ^*, is given accurately by rounding the random variable

$$\mathbf{Rk}(\theta^*) = 2^{16}(1 - \Phi(J(\theta^*; x)))$$

to the nearest integer.

The distribution function $F_{\mathbf{Rk}(\theta^*)}$ of this (unrounded) rank $\mathbf{Rk}(\theta^*)$ of θ^* is given by

$$\begin{aligned} F_{\mathbf{Rk}(\theta^*)}(z) &= \Pr\left(\mathbf{Rk}(\theta^*) \leq z\right) = \Pr\left(2^{16}(1 - \Phi(J(\theta^*; x))) \leq z\right) \\ &= \Pr\left(J(\theta^*; x) \geq \Phi^{-1}\left(1 - 2^{-16}z\right)\right) = 1 - F_*\left(\Phi^{-1}\left(1 - 2^{-16}z\right)\right), \end{aligned}$$

where F_* is the distribution function of $J(\theta^*; x)$, that is to say of an $\mathrm{N}\left(\mu^{\frac{1}{2}}|\delta|, 1\right)$ distribution.

Figure 5.1 shows the cumulative distribution function of the rank $\mathbf{Rk}(\theta^*)$ for different numbers of ciphertexts, N, for the specific value $T = 2^6$. It can be seen that as N approaches 2^{32}, it becomes highly likely that the rank of θ^* is rather small. On the other hand, when N drops below 2^{28}, the attack does not have much advantage over random guessing (which would produce a diagonal line on the cumulative distribution plot).

The median of $\mathbf{Rk}(\theta^*)$, which is very close to the mean of $\mathbf{Rk}(\theta^*)$, is the value of z satisfying $F_{\mathbf{Rk}(\theta^*)}(z) = \frac{1}{2}$, that is to say

$$\begin{aligned} \mathrm{Median}\left(\mathbf{Rk}(\theta^*)\right) &= 2^{16}\left(1 - \Phi\left(F_*^{-1}\left(\tfrac{1}{2}\right)\right)\right) = 2^{16}\left(1 - \Phi\left(\mu^{\frac{1}{2}}|\delta|\right)\right) \\ &= 2^{16}\Phi\left(-2^{-8}N^{\frac{1}{2}}|\delta|\right). \end{aligned}$$

N	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}	2^{35}	2^{36}	2^{37}
$T = 2^1$	28236	26390	23838	20387	15920	10628	5353	1596	174	3	1
$T = 2^3$	22081	18078	13105	7664	3024	566	25	1	1	1	1
$T = 2^6$	15735	10423	5176	1502	155	2	1	1	1	1	1

Table 5.2: Median rank of maximum likelihood estimate of plaintext parameter

Table 5.2 shows some median rankings for the value of $J(\theta^*; x)$ within the set of all such 2^{16} values of $J(\theta; x)$. A median rank of "1" indicates that the maximum likelihood estimate $\widehat{\theta}$ gives the true plaintext parameter θ^* with high probability.

Performance of Plaintext Ranking in Variant Attacks. The above analysis is easily extended to evaluate the performance of the variant attacks described in Section 5.3.2.

For variant 1, in which the unknown plaintext bytes are not contiguous with the known plaintext bytes, we need only replace the value of $|\delta|$ with the appropriate value computed from the biases actually used in the attack. For variant 2, where known plaintext bytes are located on both sides of the unknown plaintext bytes, the same is true, but this time δ increases; the analysis is otherwise identical. For example, $|\delta|^2$ doubles when we use an additional T known plaintext bytes $p_{T+3}, \ldots, p_{2T+2}$ in concert with $p_1, \ldots, p_T$. Recalling that $J(\theta^*; x)$ has a $\mathrm{N}\left(\mu^{\frac{1}{2}}|\delta|, 1\right)$ distribution with $\mu = 2^{-16}N$, it can be seen that the effect of doubling $|\delta|^2$ by using "double-sided" biases in this way is the same as that of doubling N in the attack; put another way, using double-sided biases reduces the number of ciphertexts

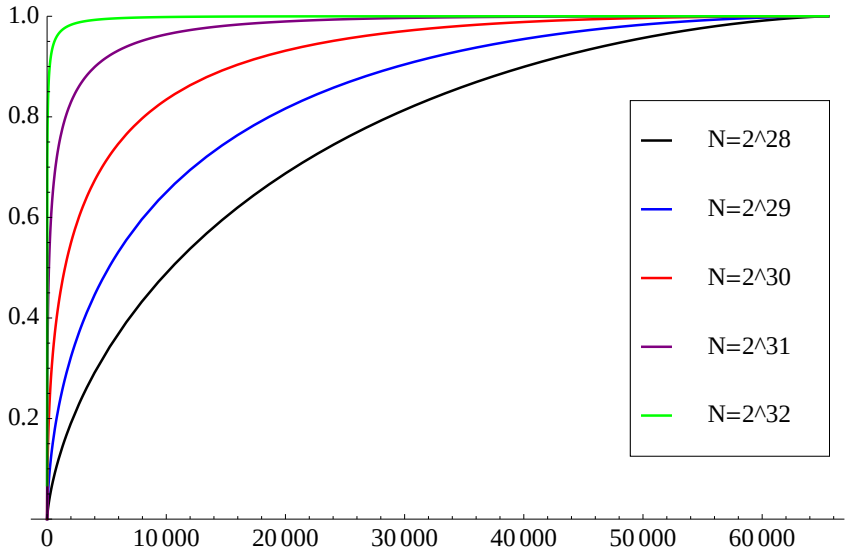


Figure 5.1: Cumulative distribution function of the rank $\mathbf{Rk}(\theta^*)$ for different numbers of ciphertexts, N ($T = 2^6$).

needed to obtain a given median ranking for the value of $J(\theta^*; x)$ by a factor of 2.

Variants 3 and 4 both concern the case where the plaintext space for the pair (P_T, P_{T+1}) is reduced from a set of 2^{16} candidates to some smaller set of candidates, $\mathcal{C}$ say. For example, in variant 3, where one of the plaintext bytes is known, $|\mathcal{C}| = 2^8$. This means that our fundamental statistical problem becomes one of distinguishing a realisation of a normal $\mathrm{N}(\mu^{\frac{1}{2}}|\delta|, 1)$ random variable (corresponding to $J(\theta^*; x)$) from a now smaller set $\mathcal{R} = \{J(\theta; x) | \theta \in \mathcal{C} \setminus \theta^*\}$ of $|\mathcal{C}| - 1$ realisations of independent standard normal $\mathrm{N}(0, 1)$ random variables. Our previous analysis goes through as above, except that we simply replace 2^{16} by $|\mathcal{C}|$ where appropriate, resulting in

$$\text{Median}\left(\mathbf{Rk}(\theta^*)\right) = |\mathcal{C}| \cdot \Phi\left(-2^{-8} N^{\frac{1}{2}} |\delta|\right).$$

The effect of this is to divide all the entries in Table 5.2 by $2^{16}/|\mathcal{C}|$. For example, in variant 3 where $|\mathcal{C}| = 2^8$, we would expect a median rank of roughly 6 with $N = 2^{30}$ ciphertexts and $T = 2^6$.

Note that these two effects are cumulative. For example, using double-sided biases and assuming one byte of plaintext from the pair (P_{T+1}, P_{T+2}) is known has the effect of both reducing N by a factor of 2 *and* dividing the median rank by 2^8. Then, for example, with only $N = 2^{29}$ ciphertexts and $T = 2^6$ we would expect $J(\theta^*; x)$ to have a median rank of about 6, meaning that the correct plaintext θ^* can be expected to have a high ranking.

Experimental Validation. We carried out an experimental validation of our statistical analysis, performing experiments with $T = 2^6$ for different numbers of ciphertexts, N, and computing the cumulative distribution function of the rank $\mathbf{Rk}(\theta^*)$. The results are shown in Figure 5.2 for $N = 2^{28}$, 2^{29} and 2^{30}. Good agreement can be seen between the experimental results and the predictions made by our statistical analysis, with the experiments slightly outperforming the theoretical predictions in each case.

5.3.4 Incorporating Prior Information about Plaintext Bytes

Prior information about the unknown plaintext bytes is frequently available and can be exploited (as was done in the previous chapter) to improve attacks.

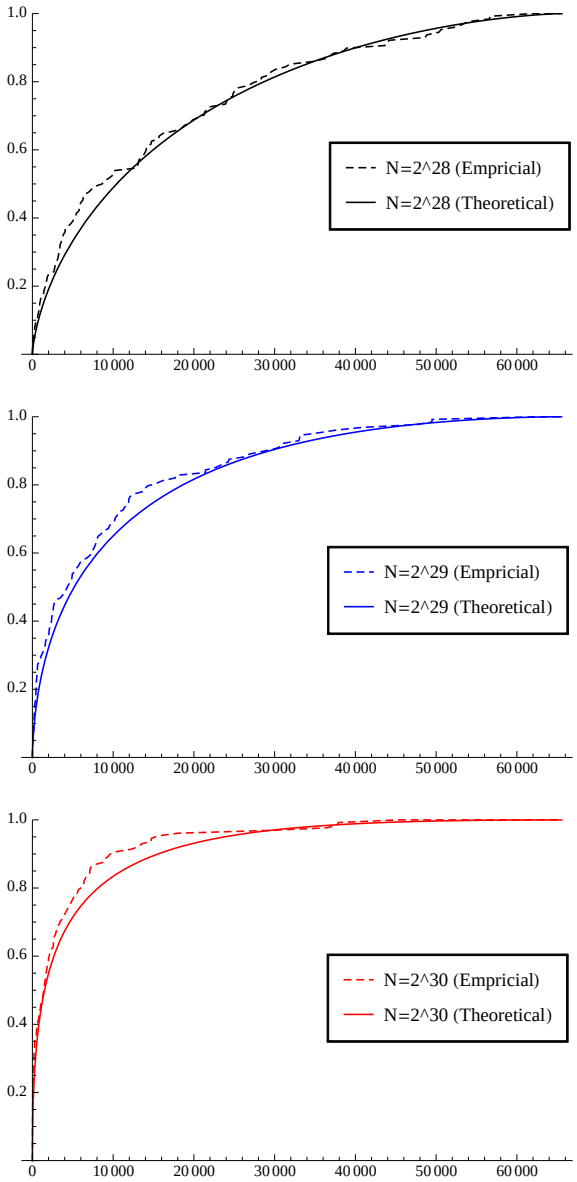


Figure 5.2: Cumulative distribution function of the rank $\mathbf{Rk}(\theta^*)$ for different numbers of ciphertexts, N ($T = 2^6$): $N = 2^{28}$ (top), $N = 2^{29}$ (middle), $N = 2^{30}$ (bottom).

Prior information in our setting can be incorporated using the inferential form of Bayes Theorem, which can be loosely expressed as

$$\text{Posterior} \;\propto\; \text{Likelihood} \;\times\; \text{Prior},$$

or equivalently in its logarithmic form as

$$\text{Log-Posterior} \;=\; \text{Log-Likelihood} \;+\; \text{Log-Prior} \;+\; \text{Constant}.$$

If we let $\pi(\theta)$ denote the prior probability of the plaintext parameter $\theta = (P_{T+1}, P_{T+2})$

and $\pi(\theta; x)$ the posterior probability of the parameter θ given the data x, then we have

$$\begin{aligned} \log \pi(\theta; x) &= \mathcal{L}(\theta; x) + \log \pi(\theta) + \text{ Constant} \\ &\approx \delta^T S(\theta; x) + \log \pi(\theta) + \text{ Constant}. \end{aligned}$$

This suggests that for purposes such as posterior plaintext ranking, we consider an adaptation of $J(\theta; x)$ given by

$$J_\pi(\theta; x) = \frac{\delta^T S(\theta; x) + \log \pi(\theta) - \mu \mathbf{1}^T \delta}{\mu^{\frac{1}{2}} |\delta|} = J(\theta; x) + \frac{\log \pi(\theta)}{\mu^{\frac{1}{2}} |\delta|}.$$

We note that $J_\pi(\theta; x)$ has a univariate normal distribution with unit variance as we have

$$\begin{aligned} J_\pi(\theta^*; x) &\sim \mathrm{N}\left(\mu^{\frac{1}{2}} |\delta| + \frac{\log \pi(\theta^*)}{\mu^{\frac{1}{2}} |\delta|}, 1\right) \\ \text{and} \quad J_\pi(\theta; x) &\sim \mathrm{N}\left(\frac{\log \pi(\theta)}{\mu^{\frac{1}{2}} |\delta|}, 1\right) \qquad \text{for } \theta \neq \theta^*. \end{aligned}$$

It is clear that when N or equivalently $\mu = 2^{-16} N$ is small, that is roughly speaking when $\mu |\delta|^2 << |\log \pi(\theta)|$, the mean value of the posterior scoring function is given by $\mathbf{E}\left(J_\pi(\theta; x)\right) \approx \mu^{-\frac{1}{2}} |\delta|^{-1} \log \pi(\theta)$ for both $\theta = \theta^*$ and $\theta \neq \theta^*$. Thus when N or μ is small, the posterior scoring function essentially orders the plaintext parameters π according to the prior distribution π; analysis of the available ciphertexts does not yield enough evidence to "overturn" the evidence given by the prior distribution. By contrast when N or μ is large, that is roughly speaking when $\mu |\delta|^2 >> |\log \pi(\theta)|$, then $\mathbf{E}\left(J_\pi(\theta^*; x)\right) \approx \mu^{\frac{1}{2}} |\delta|$ and $\mathbf{E}\left(J_\pi(\theta; x)\right) \approx 0$ for $\theta \neq \theta^*$. In this situation, the evidence of the experiment "overwhelms" the evidence given by the prior distribution, and we are essentially considering the previous scenario.

The interesting situation is therefore when $\mu |\delta|^2$ and $|\log \pi(\theta)|$ are of roughly comparable size. We consider how much data is needed to "overturn" an ordering of plaintext parameters according to their prior probabilities. In this situation, the scoring function for the plaintext parameter has means given by

$$\mathbf{E}\left(J_\pi(\theta^*; x)\right) = \mu^{\frac{1}{2}} |\delta| + \frac{\log \pi(\theta^*)}{\mu^{\frac{1}{2}} |\delta|} \quad \text{and} \quad \mathbf{E}\left(J_\pi(\theta; x)\right) = \frac{\log \pi(\theta)}{\mu^{\frac{1}{2}} |\delta|} \text{ for } \theta \neq \theta^*.$$

Thus the scoring function for the correct plaintext parameter θ^* is expected to exceed that

of the plaintext parameter θ when $\mathbf{E}\left(J_\pi(\theta^*; x)\right) > \mathbf{E}\left(J_\pi(\theta; x)\right)$, that is to say when

$$\mu > \frac{1}{|\delta|^2} \log \frac{\pi(\theta)}{\pi(\theta^*)} \quad \text{or equivalently when } N > \frac{2^{16}}{|\delta|^2} \log \frac{\pi(\theta)}{\pi(\theta^*)}$$

The interesting case is obviously when $\pi(\theta) > \pi(\theta^*)$, that is to say when θ is *a priori* a more likely plaintext parameter than θ^*. In this case, the above expression indicates how many samples are likely to be required to be able to place an *a posteriori* rank θ^* above that for θ. Clearly, the answer depends on the specifics of the distribution π.

5.4 Recovering Multiple Plaintext Bytes

We now extend the attacks and analysis presented in the previous section to consider the situation where the target plaintext extends over multiple bytes. As in previous [10, 68, 76, 113, 115, 116] and concurrent [148] works, this is important in building practical attacks targeting HTTP cookies and passwords. We are particularly interested in attack algorithms that output lists of candidates rather than single candidates, since in many practical situations, many suggested candidates can be tried one after another, as was first suggested in [10]. Throughout, we let W denote the byte-length of the target plaintext.

This problem was already addressed in [10] and [113] for attacks exploiting Fluhrer-McGrew and Mantin biases, respectively. Although not explicit in [10], the algorithm used there is a Viterbi algorithm and is guaranteed to output the best plaintext candidate on W bytes according to an approximate log likelihood metric; roughly 2^{33} – 2^{34} ciphertexts were needed to recover a 16-byte plaintext with high success rate. The algorithm in [113] proceeds on a byte-by-byte basis and the success probability of it recovering the correct plaintext is the product of success rates for single bytes. This, unfortunately, means that the success rate drops rapidly as a function of W. For example, with $N = 2^{32}$ ciphertexts and $T = 66$ known plaintext bytes, the algorithm of [113] achieves a success rate of 0.7656 for a single byte, but this would be reduced to $(0.7656)^{16} = 0.014$ for $W = 16$ bytes.

5.4.1 A Likelihood Analysis for Multiple Plaintext Bytes

As before, we assume plaintext bytes $p_1, \ldots, p_T$ are known. Our task now is to recover the W unknown bytes $\theta = (P_{T+1}, \ldots, P_{T+W})$. We let θ_w denote (P_{T+w}, P_{T+w+1}) for $1 \leq w \leq W-1$. Using the methods of Section 5.3, we can form $W-1$ ranked lists of values for $\mathcal{L}(\theta_w; x)$, where as before x denotes the collection of N data vectors $x_1, \ldots, x_N$ derived from known plaintext-ciphertext bytes. Note here that when $w \geq 2$, these log-likelihoods will be computed using progressively weaker Mantin biases with $G \geq 1$.

To evaluate the overall log-likelihood $\mathcal{L}(\theta; x)$, we will replace this quantity with the sum:

$$\sum_{w=1}^{W-1} \mathcal{L}(\theta_w; x) \tag{5.2}$$

of log-likelihoods for the byte pairs θ_i.

This replacement is formally justified as follows. Consider the probability mass function of a data vector x_i given the unknown byte pairs $\theta = (\theta_1, \ldots, \theta_{W-1})$. This can be approximated as

$$f(x_i; \theta_1, \ldots, \theta_{W-1}) \approx \begin{cases} 2^{-16}(1 + \delta_{G+w-1}) & x_{i,G} = \theta_w \\ 2^{-16} & \text{otherwise.} \end{cases}$$

Here, the nature of the approximation is similar to that made in our analysis in Section 5.3: it assumes that at most one low probability event $x_{i,G} = \theta_w$ occurs for each i.

However, the probability mass function of a data vector x_i given a single unknown byte pair θ_w can be approximated as

$$f(x_i; \theta_w) \approx \begin{cases} 2^{-16}(1 + \delta_{G+w-1}) & x_{i,G} = \theta_w \\ 2^{-16} & \text{otherwise,} \end{cases}$$

so the product of all such probability mass functions can be approximated as

$$\prod_{w=1}^{W-1} f(x_i; \theta_w) \approx \begin{cases} 2^{-16(W-2)}\ 2^{-16}(1 + \delta_{G+w-1}) & x_{i,G} = \theta_w \\ 2^{-16(W-2)}\ 2^{-16} & \text{otherwise.} \end{cases}$$

This enables us to give an approximate proportionality relationship between the the probability mass function of a data vector x_i given the unknown byte pairs $\theta = (\theta_1, \ldots, \theta_{W-1})$

and the probability mass functions of a data vector x_i given *single* unknown byte pairs θ_w since we now see that

$$f(x_i; \theta_1, \ldots, \theta_{W-1}) \propto \prod_{w=1}^{W-1} f(x_i; \theta_w).$$

This can be re-formulated in terms of likelihood functions as

$$L(\theta; x_i) = L(\theta_1, \ldots, \theta_{W-1}; x_i) \propto \prod_{w=1}^{W-1} L(\theta_w; x_i).$$

The likelihood function of the byte pairs $\theta = (\theta_1, \ldots, \theta_{W-1})$ given all the data vectors $x = (x_1, \ldots, x_N)$ is therefore proportional (to a good approximation) to a product of individual likelihood functions, that is to say

$$L(\theta; x) \propto \prod_{i=1}^{N} \left(\prod_{w=1}^{W-1} L(\theta_w; x_i) \right) = \prod_{w=1}^{W-1} \left(\prod_{i=1}^{N} L(\theta_w; x_i) \right) = \prod_{w=1}^{W-1} L(\theta_w; x),$$

which can be expressed in log-likelihood terms (for some constant C) as

$$\mathcal{L}(\theta; x) \approx C + \sum_{w=1}^{W-1} \mathcal{L}(\theta_w; x).$$

Thus maximising the overall log-likelihood $\mathcal{L}(\theta; x)$ can be achieved (to a good approximation) by maximising the sum $\sum_{w=1}^{W-1} \mathcal{L}(\theta_w; x)$ of individual log-likelihoods.

5.4.2 Algorithms for Recovering Multiple Plaintext Bytes

It follows from the above analysis that, to find high log-likelihood candidates for θ, we need to find sequences of overlapping byte pairs θ_w for which the sums in (5.2) are large, given the $W - 1$ lists $\mathcal{L}(\theta_w; x)$. This is a classic problem in dynamic programming that can be solved by a number of different approaches. We consider two such standard approaches:

List Viterbi. Recall that in its general form the list Viterbi algorithm finds the L lowest cost state sequences through a complete trellis of some width W on some state space, given an initial state and a final state and where each state transition in the trellis has an associated cost. The algorithm is easily adapted to the problem at hand by setting the edge weights to be the log-likelihood values $\mathcal{L}(\theta_w; x)$ and interpreting the states as

byte values.[2] The algorithm is relatively memory intensive and somewhat slow, requiring roughly $256 \cdot W$ times as much storage as the beam search algorithm to return a final list of L candidates. However, the algorithm has the advantage that it guarantees to return the L *best* plaintext candidates on W bytes, that is the top L candidates according to the metric represented by (5.2).

Beam Search. In the beam search algorithm, we generate a list of L candidates on j positions $T+1, \ldots, T+j$, each candidate being accompanied by a partial sum $\sum_{w=1}^{j-1} \mathcal{L}(\theta_w; x)$. We then expand the list to include all $256 \cdot L$ candidates that are 1-byte extensions of candidates on the list, computing a new sum $\sum_{w=1}^{j} \mathcal{L}(\theta_w; x)$ for each candidate by adding a term $\mathcal{L}(\theta_w; x)$. We then prune the list back to L candidates again, by keeping just the top L candidates, but now on $w+1$ positions. The process is initialised using the top L values for $\mathcal{L}(\theta_1; x)$ on the first two unknown plaintext bytes. The process is finalised when $w = W - 1$, and the list need not be pruned at the final step, though we do so in our implementation to provide a fair comparison with the list Viterbi algorithm. The algorithm is deemed successful if the correct plaintext $(P_{T+1}, \ldots, P_{T+W})$ appears on the final pruned list of L candidates. In a further enhancement, we may assume the first and last byte of the plaintext are known, and force the candidate plaintexts to begin and end with those known bytes. The beam search algorithm is fast and memory-efficient, but does not provide any guarantees about the quality of its outputs (that is to say, we do not know if it will successfully include the highest log-likelihood plaintext on its output list).

Note that both algorithms extend smoothly to the double-sided case where some plaintext bytes are known on both sides of the W unknown bytes; the only modification is to the computation of the log likelihoods $\mathcal{L}(\theta_w; x)$ that are input to the algorithms. Again we will be forced to use Mantin biases starting with non-zero values of G in computing the values $\mathcal{L}(\theta_w; x)$, because of the presence of a run of unknown plaintext bytes before reaching the known plaintext bytes. Both algorithms also generalise easily to the case where the plaintext space is constrained in some way, simply by considering only restricted sets of plaintext bytes when extending candidates (in beam search) or traversing the trellis (in the list Viterbi case).

[2] Several additional notational and conceptual changes are needed compared to the original description in [145]. In particular, the initialisation process described in [145] contains a small error, and we wish to maximise rather than minimise the cost of state sequences. The basic algorithm also requires the first and last bytes of plaintext, P_{T+1} and P_{T+W} to be known.

5.5 Simulation Results

We performed experiments with the beam search and list Viterbi algorithms, for a variety of attack parameters.

5.5.1 Methodology

We focus on recovering 16 unknown plaintext bytes, a length typical of HTTP cookies, and on attacks using single-sided and double-sided biases with, respectively, $T = 66$ and 130 known plaintext bytes – in the case of List Viterbi, we require a trellis of width 18 as the first and last plaintext bytes need to be known, and for beam search we assume known plaintext bytes, one on either side of the 16 unknown target plaintext bytes. We are most interested in how the attack performance varies with N, the number of available ciphertexts, and L, the pruned list size/output list size in the two algorithms. Further experiments to explore how performance changes with T and W, and for the case of a constrained plaintext space, would be of interest, but we did not have the computing resources available to perform these. Notably, target plaintexts such as cookies often have symbols coming from a much reduced plaintext space, a fact exploited in [148] to reduce their attack's ciphertext requirements.

Our experiments ran in two phases: in phase 1, we generated 2^{12} keystream groups, each group containing $N = 2^{27}$ blocks of keystream bytes. On the fly, for each group, we computed and stored the single-sided and double-sided log-likelihood measures $\mathcal{L}(\theta_w; x)$ for each of the 2^{16} possible values of θ_w for each of 17 overlapping pairs of positions, yielding log-likelihood information for 18 consecutive unknown plaintext bytes. Then, in phase 2, we collated the measures coming from different groups to create measures for groups corresponding to progressively larger sets of blocks. This enabled us to carry out 128 plaintext recovery attacks on up to $N = 2^{32}$ ciphertexts each, using our beam search and list-Viterbi algorithms. We ran each of these algorithms with $L = 2^{16}$ and computed the success rate across different values of N (typical values of N are $n \cdot 2^{27}$ where $n \in \{8, 10, 11, 12, 13, 14, 15, 16, 18, 20, 24, 28, 32\}$).

All computations were performed on the Google Compute Engine (GCE), and we optimised various parameters internal to our code for this platform. Each list Viterbi execution with

$L = 2^{16}$ on a trellis of width 18 took around 2 hours on a single GCE core; by contrast, the execution of the beam search algorithm completed in a only a couple of minutes for the same parameter L. This favourable running time inspired us to conduct further beam search experiments for higher values of L. For $L = 2^{17}$ each beam search experiment took about 20 minutes, and for $L = 2^{18}$, the running time was roughly 2.5 hours per experiment. We attribute this unfortunate scaling in the running time to an increasing number of cache misses as L grows. In total we used around 6,200 GCE core-hours of computation for the experiments.

5.5.2 Results

We present our results for the attack simulations starting with those for the list Viterbi algorithm. We then discuss a number of results for the beam search algorithm and conclude this section with a comparison of the two algorithms.

List Viterbi. Figure 5.3 shows how the success rate varies with N, the number of ciphertexts available, for the list Viterbi algorithm with double-sided biases (130 known plaintext bytes split either side of 16 unknown bytes, with 2 of the known bytes being used in the list Viterbi algorithm and the remaining 128 being used for computing log likelihoods). Each curve represents a different value of L. It can be seen that, for fixed N, the success rate increases steadily with L and that a threshold phenomenon is observable, where above roughly 2^{30} ciphertexts, the success rate takes off rapidly. For example, with $N = 2^{31}$ we see a success rate 86% for $L = 2^{16}$. We are confident that the success rate would continue to improve with increasing L and with a larger number of known plaintext bytes, bringing our results into contention with those of [148] (which used 256 known bytes instead of our 130, the significantly larger $L = 2^{23}$ in the list Viterbi algorithm, and an undisclosed reduced plaintext space to achieve a success rate of 94% for recovering a 16-byte plaintext with $9 \cdot 2^{27}$ ciphertexts, a little over 2^{30} ciphertexts).

Figure 5.4 compares the performance of the single-sided and double-sided version of the attacks. Not surprisingly, the use of double-sided biases significantly improves the attack performance.

Beam Search. Unless otherwise stated, we use the enhancement of assuming the bytes

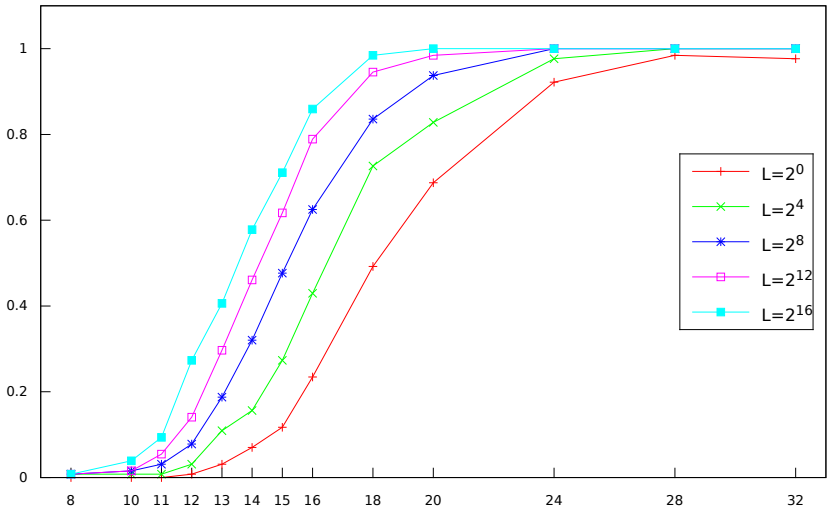


Figure 5.3: Success rate of list Viterbi algorithm in recovering a 16-byte unknown plaintext for different numbers of ciphertexts, N and different list sizes L, using double-sided biases, and 130 known plaintext bytes. The x-axis shows the number of ciphertexts divided by 2^{27}.

directly adjacent to the 16 target plaintext bytes to be known, and we force our respective 18-byte candidates to start and end with these bytes. Figure 5.5 shows the performance of the beam search algorithm for varying numbers of ciphertexts, N, and for $L = 2^{16}, 2^{17}$ and 2^{18}. As expected, we do see an improvement in success rates as L grows. For example, with $N = 2^{31}$ we see a success rate increase of 3% in going from $L = 2^{16}$ to $L = 2^{18}$. Significant gains, however, are likely to be made with larger values of L, say $L = 2^{20}$.

In order to determine the extent to which assuming adjacent bytes to be known improves attack performance, we ran the following two sets of experiments: We assumed the first byte adjacent to the 16 target plaintext bytes to be known and used the single-sided biases to recover 17-byte candidates (in other words, $W = 17$ with P_{T+1} known). We then used the single-sided biases to recover 16 unknown target bytes ($W = 16$ and P_{T+1} unknown).[3] Figure 5.6 shows that there is a small advantage to using this enhancement. For instance, with $N = 2^{32}$ we see the success rate increase by 3%.

In a further enhancement, we did not prune the list of plaintext candidates in the final stage of the beam search algorithm. In other words, we retained $2^8 \cdot L$ candidates in the last step of the process and declared success if the correct plaintext appeared on this larger

[3]Using the generated double-sided biases with $W = 18$ for the recovery of 16-byte plaintexts would have resulted in us not being able to use some of the strongest biases for plaintext recovery; targeting bytes P_{T+2} to P_{T+17} would mean not using biases when $G = 0$, and targeting bytes P_{T+1} to P_{T+16} would mean not using biases for each G between 0 and 2 in the recovery of P_{T+15} and P_{T+16}.

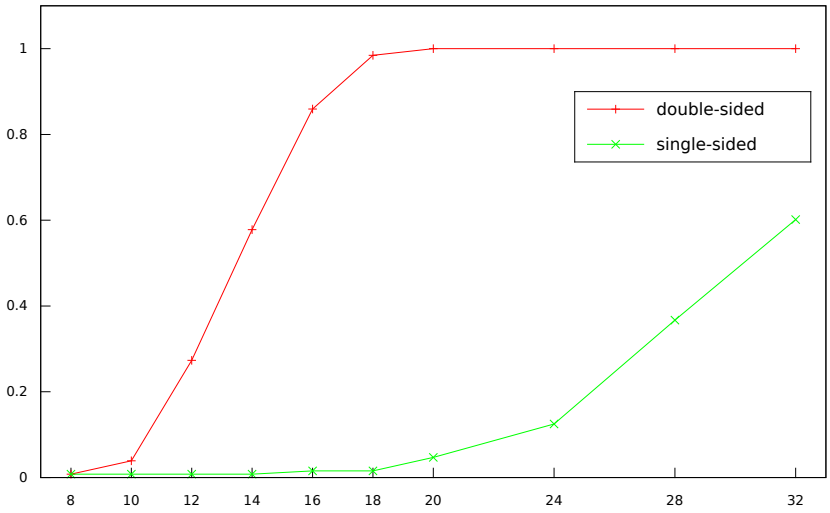


Figure 5.4: Success rate of list Viterbi algorithm in recovering a 16-byte unknown plaintext for different numbers of ciphertexts, using single-sided and double-sided biases (with 66 and 130 known plaintext bytes, respectively) and $L = 2^{16}$. The x-axis shows the number of ciphertexts divided by 2^{27}.

list of candidates. Figure 5.7 shows the performance of the beam search algorithm using this enhancement in comparison to the case in which this enhancement is not used. We see a very slight improvement in attack performance as a result of this enhancement.

Comparing List Viterbi and Beam Search. Figure 5.8 compares the performance of list Viterbi and beam search algorithms with L set to 2^{16} in both cases. It can be seen that the beam search algorithm performs very well, close to the optimal attack that is represented by list Viterbi. It may make for an attractive alternative in practice, especially for such large values of L where the memory consumption of the list Viterbi algorithm becomes prohibitive.

5.6 Conclusion

Exploiting the Mantin biases that are present in the RC4 keystream, we have developed a statistical framework that enables us to make accurate predictions about the performance of plaintext recovery attacks targeting adjacent pairs of plaintext bytes. A particular novelty is the introduction of order statistics, enabling the expected rank of the true plaintext amongst all possible candidates to be computed. We extended our 2-byte attacks to the situation of multiple unknown plaintext bytes, and provided an experimental evaluation of

two different attacks for this setting, using the list Viterbi and beam search algorithms, respectively.

Our attacks target unknown bytes of plaintext that are located close to sequences of known plaintext bytes, a situation that is commonplace in practice when RC4 is used in TLS. Our experiments have shown that we can successfully recover 16-byte plaintexts, such as TLS session cookies, with an 80% success rate using 2^{31} ciphertexts. This is an improvement over the preferred cookie-recovery attack of AlFardan *et al.* [10].

Although this work has undoubtedly contributed to the body of work responsible for bringing the use of RC4 in TLS to its knees (and thereby highlighting the folly of allowing the use of flawed primitives in TLS 1.2 and below), it is its potential for predictive power that makes it a valuable contribution to the RC4 attack space. This is especially true given recent attempts at making the use of RC4 viable in TLS, such as the TLS Scramble and MCookies techniques of Levillain *et al.* [94]. Using techniques introduced by the work presented in this chapter, Paterson and Schuldt [118] show that the techniques put forth in [94] only moderately increase the security of RC4 in TLS. In the case of TLS Scramble, the authors of [118] were able to adapt the techniques developed in our work to perform a theoretical evaluation of the number of ciphertexts needed to guarantee the success of their cookie-recovery attack with high probability.

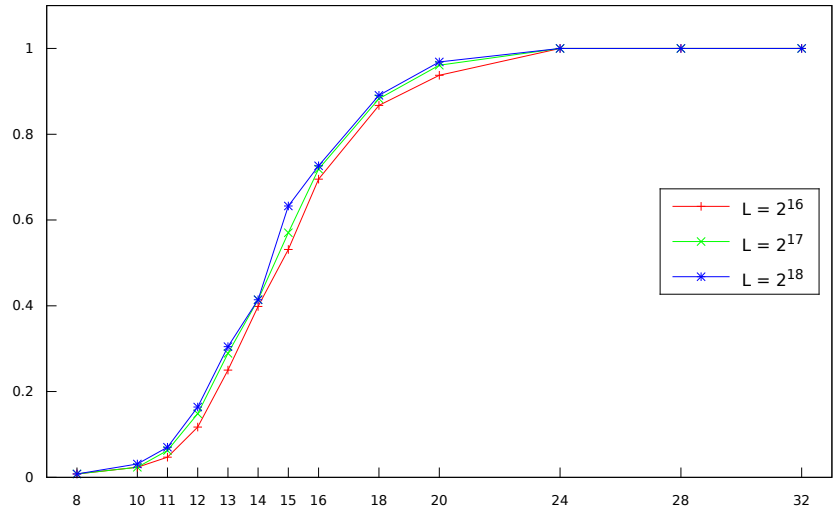


Figure 5.5: Success rate of beam search algorithm in recovering a 16-byte unknown plaintext for different numbers of ciphertexts, N, and different sizes of L, using double-sided biases and 130 known plaintext bytes. The x-axis shows the number of ciphertexts divided by 2^{27}.

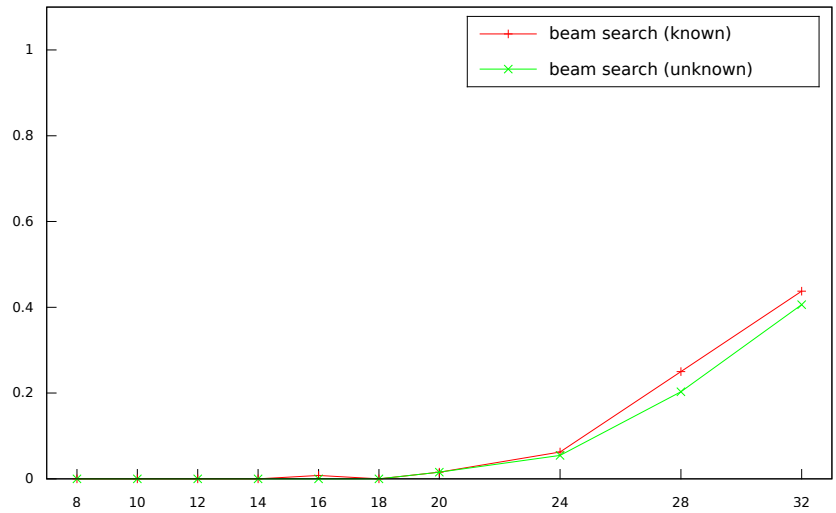


Figure 5.6: Success rate of the beam search algorithm in recovering a 17-byte plaintext (first byte known) using single sided-biases with 65 known plaintext bytes compared to recovering a 16-byte unknown plaintext using single-sided biases with 64 known plaintext bytes, for different numbers of ciphertexts, N, and for $L = 2^{16}$. The x-axis shows the number of ciphertexts divided by 2^{27}.

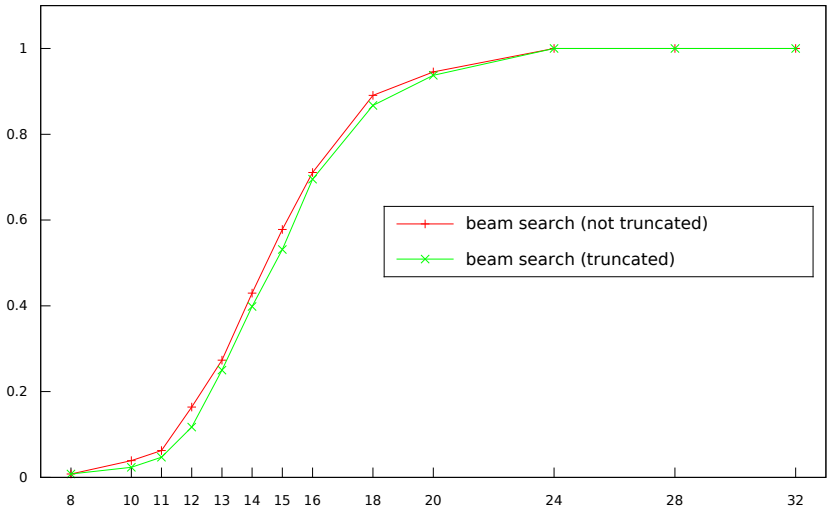


Figure 5.7: Success rate of beam search algorithm without final list pruning compared to use of final list pruning in recovering a 16-byte unknown plaintext for different numbers of ciphertexts, N, using double-sided biases and 130 known plaintext bytes, and for $L = 2^{16}$. The x-axis shows the number of ciphertexts divided by 2^{27}.

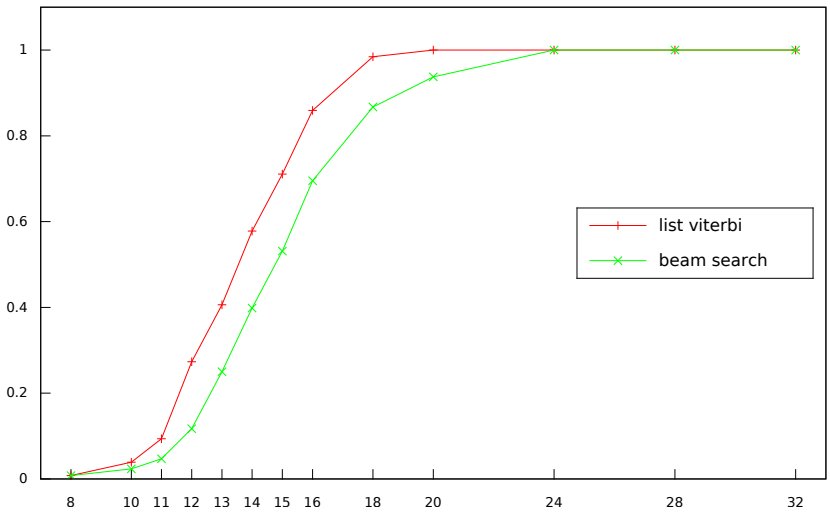


Figure 5.8: Success rate of list Viterbi algorithm compared to beam search algorithm in recovering a 16-byte unknown plaintext for different numbers of ciphertexts, N, using double-sided biases, $L = 2^{16}$, and 130 known plaintext bytes. The x-axis shows the number of ciphertexts divided by 2^{27}.

Part III

Verifying TLS 1.3

Chapter 6

Automated Analysis and Verification of `draft-10`

This chapter covers our symbolic analysis of the TLS 1.3 draft specification using the Tamarin *prover. We introduce the symbolic setting and cover the necessary* Tamarin *fundamentals. We formally model* `draft-10` *of the specification and encode the desired security properties, as laid out in the draft, using the* Tamarin *specification language. We use a mixture of automated inspection and manual interaction with the tool to analyse these properties. We also extend this model to include the post-handshake authentication mechanism as suggested for* `draft-11`*. Our results represent some of the first supporting evidence of the security of several handshake mode interactions in TLS 1.3.*

6.1 Introduction

Since its emergence in the Spring of 2014, the TLS 1.3 specification has been subject to much analysis and refinement, as described in Chapter 3 of this thesis. The flaws identified in TLS 1.2 and below, of which the preceding two chapters serve as testament, prompted the TLS WG into taking a different approach to the design of TLS 1.3, this time incorporating input from the academic community, and analysing the protocol thoroughly prior to deployment. A notable departure from the TLS 1.2 design is marked by the influence of the OPTLS protocol [91] of Krywczyk and Wee – this protocol was designed to offer a zero Round-Trip Time (0-RTT) exchange and to ensure perfect forward secrecy, meeting important designs goals for TLS 1.3. Its simple structure lends itself to analysis via manual and automated means, a benefit that was deemed desirable for TLS 1.3, but its influence introduced complexity into the protocol via the addition of new handshake

components, including the 0-RTT functionality and a new resumption mechanism based on Pre-Shared Keys (PSKs). Ensuring the security of TLS 1.3 in the face of this complexity is a vital task if attacks exploiting the interaction of differing handshake modes, such as the Triple Handshake attack against TLS 1.2 [29], are to be avoided.

Our work on `draft-10` of the TLS 1.3 specification tackles this task: We formally analyse the interaction of the various `draft-10` handshake modes, showing the absence of an attack against the `draft-10` specification.[1] Prior to our work on `draft-10`, the most up-to-date analysis pertaining to TLS 1.3 was the analysis of OPTLS by its designers [91]. In this work, the authors noted that their analysis was not intended to cover the full TLS 1.3 specification. In particular, they only considered the different handshake modes in isolation, and the work did not cover client authentication or resumption.

Our analysis complements the work from [91] and previous works [54, 85] by covering the following aspects:

(i) The security, and secure interaction, of the following handshake modes: regular (EC)DHE mode, PSK mode, PSK-DHE mode and 0-RTT mode.

(ii) The PSK-resumption handshake when composed with any acceptable initial handshake, namely, an (EC)DHE handshake, a PSK handshake, a PSK-DHE handshake or a 0-RTT handshake.

We go on to extend our `draft-10` model to incorporate the post-handshake authentication mechanism suggested for `draft-11` on the TLS WG mailing list [123], and hence a third aspect covered by our work includes:

(iii) The security of the proposed `draft-11` post-handshake authentication mechanism in the context of all previous modes.

For our analysis, we use the Tamarin prover [6], a state-of-the-art tool for the symbolic analysis of security protocols. The Tamarin framework enables us to specify and analyse the secrecy and authentication properties of the various handshake modes. Furthermore,

[1] We note that since the release of `draft-10`, the TLS 1.3 protocol has undergone many changes. The work presented in the next chapter of this thesis analyses `draft-21`, a near-final version of the protocol.

Tamarin's multiset-rewriting semantics are well-suited for modelling the complex transition system implied by the TLS 1.3 specification; the tool allows for the analysis of the interaction of the assorted handshake modes and the resulting infinite loops that may occur, as well as an unbounded number of concurrent TLS sessions. We note here that this thesis will not delve into the tool's theoretical foundations; we employ the tool as a means of analysing the TLS 1.3 protocol and hence only provide as much detail as is necessary for the understanding of this analysis. A formal treatment of Tamarin's theoretical foundations can be found in [141], [107] and [142].

Our analysis considers a Dolev-Yao adversary: an adversary that is able to observe network messages, alter and drop these messages, and possibly inject their own messages into the network, i.e., an adversary that has complete control of the network. Our adversary is also capable of revealing the long-term private keys of honest protocol participants. Our Tamarin model includes both the client authentication mechanism as well as session resumption. The OPTLS analysis did not include these components so our property specifications go well beyond the basic session key secrecy considered in [91].

We find that `draft-10` achieves the standard goals of authenticated key exchange. In particular, we show that a client has assurances regarding the secrecy of the established session key, as well as assurances regarding the identity of the server with whom it has established this key. The server obtains equivalent assurances when authenticating the client in the standard way, i.e., as part of an initial handshake, and when using the newly introduced 0-RTT mechanism. Our analysis confirms perfect forward secrecy of session keys and also covers the properties of handshake integrity (transcript agreement) and secrecy of early data keys. We verify these desirable properties in the presence of composable handshake modes and for an unbounded number of concurrent TLS sessions, something which had not been done in previous TLS 1.3 analyses.

Our exploration of the initial proposal for a post-handshake client authentication mechanism [123] resulted in the discovery of an attack. Specifically, an adversary is able to impersonate a client when communicating with a server owing to a vulnerability in the client authentication mechanism of the PSK-resumption handshake. Our attack highlights the strict necessity of creating a binding between consecutive TLS 1.3 handshakes.

Related Work. The 0-RTT mechanism of OPTLS, and hence of TLS 1.3, is similar to

that of Google's Quick UDP Internet Connections (QUIC) protocol [93]. Lychev *et al.* introduce a security model for what they term *Quick Connections (QC)* protocols and analyse QUIC within this framework [98]. Although they do not focus on TLS 1.3, they do point out that the 0-RTT mode of TLS 1.3, as described in `draft-10`, fits the definition of a QC protocol. Fischlin and Günther also provide an analysis of QUIC [60] by developing a Bellare-Rogaway style model [23] for *multi-stage* key exchange protocols.

The TLS 1.3 Handshake Protocol can be viewed as a multi-stage key exchange protocol because the communicating parties establish multiple session keys during an exchange, potentially using one key to derive another. In work by Dowling *et al.* [54], two TLS 1.3 drafts, specifically `draft-05` and `draft-dh` are analysed using the multi-stage framework. Although the authors showed that keys output by the Handshake Protocol could be securely used by the Record Protocol, at the time of writing, the TLS drafts did not include a 0-RTT mode and resumption had not yet been merged with the PSK mode. Kohlweiss *et al.* also produced an analysis of `draft-05` using a *constructive-cryptography* approach [85].

Although there were changes including a reduction in handshake latency, removal of renegotiation and a switch to AEAD ciphers in the earlier drafts of TLS 1.3, it is not until `draft-07`, with the encroaching influence of OPTLS, that we see a shift in the design of the protocol away from TLS 1.2. Hence, the results described above do not easily transfer to later drafts.

In concurrent work to ours, Li *et al.* analyse `draft-10` in the computational setting using their *multi-level&stage* security model which examines the composition of the various TLS 1.3 handshake modes [95]. Their work, however, does not include the proposed post-handshake authentication mechanism (not surprising as this was not part of the `draft-10` specification). In an update on their previous computational analysis, Dowling *et al.* released results on `draft-10` at a similar time to our work [55]. Their work shows the initial (EC)DHE handshake to be secure in the multi-stage key exchange model. To our knowledge, ours is the first published work examining TLS 1.3 in the symbolic setting, and indeed the first to consider the full interaction of the various components of TLS 1.3.[2]

[2]Work relating to later drafts of TLS 1.3 will be covered in Chapter 7.

6.2 Preliminaries

We now present the notation, concepts and definitions used throughout this chapter, including a description of symbolic analysis as well as an introduction to the Tamarin prover [6], covering the fundamentals necessary for the understanding of our TLS 1.3 analysis.

6.2.1 Symbolic Analysis

Symbolic analysis is one means by which to analyse critical security protocols such as TLS. Other methods include inspecting concrete implementations of the protocol (program verification), or producing what are traditionally thought of as "pen and paper security proofs" via computational analysis. In the symbolic model, introduced by Needham and Schroeder [112] and Dolev and Yao [53], cryptographic primitives are represented as *function symbols*, and messages are represented by abstract *terms* on these symbols (as opposed to bitstrings). For instance, a symmetric encryption primitive could be represented by the two function symbols `enc` and `dec`, and encryption and decryption of a message `m` under a key `k` is modelled using the following equality:

```
dec(enc(m,k), k) = m.
```

All cryptographic primitives are modelled in this fashion, using the appropriate function symbols and specifying the necessary equations. A consequence of this approach is that all primitives are considered to be *perfect*, meaning that the only operations possible are those that satisfy the specified primitive equations, as in the encryption example above. An adversary would not be able to learn anything about the length of the message being encrypted, for instance. We expound more on this notion of *perfect cryptography* in Section 6.2.2.

In the symbolic model, all honest protocol participants, as well as the adversary, only have access to the primitives (and equations) which have been specified. This is in contrast to the popular computational model in which primitives are modelled as functions from bitstrings to bitstrings, and where all network actors may invoke equalities which have not been explicitly specified. We do not go into detail here concerning the differences between the symbolic model and the computational model – a more considered treatment of the

differences between these two models is given in [36].

As pointed out in [36], security properties generally fall into one of two categories, namely, *trace properties* and *equivalence properties*. Trace properties are properties that can be defined for each possible run of the protocol; a trace property is satisfied when it holds for each applicable run of the protocol. Equivalence properties capture the ability of an adversary to distinguish between two processes, or two instantiations of a protocol.[3] Symbolic analysis typically concerns itself with trace properties, making it well-suited to automation, and indeed, many automated symbolic analysis tools have come to the fore in recent years, including the Scyther [5], ProVerif [4] and Tamarin [6] tools. We note that automated tools are also available for the computational setting, see CryptoVerif [1] for instance, but these are not the focus of this chapter.

We elaborate on the concepts described above and make them more concrete by introducing the Tamarin tool.

6.2.2 Tamarin Fundamentals

The Tamarin specification language facilitates the construction of highly detailed models of security protocols, their security requirements, and powerful Dolev-Yao-style attackers. The Tamarin prover takes as input the security protocol model, a specification of the adversary, and a specification of the protocol's desired properties. It then attempts to construct a proof showing that the protocol meets the specified properties in the presence of the specified attacker, even when arbitrarily many interleaved protocol executions are instantiated.

The steps required for analysing a protocol with the Tamarin prover include: (i) building an abstract model of the protocol using the tool's specification language, including modelling honest protocol participants, as well as the network-controlling adversary, (ii) encoding the desired protocol security properties, also using the Tamarin specification language, and (iii) constructing proofs for the specified properties within the Tamarin framework.

As a precursor to our TLS 1.3 analysis, we introduce the Tamarin specification language

[3] This is what is known as indistinguishability in the computational model.

and discuss these three steps by first considering a simpler network protocol, namely the Basic Station-to-Station (STS) protocol introduced by Diffie, van Oorschot and Wiener [52], depicted in Figure 6.1.

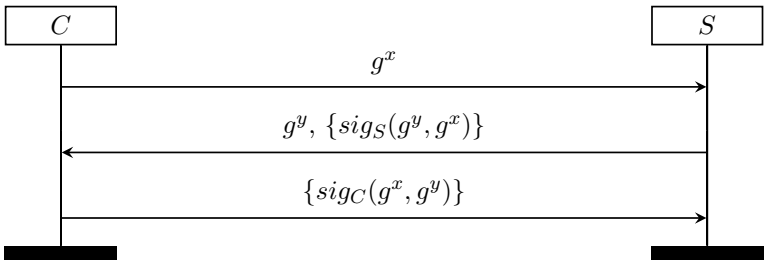


Figure 6.1: The Basic Station-to-Station (STS) Protocol. Brackets of the type {} indicate encryption under a key, k, which is shared by C and S.

We assume that entities make use of known, system-wide Diffie-Hellman parameters p and g. The parameter p is a large prime and g is a generator of the multiplicative group of integers modulo p. We also assume that entities have access to the public keys of peers via a Public Key Infrastructure (PKI). The client, C, generates a Diffie-Hellman exponent, x, and sends its key share, g^x, to the server, S. Upon receipt of the client's key share, the server generates a Diffie-Hellman exponent, y, as well as its key share, g^y. The server computes a shared key, k, from the value $(g^x)^y$ and signs (g^y, g^x) using its private key. The server sends this signature to the client encrypted under k, along with its key share, g^y. Upon receipt of the server's key share, the client computes k from $(g^y)^x$ and decrypts the signature sent by the server. The client then uses the server's public key to verify the server's signature on (g^y, g^x). The client signs (g^x, g^y) using its public key and encrypts this signature using k. Upon receipt of this message from the client, the server decrypts and verifies the client signature.

The STS protocol was constructed to overcome the well-known Man-In-The-Middle (MITM) attack against Diffie-Hellman key agreement [52]. In addition to providing secrecy of the shared key k, or any other session key derived from $(g^x)^y$, the protocol provides mutual authentication of the communicating entities. This thwarts the attack against the classic Diffie-Hellman protocol in which an adversary exploits the lack of entity authentication to establish distinct session keys with C and S, respectively, with both parties being none the wiser.

We now explain how the STS protocol can be analysed within the TAMARIN framework, describing how we (i) build a model, (ii) encode security properties and (iii) construct proofs to verify these properties. We introduce all of the necessary TAMARIN concepts for the execution of each of these steps prior to describing the STS example.

6.2.2.1 Modelling

A TAMARIN model defines a labelled transition system whose state space comprises a multiset or "bag" of *facts*, representing the adversary's knowledge, messages on the network, and the state of protocol participants, respectively. The permissible state transitions are specified by TAMARIN *rules* which encode the behaviour of protocol participants, as well as adversarial capabilities. A sequence of state transitions within the model gives rise to a *trace* – the sequence of labels produced upon instantiation of the model rules. Rules and facts make up the basic ingredients necessary for building an abstract model in TAMARIN, along with terms and *equational theories*. The role of each of these elements in the modelling process will be made clear in the subsections to follow:

Facts and Rules. The concept of terms in symbolic analysis was introduced above in Section 6.2.1 – these are how we model cryptographic messages and quantities. Facts are of the form `F(t1,t2,...,tn)` where `F` is a fact symbol and the `ti` are terms. Facts are fundamental to the concept of TAMARIN rules, which we introduce by considering the `Fresh` rule:

```
rule Fresh:
[ ]--[ ]->[ Fr(~x) ]
```

TAMARIN rules consist of three respective parts: a *left-hand side* (`[ ]`), *actions* (`[ ]`), and a *right-hand side* (`[ Fr(~x) ]`). The rules are used to define a transition system, whose global state is maintained as a multiset of facts. The initial state of the transition system is the empty multiset. A rule can only be executed if all the facts on its left-hand side are available in the current state. When a rule is executed, it will *consume* the facts on the left, i.e., remove them from the global state, and *produce* facts on the right, i.e., add them to the global state. This "rewriting" of the state sometimes leads us to refer to TAMARIN rules as *rewrite rules*. Facts are either *linear* or *persistent*. While linear facts model limited resources that cannot be consumed more times than they are produced, persistent facts

model unlimited resources, which can be consumed any number of times once they have been produced. In the `Fresh` rule above there are no premises (left-hand side facts) or actions, and every execution of the rule produces a single linear `Fr(~x)` fact. Only the `Fresh` rule can produce `Fr` facts and each of these facts will be unique and will represent a discrete value within the TAMARIN framework. Use of the ∼ symbol denotes a variable of the type `Fresh`. Other variable types include `Public`, denoted by $, and `Temporal`, denoted by #.

Actions do not influence transitions but are "logged" when rules are triggered as a means of incrementally constructing observable labels for use in traces, which in turn represent a record of a specific execution. Actions are used to express security properties, thereby forming the glue between the defined transition system and the specified protocol properties.

In TAMARIN there are three special types of facts: `Fr` facts, as described above, `In` facts and `Out` facts. Whereas `Fr` facts model the generation of unique, fresh values, `In` facts and `Out` facts model interaction with an untrusted network. An `In` fact models a protocol participant receiving a message from the untrusted network, a message that has potentially been manipulated by a network-controlling (Dolev-Yao) adversary. An `Out` fact models a protocol participant sending a message to the untrusted network, where it could potentially be manipulated by a Dolev-Yao adversary. `In` facts can only be present on the left-hand side of TAMARIN rewrite rules, and `Out` facts can only occur on the right-hand side of TAMARIN rewrite rules. The `Fresh` rule described above is a built-in TAMARIN rule, and the tool also contains sets of adversary rules which consume `Out` facts and produce `In` facts, modelling how an adversary may interact with, and manipulate, network messages.

Cryptographic Primitives. The ability to model cryptographic protocols requires the representation of cryptographic primitives. In the symbolic setting, we model cryptographic functionality using functions and sets of equations that describe the relationship between these functions. In TAMARIN, symmetric encryption, for instance, is modelled using two binary functions, `senc` and `sdec`, and an equation of the form

```
sdec(senc(m, k), k) = m
```

where `k` is a shared secret key and `m` is a message. Hence, symbolically, the term `senc(m, k)` can be combined with `k` to recover `m`. Without explicitly defining a rule that allows the adversary to obtain the shared secret `k`, there is no way for the adversary to learn `k` and

```
rule Gen_keypair:
 [ Fr(~ltkA) ]--[ GenLtk($A, ~ltkA)
 ]->
 [ !Ltk($A, ~ltkA), !Pk($A, pk(~ltkA))]

rule Reveal_Ltk:
  [ !Ltk($A, ~ltkA) ] --[ RevLtk($A) ]-> [ Out(~ltkA) ]
```

Figure 6.2: TAMARIN PKI rules for the STS protocol.

recover `m`. In other words, as is to be expected, the symmetric encryption equational theory does not allow for the adversarial extraction of `k` from `senc(m,k)` – there is no defined equation describing this type of extraction.

As certain primitives are used repeatedly across many cryptographic protocols, there are built-in definitions for them. The TAMARIN *built-ins* include equational theories for Diffie-Hellman group operations, asymmetric encryption, symmetric encryption, digital signatures and hashing.

Perfect Cryptography. The use of TAMARIN built-ins results in the abstraction that cryptographic primitives are perfect, as is to be expected in the symbolic model. For example, the encryption mechanism reveals nothing about the underlying plaintext since all cryptographic inputs and outputs are considered to be abstract quantities, without being sampled with a specific length or from a specified distribution. Similarly, signatures are unforgeable, hash functions act as random oracles (with zero collision probability), MACs are unforgeable, and all parties generate truly random values. The assumption of perfect cryptography is common in the symbolic setting, however, it is indeed possible to model weak primitives. We could, for example, introduce rules which let the adversary create signature forgeries, or MAC forgeries, but all weaknesses and flaws need to be *explicitly* defined.

We present the rules necessary for modelling the STS protocol in Figures 6.2, 6.3 and 6.4.

Our first client rule generates a fresh Diffie-Hellman exponent (`Fr[~a]`), logs actions `Start(~a, $C, 'client')` and `DH($C, ~a)`, and sends the client's key share out to the network (`Out(<$C,'g'^~a>)`). The rules to follow become somewhat more complex as long-term keys are retrieved from the PKI rules, and as data items are encrypted and signed using the TAMARIN builtins – see `senc{sign{<'g'^~b, ckeyshare>}~ltkS}k` in the `Server_1`

```
rule Client_1:
    [ Fr(~a)
    ]
  --[ C1(~a)
    , Start(~a, $C, 'client')
    , DH($C, ~a)
    ]->
    [ Client_1($C, ~a)
    , Out(<$C,'g'^~a>)
    ]

rule Client_2:
let
    k = skeyshare^~a
in
    [ Client_1($C, ~a)
    , !Ltk($C, ~ltkC)
    , !Pk(S, pk(~ltkS))
    , In(<S, skeyshare, s_encryptedsignature>)

    ]
  --[ C2(~a)
    , Neq(S, $C)
    , Eq(verify(sdec{s_encryptedsignature}k, <skeyshare, 'g'^~a>,
    pk(~ltkS)), true)
    , SessionKey($C, S, k, 'authenticated')
    , SignC('g'^~a, skeyshare)
    , Running($C, S, 'client', 'g'^~a, skeyshare)
    , Commit($C, S, 'client', 'g'^~a, skeyshare)
    ]->
    [ Client_2($C, S, ~a, skeyshare, k)
    , Out(senc{sign{<'g'^~a, skeyshare>}~ltkS}k)
    ]
```

Figure 6.3: Tamarin client rules for the STS protocol.

```
rule Server_1:
let
    k = ckeyshare^~b
in
    [ Fr(~b)
    , !Ltk($S, ~ltkS)
    , In(<C,ckeyshare>)
    ]
  --[ S1(~b)
    , Start(~b, $S, 'server')
    , UseSessionKey(C,k)
    , Neq($S, C)
    , SignS('g'^~b, ckeyshare)
    , Running($S, C, 'server', 'g'^~b, ckeyshare)
    ]->
    [ Server_1($S, C, ~b, ckeyshare, k)
    , Out(<$S,'g'^~b, senc{sign{<'g'^~b, ckeyshare>}~ltkS}k>)

    ]

rule Server_2:
    [ Server_1($S, C, ~b, ckeyshare, k)
    , !Pk(C, pk(~ltkC))
    , In(c_encryptedsignature)
    ]
  --[ S2(~b)
    , Eq(verify(sdec{c_encryptedsignature}k, <ckeyshare, 'g'^~b>,
    pk(~ltkC)), true)
    , SessionKey($S, C, k,'authenticated')
    , Commit($S, C, 'server', 'g'^~b, ckeyshare)
    ]->
    [ Server_2($S, C, ~b, ckeyshare, k)
    ]
```

Figure 6.4: TAMARIN server rules for the STS protocol.

rule, for instance. The `senc` and `sign` constructs use the keys `k` and `~ltkS` to encrypt and sign data items, respectively. Decryption and subsequent verfication of the encrypted signatures occur by applying the `sdec` function, and by checking that the signature verification returns a `true` result (`Eq(verify(sdec{c_encryptedsignature}k, <ckeyshare, 'g'^~b>, pk(~ltkC)), true)`). State transitions within the model occur via the production and subsequent consumption of *state facts*. The `Client_1` rule produces the `Client_1($C, ~a)` state fact. In order for the next client rule, `Client_2`, to fire, this fact needs to be consumed by the left-hand side of the second client rule, which indeed it is. This 'passing' of state facts within the model gives rise to the model transition system. The rule actions serve as labels for the state transitions, thereby creating a labelled transition system.

6.2.2.2 Encoding Security Properties

In order to prove security properties within the Tamarin framework, properties need to be encoded using the Tamarin specification language. In Tamarin, this is done by constructing what are known as Tamarin *lemmas*. These lemmas are specified in a guarded fragment of first-order logic, allowing for quantification over messages and timepoints, and the resultant logic formulae are considered across applicable traces.

As pointed out in Section 6.2.1, properties fall into one of two categories: trace or equivalence properties. A trace property is satisfied when it holds for all applicable execution traces of the protocol. Equivalence properties allow for the encoding of more nuanced security properties since these capture the ability of an adversary to distinguish between two processes, or two instantiations of the protocol. Symbolic tools, in general, focus on proving trace properties as the automation of equivalence properties is more difficult to achieve.[4] Tamarin also offers a mechanism for restricting the traces considered. We discuss these *restrictions* shortly.

Lemmas. Since Tamarin formulae are specified in a fragment of first-order logic they offer the usual connectives (where `&` and `|` denote *and* and *or*, respectively), quantifiers `All` and `Ex`, and timepoint ordering `<`. In formulae, the prefix `#` denotes that the variable to follow is of type *timepoint*. The expression `Action(args)@t` denotes that `Action(args)` is logged in the action trace at point `t`, resulting from an instantiation of a rule.

[4] This is true for the Tamarinprover, however, recent work [20] has extended the tool to accommodate the proving of equivalence properties.

We use the Tamarin property language to encode different kinds of properties as Tamarin lemmas. These can include, for example, basic state reachability tests as well as security properties. Constructing state reachability lemmas is an important part of the symbolic analysis process – ensuring that each state within the model is reachable ensures that property lemmas will indeed be considered across *all* applicable traces, and we are also provided with some assurances regarding the correctness of our model. The successful execution of state reachability tests instils confidence in the model's *state machine*, which in turn serves as a valuable resource when it comes to proving protocol properties. In order to perform state reachability tests, it is convenient to temporarily remove the network-controlling adversary from the model. In other words, we would like to ensure the existence of a secure channel between honest protocol participants. We do this by replacing `In` and `Out` facts with `AuthMessage` facts. For instance, `AuthMessage($A, $B, m)` represents sending message `m` from `$A` to `$B` in a secure fashion as the `AuthMessage` facts will only be passed from honest party rule to honest party rule and will never become adversary knowledge since `In` and `Out` facts are never consumed or produced.

Restrictions. The Tamarin tool also allows for the specification of *restrictions*. These restrict the set of traces considered during analysis. For example, the following restriction instructs the Tamarin tool to only consider traces where the specified equality check succeeds:

```
restriction Equality_Check_Succeeds:
   "All x y #i. Eq(x,y) @ i ==> x = y"
```

We typically use restrictions to avoid traces where protocol participants initiate sessions with themselves, or where large numbers of key pairs are generated for a single protocol participant as this may make it difficult to isolate the correct key pair for the protocol thread under analysis.

In order to check correctness of our STS Tamarin model, we replace the `In(m)` and `Out(m)` facts with `AuthMessage($C, $S, m)` to construct reachability lemmas. When checking reachability we employ the following restriction, where `ACTOR` can be `C` or `S`, respectively, and `*` represents the state label (i.e., `1` or `2`):

```
restriction At_most_1_of_ACTOR*:
   "All #i #j tid1 tid2. ACTOR*(tid1)@#i & ACTOR*(tid)@#j ==> #i = #j"
```

This ensures that we only ever instantiate one client and one server. We can then write simple reachability lemmas of the form:

```
lemma exists_ACTOR*:
exists-trace
"Ex tid #i. ACTOR*(tid)@i"
```

This lemma simply checks whether or not the action `ACTOR*(tid)` is triggered (for a particular thread identifier, `tid`), which corresponds to the protocol participant `ACTOR` reaching state `*`. Our STS model gives rise to the simple state machine presented in Figure 6.5. We check the above lemma for all states so as to confirm that all states in our state machine are reachable. The verification of lemmas in Tamarin will be discussed in the next section, Section 6.2.2.3.

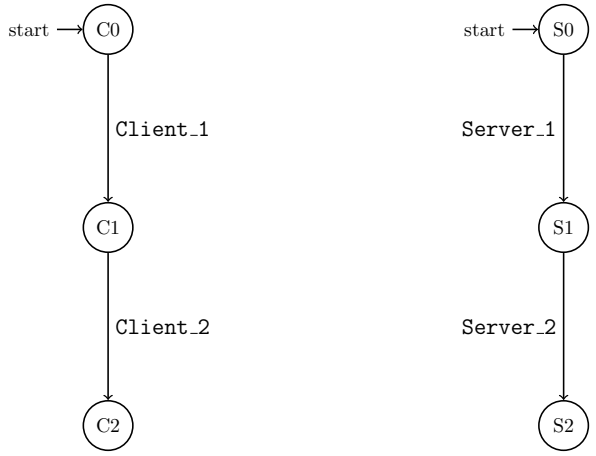


Figure 6.5: Simple state diagram for the STS protocol, as encoded in our STS Tamarin model.

The trace properties typically considered in the analysis of Authenticated Key Exchange (AKE) protocols include secrecy and authentication. In the case of the STS protocol, we would like to show that the shared key k, is known only to the honest protocol participants. We encode this secrecy property in the following lemma:

```
lemma session_key_secrecy:
(1) "All actor peer k #i.
(2)  SessionKey(actor, peer, k,'authenticated')@i
(3)  & not ((Ex #r. RevLtk(peer)@r & #r < #i)
        | (Ex #r. RevLtk(actor)@r & #r < #i))
(4)  ==> not Ex #j. K(k)@j"
```

This lemma requires that for all protocol behaviours, and for all variables listed in (1), if an authenticated session key is accepted (encoded in the `SessionKey` action) (2), and the adversary has not revealed the long-term keys of honest protocol participants (3), then the

adversary does not the know the key, `k` (4) (adversarial knowledge is encoded via the `K()` construct). Typically, secrecy is represented as the adversary not having knowledge of a data item, which is in this case the shared session key.

We would also like to show that the protocol participants have some guarantees regarding the identity of their communication partners. Although this general notion of entity authentication provides an intuitive understanding of the property, it is more instructive to consider Lowe's more granular set of definitions for authentication [97]. Lowe constructs a hierarchy of authentication properties, starting with basic *aliveness*, running all the way through to *full agreement*. The weakest authentication property, aliveness, captures the notion that at some point, protocol participants were engaged in a run of the protocol. The strongest authentication property, full agreement, captures the notion that both protocol participants agree on all possible data items that could have been exchanged, or created as a result of the exchange, during a run of the protocol, and that there is a one-to-one relationship between the protocol runs, i.e., every protocol run of the initiator corresponds to a unique run of the protocol by the responder. Prior to reaching full agreement in the hierarchy, we encounter Lowe's notions of *agreement* and *injective agreement*. Agreement guarantees that protocol participants agree on a set of data items (not necessarily all possible data items), and injective agreement, whilst also providing agreement, additionally incorporates the one-to-one protocol run condition. Hence, full agreement is injective agreement expanded to cover all possible data items. Each of Lowe's authentication properties can be strengthened by incorporating a notion of *recentness* – this incorporates the additional requirement that protocol participants are currently running the protocol. This is achieved by participants agreeing on fresh values, such as nonces, timestamps or sequence numbers.

These agreement-style notions are typically used for capturing authentication properties in symbolic analysis, and in our STS example, an authentication lemma (encoded as an agreement-style property) may look like this:

```
lemma entity_authentication:
(1) "All actor peer keyshare1 keyshare2 #i.
(2) Commit(actor, peer, 'server', keyshare1, keyshare2)@i
(3)  & not ((Ex #r. RevLtk(peer)@r) | (Ex #r. RevLtk(actor)@r))
(4)  ==> (Ex #j. Running(peer, actor, 'client', keyshare2, keyshare1)@j
(5)       & #j < #i)"
```

This lemma requires that for all protocol behaviours, and for all variables listed on the

first line (1), that when a server logs that it has observed certain key shares (2), and the long-term key of neither party has been revealed (3), then there existed a thread of the peer, running as the client, that agreed on these key shares at an earlier point in time (5). This corresponds to Lowe's notion of agreement.

6.2.2.3 Constructing Proofs

The Tamarin verification algorithm is based on constraint solving and multiset-rewriting techniques. A proof for a property, encoded as a guarded first-order logic formula, is constructed using a backwards search over applicable traces to check satisfiability of the formula. This is done by considering the negation of the encoded property as a constraint solving problem. If the constraints cannot be met, the property cannot be negated, and hence the property holds. If the constraints can be met, then we have a counterexample, which typically constitutes an attack. In other words, a proof consists of showing that the logic statement holds across all applicable traces.

In proving our secrecy lemma for the STS protocol, the Tamarin tool attempts to find a contradiction – a state in which `SessionKey(...)@i` and `K(k)@j` hold, without the long-term keys of honest participants having been revealed. The tool starts by considering all states in which the `SessionKey(...)` action occurs. This action may be present in multiple states. Using the process outlined above, the Tamarin prover then works backwards through applicable traces until either a contradiction is found, or the applicable traces are shown to be contradiction-free. In more detail, Tamarin's proof state is a collection of constraints that represent partial information concerning a set of traces in which the negated property may indeed hold. The tool uses constraint solving techniques and case distinctions in an attempt to collect enough information to either establish that the set is empty, or to construct a member of set, i.e., a counterexample. It is also possible for the tool to make use of previously proven properties (lemmas) whilst trying to prove the current property; the backwards search may trigger a previously proven property and conclude that the current property holds. This behaviour is extremely advantageous when attempting to prove properties for complicated protocols. Being able to prove, and subsequently use, auxiliary lemmas allows for a modular approach to proving security properties within the Tamarin framework.

```
==============================================================================
summary of summaries:

analyzed: sts.spthy

  session_key_secrecy (all-traces): verified (47 steps)

==============================================================================
```

Figure 6.6: Tamarin verification of the STS secrecy lemma via the tool's command line option.

Tamarin also allows for the construction of proofs via induction[5] – instead of employing backwards reasoning, i.e., deriving information about earlier states by starting with later states, the tool is able to reason forwards by making assumptions about earlier states to derive information about later states. We do not describe the specialised mechanism by which this is achieved in Tamarin, a thorough treatment of the topic is provided in [107]. Invoking Tamarin's support for inductive proofs is remarkably useful when trying to prove properties about protocols which may exhibit looping behaviour (such as TLS).

In Tamarin a proof can be obtained in one of two ways: First, Tamarin has a fully automated mode that performs the proof search. If the automated proof search terminates, then the tool either returns a proof of correctness or a counter-example, demonstrating an attack against the property in question. However, as the correctness of security protocols is an undecidable problem,[6] the tool may not terminate for a given verification problem. In this case, users have the second option of turning to Tamarin's interactive mode, comprising an extensive graphical user interface that enables the visualisation and interactive construction of proofs. Manually guiding the tool through its backwards search employs the user's intimate knowledge of the modelled protocol to verify a property more efficiently than would be done via the Tamarin heuristics employed in the tool's automated mode. This greatly improves the likelihood of termination.

Using Tamarin's automated proof search for the STS secrecy lemma results in verification of the lemma in 47 steps, as displayed when calling the Tamarin prover from the command line. This result is depicted in Figure 6.6. Using the graphical interface allows for interaction with Tamarin graphs, as depicted in Figures 6.7 and 6.8. In these graphs, green boxes represent model rules, with premises (left-hand side facts) and conclusions (right-hand side facts) on the top and the bottom, respectively. Grey arrows represent premise sources, and

[5] Tamarin supports a limited form of inductive reasoning. Induction can be performed over actions arising from user-defined rules, and not over built-in adversary rules.

[6] A problem is undecidable if it is impossible to construct an algorithm that consistently returns a correct 'yes' or 'no' answer to the problem.

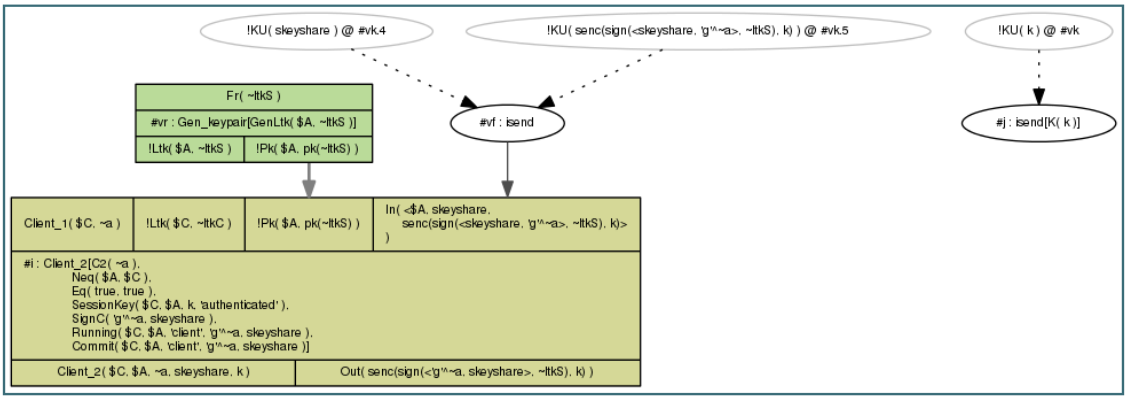


Figure 6.7: Partial Tamarin graph for the STS secrecy lemma. The adversary will attempt to construct the `In` fact of the `Client_2` rule from information that it might learn from the network.

grey bubbles represent adversary goals. We note that the graphs presented in Figures 6.7 and 6.8 were constructed during an early phase of the lemma proof process, and hence they are not populated with many elements. Graphs such as these are generated by user-selection of which goal (typically a premise source or an adversary goal) should be solved by the tool.

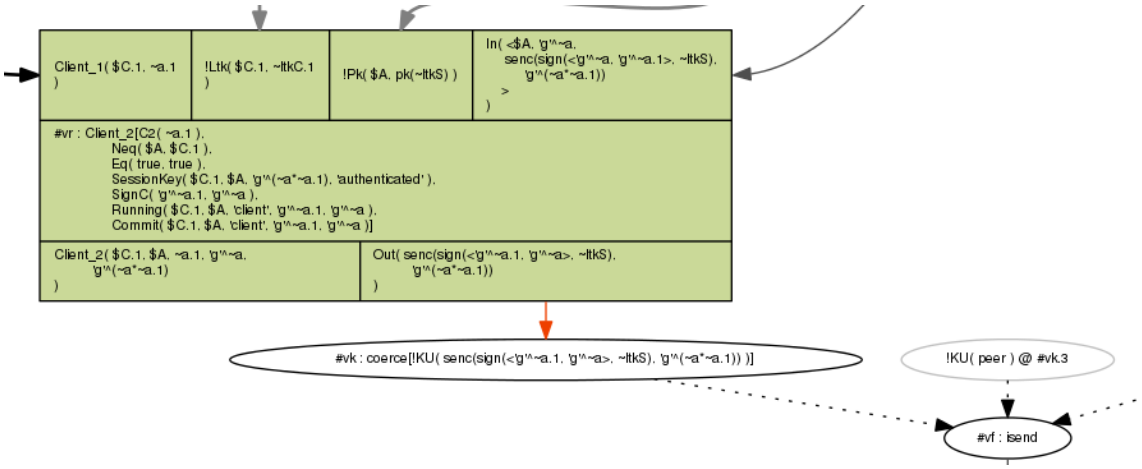


Figure 6.8: Excerpt from a partial Tamarin graph for the STS secrecy lemma. Red arrows represent the adversary obtaining information from the network, and then possibly being able to use this information to construct future rule premises, represented by dashed arrows.

The full STS Tamarin security protocol theory (.spthy) file is given in Appendix A. The authentication lemmas given in the file are also auto-provable.[7]

[7]It is possible to capture simple protocols and their properties in Tamarin with a few rules and lemmas. More complex protocols, such as TLS, require vastly more rules, and many more lemmas.

6.2.2.4 TAMARIN for TLS

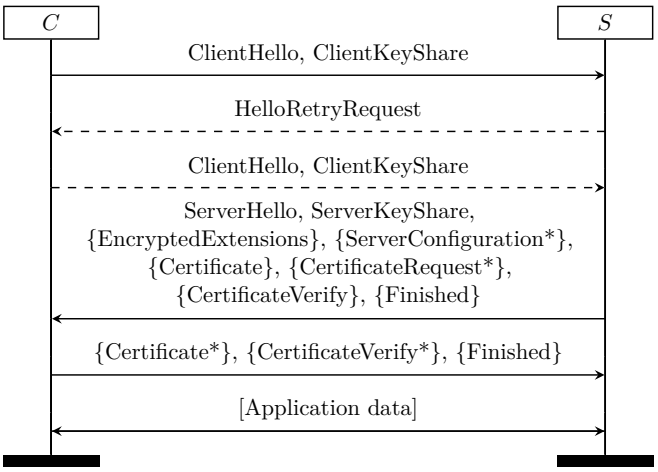
The TAMARIN prover inherently supports non-monotonic state, i.e., a state which need not necessarily grow after each rule is executed, and is thus ideal for the analysis of complex protocols exhibiting branches and loops. Monotonic state provers, such as the PROVERIF tool [4], struggle with loops because all facts are persistent and therefore once added to the state space, cannot be removed or overwritten, a feature which complicates the representation of looping as this necessitates the overwriting of facts. Also, since facts cannot be removed in monotonic state provers, support for ephemeral values is limited. This is not the case in TAMARIN. Non-monotonic state, together with TAMARIN's multiset-rewriting semantics provides the tool's support for branching. Moreover, the TAMARIN prover offers state-of-the-art symbolic Diffie-Hellman support, making it very useful in the analysis of Diffie-Hellman-based protocols such as TLS. An in-depth explanation of the tool's support for Diffie-Hellman key exchange can be found in [142].

6.3 draft-10 Analysis

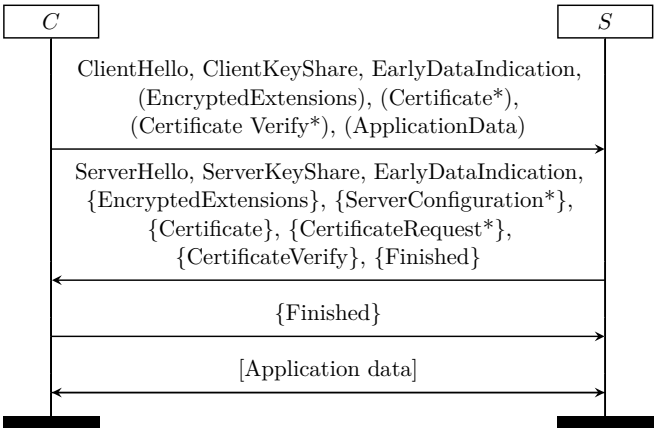
Using the TAMARIN framework, we build a formal model of the Handshake and Record protocols of draft-10 of TLS 1.3. The TAMARIN tool is well-suited for the analysis of TLS 1.3 for several reasons: First, TAMARIN's multiset-rewriting semantics enable a direct specification of the complex state machines of TLS 1.3, including the complex interactions between all of the handshake modes. Second, its state-of-the-art support for Diffie-Hellman allows for a high degree of precision in modelling the protocol. Third, its property specification language lets us model the TLS 1.3 security properties intuitively and accurately.

Our aim is to verify the properties of TLS 1.3 as an AKE protocol. Details concerning the various TLS 1.3 handshake modes, as well as the protocol's desired security properties, are given in Chapter 2; for ease of readability, we repeat these details here (see Figure 6.9 for handshake modes).

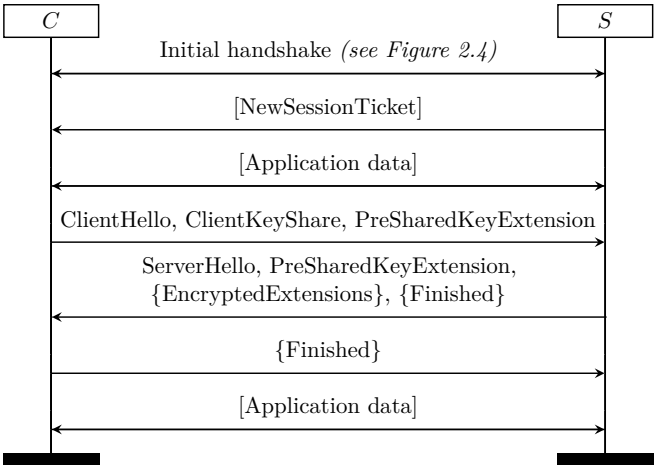
The TLS 1.3 draft-10 security properties include:



(a) draft-10 (EC)DHE handshake



(b) draft-10 0-RTT handshake



(c) draft-10 PSK resumption handshake

Figure 6.9: Handshake modes for draft-10.

1. **Secrecy of Session Keys.** Upon completion of the handshake, the client and the server should have established a set of session keys which are known to the client and the server only.

2. **Perfect Forward Secrecy (PFS).** In the case of compromise of either party's long-term key material, sessions completed before the compromise should remain secure. This property is not claimed to hold in the PSK-only handshake mode, nor in the 0-RTT handshake mode.

   In a PSK-only handshake, if compromised, the PSK could be used to decrypt all messages previously protected by the PSK. In the 0-RTT case, the client is the only party to have provided freshness, therefore early data messages may be replayed. In addition, the security of the early data depends on the semi-static (elliptic curve) Diffie-Hellman share, which may have a considerable validity period, and therefore a large attack window. For these reasons, early data cannot be considered to be forward secure.

3. **Peer (Entity) Authentication.** In the case of unilateral authentication, upon completion of the handshake, if a client `C` believes it is communicating with a server `S`, then it is indeed `S` who is in the server role. An analogous property for the server also holds in the mutual authentication case. Authentication of the server is mandatory in all handshake modes. Mutual authentication, i.e., additional authentication of the client, is optional.

4. **Integrity of Handshake Messages.** An active attacker should not be able to successfully tamper with the handshake messages, potentially causing the client and the server to adopt weak cipher suites.

Specifically, we would like to show that TLS 1.3 meets the listed properties in the presence of a Dolev-Yao adversary, and that all of the desired security properties hold when the interaction of all of the TLS 1.3 components is considered.

As with the STS protocol, we analyse the TLS 1.3 protocol using the Tamarin tool by (i) building an abstract model of the protocol, (ii) encoding the desired security properties as Tamarin lemmas, and (iii) constructing proofs for the specified properties using the Tamarin verification algorithm.

6.3.1 Building the Model

6.3.1.1 Constructing Model Rules

Modelling TLS 1.3 in Tamarin involves depicting the protocol as sequence of Tamarin rules, which capture client, server and adversary actions alike. In the case of legitimate clients and servers, our constructed model rules generally correspond to all processing actions associated with respective flights of messages. For instance, our first client rule captures a client generating and sending all necessary parameters as part of the first flight of an (EC)DHE handshake, as well as transitioning to the next client state within the model. In Figure 6.10, the `let...in` block allows us to perform basic variable substitutions. In practice, this is useful for enforcing the type of variables, such as `C = $C`, or for keeping the rule computations logically separated. We use the variable `tid` to name the newly created client thread. The action `DH(C,~a)` allows us to map the private Diffie-Hellman exponent `~a` to the client `C`. The `Start(tid, C, 'client')` action signifies the instantiation of the client `C` in the role of `'client'` and the `Running(C, S, 'client', nc)` action indicates that the client `C` has initiated a run of the protocol with the server `S`, using the fresh value `nc` as the `client_random` value as specified in TLS 1.3. The `C1` action simply marks the occurrence of the `C_1` rule with its associated `tid`. The `St_C_1_init` fact encodes the local state of thread `tid`, which doubles as a program counter by allowing the client to recall sending the first message in thread `tid`. The `Out` fact represents sending the first client message to the network, after which it becomes adversarial knowledge.[8]

Figures 6.11 and 6.12 capture all relevant model rules and represent the union of all the options that a client and a server have in a single execution. We explain the client-side behaviour and map it to the corresponding transitions while briefly mentioning the intended server interaction: The client can initiate three types of handshake, namely, an (EC)DHE handshake (`C_1`), a PSK handshake (`C_1_PSK`) and a 0-RTT handshake (`C_1_KC`); we use KC (an abbreviation for Known Configuration) to denote 0-RTT handshakes. In the (EC)DHE handshake, the server may reject the client parameters due to a possible mismatch, whereafter the client needs to provide new parameters (`C_1_retry`). Additionally, the client may optionally authenticate in the 0-RTT case (`C_1_KC_Auth`). While the (EC)DHE and 0-RTT handshakes only have a single continuation (respectively `C_2` and `C_2_KC`),

[8] The increase in complexity in comparison to the first client rule of the STS example is an inevitable consequence of modelling a much more complex protocol.

```
rule C_1:
let
    // Default C1 values
    tid = ~nc

    // Client Hello
    C   = $C
    nc  = ~nc
    pc  = $pc
    S   = $S
    $

    // Client Key Share
    ga   = 'g'^~a

    messages = <nc, pc, ga>
in
    [ Fr(nc)
    , Fr(~a)
    ]
  --[ C1(tid)
    , Start(tid, C, 'client')
    , Running(C, S, 'client', nc)
    , DH(C, ~a)
    ]->
    [ St_C_1_init(tid, C, nc, pc, S, ~a, messages, 'no_auth')
    , Out(<C, nc, pc, ga>)
    ]
```

Figure 6.10: Rule C_1 in our Tamarin model of TLS 1.3 draft-10.

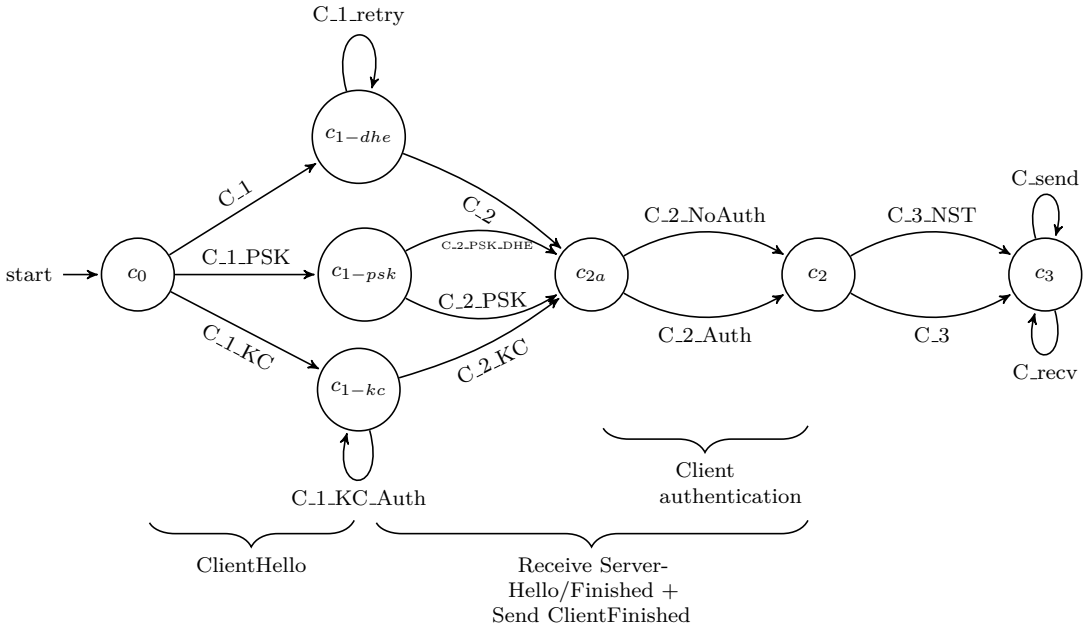


Figure 6.11: Partial client state machine for `draft-10` as modelled in our Tamarin analysis. The diagram represents the union of all the options for a client in a single execution. Not depicted are the additional transitions representing a client starting a new handshake using either a PSK established by `C_3_NST` or a `ServerConfiguration` for a 0-RTT handshake. Note that the `C_1_KC_Auth` edge may only occur once per handshake.

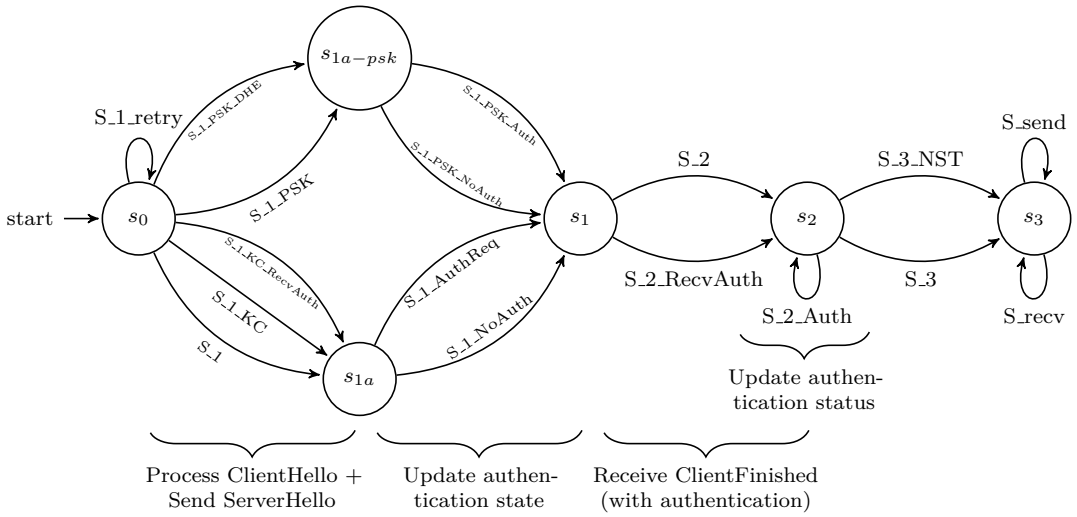


Figure 6.12: Partial server state machine for `draft-10` as modelled in our Tamarin analysis. The diagram represents the union of all the options for a server in a single execution. Not depicted are the additional transitions representing a server starting a new handshake using either a PSK established by `S_3_NST` or a `ServerConfiguration` for a 0-RTT handshake. Note that the `S_2_Auth` edge may only occur at most once per handshake.

the PSK handshake has two different modes: plain PSK (C_2_PSK) and PSK with DHE (C_2_PSK_DHE). The latter is used to obtain PFS guarantees by means of adding an ephemeral (elliptic curve) Diffie-Hellman value to the applicable key derivations.

We model the server as always requesting client authentication, but allow traces to capture when the server accepts and both rejects authentication of the client. If the client decides to authenticate, it sends the authentication messages along with the client Finished message (C_2_Auth). Otherwise, only the Finished message is transmitted (C_2_NoAuth). The handshake concludes with the client either receiving a new session ticket (C_3_NST), which can be used for resumption in a later PSK handshake, or doing nothing (C_3). The client can then proceed to send (C_Send) and receive (C_Recv) any finite number of application data messages.

We note that the Figure 6.11 represents a 'snapshot' in time beyond the initial establishment of a connection. Consequently, it lacks certain states and transitions in which the client, or the server, responds to an established connection. We list the omissions here as rules that can follow a previously executed rule, i.e., the rule given on the right-hand side of the arrow can follow the rule given on the left-hand side of the arrow:

C_2→C_1_KC
C_2_KC→C_1_KC
S_1→S_1_KC
S_1_KC→S_1_KC
S_1→S_1_KC_RecvAuth
S_1_KC→S_1_KC_Recvauth
C_3_NST→C_1_PSK
S_3_NST→S_1_PSK
S_3_NST→S_1_PSK_DHE.

Configuration Parameters. We note that we simplify our model by treating certain parameters as abstract quantities within the model. For example, the EncryptedExtensions message of a client in the 0-RTT handshake will be logically bundled together with all other messages of this kind and represented by the single public value exts. However, since these components comprise part of the handshake transcript, we establish whether the client and the server agree on these values by the end of the handshake through the transcript integrity property.

Alert Messages. We also do not explicitly model the TLS 1.3 alert protocol; our model does not capture errors arising from deviations in the protocol that would result in the

immediate termination of a connection (fatal alerts), or acknowledgements of a graceful shutdown (closure alerts). From the perspective of our model, and the security properties we are capturing, an alert and subsequent connection closure is equivalent to a trace which simply does not have any subsequent actions beyond the point at which the connection terminated.

Over-approximations. In certain situations, we assume that the client, or server, will send the maximal message load. For example, the client will always send 0-RTT data in the 0-RTT case. Similarly, we model the server as always including a `CertificateRequest` message in the first flight. However, the client does not always send authentication parameters, and the server does not necessarily accept these parameters if sent. Therefore, the possible traces we observe are equivalent to those in which the server optionally sends the certificate request.

6.3.1.2 Managing Model Complexity

The complexity of TLS 1.3 presents an interesting challenge for automated symbolic analysis. As Figures 6.11 and 6.12 demonstrate, the many handshake modes make for a large number of state transitions. In software engineering, conditional branches are a fairly mundane part of code. For example, the code might perform the check: "`if` received client authentication `then` verify signature and set client status to authenticated `else` do nothing". However, in Tamarin we require two distinct state transitions representing these two possibilities. The TLS 1.3 handshake exhibits such conditional branching. Ideally, branches in the protocol would be represented by as few rules as possible, which can be done by merging some of the resulting states into one. For example, by the end of the server's first phase, the state needs to contain a transcript of the received messages, the computed values of `ss` and `es`, and the authentication status of the client. While all four handshake modes will compute these values in a different way, from that point of computation onwards the server's behaviour does not depend on the handshake mode. Therefore these can be merged into the resulting s_1 state. With this approach, we can create simple rules that ensure the composability of the various protocol modes and closely follow the original specification. For example, the numbering of states (c_1, c_2, etc.) corresponds to message flights. In some cases, we require two rules to construct a single message flight, e.g., `C_2` and `C_2_Auth`, wherein a client optionally adds a signature to its final handshake message.

6.3.1.3 Examples of Complex Interactions

By defining the client and server rules as outlined above, we now have the ability to model the interaction of an unbounded number of interleaved handshakes. That is, while we express properties in terms of a specific client and a specific server, there may exist an unbounded number of other interacting agents, which the adversary may additionally compromise through possibly revealing their long-term keys. The adversary can then impersonate these agents, leading to an increase in the number of possible interactions.

Also, consider the following scenario: a client and a server have derived session keys after agreeing to use a PSK. We know that at some point the client must have authenticated the server (assuming the PSK is not from the out-of-band mechanism). However, we potentially need to resolve an unbounded number of handshakes before we arrive at the initial handshake in which the client verified the server's signature. The Tamarin prover allows us to reason inductively about such scenarios, facilitating the verification of important security properties that are typically out of reach of backwards unfolding.

Our full Tamarin model is available for inspection at [44].[9]

6.3.2 Encoding Security Properties

We now describe our threat model, and how we formally model the required TLS 1.3 security properties in Tamarin.

6.3.2.1 General Approach and Threat Model

Our aim is to analyse the core security properties of the TLS 1.3 protocol. Our work also considers all of the possible complex interactions between the various handshake modes. For these interactions, we prove both secrecy and authentication properties. Prior to our analysis, the work on TLS 1.3 generally considered handshake modes in isolation.

The threat model that we consider in our analysis is an active network adversary that can

[9] This is a stable URL containing links to our `draft-10` and `draft-21` source code, as well as supplementary material.

compromise the long-term keys of agents. In particular, we consider adversaries that can compromise the long-term keys of all agents after the thread under attack ends (to capture PFS) as well as the long-term keys of agents that are not the actor or the intended peer of the attacked thread, at any time (to capture Lowe-style MITM attacks and to contain the consequences of long-term key compromise).

Similarly to standard AKE models, our threat model has two components: the Tamarin rule that encodes the full capability (i.e., the ability to compromise an agent's long-term private key) and a restriction on the relevant security notion that prevents the adversary from compromising all the keys (corresponding to the fresh or clean predicates in AKE models [41]). We present the `Reveal_Ltk` rule here, and return to the restrictions when we describe the specific properties:

```
rule Reveal_Ltk:
  [ !Ltk($A, ~ltkA) ] --[ RevLtk($A) ]-> [ Out(~ltkA) ]
```

This rule can be triggered if a long-term private key `~ltkA` was previously generated for the agent `$A`. The right-hand side of the rule encodes that `~ltkA` is sent on the network, effectively adding it to the adversary's knowledge. Additionally, we log the action `RevLtk($A)`, which enables us to restrict this capability in the property specifications.

Our model describes the behaviour of the TLS 1.3 protocol in the presence of an active network adversary, and makes use of the standard perfect cryptography abstraction, as outlined in Section 6.2.2.1. This view simplifies the proofs and enables the analysis of many different security contexts. In the following two sections, we model and verify the required secrecy and authentication properties.

6.3.2.2 Secrecy Properties

We formally model and analyse two main secrecy properties. The first is the secrecy of session keys, implying perfect forward secrecy in the presence of an active adversary. The formal property we wish to verify is given in the `secret_session_keys` lemma.

Intuitively, the property requires that for all protocol behaviours and for all possible values of the variables on the first line (`All`) (1): if an authenticated session key `k` is accepted (encoded by the occurrence of the `SessionKey` action) (2), and the adversary has not

```
lemma secret_session_keys:
(1) "All actor peer role k #i.
(2)  SessionKey(actor, peer, role, <k, 'authenticated'>)@i
(3)  & not ((Ex #r. RevLtk(peer)@r  & #r < #i)
           |(Ex #r. RevLtk(actor)@r & #r < #i))
(4)  ==> not Ex #j. KU(k)@j"
```

revealed the long-term private keys of the actor or the peer before the session key is accepted (3), then the adversary cannot derive the key `k` (4).[10]

Our modelling of this property is very flexible: In the unilateral authentication mode, only the client establishes a session key with the flag `authenticated`. In the mutual authentication mode, both roles log this action. As we will see later, these actions are also suitable for the more flexible post-handshake client authentication modes that will be part of the final TLS 1.3 specification.

The second property that we prove is that the client's early data keys are secure as long as the long-term private key of the server is not revealed.

```
lemma secret_early_data_keys:
(1) "All actor peer k #i.
(2)  EarlyDataKey(actor, peer, 'client', k)@i
(3)  & not  (Ex #r. RevLtk(peer)@r)
(4)  ==> not Ex #j. KU(k)@j"
```

In particular, each time (1) that a client logs production of an early data key (2) and the peer's long-term private keys are not compromised (3), the adversary does not know the early data key (4).

6.3.2.3 Authentication Properties

We model authentication properties as agreement on certain values, such as agent identities and nonces. As discussed previously, this is a standard way of defining authentication [97]. The first property that we model is that when a client assumes there is a peer with whom it shares nonces, then this is actually the case:

[10] Our `draft-10` lemmas employ `KU` to denote adversary knowledge. Although functional, technically, this syntax is internal to Tamarin; the more suitable `K` should be used in externally exposed lemmas.

```
lemma entity_authentication:
(1) "All actor peer nonces #i.
(2)  CommitNonces(actor, peer, 'client', nonces)@i
(3)  & not (Ex #r. RevLtk(peer)@r)
(4)  ==> (Ex #j peer2.
(5)  RunningNonces(peer, peer2, 'server', nonces)@j
(6)  & #j < #i)"
```

In detail, this formula specifies that when the client logs that it has observed certain nonces at the end of its thread and believes it is communicating with a specific peer (1,2), and the long-term private key of this peer was not compromised (3), then there existed a thread (4) of that peer in the server role that agrees on the nonces (5) at an earlier point in time (6). Note that we would like to perhaps verify that `peer2` is equal to `actor`, but in the unilaterally authenticated mode, no such guarantee can be obtained.

The second property encodes that not only do the actor and the peer agree on the nonces and who the server is, they in fact agree on the complete transcript:

```
lemma transcript_agreement:
"All actor peer transcript #i.
 CommitTranscript(actor, peer, 'client', transcript)@i
 & not (Ex #r. RevLtk(peer)@r)
 ==> (Ex #j peer2.
 RunningTranscript(peer, peer2, 'server', transcript)@j
 & #j < #i)"
```

The above two properties only provide guarantees for the client, as in the main use case for TLS, where only the server is authenticated.

We now turn to the (optional) server guarantees. We have equivalent properties in the mutual authentication case, however, since both parties authenticate each other, we can achieve a stronger notion of authentication but with the restriction that the adversary cannot reveal either long-term key. The third property, encoded in the `mutual_entity_authentication` lemma, represents the authentication guarantee for the server, which can be obtained if the server performs the mutually authenticated handshake or requests client authentication (perhaps at a later time, as proposed for post-handshake client authentication).

```
lemma mutual_entity_authentication:
"All actor peer nonces #i.
 CommitNonces(actor, peer, 'server', nonces)@i
 & not ((Ex #r. RevLtk(peer)@r)
       |(Ex #r. RevLtk(actor)@r))
 ==> (Ex #j.
 RunningNonces(peer, actor, 'client', nonces)@j
 & #j < #i)"
```

The fourth property is analogous to the second, and ensures that the server obtains a guarantee on the agreement on the transcript with the client, after it has been authenticated.

```
lemma mutual_transcript_agreement:
"All actor peer transcript #i.
 CommitTranscript(actor, peer, 'server', transcript)@i
 & not ((Ex #r. RevLtk(peer)@r)
       |(Ex #r. RevLtk(actor)@r))
 ==> (Ex #j.
 RunningTranscript(peer, actor, 'client', transcript)@j
 & #j < #i)"
```

We note that all of the properties specified in our lemmas are trace properties, as is fairly typical in symbolic analysis.

6.3.3 Analysis and Results

6.3.3.1 Positive Results

Our model from Section 6.3.1 covers the many possible, complex interactions between the various handshake modes of TLS 1.3 for an unbounded number of sessions. When combined with the security properties in the preceding section, this gives rise to complex verification problems. Nevertheless, we successfully prove the main properties of TLS 1.3 (as given in Chapter 2, and above) and our results imply the absence of a large class of attacks, many of which are not covered by other analysis methods, i.e., attacks exploiting the interaction between various handshake modes. This is a very encouraging result, since it shows that the core design underlying `draft-10` is sound.

6.3.3.2 Proof Approach in Tamarin

Many of the security properties of TLS 1.3 stem from the secrecy of the shared secrets, i.e., the ephemeral secret (es) and the static secret (ss). Proving the secrecy of these components may seem simple; at its core, the main TLS 1.3 mechanism includes an authenticated Diffie–Hellman exchange and hence secrecy results should simply rely on standard Diffie-Hellman assumptions. However, complications arise due to the interactions between different handshake modes in an unbounded number of sessions and connections, and the possibility of powerful adversarial interference.

As a first step, it is necessary to prove a few fundamental invariant properties. These include properties which remain unchanged as the protocol progresses from one state to the next. A simple example of an invariant property in our model is captured by the following lemma:

```
lemma static_dh_invariant [reuse, use_induction]:
(1) "All actor x #i.
(2) UseServerDH(actor, x)@i
(3) ==> Ex #j. GenServerDH(actor, x)@j
(4) & #j < #i"
```

This lemma states that when an actor (which will be the server in our model) uses a semi-static Diffie-Hellman exponent as part of a semi-static server key share (1,2), then a corresponding generation of this semi-static exponent (3) took place prior to the exponent being used. Such properties help all future proofs by either reducing the number of contradictory dead-ends which the prover would otherwise explore, or by 'shortcutting' the proof by skipping a number of common intermediate steps. The property above, for instance, helps Tamarin to easily, and quickly, reason back to the source of the semi-static Diffie-Hellman exponent. The use_induction flag indicates to the Tamarin tool that it should use induction as its method of proof and the reuse flag indicates that this result may be used in the proving of all other lemmas going forward. In other words, the lemma is assumed to hold going forward.

In addition to simple lemmas such as these, the application of some inductive proofs is essential so as to avoid falling into the many infinite loops present. An example of an infinite loop in TLS 1.3 is the PSK-resumption loop – after an initial handshake has completed, a client may infinitely resume the established connection and the Tamarin

prover could get stuck in rolling backwards through this loop. We address this by proving lemmas, via induction, that ensure that a PSK-resumption handshake has its roots in an initial handshake, i.e., we enable Tamarin to quickly, and easily, find its origin.

Once problems such as looping have been resolved, simple auxiliary lemmas can be constructed. These lemmas help in piecing together more complicated proofs in a modular fashion. The auxiliary lemmas are sufficiently small, as well as sufficiently incremental, that they can be proved using Tamarin's fully automated functionality. Since each describes a small property which is likely to remain consistent throughout model changes, these auxiliary lemmas can be used to quickly check changes and reproduce proofs. The proofs for secrecy of `ss` and `es` follow from the auxiliary lemmas in a more manageable way than would otherwise be the case. For example, a common deduction uses the fact that knowledge of the resumption PSK implies that the adversary must also have knowledge of some (`ss`, `es`) pair from a previous handshake. This is captured in an auxiliary lemma, proven in Tamarin, and subsequently used in proving the secrecy of `ss` and `es` for the PSK-resumption case. The main burden of proof is, for all handshake modes, to unravel the client and server states to a point where either the adversary needs to break the standard Diffie-Hellman assumptions, or else secrecy follows from inductive reasoning.

Finally, the proof of session key secrecy then follows from the secrecy proofs for the `ss` and `es` values, which are both used as key derivation inputs. Using this approach, we successfully verify properties in Tamarin for the full interaction of the various handshake modes.

We note that the construction of the auxiliary lemmas and the proving of the secrecy of `ss` and `es` requires an intimate knowledge of TLS 1.3 and this part of the analysis is not a straightforward application of the Tamarin tool – considerable interaction with the tool is required so as to correctly guide it through the proof trees of the respective `ss` and `es` lemmas.

In total, our Tamarin analysis effort required roughly 6 person-months worth of work, including constructing the model, writing the lemmas, and subsequently proving the lemmas. We used a quad-core machine with 16GB of RAM. Of the lemmas which were auto-provable, the most complicated required approximately 20 minutes to complete. As a point of comparison, the STS lemmas prove in a matter of seconds.

6.3.3.3 Separation of Properties

One of the decisions made when specifying the security properties was to separate the secrecy and authentication requirements. Note that we could have equally combined both into a single property, as commonly defined in AKE models. The benefit of our approach is twofold: First of all, separating the properties results in a richer understanding of the security of the protocol. For instance, the structure of the proof confirms the intuition that the secrecy of session keys depends largely on the use of a Diffie-Hellman key exchange. The second benefit of this approach is that it provides a better foundation for future analysis. While our `draft-10` model considers the security of all handshake modes equally, there are some discrepancies in the guarantees provided by the various handshake modes. For example, if we were to allow adversarial compromise of semi-static secrets such as PSKs and server semi-static Diffie-Hellman exponents, we would discover that the secrecy of the static secret `ss` is not immediate in all handshake modes. Choosing to keep the properties separate allows for the development of a more nuanced security model, as will be shown to be the case in Chapter 7.

6.3.3.4 Implicit Authentication

In building the series of lemmas which lead to the final security properties, the most problematic areas coincided with the PSK handshake modes. In particular, the security of the PSK handshake relies on knowing that the resumption secret can only be known by a previous communication partner. This is an implicit authentication property. While we were able to overcome this challenge (by writing auxiliary lemmas to establish the source of the resumption secret) and eventually prove that this property holds, it does identify a potentially troublesome component to analyse. As we will see in the next section, the PSK mode certainly requires close attention.

6.3.4 Attacking Post-handshake Client Authentication

While `draft-10` does not permit certificate-based client authentication in PSK mode (and in particular in resumption using a PSK), we extended our model as specified in one of the proposals for this intended functionality [123]. By enabling client authentication

with a post-handshake signature over the most recent handshake hash (as dictated by the proposal), our TAMARIN analysis finds an attack. The result is a violation of client authentication, as the adversary can impersonate a client when communicating with a server.

6.3.4.1 The Attack

Our inability to prove the mutual authentication lemma in the post-handshake authentication case alerted us to a problem with this mechanism. After careful examination of the lemma sub-case, i.e., the proof search sub-tree, in which the property did not hold, we confirmed the attack.

Our attack is depicted in Figure 6.13 and we describe it in more detail here: Alice plays the role of the victim client, and Bob the role of the targeted server. Charlie is an active MITM adversary, whom Alice believes to be a legitimate server. In the interest of clarity, we have omitted message components and computations which are not relevant to the attack.

The attack proceeds in three main steps, each involving different TLS 1.3 mechanisms:

Step 1: Establish Legitimate PSKs. In the first stage of the attack, Alice starts a connection with Charlie, and Charlie starts a connection with Bob. In both connections, a PSK is established. At this point, both handshakes are computed honestly. Alice shares a PSK, denoted PSK_1, with Charlie, and Charlie shares a PSK, denoted PSK_2, with Bob. Note that Charlie ensures the session ticket (*psk_id*) is the same across both connections by replaying the *psk_id* value obtained from Bob.

Step 2: Resumption with Matching Freshness. In the next step, Alice wishes to resume a connection with Charlie using PSK_1. As usual, Alice generates a random nonce *nc*, and sends it to Charlie together with the PSK identifier, *psk_id*.

Charlie re-uses the value *nc* to initiate a PSK-resumption handshake with Bob, using the same identifier, *psk_id*. Bob responds with a random nonce *ns*, and the server `Finished` message computed using PSK_2.

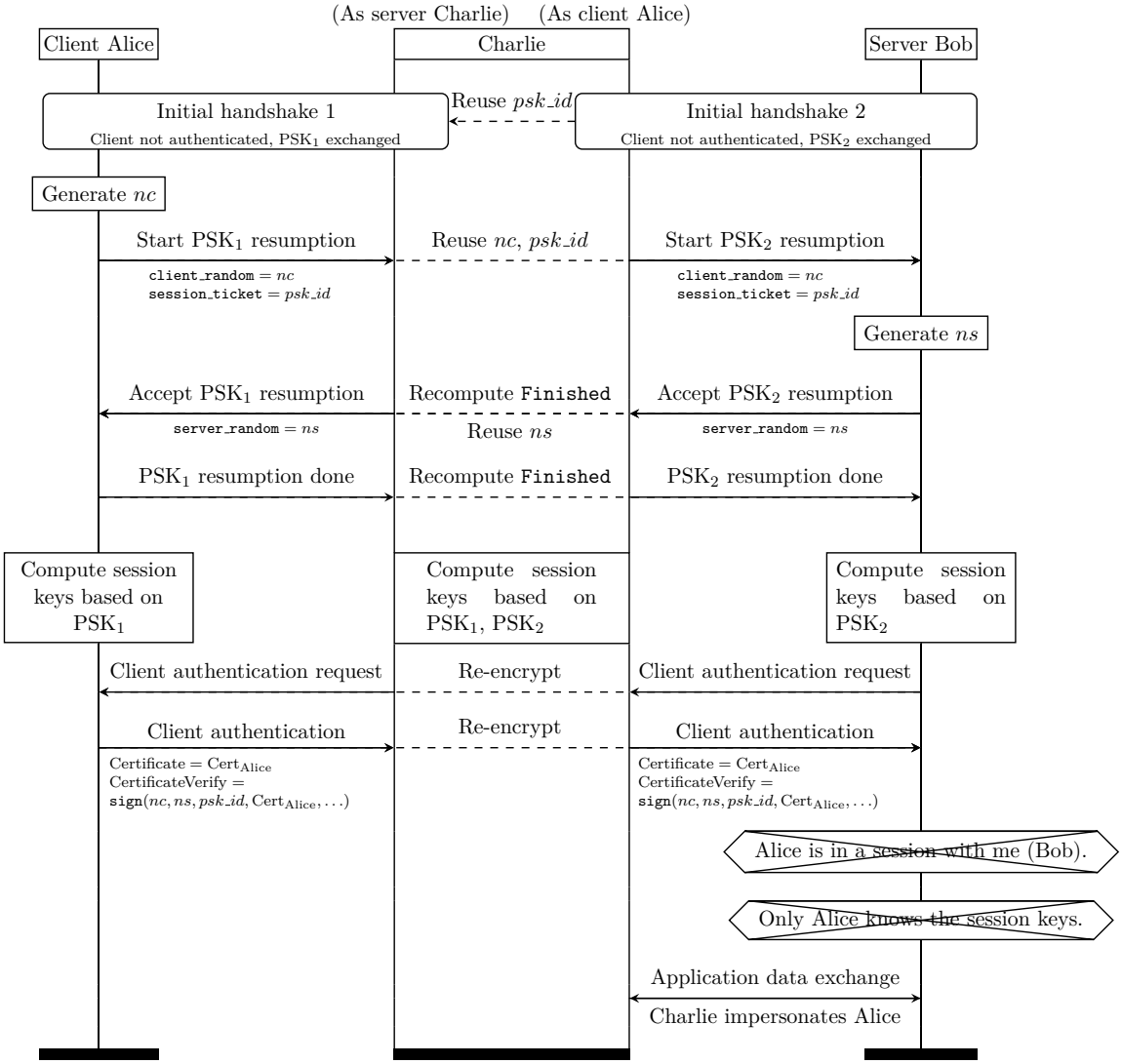


Figure 6.13: Client impersonation attack on TLS 1.3 if post-handshake client authentication is allowed in PSK mode (`draft-10`+). The attack exploits an initial handshake, a PSK-resumption handshake, and a post-handshake client authentication request.

Charlie now re-uses the nonce ns, and recomputes the server `Finished` message using PSK_1. Alice returns her `Finished` message to Charlie, who recomputes it using PSK_2.

At this point, Alice and Charlie share session keys (i.e., application traffic keys) derived from PSK_1, and Charlie and Bob share session keys derived from PSK_2. Note that the keys that Charlie shares with Alice and with Bob respectively, are distinct.

Step 3: Post-handshake Client Authentication. Following the resumption handshake, Charlie attempts to make a request to Bob over their established TLS channel. The request calls for client authentication, so Charlie is subsequently prompted for his certificate and verification.[11] Charlie re-encrypts this request for Alice.

In order to compute the verification signature, Alice uses the `session_hash` value, which is defined as the hash of all handshake messages excluding `Finished` messages. In particular, the session hash will contain nc, ns, and the session ticket psk_id.

Notice that this session hash will match the one of Charlie and Bob. Therefore, this signature will also be accepted by Bob. Hence, Charlie re-encrypts the signature for Bob, who accepts Alice's certificate and verification as valid authentication for Charlie.

Charlie has therefore successfully impersonated Alice to Bob, and has full knowledge of the session keys for both connections. This enables Charlie to impersonate Alice in future communication with Bob, allowing him to fake messages or to access confidential resources, and to violate the secrecy of messages that Bob tries to send to Alice. Thus, the attack completely breaks client authentication.

6.3.4.2 Underlying Cause and Mitigation

The above attack is possible due to the absence of a strong binding between the client signature and the connection for which it is intended. Therefore, the attacker is able to reuse the signature it receives to impersonate the client to a server on another connection. The second component of the attack is that the attacker is able to force the two resumption sessions to have matching transcripts.

[11] This is one of the main use cases for the post-handshake client authentication mechanism [123].

This suggests several potential ways to mitigate the attack. The most direct approach involves including the server certificate in the handshake hash. A similar mechanism is used in the 0-RTT case, where the server certificate is bound to the semi-static Diffie-Hellman share. However, this is not ideal because it complicates the out-of-band mechanism. Another potential option includes implementing an explicit authentication step as part of the PSK mechanism.

In parallel to our analysis, the TLS WG proposed several modifications to `draft-10` in the move towards `draft-11`. One of these proposals, PR#316 [133] (which takes a different approach to [123]), explicitly allows client authentication in the context that we analyse and redefines the client signature based on a new `HandshakeContext` value, which includes the server `Finished` message. Intuitively, this definition appears to address the attack because the adversary will need to force the `Finished` messages to match across the two sessions – a complicated, if not impossible, task as the `Finished` message is bound to the PSK, which is derived from a previously authenticated session (whether using certificates or out-of band mechanisms).

Discussions with members of the TLS WG revealed that the WG was previously not aware of the possibility of our attack, and the resulting strict necessity for a stronger binding between the client certificate and the security context that emerges from combining the PSK mode with post-handshake client authentication.

6.4 Conclusion

Even though symbolic analysis traditionally allows for the verification of less fine-grained security properties in comparison to computational analysis (i.e., trace vs equivalence properties), we argue that it is still of great benefit to the analysis of real-world protocols such as TLS 1.3.[12] Symbolic analysis tools such as the Tamarin prover allow for the automated verification of security properties, reducing burden on the proof process. This automated functionality also accommodates protocol changes more readily than computational analysis, allowing for the quicker checking of security properties after changes have been effected. A major strength of symbolic analysis tools is their ability to verify properties in the presence of protocol mode compositions, thereby offering an avenue

[12] And with ongoing work on enhancing observational equivalence proofs [20], this gap is closing.

for discovery of attacks exploiting the interaction of protocol components, as is clearly exhibited in the attack described in Section 6.3.4.

Tamarin's many features make it a good fit for the modelling and in-depth analysis of highly complex protocols such as TLS 1.3. In our analysis, the support for branching allowed us to model decisions that the protocol participants could make during execution, the looping support was instrumental in covering repeated resumptions within a single session, and the main security aspects of TLS 1.3 critically depend on Diffie-Hellman key exchange, meaning that we could leverage the excellent support for Diffie-Hellman in Tamarin. Also, the visualisations of attacks found by Tamarin provided us with a means of rapidly identifying potential problems, with either the protocol or our model – the graphical user interface proved a valuable asset in guiding our TLS 1.3 analysis.

The work presented in this chapter meaningfully contributes to the 'analysis-prior-to-deployment' design process adopted by the IETF in the development of TLS 1.3, and serves as an excellent example of the benefits of a pre-deployment analysis design paradigm. Our attack highlighted a flaw in the protocol prior to its official release, giving the TLS WG time to implement a fix for the problem before the flaw could be exploited by malicious agents in the wild. Also, our work brought into existence a symbolic model for TLS 1.3, meaning that changes to the protocol can be checked fairly quickly, a clear advantage for a rapidly moving target such as the TLS 1.3 design specification. In fact, our `draft-10` model serves as the basis for the work on `draft-21` described in the next chapter.

Chapter 7

Automated Analysis and Verification of `draft-21`

In this chapter we model and analyse `draft-21` *of the TLS 1.3 specification, reforming our* `draft-10` *model to incorporate the changes made to the specification since* `draft-10`*. Our work on* `draft-21` *establishes the security of several handshake mode interactions in TLS 1.3 for a near-final version of the protocol.*

7.1 Introduction

As has been discussed at length in the preceding chapters, there have been substantial efforts from the academic community in the areas of program verification – analysing implementations of TLS 1.3 [32, 47], the development of computational models – analysing TLS within Bellare-Rogaway style frameworks [54, 55, 61, 85, 91], and the use of formal methods tools such as ProVerif [4] and Tamarin [6], i.e., the work presented in Chapter 6, to analyse symbolic models of TLS [14, 27, 46, 75]. All of these endeavours have helped to both find weaknesses in the protocol and confirm and guide the design decisions of the TLS WG, as was displayed in the previous chapter.

The TLS 1.3 draft specification, however, has been a rapidly moving target, with large changes being effected in a fairly regular fashion. This has often rendered much of the analysis work outdated within the space of a few months, as large changes to the specification effectively result in a new protocol, requiring a new wave of analysis.

In this chapter we present a tool-supported, symbolic verification of a near-final draft of TLS 1.3, adding to the large effort by the TLS community to ensure that TLS 1.3 is

free of the many weaknesses affecting earlier versions, and that it is imbued with security guarantees befitting such a critical protocol. As of `draft-21`, many of the cryptographic mechanisms of TLS 1.3 have reached a stable state, and at the time of our analysis, we did not expect substantial changes to be implemented in subsequent drafts. And indeed, the current version of the specification, `draft-26`[1], does not incorporate any changes that affect the cryptographic logic of the protocol – the changes are largely implementation-specific.

Our main contributions in this chapter are as follows:

(i) Using our `draft-10` model as a foundation, we develop a symbolic model of `draft-21` of the TLS 1.3 specification that considers all the possible interactions of the available handshake modes, including the Pre-Shared Key (PSK)-based resumption and zero-Round Trip Time (0-RTT) exchanges. Its fine-grained, modular structure greatly extends and refines the coverage of our previous symbolic model. Our model effectively captures a new TLS 1.3 protocol, incorporating the many changes that have been made to the protocol since the development of our previous model.

(ii) We prove the majority of the specified security requirements of TLS 1.3, including the secrecy of session keys, perfect forward secrecy of session keys (where applicable), peer (entity) authentication, and Key Compromise Impersonation (KCI) resistance. We also show that after a successful handshake the client and server agree on identical session keys and that session keys are unique across handshakes.

(iii) We uncover a strange authentication behaviour that may lead to security complications in applications that assume that TLS 1.3 provides strong authentication guarantees. More specifically, we show that a client and a server may not agree on the authentication status of the client after the post-handshake authentication mechanism has been applied.

Related Work. As mentioned, there has been a great deal of work conducted in the complementary analysis spheres pertinent to TLS 1.3. Of most interest to this work are the symbolic analyses presented in [14], [27], and the previous chapter.

Since our `draft-10` analysis, there have been multiple changes made to the TLS 1.3 specification – 11 drafts-worth of changes to be precise. These updates have included

[1] Released on March 4th, 2018.

major revisions of the 0-RTT mechanism and the key derivation schedule. In `draft-10`, the sending of early data required a client to possess a semi-static (EC)DH value of the server. This particular handshake mode was removed and replaced by a PSK-based 0-RTT handshake mode – in `draft-21` early data can only be encrypted using a PSK. In fact, the PSK mechanism has been greatly enhanced since `draft-10`, with new PSK variants and binding values being incorporated into the specification. Post-handshake authentication was officially incorporated from `draft-11` onwards and a few drafts later, post-handshake authentication was enabled to operate within the PSK handshake mode. Another change to be incorporated after `draft-10` was the inclusion of 0.5-RTT data – the server being able to send fully protected application data as part of its first flight of messages.

All of these changes have resulted in what is effectively a very different TLS 1.3 protocol, particularly from a symbolic perspective. As a Tamarin model aims to consider the interaction of all possible handshake modes and variants, changes to these modes, as well as the inclusion of new post-handshake combinations, results in a very different set of traces to be considered when proving security properties. Hence, this work presents a substantially different model to the model presented in Chapter 6, with differences between the two models being highlighted in Section 7.3.1, and in addition to developing a new model, we follow a far more fine-grained and flexible approach to modelling TLS 1.3.

The work in [14] is an analysis of TLS 1.3 by the Cryptographic protocol Evaluation towards Long-Lived Outstanding Security (CELLOS) Consortium using the ProVerif tool. Announced on the TLS WG mailing list at the start of 2016, it showed the initial (EC)DHE handshake of `draft-11` to be secure in the symbolic setting. In comparison to our work, this analysis covers only one handshake mode of a draft that is now somewhat outdated.

The ProVerif models of `draft-18` presented by Bhargavan *et al.* in [27] include most TLS 1.3 modes, and cover rich threat models by considering downgrade attacks (both with weak cryptographic mechanisms and downgrade to TLS 1.2). However, unlike our work, they do not consider all modes, as they do not consider the post-handshake client authentication mode. While they cover relatively strong authentication guarantees (which led to the discovery of an unknown key-share attack[2]), their analysis did not uncover the

[2]In their attack, Bhargavan *et al.* show that a dishonest server, M, can convince a client, C, that it is connected to M, when indeed it is connected to a server S.

potential mismatch between the client and server views that we describe in Section 7.3.3.3.

7.2 Preliminaries

All of the preliminary material presented in Chapter 6, Section 6.2, is relevant here. We do, however, make shrewder use of the Tamarin tool. In our `draft-21` analysis, we make better use of what are known as *typing lemmas*. Tamarin's backwards search relies on the ability to identify the sources of rule premises, i.e., where facts on the left-hand side of rules come from. Typing lemmas greatly help to refine the origins of facts, aiding Tamarin's verification algorithm.[3] As with induction and reusable lemmas, typing lemmas are annotated with an instruction label, namely, `typing`. Further theoretical details concerning typing lemmas can be found in [107].

7.3 `draft-21` Analysis

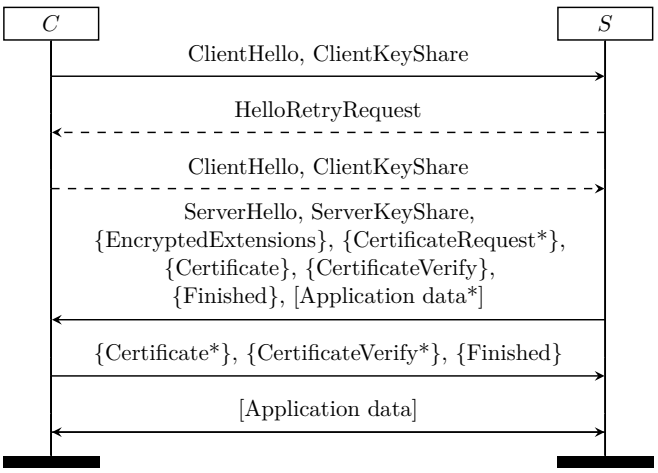
Using Tamarin's modelling framework we construct a comprehensive symbolic model of `draft-21` of TLS 1.3, capturing the specified protocol behaviours, as well as unexpected behaviours that might arise from the complex interaction of an unbounded number of sessions. Our model captures these behaviours in the presence of a powerful Dolev-Yao attacker.

Our aim is to verify the claimed security properties of TLS 1.3, as laid out in the `draft-21` specification. Details concerning the various handshake modes for `draft-21`, as well as the desired security properties, are given in Chapter 2; for ease of reference, we repeat the `draft-21` handshake diagrams (see Figure 7.1), as well as the security property descriptions, here:

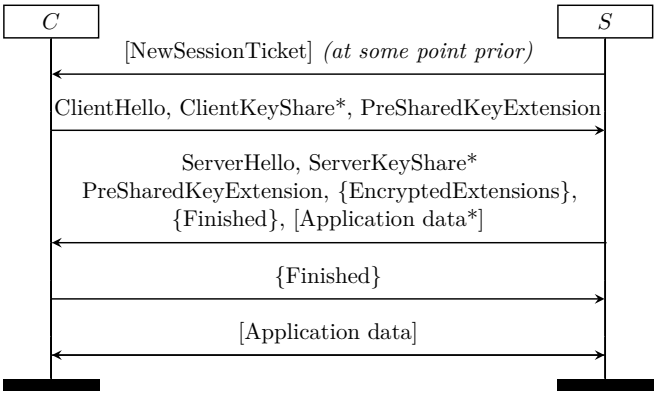
The TLS 1.3 `draft-21` security properties include:

1. **Establishing Identical Session Keys**. Upon completion of the handshake, the client and the server should have established a set of session keys on which they both

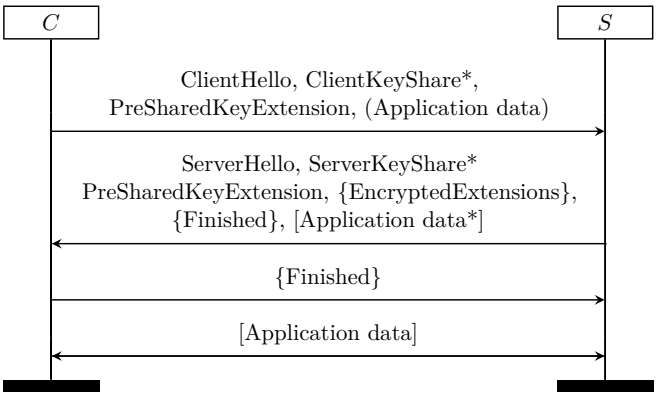
[3] In the latest version of Tamarin, typing lemmas are more aptly named *sources* lemmas.



(a) draft-21 (EC)DHE handshake



(b) draft-21 PSK resumption handshake



(c) draft-21 0-RTT handshake

Figure 7.1: Handshake modes for draft-21.

agree.

2. **Secrecy of Session Keys**. Upon completion of the handshake, the client and the server should have established a set of session keys which are known to the client and the server only.

3. **Peer (Entity) Authentication**. In the unilateral case, upon completion of the handshake, if a client C believes it is communicating with a server S, then it is indeed S who is executing the server role. An analogous property for the server also holds in the mutual authentication case. Authentication of the server is mandatory and mutual authentication is optional.

4. **Uniqueness of Session Keys**. Each run of the protocol should produce distinct, independent session keys.

5. **Downgrade Protection**. An active attacker should not be able to force the client and the server to employ weak cipher suites, or older versions of the TLS protocol.

6. **Perfect Forward Secrecy (PFS)**. In the case of compromise of either party's long-term key material, sessions completed before the compromise should remain secure. This property is not claimed to hold in the PSK key exchange mode.

7. **Key Compromise Impersonation (KCI) Resistance**. Should an attacker compromise the long-term key material of party A, the attacker should not be able to use this key material to impersonate an uncompromised party in communication with A.

8. **Protection of Endpoint Identities**. The identity of the server cannot be revealed by a passive attacker that observes the handshake, and the identity of the client cannot be revealed even by an active attacker that is capable of tampering with the communication.

In comparison to our draft-10 analysis, the changes to the TLS 1.3 specification which most affect our modelling effort include:

(i) New post-handshake mechanisms. In draft-21, new session tickets can be sent at *any* time. In draft-10 a new session ticket could only be sent after a full (EC)DHE handshake. draft-21 also includes a key update mechanism which can be employed at any time. As read and write keys differ, either party can immediately update

their write key after requesting a key update. This mechanism had not been finalised in draft-10. The post-handshake client authentication mechanism, with a fix to combat the weakness we uncovered, was officially included in the TLS 1.3 specification from draft-11 onwards and has remained in place.

(ii) A new 0-RTT mechanism. Our previous analysis considered a dedicated 0-RTT mechanism. In draft-21, the 0-RTT functionality has been rolled into a PSK handshake, i.e., early data can be sent by a client using a PSK. The use of a semi-static server key share is no longer an option.

(iii) PSK binders. The draft-10 specification did not include PSK binder values. These values bind PSKs to the handshake in which they are being used, as well as the handshake in which they were generated.

(iv) A new key schedule. In comparison to draft-10, draft-21 includes a key derivation schedule that is more streamlined and better suited to implementation. The differences between the respective key derivation schedules are detailed in Chapter 2.

As with our previous work, we analyse draft-21 of the TLS 1.3 specification using the Tamarin tool. Specifically, we (i) build an abstract model of the protocol, (ii) encode the desired security properties (as listed in the draft-21 specification), and (iii) construct proofs for the specified properties using the Tamarin verification algorithm.

Many TLS 1.3 analyses consider the constituent parts of TLS 1.3 in isolation, viewing these as separate protocols, and then proceed to tie the individual proofs together with a composability result. For instance, [27] considers the resumption mechanism as a separate protocol in which both the client and the server take as input a symmetric value – the PSK. If the PSK remains unknown to the attacker in every execution of the resumption protocol, a gap remains to be filled before concluding that the full series of handshakes always completes without the attacker knowing the PSK. This gap is filled by a manual composability proof. In our work, there is no need for such manual proofs; composition is automatically guaranteed by our comprehensive model, as Tamarin considers all possible component interactions in the proving of each property.

7.3.1 Building the Model

Although our model undoubtedly draws from the Tamarin model described in Chapter 6 of this thesis, here we opt to model TLS 1.3 with a significant increase in fidelity to the draft specification. Such an approach results in an improved ability to capture the full functionality of TLS 1.3, and hence broader coverage of realistic attacks, including complicated interaction attacks, such as the post-handshake client authentication attack discussed in Chapter 6. Additionally, by closely matching our model to the specification and allowing for an almost line-per-line comparison, we achieve full transparency regarding which parts of the specification we abstract away from, and the assumptions upon which our modelling process relies.

Not only is our model more comprehensive than the Tamarin model described in the previous chapter, it also incorporates the many changes to the TLS 1.3 specification that have materialised since the development of the `draft-10` model. In what follows, we describe the modelling process for `draft-21` and point out enhancements over the previous model.

7.3.1.1 Constructing Model Rules

As with our `draft-10` model, we employ the use of Tamarin rules to model state transitions within the TLS 1.3 protocol. However, our state transitions are far more fine-grained and modular in comparison to the model discussed in the previous chapter – we model the effective change in state as a result of transmission, receipt and processing of cryptographic parameters. For instance, a basic, initial TLS 1.3 handshake invokes up to 21 different rules and the associated state transitions before post-handshake operations can commence. Our `draft-10` model only invokes a maximum of 5 rules in transitioning through an initial handshake.

The `draft-21` initial handshake state transitions are depicted in Figure 7.2, and correspond to messages sent and the resultant cryptographic processing performed, as displayed in Figure 7.3. In comparison to the depiction of the basic, initial handshake in Figure 7.1a, the handshake displayed in Figure 7.3 represents message flows in a far more modular fashion, effectively breaking up the server's first message flight into three smaller 'sub-flights'.

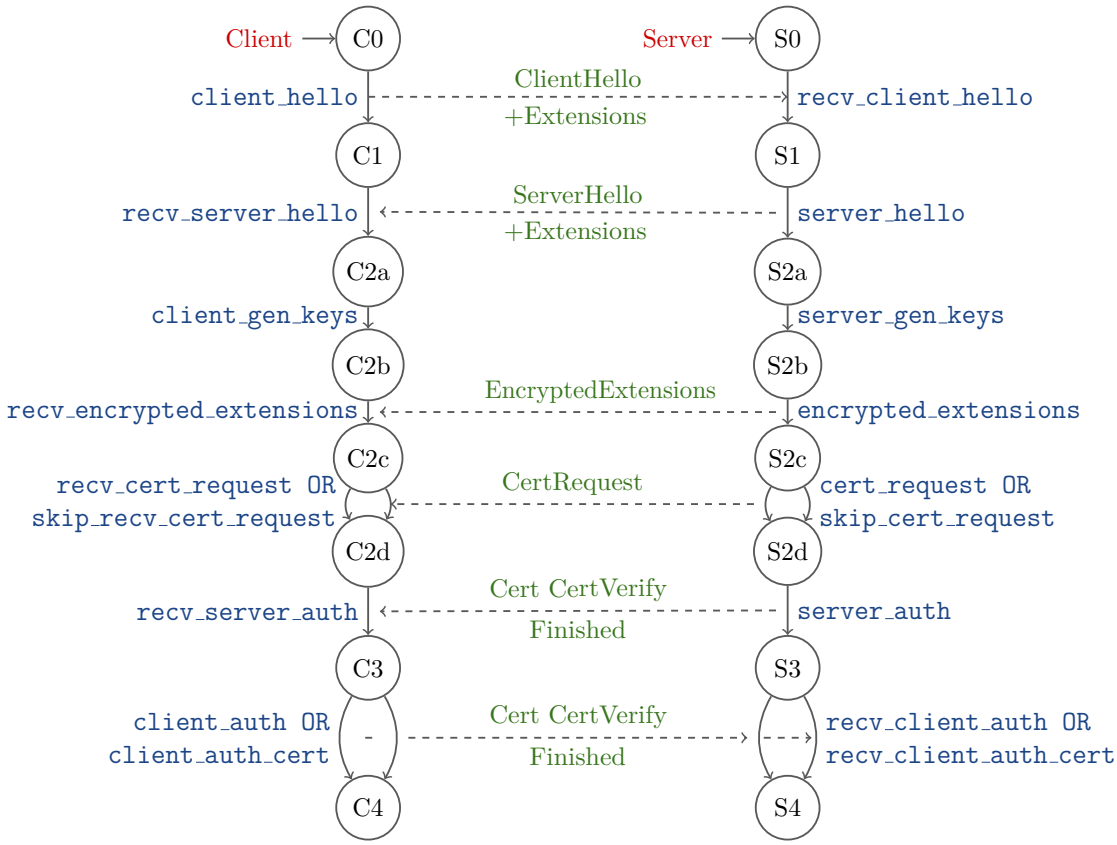


Figure 7.2: Partial state diagram for full TLS 1.3 handshake. TAMARIN rules are indicated in blue. The messages exchanged between entities are given in green.

Building our model according to this intended deconstruction of message flights allows for a more granular and flexible TAMARIN model.

In Figure 7.2, the transition from the the starting state, `C0`, is invoked by the firing of the first client rule, `client_hello`. This rule does not require any previous rule to trigger and moves the client into state `C1`, from which the next client rule, `recv_server_hello` can be triggered. In addition to the client needing to be in the correct state, the `recv_server_hello` rule also requires inputs (facts) that have been generated by the `server_hello` rule (this is marked in green in the diagram). Both server and client continue to transition according to the rules and message flights depicted in Figures 7.2 and 7.3 until such time as the initial handshake completes. We note that rule options that transition from `C2c`/`S2c` to `C2d`/`S2d` and from `C3`/`S3` to `C4`/`S4`, respectively, capture the modelling of unilateral versus mutual authentication; the server may request the client to authenticate in the initial handshake (`cert_request`), and the client should respond to this request (`client_auth_cert`). In the unilateral authentication case, the `skip_cert_request` and `client_auth` fire instead.

As an example of a model rule, we present our first client rule, `client_hello`, in Figure 7.4.

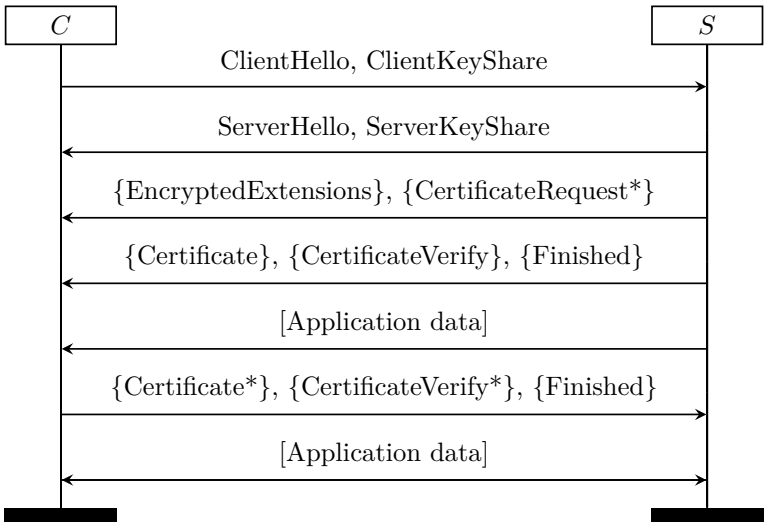


Figure 7.3: Full (EC)DHE handshake for `draft-21`, represented in sub-flights. The messages associated with a Hello Retry Request (HRR) have been omitted for the sake of simplicity.

As before, the `let...in` block allows for basic variable substitutions, and again we use the `tid` to name the newly created client thread (which we tie to the client nonce in the model). The `C0(tid)` action marks the occurrence of the `C0` state as a result of the `client_hello` rule being triggered, with its associated `tid`. The `Start(tid, C, 'client')` action signifies the instantiation of the client, `C`, in the role of `'client'`, and the `running_client` action indicates that client has initiated a run of the protocol. The `DH(tid, C, X)` action indicates the mapping of the private Diffie-Hellman exponent, `x`, to the client `C`. The `State(C1, tid, ...)` fact captures the local state of the thread `tid`, and is a rule output which will be consumed by the next applicable client rule, thereby transitioning the client thread to the next state, `C1`. The `DHExp` fact exists for consumption by adversary rules, if applicable.[4] The `Out` fact represents sending the `ClientHello` message to the network, after which it may become known to the adversary.[5]

We note the extensive use of macros in our model, enabled by the `m4` preprocessor, which allows us to cover most of the specification, whilst syntactically keeping our model close to it. For example, our `ClientHello` message is a macro that expands to:

[4]The need for this output fact will become clear in the discussion of our threat model in Section 7.3.2.1.

[5]The `HonestUse` actions now exist as an artefact in our model. They were constructed to help ensure that the intended Diffie-Hellman parameters were used as Diffie-Helman parameters only, and not misused by an adversary as nonces, or other message values.

```
rule client_hello:
let
    // Initialise state variables to zero.
    init_state()

    // Abstract client identity - does not currently correspond to
    // anything concrete.
    C = $C

    // Server identity - can be interpreted as the hostname.
    S = $S

    // Client nonce
    nc = ~nc

    // Reuse the client nonce to be a thread identifier.
    tid = nc

    // Group, DH exponent, key share
    g1 = $g1
    g2 = $g2
    sg = <g1, g2>
    client_sg = <g1, g2>
    g = g1
    x = ~x
    gx = g^x

    messages = <messages, ClientHello>
    es = EarlySecret
in
    [ Fr(nc),
      Fr(x)
    ]
  --[ C0(tid),
      Start(tid, C, 'client'),
      running_client(Identity, C),
      Neq(g1, g2),
      DH(tid, C, x),
      HonestUse(~x),
      HonestUse(gx)
    ]->
    [
      State(C1, tid, C, S, ClientState),
      DHExp(x, tid, C),
      Out(ClientHello)
    ]
```

Figure 7.4: First client rule, `client_hello`

```
handshake_record('1',
    ProtocolVersion,
    ClientRandom,
    '0', // legacy_session_id
    $cipher_suites,
    '0', // legacy_compression_methods
    ClientHelloExtensions)
```

which reflects almost exactly how this message is presented in the draft-21 specification.[6] ClientRandom is itself a macro, defined to be the value of the client nonce nc. ClientHelloExtensions is yet another macro which expands according to the following:

```
define(<!ClientHelloExtensions!>, <!<SupportedVersions,
NamedGroupList, SignatureSchemeList, KeyShareCH >!>),
```

again reflecting our intention of modelling the draft-21 specification as closely as possible, allowing for a direct syntactic comparison between our model and the specification. Our previous TAMARIN model also employs macros, but the connection to the specification is much less evident. For instance, in the draft-10 model, ClientHello is defined to be the pair of values nc,pc, representing the client's nonce and 'parameters', which serves as a placeholder for handshake values that are abstracted away.

In our model we have tried to define cryptographic components in a way that is reminiscent of imperative programming. As in the specification, we compute the handshake secret by computing the function HKDF-Extract(gxy,es), and the handshake keys are computed by applying a Derive-Secret (HKDF-Expand) function to this value. This is not strictly necessary due to the assumption of perfect cryptography – we could ignore the HKDF computations and focus solely on the secrets involved – but this makes it easier to connect our model to the specification.

The full state machine diagram of our draft-21 model can be viewed in Figures 7.5 and 7.6. Both figures display the post-handshake mechanisms discussed previously, i.e., new session tickets (new_session_ticket, recv_new_session_ticket), key updates (update_req_server, update_recv_client, update_fin_server and the corresponding client-initiated key update rules), and post-handshake authentication (certificate_request_post, recv_certificate_request_post, client_auth_post, recv_client_auth_post). We note that the post-handshake rules effectively loop back into the C4 and S4 states – for ease of representation, we represent these rules vertically in Figure 7.6.

[6] In TAMARIN's syntax, constants are enclosed by single quotes.

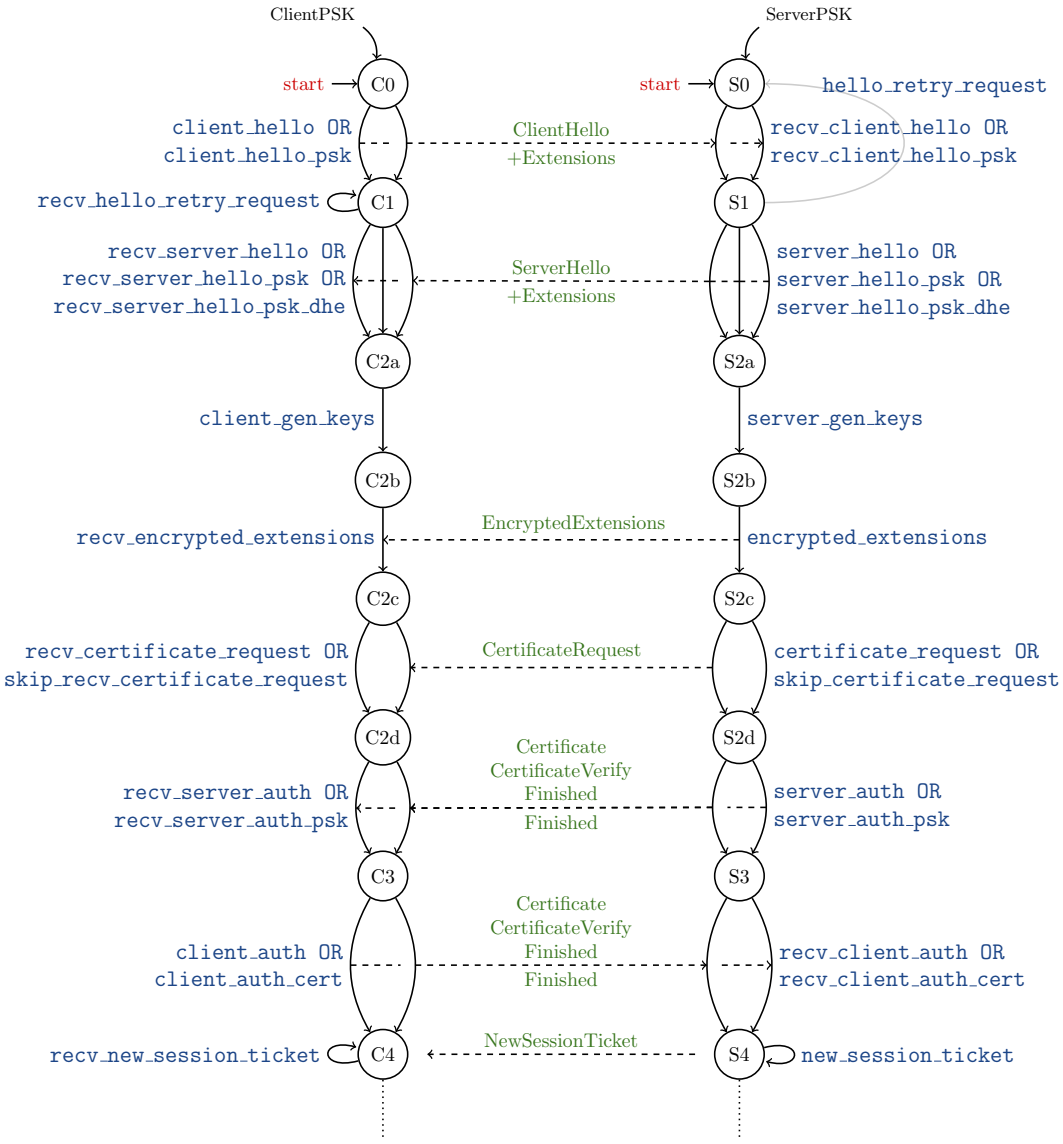


Figure 7.5: Part 1 of the full state diagram for our Tamarin model, showing all rules covered in the initial handshake (excluding rules dealing with the record layer).

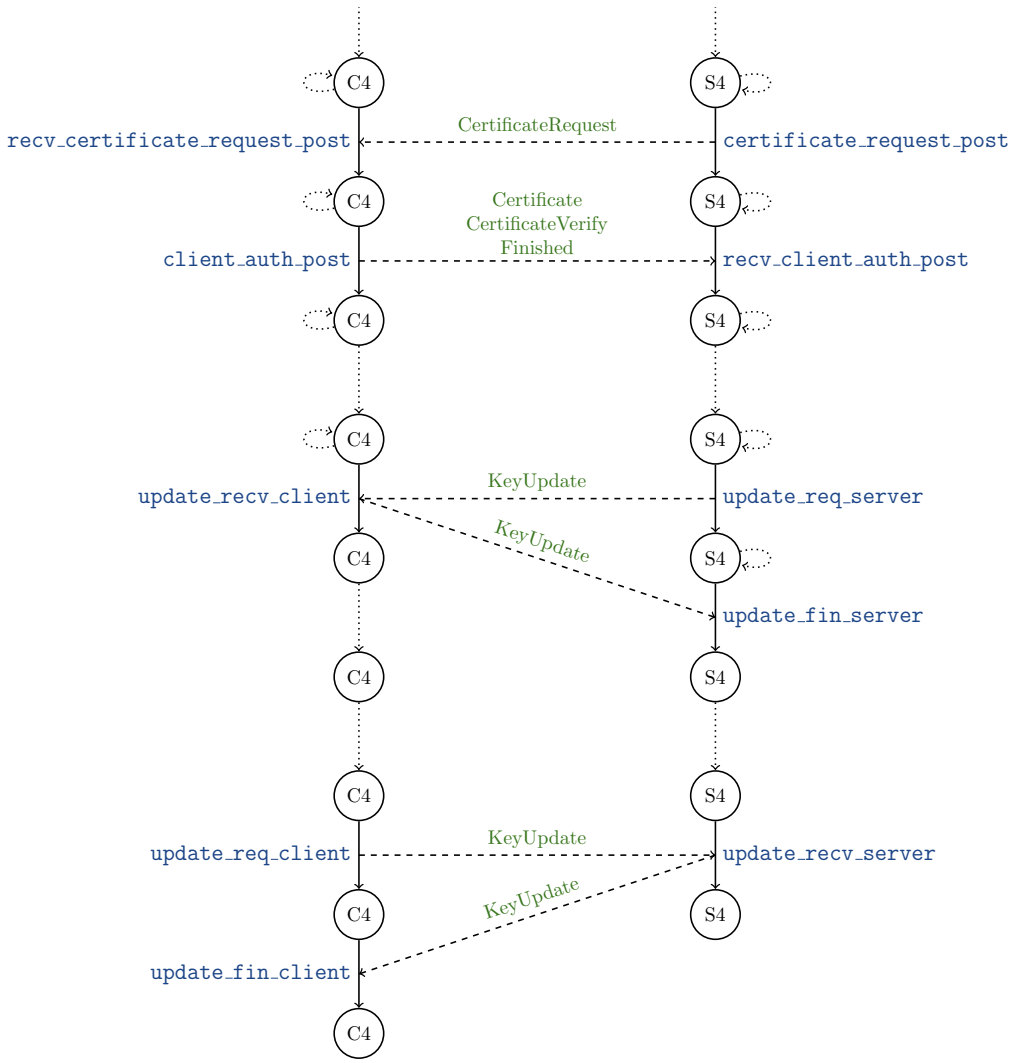


Figure 7.6: Part 2 of the full state diagram for our Tamarin model, showing all post-handshake rules covered.

For ease of readability, our state machine diagrams do not convey our modelling of the Record Protocol. In comparison to our draft-10 model, we model the Record Protocol more accurately in that we separate the Record Protocol rules from the Handshake Protocol rules, effectively modelling two different protocols, in accordance with the TLS 1.3 specification. Previously, we modelled the application data *send* and *receive* rules as looping back into Handshake Protocol states (see Figure 6.11 in Chapter 6). Whilst not incorrect, a more accurate approach would call for separation of the respective protocol states, and in our draft-21 model, we create Record Protocol states and applicable rules, which exist alongside the Handshake Protocol states and rules. This can be viewed in Figure 7.7. Note that this still allows for the Handshake Protocol to run over the Record Protocol, as is most definitely the case for the post-handshake messages as these are encrypted with record layer keys. One can imagine overlaying Figure 7.7 on top of Figures 7.5 and 7.6 to get a sense of our complete draft-21 model.

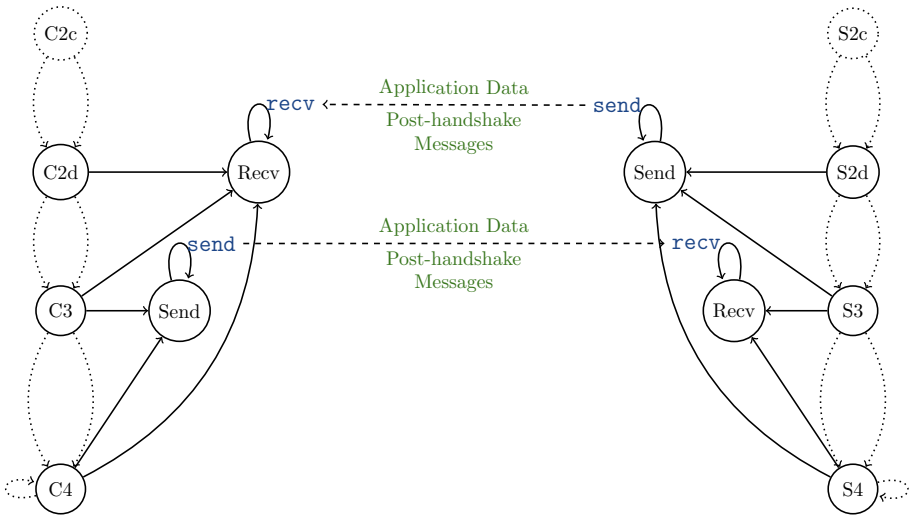


Figure 7.7: Record layer state diagram for TLS 1.3 draft-21. Record layer rules are indicated in blue. The messages exchanged between the client and the server, using distinct upstream and downstream traffic keys, are given in green.

7.3.1.2 Advanced Features

In our model we capture a number of complicated interactions and logic flows inherent to the TLS 1.3 handshake, greatly improving on our preceding model, adding features to the model which we consider to be 'advanced'.

Group Negotiation. We model the client and the server as having the ability, albeit limited, to negotiate the group used in the Diffie–Hellman key exchange. In Tamarin's syntax, variables that are always instantiated with public values are prefixed by `$`. In our model, the client starts with a pair of public values `$g1,$g2` that represent two supported groups, and offers these to the server along with a corresponding key share for `$g1`. Similarly, the server starts with a supported group `$g`. The model allows the server to return a `HelloRetryRequest` to the client, enforcing that `$g` is not equal to `$g1`, and expects the client to return instead a key share that matches `$g2`.

This interaction enables a much greater coverage of Diffie-Hellman key exchange with respect to our `draft-10` model. Previously, a `HelloRetryRequest` would only enable a client to pick a new Diffie-Hellman exponent for use with the same `$g`; we did not model negotiation of the Diffie-Hellman group.

Handshake Flows. One of the most complex elements inherent to modelling TLS 1.3 is the vast number of possible state machine transitions. After a session resumption, the server can choose between using the PSK only, or using the PSK along with a Diffie-Hellman key share. Alternatively, the server might reject the PSK entirely, and fall back to a regular (initial) handshake, or request that the client use a different group for the Diffie-Hellman exchange. Additionally, as discussed, there are several complex messages that can be sent in the post-handshake state: client authentication requests, new session tickets, and key update requests.

Since all of the above interactions can happen asynchronously, the resulting model becomes very complex and requires sophisticated handling logic. A number of complicated protocol flows, involving any number of sequential handshake modes and post-handshake extensions can, and will, transpire and we deal with this eventuality via our very modular approach to modelling, employing this strategy for all possible handshake modes.

Our full model is available at [44].[7]

[7] This is a stable URL containing links to our `draft-10` and `draft-21` source code, as well as supplementary material.

7.3.2 Encoding Security Properties

We describe our threat model and our formal modelling of the desired TLS 1.3 security properties within the Tamarin framework.

7.3.2.1 Threat Model

We consider an extension of the Dolev-Yao adversary as our threat model. As described in Chapter 6, Section 6.3.2.1, the Dolev-Yao adversary has complete control of the network, and can intercept, send, replay, and delete any message. In order to construct a new message, the adversary can combine any information previously obtained but our assumption of perfect cryptography implies that the adversary cannot encrypt, decrypt or sign messages without knowledge of the appropriate keys. In the previous chapter, we consider a Dolev-Yao adversary that has the ability to compromise the long-term keys of protocol participants. In order to consider different types of compromise, in this chapter, we additionally allow the adversary to compromise the PSKs of protocol participants (whether created out-of-band or through a new session ticket (NST)), as well as their Diffie-Hellman exponents. We model these additional adversary capabilities using the following Tamarin rules:

```
  rule Reveal_PSK:
   [ SecretPSK($A, res_psk)]--[ RevealPSK($A, res_psk) ]->[Out(res_psk)]

rule Reveal_DHExp:
 [ DHExp(~x, ~tid, $A) ]--[ RevDHExp(~tid, $A, ~x) ]->[ Out(~x), DHExp(~x,
     ~tid, $A)]
```

The `PSK_Reveal` rule can be triggered if a PSK (represented by `res_psk`) is generated for the agent `$A`. The right-hand side of the rule encodes that `res_psk` is sent out on the network, at which point it becomes adversary knowledge. The `Reveal_DHExp` rule can be triggered when a Diffie-Hellman exponent is generated for the agent `$A`, as is done in the `client_hello` rule in Section 7.3.1.1; the `DHExp(~x, ~tid, $A)` is produced by the `client_hello` rule and can thus be consumed by the `Reveal_DHExp` rule.

Our `Reveal_Ltk` lemma remains unchanged:

```
rule Reveal_Ltk:
  [ !Ltk($A, ~ltkA) ] --[ RevLtk($A) ]-> [ Out(~ltkA) ]
```

TLS 1.3 is not intended to be secure under the full combination of all of the types of compromise listed above. For example, session key secrecy can be broken by an adversary who eavesdrops on the communication and compromises the Diffie-Hellman values of a single protocol participant. A natural approach when encoding security properties is to weaken the attacker model by adding realistic constraints until either the claimed security goals of the protocol are achieved, or the corresponding attackers become weaker than the ones we expect to face in practice (in which case proving the property in question would amount to nothing meaningful). Hence, this approach requires us to express exactly what needs to be protected, and when each of the claimed TLS 1.3 security properties can be expected to hold.

We now present our Tamarin encodings of the TLS 1.3 security properties as given in the `draft-21` specification (and as discussed in Chapter 2, Section 2.5.4, and repeated in this chapter). In comparison to `draft-10`, the `draft-21` specification has a more developed overview of the desired TLS 1.3 security properties (see Appendix E of [128]). Hence, in addition to secrecy and authentication properties, we cover the other properties listed in the specification, namely, *uniqueness of session keys*, *identical session key establishment*, *downgrade protection*, and *Key Compromise Impersonation (KCI) resistance*.

7.3.2.2 Secrecy Properties

We now discuss the secrecy of session keys as well as perfect forward secrecy with respect to long-term keys:

Secrecy of Session Keys. Our `secret_session_keys` lemma, given below, is our encoding of property 2., the secret session keys property, given earlier.

```
lemma secret_session_keys:
(1) "All tid actor peer write_key read_key auth_status #i.
(2) SessionKey(tid, actor, peer, <auth_status, 'auth'>, <write_key, read_key>)@i &
(3) not (Ex #r. RevLtk(peer)@r & #r < #i) &
(4) not (Ex tid3 x #r. RevDHExp(tid3, peer, x)@r & #r < #i) &
(5) not (Ex tid4 y #r. RevDHExp(tid4, actor, y)@r & #r < #i) &
(6) not (Ex resumption_master_secret #r. RevealPSK(actor, resumption_master_secret)@r) &
(7) not (Ex resumption_master_secret #r. RevealPSK(peer, resumption_master_secret)@r)
(8) ==> not Ex #j. K(read_key)@j"
```

The intuition for this lemma is that if an actor believes it has established a session key with an authenticated peer (2), then the attacker does not know the key (8). However,

given the capabilities of the attacker, this will not hold without imposing some restrictions. This is why the additional clauses are required, (3) – (7).

The five adversary conditions stated in the depicted lemma are generally repeated across all lemmas, and encapsulate the basic assumptions we make about our attacker. We describe them in more detail here. The first imposes the restriction that the long-term signing key of the peer is not compromised. This restriction can additionally be understood to signify that the actor is communicating with an honest peer, since the attacker can effectively impersonate a party when in possession of its long-term key. Furthermore, it should be noted that the attacker is still allowed to compromise the peer's long-term key (LTK) *after* the session key is established. Hence we show that the session keys achieve perfect forward secrecy (PFS) with respect to the LTK.

The second and third clauses bar the attacker from revealing any Diffie-Hellman exponents generated by the client or the server, respectively, before the session key is established. The attacker may reveal exponents that are generated after the session key is established.

The last two clauses specify that the attacker cannot compromise a PSK associated with either the actor or the peer. Note that the attacker is restricted from revealing these PSKs at any time, i.e., before or *after* the session key has been established, which corresponds to the proviso in the specification that the PSK-only exchange mode does *not* provide PFS. We note that the PSK and Diffie-Hellman restrictions are necessary when considering the secrecy of session keys as the key derivation schedule (Section 2.5.1) employs both of these secrets, in the applicable handshake modes, to establish session keys.

Forward Secrecy with Respect to Long-term Keys. The PFS property is briefly touched upon above in the context of the long-term signing keys and the secrecy of session keys. However, in those cases, we do not cover the requirement for forward secrecy with regards to the PSK coupled with a Diffie-Hellman key exchange. In this instance, forward secrecy should be achieved, whereas with a PSK-only handshake, this is not guaranteed. We have an additional lemma, `secret_session_keys_pfs`, which captures that, in either a full DHE or PSK-DHE handshake, the secrecy of the session keys does not depend on the PSK remaining secret after the session is concluded.

In order to achieve this, we modify the `secret_session_keys` lemma depicted above by adding a condition for the key-exchange mode, `not psk_ke_mode = psk_ke` (i.e., not a PSK-only handshake), and loosening the restrictions on the attacker such that the `RevealPSK` action is only forbidden for any time point `#r < #i`. In proving this lemma we show that the session keys are forward secure after a Diffie-Hellman exchange.

7.3.2.3 Authentication Properties

Peer (Entity) Authentication. The specification defines this property, somewhat informally, as a form of authentication whereby both parties should agree on the identity of their peer. Again, looking at this more formally through the lens of Lowe's authentication hierarchy [97], this definition corresponds to weak agreement, i.e., the protocol guarantees to the client that when it completes a run of the protocol, apparently with the server, then the server was previously running the protocol, apparently with the client. In particular, we note that this property does not imply recentness – the requirement that the peer is currently running the protocol – nor does it specify whether any other values should be agreed upon.

We model entity authentication in two parts so as to capture the distinction between the mutual and unilateral authentication cases. Authentication in the unilateral case means that if a client completes a TLS handshake, apparently with a server, then the server previously performed a TLS handshake with the client, and they both agree on certain data values of the handshake, including the identity of the server and the nonces used. Note that this is already a stronger property than is stipulated in the specification. Here we also prove non-injective agreement[8] on the nonces, which additionally provides recentness since both parties contribute a fresh nonce to the handshake. The unilateral entity authentication lemma we aim to prove is the following:

```
lemma entity_authentication [use_induction, reuse]:
(1) "All tid actor peer nonces client_auth_status #i.
(2) CommitNonces(tid, actor, 'client', nonces)@i &
(3) CommitIdentity(tid, actor, 'client', peer, <client_auth_status, 'auth'>)@i &
(4) not (Ex #r. RevLtk(peer)@r & #r < #i) &
(5) not (Ex tid3 x #r. RevDHExp(tid3, peer, x)@r & #r < #i) &
(6) not (Ex tid4 y #r. RevDHExp(tid4, actor, y)@r & #r < #i) &
(7) not (Ex resumption_master_secret #r. RevealPSK(actor, resumption_master_secret)@r & #r < #i) &
(8) not (Ex resumption_master_secret #r. RevealPSK(peer, resumption_master_secret)@r & #r < #i)
(9) ==> (Ex tid2 #j. RunningNonces(tid2, peer, 'server', nonces)@j & #j < #i)"
```

[8]The client and the server agree on the nonces but there is no guarantee that there is a one-to-one relationship between the protocol runs of the client and the protocol runs of the server.

The intuition for this lemma is that if a client believes it has agreed on a pair of nonces with a server, (2) and (3), then the server was, at some point prior, engaged in a run of the protocol using these nonces (9). We again impose the necessary restrictions on the attacker to achieve this property, (4) – (8). The property can only hold if the attacker does not acquire any of the secrets prior to the client agreeing on nonces. While one might expect that only the legitimacy of the signing key is necessary for authentication, if the attacker is able to obtain the PSK then the attacker is able to resume a session and impersonate the peer.

We encode mutual entity authentication with an additional lemma that is almost identical to the entity authentication lemma given above – it is written with the server as the actor, i.e., all `'client'` instances are replaced with `'server'` instances. The two lemmas individually capture authentication of the server and the client, respectively, and together capture mutual entity authentication.

In addition to entity authentication, we consider a transcript agreement property, where the value agreed upon is a hash of the session transcript. This provides us with near-full agreement[9] – there are a couple of notable omissions. Firstly, the Handshake Protocol continues after the initial exchange of messages in the handshake, although none of these delayed, post-handshake messages are included in the session transcript. Secondly, we observe that the actors do not necessarily agree on the current authentication status of the handshake, an oddity we cover in more detail in Section 7.3.3.3.

Finally, we also aim to prove an injective variant of mutual transcript agreement (which TLS should naturally achieve by agreeing on fresh nonces). Hence, we aim to show that TLS achieves a relatively strong authentication notion: mutual agreement on a significant portion of the state with recentness.

7.3.2.4 Additional Properties

We now discuss our encoding of the remaining TLS 1.3 properties listed in Chapter 2, and repeated above.

[9] We recall that full agreement captures the notion that both protocol participants agree on all possible data items that could have been exchanged, or created as a result of the exchange, during a run of the protocol, and that there is a one-to-one relationship between the protocol runs.

Identical Session Keys. The definition of this property as given in Section 2.5.4 (and above) is taken from [41], where it is referred to as a consistency property. However, there is ambiguity in the circumstances that are necessary and sufficient for two protocol participants to establish the same keys. An answer to this question is typically given through the well-established practice of defining *session partnering* [23,41,92]. One possible way to do so is to assign session identifiers in terms of a value (or pair of values) on which the two parties agree. We opted for the least restrictive session identifier, namely the pair of nonces generated by the client and the server. Therefore, if a *partnered* client and server complete the handshake, then they must agree on session keys.

We consider this property with respect to an attacker that can compromise all session keys except for those of the test session, i.e., the session in which the attacker attempts to obtain information about the key [41, 92]. This property is captured by our lemma `session_key_agreement`:

```
lemma session_key_agreement:
(1) "All tid tid2 actor peer actor2 peer2 nonces keys keys2 cas as2 #i #j #k #l.
(2) SessionKey(tid, actor, peer2, <cas, 'auth'>, keys)@i &
(3) running(Nonces, actor, 'client', nonces)@j &
(4) SessionKey(tid2, peer, actor2, as2, keys2)@k &
(5) running2(Nonces, peer, 'server', nonces)@l &
(6) not (Ex #r. RevLtk(peer)@r & #r < #i & #r < #k) &
(7) not (Ex tid3 x #r. RevDHExp(tid3, peer, x)@r & #r < #i & #r < #k) &
(8) not (Ex tid4 y #r. RevDHExp(tid4, actor, y)@r & #r < #i & #r < #k) &
(9) not (Ex rms #r. RevealPSK(actor, rms)@r & #r < #i & #r < #k) &
(10)not (Ex rms #r. RevealPSK(peer, rms)@r & #r < #i & #r < #k)
(11)==> keys = keys2"
```

Intuitively, this lemma states that if two actors have partnered with the same nonces, (3) and (5), then they have established identical session keys (11).

Uniqueness of Session Keys. In order to capture the uniqueness of session keys, we aim to prove that any two matching session keys generated must be from the same session, and indeed from the same TLS connection. In other words, each run of the protocol produces distinct session keys. This should hold without any restriction on the attacker, since it is a straightforward consequence of an actor generating a fresh nonce for each session (since nonces are included in the computation of traffic keys). The property as listed in the `draft-21` specification also mentions that session keys should be independent. We do not aim to prove anything about the independence of session keys, or whether or not two session keys are related, since this trivially follows from our assumption of perfect cryptography. We capture the uniqueness property in our `Unique_session_keys` lemma:

```
lemma unique_session_keys:
"All tid tid2 actor peer peer2 kr kw as as2 #i #j.
SessionKey(tid, actor, peer, as, <kr, kw>)@i &
SessionKey(tid2, actor, peer2, as2, <kr, kw>)@j
==> #i = #j"
```

Downgrade Protection. The draft-21 specification cites the work by Bhargavan *et al.* [28] for a definition of downgrade protection, as given in Section 2.5.4 and above. This definition is not directly equivalent to any of Lowe's classical agreement definitions; it only requires that both parties negotiate the *same* configuration parameters as would be the case without the presence of an attacker. Specifically, we observe that agreeing on the configuration parameters (in the sense of non-injective agreement) is sufficient to achieve this, but not necessary. Therefore, *within our model* we capture downgrade protection through our authentication lemmas. However, we note that this does not accurately capture the spirit of downgrade protection owing to the fact that we assume all cryptographic primitives to be perfect, and given that we do not model previous versions of TLS. The weakening of primitives, as well as the modelling of previous versions of TLS in Tamarin, would make for interesting future work.

Key Compromise Impersonation (KCI) Resistance. The only restriction we place on compromising long-term keys is that the *peer's* LTK must not be compromised. None of our security properties rely on the *actor's* LTK being hidden from the attacker.[10] This fact coupled with our authentication properties additionally captures the KCI resistance property as laid out in Section 2.5.4 (and above).

7.3.2.5 Parameter Negotiation

The security of a TLS session critically depends on the integrity of the parameters negotiated during the corresponding TLS handshakes in the initial and subsequent connections in the session. In TLS these parameters include the protocol version, the cipher suite, and the signature algorithm. Depending on the negotiated protocol version, additional values may or must be negotiated, such as the handshake mode, the Diffie-Hellman group, and/or the PSK to be used.

[10] A minor exception to this is that the adversary cannot use the actor's long-term key to impersonate the actor to itself since in this case, the actor is also the peer.

For Diffie-Hellman group negotiation, we model the client sending a list of two symbolic groups from which the server may choose. This feature allows us to provide a limited coverage of the Hello Retry Request (HRR) functionality of the protocol. We also provide support for PSKs but limit the number of PSKs offered to one per handshake.

With our transcript agreement lemmas, we capture the client and the server agreeing on the transcript of the protocol and hence on the values selected during negotiation. This means that an adversary should not be able to force the client or the server to accept a value that they did not initially offer.

There are two main classes of parameter negotiation attacks: forcing the use of bad cipher suites [8, 34] or bad signature algorithms [26], and forcing the use of older and insecure versions of SSL/TLS [16, 110]. As mentioned, because we model perfect cryptography and cover TLS 1.3 only, these attacks are not captured by our analysis.

7.3.3 Analysis and Results

We now provide a detailed description of our analysis, including a discussion of our results and an exploration of an authentication anomaly uncovered by our work.

7.3.3.1 Positive Results

In general, we find that TLS 1.3 meets the properties outlined in the `draft-21` specification, those that our modelling process was able to capture at least. We show that TLS 1.3 enables a client and a server to agree on secret session keys and that these session keys are unique across, as well as within, handshake instances. Our analysis shows that PFS of session keys holds in the expected situations, i.e., in the (EC)DHE and PSK+(EC)DHE handshake modes. We also show that TLS 1.3, by and large, provides the desired authentication guarantees in both the unilateral and mutual authentication cases. The situation in which this is not the case is covered in Section 7.3.3.3.

We present our results in Table 7.1. For each property discussed in Section 2.5.4 (and above), we indicate our findings. We use $*$ to indicate that the property holds in most

situations. As mentioned, we discuss cases in which the property does not hold to the expected degree in sections to follow. We also list our applicable Tamarin lemma(s) in Table 7.1.

Property Proven	Lemma(s)
(1) Secret session keys	secret_session_keys
(2) Perfect forward secrecy	secret_session_keys_pfs
(3) Peer authentication*	entity_authentication mutual_entity_authentication
(4) Identical session keys	session_key_agreement
(5) Unique session keys	unique_session_keys
(7) Key Compromise Impersonation (KCI) resistance	entity_authentication mutual_entity_authentication

Table 7.1: TLS 1.3 draft-21 Tamarin results

We recall that our model does not truly cover downgrade protection. A treatment of downgrade protection across TLS protocol versions would require modelling the earlier versions of TLS in a way that is consistent with the TLS 1.3 model as developed here. In order to consider the downgrade protection of cipher suites, we would need to relax our current assumption of perfect cryptography through rules that, for instance, allow an attacker to learn the payload of encrypted messages without knowing the key. In spite of the fact that these additional considerations would substantially complicate the model and the proof process, our model is well-suited to their inclusion and could form the basis of future work.

7.3.3.2 Proof Approach in Tamarin

For models as complex as those needed to capture TLS 1.3, proving lemmas in Tamarin is a multi-stage process, and proving complex lemmas directly is often infeasible as proof trees can become very large. Tamarin provides a number of features that allow complex proofs to be broken down into more manageable sections. Writing auxiliary lemmas, or sub-lemmas, provides hints to the Tamarin constraint solving algorithm, allowing it to solve complex sections of a larger proof directly, making the overall proof more manageable. For our draft-21 model, we use several types of lemmas: Helper lemmas are used to quickly solve repetitive subsections of a larger proof without repeatedly unrolling the entire sub-tree (corresponding to the repetitive subsection), and typing lemmas provide hints to the Tamarin engine about the potential sources of messages, reducing the branching of

a proof tree. Inductive lemmas instruct Tamarin to prove lemmas inductively, allowing us to break out of loops in the protocol, which otherwise can produce infinite proof trees. Proving the main properties of draft-21 required many auxiliary lemmas, of all of these kinds.

The Tamarin engine can also use heuristics to automate the proving of lemmas, which proved invaluable in quickly re-proving large sections of properties after making changes to the model. By investing time in writing auto-provable sub-lemmas, we could easily incorporate changes made to the specification without having to restart our analysis from scratch.

The more complex lemmas used in our analysis of draft-21, however, required manual proving in the Tamarin interactive prover. We note that by manual proving in this context we mean manually guiding the Tamarin prover through a proof by using the Tamarin graphical user interface.

In order to give an indication of the number of auxiliary lemmas required, and the relationship between all of our lemmas, we have constructed a lemma map, displayed in Figure 7.8. The map also indicates which lemmas were auto-proved by Tamarin, and which ones needed manual guidance for Tamarin to be able to prove them.

In total, the analysis effort represents approximately 3 person-months worth of work. However, the vast majority of that time can be allocated to the process of writing lemmas to break down the overall proving effort into smaller, auto-provable chunks. With these lemmas in place, proving the entire model takes about a week, and significant computing resources. We made use of a 48-core machine with 512GB of RAM. The model itself takes over 10GB RAM to load, and can easily consume 100GB RAM in the course of a proof. The underlying language on which Tamarin is built, Haskell[11], automaticially detects the number of cores available on a machine and employs as many threads, using multi-treading to speed up the proof search. Once produced, our proofs can be verified within the space of a day, still requiring vast amounts of RAM.

[11] See https://www.haskell.org/.

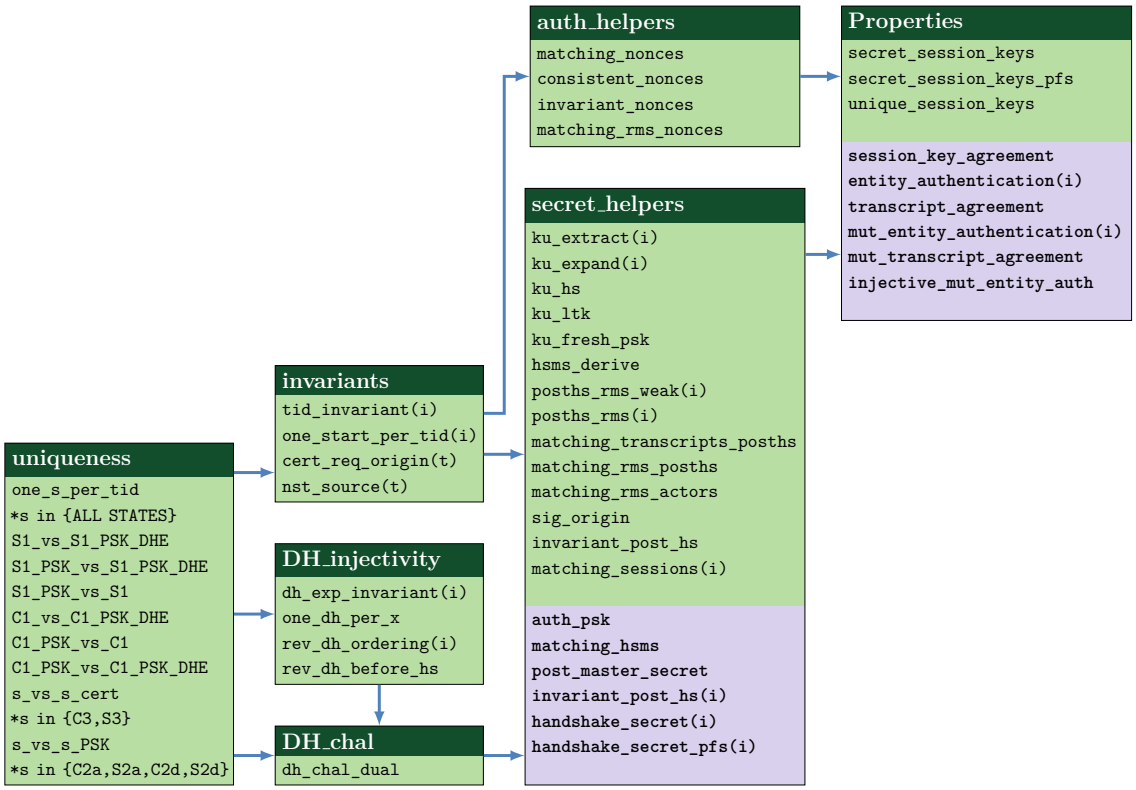


Figure 7.8: Lemma Map. Lemma names with a purple background indicate where manual interaction via the Tamarin visual interface was required. The remaining lemmas were automatically proven by Tamarin, without manual interaction. An arrow from one category to another implies that the proof of the latter depends on the former, and `(i)` and `(t)` indicate inductive and typing lemmas, respectively. The **Properties** box contains the main TLS 1.3 properties.

7.3.3.3 Authentication Mismatch

During the development of our `draft-21` model, and in particular the analysis of the post-handshake client authentication, we encountered a possible behaviour that suggested that TLS 1.3 fails to meet certain strong authentication guarantees. While there are many definitions of authentication, the common thread among strong authentication guarantees is that both parties share a common view of the session, i.e., that they agree on exchanged and computed data. During our analysis of the post-handshake client authentication mechanism, it became apparent that the client does not receive any explicit confirmation that the server has successfully received the client's response. Due to the asynchronous nature of this post-handshake client authentication, the client may keep receiving data from the server, and will not be able to determine if the server has received its authentication message. As a consequence, the client cannot be sure as to whether or not the server has sent subsequent data under the assumption that the client has been authenticated.

In concurrent work by Bhargavan *et al.* [27], a similar issue was uncovered for the 0.5-RTT case. A discussion with the TLS WG on draft-21 revealed that an equivalent problem also exists within the main (initial) handshake. During the main handshake, the server can request a client certificate, and may decide to reject it (for example because it violates certain domain-specific policies), *but still continue with the connection* as if the certificate was accepted. Therefore, the client cannot be sure (after what appears to be a main handshake with mutual authentication) that the server considers the client to be authenticated. Thus, this phenomenon leaves the client uncertain as to whether or not the server considers it to be authenticated, even though (i) the server asked for a certificate, and (ii) the client supplied it and the corresponding authentication signature.

In order to see why this may become a problem at the application level, consider the following application: Assume a TLS 1.3 enabled client and server pair where the server implements the policy that any data received over a mutually authenticated connection are stored in a secure database; all data received over connections where the client is not authenticated are stored in an insecure log. The client connects to the server, the server requests a certificate (and the corresponding authentication signature), which the client duly provides, but the server rejects the client's certificate (and hence the authentication signature) and continues regardless. Since the server rejected the client's certificate, it continues to store incoming messages in the insecure log. However, the client may assume it has been authenticated, and therefore may start sending sensitive data, which is consequently stored in the insecure log, rather than in the secure database as the client may expect.

The TLS WG has decided not to fix this behaviour for TLS 1.3, meaning that this remains true for all subsequent TLS 1.3 drafts, and has not introduced any mechanism that informs the client of the server's view of the client's authentication status. If a client wants to be certain that the server considers it to be authenticated, this confirmation needs to be supplied at the application layer. We anticipate that some client applications will incorrectly assume that sending a client certificate (with the associated authentication information) and obtaining further server messages indeed guarantees that the server considers the connection to be mutually authenticated. As stated, this is not the case in general, and may lead to serious security issues despite there being no direct violation of the specified TLS 1.3 security requirements.

7.4 Conclusion

In this work we modelled `draft-21` of the TLS 1.3 specification within the symbolic analysis framework of the TAMARIN prover, and used the tool to verify the majority of the security guarantees that TLS 1.3 claims to offer.

We focus on ruling out complex interaction attacks by considering an unbounded number of concurrent connections, and all of the TLS 1.3 handshake modes. We cover both unilateral and mutual authentication, as well as session key secrecy in all of the TLS 1.3 handshake modes with respect to a Dolev-Yao attacker. We also capture more advanced security properties such as perfect forward secrecy and Key Compromise Impersonation resistance. Our `draft-21` TAMARIN modelling effort embraces a far more modular approach, accurately capturing the draft specification.

Besides verifying that `draft-21` of the TLS 1.3 specification meets the claimed security properties in most of the handshake modes and variants, we also discover an unexpected authentication behaviour which may have serious security implications for implementations of TLS 1.3. This unexpected behaviour, at a high level, implies that TLS 1.3 provides no direct means for a client to determine its authentication status from the perspective of a given server. As a server may treat authenticated data differently to unauthenticated data, the client may end up in a position in which its sensitive data gets processed as non-sensitive data by the server.

The work presented in this chapter again contributes to the newer, proactive TLS 1.3 design approach, confirming the TLS WG's design choices, as well as pointing out weaknesses in the protocol which may lead to complications in the future.

Part IV

Concluding Remarks

Chapter 8

Conclusion

In this chapter we conclude and briefly mention avenues for future work.

In the development of the latest version of TLS, namely TLS 1.3, the IETF has embraced a more collaborative and proactive design process, welcoming analyses of the protocol by the academic community prior to the protocol's official release with a view to catch and remedy flaws before the protocol is finalised. This thesis has examined the reasons for the IETF's decision to adopt a new development process, and has presented work that supports this design shift.

In **Chapter 3** we presented an account of TLS standardisation, starting with the early versions of TLS, right up until TLS 1.3, which is, at the time of writing, nearing completion. We described how the process for TLS 1.2 and below fits the design-release-break-patch cycle of standards development, and how a shift in the process has resulted in the standardisation of TLS 1.3 conforming to the design-break-fix-release development cycle. We commented on the factors that have influenced the change in the TLS WG's design methodology, namely, the protocol analysis tools available, the levels of involvement from the research community, and the incentives driving the relevant stakeholders.

In **Chapters 4 and 5** we presented work that attacks the use of RC4 in TLS 1.2 and below, showing that the Record Protocol does not meet its intended confidentiality goal. In Chapter 4 we showed how to recover passwords when protected by RC4 in TLS, and in Chapter 5 we attacked 16-byte plaintexts protected by RC4, introducing techniques that enable predictions regarding the number of ciphertexts needed for the successful recovery of target plaintexts. As mentioned in Chapter 4, avenues for future work include adapting our password-recovery attacks to operate with an unknown password length, and examining

the effect of dictionary choice on attack success rates in more detail.

The material presented in Chapters 4 and 5 adds to a large body of work that confirms the need for a new version of TLS – a version that has undergone analysis *prior* to release. In **Chapters 6 and 7** we presented work that contributes towards this goal. We analysed two drafts of TLS 1.3, namely `draft-10` and `draft-21`, using the TAMARIN prover . In Chapter 6 we showed that the `draft-10` Handshake Protocol meets its desired security properties but we found an attack against the post-handshake client authentication mechanism. This attack informed the next revision of TLS 1.3 which indeed implemented a patch for this attack. In Chapter 7 we showed that `draft-21`, by and large, meets its intended security goals. We did, however, observe a strange authentication behaviour in the post-handshake client authentication mechanism which could have security implications for real-world systems. This behaviour was reported to the TLS WG.

Our symbolic analyses in Chapters 6 and 7 employed the use of perfect cryptography. One possibility for extending our analyses is to weaken cryptographic primitives by introducing rules which, for instance, allow an adversary to create hash function collisions, or signature and MAC forgeries. Another area of future exploration includes incorporating TAMARIN models for TLS 1.2 and below so as to fully cover downgrade protection. This, however, could prove to be computationally expensive.

The newer standardisation process followed for TLS 1.3 by the IETF exhibits benefits over the process employed previously as it allows for the pre-emptive detection and fixing of weaknesses, thus producing a potentially stronger protocol and reducing the need for patches post-release. A proactive process such as this is truly befitting of a critical protocol such as TLS.

Bibliography

[1] CryptoVerif: Cryptographic protocol verifier in the computational model. Available at: `http://prosecco.gforge.inria.fr/personal/bblanche/cryptoverif/`.

[2] FlexTLS: A tool for testing TLS implementations. Available at: `https://mitls.org/pages/flextls`.

[3] miTLS: A verified reference implementation of TLS. Available at: `https://mitls.org/`.

[4] ProVerif: Cryptographic protocol verifier in the formal model. Available at: `http://prosecco.gforge.inria.fr/personal/bblanche/proverif/`.

[5] Scyther tool. Available at: `https://www.cs.ox.ac.uk/people/cas.cremers/scyther/`.

[6] Tamarin prover (develop branch). Available at `https://github.com/tamarin-prover/tamarin-prover`.

[7] Anne Adams and Martina A. Sasse. Users are not the enemy. *Communications of the ACM*, 42(12):40–46, December 1999.

[8] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15: 22nd Conference on Computer and Communications Security*, pages 5–17, Denver, CO, USA, October 12–16, 2015. ACM Press.

[9] Martin R. Albrecht and Kenneth G. Paterson. Lucky microseconds: A timing attack on Amazon's s2n implementation of TLS. In Marc Fischlin and Jean-Sébastien

Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 622–643, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany.

[10] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the security of RC4 in TLS. In *Proceedings of the 22nd USENIX Conference on Security*, USENIX Security 13, pages 305–320, Berkeley, CA, USA, 2013. USENIX Association.

[11] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540, Berkeley, CA, USA, May 19–22, 2013. IEEE Computer Society Press.

[12] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Verifiable side-channel security of cryptographic implementations: Constant-time MEE-CBC. In Thomas Peyrin, editor, *Fast Software Encryption – FSE 2016*, volume 9783 of *Lecture Notes in Computer Science*, pages 163–184, Bochum, Germany, March 20–23, 2016. Springer, Heidelberg, Germany.

[13] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *ASIACCS 15: 10th ACM Symposium on Information, Computer and Communications Security*, pages 85–96, Singapore, April 14–17, 2015. ACM Press.

[14] Kenichi Arai. Formal verification of TLS 1.3 full handshake protocol using Proverif. Technical report, Cryptographic protocol Evaluation toward Long-Lived Outstanding Security Consortium (CELLOS), February 2016. Available at: `https://www.cellos-consortium.org/studygroup/TLS1.3-fullhandshake-draft11.pv`.

[15] Barry C. Arnold, N. Balakrishnan, and H.N. Nagaraja. *A First Course in Order Statistics*. SIAM, Philadelphia, PA, USA, 2008.

[16] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS using SSLv2. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 689–706, Austin, TX, 2016. USENIX Association.

[17] M. Badra. Pre-Shared Key Cipher Suites for TLS with SHA-256/384 and AES Galois Counter Mode. RFC 5487 (Proposed Standard), March 2009.

[18] Gregory V. Bard. The vulnerability of SSL to chosen plaintext attack. Cryptology ePrint Archive, Report 2004/111, 2004. `http://eprint.iacr.org/2004/111`.

[19] Gregory V. Bard. A challenging but feasible blockwise-adaptive chosen-plaintext attack on SSL. In *SECRYPT*, pages 99–109, 2006.

[20] David A. Basin, Jannik Dreier, and Ralf Sasse. Automated symbolic proofs of observational equivalence. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15: 22nd Conference on Computer and Communications Security*, pages 1144–1155, Denver, CO, USA, October 12–16, 2015. ACM Press.

[21] Thomas Bayes. An essay towards solving a problem in the doctrine of chances. *Philosophical Transactions*, 53:370–418, 1763.

[22] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545, Kyoto, Japan, December 3–7, 2000. Springer, Heidelberg, Germany.

[23] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249, Santa Barbara, CA, USA, August 22–26, 1994. Springer, Heidelberg, Germany.

[24] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996.

[25] Benjamin. Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cedric Fournet, S. Ishtiaq, Markulf Kohlweiss, Jonathan Protzenko, Nikhil Swamy, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. Towards a provably secure implementation of TLS 1.3. Presented at TRON 1.0, San Diego, CA, USA, February 21, 2016.

[26] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of

TLS. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press.

[27] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy*, pages 483–502, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press.

[28] Karthikeyan Bhargavan, Christina Brzuska, Cédric Fournet, Matthew Green, Markulf Kohlweiss, and Santiago Zanella Béguelin. Downgrade resilience in key-exchange protocols. In *2016 IEEE Symposium on Security and Privacy*, pages 506–525, San Jose, CA, USA, May 22–26, 2016. IEEE Computer Society Press.

[29] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *2014 IEEE Symposium on Security and Privacy*, pages 98–113, Berkeley, CA, USA, May 18–21, 2014. IEEE Computer Society Press.

[30] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 445–459, 2013.

[31] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella Béguelin. Proving the TLS handshake secure (as it is). In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 235–255, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

[32] Karthikeyan Bhargavan, Nadim Kobeissi, and Bruno Blanchet. ProScript TLS: Building a TLS 1.3 implementation with a verifiable protocol model. Presented at TRON 1.0, San Diego, CA, USA, February 21, 2016.

[33] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16: 23rd Conference on Computer and Communications Security*, pages 456–467, Vienna, Austria, October 24–28, 2016. ACM Press.

[34] Karthikeyan Bhargavan and Gaëtan Leurent. Transcript collision attacks: Breaking authentication in TLS, IKE and SSH. In *ISOC Network and Distributed System Security Symposium – NDSS 2016*, San Diego, CA, USA, February 21–24, 2016. The Internet Society.

[35] Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 11–13, 2001. IEEE Computer Society Press.

[36] Bruno Blanchet. Security protocol verification: Symbolic and computational models. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust*, pages 3–29, Tallinn, Estonia, March 24 – April 1, 2012. Springer, Heidelberg, Germany.

[37] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Heidelberg, Germany.

[38] Joseph Bonneau. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *2012 IEEE Symposium on Security and Privacy*, pages 538–552, San Francisco, CA, USA, May 21–23, 2012. IEEE Computer Society Press.

[39] Joseph Bonneau and Sören Preibusch. The password thicket: Technical and market failures in human authentication on the web. In *9th Annual Workshop on the Economics of Information Security, WEIS 2010, Harvard University, Cambridge, MA, USA, June 7 - 8*, 2010.

[40] Remi Bricout, Sean Murphy, Kenneth G. Paterson, and Thyla van der Merwe. Analysing and exploiting the Mantin biases in RC4. *Designs, Codes and Cryptography*, 86(4):743–770, April 2018.

[41] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In Birgit Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 453–474, Innsbruck, Austria, May 6–10, 2001. Springer, Heidelberg, Germany.

[42] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password interception in a SSL/TLS channel. In Dan Boneh, editor, *Advances in Cryptology –*

CRYPTO 2003, volume 2729 of *Lecture Notes in Computer Science*, pages 583–599, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Heidelberg, Germany.

[43] Douglas E. Comer. *Internetworking with TCP/IP*, volume 1. Pearson, Harlow, Essex, UK, sixth edition, 2013.

[44] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. Automated analysis of TLS 1.3: Symbolic analysis using the Tamarin prover. Website, November 2017. `https://tls13tamarin.github.io/TLS13Tamarin/`.

[45] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1773–1788, New York, NY, USA, 2017. ACM.

[46] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy*, pages 470–485, San Jose, CA, USA, May 22–26, 2016. IEEE Computer Society Press.

[47] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *2017 IEEE Symposium on Security and Privacy*, pages 463–482, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press.

[48] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176, 7465, 7507.

[49] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), April 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176, 7465, 7507.

[50] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685.

[51] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, September 2006.

[52] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992.

[53] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

[54] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15: 22nd Conference on Computer and Communications Security*, pages 1197–1210, Denver, CO, USA, October 12–16, 2015. ACM Press.

[55] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 draft-10 full and pre-shared key handshake protocol. Cryptology ePrint Archive, Report 2016/081, 2016. `http://eprint.iacr.org/2016/081`.

[56] Benjamin Dowling and Douglas Stebila. Modelling ciphersuite and version negotiation in the TLS protocol. In Ernest Foo and Douglas Stebila, editors, *ACISP 15: 20th Australasian Conference on Information Security and Privacy*, volume 9144 of *Lecture Notes in Computer Science*, pages 270–288, Wollongong, NSW, Australia, June 29 – July 1, 2015. Springer, Heidelberg, Germany.

[57] Thai Duong and Juliano Rizzo. Here come the $\oplus$ ninjas. Unpublished manuscript, May 2011.

[58] Thai Duong and Juliano Rizzo. The CRIME attack. Ekoparty Security Conference presentation, 2012.

[59] P. Eronen and H. Tschofenig. Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279 (Proposed Standard), December 2005.

[60] Marc Fischlin and Felix Günther. Multi-stage key exchange and the case of Google's QUIC protocol. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14: 21st Conference on Computer and Communications Security*, pages 1193–1204, Scottsdale, AZ, USA, November 3–7, 2014. ACM Press.

[61] Marc Fischlin, Felix Günther, Benedikt Schmidt, and Bogdan Warinschi. Key confirmation in key exchange: A formal treatment and implications for TLS 1.3. In

2016 IEEE Symposium on Security and Privacy, pages 452–469, San Jose, CA, USA, May 22–26, 2016. IEEE Computer Society Press.

[62] Dinei Florencio and Cormac Herley. A large-scale study of web password habits. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 657–666, New York, NY, USA, 2007. ACM.

[63] Scott R. Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of RC4. In Serge Vaudenay and Amr M. Youssef, editors, *SAC 2001: 8th Annual International Workshop on Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science*, pages 1–24, Toronto, Ontario, Canada, August 16–17, 2001. Springer, Heidelberg, Germany.

[64] Scott R. Fluhrer and David A. McGrew. Statistical analysis of the alleged RC4 keystream generator. In Bruce Schneier, editor, *Fast Software Encryption – FSE 2000*, volume 1978 of *Lecture Notes in Computer Science*, pages 19–30, New York, NY, USA, April 10–12, 2001. Springer, Heidelberg, Germany.

[65] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard), June 1999. Obsoleted by RFCs 7235, 7615, 7616, 7617.

[66] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), August 2011.

[67] Sebastian Gajek, Mark Manulis, Olivier Pereira, Ahmad-Reza Sadeghi, and Jörg Schwenk. Universally composable security analysis of TLS. In Joonsang Baek, Feng Bao, Kefei Chen, and Xuejia Lai, editors, *ProvSec 2008: 2nd International Conference on Provable Security*, volume 5324 of *Lecture Notes in Computer Science*, pages 313–327, Shanghai, China, October 31 – November 1, 2008. Springer, Heidelberg, Germany.

[68] Christina Garman, Kenneth G. Paterson, and Thyla van der Merwe. Attacks only get better: Password recovery attacks against RC4 in TLS. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 113–128, Washington, D.C., 2015. USENIX Association.

[69] David Garret. Banning SHA-1 in TLS 1.3, a new attempt. TLS mailing list post, October 2015. Available at `http://www.ietf.org/mail-archive/web/tls/current/msg17956.html`.

[70] David Garret. MD5 diediedie (was Re: Deprecating TLS 1.0, 1.1 and SHA1 signature algorithms). TLS mailing list post, January 2016. Available at `http://www.ietf.org/mail-archive/web/tls/current/msg18977.html`.

[71] Florian Giesen, Florian Kohlar, and Douglas Stebila. On the security of TLS renegotiation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 387–398, Berlin, Germany, November 4–8, 2013. ACM Press.

[72] Sourav Sen Gupta, Subhamoy Maitra, Goutam Paul, and Santanu Sarkar. Proof of empirical RC4 biases and new key correlations. In Ali Miri and Serge Vaudenay, editors, *SAC 2011: 18th Annual International Workshop on Selected Areas in Cryptography*, volume 7118 of *Lecture Notes in Computer Science*, pages 151–168, Toronto, Ontario, Canada, August 11–12, 2012. Springer, Heidelberg, Germany.

[73] Sourav Sen Gupta, Subhamoy Maitra, Goutam Paul, and Santanu Sarkar. (Non-)random sequences from (non-)random permutations - analysis of RC4 stream cipher. *Journal of Cryptology*, 27(1):67–108, January 2014.

[74] P. Gutmann. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366 (Proposed Standard), September 2014.

[75] Marko Horvat. *Formal Analysis of Modern Security Protocols in Current Standards*. PhD thesis, University of Oxford, 2016.

[76] Takanori Isobe, Toshihiro Ohigashi, Yuhei Watanabe, and Masakatu Morii. Full plaintext recovery attack on broadcast RC4. In Shiho Moriai, editor, *Fast Software Encryption – FSE 2013*, volume 8424 of *Lecture Notes in Computer Science*, pages 179–202, Singapore, March 11–13, 2014. Springer, Heidelberg, Germany.

[77] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 273–293, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[78] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15: 22nd Conference on Computer*

and Communications Security, pages 1185–1196, Denver, CO, USA, October 12–16, 2015. ACM Press.

[79] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), February 2003.

[80] Jakob Jonsson and Burton S. Kaliski Jr. On the security of RSA encryption in TLS. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 127–142, Santa Barbara, CA, USA, August 18–22, 2002. Springer, Heidelberg, Germany.

[81] B. Kaliski. PKCS #1: RSA Encryption Version 1.5. RFC 2313 (Informational), March 1998. Obsoleted by RFC 2437.

[82] John Kelsey. Compression and information leakage of plaintext. In Joan Daemen and Vincent Rijmen, editors, *Fast Software Encryption – FSE 2002*, volume 2365 of *Lecture Notes in Computer Science*, pages 263–276, Leuven, Belgium, February 4–6, 2002. Springer, Heidelberg, Germany.

[83] Vlastimil Klíma, Ondrej Pokorný, and Tomás Rosa. Attacking RSA-based sessions in SSL/TLS. In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, pages 426–440, 2003.

[84] Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DH and TLS-RSA in the standard model. Cryptology ePrint Archive, Report 2013/367, 2013. `http://eprint.iacr.org/2013/367`.

[85] Markulf Kohlweiss, Ueli Maurer, Cristina Onete, Björn Tackmann, and Daniele Venturi. (De-)Constructing TLS. Cryptology ePrint Archive, Report 2014/020, 2014. `http://eprint.iacr.org/2014/020`.

[86] H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869 (Informational), May 2010.

[87] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany.

[88] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 631–648, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Heidelberg, Germany.

[89] Hugo Krawczyk. OPTLS: Signature-less TLS 1.3. TLS mailing list, November 2014. `http://www.ietf.org/mail-archive/web/tls/current/msg14385.html`.

[90] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 429–448, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.

[91] Hugo Krawczyk and Hoeteck Wee. The OPTLS protocol and TLS 1.3. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 81–96, Saarbrücken, Germany, March 21-24, 2016. IEEE Computer Sociciety Press.

[92] Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *ProvSec 2007: 1st International Conference on Provable Security*, volume 4784 of *Lecture Notes in Computer Science*, pages 1–16, Wollongong, Australia, November 1–2, 2007. Springer, Heidelberg, Germany.

[93] Adam Langley and Wan-Teh Chang. QUIC Crypto, June 2013. Available at `https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/`.

[94] Olivier Levillain, Baptiste Gourdin, and Hervé Debar. TLS record protocol: Security analysis and defense-in-depth countermeasures for HTTPS. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *ASIACCS 15: 10th ACM Symposium on Information, Computer and Communications Security*, pages 225–236, Singapore, April 14–17, 2015. ACM Press.

[95] Xinyu Li, Jing Xu, Zhenfeng Zhang, Dengguo Feng, and Honggang Hu. Multiple handshakes security of TLS 1.3 candidates. In *2016 IEEE Symposium on Security and Privacy*, pages 486–505, San Jose, CA, USA, May 22–26, 2016. IEEE Computer Society Press.

[96] Yong Li, Sven Schäge, Zheng Yang, Florian Kohlar, and Jörg Schwenk. On the security of the pre-shared key ciphersuites of TLS. In Hugo Krawczyk, editor, *PKC 2014: 17th International Conference on Theory and Practice of Public Key Cryptography*, volume 8383 of *Lecture Notes in Computer Science*, pages 669–684, Buenos Aires, Argentina, March 26–28, 2014. Springer, Heidelberg, Germany.

[97] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*, CSFW '97, pages 31–, Washington, DC, USA, 1997. IEEE Computer Society.

[98] Robert Lychev, Samuel Jero, Alexandra Boldyreva, and Cristina Nita-Rotaru. How secure and quick is QUIC? Provable security and performance analyses. In *2015 IEEE Symposium on Security and Privacy*, pages 214–231, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press.

[99] Subhamoy Maitra, Goutam Paul, and Sourav Sengupta. Attack on broadcast RC4 revisited. In Antoine Joux, editor, *Fast Software Encryption – FSE 2011*, volume 6733 of *Lecture Notes in Computer Science*, pages 199–217, Lyngby, Denmark, February 13–16, 2011. Springer, Heidelberg, Germany.

[100] James Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 230–238, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany.

[101] Itsik Mantin. Predicting and distinguishing attacks on RC4 keystream generator. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 491–506, Aarhus, Denmark, May 22–26, 2005. Springer, Heidelberg, Germany.

[102] Itsik Mantin. Bar Mitzvah Attack: Breaking SSL with a 13-year old RC4 weakness, March 2015. Presented at Blackhat Asia 2015. Paper available at `https://www.blackhat.com/docs/asia-15/materials/asia-15-Mantin-Bar-Mitzvah-Attack-Breaking-SSL-With-13-Year-Old-RC4-Weakness-wp.pdf`.

[103] Itsik Mantin and Adi Shamir. A practical attack on broadcast RC4. In Mitsuru Matsui, editor, *Fast Software Encryption – FSE 2001*, volume 2355 of *Lecture Notes*

in Computer Science, pages 152–164, Yokohama, Japan, April 2–4, 2002. Springer, Heidelberg, Germany.

[104] Shin'ichiro Matsuo. Formal verification of TLS 1.3 full handshake protocol using Proverif (draft-11). TLS mailing list post, February 2016. Available at `https://www.ietf.org/mail-archive/web/tls/current/msg19339.html`.

[105] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. A cross-protocol attack on the TLS protocol. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12: 19th Conference on Computer and Communications Security*, pages 62–72, Raleigh, NC, USA, October 16–18, 2012. ACM Press.

[106] D. McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116 (Proposed Standard), January 2008.

[107] Simon Meier. *Advancing Automated Security Protocol Verification*. PhD thesis, ETH Zurich, 2013.

[108] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 733–748, San Diego, CA, 2014. USENIX Association.

[109] Bodo Möller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures, 2004. Unpublished manuscript. Available at `https://www.openssl.org/~bodo/tls-cbc.txt`.

[110] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: Exploiting the SSL 3.0 fallback, 2014. Unpublished manuscript. Available at `https://www.openssl.org/~bodo/ssl-poodle.pdf`.

[111] Paul Morrissey, Nigel P. Smart, and Bogdan Warinschi. The TLS handshake protocol: A modular analysis. *Journal of Cryptology*, 23(2):187–223, April 2010.

[112] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the Association for Computing Machinery*, 21(21):993–999, December 1978.

[113] Toshihiro Ohigashi, Takanori Isobe, Yuhei Watanabe, and Masakatu Morii. How to recover any byte of plaintext on RC4. In Tanja Lange, Kristin Lauter, and Petr

Lisonek, editors, *SAC 2013: 20th Annual International Workshop on Selected Areas in Cryptography*, volume 8282 of *Lecture Notes in Computer Science*, pages 155–173, Burnaby, BC, Canada, August 14–16, 2014. Springer, Heidelberg, Germany.

[114] Kenneth G. Paterson and Nadhem J. AlFardan. Plaintext-recovery attacks against datagram TLS. In *ISOC Network and Distributed System Security Symposium – NDSS 2012*, San Diego, CA, USA, February 5–8, 2012. The Internet Society.

[115] Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. Big bias hunting in amazonia: Large-scale computation and exploitation of RC4 biases (invited paper). In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 398–419, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Heidelberg, Germany.

[116] Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. Plaintext recovery attacks against WPA/TKIP. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption – FSE 2014*, volume 8540 of *Lecture Notes in Computer Science*, pages 325–349, London, UK, March 3–5, 2015. Springer, Heidelberg, Germany.

[117] Kenneth G. Paterson, Thomas Ristenpart, and Thomas Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 372–389, Seoul, South Korea, December 4–8, 2011. Springer, Heidelberg, Germany.

[118] Kenneth G. Paterson and Jacob C.N. Schuldt. Statistical attacks on cookie masking for RC4. Cryptology ePrint Archive, Report 2018/093, 2018. Available at `https://eprint.iacr.org/2018/093`.

[119] Kenneth G. Paterson and Mario Strefler. A practical attack against the use of RC4 in the HIVE hidden volume encryption system. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *ASIACCS 15: 10th ACM Symposium on Information, Computer and Communications Security*, pages 475–482, Singapore, April 14–17, 2015. ACM Press.

[120] Goutam Paul and Subhamoy Maitra. Permutation after RC4 key scheduling reveals the secret key. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors,

SAC 2007: 14th Annual International Workshop on Selected Areas in Cryptography, volume 4876 of *Lecture Notes in Computer Science*, pages 360–377, Ottawa, Canada, August 16–17, 2007. Springer, Heidelberg, Germany.

[121] Lawrence C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Transactions on Information Systems Security*, 2(3):332–351, August 1999.

[122] A. Popov. Prohibiting RC4 Cipher Suites. RFC 7465 (Proposed Standard), February 2015.

[123] Andrei Popov. TLS 1.3 client authentication. In *Meeting proceedings of the IETF-93 Workshop, Prague*. Retrieved from `https://www.ietf.org/proceedings/93/slides/slides-93-tls-2.pdf`, 2015.

[124] J. Postel. DoD standard Transmission Control Protocol. RFC 761, January 1980. Obsoleted by RFC 793.

[125] J. Postel. User Datagram Protocol. RFC 768 (INTERNET STANDARD), August 1980.

[126] E. Rescorla. Keying Material Exporters for Transport Layer Security (TLS). RFC 5705 (Proposed Standard), March 2010.

[127] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3 (draft, revision 10), October 2015. Available at `https://tools.ietf.org/html/draft-ietf-tls-tls13-10`.

[128] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3 (draft, revision 21), October 2015. Available at `https://tools.ietf.org/html/draft-ietf-tls-tls13-21`.

[129] E. Rescorla and B. Korver. Guidelines for Writing RFC Text on Security Considerations. RFC 3552 (Best Current Practice), July 2003.

[130] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347 (Proposed Standard), April 2006. Obsoleted by RFC 6347, updated by RFCs 5746, 7507.

[131] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012. Updated by RFC 7507.

[132] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), February 2010.

[133] Eric Rescorla. TLS 1.3 specification pull request: WIP client auth revision #316. https://github.com/tlswg/tls13-spec/pull/316/.

[134] Eric Rescorla. TLS 1.3 Status. In *Meeting proceedings of the IETF-93 Workshop, Prague*. Previously available at https://www.ietf.org/proceedings/93/slides/slides-93-tls-8.pdf, 2015.

[135] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communicaitons of the ACM*, 21(2):120–126, February 1978.

[136] Phillip Rogaway. Problems with proposed IP cryptography. Unpublished manuscript, 1995. http://www.cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt.

[137] J. Roskind. QUIC: Quick UDP Internet Connections, April 2012. Available at https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit?pref=2&pli=1.

[138] J. Salowey, A. Choudhury, and D. McGrew. AES Galois Counter Mode (GCM) Cipher Suites for TLS. RFC 5288 (Proposed Standard), August 2008.

[139] Pratik Guha Sarkar and Shawn Fitzgerald. Attacks on SSL – a comprehensive study of BEAST, CRIME, TIME, BREACH, Lucky 13 and RC4 biases, August 2013. White paper. Available at https://www.isecpartners.com/media/106031/ssl_attacks_survey.pdf.

[140] Santanu Sarkar, Sourav Sen Gupta, Goutam Paul, and Subhamoy Maitra. Proving TLS-attack related open biases of RC4. Cryptology ePrint Archive, Report 2013/502, 2013. http://eprint.iacr.org/2013/502.

[141] Benedikt Schmidt. *Formal Analysis of Key Exchange Protocols and Physical Protocols*. PhD thesis, ETH Zurich, 2012.

[142] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, pages 78–94, Washington, DC, USA, 2012. IEEE Computer Society.

[143] Pouyan Sepehrdad, Serge Vaudenay, and Martin Vuagnoux. Discovery and exploitation of new biases in RC4. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *SAC 2010: 17th Annual International Workshop on Selected Areas in Cryptography*, volume 6544 of *Lecture Notes in Computer Science*, pages 74–91, Waterloo, Ontario, Canada, August 12–13, 2011. Springer, Heidelberg, Germany.

[144] Pouyan Sepehrdad, Serge Vaudenay, and Martin Vuagnoux. Statistical attack on RC4 - distinguishing WPA. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 343–363, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany.

[145] Nambi Seshadri and Carl-Erik W. Sundberg. List Viterbi decoding algorithms with applications. *IEEE Transactions on Communications*, 42(234):313–323, 1994.

[146] Transport Layer Security Charter, February 2014. Available at `https://datatracker.ietf.org/wg/tls/charter`.

[147] S. Turner and T. Polk. Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176 (Proposed Standard), March 2011.

[148] Mathy Vanhoef and Frank Piessens. All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 97–112, Washington, D.C., 2015. USENIX Association.

[149] Serge Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS... In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Heidelberg, Germany.

[150] Serge Vaudenay and Martin Vuagnoux. Passive-only key recovery attacks on RC4. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *SAC 2007: 14th Annual International Workshop on Selected Areas in Cryptography*, volume 4876 of *Lecture Notes in Computer Science*, pages 344–359, Ottawa, Canada, August 16–17, 2007. Springer, Heidelberg, Germany.

[151] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *Proceedings of the 2nd Conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*, WOEC'96, pages 4–4, Berkeley, CA, USA, 1996. USENIX Association.

[152] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35, Aarhus, Denmark, May 22–26, 2005. Springer, Heidelberg, Germany.

[153] Matt Weir, Sudhir Aggarwal, Michael Collins, and Henry Stern. Testing metrics for password creation policies by attacking large sets of revealed passwords. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 10: 17th Conference on Computer and Communications Security*, pages 162–175, Chicago, Illinois, USA, October 4–8, 2010. ACM Press.

[154] Jeff Yan, Alan Blackwell, Ross Anderson, and Alasdair Grant. Password memorability and security: Empirical results. *IEEE Security and Privacy*, 2(5):25–31, September 2004.

[155] Moshe Zviran and William J. Haga. Password security: An empirical study. *Journal of Management Information Systems*, 15(4):161–185, March 1999.

Appendix A

STS .spthy File

```
theory sts
begin

builtins: diffie-hellman, hashing, asymmetric-encryption, symmetric-encryption,
signing

/*
Modeling the Public Key Infrastructure
--------------------------------------
*/

// Registering a public key
rule Gen_keypair:
 [ Fr(~ltkA) ]--[ GenLtk($A, ~ltkA)
 ]->
 [ !Ltk($A, ~ltkA), !Pk($A, pk(~ltkA))]

// Revealing a long-term key
rule Reveal_Ltk:
  [ !Ltk($A, ~ltkA) ] --[ RevLtk($A) ]-> [ Out(~ltkA) ]
```

```
/*
Modeling the Protocol
---------------------
*/

rule Client_1:
    [ Fr(~a)                              // Choose fresh DH exponent.
    ]
  --[ C1(~a)
    , Start(~a, $C, 'client')             // Log a start action.
    , DH($C, ~a)                          // Log a DH action.
    ]->
    [ Client_1($C, ~a)                    // Store DH exponent for the next step.
    , Out(<$C,'g'^~a>)                    // Send DH exponent to S.
    ]

rule Server_1:
let
    k = ckeyshare^~b
in
    [ Fr(~b)                              // Choose fresh DH exponent.
    , !Ltk($S, ~ltkS)                     // Retrieve server long-term key.
    , In(<C,ckeyshare>)                   // Receive client key share.
    ]
  --[ S1(~b)
    , Start(~b, $S, 'server')             // Log a start action.
    , UseSessionKey(C, k)                 // Log use of session key.
    , Neq($S, C)                          // Avoiding trivial adversarial wins.
    , SignS('g'^~b, ckeyshare)            // Log signature on the key shares.
    , Running($S, C, 'server', 'g'^~b, ckeyshare)   // Log a running action.
    ]->
    [ Server_1($S, C, ~b, ckeyshare, k)   // Store state.
    , Out(<$S,'g'^~b, senc{sign{<'g'^~b, ckeyshare>}~ltkS}k>) // Send key share
                                                              // and encrypted
                                                              // signature.
    ]
```

```
rule Client_2:
let
    k = skeyshare^~a
in
    [ Client_1($C, ~a)                    // Retrieve DH exponent from previous
                                          // step.
    , !Ltk($C, ~ltkC)                     // Retrieve client long-term key.
    , !Pk(S, pk(~ltkS))                   // Retrieve public key.
    , In(<S,skeyshare, s_encryptedsignature>)  // Receive server key share and
                                               // encrypted signature.
    ]
  --[ C2(~a)
    , Neq(S, $C)                          // Avoiding trivial adversarial wins.
    , Eq(verify(sdec{s_encryptedsignature}k, <skeyshare, 'g'^~a> , pk(~ltkS)),
      true)                               // Verify the server signature.
    , SessionKey($C ,S ,k, 'authenticated')  // Log shared session key.
    , SignC('g'^~a, skeyshare)            // Log signature on the key shares.
    , Running($C, S, 'client', 'g'^~a, skeyshare)   // Log a running action.
    , Commit($C, S, 'client', 'g'^~a, skeyshare)    // Log a commit action.
    ]->
    [ Client_2($C, S, ~a, skeyshare, k)             // Store state.
    , Out(senc{sign{<'g'^~a, skeyshare>}~ltkS}k)    // Send encrypted signature.
    ]


rule Server_2:
    [ Server_1($S, C, ~b, ckeyshare, k)   // Retrieve DH exponent, client key
                                          // share and shared key from previous
                                          // step.
    , !Pk(C, pk(~ltkC))                   // Retrieve client public key.
    , In(c_encryptedsignature)            // Receive server key share and
                                          // encrypted signature.
    ]
  --[ S2(~b)
    , Eq(verify(sdec{c_encryptedsignature}k, <ckeyshare, 'g'^~b> , pk(~ltkC)),
    true)                                 // Verify the client signature.
    , SessionKey($S,C,k,'authenticated')            // Log shared session key.
    , Commit($S, C, 'server', 'g'^~b, ckeyshare)    // Log a commit action.
    ]->
    [ Server_2($S, C, ~b, ckeyshare, k)             // Store state.
    ]
```

```
/*
Encoding Properties
-------------------------------
*/

restriction Equality_Checks_Succeed:
"All x y #i. Eq(x,y) @ i ==> x = y"

restriction NEquality_Checks_Succeed:
"All x y #i. Neq(x, y)@i ==> not (x = y)"

restriction One_Ltk:
"All A x y #i #j. GenLtk(A, x)@i & GenLtk(A, y)@j ==> #i = #j"

restriction One_Role_Per_Actor:
"All actor tid1 tid2 role1 role2 #i #j. Start(tid1, actor, role1)@i
 & Start(tid2, actor, role2)@j
 ==> role1 = role2"

/*
Reachability tests -- replace In and Out with AuthMessage when checking
reachability, and comment this section out when proving other security
properties.
*/

restriction At_most_1_of_C1:
"All #i #j tid1 tid2. C1(tid1)@i & C1(tid2)@j ==> #i = #j"

restriction At_most_1_of_S1:
"All #i #j tid1 tid2. S1(tid1)@i & S1(tid2)@j ==> #i = #j"

restriction At_most_1_of_C2:
"All #i #j tid1 tid2. C2(tid1)@i & C2(tid2)@j ==> #i = #j"

restriction At_most_1_of_S2:
"All #i #j tid1 tid2. S2(tid1)@i & S2(tid2)@j ==> #i = #j"

lemma exists_C1:
  exists-trace
  "Ex tid #i. C1(tid)@i"
```

```
lemma exists_S1:
  exists-trace
  "Ex tid #i. S1(tid)@i"

lemma exists_C2:
  exists-trace
  "Ex tid #i. C2(tid)@i"

lemma exists_S2:
  exists-trace
  "Ex tid #i. S2(tid)@i"

/*
Encoding Security Properties
--------------------------------
*/

lemma entity_authentication[reuse]:
  "All actor peer keyshare1 keyshare2 #i.
  Commit(actor, peer, 'client', keyshare1, keyshare2)@i
  & not (Ex #r. RevLtk(peer)@r)
  ==> Ex #j peer2. Running(peer, peer2, 'server', keyshare2, keyshare1)@j
  & #j < #i"

lemma mutual_authentication[reuse]:
  "All actor peer keyshare1 keyshare2 #i.
  Commit(actor, peer, 'server', keyshare1, keyshare2)@i
  & not ((Ex #r. RevLtk(peer)@r) | (Ex #r. RevLtk(actor)@r))
  ==> (Ex #j. Running(peer, actor, 'client', keyshare2, keyshare1)@j
  & #j < #i)"

lemma session_key_secrecy:
  "All actor peer k #i. SessionKey(actor, peer, k,'authenticated')@i
  & not ((Ex #r. RevLtk(peer)@r & #r < #i)
        | (Ex #r. RevLtk(actor)@r & #r < #i))
  ==> not Ex #j. K(k)@j"

end
```