20 Years of SSL/TLS Research An Analysis of the Internet's Security Foundation

Christopher Meyer
(Place of birth: Marburg/Wehrda)
christopher.meyer@rub.de

9th February 2014

Ruhr-University Bochum
Horst Görtz Institute for IT-Security
Chair for Network and Data Security

Dissertation zur Erlangung des Grades eines Doktor-Ingenieurs der Fakultät für Elektrotechnik und Informationstechnik an der Ruhr-Universität Bochum

First Supervisor: Prof. Dr. rer. nat. Jörg Schwenk
Second Supervisor: Prof. Dr.-Ing. Felix Freiling

www.nds.rub.de

This work is dedicated to my deceased grandparents who always believed in me and I hereby hopefully make proud. Wherever you are, I will never forget you! Thank you for the wonderful time we had together!

Bochum, 9th February 2014

This thesis was defended on the 7^{th} of February, 2014 at the Faculty of Electrical Engineering and Information Technology, Ruhr-University Bochum.

Examination Committee:

- Prof. Dr.-Ing. D. Göhringer (Chairwoman of the Committee)
- Prof. Dr.-Ing. T. Güneysu (Member of the Committee)
- Prof. Dr.-Ing. T. Holz (Member of the Committee)
- Prof. Dr. rer. nat. J. Schwenk (First Thesis Supervisor)
- Prof. Dr.-Ing. F. Freiling (Second Thesis Supervisor)

Since its introduction in 1994 the Secure Socket Layer (SSL) protocol (later renamed to Transport Layer Security (TLS)) evolved to the de facto standard for securing the transport layer. SSL/TLS can be used for ensuring data confidentiality, integrity and authenticity during transport. A main feature of the protocol is its flexibility. Modes of operation and security goals can easily be configured through different cipher suites.

This thesis focuses on the SSL/TLS protocol suites for secure communication over untrusted media (the internet). It explains the protcol in depth (in the – at the time of writing – most common version TLS 1.0), provides a brief summary on its evolution and gives a detailed chronology of attacks and attack categories. New techniques for fingerprinting particular SSL/TLS implementations (stacks), partially down to patch level, are introduced and implemented as a tool for SSL/TLS stack inspection.

Moreover, new attacks on state of the art implementations are given and a new framework, called *T.I.M.E.*, especially designed for deep protocol inspection and penetration is presented. The attacks undermine the security goal confidentiality. Exploits for the discussed attacks are implemented by using the *T.I.M.E* framework. The results caused changes (fixes) in the affected implementations and highlight the need for tested, specification compliant and security optimized software.

Contents

Glossary

Abstract Syntax Notation One ASN.1 is an ITU-T standard for structured definition of data formats.
See e.g. ITU-T ASN.1 Specification Website.

Advanced Encryption Standard The Advanced Encryption Standard is a symmetric encryption algorithm standardized by NIST that supersedes DES. AES's original name is Rijndael, a combination of the lastnames of the inventors Joan Daemen and Vincent Rijmen.
See "FIPS PUB 197" [1].

Application Programming Interface An API defines the interface for software components to interact with each other.
See e.g. Wikipedia Website.

Authenticated Encryption with Associated Data AEAD is an umbrella term for block cipher modes of operation offering authenticated encryption.
See "Authenticated-Encryption with Associated-Data" [2].

C programming language C is a procedural programming language designed in the early '70s. The history of C is closely related to the history of UNIX operation systems, since wide parts of the OS are written in C. Today, C is still one of the most frequented programming languages.
See e.g. ISO/IEC 9899:2011 [3].

Certificate Authority In a PKI a Certificate Authority is an entity responsible and legitimated to issue and sign sub certificates. Therefore, it is crucial that the CA can be trusted, otherwise the whole certificate chain cannot be trusted.
See e.g. Wikipedia Website.

Certificate Signing Request A Certificate Signing Request is a standardized format for requesting digitally signed certificates from a CA. It contains all necessary details to be included in a certificate.
See RFC 2986 [4].

Chinese Remainder Theorem By the use of the Chinese Remainder Theorem it is possible to regain the initial, but unknown, number resulting in known remainders when divided by known divisors.
See "Handbook of Applied Cryptography" [5].

Chosen Ciphertext Attack In a chosen ciphertext attack an adversary is able to decrypt ciphertexts of her choice and gain knowledge by observing and subsequently crafting these ciphertexts in a speical manner.
See "Handbook of Applied Cryptography" [5].

Cipher Block Chaining CBC is an operation mode for block ciphers coupling data blocks dependent on their predecessor or IV.
See Patent US 4074066 A [6].

Client for URLs cURL is a command line tool for data transport supporting manifold protocols such as HTTP{s}, ...
See cURL Website.

Common Vulnerabilities and Exposures A CVE defines a standard for unitary description of security vulnerabilities of software. The Common Vulnerabilities and Exposure database is being maintained by The MITRE Corporation.
See MITRE Website.

Common Weakness Enumeration CWE represents a list of software weaknesses and vulnerabilities.
See MITRE Website.

Counter with CBC-MAC CCM is an AEAD mode of operation for block ciphers offering authenticated encryption combining the counter mode with CBC-MACs.
See "NIST Special Publication 800-38D" [7].

Cyclic Redundancy Check The Cyclic Redundancy Check is an error detection code based on polynomial division that outputs a check value which verifies correctness of data.
See "Cyclic Codes for Error Detection" [8].

Data Encryption Standard DES is a symmetric encryption algorithm standardized in 1976 which should today be considered as insecure for highly confident applications, due to its short key length of 56 bits. DES has influenced notably research, understanding and analysis of cryptographic algorithms.
See "FIPS PUB 46" [9].

Datagram Transport Layer Security DTLS defines a TLS based protocol for providing authenticity, integrity and confidentiality by using non-reliable transport protocols, such as UDP.
See RFC 4347 [10] and RFC 6347 [11].

Denial of Service A Denial of Service attack aims at slowing down or preventing a machine from working properly, directly influencing the security goal reliability.
See e.g. RFC 4732 [12].

Diffie Hellman Key Exchange The Diffie Hellman Key Exchange is a way for two parties (or in an iterative version multiple parties) to agree on a shared secret without presence of a secure channel.
See RFC 2631 [13] and Patent US4200770 [14].

Digital Signature Algorithm The Digital Signature Algorithm is a specific algorithm for digital signatures and included in the DSS.
See "FIPS PUB 186" [15].

Digital Signature Standard The Digital Signature Standard is a NIST provided standard for digital signatures and combines multiple signature types and algorithms.
See "FIPS PUB 186" [15].

Elliptic Curve Cryptography Elliptic curve cryptography defines an asymmetric cryptosystem which enables the use of relatively small key sizes. All operations are based on elliptic curves over finite fields. The security assumption is based on the problem of finding discrete logarithms efficiently.
See RFC 6090 [16].

Elliptic Curve Digital Signature Algorithm ECDSA defines how to adopt the Digital Signature Algorithm for elliptic curve cryptography.
See ANSI X9.62 [17].

Federal Information Processing Standard FIPS references to publicly standardizations of the U.S. Federal Government for computer systems.
See e.g. FIPS Website.

Galois/Counter Mode The Galois/Counter Mode is an AEAD mode of operation for block ciphers offering authenticated encryption.
See "NIST Special Publication 800-38D" [18].

GnuTLS - The GNU Transport Layer Security Library GnuTLS is a free library for SSL, TLS and DTLS and includes SSL/TLS stack libraries as well as utility software.
See GnuTLS Website.

Graphical User Interface A Graphical User Interface allows users to interact with software through a graphical representation of the software interface. This is in contrast to command line interface where users have to write commands to interact with the software.
See Wikipedia Website.

Hash Message Authentication Code A MAC function build out of hash functions.
See "FIPS PUB 198" [19].

HttpComponents HttpClient The Apache HttpClient library implements a feature enhanced client part of HTTP, fulfilling all the needs of HTTP clients.
See HttpClient Website.

Hypertext Transfer Protocol HTTP is a protocol for data transport and can be seen as the base protocol for the world wide web, since nearly every website is delivered through HTTP.
See RFC 1945 [20] and RFC 2616 [21].

Initialization Vector An IV defines an initial state for algorithms depending on previous states (e.g. symmetric ciphers in CBC mode).
See e.g. Patent US 4074066 A [6].

International Data Encryption Algorithm IDEA is a formerly commercial symmetric block cipher, designed by Massey and Lai as a replacment for DES. The patents expired in 2011-2012.
See "A Proposal for a New Block Encryption Standard" [22].

Internet Engineering Task Force IETF aims at improvement and development of the internet by providing standards and technical documentation.
See IETF Website.

Internet Message Access Protocol The Internet Message Access Protocol is a protocol for mail retrieval. IMAP manages mails directly on the server without a need to download the mails.
See RFC 3501 [23].

ITU Telecommunication Standardization Sector ITU-T is an organization responsible for coordination and development of standards concerning the telecommunications sector.
See ITU-T Website.

Java Secure Socket Extension The JSSE is a Java technology implementing technologies for secure communication over sockets, such as SSL and TLS.
See JSSE Documentation Website.

Java WebSocket Client Library Weberknecht - Java WebSocket Client Library is an open source implementation of the IETF WebSocket Protocol.
See Weberknecht Website.

Man-in-the-middle A man-in-the-middle describes an attacker between two communication partners able to passively eavesdrop or even actively interfere with exchanged data.
See Wikipedia Website.

MatrixSSL - Open Source Embedded SSL and TLS MatrixSSL is a SSL/TLS implementation available as open source, specially designed for embedded devices.
See MatrixSSL Website.

Message Authentication Code A MAC is a cryptographic checksum over particular data computed with the help of a secret input. MACs can be seen as cryptographic keyed hash functions and confirm integrity and authenticity of an input.
See e.g. FIPS 113 [24].

Message-Digest Algorithm The Message-Digest Algorithm is a family of cryptographic hash functions developed by Ronald L. Rivest.
See RFC 1319 [25] and RFC 1320 [26] and RFC 1321 [27].

National Institute of Standards and Technology NIST is a is a non-regulatory US agency that develops and maintains official standards.
See NIST Website.

Online Certificate Status Protocol The OCSP is a protocol allowing queries of the revocation status of X.509 certificates.
See RFC 2560 [28].

OpenSSL - The Open Source toolkit for SSL/TLS OpenSSL is a SSL/TLS library available as open source and includes SSL/TLS stack libraries as well as utility software.
See OpenSSL Website.

Operating system An operating system is a set of software necessary to control and manage hardware devices and resources. Applications of higher layers require an operating system to function and rely on the services offered by the OS.
See e.g. "Modern Operating Systems" [29].

Optimal Asymmetric Encryption Padding OAEP is a padding scheme for e.g. RSA based encryption which offers probabilistic encryption together with partial decryption prevention.
See RFC 3447 [30]

PASCAL programming language PASCAL is a programming language named after the mathematician Blaise Pascal. designed to be easy, flexible and efficient. The language was designed as an educational language to encourage structured and efficient programming.
See e.g. PASCAL Central Website.

PHP: Hypertext Preprocessor PHP is a free scripting language mainly used for, but not limited to, dynamic websites.
See PHP Website.

Pseudo-Random Number Generator A PRNG generates pseudo-random numbers based on a deterministic algorithm which is seeded with a start value. The PRNG will always compute the same sequence of numbers for the same seed. PRNGs try to produce seemingly random output of nearly equal entropy when compared to a real random number generator, but

remain a pseudo-random device due to generation with a deterministic algorithm.
See "The Art of Computer Programming, Volume 2 (3rd ed.): Seminumerical Algorithms" [31].

Public Key Infrastructure A PKI defines a hierarchic infrastructure necessary for issuing, distribution and management of digital certificates. The infrastructure consists of multiple technical, procedural, contractual and human components. Asymmetric cryptography forms the foundation of a PKI. The main purpose of a PKI is binding key material to identities and attest the correct binding.
See e.g. RFC 5280 [32].

Public-Key Cryptography Standards PKCS' define specifications and usage recommendations for asymmetric cryptography.
See RSA Labs Website.

Python Python is programming language focusing on development simplicity and readability of source code.
See Python Website.

Random Number Generator An RNG generates random numbers based on an undeterministic source of real entropy.
See "The Art of Computer Programming, Volume 2 (3rd ed.): Seminumerical Algorithms" [31].

Request for Comments An RFC is a document on topics concerning the Internet, such as standards, standard-proposals, extensions, best-practices and so on. RFCs are published by the IETF.
See RFC Editor Website.

Rivest Shamir Adleman Cryptosystem The RSA Cryptosystem is an asymmetric cryptosystem supporting en-/decryption and capabilities to digitally sign and verify data. RSA belongs to the category of public key cryptography.
See Patent US4405829 [33] and "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems" [34].

Ron's Cipher Ron's Cipher is a set of cryptographic algorithms developed by Ronald L. Rivest. One of the most prominent algorithms is the stream cipher RC4.
See RFC 2268 [35], Cypherpunks Mailarchive, "The RC5 Encryption Algorithm" [36] and "The RC6 Block Cipher" [37].

Secure Hash Algorithm The Secure Hash Algorithm is a family of cryptographic hash functions standardized by the NIST.
See "FIPS PUB 180" [38].

Secure Sockets Layer SSL provides confidentiality, authenticity and integrity for data above the transportation layer. The protocol applies asymmetric, as well as symmetric cryptography to provide data security. SSL is a protocol divided into multiple phases (handshaking and application) to strictly separate the establishment of cryptographic primitives, such as algorithms or key lengths, and the application of the agreed parameters.
See Internet Draft "The SSL Protocol" [39] and RFC 6101 [40].

Top-level domain A TLD specifies a domain at the highest level in the Domain Name System.
See RFC 1591 [41].

Transmission Control Protocol TCP defines a reliable protocol for data transportation with integrated package loss detection. TCP can be seen as a foundation for the internet as we know it nowadays.
See RFC 793 [42].

Transport Layer Security TLS is an enhanced version of the renamed SSL protocol providing confidentiality, authenticity and integrity for data above the transportation layer. The protocol can be seen as the defacto standard for securing internet connections.
See RFC 2246 [43], RFC 4346 [44] and RFC 5246 [45].

Unified Modeling Language The Unified Modeling Language is used to model and visualize software architecture and interaction.
See UML Specification Website.

Uniform Resource Identifier A URI is a string clearly identifying a resource or entity.
See RFC 3986 [46].

Uniform Resource Locator URLs point to resources in (mostly network-based) computer systems and uniquely identify these resources.
See RFC 1738 [47].

UNIX UNIX is a multi-user and multitask opration system develop in the early '70s, closely related to the C programming language. The OS had major influences on subsequent operating system devleopment and design.
See e.g. "The UNIX Time-Sharing System" [48].

User Datagram Protocol UDP defines an unreliable protocol for data transportation without package loss detection.
See RFC 768 [49].

Virtual Private Network A Virtual Private Network defines a cryptographicaly secured network, operating on a commonly used network such as the internet. This creates a separated part of network only available for the VPN participants. The VPN remains isolated even thus piggybacked by an unsecure network.
See "Computer Networks" [50].

X.509 X.509 defines components for a public key infrastructure, such as e.g., certificate and certificate revocation list formats or an algorithm for certification path validation. The specification is an official standard by ITUT. *See ITU-T X.509 Specification Website and RFC 5280 [32].*

Acronyms

AEAD Authenticated Encryption with Associated Data

AES Advanced Encryption Standard

Apache HttpClient HttpComponents HttpClient

API Application Programming Interface

ASN.1 Abstract Syntax Notation One

C C programming language

CA Certificate Authority

CBC Cipher Block Chaining

CCA Chosen Ciphertext Attack

CCM Counter with CBC-MAC

CRC Cyclic Redundancy Check

CRT Chinese Remainder Theorem

CSR Certificate Signing Request

cURL Client for URLs

CVE Common Vulnerabilities and Exposures

CWE Common Weakness Enumeration

DES Data Encryption Standard

DH Diffie Hellman Key Exchange

DoS Denial of Service

DSA Digital Signature Algorithm

DSS Digital Signature Standard

DTLS Datagram Transport Layer Security

ECC Elliptic Curve Cryptography

ECDSA Elliptic Curve Digital Signature Algorithm

FIPS A Federal Information Processing Standard

GCM Galois/Counter Mode

GnuTLS GnuTLS - The GNU Transport Layer Security Library

GUI Graphical User Interface

HMAC Hash Message Authentication Code

HTTP Hypertext Transfer Protocol

IDEA International Data Encryption Algorithm

IETF Internet Engineering Task Force

IMAP Internet Message Access Protocol

ITUT ITU Telecommunication Standardization Sector

IV Initialization Vector

JSSE Java Secure Socket Extension

MAC Message Authentication Code

MatrixSSL MatrixSSL - Open Source Embedded SSL and TLS

MD Message-Digest Algorithm

MitM Man-in-the-middle

NIST National Institute of Standards and Technology

OAEP Optimal Asymmetric Encryption Padding

OCSP Online Certificate Status Protocol

OpenSSL OpenSSL - The Open Source toolkit for SSL/TLS

OS Operating system

PASCAL PASCAL programming language

PHP PHP: Hypertext Preprocessor

PKCS Public-Key Cryptography Standards

PKI Public Key Infrastructure

PRNG Pseudo-Random Number Generator

Python Python

RC Ron's Cipher

RFC Request for Comments

RNG Random Number Generator

RSA Rivest Shamir Adleman Cryptosystem

SHA Secure Hash Algorithm

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TLD Top-level domain

TLS Transport Layer Security

UDP User Datagram Protocol

UML Unified Modeling Language

UNIX UNIX operating system

URI Uniform Resource Identifier

URL Uniform Resource Locator

VPN Virtual Private Network

Weberknecht Java WebSocket Client Library

X509 X.509 (Version 3)

"The problem of distinguishing prime numbers from composite numbers and of resolving the latter into their prime factors is known to be one of the most important and useful in arithmetic."

Carl Friedrich Gauss

1 Introduction

Introduced in 1994 as Secure Sockets Layer (SSL) by Netscape Inc. to secure e-commerce, the protocol evolved to be the de facto standard for Transport Layer Security, more specific for Transmission Control Protocol (TCP) based connections. The SSL/Transport Layer Security (TLS) protocol suite is the weapon of choice when dealing with confidentiality, authenticity and integrity of network communication. TLS (1.0) is the official name assigned by the Internet Engineering Task Force (IETF)[1] to SSL starting from Version 3.1 to the – at the time of writing – most frequently used revision. With the new name, new flexibility was added to SSL, such as the expandability through optional extensions. Customised variants of the protocol, as for example the adaptation for unreliable network protocols (e.g. User Datagram Protocol (UDP)) and extended cryptographic primitives (e.g. Elliptic Curve Digital Signature Algorithm (ECDSA)) became popular over time. Overall, the protocol suite underwent five public revisions:

- SSL 2.0 [39]
- SSL 3.0 [40]
- TLS 1.0 [43]
- TLS 1.1 [44]
- TLS 1.2 [45]

The security of SSL/TLS has been heavily discussed, evaluated and enhanced for nearly two decades (for details see Chapter 5). As SSL/TLS is one of the foundations of secure network-based communication, it is necessary to deeply investigate its security and robustness. An analysis based on checking cryptographic primitives (such as certificates and key lengths) is just one step towards

[1] `http://www.ietf.org/`

reliable and secure implementations. Even proving cryptography to be accurate and secure is not sufficient. A comprehensive analysis has to bridge the gap between theory and practice. Therefore, implementation peculiarities, reactions to malformed input and corrupted communication, as well as side channel resistance have to be taken into consideration, too.

1.1. Contribution

This thesis focuses on the security of SSL/TLS and especially on secure implementation of the specifications. The contribution is manifold and the results can be grouped into four areas:

Chronicle of Attacks on SSL/TLS. Previous attacks on the protocols are highlighted and grouped into categories. The chronicle serves as systematization of knowledge and provides brief summaries of existing attacks.

Development of a New Framework for Deep Protocol Inspection. For easy to use protocol interaction without the need for patching existing stacks (protocol implementations) a new framework was developed. By the use of this framework it is easily possible to hook into the protocol flow at any point of interest and alter behaviour as well as data.

Development of a Fingerprinting Technique for SSL/TLS Libraries. During research on the framework it was discovered that many implementations behave absolutely different to particular modifications and thus are distinguishable. This observation motivated the implementation of a new fingerprinting technique, partly independent from configuration changes as it mainly focuses on non-configurable stack behaviour (behaviour that cannot be influenced by particular configuration changes).

Practical and Theoretical Attacks Targeting Popular Stacks. The behaviour analysis and code inspection of SSL/TLS libraries revealed major weaknesses that could be turned into practical attacks. Therefore, new, unpublished attacks of practical and theoretical nature are presented.

If not explicitly annotated, this thesis focuses on TLS 1.0 as specified in Request for Comments (RFC) 2246 [43].

"It is obviously an attempt to convey secret information."

Sherlock Holmes

2 SSL/TLS – The Big Picture

2.1. SSL/TLS – The Abstract View

2.1.1. Security Goals and Demarcation

It is necessary to clearly point out that SSL/TLS only provides transport level security, not persistent data protection. Message level security requires additional means at a higher level in the protocol stack. This implies that the protocol suite cannot be used for (multi-hop) end-to-end protection due to lack for persistent encryption and integrity protection. Security guarantees are only available for point-to-point connections - communication channel security – and not for end-to-end payload security.
SSL/TLS provides multiple, configurable security goals[1], such as:

- *Authentication* – by the use of digital certificates
- *Confidentiality* – by the use of encryption
- *Integrity* – by the use of Message Authentication Code (MAC)s
- *(Non-repudiation)* – by the use of digital certificates
- *Replay protection* – by the use of implicit sequence numbers

2.1.2. Structure

Two-Phase Protocol

SSL/TLS are both multi-layer protocols in all revisions (multiple sub-protocols, but all build upon the *Record Protocol*) consisting of a two-phase architecture:

- Negotiation (handshake) phase

[1] Under the assumption of an appropriate configuration and the use of adequate cipher suites.

- Application phase

During the handshake phase the security parameters and cipher suites are negotiated, as well as the establishment of a *MasterSecret* through (authenticated) key exchange or agreement (dependent on the cipher suite). All key material required for a secure communication is derived from the *MasterSecret*. After handshake phase completion all cryptographic primitives are configured, initialized and already tested. Therefore, both endpoints are negotiated to the same cryptographic state. The protocol continues with the application phase where the real payload data is exchanged securely through the established secure communication channel.

Modular Design

The TLS protocol and its predecessor SSL represent a sophisticated, highly modular architecture. The protocol design is layered and consists of different sub-protocols, as well as configurable and replaceable cryptographic algorithms. In particular the standard(s) define the following (sub-) protocols:

- *Record Protocol*
  - *Handshake Protocol*
  - *ChangeCipherSpec Protocol*
  - *Alert Protocol*
  - *Application Data Protocol*

All messages have at least to be embedded in the *Record Protocol* as depicted in Figure 2.1.

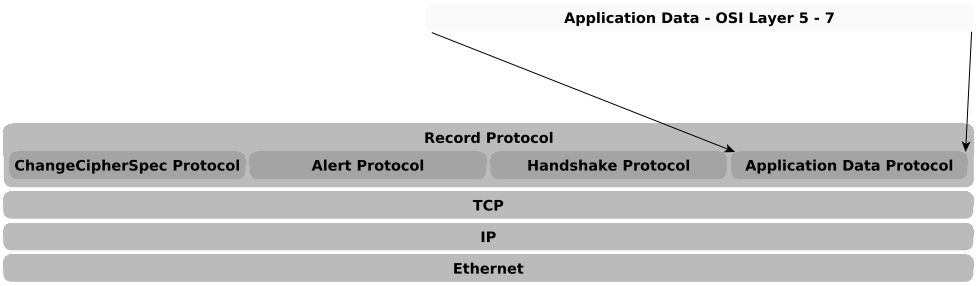

Figure 2.1.: Protocol nesting - *based on source RFC 2246 [43]*

The *Record Layer* is a communication protocol layer that accepts a stream of bytes and splits these bytes into blocks which are processed and thus handed over to TCP.
At start, the processing does not include encryption or integrity protection which is enabled during the *Handshake Phase*.

Session Management

Session management is an essential part of SSL/TLS. Sessions include, with respect to SSL/TLS, a particular, negotiated, configuration of an established connection. This includes cryptographic algorithms, as well as key material (*MasterSecret*[2]) – in the specifications this setup is held in a special structure: the `SecurityParameters`.

A connection, on the other hand refers to a communication channel e.g. a network socket. Sessions can be resumed and reused for different connections to shorten the (computational exprensive) negotiation phase. Surely, this comes with a risk – in case the established *MasterSecret* was compromised -, but on the other hand simplifies the process of establishing a channel. Session resumption is not only limited to single channels, but can also be used to manage multiple connections in parallel, all relying on the same parameters. E.g. a web interface to a corporate email system consists of multiple parts (images, additional sites, etc.) which have to be fetched from the server. In this scenario session resumption can be used to speed up the process of fetching data, by reusing the already established session parameters.

Crucial to note is that only the *MasterSecret* is reused, whereas the key derivation is performed for every resumed session to create new short term keys. That means every resumed session, no matter if in parallel or successively, encrypts or decrypts with different key material. This fact is related to the key derivation process which depends on client and server provided random values, sent as part of the first two messages in a handshake (even if abbreviated). Sessions are identified by session identifiers which are also part of the first two handshake messages (`ClientHello`, `ServerHello`).

For clarification purpose, Table 2.1 summarizes the differences of sessions, connections and channels.

Term	**Description**
TLS Session	Refers to a set of established cryptogaphic parameters (cipher suite and *MasterSecret*) associated with a session identifier. A session can be reused, but the key material will differ everytime (because of different random values in the `ClientHello` and `ServerHello` messages).
TLS Connection	Refers to any TLS connection without specifying particular parameters.
TLS Channel	Refers to a particular TLS connection with a particular set of keys and cryptographic parameters. Generally, a TLS channel is unique, even if it was established by reusing an existing TLS session.

Table 2.1.: Differences of TLS Sessions, Connections and Channels.

[2]Will be described in detail in Section 2.2.

Trust

Many of the aforementioned security goals rely on a trust relationship between the communication partners. Typically, this trust relationship is based on digital certificates, signed by a trusted Certificate Authority (CA). The basic idea is that each communication partner – or at least one of them – proves its identity by performing tasks that only a legitimate owner of the digital certificate[3] is able to complete successfully. The concept of digitally signed certificates and hierarchic attested trust is known as Public Key Infrastructure (PKI).
In the most common web scenario the trust relationship remains asymmetric, meaning that the client is interested that the server proves its identity, but the server does not require any authentication of the client. When browsing the web, this is a very common level of trust: A user is interested in genuine content, but the web server is not interested who requests this content. Although modern browsers support mutual authentication, client authentication is mostly done on higher levels in the protocol stack (as for example the case for web mail accounts where a correct account name/password pair proves the identity). The reason for this is mostly simplicity, because it is easier for a client to set its own password than creating an own digital certificate and getting it signed by a trusted CA.
Displaying the trustworthiness of websites in e.g. web browers is a challenging and complicated task and is ongoing subject to research (e.g. cf. [51], [52]). Each browser signalizes trustworthiness in a slightly different way. Figure 2.2 illustrates the behaviour of Firefox v19.0.2 when visiting a SSL/TLS secured site over Hypertext Transfer Protocol (HTTP)S which presents a valid certificate signed by a trusted CA.

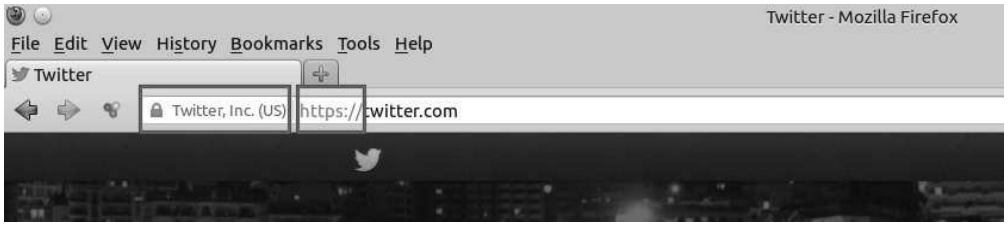

Figure 2.2.: Firefox v19.0.2 signalizing trustworthiness of a web server.

2.2. SSL/TLS – The Detailed View

2.2.1. Structure

The detailed structure of SSL/TLS is far more complex and includes many aspects concerning implementation. Generally, the security goals confidentiality and integrity are ensured by the *Record Layer* which is exactly defined by the *Record Layer Protocol*, whereas authentication and non-repudiation are only ensured once during the *Handshake Phase* which is specified by the *Handshake*

[3] Or at least anyone in posession of the correct private key.

Protocol. Internally, the stack has to keep track of established sessions and the associated parameters (cryptographic primitives, such as algorithms and key lengths, established or agreed key material and finally the corresponding session identifier[4]).

2.2.2. Record Layer

The *Record Layer* is defined by the *Record Protocol*, part of the SSL/TLS specifications[5]. This protocol encapsulates all higher protocols (cf. Figure 2.1) and abstracts encryption, integrity protection and compression mechanisms (depending on the configured cipher suite). Each *Record*'s (fragment) maximum message size may not exceed 2^{14} bytes[6]. The mandatory *Record* header fields are not part of this limitation and added even if the fragment is of maximum size. *Records* can optionally be encrypted (either by the use of a block- or stream-cipher) and/or compressed. The compression algorithm must be lossless and may not add more than 1024 additional bytes. The structure of a *Record* is depicted in Figure 2.3.

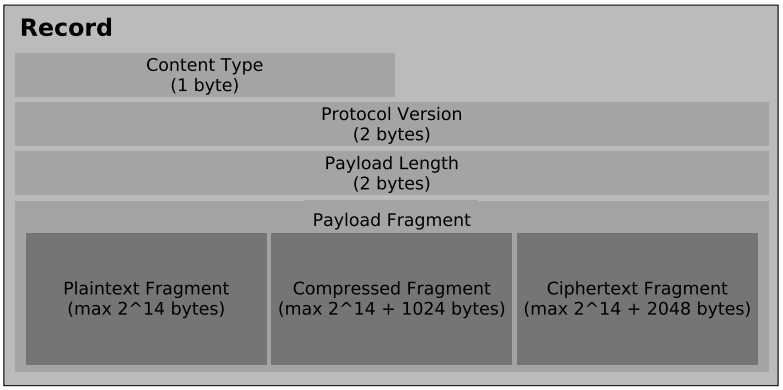

Figure 2.3.: Structure of a Record.

As can be seen, it is possible to send encrypted, but uncompressed; plain and compressed; plain and uncompressed; as well as encrypted and compressed data. Integrity protection is available in the form of an Hash Message Authentication Code (HMAC) and only if `TLSCiphertext` structures are used. Additionally, the unencrypted, but encapsulated payload `TLSPlaintext` may also be compressed (encapsulated to `TLSCompressed`) and finally encrypted (encapsulated to `TLSCiphertext`), to offer encryption, integrity protection and compression at the same time[7]. The fragmentation and encapsulation process of unencrypted payload data into TLS records, based on RFC 2246 [43], is given in Figure 2.4.

[4] If session resumption is supported.

[5] All technical details in this description are exemplarily taken from the TLS 1.0 specification of RFC 2246 [43].

[6] A message length of 0 bytes + header field bytes is not prohibited.

[7] In this case the `TLSCompressed` structure is converted into `TLSCiphertext` and vice versa – the input values of the MAC function are the compressed payload and the implicit sequence number.

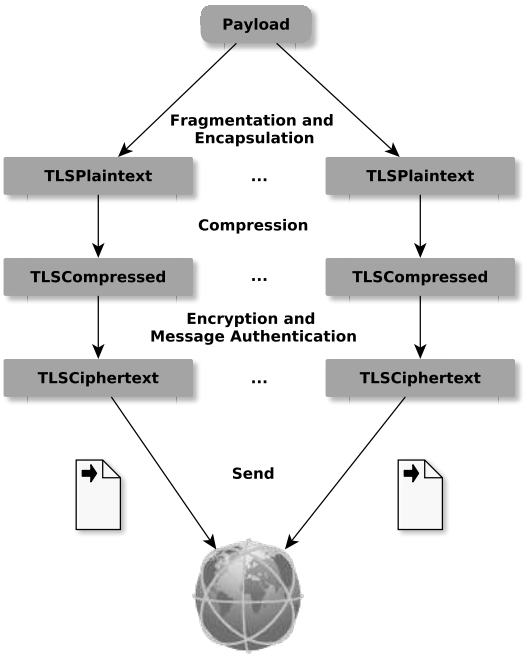

Figure 2.4.: Payload fragmentation and encapsulation.

Encryption

In terms of encryption, SSL/TLS specifies support for stream- and block-ciphers in different operation modes. The details on this vary from revision to revision and define different algorithms and modes of operation for each cipher scheme.

Mac-then-Pad-then-Encrypt SSL/TLS in combination with a block-cipher follows the *Mac-then-Pad-then-Encrypt* paradigm where the `TLSPlaintext` or `TLSCompressed` *Record* is at first integrity protected by a MAC and then padded (up to 256 bytes), to a length which is a multiple of the blocksize and finally encrypted. The MAC includes a *non-visible* feature – a sequence number. The sequence number is at no time delivered, except as part of the input to the MAC function, and prevents replay attacks. MAC and padding (and if necessary a padding length indicator) are attached to the plaintext. Summarizing, the process works as follows: Plaintext ⇒ MAC ⇒ PAD which finally leads to a structure `MSG|MAC|PAD|PADLEN`. This structure serves as input to the encryption function. The resulting ciphertext will be the payload (`fragment` of the `TLSCiphertext` *Record*). As an additional feature the padding can range across multiple blocks of the block cipher to hide the real length of the plaintext.
Special characteristics of the used padding schemes and Initialization Vector (IV)s for the operation modes, as well as implementational details led to serious security breaches in the past[8]. As a countermeasure to attacks of this type, it is crucial to strictly follow the recommended process for decryption:

[8]Vulnerabilities and security breaches are discussed in detail in Chapter 5.

1. Ciphertext decryption
2. Validity check of separated padding
3. Validity check of separated MAC

If an error occurs at any processing step no distinguishable error messages should be returned. Every error must lead to the same error message to prevent attacks. Ideally, each processing should consume the same amount of time, independent of the outcome, to impede side channel attacks.

SecurityParameters Every session establishes a configuration of cryptographic algorithms, parameters and key material. This configuration is called the `SecurityParameters`. The detailed specification of the `SecurityParameters` is given in Listing 2.1 which directly quotes RFC 2246 [43] (the structure slightly differs for each revision). Table 2.2 briefly describes each parameter.

```
1
2       struct {
3             ConnectionEnd            entity;
4             BulkCipherAlgorithm      bulk_cipher_algorithm;
5             CipherType               cipher_type;
6             uint8                    key_size;
7             uint8                    key_material_length;
8             IsExportable             is_exportable;
9             MACAlgorithm             mac_algorithm;
10            uint8                    hash_size;
11            CompressionMethod        compression_algorithm;
12            opaque                   master_secret[48];
13            opaque                   client_random[32];
14            opaque                   server_random[32];
15        } SecurityParameters;
```

Listing 2.1: `SecurityParameters` as defined by TLS 1.0 – *Source: RFC 2246 [43]*

It is important to note that not all values of the `SecurityParameters` can be reused for session resumption. The random values of client and server are always updated with every new connection attempt.

2.2.3. Handshake Layer

The *Handshake Layer* is defined by the *Handshake Protocol* – a very flexible, but complex protocol. It defines a set of messages, their order of occurrence and meaning. The handshake is necessary to establish algorithms, cryptographic primitives and key material by which the following, secured channel is protected. All messages are embedded into *Records* and can thus be protected by cryptographic means[9]. During the initial connection attempt the handshake messages are exchanged unencrypted until the *ChangeCipherSpec Protocol*[10]

[9] For convenience, multiple handshake messages may be grouped and sent within a single *Record* to reduce overhead of multiple header fields.

[10] Although the *ChangeCipherSpec Protocol* is used during a handshake it is not part of the *Handshake Protocol*, but rather a separate protocol consisting of only a single message.

Parameter	Description
`entity`	This parameter defines the role during the communication which can either be `server` or `client`.
`bulk_cipher_algorithm`	This specifies the algorithm used for encryption.
`cipher_type`	Holds the type of encryption algorithm (`stream`- or `block`-cipher).
`key_size`	The size of the secret part of the key material is specified with this parameter.
`key_material_length`	This parameter defines the whole length of the key material, including non-secret parts.
`is_exportable`	This value indicates if the encryption algorithm uses export weakened key sizes (`true`/`false`).
`mac_algorithm`	The algorithm to be used for MAC creation.
`hash_size`	This specifies the output size of the chosen MAC algorithm.
`compression_algorithm`	This parameter defines the used compression algorithm.
`master_secret`	This parameter holds the `master_secret` of this session.
`client_random`	The `client_random` value of this channel is specifed by this parameter.
`server_random`	This value defines the `server_random` of this channel.

Table 2.2.: `SecurityParameters` description.

is invoked to activate cryptographic protection at both parties – this message switches the state of the *Record Layer*. Handshake messages may only occur in an exactly specified order and follow the predefined workflow.
Every SSL/TLS connection starts with a mandatory handshake, even when session resumption is requested. Session resumption enables to *reuse* already established cryptographic setups. Although, the setup can be reused new key material has to be established/agreed on. The concept of session resumption is called abbreviated handshake and discussed later in this Subsection. To complicate handshakes even more, it is legal to renegotiate on a new setup within an already secured channel – this process is called *Renegotiation*. Caused by protocol design flaws *Renegotiation* led in the past to a serious attack on SSL/TLS (for details see Subsection 5.2.19) and is therefore possibly disabled by default[11].
A third protocol that is involved in a handshake is the *Alert Protocol*. As the name suggests the *Alert Protocol* is responsible for indicating errors. The errors can be either `fatal` which cause an immediate session termination and invalidation of all session associated parameters (keys, algorithms, etc.)[12] or only

[11] If Renegotiation is needed one might consider using [53].
[12] To be precise, the session identifier should be invalidated which renders the session unresumable. Running sessions are not affected.

a `warning` indicating problems which can be tolerated. An important *Alert Type* is the `close_notify` type. This alert smoothly terminates an active channel. Channels which are not terminated with a valid `close_notify` invalidate the session identifier and will be unresumable[13]. The explicit shutdown of a communication channel aims at preventing truncation attacks.

A schematic handshake is given in Figure 2.5. The messages mentioned in the Figure are described in the following. All explanation is based on RFC 2246 [43].

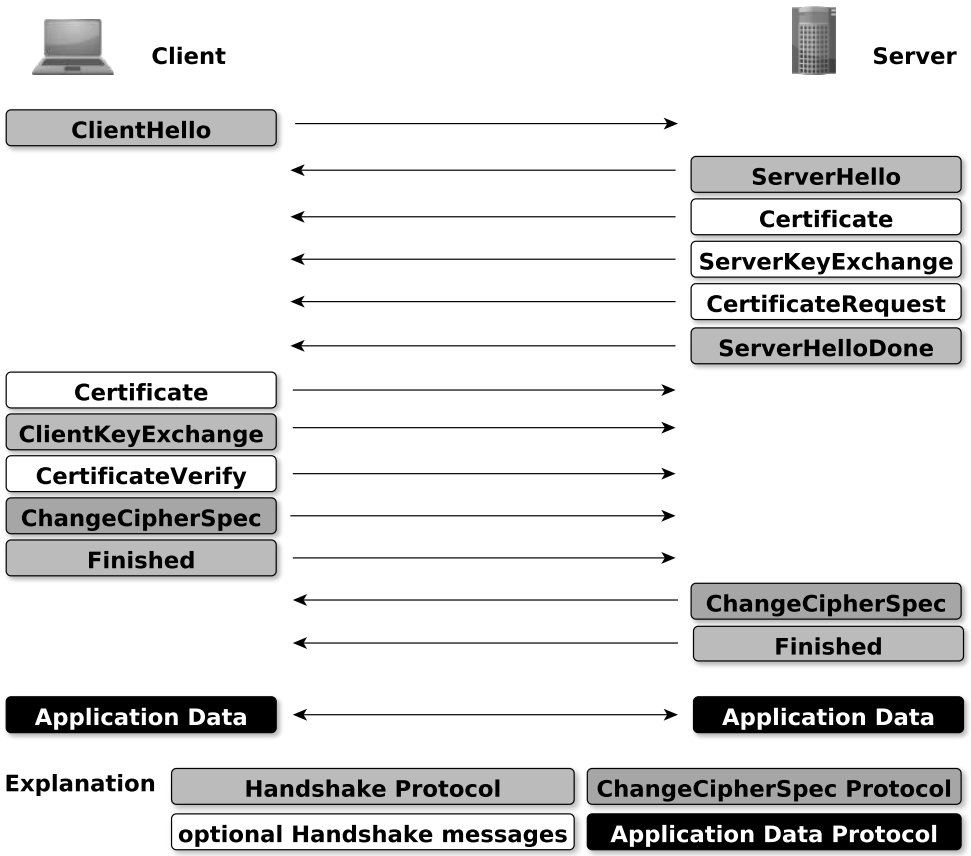

Figure 2.5.: TLS 1.0 Handshake – *based on Source: RFC 2246 [43]*

Handshake Protocol

The *Handshake Protocol* defines an additional, mandatory header field indicating the type (message) of the payload data. All messages of the *Handshake Protocol* carry this header field[14]. The payload data itself is one of the predefined messages which are discussed in the following. Protocol extensions are not

[13] This is only the case up to TLS 1.1 where this behaviour changed. Unclosed session are no longer required to be unresumable if not closed accurately.

[14] Please keep in mind that neither the *ChangeCipherSpec Protocol*, nor the *Alert Protocol* are part of the *Handshake Protocol*, although both are used during a valid SSL/TLS handshake.

covered here – for details on TLS Extensions please refer to [45], [54] and [55].

(HelloRequest) This message is sent by the server and requests a new handshake which the receiver (client) should initiate. A distinctive feature is the exclusion of this message from the hashing process. The hash is used in the `Finished` message to acknowledge that both sides sent and received the right messages and no Man-in-the-middle (MitM) altered the communication. Another distinctive feature that makes this message special is that it can be sent by the server at any time, even though it may be ignored:

> "This message will be ignored by the client if the client is currently negotiating a session. This message may be ignored by the client if it does not wish to renegotiate a session, or the client may, if it wishes, respond with a no_renegotiation alert." – *Source: RFC 2246 [43]*

Figure 2.6 depicts the message and its contents.

Figure 2.6.: `HelloRequest` message – *based on Source: RFC 2246 [43]*

ClientHello The `ClientHello` message is usually the first message of a handshake and informs the server about supported cryptographic algorithms (cipher suites), the highest available SSL/TLS version, supported compression algorithms (if any), a session id which either identifies an existing session or an empty session id field, if a new session should be started, and finally a random value consisting of 4 bytes of the current GMT UNIX time[15] and 28 randomly chosen bytes. The random value is of major importance when it comes to key derivation and finishing the handshake. Additionally, supported TLS extensions may be attached to the `ClientHello` message.

Figure 2.7 depicts the message and its contents.

ServerHello The `ServerHello` message follows on a `ClientHello` message and carries the chosen cipher suite, SSL/TLS version and compression algorithm. Furthermore, either a new session id, if the server is not willing to resume the desired session (chosen by the client) or the received session id

[15] Paul Kocher and Marsh Ray stated during personal communication that this is a reaction on the broken Netscape Pseudo-Random Number Generator (PRNG) flaw (for details see Subsection 5.2.1). The protocol should be protected from completely broken PRNGs creating static random values.

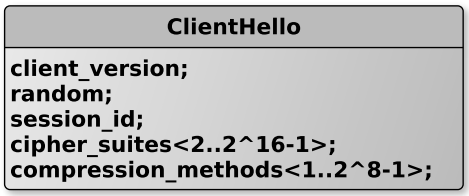

Figure 2.7.: `ClientHello` message – *based on Source: RFC 2246 [43]*

field was empty, or the same session id as received by the client, if the desired session will be resumed. And finally, a random value independent from the client's `ClientHello` message random value. As for the `ClientHello` message it is possible to add extensions.

Figure 2.8 depicts the message and its contents.

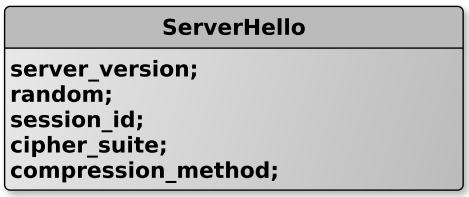

Figure 2.8.: `ServerHello` message – *based on Source: RFC 2246 [43]*

Certificate The `Certificate` message can be sent by server and client and is an optional message. Usually the server sends a `Certificate` message directly after the `ServerHello` message. The client should only send a `Certificate` message if requested by the server (in case mutual authentication is requested). Certificates are always required if the chosen key exchange algorithm is not anonymous. The `Certificate` message contains a certificate chain starting with the sender's certificate (of an appropriate key type, e.g. Rivest Shamir Adleman Cryptosystem (RSA)) and up to (optionally) its CA or the last self signed certificate. Certificates are usually of type X.509 (Version 3) (X509) and required to carry key material appropriate to the agreed key exchange algorithm. Table 2.3 shows the mapping from chosen key exchange algorithm to required key type, as defined in RFC 2246 [43].

Figure 2.9 depicts the message and its contents.

ServerKeyExchange The `ServerKeyExchange` message is sent by the server only in case one of the following key exchange algorithms is used:

- RSA_EXPORT (only if the server's public key is > 512 bit)
- DHE_DSS
- DHE_DSS_EXPORT

Key Exchange Algorithm	Required Key Type (carried within the certificate)
RSA	RSA public key flagged for encryption
RSA_EXPORT	Export restricted RSA public key flagged for either encryption or signing
DHE_DSS	Digital Signature Algorithm (DSA) public key
DHE_DSS_EXPORT	Export restricted DSA public key
DHE_RSA	RSA public key flagged for signing
DHE_RSA_EXPORT	Export restricted RSA public key flagged for signing
DH_DSS	Diffie Hellman Key Exchange (DH) public value, Digital Signature Standard (DSS) is used for signing
DH_RSA	DH public value, RSA is used for signing

Table 2.3.: Mapping of key exchange algorithm and required key type – *based on Source: RFC 2246 [43]*

Figure 2.9.: `Certificate` message – *based on Source: RFC 2246 [43]*

- DHE_RSA
- DHE_RSA_EXPORT
- DH_anon

The server is not allowed to send this message if any other algorithms are agreed-upon.

A `ServerKeyExchange` message carries additional parameters required by the chosen key exchange algorithm in order to be able to exchange or agree on a *PreMasterSecret* in the following steps. The encapsulated server parameters (DH or RSA) have to be signed, in case non-anonymous key exchange algorithms are used.

Figure 2.10 depicts the message and its contents.

CertificateRequest If mutual-authentication is used, this message signalizes the client that the server requests a certificate. The message carries information about the required certificate type and accepted CAs in form of a list of *Distinguished Names*.

Figure 2.11 depicts the message and its contents.

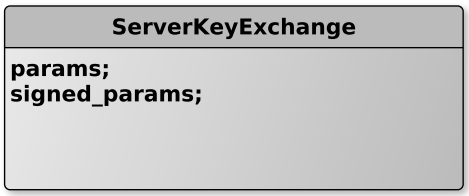

Figure 2.10.: `ServerKeyExchange` message – *based on Source: RFC 2246 [43]*

Figure 2.11.: `CertificateRequest` message – *based on Source: RFC 2246 [43]*

ServerHelloDone After the `ClientHello` and `ServerHello` messages are exchanged, the `ServerHelloDone` message indicates that the server's part of the key exchange/agreement is finished and the server is waiting for client messages, continuing the process. This message is for notification purpose only and contains no data fields.

Figure 2.12 depicts the message and its contents.

Figure 2.12.: `ServerHelloDone` message – *based on Source: RFC 2246 [43]*

ClientKeyExchange The `ClientKeyExchange` message carries, dependent on the negotiated cipher suite, either directly an RSA encrypted *PreMasterSecret* or the counterpart required for key agreement (the client's public value for DH(E)). If RSA is chosen for key exchange the message contains a *PreMasterSecret* consisting of two bytes identifying the newest supported protocol version of the client[16] and 46 randomly chosen bytes. The 48 *PreMasterSecret* bytes are encrypted with the server's RSA public key of the previously exchanged certificate or with the temporary key that was part of the `ServerKeyExchange`

[16]To counter Version Rollback Attacks (see Subsection 5.2.6 for details). By attaching the protocol version to the *PreMasterSecret* the server can detect a Version Rollback Attack to the weaker SSL 2.0 protocol.

message. In case of key agreement (DH or DHE) two possible cases are possible (for details see RFC 2246 [43] 7.4.7.2.):

1. Implicit DH public value: Public value is contained in the previously exchanged certificate (contained in the `Certificate` message).
2. Explicit DH public value: Public value is part of this message.

In the case of an implicit public value the message will contain no data fields, where as in the other case the message contains the required public value (see Table 2.4).

Case	**Message Content**
Implicit	No data
Explicit	Public value

Table 2.4.: Message content for implicit/explicit DH Public Value.

Figure 2.13 depicts the message and its contents.

Figure 2.13.: `ClientKeyExchange` message – *based on Source: RFC 2246 [43]*

CertificateVerify With the `CertificateVerify` message a client proves its posession of the corresponding private key to the previously sent certificate (as part of the `Certificate` message, sent by the client). Therefore, a signature over all yet exchanged handshake messages[17], excluding this messages, is part of this message. The hash algorithm used for hash value computation is defined by the `ServerKeyExchange` message and can either be Secure Hash Algorithm (SHA) or Message-Digest Algorithm (MD)5 (in the case of TLS 1.0). The hash value is accordingly digitally signed with the private key belonging to the certificate.

Figure 2.14 depicts the message and its contents.

Finished The `Finished` messages of client and server (not necessarily in this order) indicate the completion of the *Handshake Phase*. After reception of valid `Finished` message both participants are ready to exchange application data – this is the start of the *Application Data Phase*. The `Finished` messages are the first messages of this communication which are protected by the negotiated

[17] Without the *Record Layer* header fields.

Figure 2.14.: `CertificateVerify` message – *based on Source: RFC 2246 [43]*

algorithms and key material (e.g. encryption...). `Finished` messages perform multiple tasks:

- Confirm what was sent/received
- Confirm the exchanged/agreed key material

Although not part of the Handshake Protocol, it is mandatory to respect the correct order of messages: no ***`Finished`*** *message must be sent before reception of a* ***`ChangeCipherSpec`*** *message. Additionally, it is very important that each communication party verifies the correctness of the received* ***`Finished`*** *message and the validity of its contents.*

Part of every `Finished` message is a specially crafted `verify_data` field. This field is build according to Listing 2.2 and is the output of the *PRF* function. The *PRF* function takes the following values as input:

- two hashes – one of type SHA-1 and another one of type MD5 (for TLS 1.0) – over all yet exchanged messages of the *Handshake Protocol*[18,19]
- the *MasterSecret* – a derivation of the *PreMasterSecret*
- a `finished_label` which is either `"client finished"` if this message is sent by the client or `"server finished"` if it is sent by the server

All values are used as input to the pseudo-random function `PRF` which output is taken as value of the `verify_data` field. For convenience the following description is taken from RFC2246 [43]:

```
1  verify_data = PRF(master_secret, finished_label,
2      MD5(handshake_messages) + SHA-1(handshake_messages)) [0..11];
```

Listing 2.2: `verify_data` computation as defined by TLS 1.0 – *based on Source: RFC 2246 [43]*

The pseudo-random function and key derivation process will be discussed in detail later in this Subsection.

Figure 2.15 depicts the message and its contents.

Since the `Finished` message is the first protected message Figure 2.16 shows the complete structure including the `Record Layer` header fields (the example assumes that a cipher suite with encryption support was chosen).

[18] Bearing in mind that `ChangeCipherSpec` and `Alert` messages are NOT part of the *Handshake Protocol* and thus must not be part of the hashes.

[19] Without the *Record Layer* header fields.

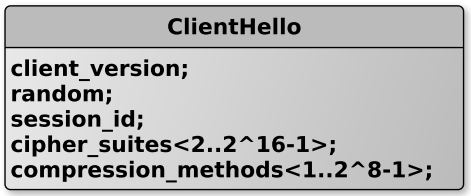

Figure 2.15.: `Finished` message – *based on Source: RFC 2246 [43]*

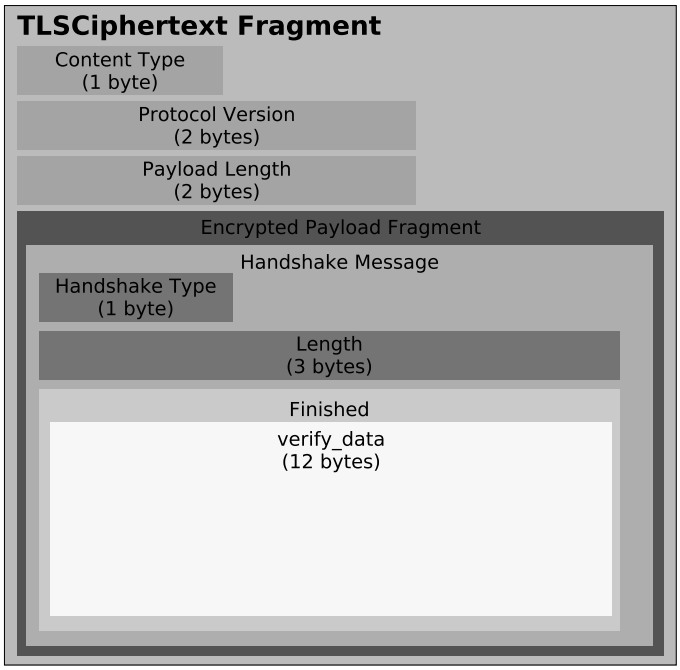

Figure 2.16.: Encrypted `Finished` message – *based on Source: RFC 2246 [43]*

ChangeCipherSpec Protocol

The *ChangeCipherSpec Protocol* consists of only a single message which indicates that the communication party which origins the message switches to the pending state of negotiated algorithms and parameters. This message notifies the receiving party that the sender will switch to message protection after this message.

ChangeCipherSpec Message This message contains only a single 0x01 byte. Figure 2.17 depicts the message and its contents.

Alert Protocol

The *Alert Protocol* is used to communicate errors of various type and severity. To distinguish between errors that can be tolerated and critical errors, the protocol defines two different types of errors:

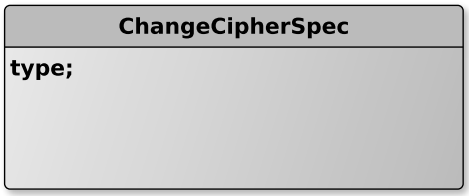

Figure 2.17.: `ChangeCipherSpec` message – *based on Source: RFC 2246 [43]*

- `warning`
- `fatal`

While an error of type `warning` can be tolerated, errors of type `fatal` lead to immediate termination of the current session and invalidate the associated session id. The invalidation of the session id only affect future session resumption attempts. Running sessions remain unaffected and continue.

All *Alert Protocol* messages are encapsulated by the *Record Protocol* and thus protected by the currently negotiated means.

Alert Message An `Alert` message consists of the alert type (which is either `warning` or `fatal`) and a predefined alert description in form of an unique identifier. TLS 1.0 defines 23 different alert descriptions (see RFC 2246 [43]).

A special alert is the closure alert (`close_notify` , identifier 0). To avoid session truncation on an unused, but alive communication channel (an established and running SSL/TLS session) both parties must be able to inform when a communication channel is no longer needed. In this case, both parties should shutdown the running conversation – each attempt to exchange any further data is very likely to be a truncation attack. According to the specification of SSL 3.0 and TLS 1.0, it is necessary that both parties send a closure alert. The first alert's origin may stop its read operation on the socket before reception of the acknowledging closure alert from its counterpart. A session termination without closure alert is required to be unresumable[20].

Figure 2.18 depicts the message and its contents.

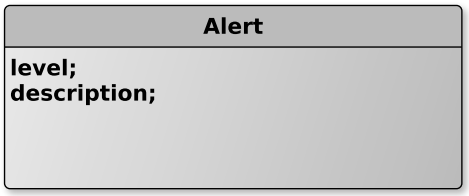

Figure 2.18.: `Alert` message – *based on Source: RFC 2246 [43]*

[20]This behaviour changes in TLS 1.1.

Handshake for RSA based Authenticated Key Exchange (TLS-RSA)

The valid handshake messages for an RSA based key exchange are depicted in Figure 2.19.

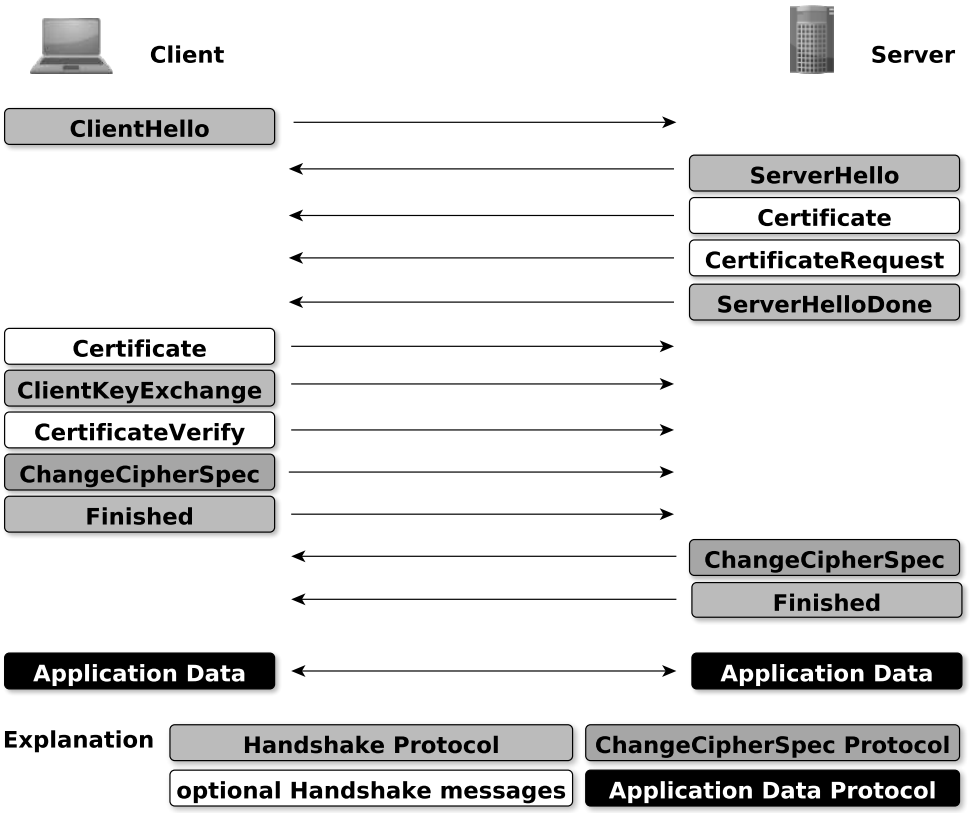

Figure 2.19.: RSA based authenticated key exchange TLS 1.0 Handshake – *based on Source: RFC 2246 [43]*

In this scenario the *PreMasterSecret* is exclusively chosen by the client and sent as an RSA encrypted (and Public-Key Cryptography Standards (PKCS)#1 v1.5 encoded) data block to the sever. The server cannot influence the *PreMasterSecret* generation.

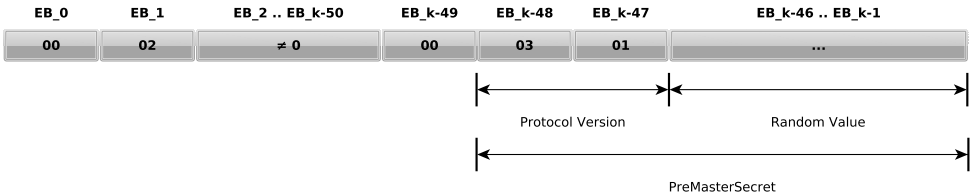

Figure 2.20.: PKCS encoded *PreMasterSecret* – *based on Source: RFC 2246 [43]*

The PKCS#1 v1.5 encoding is explained in Subsection A.1.2 of the Ap-

pendix. When additionally taking into account the predefined structure of the RSA encrypted *PreMasterSecret* (see RFC 2246 [43]), a valid structure has to fulfil the preconditions of Table 2.5 which is graphically presented in Figure 2.20. The PKCS structure depends on the size of the RSA modulus, n (see Subsection A.1.1). Therefore, $k = \lceil \frac{\log_2(n)}{8} \rceil$.

Byte	**Value**
EB_0	00
EB_1	02
$EB_2..EB_{k-50}$	$\neq 00$
EB_{k-49}	00
EB_{k-48}	Major protocol version
EB_{k-47}	Minor protocol version
$EB_{k-46}..EB_k$	Random value

Table 2.5.: PKCS encoded *PreMasterSecret – based on Source: RFC 2246 [43]*

Handshake for DH based Authenticated Key Exchange (TLS-DH)

In the DH based key agreement, also known as *static Diffie-Hellman*, both parties present either a certificate containing the public DH value or at least the server presents a certificate with its public DH value and the client chooses a value each session. Figure 2.21 illustrates the handshake message flow for DH based key agreement. The negotiated DH key is used as *PreMasterSecret*. As the parameters in the certificates are static and not chosen on-the-fly the *PreMasterSecret* is exactly the same for every connection (but the derived `MasterSecret` differs, as it contains changing values from the `ClientHello` and `ServerHello` messages). It is obvious that both parties have to present values based on the same public DH parameters (moduli and generator).

Both parties participate on the generation of the *PreMasterSecret*. It is advisable to check if the DH parameters are somehow *good*, for example by checking a list of valid groups with valid generators and moduli.

Handshake for DHE based Authenticated Key Exchange (TLS-DHE)

The handshake for ephemeral DH based authenticated key exchange is illustrated in Figure 2.22. Please note the mandatory `ServerKeyExchange` message in this handshake. The authentication of the exchanged parameters is done by using digital signatures. For this purpose SSL/TLS support either RSA or DSS as signature algorithms. The necessary DH parameters are send together with a signature protecting the integrity within the `ServerKeyExchange` message. The client returns in response the public DH parameters within the `ClientKeyExchange` message, but without an integrity protecting signature. The negotiated DH key is used as *PreMasterSecret*.

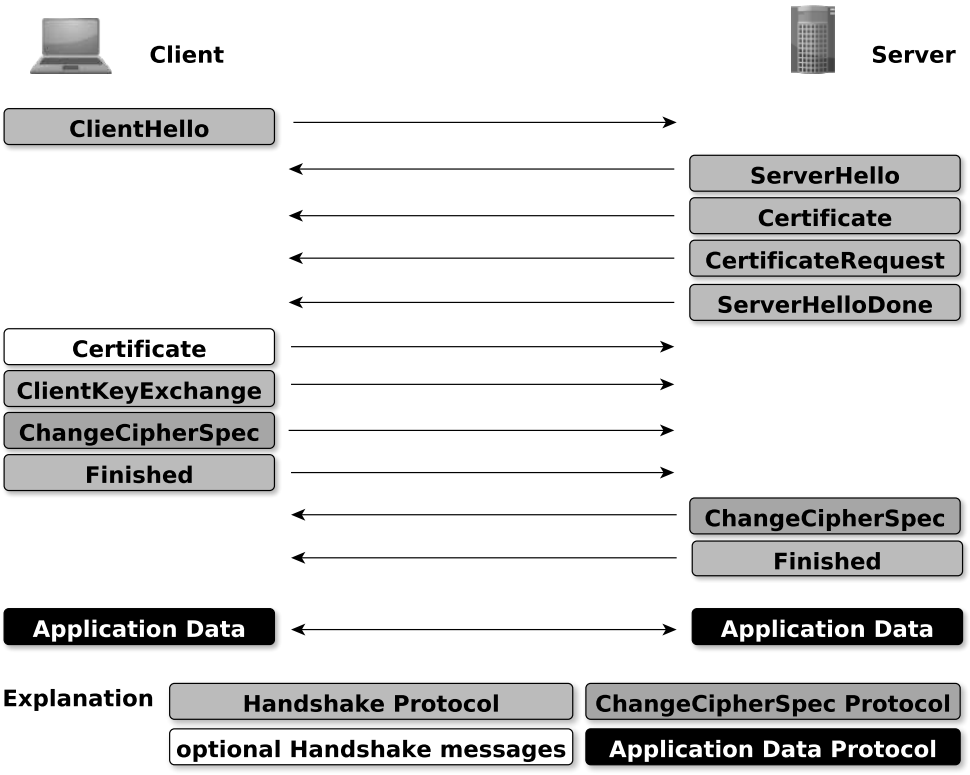

Figure 2.21.: DH based authenticated key exchange TLS 1.0 Handshake – *based on Source: RFC 2246 [43]*

Both parties participate on the generation of the *PreMasterSecret*. It is advisable to check if the DH parameters are somehow *good*, for example by checking a list of valid groups with valid generators and moduli.

Abbreviated Handshake through Session Resumption

Figure 2.23 shows an abbreviated TLS 1.0 (see RFC 2246 [43] for details) handshake. Compared to the full handshake in Figure 2.5 the key agreement/exchange messages are missing, because both parties already know a *MasterSecret* which is linked to the session id. With the existing *MasterSecret* and the (new) random values of the `ClientHello` and `ServerHello` messages, both parties are able to compute fresh key material. This ensures always fresh keys – even for a resumed session the used key material is different (the key derivation process is discussed later in this Subsection).

The server signalizes willingness to resume a session by responding with the client provided session id in the `ServerHello` message. If the server is unable or not willing to resume a session it will respond with a new session id in the `ServerHello` message – this causes a new and complete (non-abbreviated) handshake.

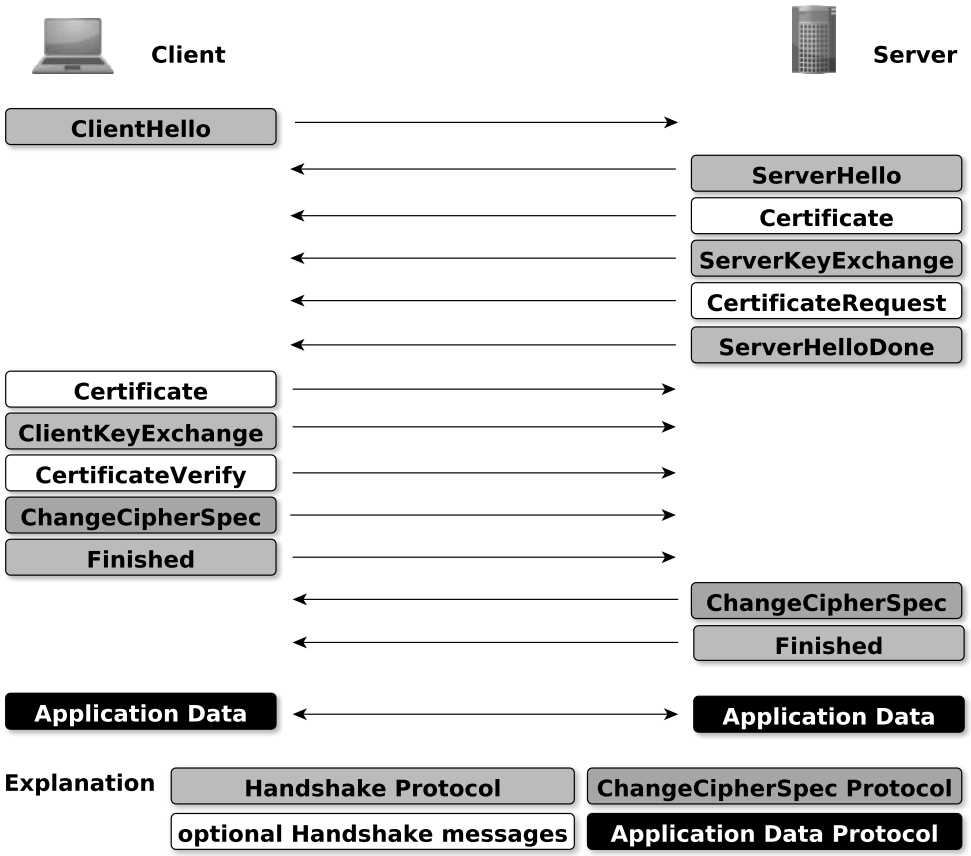

Figure 2.22.: DHE based authenticated key exchange TLS 1.0 Handshake – *based on Source: RFC 2246 [43]*

Pseudo-random Function

The pseudo-random function is a conceptual block of SSL/TLS, mostly referred to as `PRF`. It is used for the expansion of secrets and is a crucial part of the key derivation process. In TLS 1.0 the function is based on two hardcoded hash functions (MD5 and SHA-1) which are used for HMAC computations[21]. All further discussion is based on RFC 2246 [43].

The `PRF` is defined in Listing 2.3 and takes three input values:

1. A secret value which has to be split in two halves
2. An identifying label
3. A seed value

```
1 PRF(secret, label, seed) = P_MD5(S1, label + seed) XOR
2           P_SHA-1(S2, label + seed);
3
```

[21] This changes in TLS 1.2.

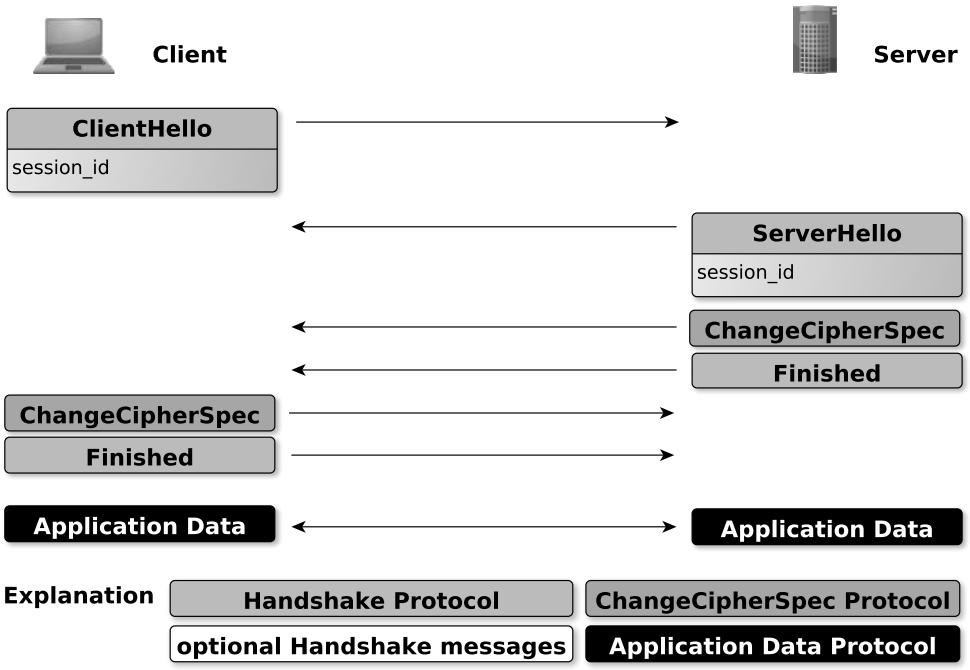

Figure 2.23.: Abbreviated TLS 1.0 Handshake – *based on Source: RFC 2246 [43]*

```
4  L_S = length in bytes of secret;
5  L_S1 = L_S2 = ceil(L_S / 2);
6
7  S1 = secret[0..L_S1];
8  S2 = secret[L_S2..L_S];
9    S1 and S2 are the two halves of the secret and each is the same
10   length. S1 is taken from the first half of the secret, S2 from
11   the second half. Their length is created by rounding up the
12   length of the overall secret divided by two; thus, if the
13   original secret is an odd number of bytes long, the last byte of
14   S1 will be the same as the first byte of S2.
```

Listing 2.3: Pseudo-random function as defined by TLS 1.0 – *based on Source: RFC 2246 [43]*

A graphical representation of the pseudo-random function can be found in Figure 2.24.

The expansion functions `P_MD5` and `P_SHA-1` are used to expand the secret value. The core operation is an iterated HMAC computation with chained input data. Listing 2.4 gives the definition of RFC 2246 [43].

```
1    P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
2                           HMAC_hash(secret, A(2) + seed) +
3                           HMAC_hash(secret, A(3) + seed) + ...
4
5    Where + indicates concatenation.
6
7    A() is defined as:
```

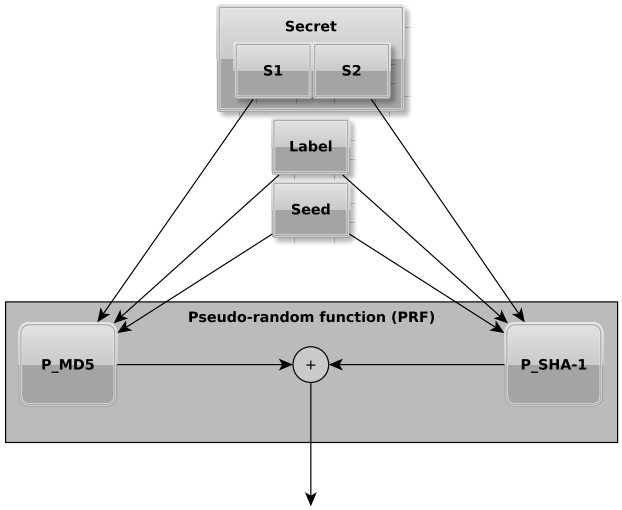

Figure 2.24.: Pseudo-random function – *based on Source: RFC 2246 [43]*

```
8         A(0) = seed
9         A(i) = HMAC_hash(secret, A(i-1))
```

Listing 2.4: Data Expansion Function as defined by TLS 1.0 – *Source: RFC 2246 [43]*

A graphical representation of the expansion function can be found in Figure 2.25.

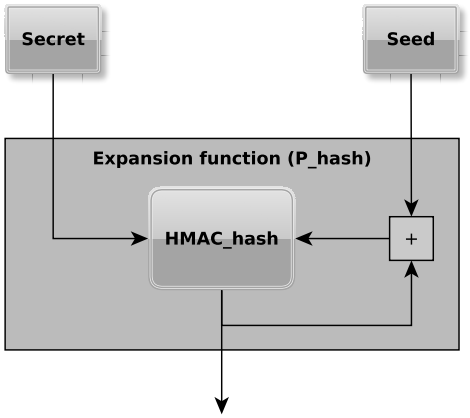

Figure 2.25.: Expansion function – *based on Source: RFC 2246 [43]*

The HMAC computation is iterated until the desired amount of expansion data is gathered. The excess data of the final iteration is simply discarded. A single iteration generates data corresponding to the output length of the hash function.

An example: A single `HMAC_MD5` iteration returns 128 bit = 16 bytes. If 100 bytes are required the function has to be called seven times leading to 112 output bytes ($7 * 16 = 112$), where the last 12 bytes of the 7th iteration are discarded.

Derivation of Key Material

The derivation of sufficient key material (SSL/TLS defines multiple keys for different cryptographic primitives, such as encryption and integrity protection) is done by utilizing the Pseudo-Random Function defined in Subsection 2.2.3 (this covers only TLS 1.0). All further explanation of this Subsection is based on RFC 2246 [43].

MasterSecret Computation The *MasterSecret* is derived from the *PreMasterSecret* (which is established during the handshake) and the random values of client and server which are part of the `ClientHello` and `ServerHello` messages. Each *MasterSecret* is of exactly 48 bytes and used for further derivation of key material. To compute the *MasterSecret* the *PreMasterSecret*, the random values of client and server and the label `"master secret"` are passed to the pseudo-random function. Listing 2.5 details the computation (+ denotes the concatenation operator).

```
1     master_secret = PRF(pre_master_secret, "master secret",
2                         ClientHello.random + ServerHello.random)
```

Listing 2.5: *MasterSecret* computation as defined by TLS 1.0 – *Source: RFC 2246 [43]*

MasterSecret Key Derivation and Splitting

After computation, the *MasterSecret* is expanded and finally split into four different keys and two IVs. The `PRF` is used to expand the amount of key material where the *MasterSecret* provides the entropy, salted with client and server provided random values. The process is depicted in Listing 2.6. The specification denotes the random values as `SecurityParameters.client_random` and `SecurityParameters.server_random` which are stored as values of the session dependent `SecurityParameters`. Values in quotation define labels (simply strings) and the concatenation operator is +. `SecurityParameters.hash_size` is the output size of the hash algorithm used for message authentication (as defined by the used cipher suite, `SecurityParameters.key_material_length` identifies the key length and `SecurityParameters.IV_size` defines the size of the IVs to be used, if any (in case of block ciphers). The IVs are only valid if block ciphers are used.

```
1   To generate the key material, compute
2
3       key_block = PRF(SecurityParameters.master_secret,
4                       "key expansion",
5                       SecurityParameters.server_random +
6                       SecurityParameters.client_random);
```

```
7
8     until enough output has been generated. Then the key_block is
9     partitioned as follows:
10
11          client_write_MAC_secret[SecurityParameters.hash_size]
12          server_write_MAC_secret[SecurityParameters.hash_size]
13          client_write_key[SecurityParameters.key_material_length]
14          server_write_key[SecurityParameters.key_material_length]
15          client_write_IV[SecurityParameters.IV_size]
16          server_write_IV[SecurityParameters.IV_size]
```

Listing 2.6: Key derivation as defined by TLS 1.0 – *Source: RFC 2246 [43]*

Exportable algorithms require further processing. Please refer to RFC 2246 [43] for details on the special key derivation process in TLS 1.0[22].

The whole TLS 1.0 key derivation process is graphically depicted in Figure 2.26.

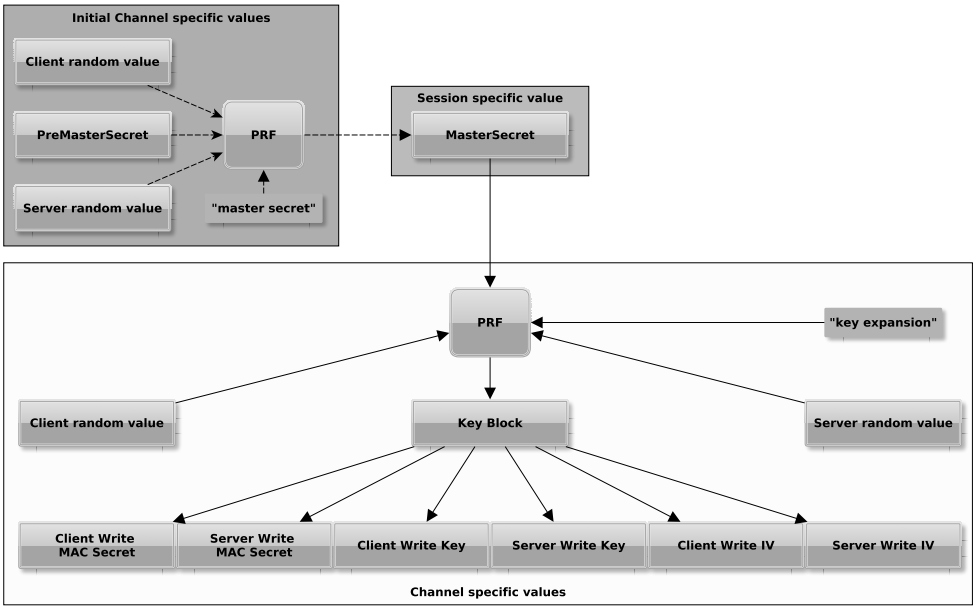

Figure 2.26.: Key derivation process of TLS 1.0 – *based on Source: RFC 2246 [43]*

[22]The mandatory support for export weakened ciphers was removed in TLS 1.1.

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

C.A.R. Hoare

3 From SSL to TLS

This section sketches the evolution of SSL/TLS and details the particular changes. I would like to thank Martin Abadi and Kipp Hickman who provided the earliest drafts of SSL. Without their help this chapter would definitely be incomplete.

SSL went through several revisions to the currently latest version TLS 1.2. The development was mostly driven by security concerns, support for more and stronger cipher suites and further enhancements. All versions have in common that the communication process is divided into two phases: the Handshake phase and the Application phase, see Subsection 2.1.2.

3.1. SSL 1.0

The first version of the protocol (suite), designed by Kipp E.B. Hickman of Netscape Communications Corp., was never made public due to conceptual weaknesses with regards towards security. Actually there has never been anything like a "Version 1.0" specification, to the best of my knowledge (the `SSL_CLIENT_VERSION` and `SSL_SERVER_VERSION` codes are already set to `0x0002` in the earliest available documents). Therefore, it is hard to subdivide all improvements to the protocol to a specific release. As a consequence, the history of SSL 1.0 is, differing to the following sections, a flowing illustration of it's development. All documents, kindly provided by Kipp E.B. Hickman and Martin Abadi, are dated November 1994.

The major goal of SSL version 1.0 was to establish a confidential communication channel over an insecure network. Integrity protection was not yet in focus. The protocol suffered from conceptual weaknesses related to authentication and the complete lack for timestamps or sequence numbers to prevent packet replays. As a consequence a later revision (which also did not see the light of public day) added sequence numbers and a Cyclic Redundancy Check (CRC) checksum as an enhancement addressing integrity issues. An interesting

fact is that the same revision removed the support for Data Encryption Standard (DES) and TripleDES. The first one because of security concerns (key length of 64 bits with a real strength of 56 bit was not sufficient, even in 1995) and the second one because of the fact that the specification relied on MD5 for key generation[1]. Additional reasons for removing DES and TripleDES were US export regulations for strong cryptography. The last revision replaced the CRC with a MAC based on MD5 (no HMAC, yet). The available cipher suites were either based on Ron's Cipher (RC)2, RC4 or International Data Encryption Algorithm (IDEA). Although SSL was still in an early specification phase it already supported features such as session-resumption and client authentication. The handshake phase and the messages are different from those used in SSL 3.0 and above. Please note the complete abscence of the `ChangeCipherSpec` protocol. `Alert` messages were already part of the specification. Figure 3.1 illustrates the full handshake of SSL 1.0 without abbreviation.

Both parties participate on the key derivation process, since the client chooses the *MasterSecret* and the challenge data, but the server dictates the connection id. All of the three values are mandatory inputs to the MD5 based key derivation function.

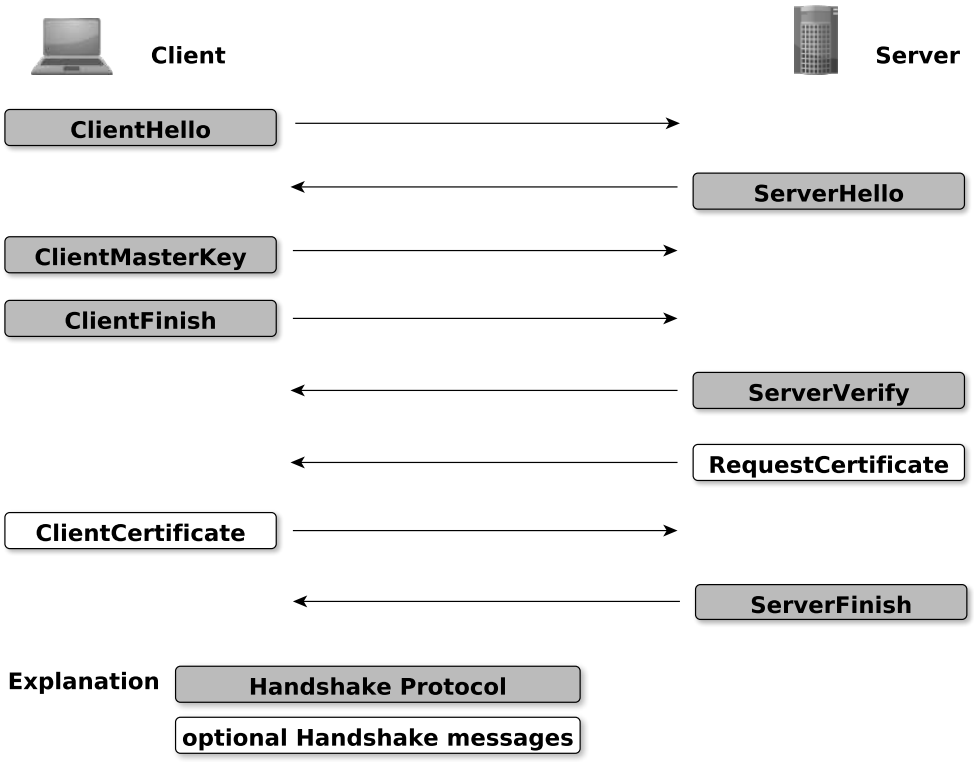

Figure 3.1.: (Non-abbreviated) Handshake protocol of SSL 1.0 – *Source: The SSL Protocol, Nov. 25th, 1994*

[1]MD5 returns a 128 bit hash value while TripleDES requires 168 key bits (3*56 real key bits) – although three different keys are used the real key strength is limited to 112 bit due to a meet-in-the-middle attack [56].

ClientHello The `ClientHello` message, depicted in Figure 3.2, contains a message identifier, its own SSL version, length fields for the contained data (2 byte for each length field), a list of session cipher specs (cipher kind identifier and two key length bytes), the session id and some challenge data. The list of supported ciphers and the challenge data must not be zero, while the session id is optional and only sent if the client could find a session in its cache for the server, in this case a session id is of exactly 16 bytes length. Otherwise, the length field is set to zero and no data is attached for the session id. The challenge data must be of length between 16 and 32 bytes and is used to provide key confirmation: the server has to respond with the encrypted (PKCS#1 encoded) representation of this challenge in the `ServerVerify` message (the newly agreed on keys should be used). This message is always the first message in a SSL 1.0 communication.

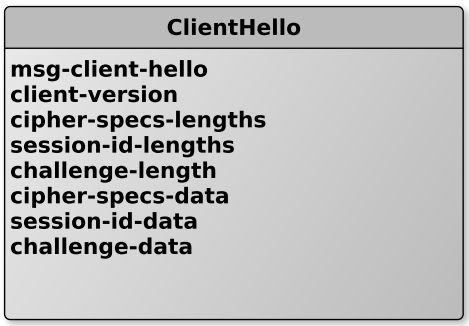

Figure 3.2.: `ClientHello` message of SSL 1.0 – *Source: The SSL Protocol, Nov. 25th, 1994*

The only supported cipher kinds (encryption algorithms) are:

- RC4
- Export weakened RC4 (40 bytes)
- RC2 in Cipher Block Chaining (CBC) mode
- Export weakened RC2 in CBC mode (40 bytes)
- IDEA in CBC mode

ServerHello The `ServerHello` message is the direct response to the client initiated `ClientHello` message and contains a message identifier, a boolean flag which indicates if the server is able/willing to resume a session, a certificate type (1 byte), the servers SSL version (2 bytes), length fields for the certificate data, the cipher specs data and connection id fields. If the server indicates that the session will be resumed the length fields for the certificate and cipher specs will be zero and the certificate type field as well. In this early version of SSL only X509 is defined as certificate type (but can be extended by defining new types). If this session is resumed (the session id hit field is non-zero), client and server recompute a new set of keys. These (session) keys are derived from the

MasterKey (which is yet present for both parties on a resumed session, since it has already been established in a previous handshake), the challenge data, the connection id and a boolean value (indicating if the derived value will be the write or the read key) by the use of MD5. Otherwise, the server will receive the (encrypted) *MasterKey* as part of the `ClientMasterKey`. In a non-resumed case, the server also issues the connection id (randomly) which is later on used for key confirmation. Finally, the certificate data field contains the server's certificate including its public key (in the non-resumed case) which is later on used to encrypt the client-chosen *MasterKey*. The cipher spec data contains a list of session cipher specs the server is willing to accept (cipher kind (1 byte) and key length (2 bytes)).

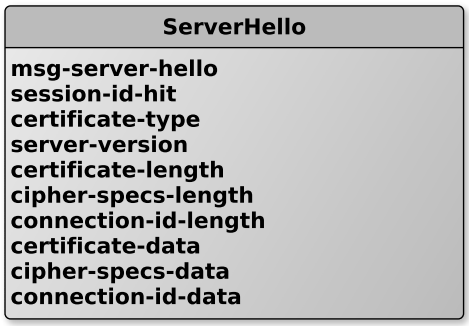

Figure 3.3.: `ServerHello` message of SSL 1.0 – *Source: The SSL Protocol, Nov. 25th, 1994*

Figure 3.3 illustrates the `ServerHello` message.

ClientMasterKey The `ClientHello` message, depicted in Figure 3.4, contains a message identifier, the index of the chosen cipher kind (the index is selected according to the list of chosen ciphers by the server in the `ServerHello` message) (1 byte), the key length (2 bytes), the non-secret part of a *MasterKey* (e.g. parity bytes), the secret bytes of the *MasterKey* encrypted with the server's public key and optionally key arguments, such as for example an IV. The session keys are derived from the *MasterSecret* as mentioned in the `ServerHello` message paragraph.

This message must be sent if the server was either not able or willing to resume the session (session ID hit == 0).

ClientFinished Although specified as `ClientFinish` in the handshake workflow diagram (see Figure 3.1) the message is named `ClientFinished`. Part of this message is the message identifier and the mirrored connection id (which was sent by the server in its `ServerHello` message).

ServerVerify Figure 3.6 shows the `ServerVerify` message which contains only a message identifier and the encrypted, client-provided challenge data (which was received with the `ClientHello` message). he challenge data is encrypted

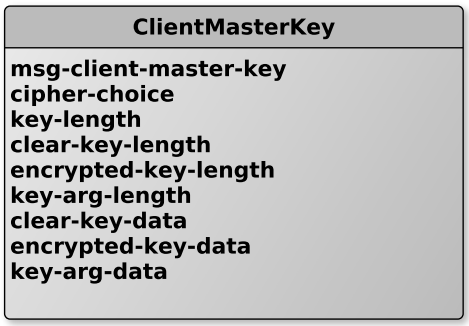

Figure 3.4.: `ClientMasterKey` message of SSL 1.0 – *Source: The SSL Protocol, Nov. 25th, 1994*

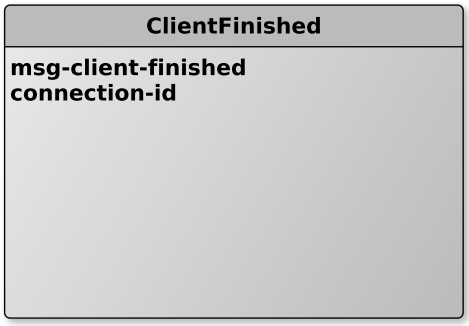

Figure 3.5.: `ClientFinished` message of SSL 1.0 – *Source: The SSL Protocol, Nov. 25th, 1994*

with the established set a session keys. The purpose of this message is the conformation of the established keys. Additionally, this key conformation implicitly proves that the server is in posession of the private key belonging to the sent certificate (because only the owner of the private key is able to decrypt the client-chosen *MasterKey* – necessary for key derivation – from the `ClientMasterKey` message).

The `ServerVerify` message is the first message during a handshake that is sent encrypted with the newly established keys.

RequestCertificate The `RequestCertificate` illustrated in Figure 3.7 contains a message identifier, an authentication type and certificate challenge data (16 to 32 bytes) that has to be signed by the client in its response message. This message initiates mutual authentication. The server is already authenticated, but forces the client to authenticate, too. This specification only defines an authentication message based on RSA signed MD5 hash values of the challenge data, but can easily be extended.

ClientCertificate This message is sent encrypted (with the *MasterKey* and only valid if requested by the server (`RequestCertificate` message received).

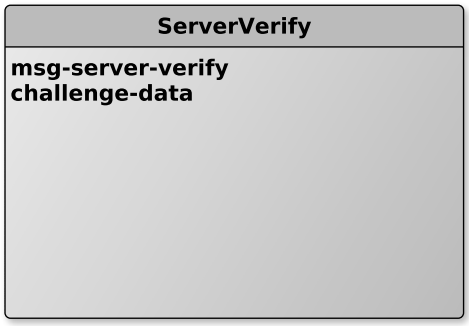

Figure 3.6.: `ServerVerify` message of SSL 1.0 – *Source: The SSL Protocol, Nov. 25th, 1994*

Figure 3.7.: `RequestCertificate` message of SSL 1.0 – *Source: The SSL Protocol, Nov. 25th, 1994*

It contains a message identifier, the contained certificate type, length fields (2 bytes) for the certificate and response data fields. The certificate data contains the certificate and the response data contains a digital signature (in the RSA case) over the hash of the concatenated client's read and write keys, the challenge data of the `RequestCertificate` message and finally the server's certificate. The signature is PKCS#1 encoded.

Once again, it can be seen that the protocol was intended and specially designed to be easily extendable.

ServerFinished This message contains a message identifier and a the (server-chosen) session identifier that could be used to resume this session. For this purpose both parties have to store at least this session identifier and the associated *MasterKey* in their local caches.

The message structure is shown in Figure 3.9.

3.2. SSL 2.0

The second revision of the SSL protocol [39] of April 1995 was the first published specification of the protocol and at the same time patented by Taher ElGamal

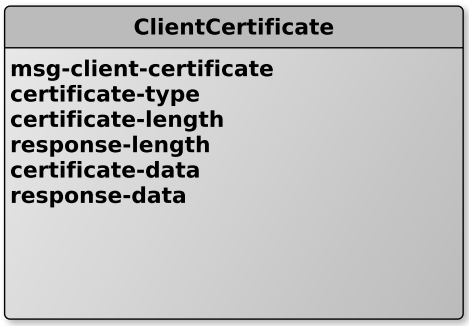

Figure 3.8.: `ClientCertificate` message of SSL 1.0 – *Source: The SSL Protocol, Nov. 25th, 1994*

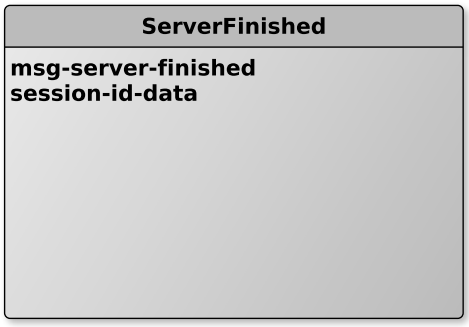

Figure 3.9.: `ServerFinished` message of SSL 1.0 – *Source: The SSL Protocol, Nov. 25th, 1994*

and Kipp E.B. Hickman. Revision 2 was very equal to the prior version and the result of ongoing enhancement. However, the protocol suffered from multiple weaknesses, mostly related to the handshake phase:

- The MAC does not cover the padding length field (see Subsection 5.2.2)
- No signalization functionality if the a connection should be shut down (this feature was introcuded with SSL 3.0 [40] and discarded with the release of TLS 1.2 [44])
- Missing handshake authentication, an attacker could simply alter the list of supported cipher suites and force the use of weaker ones (see Subsection 5.2.3)
- Usage of weak, partly vulnerable algorithms(MD5 – see Subsection 5.2.14, RC4 – see Subsection 5.2.37) or problematic mode of operation (CBC – see Subsections 5.2.8, 5.2.13, 5.2.15, 5.2.25)
- Keys are used for multiple cryptographic primitives (integrity protection and encryption use the same keyset)

RFC 6176 [57] detailly points out why the use of SSL 2.0 should be avoided.

The handshake workflow is equal to the one of SSL 1.0, but the specification reintroduces DES and TripleDES cipher suites. Further differences are of minor nature (such as changes in the naming conventions, e.g. in the cipher specs data structure). Changes are also obvious in the key derivation section, because different key types require different key derivation algorithms (DES and TripleDES need special treatment). The maximum *MasterKey* length doubled with version 2 to 256 bits.

3.3. SSL 3.0

SSL 3.0 [40] addressed many vulnerabilities in the prior revisions by introducing a variety of new features and mechanisms. Now, the handshake consists of additional (partly optional) messages and a completely new protocol named `ChangeCipherSpec`. Furthermore, the alert messages are now part of a designated `Alert` protocol. Additionally, the nomenclature of messages, message parts and structures changed, as well as the protcol and message structures. Thus, the differences between SSL 2.0 and 3.0 are immense. The overall impression is that the protocol evolved to be more structured and designed in layers (message and protocol encapsulation, ...), missing features were added and criticism with regards towards security were addressed. In spite of all the differences, a particular design goal was backwards compatibility with SSL 2.0. This is the reason why e.g. the `ClientHello` messages of both revisions appear to be very equal.

The following list briefly summarizes the security enhancements compared to SSL version 2.0:

- More cipher suites including new cryptographic algorithms (DSS, SHA-1).
- Support for key agreement (DH and FORTEZZA) during the handshake phase.
- Renegotiation of cryptographic parameters at any time of a running session.
- Optional message compression support.
- Integrity protection through MACs with configurable algorithms[2]: MD5 or SHA-1. Additionally, the MAC now covers the length of the padding[3].
- Different keys for different cryptographic primitives. Message integrity and message encryption are now using separate key material[4].
- The new `Alert` protocol is responsible for signalizing any errors of two categories: `Warning` and `Fatal` alerts[5].

[2] These are no HMACs yet!
[3] As a consequence of the Padding Manipulation Attack (see Subsection 5.2.2)
[4] Which was a major concern, especially when using weak export ciphers
[5] `Fatal` alerts immediately terminate and invalidate the running session.

- Termination of a connection must now be communicated to each other with a special `close_notify` alert. Closing a connection without notification invalidates the session (no longer resumable).

- Support for certificate chains instead of only a single certificate.

- Handshake protection by authenticating all sent and received handshake messages. Now, a manipulation of the handshake is immediately detected when the `Finished` messages are verifed[6].

- Introduction of the `ChangeCipherSpec` protocol to activate changes to the current security parameter configuration (switch to the pending state).

- The encrypted `PreMasterSecret` contains the currently used version number preventing Version Rollback attacks (see Subsection 5.2.6)[7].

- Support for unauthenticated (anonymous) sessions.

Since the differences between TLS 1.0 and SSL 3.0 are of minor, a detailed presentation is skipped. In favour, a detailed presentation of TLS 1.0 is given in Chapter 2.

3.4. TLS 1.0

TLS 1.0 is a re-branded adaption of SSL 3.0 with some minor changes. The name change was part of the standardization process by the IETF. Internally, TLS 1.0 has the version number (SSL) 3.1 to refer to the only minor changes. TLS 1.0 was the first revision with Netscape Communications Inc. not being the leading company behind the specification process. The *refurbished* revision includes the detailed specification of a pseudo-random function which is used for key derivation purposes. In detail, the function is used for the expansion of key material and digest computation in the `Finished` message. Beyond that, changes on the `Alert` messages, as well as `CertificateVerify` and `Finished` messages were made. The processing of the `Certificate` message differs slightly (the message must now be sent by the client, even if the client is not in posession of a certificate – in this case the message contains no certificate(s)). FORTEZZA cipher suites are no longer supported and, as a major change, the MAC is replaced by an HMAC construction. The `ChangeCipherSpec` drop attack against SSL 3.0 is fixed by enforcing that a `ChangeCipherSpec` must prepend each `Finished` message – otherwise the handshake fails with a `FATAL` alert.

A detailed description of TLS 1.0 is given in Chapter 2.

[6] This directly counters attacks such as e.g. the Cipher Suite Rollback Attack (see Subsection 5.2.3)

[7] It is sufficient to sepcify a fix in RSA based key exchange only, since DH was not offered as an option for key agreement in SSL 2.0.

3.5. TLS 1.1

TLS 1.1 is another minor update. The most obvious change prevents attacks related to the CBC operation mode of block ciphers (e.g. Subsection 5.2.8). The structure of encrypted records (`TLSCiphertext`) used by TLS when block ciphers are used (`GenericBlockCipher`) now includes the IV, as can be seen in Listing 3.1.

```
1       block-ciphered struct {
2            opaque IV[CipherSpec.block_length];
3            opaque content[TLSCompressed.length];
4            opaque MAC[CipherSpec.hash_size];
5            uint8 padding[GenericBlockCipher.padding_length];
6            uint8 padding_length;
7        } GenericBlockCipher;
```

Listing 3.1: `GenericBlockCipher` as defined by TLS 1.1 – *Source: RFC 4346 [44]*

The IV is now explicit for each encryption. This implies that IVs no longer have to be computed what simplifies the key derivation process: Instead of 6 keys, now only 4 keys have to be derived from the `MasterSecret` (see Figure 3.10).

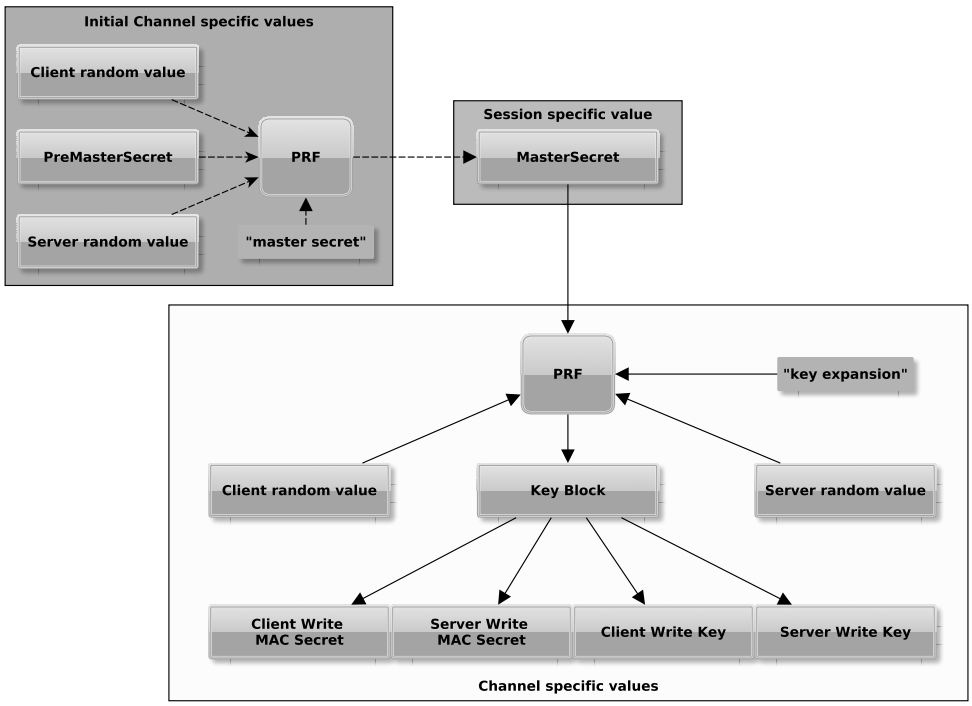

Figure 3.10.: Key derivation process of TLS 1.1 – *based on Source: RFC 4346 [44]*

To prevent attacks as presented by Bleichenbacher (see Subsection 5.2.7) the handling of padding errors changed to be undistinguishable from invalid MAC

errors. Beyond that, TLS 1.1 prohibits the use of deliberately weakened cipher suites to comply with former export restrictions. Finally, terminating connections without sending a `close_notify` alert no longer marks sessions as unresumable. According to the specification, this is "to conform with widespread implementation practice" – *Source: RFC 4346 [44].*

The mandatory cipher suite, required to be implemented by every TLS stack, changed from `TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA` for TLS 1.0 [43] to `TLS_RSA_WITH_3DES_EDE_CBC_SHA` for TLS 1.1 [44].

3.6. TLS 1.2

The latest revision of the protocol suite is TLS version 1.2 and brings a lot of enhancements. The new version bans MD5 and SHA-1 from the pseudo-random function and replaces them with a configurable hash function. Figure 3.11 depicts the modified pseudo-random function.

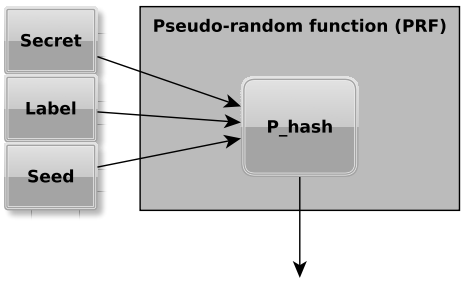

Figure 3.11.: Pseudo-random function of TLS 1.2 – *based on Source: RFC 5246 [45]*

The shift from two to only a single hash function utilized by the *PRF* simplifies the computation. The secret does not have to be split into two halves, each one processed by another hash function. This simplification also affects the payload (handshake verifying hash) of the `Finished` messages[8] and the hash values of all signed data fields.

The hash function to be used is part of the cipher suite definition.
As a major security enhancement, the support for authenticated encryption (Galois/Counter Mode (GCM), Counter with CBC-MAC (CCM) mode) was added which introcudes a new structre for Authenticated Encryption with Associated Data (AEAD) ciphers and thus a new cipher type (`GenericAEADCipher`, see Listing 3.2). This also bears a disadvantage, because some of the AEAD ciphers require IVs which are not explicitly contained in the according structures. Therefore, the creation of client and server IVs is reintroduced to the key derivation process (cf. Figure 2.26). The special treatment of AEAD IVs is discussed in RFC 5246 [45] in Subsections 6.2.3.3 and 3.2.1.

[8] Which also causes a variable length, depending on the cipher suite of the `verify_data` field.

```
1       struct {
2          opaque nonce_explicit[SecurityParameters.record_iv_length
               ];
3          aead-ciphered struct {
4              opaque content[TLSCompressed.length];
5          };
6       } GenericAEADCipher;
```

Listing 3.2: `GenericAEADCipher` as defined by TLS 1. – *Source: RFC 5246 [45]*

Each AEAD cipher require as input values a key, a nonce, the plaintext to be encrypted and some additional data. The additional data is clearly defined by the TLS 1.2 [45] specification and has to be build according to Listing 3.3 and includes the internal sequence number.

```
1 additional_data = seq_num + TLSCompressed.type +
2                           TLSCompressed.version + TLSCompressed.
                                length;
```

Listing 3.3: Additional AEAD data as defined by TLS 1. – *Source: RFC 5246 [45]*

New cipher suites using the new operation modes (GCM, CCM) and hash functions (SHA-256) were added while DES and IDEA cipher suites are now optional. TLS extensions [54] are now part of the main specification and are no longer separate documents and SSL 2.0 was clearly marked as deprecated (but not removed, yet) and the recommendation is to avoid backwards compatibility to SSL 2.0.

Finally, the mandatory cipher suite changed once again with this revision to `TLS_RSA_WITH_AES_128_CBC_SHA`.

"Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning."

Rich Cook

4 Popular Stacks

The following chapter presents popular implementations of SSL/TLS stacks. Due to the manifold number of stacks this list is far from being complete and highlights only the most common ones. Since nearly all of the mentioned implementations are continuously updated the details provided in this chapter may be outdated very soon. A recently updated comparison of SSL/TLS stacks is available at Wikipedia[1].

4.1. Implementations

4.1.1. OpenSSL

OpenSSL[2] is an open source implementation of SSL/TLS. The project is completely implemented in C programming language (C) and supports SSL 2.0/3.0, TLS 1.0/1.1/1.2 and Datagram Transport Layer Security (DTLS) 1.0. Implementations are available for Unix and Windows Operating system (OS)s. The stack is used by many well-known applications and very popular, especially in the Unix-world.

In addition to the SSL/TLS stack, OpenSSL also ships with own implementations of cryptographic algorithms and utility functionality for working with X509 certificates. The library is able to decode, create and sign certificates and Certificate Signing Request (CSR)s. Finally, routines for the conversion of different encoding formats and key generation are available. Because of this additional functionality, OpenSSL can be seen as a fully featured toolkit for PKI creation and management.

For basic testing and debugging purposes, OpenSSL ships with rudimentary client and server capabilities which is of special relevance to penetration testers and system administrators.

[1] `https://en.wikipedia.org/w/index.php?title=Comparison_of_TLS_implementations`
[2] `https://www.openssl.org`

Finally, OpenSSL is partly A Federal Information Processing Standard (FIPS) 140-2 certified (*OpenSSL FIPS Object Module*).

4.1.2. JSSE – Java Secure Socket Extensions

Java Secure Socket Extension (JSSE)[3] is a part of the Java Standard Edition Platform since version 1.4 and contains Java's default SSL/TLS implementation. The stack is completely written in Java and supports SSL 3.0 and TLS 1.0/1.1/1.2. As a part of the JavaSE platform, the library is available for all platforms with a JavaSE Runtime Environment (Windows, Unix, ...). Since it is shipped with every JavaSE/EE Runtime Environment (since version 1.4), Java based applications relying on SSL/TLS are very likely to use this stack – unless they integrate a 3rd party implementation.

The stack utilizes the Java Cryptography Architecture for cryptographic operations and can thus use different cryptographic implementations (Security `Provider` concept[4]).

The implementation can be configured to operate in an *FIPS mode* where only FIPS 140 certified implementations of the cryptographic routine are used. This comes with additional limitations, such as a reduced set of cipher suites or exclusively TLS mode support[5].

4.1.3. Legion of Bouncy Castle

Bouncy Castle[6] is much more than a conventional SSL/TLS stack, but rather a complete implementation of manifold cryptographic algorithms and protocols. The library ships with a stack only supporting TLS 1.0. APIs are available for either Java or C#. Since Bouncy Castle ships with a pluggable Security `Provider` for the Java Cryptography Architecture it can be used to replace Java's default implementation. The library supports extensive functionality for Abstract Syntax Notation One (ASN.1), X509, PKCS and Online Certificate Status Protocol (OCSP) processing and much more. Many applications written in Java, requiring additional cryptographic routines and algorithms, use the library either in addition to or as a replacement for the standard libraries shipped with Java. Especially, the routines for the creation of X509 certificates are frequently used.

The use of Bouncy Castle in sensitive environments (as for example required for military devices) can be difficult, since the library is not certified (e.g. by FIPS) yet.

[3] `https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html`
[4] `http://docs.oracle.com/javase/7/docs/technotes/guides/security/overview/jsoverview.html`
[5] `http://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/FIPS.html`
[6] `https://www.bouncycastle.org`

4.1.4. GnuTLS

GnuTLS[7] is another open source stack implemented in C. The stack supports SSL 3.0, TLS 1.0/1.1/1.2 and DTLS 1.0. Implementations are available for Unix and Windows OSs. The stack is used by many well-known applications and very popular in the Unix-world.

The command-line tools of GnuTLS ship, like OpenSSL, with rudimentary client and server tools, as well as utilities for managing X509 certificates.

GnuTLS does not ship with own implementations of cryptographic algorithms, but relies on libgcrypt by default. Optionally, the library can be manually replaced (for example by a FIPS certified one).

4.1.5. Microsoft SChannel

Secure Channel (SChannel)[8] by Microsoft is the default implementation of SSL/TLS on Windows OSs. SChannel supports the broad range of SSL 2.0/3.0, TLS 1.0/1.1/1.2 and DTLS 1.0/1.2 and is available exclusively for the Windows platform. The library is integrated as an Security Support Provider and can thus be used by any application working with the Security Support Provider Interface.

SChannel can be configured to use FIPS certified implementations of cryptographic algorithms, thus complying with the demand for such certifications when used in sensitive environments.

4.1.6. Network Security Services (NSS)

The NSS library[9] was created from the original Netscape Inc. code and is implemented in C and Assembler. NSS supports SSL 2.0/3.0 and TLS 1.0/1.1 and is today mostly, but not exclusively, used by browser and client software.

NSS also offers a FIPS certified crypto module and a native interface for Java applications.

4.1.7. Others

Many other SSL/TLS stacks are available, but rarely used. The following gives a (maybe non-exhaustive) list of implementations that have not been discussed in detail in the previous Subsections.

- cryptlib – `https://www.cs.auckland.ac.nz/~pgut001/cryptlib`
- CyaSSL – `https://www.wolfssl.com/yaSSL/Home.html`
- MatrixSSL – `https://www.matrixssl.org`
- PolarSSL – `https://polarssl.org`

[7] `https://www.gnutls.org`
[8] `http://msdn.microsoft.com/en-us/library/windows/desktop/ms678421(v=vs.85).aspx`
[9] `https://developer.mozilla.org/en-US/docs/NSS`

4.2. Usage Statistics

No accurate statistics on the spread of the implementations are available, but a rough estimation is possible by analyzing usage statistics of web browsers and servers, since the SSL/TLS stack each software is based on is known in most cases. Table 4.1 lists frequently used software and the used SSL/TLS stack (only those software is chosen that mostly relies on the same stack). But, this is just an estimation – detecting the real stack remains difficult. The Fingerprinting approach introduced in Chapter 7 (based on the T.I.M.E. framework, see Chapter 6) presents a technique on how to gain more precise statistics.

Software	**Type**	**Stack**
Apache	Webserver	OpenSSL
Microsoft IIS	Webserver	SChannel
GlassFish	Webserver	JSSE
Resin	Webserver	JSSE
nginx	Webserver	OpenSSL
Mozilla Firefox	Browser	NSS
Internet Explorer	Browser	SChannel

Table 4.1.: Frequently used software and the used SSL/TLS stack.

Figure 4.1 gives a statistic on the spread of the server software and Figure 4.2 the market share of the browser software listed in Table 4.1. It is not possible to clearly determine which SSL/TLS stack is used by the remaining, unlisted software, either because no details are available or the stack is platform or compilation dependent.

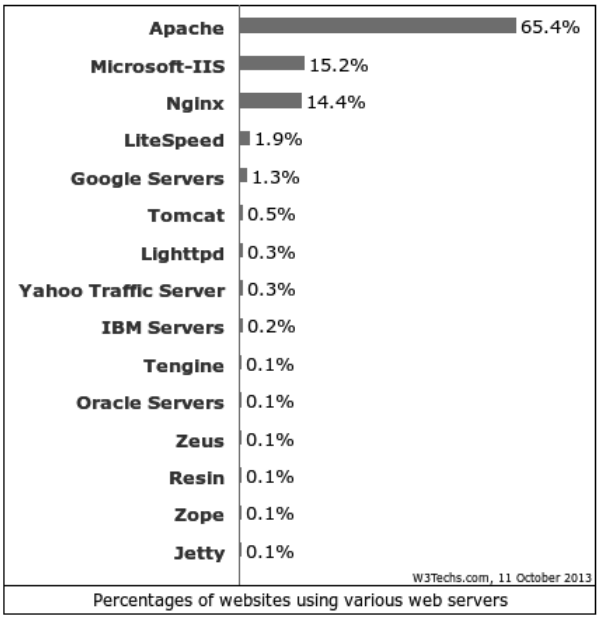

Figure 4.1.: Webserver market share – *Source: W3Techs.com*

Browser Statistics

2013	Internet Explorer	Firefox	Chrome	Safari	Opera
September	12.1 %	27.8 %	53.2 %	3.9 %	1.7 %
August	11.8 %	28.2 %	52.9 %	3.9 %	1.8 %

Figure 4.2.: Browser market share – *Source: W3Schools.com*

As can be seen, the Apache webserver is by far the most used server which relies on OpenSSL. In contrast, Google Chrome is by far the most used browser, but the stack used by Chrome depends on the OS.

Finally, it seems to be valid to assume that OpenSSL is the most used SSL/TLS stack on the web, followed by Microsoft SChannel.

Once again, this is just an estimation as only webservers and browsers were focused (many other software uses SSL/TLS as well!). Additionally, for some software it is not obvious which stack is used.

"If you want a bug fixed quickly, sell it on the Russian blackmarket. It'll be so heavily abused that the vendor will patch out of cycle."

Unconvenient truth fitting in a single Tweet

5 Attack Chronicle

The following Chapter is an extended version of the publication *SoK: Lessons Learned From SSL/TLS Attacks* together with Jörg Schwenk presented at the *International Workshop on Information Security Applications (WISA2013)*, Jeju Island, South Korea – August 2013 [58] (Best Paper Award).

Soon after the introduction of SSL Version 2.0 security concerns arose and some security weaknesses were fixed in Version 3.0. Many attacks of theoretical and practical nature have been found since and partly successful exploited. Ongoing research improves recent attacks and aims at proving security or finding further security related flaws. This chapter summarizes all known attacks on the protocol(s) so far in chronological order. All attacks outlined in this Chapter are grouped into four different attack categories:

- Attacks on the *Handshake Phase* and the *Handshake Protocol*
- Attacks on the *Record Protocol* or the interplay with the *Application Data Protocols*
- Attacks on the PKI
- Other attacks

Figure 5.1 gives a chronological overview of all attacks.

Figure 5.1.: Timeline of attacks on SSL/TLS

Attacks on the Handshake Phase The *Handshake Protocol* is the first phase of every SSL/TLS secured communication. During the *Handshake Phase* all cryptographic primitives, responsible for connection protection, are established. As this phase is undoubtly the most critical part when establishing a secured connection[1] and is at first mainly unprotected. This is why it offers the widest surface for attacks. This group of attacks covers all attacks targeting exchanged information during the *Handshake Phase*, such as tampering with security critical messages, recovering the *PreMasterSecret* and deliberate confusion of the handshake process.

Figure 5.2 depicts all attacks on the *Handshake Phase* (marked with a cross).

Attacks on the Record Layer Many flaws related to the *Record Layer* of SSL/TLS have been found over time. This includes problems with the (H)MAC protecting the Record's integrity (side channels, timing problems, ...), pitfalls with the operation modes of block-ciphers – in particular all CBC related attacks –, side channels introduced by compression and traffic analysis, as well as security issues when using stream-ciphers like RC4.

Figure 5.3 depicts all attacks on the *Record Layer* (marked with a cross).

Attacks on the PKI Attacks on the PKI summarize all kinds of attacks related to certificate validation, certificate issuance and CA compromises.

Figure 5.4 depicts all attacks on the PKI (marked with a cross).

Other Attacks This category includes all attacks that do not fit in one of the three main categories. Attacks of this category include issues with PRNGs, logical flaws related to session renegotiation, Denial of Service (DoS) and SSL/TLS disabling.

Figure 5.5 depicts all other attacks (marked with a cross).

5.1. Related Theoretical Work

As this thesis focuses on practice, this Section will only briefly summarize the theoretical work on the security of SSL/TLS. Since the author is not familiar with the theoretical results, this Section does not claim to be complete and is based on the most recent publications on this topic: *"On the Security of TLS-DHE in the Standard Model"* [59] and *"On the Security of the TLS Protocol: A Systematic Analysis"* [60]. For a complete and detailed overview please refer to these publications.

Many research (e.g. [61, 62, 63, 64]) was dedicated to the PKCS#1 structure of SSL/TLS for encapsulating RSA encrypted or RSA signed data (see RFC 2313 [65]). Moreover, the underlying cryptographic scheme – Mac-then-Encode-then-Encrypt (MEE) – has been analyzed (see e.g. Krawczyk [66]). Different lines of research focus on proving the security of SSL/TLS. Most results assume simplified (partly non-standard) versions of the protocol and thus

[1] The connection switches firstly from unprotected to protected mode at the end of the *Handshake Phase*.

Figure 5.2.: Timeline of attacks on the *Handshake Phase* of SSL/TLS

Figure 5.3.: Timeline of attacks on the *Record Layer* of SSL/TLS

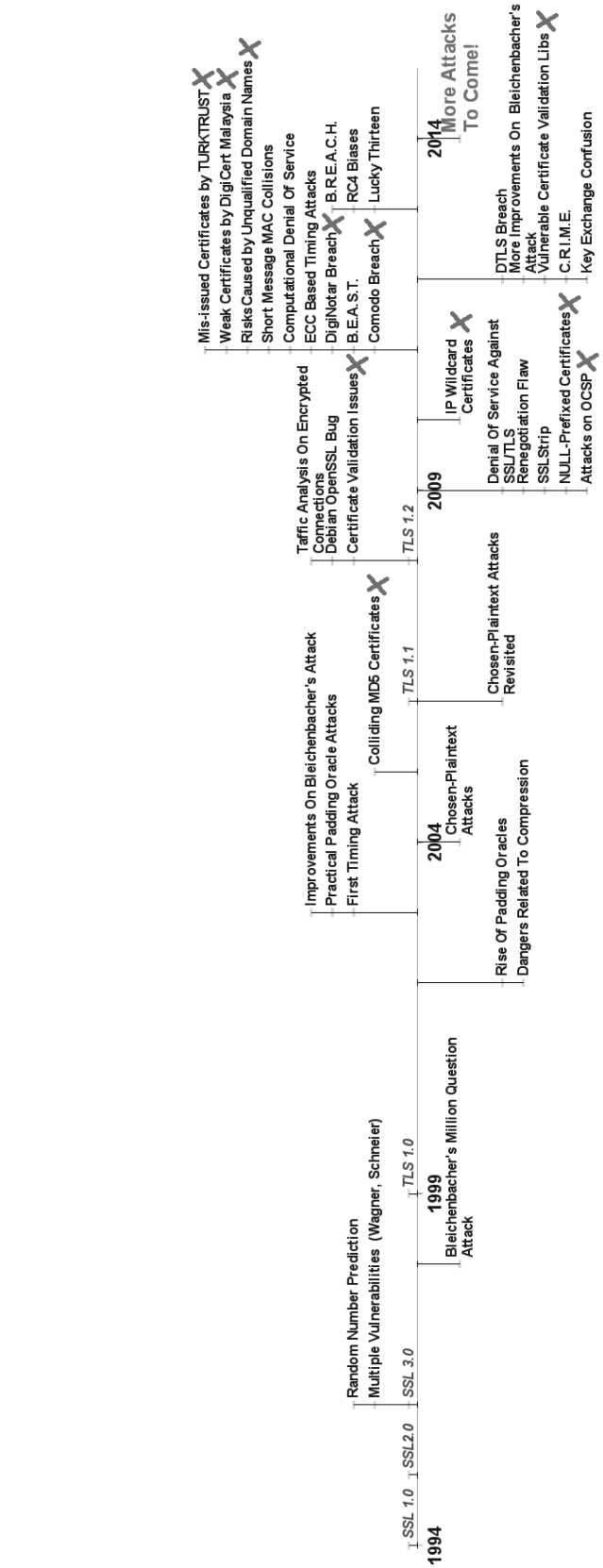

Figure 5.4.: Timeline of attacks on the PKI of SSL/TLS

Figure 5.5.: Timeline of all other attacks on SSL/TLS

proof the security not for the real SSL/TLS protocol (see [62, 67, 68, 69]). Additional proofs are presented in [70]. Research was also conducted in the area of applying automated proof techniques [71] or creating proofs directly from source code analysis [72]. Additionally, Bhargavan et al. presented results on the implementation of TLS in compliance with verified cryptography [73].

Source code analysis was also used to automatically detect vulnerabilities in SSL/TLS stacks [74]. Finally, the modes of operation of block ciphers have been focus of research [75].

Results covering both, theoretical and practical aspects, are excluded at this point and presented in the following Section 5.2.

5.2. Attacks with Practical Relevance

The following attacks are presented in chronological order.

5.2.1. Random Number Prediction

Category: ***Other Attacks***
In January 1996, Goldberg and Wagner published an article [76] analyzing the quality of random numbers used for SSL connections by the Netscape browser. The authors decompiled the browser and identified weaknesses in the algorithm responsible for random number generation.
Listing 5.1 shows the pseudo code provided by the authors which was examined (extracted from the source code of the Netscape Navigator browser).

```
1  global variable seed;
2
3  RNG_CreateContext()
4      (seconds, microseconds) = time of day; /* Time elapsed since
           1970 */
5      pid = process ID;   ppid = parent process ID;
6      a = mklcpr(microseconds);
7      b = mklcpr(pid + seconds + (ppid << 12));
8       seed = MD5(a, b);
9
10 mklcpr(x) /* not cryptographicly significant; shown for
       completeness */
11     return ((0xDEECE66D * x + 0x2BBB62DC) >> 1);
12
13 MD5() /* a very good standard mixing function, source omitted */
```

Listing 5.1: Algorithm for pseudo-random number generation in the Netscape Navigator – *Source: Randomness and the Netscape browser [76]*

As can be seen the entropy of this algorithm relies completely on a few, predictable, values:

- Current time
- Current process id
- Process id of the parent process

Moreover, the authors show that due to export limitations the entropy of key material is extremely limited – and argue that recomputation, even in 1996, is not that problematical. The paper discusses multiple attacks aiming at limiting the room of possible values used in the algorithm.

The problem of weak random number generators is not a general problem of SSL or TLS but reminds that a good PRNG implementation is a challenging task.

5.2.2. MAC does not Cover Padding Length

Category: ***Attacks on the Record Layer***
Wagner and Schneier pointed out in [77] that SSL 2.0 contained a major weakness concerning the MAC. The MAC applied by SSL 2.0 only covered data and padding, but left the padding length field unprotected. This may lead to message integrity compromise. A practical exploit for this weakness is not presented.

5.2.3. Cipher Suite Rollback

Category: ***Attacks on the Handshake Phase***
The cipher suite rollback attack, discussed by Wagner and Schneier in [77] aims at limiting the offered cipher suite list provided by the client to weaker ones or `NULL`-ciphers. Therefore, an attacker alters the `ClientHello` message sent by the initiator of the connection, strips off the undesirable cipher suites or completely replaces the cipher suite list with a weak one and passes the manipulated message to the desired receiver. The server has no choice: either reject the connection or accept the weaker cipher suite. An example scenario is illustrated in Figure 5.6.

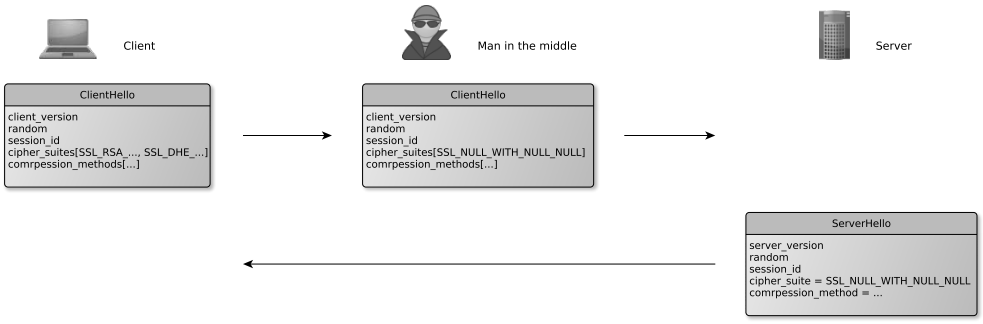

Figure 5.6.: Example scenario for the Cipher Suite Rollback Attack – *based on Source: Analysis of the SSL 3.0 protocol [77]*

This problem was fixed with the release of SSL 3.0, by authenticating **all messages of the *Handshake Protocol***.

5.2.4. ChangeCipherSpec Message Drop

Category: ***Attacks on the Handshake Phase***
This simple but effective attack described by Wagner and Schneier in [77] can be performed by an attacker acting as a MitM between a client and a server. During the *Handshake Phase* the cryptographic primitives and algorithms are determined. For activation of the new (pending) state of parameters with enabled security it is necessary for both parties to send a `ChangeCipherSpec` message. This messages inform the other party that the following communication will be secured by the previously agreed on parameters. The pending state is activated immediately after the `ChangeCipherSpec` message is sent.

An attacker located as MitM could simply drop the `ChangeCipherSpec` messages and cause both parties to never activate the pending states, both parties agreed on. The discussed example is illustrated in Figure 5.7.

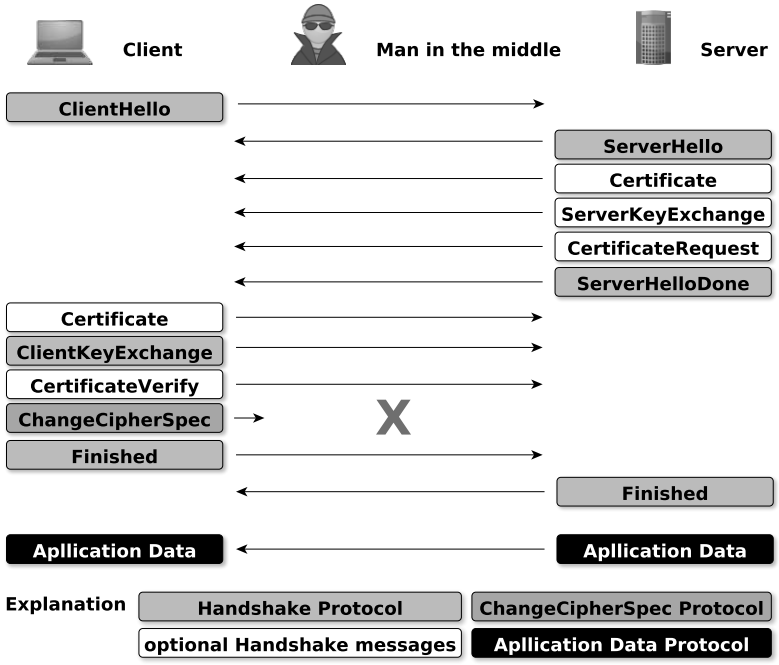

Figure 5.7.: Example scenario for the ChangeCipherSpec message drop attack – *based on Source: Analysis of the SSL 3.0 protocol [77]*

According to Wagner and Schneier the flaw was independently discovered by Dan Simon and addressed by Paul Kocher. The authors' recommendation is to force both parties to ensure that a `ChangeCipherSpec` message is received before accepting the `Finished` message. According to RFC 2246 [43], TLS 1.0 enforces exactly this recommendation:

> "It is essential that a change cipher spec message be received between the other handshake messages and the Finished message."
> – *Source: RFC 2246 [43]*

5.2.5. Key Exchange Algorithm Confusion

Category: ***Attacks on the Handshake Phase***

Another flaw pointed out by Wagner, Schneier in [77] is related to a feature concerning temporary key material. SSL 3.0 supports the use of temporary key material sent during the *Handshake Phase* (e.g. DH public parameters) signed with a long term key[2]. A problem arises from a missing definition of the type of contained data. Each party implicitly decides, based on the agreed on cipher suite, which key material is expected and decodes it accordingly. There is no information which type of key material is sent. This enables a type confusion attack.

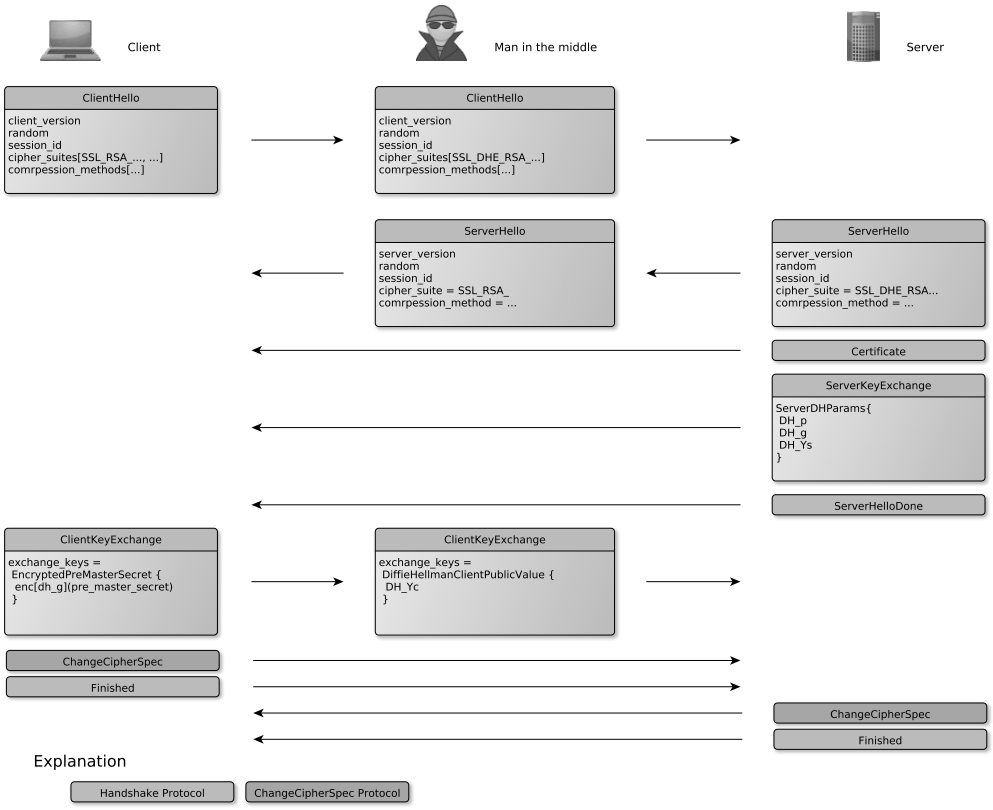

Figure 5.8.: Example scenario for the Key Exchange Algorithm Confusion Attack – *based on Source: Analysis of the SSL 3.0 protocol [77]*

It is crucial to point out that this attack remained fully theoretical at the time of writing. Multiple issues had to be solved (length checks, parameters resulting in suitable group elements, etc.). The attack sketched in Figure 5.8 aims at fooling a client into establishing a RSA-based key agreement while at the same time performing DHE (ephemeral DH key exchange) with the server. The attacker is located as a MitM between the client and the server and has to convince the client that `DH_p` concatenated with `DH_g` forms a valid `rsa_modulus`

[2]The temporary key material is used to establish a *PreMasterSecret*

and `rsa_exponent` (cf. `ServerRSAParams` as defined in RFC2246 [43]). Finding suitable parameters is a complicated and cumbersome task, since the parameters have to be valid parameters in DHE, as well as in RSA, unison. The attacker could decrypt the *PreMasterSecret* by taking the g-th root. After decrypting $PreMasterSecret^{\texttt{DH\_g}}$ mod `DH_p` the *PreMasterSecret* is known by the attacker and thus able to decrypt the whole communication.

The idea was revisited in the context of Elliptic Curve Cryptography (ECC) by Mavrogiannopoulos et al. in 2012 (see Subsection 5.2.35).

5.2.6. Version rollback

Category: *Attacks on the Handshake Phase*
Wagner and Schneier described in [77] an attack where a `ClientHello` message of SSL 3.0 is specially crafted to look like a `ClientHello` message specified of SSL 2.0. This would force a server to switch back to the more vulnerable SSL 2.0 instead of using SSL 3.0 version. In the 3rd version Paul Kocher applied, as a countermeasure, the presence of the SSL version in the PKCS#1 v1.5 encoded *PreMasterSecret*. The countermeasure is sufficient, since SSL 2.0 only supports RSA for key exchange (and not DH or DHE).

5.2.7. Bleichenbacher's Million Question Attack on PKCS#1

Category: *Attacks on the Handshake Phase*
In 1998 Daniel Bleichenbacher presented in [78] an attack on RSA-based SSL. Bleichenbacher utilized the strict structure of the PKCS#1 v1.5 format, used to deliver the encrypted *PreMasterSecret*, to build a decryption oracle. Together with a powerful algorithm, Bleichenbacher showed that it is possible to decrypt the *PreMasterSecret* in an acceptable amount of time. The *PreMasterSecret* is, in an RSA based handshake, generated by the client and sent (encrypted and PKCS#1 v1.5 formatted) within the `ClientKeyExchange` message during the *Handshake Phase*. The PKCS#1 v1.5 format is formally explained in Subsection A.1.2 of the Appendix. An attacker eavesdropping this (encrypted) message can decrypt it later on by abusing the server as a decryption oracle. The following sketch of Bleichenbacher's attack skips the mathematical background and further details (such as e.g. considerations for the adjustment of interval boundaries).

It is crucial to note that for Bleichenbacher's attack only the PKCS#1 v1.5 block type 00 02 is of relevance, since it identifies encrypted data (e.g. 00 01 would identify signed data) It is obvious that the structure is quite fixed and may not be invalidated. Invalidation will immediately cause an processing error, since payload data cannot be clearly separated from e.g. the padding data.

Bleichenbacher's attack is based on a known weakness of RSA to Chosen Ciphertext Attack (CCA)s (cf. *Chosen Signature Cryptanalysis of the RSA (MIT)Public Key Cryptosystem* [79]). The idea is to *blind* the original ciphertext, pass it to the decrypter and finally separate the blinding value.

1. Choose random integer s, must be invertible: $\exists s^{-1}$

2. Blind known ciphertext C
   $C' \equiv s^e C \mod n$

3. Let decryption oracle decrypt
   $P' \equiv C'^d \mod n$

4. Separate s value
   $P \equiv P' s^{-1} \mod n$
   $P \equiv s^{-1} s^{ed} C^{ed} \mod n$
   $P \equiv C \mod n$

The reason for this attack is known as the *multiplicative property* of RSA $Enc_{Key}(X) * Enc_{Key}(Y) \mod n = Enc_{Key}(XY \mod n)$.

As a precondition for the Bleichenbacher attack it is necessary that the target relies on PKCS#1 v1.5 structures. The attack makes use of the PKCS#1 v1.5 structure and differing processing behavior of SSL in case of compliant and non-compliant PKCS#1 v1.5 structured data. With this information it is easy to build an oracle $\mathcal{O}_{PKCS}$. The notation follows the one of Subsection A.1.2.

$$\mathcal{O}_{PKCS}(x) = \begin{cases} true & \text{if } x \text{ is PKCS\#1 v1.5 compliant} \\ false & \text{otherwise.} \end{cases}$$

If the oracle answers with `true` it can be concluded that $2b \leq x \leq 3b - 1$, b is defined as $b = 2^{8(\#n-2)}$.

At first, it is necessary to define the overall interval in which a 00 02 prefixed message is possible and define the notation in Table 5.1. **Step 1** introduced in the paper is skipped here, since as a prerequisite an attacker is already in possession of a PKCS#1 v1.5 compliant message.

Symbol	**Meaning**
p	Plain text to recover
c	Known PKCS#1 v1.5 compliant ciphertext
s_i	Blinding values
i	Counter
M_i	Set of intervals of possible solutions for c

Table 5.1.: Formal definitions for Bleichenbacher's attack

$$b = 2^{8(\#n-2)}, \text{ since two bytes are predefined (00 02)} \tag{5.1}$$

A counter i is initially set as follows, $i = 1$ and $M_0 = \{[2b, 3b - 1]\}$. The boundaries of M_0 are explained as follows.

$$\begin{aligned} 2b: &\ 0000\ 0000\ 0000\ 0010\ 0000\ 0000\ ...\ 0000 = 0x00\ 0x02\ 0x00\ ...\ 0x00 \\ 3b: &\ 0000\ 0000\ 0000\ 0011\ 0000\ 0000\ ...\ 0000 = 0x00\ 0x03\ 0x00\ ...\ 0x00 \end{aligned} \tag{5.2}$$

This defines the values of the first two bytes, which are of interest in this case. Since 00 03 is not in the valid range it has to be decreased.

$$3b-1:\ 0000\ 0000\ 0000\ 0010\ 1111\ 1111\ ...\ 1111 = 0x00\ 0x02\ 0xff\ ...\ 0xff \tag{5.3}$$

This finally leads to the only possible range for messages starting with 00 02.

The next step searches for suitable (PKCS#1 v1.5 compliant) messages. Finding a PKCS#1 v1.5 compliant message by blinding, is one of the hardest steps, since it may require manifold tries until a value is found. The simplest way to do this is increasing an integer s_1 by one, starting at the lowest possible boundary, and query the oracle $\mathcal{O}_{PKCS}$ for each candidate.

Step 2a: if $i = 1$

1. Set s_1 `=` $\frac{n}{3b}$
2. `while(`$\mathcal{O}(c(s_i)^e \mod n) \neq$ `true)` {
   a) s_1`++`
   }

If there is a PKCS#1 v1.5 compliant blinded message found, but the solution could not be computed, the algorithm simply continues the search for blinding values yielding to PKCS#1 v1.5 compliant messages.

Step 2b: if $i > 1$ and M_{i-1} contains at least 2 intervals

1. Set s_i `=` s_{i-1} `+ 1`
2. `while(`$\mathcal{O}(c(s_i)^e \mod n) \neq$ `true)` {
   a) s_i`++`
   }

In case there is only one residual interval in the current M_i left, Bleichenbacher optimizes the process of finding PKCS#1 v1.5 compliant messages by applying binary search and adjusting the boundaries of M_i (intervals in which possible values for s_i can be found).

Step 2c: if $i > 1$ and M_{i-1} contains only a single interval

1. Set $r_i = 2\frac{ls_{i-1}-2b}{n} - 1$
2. `do` {
   a) r_i`++`
   b) `for(`s_i`,` $\frac{2b+r_in}{l} \leq s_i < \frac{3b+r_in}{k}, found \neq$ `true,` s_i`++)` {
      i. $found = \mathcal{O}(c(s_i)^e \mod n)$
      }
   } `while(`$found \neq$ `true)`

After a blinding value is found that yields to a PKCS#1 v1.5 compliant, blinded ciphertext the interval of possible solutions can be futher adjusted.

Step 3

1. `for(Interval` $[k, l]$ `in` M_{i-1}`)` {
   a) `for(`r`,` $\frac{ls_i - 3b + 1}{n} \leq r \leq \frac{ks_i - 2b}{n}$`,` r`++)` {
   b) add to M_i : $[\max(k, \lceil \frac{2b + rn}{s_i} \rceil), \min(l, \lfloor \frac{3b - 1 + rn}{s_i} \rfloor)]$ }
   }

Finally, the algorithm tries to compute a solution if only a single interval with identical boundaries ($[k, k]$) is left . If no solutions could be found, the algorithm restarts by searching a new blinding value s_i.

Step 4

1. `if(size of` M_{i-1} `= 1 and` $k = l$ `of` $[k, l]$`)` {
   a) $P = k(s_0)^{-1} \mod n$
   } `else` {
   a) i`++`
   b) `go to step 2`
   }

If M_i contains only a single interval with length 1, $P \equiv C^d \mod n$ which is the plaintext of C (its RSA decryption).

5.2.8. Weaknesses Through CBC Usage

Category: ***Attacks on the Record Layer***

Serge Vaudenay introduced a new kind of attack class – padding oracle attacks – and forced the security community to rethink on padding usage in encryption schemes (cf. *Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC,WTLS...* [80]).

The attacks described by Vaudenay rely on the fact that block encryption schemes operate on blocks of fixed length, but in practice most plaintexts do not accurately fit the requested length (the plaintext length is not a multiple of the block length). Therefore, it is necessary to pad the plaintext to a multiple of the block length of the block-cipher. After padding the input data is passed to the encryption function, each plaintext block (of length of the block size) is processed and chained according to the CBC scheme. The CBC mode is explained and formalized in the Appendix, see Subsection A.1.3.

Vaudenay build a decryption oracle based on the receiver's reaction to a ciphertext in case of valid/invalid padding. Since for SSL/TLS the receiver sends an `Alert` of type `decryption_failure`, in case of invalid padding, these cases can be strictly distinguished. A basic padding oracle in formal definition is represented by $\mathcal{O}_{Padding}(x)$.

$$\mathcal{O}_{Padding}(x) = \begin{cases} true & \text{if if } x \text{ is correctly padded} \\ false & \text{otherwise.} \end{cases}$$

This oracle can be improved to decrypt the last word of a ciphertext, a whole block or even a complete message (if the IV is known). For the following it is necessary to introduce additional definitions.

$$\begin{aligned} b &: \text{Used block length in full words} \\ X &: [x_1, x_2, ..., x_l], \text{ Message X consists of l blocks} \\ x_i &: \text{full block consisting of b words } x_i = [x_i^1, ..., x_i^b] \\ x_i^j &: \text{word } j \text{ of a block } i \end{aligned} \tag{5.4}$$

A message X consisting of l blocks has a correct padding if the last block x_l ends with q words with the value set to 'q' (e.g. 1, 22, 333, 4444...) – this padding scheme is specified in PKCS#7 (see RFC 5652 [81]).

Last Word Oracle. With these definitions it is possible to build an oracle that decrypts the last word of a block Y i.e., $Dec_{Key}(y) = x = x^1, ..., x^n$. The value $r = r^1, ..., r^b$ denotes a random block consisting of b random words. The oracle should output $true$ if $r|y$ is properly padded – the | operator denotes concatenation – i.e., $\mathcal{O}(r|y) = true$, if $Dec_{Key}(y) \oplus r$ decrypts to a valid padded block. The following algorithm will output in worst case only the last decrypted word, in best case a few words with a single run.

1. Get $r = r^1, ..., r^b$ random words
2. `i = -1`
3. `while(`$\mathcal{O}(r|y) \neq$ `true) {`
   a) `i++`
   b) Flip the last word by i.
      $r = [r^1, ..., r^{b-1}, (r^b \oplus i)]$

   }
4. `i--`
5. Since the oracle indicated a valid padding, the last word (r^b of r) is valid when XOR-ed with i. This implies the algorithm succeeded in guessing the last word of the preceding encrypted block – the last word is now known! $r^b = r^b \oplus i$, $r = r^1, ..., r^b$
6. Continue with checking if more words ($r^1, ..., r^{b-1}$) can be guessed.
   `n = b+1`

7. `while(`$\mathcal{O}(r|y)$ `== true && n >` 2`){`
   a) `n--`
   b) Flip the residual words, one by one. For simplicity only test with 1.
      $r = [r^1, ..., r^{b-n}, r^{b-n+1} \oplus 1, r^{b-n+2}, ..., r^b]$
   }
8. Output the last decrypted word(s) of a block.
   `return` $[r^{b-n+1} \oplus n, ..., r^b \oplus n]$

Block Decryption Oracle. In contrast to the last word oracle, this oracle decrypts a complete block ($Dec_{Key}(y) = x$), without knowledge of the key. As a precondition some plaintext words of y have to be known. E.g., $y^j, ..., y^b$ of $y = [y^1, ..., y^j, ..., y^b]$ are already known. y^b (and even some more words if lucky) can be gathered by calling the last word oracle.
For a better understanding it is necessary to formalize the padding structure. $b - j + 2$ equals the padding length – e.g. $j - b = 2$ are the lowest index number of the foreknown words and the block length is $b = 8$ (the block looks like this yet: `x|x|x|x|x|x|3|3`, this means words y^7 and y^8 are known) $\Rightarrow b - j + 2 = 8 - 7 + 2 = 3$ value of the remaining padding word.
The following algorithm decrypts y^{j-1}. The whole block can be decrypted iteratively calling this oracle until the block decryption is finished.

1. Set $r = [r^j, ..., r^b]$
   $r^k = y^k \oplus (b - j + 2)$
   $k = j, ..., b$
2. Add $[r^1, ..., r^{j-1}]$ random words to r.
   $r = [r^1, ..., r^j, ..., r^b]$
3. `i = -1`
4. `while(`$\mathcal{O}(r|y) \neq$ `true) {`
   a) `i++`
   b) Flip the current word by i.
      $r = [r^1, ..., r^{j-2}, r^{j-1} \oplus i, r^j, ..., r^b]$
   }
5. `i--`
6. Output the decrypted word.
   `return` $[r^{j-1} \oplus i \oplus (b - j + 2)$

Decryption Oracle. The decryption oracle is the enhanced version of the block decryption oracle. A whole message consisting of l blocks and subsequently $l*b$ words, can be decrypted by iteratively invoking the block decryption oracle for each single block of the message, starting with the last one.

A problem remains if the IV is unknown – in this case it is not possible to recover the first block of a message.

Bomb Oracle. Vaudenay introduces a new type of oracle: the bomb oracle. This type of oracle gets unavailable after it is invoked with invalid input – it explodes when called with wrong input data. The bomb oracle can be used to verify a guess – either the guess is right (`true`) or the oracle explodes. Given a ciphertext $Y = [y_1, ..., y_l]$ and a word sequence $w^1, ..., w^m$, where $m \leq b$. The following algorithm checks, by the use of the bomb oracle, if $w^1, ..., w^m$ is a valid end sequence of y.

1. Get $r = [r^1, ..., r^{b-m}]$ random words.
2. Add $r^{(b-m+k)}w^k \oplus m$ to r (now the length of r equals b - $r = [r^1, ..., r^b]$). $k = 1, ..., m$.
3. $\mathcal{O}(r|y_l)$, if the oracle does not explode continue
4. Check if $r^b = w^m \oplus m \oplus 1$ is a valid end sequence of y.
   `if(`m `== 1) {`
   a) $r^{b-1} = r^{b-1} \oplus 1$, no w words more left, modify the preceding r word.
   `} else {`
   b) $r^b = w^m \oplus 1$
   `}`
   $r = [r^1, ..., r^b]$
5. Perform a second oracle call, to verify the guess.
   $\mathcal{O}(r|y_l)$, if the oracle does not explode continue
6. Since the oracle is still alive $w^1, ..., w^m$ must be a valid end sequence of y.
   `return true`

SSL/TLS provides such a bomb oracle, since an incorrect padding leads to a `fatal` *Alert* causing the connection to be closed immediately. In this case, an attacker is forced to randomly guess a word sequence and check the guess by using the decryption oracle. The optional MAC ensuring message integrity of SSL/TLS does not hinder this attack, since MAC creation takes place before the message is padded (cf. Subsection 2.2.2 for details on MAC creation and encryption order) and thus the padding is not covered by the `MAC`.

Useful Countermeasures. Vaudenay describes countermeasures that, "May work" – *Source: Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS... [80].* He suggests the following workflow to protect messages encrypted in CBC mode:

1. Pad the plaintext m, $x = m|p$
2. Authenticate and encrypt x, leading to ciphertext y
3. Send y

The decryption process is vice-versa:

1. Decrypt and check authenticity of y, leading to plaintext x
2. Check padding validity of x
3. Separate padding p from $x = m|p$ and return plaintext m

Useless Countermeasures. Additional to the useful countermeasures, the author identified ineffective fix suggestions:

- Prefix message with padding: $Enc_{Key}(padding|message)$
- Usage of double encryption leading to double CBC
- Usage of on-line ciphers or HCBC [82] mode ($y_i = Enc_{Key}(H(y_{i-1}) \oplus x_i)$
- Modified CBC mode ($\oplus$ before and after encryption, with additional operations depending on i, all $x_j, y_j, j = 1, ..., i-1$)
- Add padding and integrity preserving data (e.g. HMAC) to the message ($Enc_{Key}(message|padding|h(message|padding))$

In an email communication[3], Bodo Möller picked up Vaudenay's attack and pointed out that in SSL 3.0 the padding format is not specified in detail, making a valid padding check impossible. Only the last byte has to indicate length of the padding excluding this length byte. According to Möller, even thus the padding structure is specified in detail in TLS 1.0 some implementations do not check for valid paddings, as long as the padding length is accurately set. If padding is not checked it may offer some details to an attacker if she modifies ciphertexts and the stack does not detect this. If modified ciphertexts do not cause errors an attacker could be sure that the modified data is inside the padding range. Even worse, the added padding is not covered by the MAC due to the Mac-then-Pad-then-Encrypt structure of the protocol (this behavior is explained in detail in Subsection 2.2.2. This enables an attacker to try out different padding modifications and gain knowledge on the stacks behavior (possibly distinctive error messages).

5.2.9. Information Leakage by Use of Compression

Category: ***Attacks on the Record Layer***
In [83], Kelsey described an information leak enabled by a side channel based on compression. This is in absolute contrast to, what he calls, "folk wisdom" that applying compression leads to a security enhanced system. Kelsey shows that compression adds little security – in worst case is even the other way around –, due to the fact that compression reveals information on the plaintext.
Crypto systems aim at encrypting plaintexts in a way that the resulting ciphertext reveals ideally no single bit on information about the plaintext. Kelsey observed that by the use of compression a new side channel arises which could

[3] `http://www.openssl.org/~bodo/tls-cbc.txt`

be used to gain hints on the plaintext.He correlates the output bytes to the input bytes and makes use of the fact that lossless compression algorithms, when applied to the plaintext, reduce the size of the input (leading to a different amount of output bytes).

Compression algorithms encode redundant patterns in input data to shorter representations and substitutes each occurrence with the new representation. Different input strings of the same length compress to strings of different length. Kelsey used this observation to gain knowledge about the plaintext. As an example the author highlights the case of a surveillance camera, whose output data is of static size in case the camera captures the same static scene. If in contrast the camera captures motion the size will significantly differ. Moreover, it is possible to infer something about the type of the processed data, since e.g. binary files compress at a different ratio than text files. Another example deals with different messages of the same length which would compress at different ratios and thus be distinguishable. The same holds for the detection off looped input. All these information leaks can be exploited by simply observing the length of the ciphertext (padding issues are disregarded at this point), reduced ciphertext length implies higher redundancy in the input data.

While the attacks mentioned above are of passive nature an attacker can gain even more knowledge if being active. The following discusses attacks on how to detect and extract whole strings form ciphertexts by the use of specially chosen plaintexts. For these attacks the following formalization is used: Let P_i be a plaintext with C_i bein the corresponding ciphertext, i is a positive integer, n a prefix and s_i a string.

Detect if a Guessed String s Appears Often in a Set of l (Encrypted) Plaintexts $P_0, ..., P_l$.

1. Query an oracle for C_i, the compressed and encrypted plaintext P_i, for each $i = [0, l]$: $C_i = encAndCompress(P_i)$. This reveals the compressed output lengths of P_i (again, padding issues are disregarded).

2. Query an oracle for the encrypted and compressed representation C_i' of $P_i|s$, the plaintext with appended guess s (the plaintext remains secret!): $C_i' = encAndCompress(P_i|s)$.

3. Compress s and observe the resulting length.

4. Compute $s' = C_i' - C_i$. If the length of s' clearly differs from the expected length of s, observed in the previous step, for a significant amount of i values, there is a high probability that s is contained in many of the P_is.

Guessing Parts of P.

1. Generate a list of all possible values s_i appearing in P and sort by occurrence likelihood.

2. Combine different s_i values to a *guess* and form a query $query = guess|C$. Observe the result and conclude according to the output length. Shorter output lengths suggest a higher probability of appearance of *guess* in P.

Kelsey advices that also timing issues have to be taken into consideration, since compression as an additional step requires additional processing time.

The results of Kelsey represent the foundation of the C.R.I.M.E. attack by Rizzo and Duong (see Subsection 5.2.34).

5.2.10. Timing Attacks

Category: ***Attacks on the Handshake Phase***

Brumley and Boneh outlined a timing attack on SSL/TLS in [84], applicable when RSA cipher suites are used. The attack extracts the private key from a target server by observing the timing differences between sending a specially crafted `ClientKeyExchange` message and receiving an `Alert` message inducing an invalid formatted *PreMasterSecret*. This difference in processing time allows to draw conclusions on the used RSA parameters. As already mentioned, the attack discussed in the following is only applicable in case RSA-based cipher suites are used. Additionally, the attack requires the presence of a high-resolution clock on attacker's side. The authors were able to successfully attack OpenSSL - The Open Source toolkit for SSL/TLS (OpenSSL) due to deactivated timing attack countermeasures.

OpenSSL's implementation is based on the Chinese Remainder Theorem (CRT) in order to enhance computation, which generally is not vulnerable to Paul Kocher's timing attack (cf. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems* [85]). But additionally to the CRT optimization OpenSSL uses optimizations such as sliding-window exponentiation which in turn heavily relies on Montgomery's reduction algorithm for efficient modulo reduction. Montgomery's algorithm requires multi-precision multiplication routines to function. OpenSSL implements two different algorithms to perform such multiplications: Karatsuba algorithm and an unoptimized standard (normal) algorithm. For these algorithms the integer factors are represented as sequences of words of a predefined size. Due to efficiency reasons OpenSSL uses Karatsuba if factors with an equal number of words are multiplied and the "normal" algorithm otherwise. According to the authors the "normal" algorithm is generally much slower than Karatsuba, resulting in a measurable timing difference. But due to a peculiarity of Montogmery's algorithm, an additional extra reduction in some cases, the optimizations of Montgomery and Karatsuba counteract one another. This prevents a direct timing attack based on these observations.

Moreover, the attack of Brumley and Boneh aims at determining the dominant effect at a specific phase and leveraging the differences. The authors exemplify that their algorithm is a kind of binary search for the lower prime of the RSA modulus n. For simplicity the algorithm description is based on the following assumption: $q < p, n = pq$. The algorithm starts by "guessing" a possible value q. The initial guess of q is denoted as g with $2^{log_2(\frac{n}{2})-1} < g < 2^{log_2(\frac{n}{2})}$. Now the top most bits (the authors state 2-3 are sufficient) are flipped to every possible combination and the decrpytion times are measured. According to Brumley and Boneh this will create at least two peaks, where the first one represents q. From this observation the first few bytes of q can be recovered.

After some of the most significant bits are recovered ($i-1$ bits of q), g's bits are set according to the already recovered bits followed by an all-zero sequence. This causes (hopefully) a situation where $g < q$.

1. Set g_{hi} as g with bit i set to 1.
   The idea is if bit i of q is 1, then $g < g_{hi} < q$, or $g < q < g_{hi}$ otherwise.
2. Compute $u_g = gr^{-1} \mod n$, $u_{g_{hi}} = g_{hi}r^{-1} \mod n$.
   This counters the conversion of u_g and $u_{g_{hi}}$ to Montgomery form before exponentiation ($u_g r = g = gr^{-1}r$, $u_{g_{hi}} r = g_{hi} = g_{hi}r^{-1}r$).
3. Measure time of decryption
   $t_g = time(Decrypt(u_g))$, $t_{g_{hi}} = time(Decrypt(u_{g_{hi}}))$
4. Set $\Delta = |t_g - t_{g_{hi}}|$
5. `if(Δ >>) {`
   a) `bit` i `of` q `= 0 //` $g < q < g_{hi}$
   `} else {`
   a) `bit` i `of` q `= 1 //` $g < g_{hi} < q$
   `}`

Utilizing this algorithm results in recovering every bit of q, step by step. Finally, $p = \frac{n}{q}$.

As a countermeasure, the authors suggest as the most promising solution the use of RSA blinding. Blinding uses a a random value r and computes $p = r^e c \mod n$ before decrpytion (p : plaintext, c : ciphertext, n : RSA modulus, e : public exponent). The plaintext can be recovered by decrypting and dividing by r: $p = \frac{(r^e c)^d \mod n}{r} = \frac{(r^e p^e)^d \mod n}{r} = \frac{rp \mod n}{r}$.

The attack was significantly improved in 2005 by Aciicmez, Schindler and Koc in [86].

5.2.11. Intercepting SSL/TLS Protected Traffic

Category: ***Attacks on the Record Layer***

In [87] Canvel, Hiltgen, Vaudenay and Vuagnoux extended the attack(s) presented by Vaudenay (cf. Subsection 5.2.8) to finally decrypt a password of an SSL/TLS secured Internet Message Access Protocol (IMAP) session. The attack is based on padding oracles as introduced by Vaudenay in Subsection 5.2.8. It is necessary to recap that in SSL/TLS the order of protection operations is as described in 2.2.2 – Mac-then-Pad-then-Encrypt.

Canvel et al. suggested 3 additional attack types based on Vaudenay's observations:

Timing Attacks. The authors concluded that a successful MAC verification needs significantly more processing time compared to an early abortion caused by an invalid padding. This observation relies on the fact that performing a

padding check is less complex. To optimize the timing results it is suggested to increase the message size to its maximum, since processing time of the MAC verification depends on the message size. According to the authors, the MAC verification time increases linearly with the message length.

Multi-session Attacks. This attack type requires a critical plaintext to be present in each TLS session, such as e.g. a password, and that the corresponding ciphertext is known to the attacker. Due to security best-practice, the corresponding ciphertexts look different every session, since the key material for MAC and encryption changes with every new session. Therefore, it is advantageous to check if the plaintext related to a given ciphertext ends with a specific byte sequence which has to be identical in all sessions. This is done by building fake ciphertexts which are sent to an oracle.

Dictionary attacks. Leveraged by the previous attack type which checks for a specific byte sequence of the plaintext, this attack aims at checking for specific byte sequences of a dictionary.

A successful attack against an IMAP server was performed and the password used by the `LOGIN` command could be recovered:

```
XXX LOGIN "username" "password"<0x0d<0x0a>
```

Although the attack succeeded the authors had to face some problems when dealing with real life systems. A major problem arose from the fragmentation of the command when passed to the block-cipher. In case of an 8-byte block-cipher it may happen that the last bytes of the password are in a block together with the HMAC. The example of the paper describing this problem is listed in Listing 5.2.

```
1  |0021 LOG|IN "name|" "passw|ord"<0x0d<0x0a><HMAC1><HMAC2>|
```

Listing 5.2: Chunked example where the last 3 bytes of a password cannot be decrypted – *Source: Password Interception in a SSL/TLS Channel [87]*

In such a fragmentation, it is obvious that the last 3 characters of the password (`ord` in this example) cannot be recovered, since the HMAC changes for every new set of keys, thus the byte sequences of two identical plaintexts encrypted with different key material are not equal (the string is not fixed – in this example the 2 HMAC bytes are variable)!

As a recommendation the author's propose to change the processing order when encrypting. The MAC should also cover the padding, which implies the order PAD-then-MAC-then-Encrypt.

5.2.12. Improvements on Bleichenbacher's Attack

Category: ***Attacks on the Handshake Phase***
The researchers Klíma, Pokorny and Rosa not only improved Bleichenbacher's attack (cf. Subsection 5.2.7) in [88], but were able to defeat a common countermeasure to prevent the attack.

Breaking the Countermeasure. A commonly used countermeasure is to generate a random *PreMasterSecret* in any kind of failure (see RFC 2246 [43] Subsection 7.4.7.1) and continue with the handshake until the verification and decryption of the `Finished` message fails, due to different key material (the *PreMasterSecret* differs at client and server side). Additionally, the implementations should not return distinguishable `Alert` messages, which in turn means that only one kind of error message is used for all errors. This countermeasure is regarded as best-practice and remains valid, even after this attack. But due to a peculiarity of SSL/TLS the encrypted data of the `ClientKeyExchange` message in an RSA-based handshake includes not only the *PreMasterSecret*, but also the major and minor version number of the negotiated SSL/TLS protocol version (see Figure 2.20).

According to the authors many implementations at this time checked for equality of the received protocol version contained in the `ClientKeyExchange` message and the negotiated protocol version in the `ServerHello` message. Surely, this requires a valid PKCS#1 v1.5 structure, with at least correct leading bytes (00 02). In case of protocol version mismatch a distinguishable `Alert` message wass returned to the sender (e.g. type `decode_error` in case of OpenSSL). It is obvious that an attacker can build a new (bad version) oracle $\mathcal{O}_{BadVersion}(c^*)$ from this side channel.

$$\mathcal{O}_{BadVersion}(c^*) = \begin{cases} true & \text{if ciphertext } c^* \text{ contains a valid version number} \\ false & \text{otherwise.} \end{cases}$$

With succeeding in the construction of a new decryption oracle $\mathcal{O}_{BadVersion}$ Klíma, Pokorny and Rosa were able to mount Bleichenbacher's attack, thus recommended countermeasures are present.

Improving Bleichenbacher's Attack on PKCS#1. Additionally to the resurrection of Bleichenbacher's attack the authors could improve the algorithm for better performance.
At first Klíma, Pokorny and Rosa redefined the boundaries of possible PKCS#1 v1.5 compliant plaintext intervals M_i as listed in equation 5.5.

$$\begin{aligned} lowerBoundary &= 2b + 1 * (256^{k-3} + 256^{k-4} + ... + 256^{49}) \\ &= 2b + \frac{256^{49}(256^{k-51} - 1)}{255} \\ upperBoundary &= 2b + 255 * ((256^{k-3} + 256^{k-4} + ... + 256^{49}) + 0 + 255 \\ &\quad * (256^{47} + ... + 256^{0}) \\ &= 3b + 255 * 256^{48} - 1 \\ M_i &= [lowerBoundary, upperBoundary] \end{aligned} \tag{5.5}$$

The boundaries are chosen according to previous knowledge gained from the SSL/TLS specification:

- A *PreMasterSecret* is exactly 46 bytes long
- The *PreMasterSecret* is prefixed with two version bytes
- Padding bytes are unequal to 00
- A PKCS#1 v1.5 compliant plaintext M_i contains a null-byte separating the padding from the payload data. The length of the padding is known in advance (2 type bytes, $k-51$ padding bytes, single null byte as separator, 2 bytes version number and 46 bytes *PreMasterSecret*)

The boundaries can be adjusted even tighter, since the used protocol version is known in advance – revealing another 2 bytes.

Another optimization is made to the original algorithm by introducing a new method for finding suitable s_i values (cf. Subsection 5.2.7). The authors call this the *Beta method*.

$$\begin{aligned} p_k &= (c_k)^d \mod n \\ p_l &= (c_l)^d \mod n \end{aligned}$$

$$\begin{aligned} c_k &= (s_k)^e c \mod n \\ c_l &= (s_l)^e c \mod n \end{aligned} \tag{5.6}$$

$$\begin{aligned} s' &= (1-\beta)s_k + \beta s_l \mod n, \beta \in \mathbb{Z} \\ p' &= (c')^d \mod n \\ c' &= (s')^e c \mod n \end{aligned}$$

By choosing new values for s_i according to s' may speed up the attack, since $gcd(s_l - s_k, n) = 1$ implies that a valid triplet (s_k, s_l, s') exists for a particular β. The process of finding new values for s_i can so be reduced to choosing suitable β values and building a linear combination (which in turn requires at least two valid values s_k, s_l). However, due to the authors even this method contains pitfalls leading to inconvenient values for β or s'. But, for inconvenient values of s' the authors recommend to use $s'' = s' - n$. This value is referred to as the corresponding symmetric value of s', the argumentation is simple: $s''p \mod n = (n - s')p \mod n = -s'p \mod n = n - s'p \mod n$. When using the symmetric value the boundaries have to be adjusted, too: $lowerBoundary = n - upperBoundary$ and $upperBoundary = n - lowerBoundary$

As a last optimization Klíma, Pokorny and Rosa suggested to parallelize the search for suitable s_i values (step 2 of Bleichenbacher's original algorithm). Step 2c performs much better than steps 2a and 2b, since s_i can be chosen more efficiently if only one interval is left in M_i. Therefore, for every interval in M_i a separate thread should be started performing step 2c. When one of these threads succeeded in finding a suitable s_i this s_i is associated with M_i and the remaining threads can be stopped and discarded. This parallelization should significantly increase performance.

5.2.13. Chosen-Plaintext Attacks on SSL

Category: ***Attacks on the Record Layer***

Gregory Bard observed in [89] an interesting detail concerning IVs of block-cipher encrypted SSL messages. At first, it is necessary to make clear that every en- and decryption in CBC mode starts with an IV, as can be seen in Figure A.2. Ideally, every independent plaintext (consisting of multiple blocks) should get its own, randomly chosen, IV.

The problem with SSL is that, according to the SSL 3.0 and TLS 1.0 specification (see RFCs [43, 44]), only the IV of the first plaintext is chosen randomly. All subsequent IVs are simply the last block of the previous encrypted plaintext. This is absolutely in contrast to cryptographical best-practice – each independent plaintext should have its own randomly chosen IV.

Bard observed that an attacker can pick a specific ciphertext block ($C_{particular}$) and make a guess about its corresponding plaintext ($P_{particular}$). While this is not special, the author also observed that the attacker can additionally verify whether this guess is correct. The following assumes that the attacker tries to find out whether the (unknown) plaintext block $P_{particular}$ is of value P^{guess}. As explained previously, the IV is known to anyone who is able to eavesdrop on the connection (C_{IV}). As a prerequisite for the attack described by Bard, the attacker has to force the sender to encrypt P^{check}, revealing the block $P_1^{check} = C_{particular-1} \oplus C_{IV} \oplus P^{guess}$.

$$\begin{aligned} C_1^{check} &= Enc_{Key}(P_1^{check} \oplus C_{IV}) \\ &= Enc_{Key}(C_{particular-1} \oplus C_{IV} \oplus P^{guess} \oplus C_{IV}) \\ &= Enc_{Key}(C_{particular-1} \oplus P^{guess}) \end{aligned} \tag{5.7a}$$

According to Subsection A.1.3 (formalized CBC) the following is obvious:

$$C_{particular} = Enc_{Key}(P_{particular} \oplus C_{particular-1}) \tag{5.7b}$$

This in turn, implies the validness of the following conclusion:

$$C_1^{check} = C_{particular} \Longrightarrow P_{particular} = P^{guess} \tag{5.7c}$$

As a precondition of the attack the previous block ($C_{particular-1}$) must be known which is mostly no problem (e.g. by eveasdropping on the connection). If this is the case, determining the IV used for this encryption remains an easy task, since it is simply the last block of the preceding message. As a last task an attacker has to find a way to inject data into the first block of the subsequent message.

The vulnerability was at the same time discovered by Bodo Möller who described the weakness in detail in a series of emails on an IETF mailing-list[4]. In this series, Möller described a fix which was later used by the OpenSSL project – prepending a single record with empty content, padding and MAC, to each message.

[4] `http://www.openssl.org/~bodo/tls-cbc.txt`

As potential fixes for the IV vulnerability Bard recommended the use of pseudo-random IVs or dropping CBC completely and switching to a more secure mode of operation.

TLS 1.1, specified in [45], fixed this vulnerability by introducing an `IV` field into the `GenericBlockCipher` structure which encapsulates block-cipher encrypted (and thus potentially CBC concatenated) data. Figure 5.9 lists the different structures for `GenericBlockCipher`s of TLS 1.0 and TLS 1.1.

```
1 struct {
2   opaque IV[SecurityParameters
        .record_iv_length];
3   block-ciphered struct {
4     opaque content[
          TLSCompressed.length];
5     opaque MAC[
          SecurityParameters.
          mac_length];
6     uint8 padding[
          GenericBlockCipher.
          padding_length];
7     uint8 padding_length;
8   };
9 } GenericBlockCipher;
```

```
1 block-ciphered struct {
2   opaque content[TLSCompressed
        .length];
3   opaque MAC[CipherSpec.
        hash_size];
4   uint8 padding[
        GenericBlockCipher.
        padding_length];
5   uint8 padding_length;
6 } GenericBlockCipher;
```

Figure 5.9.: Different structures for block-ciphers – *Source: RFC 2246 [43], RFC 5246 [45]*

5.2.14. Weak Cryptogrpahic Primitves Lead to Colliding Certificates

Category: *Attacks on the PKI*

Lenstra, Wang and de Weger described in 2005 how an attacker can create two valid certificates with equal hash values by computing MD5 collisions [90]. With colliding hash values it is possible to impersonate servers, since a valid signature can be abused to create rogue certificates.

5.2.15. Chosen-Plaintext Attacks on SSL Reloaded

Category: *Attacks on the Record Layer*

The attack by Bard discussed in Subsection 5.2.13 was reused by Bard in a publication of 2006 [91]. Overall, Bard addressed the same topics as before, but provided an attack sketch how to exploit this problem. Bard described a scenario in which an attacker uses a Java applet executed on the victim's machine, to mount the attack as described in Subsection 5.2.13. As a precondition, it is necessary to be able to access a commonly used SSL connection. Bard recommends some case of HTTP-Proxy or SSL based tunneling, as in case of a Virtual Private Network (VPN).

It is outlined that this scenario does not work if compression is used. Thus, many additional preconditions have to be fulfilled in order to be able to successfully attack a SSL connection (and the author states that it is *"challenging*

but feasible"). However, it gives an example for block-wise-adaptive chosen plaintext attacks. That Bard's attack scenario is applicable, in a slightly different implementation (by using JavaScript instead of Java applets), was proven years later by Rizzo and Duong with their B.E.A.S.T. attack tool (the attack is described in detail in Subsection 5.2.25).

The vulnerability is fixed with TLS 1.1, since it dictates the use of explicit IV's.

5.2.16. Weaknesses in X.509 Certificate Constraint Checking

Category: *Attacks on the PKI*

In 2008, Matthew Rosenfeld, better known as Moxie Marlinspike, published a vulnerability report (cf. *Internet Explorer SSL Vulnerability* [92]) concerning the certificate basic constraint validation of Microsoft's Internet Explorer. Internet Explorer did not check if certificates were allowed to sign sub-certificates (to be more technical: if the certificate is in possession of a `CA:TRUE` flag, restricting the maximum path length of the certificate chain). This means that any valid certificate, signed by a valid CA, was able to issue sub certificates for *any* domain.

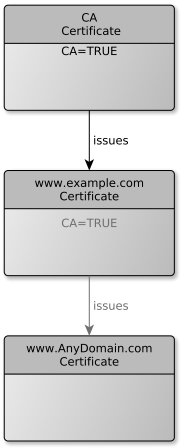

Figure 5.10.: Certificate chain with intermediate certificate – *based on Source: SSLSniff [92]*

Figure 5.10 illustrates this behavior. The `CA` flag of the intermediate certificate indicates that the subsequent certificate issuance is only valid if this flag is set to `TRUE`. Otherwise, certificate inspectors should reject the issued certificate, thus the intermediate certificate has insufficient rights to act as an intermediate CA and issue further certificates.

5.2.17. Weakened Security Through Bad Random Numbers

Category: *Other Attacks*

In 2008, Luciano Bello [93] observed during code review that the Debian-specific OpenSSL PRNG was predictable starting from version 0.9.8c-1, Sep 17 2006

until 0.9.8c-4, May 13 2008, due to an implementation bug. A Debian-specific patch removed two very important lines in the `libssl` source code[5] responsible for providing enough entropy when requesting the PRNG (cf. Listing 5.3).

```
1 MD_Update(&m, buf, j);
2   [ .. ]
3 MD_Update(&m, buf, j); /* purify complains */
```

Listing 5.3: Lines commented out lead to a serious SSL/TLS bug – *Source: Debian optimized OpenSSL Source Code*

This allowed a brute force attack, since the key space was significantly limited with these lines commented out.

5.2.18. Traffic Analysis

Category: *Attacks on the Record Layer*

George Danezis highlighted in an unpublished manuscript [94] ways how an attacker may use the obvious fact that minimal information, even thus the connection is TLS protected, remains unencrypted to analyze and track traffic.

Danezis makes use of the unencrypted fields, part of every SSL/TLS message, of the *Record* header fields. Listing 5.4 shows for example an encrypted *Record* consisting of unencrypted header fields and an encrypted payload. In Figure 5.11 the unencrypted parts are black while the encrypted parts are white.

```
1 struct {
2   ContentType type;
3   ProtocolVersion version;
4   uint16 length;
5   select (CipherSpec.cipher_type) {
6     case stream: GenericStreamCipher;
7     case block: GenericBlockCipher;
8   } fragment;
9 } TLSCiphertext;
```

Listing 5.4: TLS Ciphertext Record – *Source: RFC 2246 [43]*

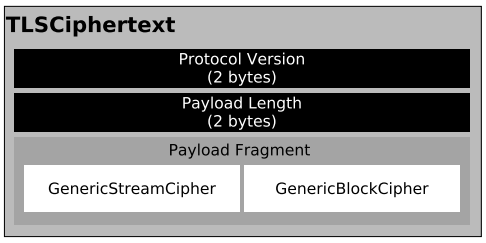

Figure 5.11.: `TLSCiphertext` record of TLS 1.0 – *based on Source: RFC 2246 [43]*

As can be seen, the fields `type`, `version` and `length` are always unencrypted – even in an encrypted record. This is necessary to be able to decode the

[5] `http://anonscm.debian.org/viewvc/pkg-openssl/openssl/trunk/rand/md_rand.c?p2=%2Fopenssl%2Ftrunk%2Frand%2Fmd_rand.c&p1=openssl%2Ftrunk%2Frand%2Fmd_rand.c&r1=141&r2=140&view=diff&pathrev=141`

Record correctly. In [77], the authors already criticized the presence of such unauthenticated fields. Additionally, RFC2246 is also aware of this information leak and advices TLS consumers to take care of this issue:

> "Any protocol designed for use over TLS must be carefully designed to deal with all possible attacks against it. Note that because the type and length of a record are not protected by encryption, care should be take to minimize the value of traffic analysis of these values." – *Source: RFC 2246 [43]*

Thus, the specification warns about this issue it remains unclear why it is necessary to leave this parts of information unauthenticated. But, even if the fields would be authenticated they would be left unencrypted.

The author identified several information leaks introduced by these unencrypted fields:

- Requests to different Uniform Resource Locator (URL)s may differ in length which results in different sized *Records*.
- Responses to requests may also differ in size which again yields to different sized *Records*.
- Different structured documents may lead to a predictable behavior of the client's application. For example a browser is normally very likely to fetch all images of a website – causing different requests and different responses.
- Content on public accessible sites is visible to everyone, so an attacker may link content (e.g. by size) to site content.

Moreover, an attacker could also actively influence the victims behavior and gain information about what she is doing (without knowledge of the encrypted content) by providing specially crafted documents with particular and distinguishable content lengths, structures, URLs or external resources. The traffic analysis is not only limited to SSL/TLS, but at least it reveals a weakness that can be exploited (the author proofs the attack to be practical by providing real world test results). Thus, the attack has to be taken into consideration when dealing with SSL/TLS attacks.

The author provides some hints on how the surface of the attack can be limited, but the practicability of the recommended measures remains questionable.

- URL padding – all URLs are of the same length
- Content padding – all content is of the same size
- Contribution padding – all submitted data is of the same size
- Structure padding – all traffic relies on an equal structure (e.g. a website with identical number of external resources a.s.o.)

This flaw was also discussed by Wagner and Schneier in [77], but in limited manner. In their paper, the authors credited Bennet Yee as the first one describing traffic analysis on SSL. As a countermeasure, Wagner and Schneier suggest support for random length padding not only for block-cipher mode, but also for all cipher modes. Moreover, they recommend not only support, but mandatory use in particular scenarios.

Sun et al. used similar observations in [95] to statistically identify websites (with partly sensitive content) with a high probability. This may enable drawbacks on encrypted content.

5.2.19. Renegotiation Flaw

Category: ***Other Attacks***

Ray and Dispensa discovered in [96] a serious flaw induced by the renegotiation feature of TLS. The flaw enables an attacker to inject data into a running connection without destroying the session. A server would accept the data, believing its origin is the client. This could lead to abuse of established sessions – e.g. a victim logged into a web application – an attacker could impersonate the legitimate client.

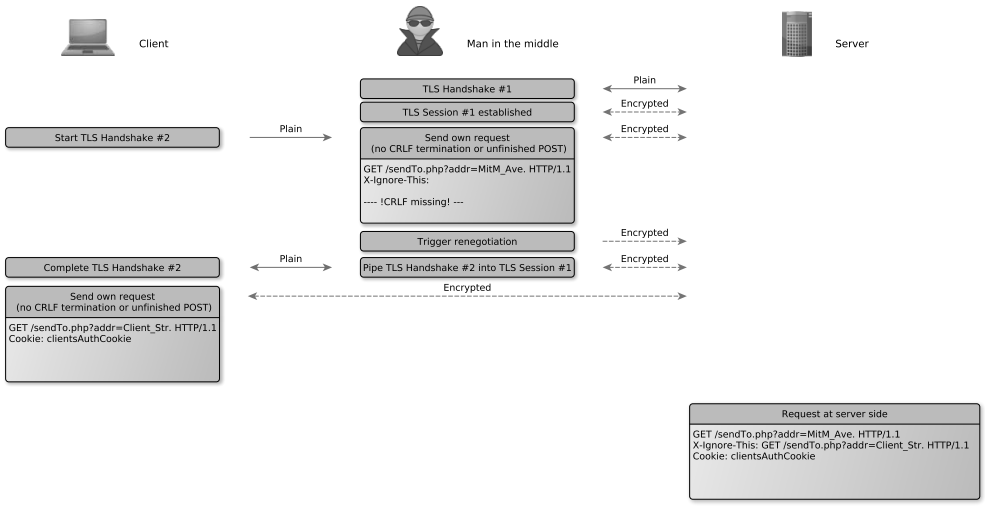

Figure 5.12.: Example scenario for the renegotiation attack – *based on Source: Renegotiating TLS [96]*

In Figure 5.12 it is shown how an attacker acting as MitM may abuse the renegotiation feature to inject arbitrary data into a TLS connection. In the example scenario an attacker concatenates two `GET` requests to send a gift to `MitM Ave.` instead of `Client Str.`. In this example, The authentication is done by cookies. The client attaches its authentication cookie to the own request. The attacker previously unfinished its own request by omitting a final `CRLF` and leaving a non-existent header field without value. The following happens at the server side: Since the last request has not been finished yet, the following data (the original client request) is concatenated with the previous data. The

`GET` request of the client becomes the value of the non-existent header field and will thus be ignored by the server. In the next line the authentication cookie remains valid (due to the necessary `\n` after the `GET` line). The result is a valid request with a valid cookie. It is important to note, that the attacker does not get the authentication cookie in plaintext, but the request is constructed to be concatenated on server side in a way that the first line of the clients request gets dropped (by using the header trick).

Anil Kurmus proved the flaw to be practical by stealing confidential data with a modified attack on Twitter[6] (he used an unfinished `POST` request) enabled by the renegotiation flaw[7].

Eric Rescorla proposed a TLS extension [53] to fix this flaw. Moreover, there are multiple ways to fix the problem. A proper solution is to enforce that data of the previous handshake has to be present in a renegotiation phase. Another solution is to disable the renegotiation feature completely at server side. But, detecting renegotiation triggered by a MitM remains a complicated task for a client.

5.2.20. Disabling SSL/TLS at a Higher Layer

Category: *Attacks on the PKI*

In February 2009, Moxie Marlinspike released `sslstrip`[8] a tool which disables the whole SSL/TLS layer. As a precondition, it is necessary for an attacker to act as MitM. To disable the security layer, the tool catches HTTP 301 – permanent redirection responses and replaces any encountered `https://` protocol definitions with `http://`. This causes the client to move to the redirected page and to communicate unencrypted and unauthenticated (when the stripping succeeds and the client does not notice that she is being fooled). The attacker can now open a fresh session to the requested server and pass-through or alter any client and server data. Figure 5.13 illustrates an attack sketch.

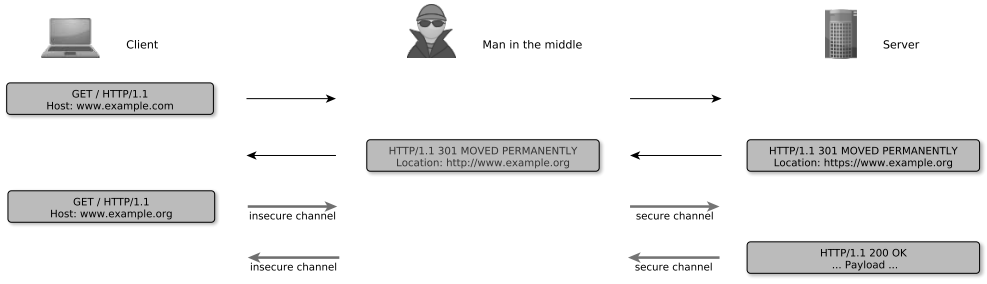

Figure 5.13.: Example scenario for a SSL stripping attack

It is important to note that this is not an attack on the SSL/TLS protocol, but it shows the importance of thoughtful designed protocols of higher levels. This flaw shows that HTTP was not explicitly designed to be carried by SSL/TLS.

[6] `http://www.twitter.com`
[7] `http://www.securegoose.org/2009/11/tls-renegotiation-vulnerability-cve.html`
[8] `http://www.thoughtcrime.org/software/sslstrip/`

The advantage of flexibility is also a major drawback in this case, since both layers can easily be separated without rising much attention.

5.2.21. Attacks on Certificate Issuer Application Logic

Category: ***Attacks on the PKI***
Attacks on the PKI by exploiting implementational bugs on CA side were demonstrated by Marlinspike in [97], who was able to trick the certificate issuer's logic by using specially crafted domain strings. Marlinspike was able to gain valid certificates for arbitrary domains, issued by trusted CAs.

As a prerequisite for the attack, it is necessary that the issuer only checks if the requester is the legitimate owner of the Top-level domain (TLD). An owner of a TLD may request for certificates issued for arbitrary sub-domains (e.g. the owner of the domain `example.com` is also allowed to request certificates for `anysubdomain.example.com`).

Marlinspike makes use of the encoding used by X509 – ASN.1. ASN.1 supports multiple String formats, all leading to slightly different PASCAL programming language (PASCAL) String representation conventions. PASCAL represents Strings length-prefixed (a leading length byte followed by the according number of bytes, each representing a single character). An example is given in Listing 5.5.

```
1 LENGTH | CONTENT
2    7   | S | S | L | N | E | R | D
```

Listing 5.5: String representation according to PASCAL convention

In contrast, C Strings are stored in `NULL`-terminated representation. Here, the character bytes are followed by a `NULL` byte signalizing the end of the String. This implies that `NULL` bytes are prohibited within regular Strings, because they would lead to preterm String termination. This scheme is illustrated in Listing 5.6.

```
1 CONTENT                     | Terminating NULL Byte
2  S | S | L | N | E | R | D | NULL
```

Listing 5.6: String representation according to C convention

This knowledge prepares the way for the `NULL`-Prefix attack:

1. Create a CSR for the target domain concatenated by a `NULL`-prefixed TLD owned by the attacker
2. Let it sign by a trusted CA
3. Impersonate the target domain

A sample domain name which could be used in a CSR is shown in Listing 5.7, assuming that the attacker is the owner of `example.com`.

```
1 www.targetToAttack.com\0.example.com
```

Listing 5.7: String representation according to C convention

The attack mentioned above works, because the CA logic only checks the TLD (`example.com`). The `NULL`-byte (`\0`) in the String is valid because of ASN.1's length-prefixed representation (where `NULL`-bytes within the payload String are valid). When the prepared domain String is presented to common application logic (mostly written in languages representing Strings `NULL`-terminated) the String is terminated before the end is reached. The result is that only the String afore the `NULL` byte (`www.targetToAttack.com`) is being validated.

A specialization of the attack are wild-card certificates: The asterisk (*) can be used to create certificates, valid – if successfully signed by a trusted CA – for **any** domain (e.g. `*\0.example.com`).

5.2.22. Attacking the PKI Directly

Category: *Attacks on the PKI*
Marlinspike describes in [98] an attack that aims at interfering the PKI to revoke certificates. By the use of OCSP a client application can check the revocation status of a certificate. OCSP will respond to queries with a `responseStatus` as can be seen in Listing 5.8.

```
1  OCSPResponse ::= SEQUENCE {
2      responseStatus         OCSPResponseStatus,
3      responseBytes          [0] EXPLICIT ResponseBytes OPTIONAL }
4
5  OCSPResponseStatus ::= ENUMERATED {
6      successful             (0),  --Response has valid confirmations
7      malformedRequest       (1),  --Illegal confirmation request
8      internalError          (2),  --Internal error in issuer
9      tryLater               (3),  --Try again later
10                                  --(4) is not used
11     sigRequired            (5),  --Must sign the request
12     unauthorized           (6)   --Request unauthorized
13 }
```

Listing 5.8: OCSP Response data type – *Source: RFC 2560 [28]*

The response structure in Figure 5.8 includes a major design flaw: The `responseBytes` field is optional, but it is the only field including a digital signature. This implies that the `responseStatus` is not protected by a digital signature, opening options for counterfeiting. Some of the `OCSPResponseStatus` types require the presence of `responseBytes`, but some do not. Especially `tryLater` does not force signed `responseBytes` to be present.

An attacker acting as MitM could respond to every query for particular certificates with `tryLater`. Due to lack for a signature the client has no chance to detect the spoofed response.

Since OCSP is a building block of the PKI of SSL/TLS this attack may influence all clients relying on OCSP for certificate revocation checks.

5.2.23. Wildcard Certificate Validation Weakness

Category: *Attacks on the PKI*
Moore and Ward published a Security Advisory (see *Multiple Browser Wildcard Certificate Validation Weakness* [99]) concerning wildcard (*) usage when IP

addresses are used as CN Uniform Resource Identifier (URI) in X509 certificates. According to RFC 2818 [100] wildcards are not allowed for IP addresses. The authors found multiple browsers treating IP addresses including wildcard characters as certificate CN as valid and matching. Browsers could be fooled to accept issued certificates with exemplarily `CN="*.168.3.48"`. This certificate was treated as valid for any server with a `".168.3.48"` postfix.

5.2.24. Conquest of a Certification Authority

Category: *Attacks on the PKI*
At 15 March 2011, the *Comodo CA Ltd.*[9] CA was compromised by an attacker [101]. The attacker used a reseller account to issue 9 certificates for popular domains, but the real purpose of the attack remains unclear.

Soon after the attack was discovered the concerned certificates have been revoked. Due to previously discussed weaknesses affecting the certificate revocation infrastructure (cf. Subsection 5.2.22) it could be possible that these certificates are considered valid to parties with out-of-date or missing certificate revocation lists.

It is crucial to note that the attacker did not compromise Comodo's infrastructure directly (key material and servers were not compromised at any time). Moreover, valid credentials of a reseller were used to sign CSRs.

5.2.25. Practical IV Chaining Vulnerability

Category: *Attacks on the Record Layer*
Rizzo and Duong presented in [102] a tool called B.E.A.S.T. – *B*rowser *E*xploit *A*gainst *S*SL/*T*LS – that is able to decrypt HTTPS traffic (e.g. cookies). In general, the authors implemented and extended ideas and attacks by Bard [89], [91], Möller and Dai[10]. Rizzo and Duong used the CBC mode applied to blockciphers (cf. Figure A.2) and predictable IVs (cf. Subsection 5.2.13) to guess plaintext blocks and verify the validness of the guess. To verify a guess x an oracle $\mathcal{O}_{Guess}(P^*)$ is required returning true or false depending on the correctness of the guess.

Such an oracle can be built if the IVs used by CBC are known to the attacker. Consider an adversary already in possession of C_{j-1} (the preceding ciphertext block). Due to CBC usage this ciphertext block is used as IV for the next plaintext block P_j. It is possible to make a plaintext guess P^* and verify it by observing whether the following equation holds (the differing indexes i and j are necessary, due to a practical problem which is discussed later):

[9] `http://www.comodo.com`
[10] `http://www.weidai.com/ssh2-attack.txt`

$$
\begin{aligned}
C^* &= Enc_{Key}(P^*) \\
P^* &= Dec_{Key}(C^*) \\
P_j &= C_{j-1} \oplus P^* \\
&= C_{j-1} \oplus Dec_{Key}(C*) \\
C_j &= Enc_{Key}(C_{j-1} \oplus P_j) \\
P_{j+1} &= C^* \oplus C_{i-1} \oplus P_i
\end{aligned}
$$

$$
\begin{aligned}
C_{j+1} &= Enc_{Key}(C_j \oplus P_{j+1}) \\
&= Enc_{Key}(Enc_{Key}(C_{j-1} \oplus P_j) \oplus P_{j+1}) \\
&= Enc_{Key}(Enc_{Key}(C_{j-1} \oplus C_{j-1} \oplus P^*) \oplus P_{j+1}) \\
&= Enc_{Key}(Enc_{Key}(P^*) \oplus P_{j+1}) \\
&= Enc_{Key}(C^* \oplus C^* \oplus C_{i-1} \oplus P_i) \\
&= Enc_{Key}(C_{i-1} \oplus P_i) \\
&= C_i
\end{aligned}
\tag{5.8}
$$

If the equation holds, $C_{j+1} = C_i$ - the attacker's guess was right. A major reason complicating the attack is that even if an attacker may control an arbitrary plaintext block P_j most practical application scenarios, such as SSL/TLS, still prepend header data. While this header data is often predictable it can still not be altered by an attacker. The authors circumvent this problem by forcing that the uncontrollable plaintext parts remain unchanged in correlating messages (C_{j-1} and C_{i-1} have identical prepended headers), so that the observed output remains usable to an attacker.

To adopt the technique to SSL/TLS and decrypt ciphertexts byte-wise the authors propose a new kind of attack named *block-wise chosen-boundary attack*. Hereby, an attacker is able to move a message before encryption in its block boundaries. This means an attacker may prepend a message with arbitrary data so that it is split into multiple blocks of the appropriate block-size of the cipher. It is for example possible to split a message of full block-size into two blocks: The first one consisting of arbitrary data and the first byte of the original message and the second block consisting of the remaining bytes and a single unused byte. So prefixing a message with an attacker defined amount of data shifts the message (if necessary into a new block). An attacker is absolutely free to prepend any data of arbitrary length. An example is given in Figure 5.14.

First Byte Decryption. The first byte decryption algorithm uses the technique illustrated in Figure 5.14 to shift the original message, so that only a single (unknown) byte is left in the first block. This enables efficient guessing the first byte of a known ciphertext, since only 1byte = 8 bit = 256 possible values = 256 guesses are necessary (in practice the room of possible values can be further reduced, since not all guesses represent valid values).

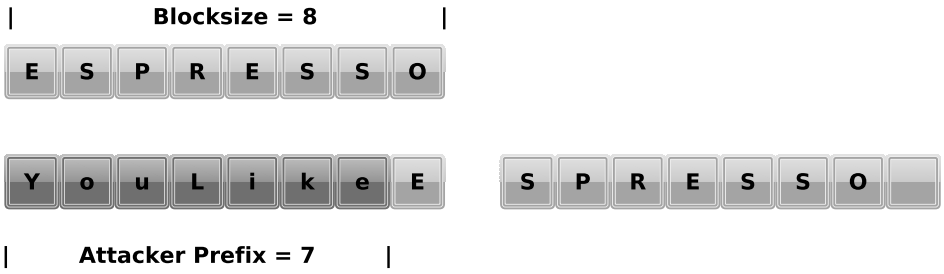

Figure 5.14.: Example boundary shifting

1. `R = R[1]..R[blockSize-1]`
   Generate random string R of length $blockSize - 1$ bytes

2. $C^* = \mathcal{O}_{Guess}(R|P)$
   R is sent to a victim which prefixes the original, unknown plaintext P with R and sends it to the oracle $\mathcal{O}_{Guess}(R|P)$. The oracle divides it into s blocks of size $blockSize$, $P_1^*, ..., P_s^*$. Due to concatenation by the victim, the first block of the plaintext P^* is of the following structure: $R[0]...R[blockSize - 1]P[1]$. The prepended plaintext P^* is encrypted in CBC mode to $C^* = [C_0^*, ..., C_s^*]$.

3. Set `i=0`

4. $C_i' = \mathcal{O}_{Guess}(P_1' = C_1^* \oplus C_s^* \oplus R|i)$

5. `if(`$C_i' = C_1^*$`) {`
   a) $P[0] = i$ `found`
   `} else {`
   a) i`++`
   b) `goto Step 4`
   }

This is an exact adoption of Definition 5.8:

$$\begin{aligned} C_i &= Enc_{Key}(C_s^* \oplus P_i) \\ &= Enc_{Key}(C_s^* \oplus C_0^* \oplus C_s^* \oplus R|i) \\ &= Enc_{Key}(C_0^* \oplus R|P[1]) \text{ , if}(i = P[1]\ ) \\ &= Enc_{Key}(C_0^* \oplus P_1^*) \\ &= C_1^* \end{aligned} \tag{5.9}$$

Full Message Decryption. To decrypt a full message, the attacker adjusts the amount of random prefix data so that the next unknown byte is always

the last byte in a block. The message is shifted in a way that the scenario illustrated in Figure 5.14 applies to the next unknown byte. Finally, this leads to a byte by byte message decryption.

To demonstrate practical attacks Rizzo and Duong targeted HTTPS (HTTP protected by SSL/TLS). The goal is to decrypt cookies send by a browser to a server for authentication purpose. For successful implying the algorithms some preconditions have to be met:

1. An attacker is able to sniff the network traffic between the victim and the server
2. An attacker can force the victim to perform HTTP(S) `POST` requests to the server
3. An attacker can fully control the path of the `POST` request's target
4. An attacker forces the victim to visit a website under control of the attacker to inject exploit code (further called agent)

When the B.E.A.S.T. Attacks. The attacker wants to decrypt message P including the desired session cookie among some partially known or predictable header fields. Due to network sniffing encrypted messages sent by the victim's browser C can be fully captured. Furthermore, $W[1..l]$ denotes all valid bytes of a HTTP request header.

1. The attacker forces the victim to perform HTTP(S) `POST` requests to the target server (path `https://www.example.org/AAAAAA`). The victim's browser creates the request (P =`POST /AAAAAA HTTP/1.1 <CR> <LF> <REQUEST HEADERS> <CR> <LF> <REQUEST BODY>`), chunked into blocks of the *blockSize* of the block-cipher, and encrypts it in CBC mode. The resulting ciphertext $C = Enc_{Key}(P)$ is send to `www.example.org`.
2. The attacker eavesdrops $C = [C_0, ..., C_n]$ and sets `i=1`.
   The data is chunked as depicted in Figure 5.15, causing the first unknown byte (the beginning of the request headers) to be the last byte in a plaintext block. Remember, this data is encrypted. The attacker influenced the content, but does not know the remaining bytes!

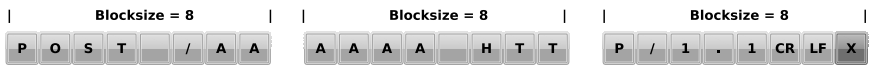

Figure 5.15.: B.E.A.S.T example

3. Compute $P_i = C_n \oplus C_2 \oplus P'$, with $P' =$ `P/1.1<CR<LF>`$W[i]$
4. Append P_i to the request body.
5. Force the victim's browser to compute $C_i = Enc_{Key}(C_n \oplus P_i)$.
   C_i is sent to the server by the victim's browser.

6. `if(`$C_i = C_3$`) {`
   a) $X = W[i]$ `found`
   `} else {`
   a) i`++`
   b) `goto Step 3`
   `}`

The author's utilized a Java applet delivered by the website under control of the attacker as agent. The applet is loaded and started by the client's browser and contains the necessary exploit code to mount the attack.

Due to this massive vulnerability, migration to TLS Version 1.1 has been recommended since by IETF.

5.2.26. Conquest of Another Certification Authority

Category: ***Attacks on the PKI***
Soon after the attack on Comodo, a Dutch Certification Authority – *DigiNotar* – was completely compromised by an attacker [103]. In contrast to the Comodo impact, the attacker was able to gain control over the DigiNotar infrastructure. The discovery was eased by Google's Chrome web browser who complained about a mismatching certificates for Google-owned domains. The browser stores hard coded copies of the genuine certificates for Google and thus was able to detect rogue certificates issued by a trusted CA.

5.2.27. ECC Based Timing Attacks

Category: ***Attacks on the Handshake Phase***
At ESORICS[11], Brumley and Tuveri [104] presented an attack on ECDSA based TLS connections. As to their research only OpenSSL (see Section 4.1.1) seemed to be vulnerable. The vulnerability is indexed with Common Vulnerabilities and Exposures (CVE) *CAN-2003-1047*[12].

The problem arose form the strict implementation of an algorithm for improving scalar multiplications which ECC heavily relies on (point multiplication, e.g. $[k]P$, where $k \in \mathbb{Z}$ and $P \in E, E(\mathbb{F}_{2^m}) : y^2 + xy = x^3 + a_2x^2 + a_6$ – *Source: Remote Timing Attacks Are Still Practical [104]).* The implemented algorithm for performing such scalar multiplications, known as the Montgomery power ladder [105] (with improvements by López and Dahab [106]), is timing resistant from a formal point of view (relevant parts are the nested `for` loops in Listing 5.9, lines 13-31), but contained a timing side-channel from implementational point of view. The developers optimized the performance of the algorithm by reducing the repetitions of internal loops (relevant part is the `while` loop in Listing 5.9 decreasing the `j` variable, line 4), thus leading to the side channel.

[11] `https://www.cosic.esat.kuleuven.be/esorics2011/`
[12] `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-1047`

```
1 /* find top most bit and go one past it */
2 i = scalar->top - 1; j = BN_BITS2 - 1;
3 mask = BN_TBIT;
4 while (!(scalar->d[i] & mask)) { mask >>= 1; j--; }
5 mask >>= 1; j--;
6 /* if top most bit was at word break, go to next word */
7 if (!mask)
8   {
9   i--; j = BN_BITS2 - 1;
10  mask = BN_TBIT;
11 }
12
13 for (; i >= 0; i--)
14   {
15   for (; j >= 0; j--)
16     {
17     if (scalar->d[i] & mask)
18       {
19       if (!gf2m_Madd(group, &point->X, x1, z1, x2, z2, ctx)) goto
           err;
20       if (!gf2m_Mdouble(group, x2, z2, ctx)) goto err;
21       }
22     else
23       {
24       if (!gf2m_Madd(group, &point->X, x2, z2, x1, z1, ctx)) goto
           err;
25       if (!gf2m_Mdouble(group, x1, z1, ctx)) goto err;
26       }
27     mask >>= 1;
28     }
29   j = BN_BITS2 - 1;
30   mask = BN_TBIT;
31   }
```

Listing 5.9: Timing side channel in `ec_GF2m_montgomery_ point_multiply` function – *Source: OpenSSL 0.9.8 Code crypto/ec/ec2_mult.c*

The problem arose from the fact that every operation within the `for` loops takes constant time, but the time of repetitions differs dependent on the function arguments. To be more precise, lines 13-31 are skipped for each leading 0 bit of the function's input.

When arranging the function input k (the integer of the scalar multiplication) in a binary tree (leading zeros stripped) the tree's height is $\lceil \log(k) \rceil - 1$ (-1 since the first bit is expected to be 1, thus ignoring the first level of the tree). This implies that the runtime of the algorithm is $t(\lceil \log(k) \rceil - 1)$, where t denotes the constant time of lines 13-31. The result is that the measurement of the processing time of the algorithm enables drawbacks on the function's input.

Brumley and Tuveri combined this side channel with the lattice attack of Howgrave-Graham and Smart [107] to recover secret keys. For this to work, they identified that creating ECDSA signatures relies on scalar multiplication and as a consequence utilize the vulnerable `ec_GF2m_ montgomery_point_multiply` function.

ECDSA signatures are generated in TLS/SSL when `ECDHE_ECDSA` cipher suites are used and thus are vulnerable to the discussed attack. The paper's authors measured the time difference between the client initiated `ClientHello` message and the reception of the `ServerKeyExchange` message during the *Handshake Phase*. The latter message contains an ECDSA signature over a digest (protecting relevant parts necessary for further establishment of cryptographic material). As this digital signature can only be created on-the-fly during the *Handshake Phase* and not in advance, an adversary is able to measure runtime of the vulnerable function.

5.2.28. Computational Denial of Service

Category: *Other Attacks*
In October, 2011 the German Hacker Group *The Hackers Choice*[13] released a tool called `THC-SSL-DOS`[14]. The tool aims at creating huge amount of load on servers by overwhelming the target with SSL/TLS handshake requests. Boosting system load can either be done by establishing new connections or by repeated renegotiation. Assuming that the majority of the computations is done by the server, the attack can be used to create more system load on the server than on the own device – leading to a DoS. The server is forced to perform costly operations again and again.

In an analysis performed by Eric Rescorla[15], it is stated that the tool is not that more effective than a conventional DoS attack opening lots of parallel connections consuming bandwidth or other of the server's resources.

5.2.29. Short Message Collisions and Busting the Length Hiding Feature of SSL/TLS

Category: *Attacks on the Record Layer*
In [108] Paterson, Ristenpart and Shrimpton outlined an attack related to the MAC-then-Pad-then-Encrypt scheme in combination with short messages. In particular, the attack is applicable if all parts of a message (message, padding, MAC) fit into a single block of the cipher's block-size (see Table 5.2) whcih is the case e.g. for truncated MACs (see RFC [109][16].

Symbol	Meaning
MessageLength	Message length in bytes
MACLength	MAC length in bytes
PaddingLength	Padding length in bytes
BlockSize	Block size of the block-cipher in bytes

Table 5.2.: Definitions for the Short Message Collisions Attack

[13] `http://www.thc.org`
[14] `http://www.thc.org/thc-ssl-dos/`
[15] `http://www.educatedguesswork.org/2011/10/ssltls_and_computational_dos.html`
[16] In the paper, the authors use a 128 bit (16 byte) block cipher and a 80 bit (10 byte) truncated MAC algorithm

Under the definition of Table 5.2 it is possible to create multiple ciphertexts leading to the same plaintext message. For the attack it is necessary to gain possession of a ciphertext $C = [C_0, C_1, C_2]$, $BlockSize = 16$ bytes and $MessageLength = 5$ bytes. The attacker now computes a new value $C' = [C'_0, C_1], C'_0 = [C_0 \oplus ([00]^{15}10)]$, ($0x00$ occurs 15 times followed by $0x10$). This leads to a new ciphertext consisting of only 2 words of $BlockSize = 16$. For better understanding a short recap of the original plaintext data follows. The corresponding plaintext of C is as listed above, whereby $PaddingLength = 17$ (16 bytes with value $0x10$, followed by the padding length $0x10$ – 17 times $0x10$ in total).

$$[MSG]^{MessageLength}|[MAC]^{MACLength}|[PAD]^{PaddingLength}|PaddingLength \tag{5.10}$$

If the last block C_2 is stripped off and the last byte of C_1 is XOR-ed with $0x10$ the message's last byte will be $PaddingLength = 0$ – no padding added (keep in mind that the last byte is reserved for the padding *length* and is not part of the padding itself – this means that a missing padding is indicated $PaddingLength = 0$). As a consequence C_2 is stripped, leading to two different ciphertexts with the same message content.

5.2.30. Message Distinguishing

Category: ***Attacks on the Record Layer***
Paterson et al. extended in [108] the attack described in Subsection 5.2.29 enabling an attacker to distinguish between two messages. Therefore, the attacker has to be able to act as MitM. The following gives an example attack sketch:

1. Attacker knows that either `YES` (3 bytes) or `NO` (2 bytes) are about to be send
2. Intercept corresponding ciphertext $C = [C_0, ...C_2]$
3. Generate $C' = [C'_0, C_1], C'_0 = [C_0 \oplus ([0x00]^{14}, 0x10, 0x10, 0x10, 0x10)]$
4. Send C' instead of C
5. In error case the message was `YES`, otherwise `NO`

For explanation it is necessary to look at the original plaintext of C which is either $YES|MAC_{YES}|[0x12]^{19}$ or $NO|MAC_{NO}|[0x13]^{20}$. The following sketches C_0 while C_1 remains untouched during the attack and contains either $[0x12]^{16}$

or $[0x13]^{16}$. C_2 is simply dropped.

$$\begin{aligned} \text{original } C_0 &: [N, O, [\text{MAC bytes}]^{10}, 0x13, 0x13, 0x13, 0x13] \\ \text{modified } C_0^{'} &: [N, O, [\text{MAC bytes}]^{10}, 0x03, 0x03, 0x03, 0x03] \\ & \text{or} \\ \text{original } C_0 &: [Y, E, S, [\text{MAC bytes}]^{10}, 0x12, 0x12, 0x12] \\ \text{modified } C_0^{'} &: [Y, E, S, [\text{MAC bytes}]^{10}, 0x02, 0x02, 0x02] \end{aligned} \tag{5.11}$$

This implies that the $\oplus$ operation with $[0x10]^4$ would affect the MAC, since 4 bytes are modified, but only 3 bytes of padding are available. The MAC value would be invalidated causing the logic to terminate the connection immediately (since `bad_record_mac` is a `fatal` *Alert type*).

The attack remains unexploitable due to the fact that it is only possible for 80 bit truncated MACs, since $BlockSize$ is either 64 for DES or 128 for Advanced Encryption Standard (AES) and $MACLength \geq 160$ for HMAC-SHA{160, 224, 256, 384, 512}). While in versions prior to TLS 1.2 the use of truncated MACs is possible, version 1.2 restricts the its use. Anyway, truncated MACs are still possible in TLS 1.2, by the use of protocol extensions.

5.2.31. Breaking DTLS

Category: ***Attacks on the Record Layer***
In [110] AlFardan and Paterson applied Vaudenay's attack (for details see Subsection 5.2.8) to DTLS. DTLS is a slightly different version of regular TLS, modified to adjust to unreliable transport protocols, such as UDP. These adjustments include two major differences when compared to standard TLS:

1. Complete absence `Alert` messages
2. Messages causing protocol errors (bad padding, invalid MAC, ...) are simply ignored, instead invalidating the key material of this session and terminating the connection

These adjustments are advantageous, as well as disadvantageous at the same time. Vaudenay's attack may only work on DTLS since bad messages do not cause session invalidation. But with a lack for `Alert` messages the oracles introduced by Vaudenay cannot be used without adjustment. The attacker gets no feedback if the message created by the attacker contained a valid padding or not. The authors adjusted Vaudenay's algorithms by using a timing oracle arising from different processing branches with unequal time consumption.

The work of AlFardan and Paterson covers the OpenSSL and GnuTLS - The GNU Transport Layer Security Library (GnuTLS) implementations, both vulnerable to a timing oracle enhanced version of Vaudenay's attack. The timing oracle of OpenSSL arises from a padding dependent MAC check. In case of correctly padded data the MAC is checked, where in case of an incorrect padding the MAC verification is skipped. This behavior leads to a measurable timing difference allowing drawbacks on the validity of the padding. GnuTLS, in contrast contains a timing side channel during the decryption process where sanity

checks are performed before decryption.
While the side channel of OpenSSL is a consequence of missing implementation of recommended countermeasures, even GnuTLS is vulnerable which implements all given countermeasures. When timing differences are too little, reliable timing differences are gained by sending multiple copies leading to an accumulation of timing differences. To force responses of the target the *Heartbeat* extension (cf. *Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension* [111]) of TLS/DTLS was used. This extension forces the ongoing mutual exchange of `Heartbeat` messages in order to verify each party's application is up and running. According to AlFardan and Paterson, it was necessary for the proof of concept attack to disable a protocol option called *anti-replay*, which is enabled by default.

Beside the proof of concept attack, the authors provided useful observations concerning processing time of en-/decryption algorithms.

- MAC verification is slower than decryption
- 3DES is slower than AES
- Particular processing time observations under special conditions

As a recommendation to future specification authors it is suggested that defining standards and specifications by only defining the differences to previous standards should be avoided (DTLS is a sub-specification of TLS).

5.2.32. Even More Improvements on Bleichenbacher's Attack

Category: ***Attacks on the Handshake Phase***
In [112] Bardou, et al. significantly improved Bleichenbacher's attack (see Subsection 5.2.7) far beyond the already known improvents by Klíma, Pokorny and Rosa (see Subsection 5.2.12).The authors observed that Step 2b of the original algorithm is executed extremly seldom – if at all – where as Step 2c is highly efficient, but not applicable in many cases. Thus, they enhanced the process of interval computing in a way that Step 2c can be applied to most cases. These optimization make use of, what the authors call *trimming* (see the original paper [112] for details). Instead of searching with interval boundaries $2b \leq p < 3b$, they provide new boundaries based on the trimming technique which siginificantly reduced the intervals sizes.

between If where that it With this knowledge, the interval M_0 from Step 1 of Bleichenbacher's original algorithm can be reduced. A further interesting optimization is an observation on 'holes' which cannot contain possible values for s_i (leading to PKCS#1 v1.5 compliant messages). During the search for possible s_i values these holes can simply be skipped.

Finally, the authors combined their results with the improvements of Klíma, Pokorny and Rosa [88] and were able to significantly speed up Bleichenbacher's algorithm.

5.2.33. Attacks on Non-Browser Based Certificate Validation

Category: *Attacks on the PKI*
At CCS 2012[17], Georgiev et al. [113] uncovered that widespread, common used libraries for SSL/TLS suffer from vulnerable certificate validation implementations. The authors revealed weaknesses in the source code of OpenSSL, GnuTLS, JSSE, HttpComponents HttpClient (Apache HttpClient)[18], Java WebSocket Client Library (Weberknecht)[19], Client for URLs (cURL)[20], PHP: Hypertext Preprocessor (PHP)[21] and Python (Python)[22] and applications build upon or with these stacks. The authors examined the root causes for the bugs and were able to exploit most of the uncovered vulnerabilities. As major causes for these problems the author's identified bad and misleading Application Programming Interface (API) specifications, lacking interest for security concerns at all (even for banking applications!) and the simple abscence of essential validation routines. Especially the widespread OpenSSL and GnuTLS libraries provide a confusing API leaving important tasks influencing security to the API consumer. Even worse, the API of cURL was attested to be "almost perversely bad" – *Source: The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software [113].*

The paper especially targets the following security tasks and the robustness of the libraries' code responsible for these tasks:

- Certificate chaining and verification
- Host name verification
- Check for certificate revocation
- X509 Extension handling and processing

5.2.34. Practical Compression Based Attacks

Category: *Attacks on the Record Layer*
In September 2012, Juliano Rizzo and Thai Duong presented the *C.R.I.M.E.* attack – *C*ompression *R*atio *I*nfo-Leak *M*ade *E*asy or *C*ompression *R*atio *I*nfo-Leak *M*ass *E*xploitation. *C.R.I.M.E* targets HTTPS and is able to decrypt traffic, steal cookies and take over sessions. It exploits a known vulnerability caused by message compression discovered by Kelsey in 2002 [83] (see Subsection 5.2.9). The concept was adopted in order to decrypt HTTPS traffic.

The basic attack makes use of the fact that compression reduces the size of a plaintext. Given that an attacker is able to obtain the (real) size of an encrypted, compressed plaintext (without padding), she might be able to decrypt

[17] `http://www.sigsac.org/ccs/CCS2012/`
[18] `http://hc.apache.org/httpcomponents-client-ga/`
[19] `http://code.google.com/p/weberknecht/`
[20] `http://curl.haxx.se/`
[21] `http://www.php.net/`
[22] `http://www.python.org/`

parts of the ciphertext by observing the differences in length. It follows a brief introduction to the attack principles[23].

Redundant plaintext can efficiently be compressed, leading to a shorter ciphertext. An attacker does neither know the plaintext, nor the compressed plaintext, but is able to observe the length of the ciphertext. Basically, the attacker simply prefixes the secret with guessed subsequences (e.g. by systematically trying possible subsequences) and observes if it can be compressed (by observing the resulting ciphertext length). A decreased ciphertext length implies redundancy, so it is very likely that the guessed subsequence prefixed to the original plaintext data caused redundancy in the plaintext which could be substituted. The attack of guessing is explained in detail in Subsection 5.2.9.

It can be concluded that with higher redundancy of input data, the better the compression ratio resulting in reduced output length. This implies that a guess having something in common with the secret, will have a higher compression rate leading to a shorter output. When such an output is observed the attacker knows that the guess has something in common with the secret. This example is for explanation only. In practice, Rizzo and Duong had to face the `DEFLATE` compression algorithm, defined in RFC 1951 [114], and its peculiarities.

As preconditions an attacker must be able to sniff the network and attract a user to visit a website under the attackers control. Further on, both - client and server – have to use compression enabled SSL/TLS. The website controlled by the attacker delivers *C.R.I.M.E.* as JavaScript to the victim's browser. Once the script is running an attacker starts guessing the right cookie value and forcing the victim to transport the guess concatenated with the desired cookie as an SSL/TLS message. In the presentation of the attack, the authors used `GET` requests to transport the guess (e.g. `GET /twid=`**GUESS**). The process of guessing the right values is depicted in Listing 5.10.

```
1  GET /twid=a
2  Host: twitter.com
3  User-Agent: Chrome
4  Cookie: twid=secret
5
6  ...
7
8  GET /twid=s
9  Host: twitter.com
10 User-Agent: Chrome
11 Cookie: twid=secret
```

Listing 5.10: C.R.I.M.E. guessing a cookie value – *Source: The CRIME attack – slides from Ekoparty conference*

It is important to know that the cookie is automatically added by the user's browser and remains unknown until decryption to the attacker. An attacker simply forces the user's browser (by the use of JavaScript) to visit `/twid=GUESS`. The latter request will be of different size as the ones before, since `/twid=s` appears twice in the request and thus can be compressed, whereas `/twid=a` is not redundant and cannot be compressed.

[23] For a complete attack description refer to Subsection 5.2.9

5.2.35. ECC Based Key Exchange Algorithm Confusion Attack

Category: ***Attacks on the Handshake Phase***
In [115], Mavrogiannopoulos, Vercauteren, Velichkov and Preneel showed that the key exchange algorithm confusion attack by Wagner and Schneier, as discussed in Subsection 5.2.5, can also be applied to ECDH(E). The attack requires a large amount (2^{40}) of eavesdropped messages (signed ECDHE public keys of the server) to succeed. Since the presented attack requires the attacker to act as an active MitM these messages need to be eavesdropped during a running session which is a very strong requirement. Mavrogiannopoulos et al. calculated the estimated runtime of an successful attack, based on data gathered from a simulated environment. According to the authors, their attack is not feasible yet, due to the large amount of eavesdropped (signed) messages. But, as already discovered with other attacks, it may be a question of time when the attack is enahnced to be practical or the computing resources for increase. Just as discussed by Wagner and Schneier the main problem is the lack for a content type field indicating the algorithm type of the contained data – which implicitly indicates how to decode.

5.2.36. Timing Attack on MAC Processing

Category: ***Attacks on the Record Layer***
AlFardan and Paterson published a timing attack based on MAC processing in [116]. Basically, the authors highlight the fact that MAC processing is no time-constant operation. Based on the amount of input, the MAC computation takes more or less time. Additionally, breaking the padding of SSL/TLS records[24] may lead to the inconvenient situation where it is not obvious on which data the MAC computation should be performed. In such a scenario, when the *real* amount of data to be MACed cannot be determined, RFCs 4346 [44] and 5246 [45] recommend to perform a MAC computation assuming zero-length padding as a countermeasure to prevent attacks as presented by Canvel et al. (see Subsection 5.2.11). While this might lower the surface for a timing side channel, the authors refute the RFCs assumption that "[...] it is not believed to be large enough to be exploitable [...]" – *Source: RFC 4346 [44] and RFC 5246 [45]* by presenting distinguishing and plaintext recovery attacks on TLS and DTLS.

The padding in SSL/TLS follows a strict scheme: The last byte of the padding contains the padding length (excluding the padding length byte). This causes a padding, even thus it is considered to be of zero length, to always consist of at least one byte, the padding length. This scheme complies with the PKCS#7 padding scheme (see RFC 5652 [81]), but additionally the padding is postfixed with a length byte. According to the previous definition, a padding has the following structure: *padding bytes — padding length*. The padding bytes always equal the number of padding bytes (the value of the padding length). As follows, a valid padding consisting of as many *padding bytes* as defined in *padding length*

[24] Due to the Mac-then-Pad-then-Encrypt scheme, one can easily create badly padded records (see e.g Subsection 5.2.11), since the padding bytes are not protected by the MAC.

and the *padding length* byte itself, is thus always of size *padding length*+1. Obviously, the highest posible padding value and thus also *padding length* value is `0xFF`. The following gives some examples of valid paddings:

- `00`
- `01 | 01`
- `02 | 02 | 02`
- ...

If the padding scheme is violated, the padding has to be treated as invalid. In such a case the previously discussed recommendation of treating the record as being of zero-length should be applied. This may cause the inclusion of bytes originally belonging to the padding to be included in the MAC computation which means that the amount of input bytes to the MAC function is higher than in the valid case. More input value could leverage timing side channels.

The attack named *Lucky Thirteen* refers to the fact that the MAC value is computed over each record payload prepended with exactly 13 bytes (8 bytes *Sequence Number* and 5 bytes obligatory *Record* header fields). Figure 5.16 illustrates these 13 bytes.

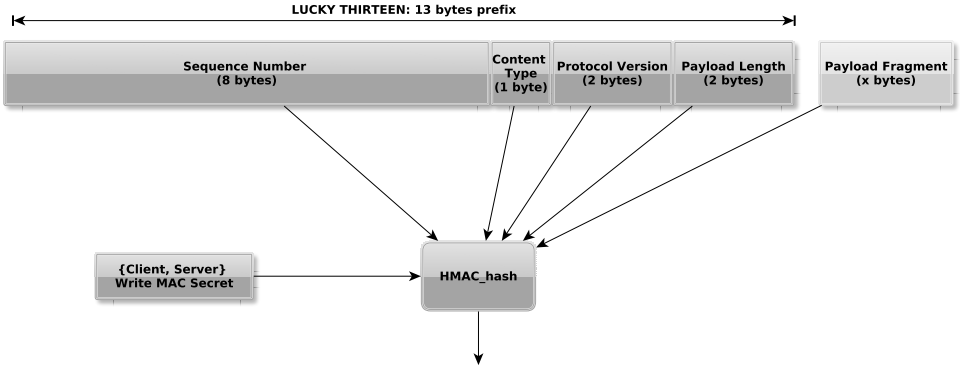

Figure 5.16.: (H)MAC computation with exactly 13 bytes prefix followed by the payload.

Distinguishing Attack. AlFardan and Paterson at first describe a distinguishing attack on two plaintext messages (M_0, M_1) of equal size block-cipher encrypted in CBC mode. Figure 5.17 shows the structure of both messages.

The attacker submits both messages to an encryption oracle $\mathcal{O}_{enc}$ which returns the encrypted plaintext C of one of both messages $M_x, x \in [0, 1]$. M_0 consists of exactly 32 bytes + 256 padding FF bytes while M_1 is of exactly 287 bytes + a single 00 byte. Now, both messages are of 288 bytes in total (leading to exactly 18 plaintext blocks if the block size is 16). If padding is applied, C is defined as follows: $C = enc(M_x|MAC|Padding|PaddingLength)$ and what will be send is $RecordHeader|C$. The attacker does not know which message

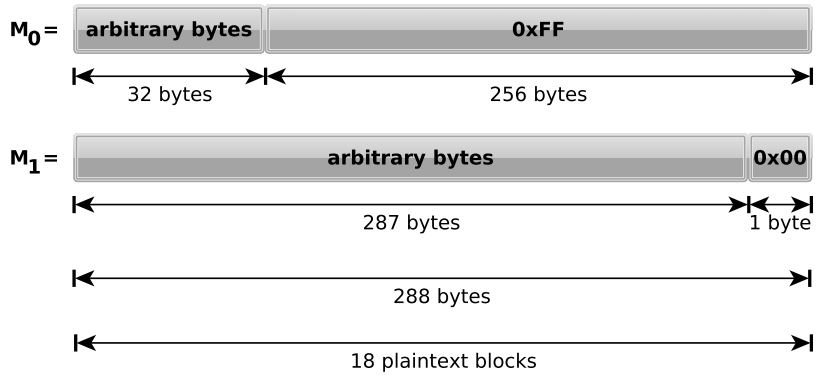

Figure 5.17.: Lucky Thirteen Message Distinguishing

was selected, so x is unknown. The attacker can gain knowledge about x by applying the following distinguishing attack:

1. Build $C' = C$ with $MAC|Padding|PaddingLength$ being removed, by stripping off all bytes beyond position 288 (or – in case explicit IV's are used – every bytes beyond the position $blocksize + 288$, since the first block contains the IV)[25].

2. Pass $RecordHeader|C'$ to an decryption oracle $\mathcal{O}_{dec}$ and measure processing time

3. $x = \begin{cases} 0, \text{ then the } \textit{padding length} \text{ is } \texttt{0xFF} \\ 1, \text{ then the } \textit{padding length} \text{ is } \texttt{0x00} \end{cases}$
   For the first case, MAC computation will take significantly shorter than the latter case (since the oracle is in the first case tricked to remove 256 bytes expected to be the padding: $[FF]^{255}|255$. In the latter case only a single padding length byte is removed: 00).

Plaintext Recovery. As already discussed, the timing differences heavily depend on the plaintext. It is worth noting that according to the authors, significant timing differences could only be observed for a large amount of valid padding bytes. In the following, the notation of Subsection A.1.3 is used. The attack is based on the following values and algorithms: $l = 15$ (i.e. 16 bytes) and SHA-1 as HMAC algorithm (20 bytes HMAC value). Please note that explicit IVs are used, so C^{att} contains a preceeding IV block (*Record* header fields are not part of the encryption and C_0 is the IV).

[25] This trick is possible, because the 288 bytes of M_x hit a block boundary.

$$
\begin{aligned}
\textit{Chosen Bytes} :& X = [X_0, ...X_l] \\
\textit{Ciphertext} :& C^{att}(X) = RecordHeader|C_0|C_1|C_2|C_3 \oplus X|C_4 \\
\textit{Corresponding Plaintext} :& P^{att} = [P_1^{att}|P_2^{att}|P_3^{att}|P_4^{att}] \\
\textit{Interesting Plaintext Block} :& P_4^{att} = Dec_{Key}(C_4) \oplus (C_3 \oplus X) \\
&= P_4 \oplus X
\end{aligned}
\tag{5.12}
$$

The authors examined 4 different cases that may appear during decryption[26]:

1. P_4^{att} ending with 00 would lead to a single byte padding (only the single padding length byte) being removed, 20 bytes being used as MAC value and finally 43 remaining plaintext bytes R ($64 - 1 - 20 = 43$). Therefore, $SequenceNumber$ — $RecordHeader$ — R is used as input to the HMAC function ($5 + 8 + 43 = 56$ bytes input).

2. P_4^{att} ends with a valid padding of at minimum 01 (1 *real* padding byte and again a single byte padding length) or more which would lead to a minimum of two padding bytes being removed, again 20 bytes of MAC value leaving at least 42 remaining bytes R as plaintext. Thus, at minimum 55 bytes are used as input to the HMAC function ($5 + 8 + 42 = 55$ bytes input).

3. P_4^{att} ends with an invalid padding which leads to previously discussed *assume padding to be of zero length* case – no bytes are removed, 20 bytes MAC value and the remaining 44 bytes plaintext R. The MAC computation thus takes 57 bytes as input ($5 + 8 + 44 = 57$).

Because all cases consume a different amount of time to compute the HMAC, the authors can distinguish between the different cases by measuring time. If an attacker observes case 2, the last two bytes of the plaintext can be recovered by using $P_4^{att} = P_4 \oplus X$ since X was chosen by the attacker and is therefore known. The trick is to choose X wisely (in worst case this takes obviously 2^{16} trials). Once these bytes are known, an attacker can start plaintext recovery from right to left (starting with the padding bytes) according to Vaudenay's attack (cf. Subsection 5.2.8).

To adapt this attack to SSL/TLS the use of a multi-session attack is necessary (see Subsection 5.2.11), since SSL/TLS destroys the key used for encryption in case of `fatal` *Alerts* (which are definitely triggered during this attack). This limitation is of course not given in DTLS as discussed in Subsection 5.2.31. AlFardan and Paterson's attack can be enhanced even further when combining it with techniques known from B.E.A.S.T. (cf. Subsection 5.2.25).

[26] It is crucial to remember that P_4^{att}'s length is 64 bytes

5.2.37. RC4 - a Vulnerable Alternative

Category: ***Attacks on the Record Layer***
As a countermeasure to Padding Oracle Attacks (see Subsection 5.2.8 RC4 seemed, for a short period of time, to be a valid solution. Since RC4 is a stream-cipher the problems related to the mode of operation of block-ciphers (e.g. CBC) are not present. However, Fluhrer, Mantin and Shamir showed even in 2001 that RC4 has weaknesses [117]. In August 2013, Nadhem et. al found biases in the RC4 keystream [118]. These biases can lead to plaintext recovery of encrypted ciphertexts. The authors present two attack techniques that are only dependent on ciphertexts.

Single Byte Bias Attack. This attack focuses on the first 256 bytes of an RC4 ciphertext, where only 220 bytes contain interesting data (the first 36 bytes contain the encrypted `Finished` message of the handshake). Statistical biases are used to recover the plaintext with a high probability, depending on the number of available ciphertexts (2^{26} ciphertexts lead to a success rate of 50% for the first 40 bytes, where 2^{32} ciphertexts increase the probability to 96% for the first 220 bytes). The attack uses all known single-byte biases and a detailed analysis on the distribution of keystream bytes. It requires many ciphertexts (with equal plaintext encrypted with different keys).

Double Byte Bias Attack. As the first attack is limited to only the first 256 bytes, the double byte bias attack is able to decrypt bytes beyond this barrier, at any position and works independent from a plethora of plaintexts encrypted under *different* keys. Ciphertexts encrypted by the use of a *single* key are sufficient. Differing from the first attack, this one relies on multi-byte biases (consecutive bytes).

According to the authors, the attack is not practical (yet) as it requires a hughe amount of ciphertexts. Anyway, the authors discuss different countermeasures and finally conclude that the use of RC4 should be deprecated and avoided in TLS.

One month earlier, Isobe, Ohigashi, Watanabe and Morrii already found biases in keystream bytes of RC4 [119] and highlighted how these biases may be used to attack encryption. Isobe et al. did not apply their attack to SSL/TLS, but provided a computer simulation on the probability of plaintext recovery. Additionally, a summarizing list of all known biases was presented that can be very useful when implementing attacks on RC4.

5.2.38. Timing Based C.R.I.M.E. Adaptions

Category: ***Attacks on the Record Layer***
At Blackhat Europe 2013, Be'ery and Shulman presented an adapted version of the C.R.I.M.E. attack (see Subsection 5.2.34) called T.I.M.E. (*T*iming-*I*nfo leak *M*ade *E*asy) [120]. The authors made at least two improvements that eliminate the major drawbacks of the original C.R.I.M.E. attack: At first, they found a way to use C.R.I.M.E. also on HTTP responses, by using web sites that reflect users input in the response (such as search queries) and second

they make it unnecessary to analyze the server's encrypted response – control over the request plaintext is sufficient. Therefore, a timing side-channel is used. The basic idea is the following: compressed data is of reduced size compared to the original, uncompressed set of data. As a consequence the transmission is faster. This results in a measureable timing difference if the payload is specially constrcuted. Be'ery and Shulman utilized a special behaviour related to TCP to make the timing differences significant. Applying this to the original attack means that shorter transmission time suggests a successful guess, as the data could be compressed.

5.2.39. C.R.I.M.E. Enhancements

Category: ***Attacks on the Record Layer***
At Blackhat 2013, Prado, Harris and Gluck presented B.R.E.A.C.H. (*B*rowser *R*econnaissance & *E*xfiltration via *A*daptive *C*ompression of *H*ypertext) [121].

Basically, the authors outlined that prohibiting the use of compression by SSL/TLS does not solve the problem. The C.R.I.M.E. attack described in Subsection 5.2.34 is still applicable if any of the on-top protocols of SSL/TLS (such as e.g. HTTP) uses data compression. The attack does not present new findings, as all the ideas have already been demonstrated by Rizzo and Duong, but it highlights that the countermeasures against C.R.I.M.E. are not sufficient.

"Simplicity is the soul of efficiency."
Austin Freeman

6 It's T.I.M.E. for a New Framework

T.I.M.E. stands for *T*LS *I*nspection *M*ade *E*asy and represents a TLS stack with easy access to all TLS communication at any step during processing. Messages and parts of them can be analyzed, altered or suppressed on the fly – during the running communication. Therefore, everything belonging to the stack is accessible through objects. Altering is easily possible by using predefined getter and setter routines of the objects or predefined message builders or utilities. Additionally, if desired, message altering is even possible at bit level without the need for a complete source code understanding and extensive trial-and-error patching.

At the time of writing, the framework is limited to a client-side only position during communication, but designed to be extended with server capabilities in the near future.

6.1. Event Based Communication

For easy intervention, the whole communication is event based. As SSL/TLS dictates a predefined workflow on how connections are established (*Handshake Phase*) and used (*Application Data Phase*), T.I.M.E. offers options to register for events occuring during such a workflow. Once an event (hookpoint) is reached, the observers are notified. This scheme is implemented according to the Observer Design Pattern (see *Head First Design Patterns* [122]). The observable states depend on the handshake workflow and are the following for the class `TLS10HandshakeWorkflow`:

- `CLIENT_HELLO`
- `SERVER_HELLO`
- `SERVER_CERTIFICATE`

- SERVER_KEY_EXCHANGE
- SERVER_CERTIFICATE_REQUEST
- SERVER_HELLO_DONE
- CLIENT_CERTIFICATE
- CLIENT_KEY_EXCHANGE
- CLIENT_CERTIFICATE_VERIFY
- CLIENT_CHANGE_CIPHER_SPEC
- CLIENT_FINISHED
- SERVER_CHANGE_CIPHER_SPEC
- SERVER_FINISHED
- ALERT

Event notifications are either sent directly *after* a server message was received and decoded or *before* the corresponding message is sent so that changes on the message are possible.

Listing 6.1 gives an example how to register for the event that a `ClientHello` message is about to be sent and, as an example, how to modify the cipher suite list, as well as attaching optional TLS extensions to this message.

```
1  public class ObserverDemo implements Observer {
2      public final void start(final String host, final int port)
           throws SocketException {
3          AWorkflow workflow = new TLS10HandshakeWorkflow();
4          workflow.connectToTestServer(host, port);
5
6          // register for ClientHello pre-sent state
7          workflow.addObserver(this, EStates.CLIENT_HELLO);
8
9          //start workflow
10         workflow.start();
11     }
12
13     @Override
14     public final void update(final Observable o, final Object arg)
           {
15         MessageContainer trace = null;
16         TLS10HandshakeWorkflow.EStates states = null;
17         ObservableBridge obs;
18         if (o != null && o instanceof ObservableBridge) {
19             obs = (ObservableBridge) o;
20             states = (TLS10HandshakeWorkflow.EStates) obs.getState
                   ();
21             trace = (MessageContainer) arg;
22         }
23         if (states != null) {
24             switch (states) {
```

```
25                      case CLIENT_HELLO:
26                          ClientHello clientHello =
27                                  (ClientHello) trace.getCurrentRecord();
28                          // modify cipher suite list
29                          CipherSuites suites = new CipherSuites();
30                          suites.setSuites(new ECipherSuite[]{
31                           ECipherSuite.TLS_ECDH_RSA_WITH_AES_128_CBC_SHA
32                          });
33                          clientHello.setCipherSuites(suites);
34
35                          // add extensions to the message
36                          Extensions extensions = new Extensions();
37                          // ... create and add desired extensions ...
38                          clientHello.setExtensions(extensions);
39
40                          // set the modified message
41                          trace.setCurrentRecord(clientHello);
42                          break;
43                      default:
44                          break;
45                  }
46              }
47      }
48  }
```

Listing 6.1: T.I.M.E. demo code to alter `ClientHello` messages.

The major part of the listing is dedicated to message altering (lines 26-41), whereas creating, registering and starting the TLS handshake only requires 4 lines of code (3-10). Each class to be notified by a workflow (workflows must extend `de.rub.nds.ssl.stack.workflows.AWorkflow`) must implement the `java.util.Observer` interface and thus the `update` method (line 14). The interplay between the workflow and its caller is depicted in Figure 6.1.

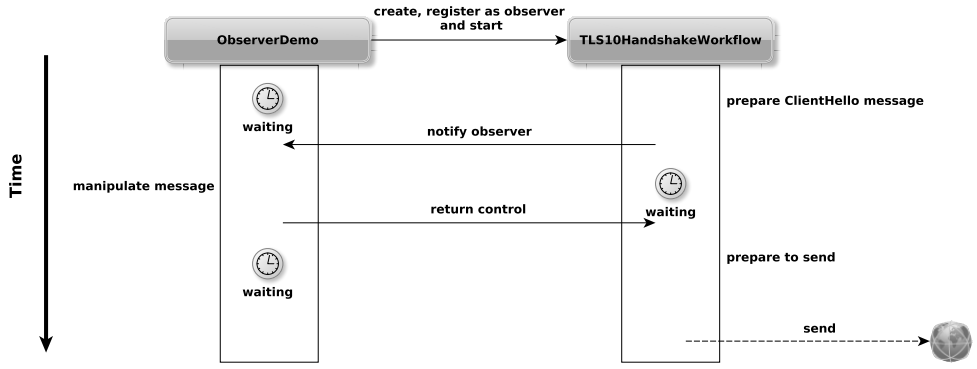

Figure 6.1.: Temporal interplay between the workflow and its caller.

6.2. Architecture

The most interesting part, when using the framework, are the classes inheriting from `de.rub.nds.ssl.stack.workflows.AWorkflow`. At the time of writ-

ing, only `TLS10HandshakeWorkflow` is available, defining a TLS 1.0 *Handshake Phase*. Figure 6.2 shows the Unified Modeling Language (UML) model of the base class. The components are discussed below.

AWorkflow A workflow represents the state machine for a communication. In a sample implementation (`TLS10HandshakeWorkflow`) a TLS 1.0 handshake phase is performed. The workflow is responsible for all actions during a communication, such as switching states, en- and decoding of messages, modification of internal parameters, error handling, calling cryptographic primitives, event processing and message handling. An overview of the methods of this class can be found in Table 6.1.

AResponseFetcher A repsonse fetcher is normally started in a separate `Thread` and listens for incoming data from the remote peer. The workflow registers as an `Observer` at the response fetcher and is notified by the whenever data is available. The preceeding 'A' in the class name signalizes that this class is `abstract`. The most common fetcher, the `StandardFetcher` listens on a given `Socket`. An overview of the methods of this class can be found in Table 6.2.

ObservableBridge This class is necessary to observe `WorkflowState`s. Because `WorkflowState` is just an interface, a bridging class is required for implementing the `Observable` interface. An overview of the methods of this class can be found in Table 6.3.

WorkflowState A `WorkflowState` represents the hook points in every workflow. Each state represents an event that is fired when the state is reached (e.g. when the `ClientHello` message is about to be sent or a `ServerHelloDone` message is received). By the use of this concept, classes can observe specific states of the workflow and get notified when the states are reached. The workflows itself rely on this event based architecture: If the workflow is waiting for messages from the counterpart the processing "sleeps" until messages are received. An overview of the methods of this class can be found in Table 6.4.

MessageContainer `MessageContainer`s are part of the message trace which records all communication in an object oriented representation. This facilitates monitoring/inspection and supersedes additional byte stream decoding (this has already been done). The container stores the current *Record*, the encoded `byte[]` representation of the current *Record*, the (old) *Record* that has been modified during processing (if modified at all), the timestamp and fragmentation information and at least keeps track of the previous and current `WorkflowState` An overview of the methods of this class can be found in Table 6.5.

ARecordFrame Inheriters of this class are the direct mappings of the *Records* defined in the SSL/TLS specification. Since each message is carried in a *Record*, this class is inherited by all implemented messages. An overview of the methods of this class can be found in Table 6.6. Please remember that these methods

Method	Description
`addObserver`	Add an observer for a specific workflow state.
`addToTraceList`	Add a new `MessageContainer` object to the `ArrayList`.
`countObservers`	Counts observers registered for a specific state.
`deleteObserver`	Delete an observer for a specific workflow state.
`deleteObservers`	Delete all observers for a specific workflow state.
`getCurrentState`	Get the current state of the handshake.
`getMainThread`	Get the main `Thread`.
`getResponseThread`	Get the response `Thread`.
`getTraceList`	Get the trace list of the whole handshake.
`hasChanged`	Tests if the changed flag of this state is set.
`nextState`	Switches to the next state.
`nextStateAndNotify`	Switches to the next state and notifies the observers.
`notifyCurrentObservers`	Notify observers of the current state and deliver the trace object.
`notifyObservers`	Notify observers and deliver the trace object.
`previousState`	Switches to the previous state.
`previousStateAndNotify`	Switches to the next state and notifies the observers.
`resetState`	Resets the internal state machine to its initial state.
`setCurrentState`	Set the current state of the handshake.
`setMainThread`	Set the main `Thread`.
`setRecordTrace`	Sets the current record of a trace and saves the previous one if present.
`setResponseThread`	Set the response `Thread`.
`start`	Start the workflow.
`switchToState`	Sets a new state and notifies the observers.
`wakeUp`	Get the `Thread` of the handshake workflow.

Table 6.1.: `AWorkflow` methods.

Method	Description
`isContinueFetching`	Checks if fetching bytes should be continued.
`stopFetching`	Stop fetching bytes from the Socket.

Table 6.2.: `AResponseFetcher` methods.

Method	Description
`getState`	Get the associated state.
`setChangedFlag`	Set the changed flag.

Table 6.3.: `ObservableBridge` methods.

Method	Description
`getID`	Get the ID of this state.

Table 6.4.: `WorkflowState` methods.

Method	Description
`getCurrentRecord`	Get the current record.
`getCurrentRecordBytes`	Get the current record bytes – if manually set.
`getOldRecord`	Get original record before manipulation.
`getPreviousState`	Get the previous state.
`getState`	Get the current state.
`getTimestamp`	Get the timestamp.
`isContinued`	Does this message belong to a multi message record.
`prepare`	Prepares the message (sets the encoded representation).
`setContinued`	Set if this message belongs to a multi message record.
`setCurrentRecord`	Set the current record.
`setCurrentRecordBytes`	Set current record bytes representation.
`setOldRecord`	Set original record before manipulation.
`setPreviousState`	Set the previous state.
`setState`	Set the current state.
`setTimestamp`	Set the timestamp.

Table 6.5.: `MessageContainer` methods.

are just the base functionality – each implementing class may add additional methods related to the particular message.

6.3. Connection Transcript

Each *Record* of a connection is added to a *Trace List* which is an `ArrayList` of `MessageContainer`s. This *Trace List* is a complete communication transcript and contains every sent, received and modified *Record*. If a *Record* is modified by an event observer the old and new values (the original *Record* and the modified *Record* which has at least been sent) and stored. This enables e.g. the option to replay a recorded session or simply offers an easy way to persist a

Method	Description
`decode`	Decodes a given byte array into a valid object if possible.
`encode`	Encodes this object.
`getContentType`	Get the content type of this record frame.
`getPayload`	Get the payload of this record frame.
`getProtocolVersion`	Get the protocol version of this record frame.
`setContentType`	Set the content type of this record frame.
`setPayload`	Set the payload of this record frame.
`setProtocolVersion`	Set the protocol version of this record frame.
`toString`	Builds the String representation of the current object.

Table 6.6.: `ARecordFrame` methods.

communication.

6.4. Network Connection

The connection is assumed to be network based and relies on the `Socket` class(es) of Java. This offers uncomplicated access to `Input-` and `OutputStream`s which represent the incoming and outgoing channels for communication with a connected target (e.g. a server). A major drawback of this setup is a blocking behaviour (depending on the `Socket`'s implementation). This means that the whole application waits for e.g. incoming data and cannot proceed until data is received or a read timeout occurs which enormously affects performance and responsiveness of the application. To prevent the workflow from being unresponsive, read operations are performed in a separate `Thread` as explained in the following Section. This of course requires the workflow to be aware of its own state – data required or proceed. The workflow can continue with processing as long as no data from the counterpart is required. If additional data is needed the workflow pauses itself until the necessary data is available.

6.5. Threaded Structure

To compensate the blocking behaviour caused by the chosen `Socket` architecture, a workflow is designed to act in a threaded environment. Basically, two different `Thread`s are involved in each communication at minimum.

- Main `Thread`
- Response `Thread`

Main Thread. This is normally the main `Thread` of the calling application that started the workflow. Every event (no matter if acting as the observer or

notifier) causes the workflow to be paused until event processing has finished. This prevents from race conditions related to event handling and guarantees that changes immediately effect the workflow.

Response Thread. This `Thread` is dedicated to fetching responses. Therefore, the `InputStream` of the connected `Socket` is permanently observed for incoming data. If bytes are received they are passed to the observers (in this case the workflow) and the execution of the workflow is paused until the bytes are processed.

6.6. T.I.M.E. Usecase: SSL Analyzer

On one hand, developed as a proof-of-concept of the T.I.M.E. stack, on the other hand a reference implementation for the fingerprinting ideas of Chapter 7 the SSL Analyzer evolved to be a unique usecase for the T.I.M.E. stack. The SSL Analyzer combines functionality for fingerprinting remote SSL/TLS stacks and vulnerability analysis. These features are, at the time of writing, unique – no other software tools are able to fingerprint stacks in this depth (application dependent down to patch levels) and comprehensive manner. At the moment, the SSL Analyzer is limited to server inspection only, but designed to be extended for additional client analysis.

6.6.1. Available Tools

Server Focused Tools. An example for SSL/TLS analysis targeting the server-side is SSLScan[1]. It can automatically detect the cipher suites supported by a server. This feature is also supported by the famous port-scanner *nmap*[2] when the *ssl-enum-ciphers* script[3] is used. Other online tools like the *COMODO SSL Analyzer*[4] or the *Qualys SSL Test*[5] can discover the cipher suites a server supports and report other interesting information about e.g. supported protocol features and the properties of the used certificates. *Qualys SSL Test* is even able to check whether a website is vulnerable against the B.E.A.S.T. [102] attack or supports insecure renegotiation. Other projects like the *EFF SSL Observatory*[6] maintain a history of certificates that have been used by a server, but do not analyze the SSL/TLS protocol itself. None of these tools or projects can detect the SSL/TLS stack of the server, except from taking the *HTTP Response Server Header* information (if included at all in a server's response). An exceptional tool is *SSL Audit*[7] by Thierry Zoller which sends malformed messages to a server and can, based on the response, distinguish between different SSL/TLS stacks, but not down to different version numbers of these stacks.

[1] `http://sourceforge.net/projects/sslscan/files/`
[2] `http://www.nmap.org`
[3] `http://nmap.org/nsedoc/scripts/ssl-enum-ciphers.html`
[4] `https://sslanalyzer.comodoca.com/`
[5] `https://www.ssllabs.com/ssltest/index.html`
[6] `https://www.eff.org/observatory/`
[7] `http://www.g-sec.lu/sslaudit/sslaudit.zip`

Client Focused Tools. *mod_ sslhaf*[8] is an Apache server module that can distinguish between different SSL/TLS clients, by analyzing properties of the `CLIENT_HELLO` message. *p0f*[9] – a widespread IP, TCP, and HTTP fingerprinting tool – supports SSL/TLS client fingerprinting, using the protocol version, cipher suites list, compression methods and supported extensions fields of the `ClientHello` message. The tool is able to distinguish between different versions of common web browsers, as long as the version change altered at least one of the properties of the `ClientHello` message. However, *p0f*[10] is strictly focused on client inspection and not focused on server-side software.

6.6.2. SSL Analyzer

The SSL Analyzer provides the user a graphical interface where the target(s) can be specified and the tests are selected. The tool is prepared for future enhancements, such as automated attacks or client-side inspection. Figure 6.4 shows a screenshot of the Graphical User Interface (GUI).

The detailed results are listed, after the scan has finished, in the *Results* tab of the GUI. The results show the final probabilities that the tested target runs one of the stacks contained in the database and additionally lists the whole communication process including messages sent and received. Figure 6.5 gives an example for such a scoring, where the version is determined down to patch level. The results also show that e.g. there are no significant changes to JSSE between Java 4.19 and 5.22.

For the uncomplicated creation of new stack fingerprints the tool includes an implementation fuzzer. The fuzzer executes all available tests and creates fingerprints of the behaviour for each test. The recorded information include:

- Test parameters (hashed)
- Communication trace
- Test case identifier
- Implementation identifier

This set of information forms the fingerprint of an implementation for a specific test. It is crucial to note that these fingerprints are not necessarily unique. If more than one implementation shows identical behaviour to a specific test, the fingerprint will be the same – except from the implementation identifier.

[8] `https://www.ssllabs.com/projects/client-fingerprinting/`
[9] `http://lcamtuf.coredump.cx/p0f3/`
[10] `https://idea.popcount.org/2012-06-17-ssl-fingerprinting-for-p0f/`

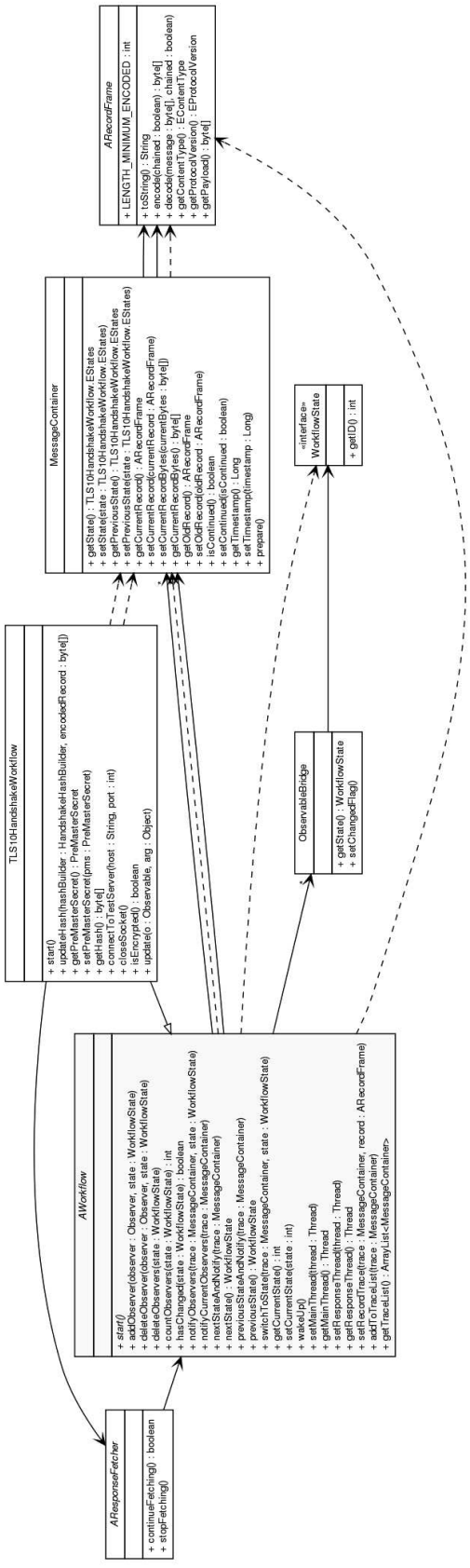

Figure 6.2.: UML model of `AWorkflow.java` and it's dependencies.

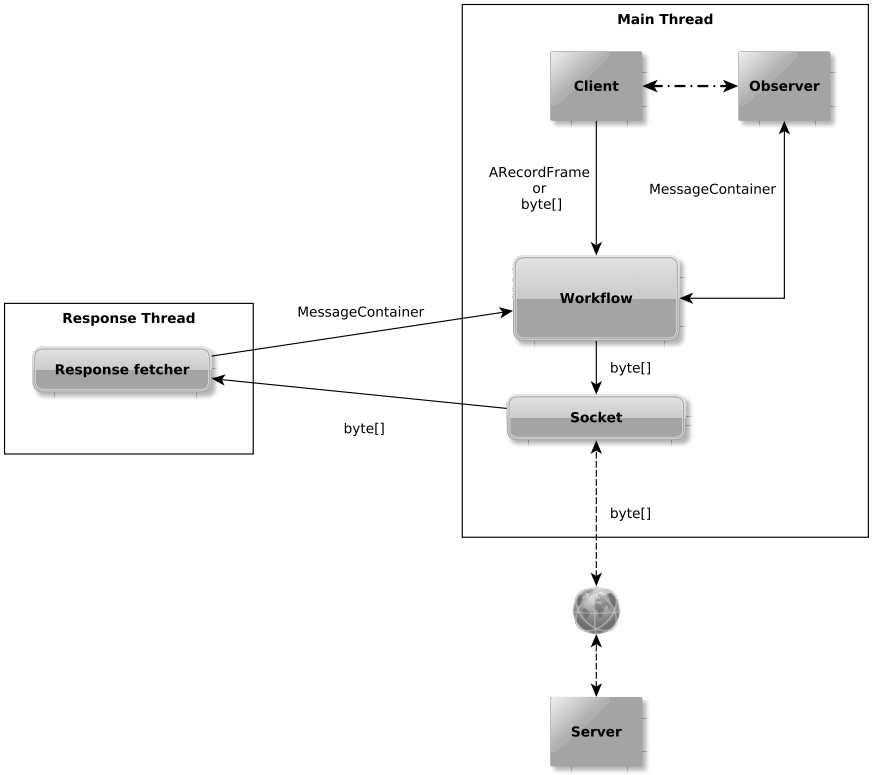

Figure 6.3.: Threaded interaction of T.I.M.E.

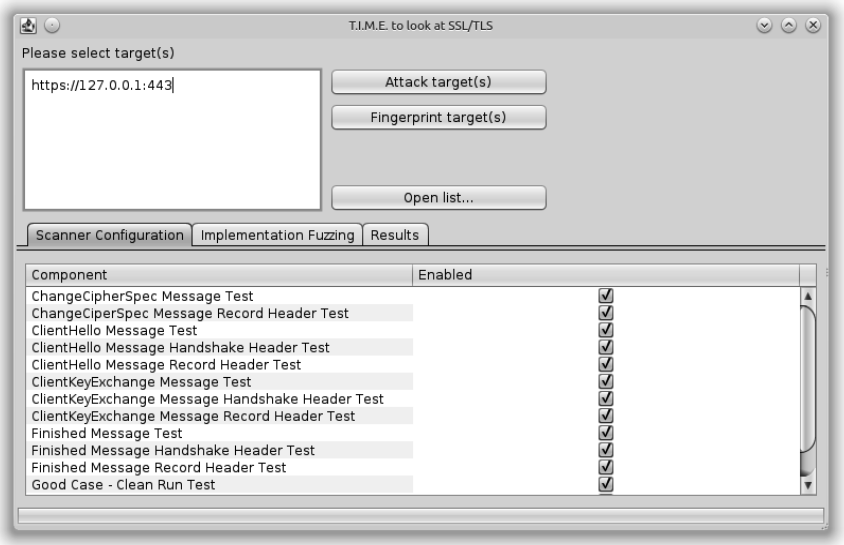

Figure 6.4.: GUI of the SSL Analyzer

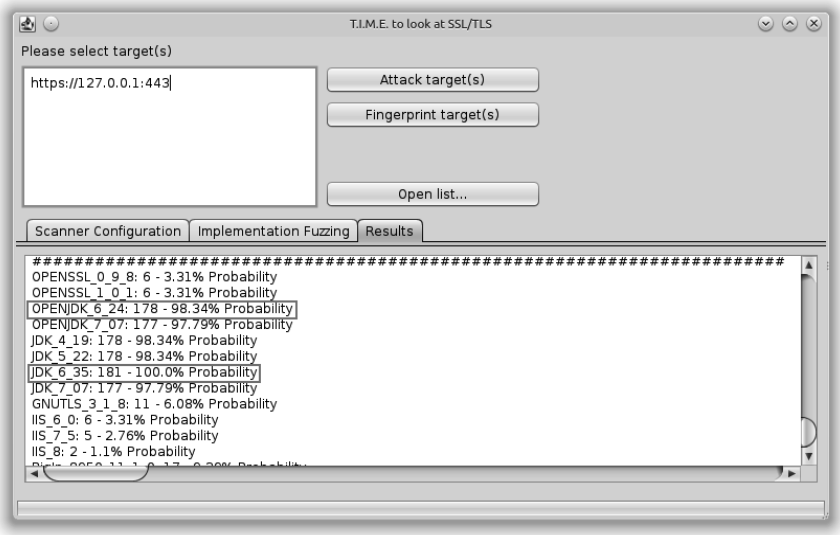

Figure 6.5.: SSL Analyzer results probability scoring

"Lance Armstrong bug: when the code never fails a test, but evidence shows it's not behaving as it should."

Unkown Origin

7 Fingerprinting SSL/TLS Stacks

The following Chapter is based on a joint work with Juraj Somorovsky, Eugen Weiss, Sebastian Schinzel, Jörg Schwenk and Erik Tews. At the time of writing the results are not published yet, but are about to be submitted to a conference.

7.1. Benefits of Fingerprinting

When it comes to the identification of vulnerable implementations it is often beneficial to collect information about the target. Especially the software being used is of significant importance. With detailed information about the vendor, patch level, enabled modules or functionality one can better prepare following steps. In contrast to other protocols, such as e.g. HTTP, SSL/TLS provides no standardized way to publish details about the used stack. This leverages the need for fingerprinting techniques to identify implementations and collect necessary information by using side channels. Fingerprinting identifies specific behaviour and links it to particular implementations.

7.2. Basic Technique

The SSL/TLS specifications do not define the deepest implementation details for implementing stacks. This leaves room for interpretation and on the other hand provides freedom to the developer. Moreover, some aspects in the specifications are optional, simply not implemented by stacks or let the developers choose between different options. These tiny differences cause different behaviour patterns. By analyzing these behaviour patterns it is possible to draw conclusions on the stack of a target. The guess is given by a probability value for a specific implementation. The value reflects the consistency of a detected behaviour with the reference behaviour of an associated implementation. The foundation of this value is a weighting method that weights unique behaviours

higher than behaviours that occur for more than a single implementation. For example if an implementation behaves in a particular test absolutely specific and distinguishable the resulting pattern is weighted higher than a behaviour shared among other implementations.

Fingerprinting a SSL/TLS library can be done actively, as well as strictly passively. While active fingerprinting requires to interact with a target and e.g. provoke special conditions where a specific behaviour of a particular library is known, passive fingerprinting is done without interaction only by eavesdropping a connection between a client and a server.

7.2.1. Passive Fingerprinting

When inspecting SSL/TLS *Records* the *Handshake Protocol* bares a flexibility that may or may not be used by implementations: The *Handshake Protocol* allows to include multiple messages in a single *Record*. A developer has the choice to either send multiple messages, e.g. `ServerHello`, `Certificate`, `ServerHelloDone`, separate or combined in a single *Record*. This behaviour allows to draw conclusions on the target implementation.

7.2.2. Active Fingerprinting

Active fingerprinting aims at detecting specific behaviour of clients that can be correlated to a particular library. A simple approach for fingerprinting is to open a connection to a server and evaluate the properties of the response messages retrieved during the handshake, such as e.g. supported cipher suites, compression algorithms, and TLS extensions.

7.3. Limitations

Of course, this technique depends on a predefined database with fingerprints linked to implementations. In addition, many implementations share nearly equal behaviour, only differing in a single test case. Furthermore, manual reconfiguration or patching changes the default behaviour leading to false patterns. In these cases the guess for an implementation may not be very accurate. For this reason a guess always refers only to a weighted scoring that the detected behaviour may belong to a specific implementation in the database. A guess of 100% for an implementation implies that a target behaves exactly like a reference implementation, fingerprinted before[1].

Additionally, many options can be configured when an SSL/TLS stack is used in an application and would not result in an accurate detection rate. Also minor updates may only fix problems triggered in error cases that will not occur during normal communication. As long as these error cases are not triggered or even worse cannot be triggered remotely and thus do not affect the stack's behaviour, these minor updates remain indistinguishable.

[1] But it is also possible taht Stack X was deliberately configured to emulate behaviour of Stack Y as exactly as possible.

7.4. Previous Work

The previous work done in this area can be grouped into two different categories:

- Fingerprinting of Clients
- Fingerprinting of Servers

7.4.1. Analyzing SSL/TLS Servers

An example for a tool capable of identifying SSL/TLS servers is `SSLScan`[2], which can automatically detect the cipher suites supported by an SSL/TLS server. Additionally, the famous port-scanner `nmap`[3] also supports cipher suite detectinon using the `ssl-enum-ciphers` script[4]. Online tools like the `COMODO SSL Analyzer`[5] or the `Qualys SSL Test`[6] can discover the cipher suites supported by a SSL/TLS server and report other interesting information about supported protocol features and the properties of the used certificates. `Qualys SSL Test` is even able to check for vulnerablities against B.E.A.S.T. (see Subsection 5.2.25), C.R.I.M.E (see Subsection 5.2.34) or RC4 biases (see Subsection 5.2.37) or insecure renegotiation (see Subsection 5.2.19). Other projects like the `EFF SSL Observatory`[7] maintain a history of certificates that have been used by a server, but do not analyze the SSL/TLS protocol itself. None of these tools can detect the SSL/TLS stack running on a server, except from showing the *HTTP Response Server Header* (if included at all). An exceptional tool is `SSL Audit`[8] by Thierry Zoller which sends malformed messages to a server and distinguishes between different SSL/TLS stacks. But at least, the tool cannot differentiate between different version numbers of these stacks.

7.4.2. Analyzing SSL/TLS Clients

More advances have been made when it comes to SSL/TLS client-side fingerprinting. `mod_sslhaf`[9] is an Apache server module that can distinguish between different SSL/TLS clients, by analyzing properties of the `CLIENT_HELLO` message. Finally, `p0f`[10] - a famous IP, TCP, and HTTP fingerprinting tool - has been extended to support SSL/TLS client fingerprinting, using the protocol version, cipher suites list, compression methods, and supported extensions fields in the `CLIENT_HELLO` message. By using this method, `p0f` is able to distinguish between different versions of common web browsers, as long as the version change altered at least one of the properties of the `CLIENT_HELLO` message. However, *p0f* cannot perform any kind of fingerprinting of SSL/TLS server stacks[11].

[2] `http://sourceforge.net/projects/sslscan/files/`
[3] `http://www.nmap.org`
[4] `http://nmap.org/nsedoc/scripts/ssl-enum-ciphers.html`
[5] `https://sslanalyzer.comodoca.com/`
[6] `https://www.ssllabs.com/ssltest/index.html`
[7] `https://www.eff.org/observatory/`
[8] `http://www.g-sec.lu/sslaudit/sslaudit.zip`
[9] `https://www.ssllabs.com/projects/client-fingerprinting/`
[10] `http://lcamtuf.coredump.cx/p0f3/`
[11] `https://idea.popcount.org/2012-06-17-ssl-fingerprinting-for-p0f/`

7.5. Details

To quickly identify SSL/TLS stacks a fingerprinting analyzer based on *T.I.M.E.* (see Chapter 6 was developed. To identify stacks down to a minor version number or patch level, it is necessary to perform an active fingerprinting method that provokes special conditions. Some of the resulting behaviours cannot be altered using configuration changes and are thus considered as being implementation specific. The behaviour characteristics are compared to patterns of particular stacks contained in a reference database.
NOTE: The fingerprinting analyzer is not yet capable of checking all listed criterias listed below or some of them are not implemented in the presented depth. A main goal for future releases of the analysis module is to have full and unlimited support of all criterias listed below.

7.5.1. Error Messages

Error messages are a very useful information when determining the target's SSL/TLS stack. By source code review one is able to clearly identify all possible occurences of error messages and the particular type. This information can be very stack specific so that stack estimation is possible. Table 7.1 lists all possible error messages that were found during research so far.

Message	OpenSSL 1.0.1c	GnuTLS 3.2.3	JSSE 7 (b147)	Schannel (IIS 6)
BAD_CERTIFICATE	x		x	
BAD_CERTIFICATE_STATUS_RESPONSE	x			
BAD_RECORD_MAC	x	x	x	
CERTIFICATE_EXPIRED	x			
CERTIFICATE_REVOKED	x			
CERTIFICATE_UNKNOWN	x		x	
CLOSE_NOTIFY	x	x	x	x
DECOMPRESSION_FAILURE	x			
DECODE_ERROR	x			
DECRYPT_ERROR	x			
DECRYPTION_FAILED	x			
EXPORT_RESTRICTION	x			
HANDSHAKE_FAILURE	x	x	x	
ILLEGAL_PARAMETER	x		x	
INTERNAL_ERROR	x	x	x	
INSUFFICIENT_SECURITY		x		
NO_CERTIFICATE	x	x	x	
NO_RENEGOTIATION	x		x	
PROTOCOL_VERSION	x	x		
RECORD_OVERFLOW	x	x		
UNEXPECTED_MESSAGE	x	x	x	
UNKNWON_PSK_IDENTITY	x			
UNKNOWN_CA	x			
UNKNOWN_CERTIFICATE			x	
UNRECOGNIZED_NAME	x			
UNSUPPORTED_CERTIFICATE	x			
UNSUPPORTED_EXTENSION			x	

Table 7.1.: Error messages sent by OpenSSL,GnuTLS, JSSE and Schannel

The error messages can be used to build a list of possible stack candidates by exclusion. If an implementation is knwon to send different error messages

in the test scenarios or does not send the received error message at all it can safely be removed from the list of candidates. Of course the behaviour can be configuration dependent (e.g. in dependence of specific cipher suites), but in most cases error message analysis revealed to be an implementation specific criteria that is hard to influence when tests concerning message structure are executed (such as e.g. tampering with the *Record* header fields or specially crafted invalid messages). Please note that the results for GnuTLS are not reliable[12].
A special case concerning error message analysis is related to Microsoft Schannel which sends no error messages at all. Errors are only logged at target's reporting console, but not sent back to the client. Instead, the connection is immediately terminated.

7.5.2. Cipher Suite Favoritism

An additional source of useful information for fingerprinting is introduced by the server sided choice for a specific cipher suite. An active fingerprinting engine can observe the stack's default choice for a cipher suite when presented with a particular subset of cipher suites and draw conclusions on the used library. It is crucial noticing that this criteria for or against a particular stack highly depends on unchanged default configurations. Otherwise, the analysis will lead to false results which outlines a major drawback of this criteria: it can be influenced by configuration and thus be used to emulate behaviour of a different stack. On the other hand, while reducing the number of supported cipher suites can easily be done by configuration, new cipher suites cannot be added to an SSL/TLS stack without changing the implementation. This implies that the presence of a certain cipher suite is an exclusion criteria for all stacks that do not support this cipher suite. To be absolutely accurate, it is also necessary to check if all offered cipher suites are really implemented by opening a session for each suite. After a successful handshake one can be sure that the cipher suite is implemented.

7.5.3. Invalid/Modified Message Tolerance

The deliberate modification of message fields with different valid or even invalid values provides another way to analyze behaviour of implementations. Header information of a *Record* like message type, length or protocol version are manipulated to provoke a particular behaviour. Furthermore, the message content itself can be used to detect specific behaviour, as for example the payload of a `ClientKeyExchange` message can be modified to a specifically destroyed PKCS#1 format to check for possible vulnerabilities to Bleichenbacher's attack (see Subsection 5.2.7).

[12] As the source code analysis of GnuTLS is extremely difficult, only those *Alerts* are marked which occured during testing or during code review. It is very likely, that other errors are send as well!

7.5.4. Extension Support and Behaviour to Specially Crafted Extension Messages

Extension support and the behaviour to modified or even invalidated extension messages or data fields may also be used to identify implementations. There are currently 21 extensions of a bride variety, reaching from server name indication to secure renegotiation support, defined for TLS[13], but according to a study of Amann et al. [123] only few of them are used *in the wild*. This may on the one hand be related to the lack for usage scenarios, but on the other hand may be caused by missing implementations. An updated list of a selection of the most important extensions and stacks supporting these can be found at Wikipedia[14] .

7.5.5. Timings

Fingerprints can also be based on timing differences during message or error processing. Timing measurement depends on hardware as well as software configurations and is thus highly volatile. During experiments the timing based fingerprinting revealed not to be accurate enough to output reliable results. Caching, Garbage Collection, Process Scheduling as well as system load and a plethora of other factors were found to have significant influence on this criteria. At the time of writing, this criteria is not considered to be reliable.

7.6. Fingerprinting Results

The following results highlight specific behaviour of different stacks suitable for the creation of fingerprints. For evaluation purpose all tests are executed negotiating TLS 1.0 in the *Handshake Phase*.

7.6.1. Tampering with RecordHeader Fields

Every SSL/TLS *Record* contains at least 3 plaintext header fields:

- Content Type
- Protocol Version
- Payload Length

Different stacks behave differently when confronted with invalid or modified header fields. The responses or connection resets may reveal useful information when trying to determine the stack used by a target. For example the TLS specification defines a dedicated *Alert* – `PROTOCOL_VERSION` – for incorrect values of the protocol version field of the handshake messages. Differing error messages or connection resets are not specification compliant.

[13] `https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml`

[14] `https://en.wikipedia.org/wiki/Comparison_of_TLS_implementations#Extensions`

The fingerprinting analyzer performs for the most common handshake messages *Record* header field tampering with the following modifications:

- Wrong Content Type (`17`)
- Protocol Version set to (`ff ff`)
- Payload length overflow (`00 00`)
- Payload length underflow (`ff ff`)

Wrong Content Type

`ClientHello` The results of this test case are listed in Table 7.2.

Stack	Reaction	Error Message
OpenSSL 0.9.8	Connection reset	
OpenSSL 1.0.1	Connection reset	
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Connection reset	
BigIp 8950 11.3.0	Connection reset	

Table 7.2.: Content type set to `17` in the `ClientHello` message

No obvious abnormalities were found. The detected behaviour is acceptable, although connection resets represent unexpected behaviour.

`ClientKeyExchange` The results of this test case are listed in Table 7.3.

An interesting change in behaviour is identified for the BigIP devices which show different *Alert* messages depending on the software version.

`ChangeCipherSpec` The results of this test case are listed in Table 7.4.

Again, a major change in the implementation can be seen for the BigIP devices. Both observed behaviours (`fatal` *Alert* and Connection reset) are suitable for the given scenario.

`Finished` The results of this test case are listed in Table 7.5.

The `BAD_RECORD_MAC` *Alerts* are not surprising, since the `Finished` message is the first protected (encrypted and integrity ensured) message during the

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenSSL 1.0.1	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`BAD_RECORD_MAC`

Table 7.3.: Content type set to `17` in the `ClientKeyExchange` message

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenSSL 1.0.1	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Connection reset	
BigIp 8950 11.3.0	`fatal` *Alert*	`UNEXPECTED_MESSAGE`

Table 7.4.: Content type set to `17` in the `ChangeCipherSpec` message

communication. As a modification clearly changes the message, the unmodified MAC is violated. Again the implementation of the BigIP devices shows significant different behaviour.

Wrong Protocol Version

`ClientHello` The results of this test case are listed in Table 7.6.

Only GnuTLS seems to respond with the expected `Alert` message, whereas ignoring the invalid protocol version is definitely not recommended. A very interesting behaviour is observed for the 11.1.0 software release of the BigIP devices which repsond with a completely wrong protocol version, the problem seems to be fixed in the newer software release.

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenSSL 1.0.1	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenJDK 6.24	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenJDK 7.07	`fatal` *Alert*	`BAD_RECORD_MAC`
Oracle JDK 4.19	`fatal` *Alert*	`BAD_RECORD_MAC`
Oracle JDK 5.22	`fatal` *Alert*	`BAD_RECORD_MAC`
Oracle JDK 6.35	`fatal` *Alert*	`BAD_RECORD_MAC`
Oracle JDK 7.07	`fatal` *Alert*	`BAD_RECORD_MAC`
GnuTLS 3.1.8	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`BAD_RECORD_MAC`

Table 7.5.: Content type set to `17` in the `Finished` message

Stack	Reaction	Error Message
OpenSSL 0.9.8	Connection reset	
OpenSSL 1.0.1	Connection reset	
OpenJDK 6.24	Ignored	
OpenJDK 7.07	Ignored	
Oracle JDK 4.19	Ignored	
Oracle JDK 5.22	Ignored	
Oracle JDK 6.35	Ignored	
Oracle JDK 7.07	Ignored	
GnuTLS 3.1.8	`fatal` *Alert*	`PROTOCOL_VERSION`
Microsoft Schannel (IIS 6.0)	Ignored	
Microsoft Schannel (IIS 7.5)	Ignored	
BigIp 8950 11.1.0	Connection reset	(Response Protocol Version 00 00)
BigIp 8950 11.3.0	Connection reset	

Table 7.6.: Protocol version set to `ff ff` in the `ClientHello` message

`ClientKeyExchange` The results of this test case are listed in Table 7.7.

OpenSSLand GnuTLS seem to be the only implementations that check the protocol version included in the `ClientKeyExchange` message against the protocol version of the containing *Record*, thus lead to a protocol version mismatch. Very interesting about the results is that the BigIP device seems to mirror the invalid protocol version and finally resets the connection.

`ChangeCipherSpec` The results of this test case are listed in Table 7.8.

The results are completely similar to those for the `ClientKeyExchange` message, no new findings can be gained.

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`PROTOCOL_VERSION`
OpenSSL 1.0.1	`fatal` *Alert*	`PROTOCOL_VERSION`
OpenJDK 6.24	Ignored	
OpenJDK 7.07	Ignored	
Oracle JDK 4.19	Ignored	
Oracle JDK 5.22	Ignored	
Oracle JDK 6.35	Ignored	
Oracle JDK 7.07	Ignored	
GnuTLS 3.1.8	`fatal` *Alert*	`PROTOCOL_VERSION`
Microsoft Schannel (IIS 6.0)	Ignored	
Microsoft Schannel (IIS 7.5)	Ignored	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	Connection reset	(Response Protocol Version ff ff)

Table 7.7.: Protocol version set to `ff ff` in the `ClientKeyExchange` message

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`PROTOCOL_VERSION`
OpenSSL 1.0.1	`fatal` *Alert*	`PROTOCOL_VERSION`
OpenJDK 6.24	Ignored	
OpenJDK 7.07	Ignored	
Oracle JDK 4.19	Ignored	
Oracle JDK 5.22	Ignored	
Oracle JDK 6.35	Ignored	
Oracle JDK 7.07	Ignored	
GnuTLS 3.1.8	`fatal` *Alert*	`PROTOCOL_VERSION`
Microsoft Schannel (IIS 6.0)	Ignored	
Microsoft Schannel (IIS 7.5)	Ignored	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	Connection reset	(Response Protocol Version ff ff)

Table 7.8.: Protocol version set to `ff ff` in the `ChangeCipherSpec` message

`Finished` The results of this test case are listed in Table 7.9.

The results are completely similar to those for the `ClientKeyExchange` message, no new findings can be gained.

Payload Overflow

`ClientHello` The results of this test case are listed in Table 7.10.

Connection resets and `fatal` *Alert*s are the expected behaviour. Once again the behaviour of the BigIP devices is special. Software version 11.3.0 responds with an illegal protocol version, whereas release 11.1.0 simply ignores the wrong *Record* length.

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`PROTOCOL_VERSION`
OpenSSL 1.0.1	`fatal` *Alert*	`PROTOCOL_VERSION`
OpenJDK 6.24	Ignored	
OpenJDK 7.07	Ignored	
Oracle JDK 4.19	Ignored	
Oracle JDK 5.22	Ignored	
Oracle JDK 6.35	Ignored	
Oracle JDK 7.07	Ignored	
GnuTLS 3.1.8	`fatal` *Alert*	`PROTOCOL_VERSION`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	Connection reset	(Response Protocol Version ff ff)

Table 7.9.: Protocol version set to `ff ff` in the `Finished` message

Stack	Reaction	Error Message
OpenSSL 0.9.8	Connection reset	
OpenSSL 1.0.1	Connection reset	
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Ignored	
BigIp 8950 11.3.0	Connection reset	(Response Protocol Version 00 00)

Table 7.10.: Payload length set to `00 00` in the `ClientHello` message

`ClientKeyExchange` The results of this test case are listed in Table 7.11.

The error message of the OpenSSL implementations is interesting, since the protcol version field was not changed during this test. It is very likely that the error message is caused by the fact that the *Record* header and the *PreMaster-Secret* differ. The BigIP 8950 device software version 11.3.0 seems to suffer from the same bug as observed for the `ClientHello` message.

`ChangeCipherSpec` The results of this test case are listed in Table 7.12.

Except form the BigIP device 11.3.0 all implementations behave like in the case before (`ClientKeyExchange`). The BigIP device recognizes the wrong

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`PROTOCOL_VERSION`
OpenSSL 1.0.1	`fatal` *Alert*	`PROTOCOL_VERSION`
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	Connection reset	(Response Protocol Version 00 01)

Table 7.11.: Payload length set to `00 00` in the `ClientKeyExchange` message

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`PROTOCOL_VERSION`
OpenSSL 1.0.1	`fatal` *Alert*	`PROTOCOL_VERSION`
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`RECORD_OVERFLOW`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`RECORD_OVERFLOW`

Table 7.12.: Payload length set to `00 00` in the `ChangeCipherSpec` message

length. This suggests that the bug seen with the `ClientKeyExchange` message is related to the handshake header processing[15].

`Finished` The results of this test case are listed in Table 7.13.

The behaviour is a valid reaction to the test case. The `DECRYPTION_FAILED` *Alert* is related to the fact that the `Finished` message is the first encrypted message during the handshake. The error message is likely to be caused by the failing decryption process. At first sight, one would expect to see `fatal` *Alerts* of type `RECORD_OVERFLOW`.

[15]The `ChangeCipherSpec` message is not part of the *Handshake Protocol*!

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`DECRYPTION_FAILED`
OpenSSL 1.0.1	`fatal` *Alert*	`DECRYPTION_FAILED`
OpenJDK 6.24	`fatal` *Alert*	`HANDSHAKE_FAILURE`
OpenJDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 4.19	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 5.22	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 6.35	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
GnuTLS 3.1.8	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`DECRYPTION_FAILED`

Table 7.13.: Payload length set to `00 00` in the `Finished` message

Payload Underflow

`ClientHello` The results of this test case are listed in Table 7.14.

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`RECORD_OVERFLOW`
OpenSSL 1.0.1	`fatal` *Alert*	`RECORD_OVERFLOW`
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`RECORD_OVERFLOW`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Connection reset	
BigIp 8950 11.3.0	`fatal` *Alert*	`RECORD_OVERFLOW`

Table 7.14.: Payload length set to `ff ff` in the `ClientHello` message

It is surprising that only OpenSSL, GnuTLS and a BigIP device respond with an error message related to the particular error case. This could either mean that all other implementations perform no length check on the payload at all (the SSL/TLS specifications clearly define the maximum length of a *Record* to be 2^{14}), simply return a more general error or could not correctly decode the *Record*.

`ClientKeyExchange` The results of this test case are listed in Table 7.15.

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`RECORD_OVERFLOW`
OpenSSL 1.0.1	`fatal` *Alert*	`RECORD_OVERFLOW`
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`RECORD_OVERFLOW`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`RECORD_OVERFLOW`

Table 7.15.: Payload length set to `ff ff` in the `ClientKeyExchange` message

Except from the BigIP 8950 11.1.0 device the behaviour is equal compared to the same test case with the `ClientHello` message.

`ChangeCipherSpec` The results of this test case are listed in Table 7.16.

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`RECORD_OVERFLOW`
OpenSSL 1.0.1	`fatal` *Alert*	`RECORD_OVERFLOW`
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`RECORD_OVERFLOW`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`RECORD_OVERFLOW`

Table 7.16.: Payload length set to `ff ff` in the `ChangeCipherSpec` message

The behaviour is equal to the one of the `ClientKeyExchange` message.

`Finished` The results of this test case are listed in Table 7.17.

Again, the behaviour does not change, compared to the `ClientKeyExchange` message case.

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`RECORD_OVERFLOW`
OpenSSL 1.0.1	`fatal` *Alert*	`RECORD_OVERFLOW`
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`RECORD_OVERFLOW`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`RECORD_OVERFLOW`

Table 7.17.: Payload length set to `ff ff` in the `Finished` message

7.6.2. Tampering with Handshake Header Fields

Every SSL/TLS handshake message contains (additionally to the 3 mandatory plaintext headers of the `Record`) 2 fields:

- Message Type
- Payload Length

Again, differing behaviour of stacks was found.
To provoke the different behaviours the following handshake header fields are modified with the following changes:

- Wrong Message Type (`ff`)
- Payload length overflow (`00 00`)
- Payload length underflow (`ff ff`

Wrong Message Type

`ClientHello` The results of this test case are listed in Table 7.18.
Both behaviours seem to be valid at this point with the connection reset being the more security related option while returning an error message holds the convenient point of view.

`ClientKeyExchange` The results of this test case are listed in Table 7.19.
In this test case, no specific abnormalities were found.

`Finished` The results of this test case are listed in Table 7.20.
Again, in this test case no specific abnormalities were found.

Stack	Reaction	Error Message
OpenSSL 0.9.8	Connection reset	
OpenSSL 1.0.1	Connection reset	
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Connection reset	
BigIp 8950 11.3.0	Connection reset	

Table 7.18.: Message type set to `ff` in the `ClientHello` message

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenSSL 1.0.1	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`UNEXPECTED_MESSAGE`

Table 7.19.: Message type set to `ff` in the `ClientKeyExchange` message

Payload Length Overflow

`ClientHello` The results of this test case are listed in Table 7.21.

Interesting on this result are the `PROTOCOL_VERSION` *Alerts* (since the protocol version was valid). All *Alerts* are likely to be related to decoding errors. The BigIp device with software release 11.1.0 behaves as seen before in the *Record* header test cases.

`ClientKeyExchange` The results of this test case are listed in Table 7.22.

The decoding of the `ClientKeyExchange` message seems to be completely independent from the handshake header of the message in this case (and at least different, compared to the `ClientHello` processing). This is interesting, because it leaves an open question: How is a *Record*, containing multiple hand-

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenSSL 1.0.1	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenJDK 6.24	`fatal` *Alert*	`HANDSHAKE_FAILURE`
OpenJDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 4.19	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 5.22	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 6.35	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
GnuTLS 3.1.8	`fatal` *Alert*	`BAD_RECORD_MAC`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`BAD_RECORD_MAC`

Table 7.20.: Message type set to `ff` in the `Finished` message

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`PROTOCOL_VERSION`
OpenSSL 1.0.1	`fatal` *Alert*	`PROTOCOL_VERSION`
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`RECORD_OVERFLOW`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Connection reset	(Response Protocol Version 00 00)
BigIp 8950 11.3.0	`fatal` *Alert*	`PROTOCOL_VERSION`

Table 7.21.: Payload length set to `00 00` in the `ClientHello` message

shake messages (handshake message stapling) correctly decoded?

`Finished` The results of this test case are listed in Table 7.23.

In contrast to the test case before, the error message for Java implementations is different. A possible explanation is that the `HANDSHAKE_FAILURE` *Alerts* are a general error message and the real cause is maybe related to failing decryption processing or MAC verification.

Payload Length Underflow

`ClientHello` The results of this test case are listed in Table 7.24.

Stack	Reaction	Error Message
OpenSSL 0.9.8	Connection reset	
OpenSSL 1.0.1	Connection reset	
OpenJDK 6.24	Ignored	
OpenJDK 7.07	Ignored	
Oracle JDK 4.19	Ignored	
Oracle JDK 5.22	Ignored	
Oracle JDK 6.35	Ignored	
Oracle JDK 7.07	Ignored	
GnuTLS 3.1.8	`fatal` *Alert*	`RECORD_OVERFLOW`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Connection reset	
BigIp 8950 11.3.0	`fatal` *Alert*	`ILLEGAL_PARAMETER`

Table 7.22.: Payload length set to `00 00` in the `ClientHello` message

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenSSL 1.0.1	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenJDK 6.24	`fatal` *Alert*	`HANDSHAKE_FAILURE`
OpenJDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 4.19	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 5.22	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 6.35	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
GnuTLS 3.1.8	`fatal` *Alert*	`BAD_RECORD_MAC`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`BAD_RECORD_MAC`

Table 7.23.: Payload length set to `00 00` in the `ClientHello` message

The `ILLEGAL_PARAMETER` *Alerts* are conspicuous, but likely to be caused by incorrect message decoding. The BigIP 8950 version 11.1.0 device behaves as seen before in an very unexpected way.

`ClientKeyExchange` The results of this test case are listed in Table 7.25.

Again, the `ILLEGAL_PARAMETER` *Alerts* are conspicuous, but likely to be caused by incorrect message decoding.

`Finished` The results of this test case are listed in Table 7.26.

This test case revealed no significant behaviours.

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`ILLEGAL_PARAMETER`
OpenSSL 1.0.1	`fatal` *Alert*	`ILLEGAL_PARAMETER`
OpenJDK 6.24	Connection reset	
OpenJDK 7.07	Connection reset	
Oracle JDK 4.19	Connection reset	
Oracle JDK 5.22	Connection reset	
Oracle JDK 6.35	Connection reset	
Oracle JDK 7.07	Connection reset	
GnuTLS 3.1.8	Connection reset	
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Connection reset	(Response Protocol Version 00 00)
BigIp 8950 11.3.0	`fatal` *Alert*	`ILLEGAL_PARAMETER`

Table 7.24.: Payload length set to `ff ff` in the `ClientHello` message

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`ILLEGAL_PARAMETER`
OpenSSL 1.0.1	`fatal` *Alert*	`ILLEGAL_PARAMETER`
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`ILLEGAL_PARAMETER`

Table 7.25.: Payload length set to `ff ff` in the `ClientKeyExchange` message

7.6.3. Message Specific Tests

Some of the performed tests change parameters that are only available in specific messages. Therefore, in the following the test results for each particular message are discussed.

`ClientHello`

Invalid Protocol Version `ff ff` The protocol version number contained in the `ClientHello` message was altered to be different from the one contained in the *Record* header fields. The results of this test case are listed in Table 7.27.

The results are very diverse. Slight modifications during releases can be seen

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenSSL 1.0.1	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenJDK 6.24	`fatal` *Alert*	`HANDSHAKE_FAILURE`
OpenJDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 4.19	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 5.22	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 6.35	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
GnuTLS 3.1.8	`fatal` *Alert*	`BAD_RECORD_MAC`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`BAD_RECORD_MAC`

Table 7.26.: Payload length set to `ff ff` in the `Finished` message

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenSSL 1.0.1	`fatal` *Alert*	`PROTOCOL_VERSION`
OpenJDK 6.24	`fatal` *Alert*	`HANDSHAKE_FAILURE`
OpenJDK 7.07	`fatal` *Alert*	`INTERNAL_ERROR`
Oracle JDK 4.19	Ignored	
Oracle JDK 5.22	Ignored	
Oracle JDK 6.35	Ignored	
Oracle JDK 7.07	`fatal` *Alert*	`INTERNAL_ERROR`
GnuTLS 3.1.8	`fatal` *Alert*	`PROTOCOL_VERSION`
Microsoft Schannel (IIS 6.0)	Ignored	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	Connection reset	

Table 7.27.: Protocol version set to `ff ff` in the `ClientHello` message

highlighting stack modifications.

Invalid Protocol Version `00 00` At first it seemed sufficient to check the behaviour of the target stack to an arbitrary modified protocol number contained in the `ClientHello` message, but after multiple checks some stacks revealed to show behaviour based on the particular version number. The results of this test case are listed in Table 7.28.

As can be seen, the behaviour of Schannel (IIS 6.0) and the different JDK versions changed when a 00 00 protcol version was sent.

Invalid Protocol Version `03 00` **(SSLv3)** The results of this test case are listed in Table 7.29.

Stack	Reaction	Error Message
OpenSSL 0.9.8	Connection reset	
OpenSSL 1.0.1	Connection reset	
OpenJDK 6.24	`fatal` *Alert*	`HANDSHAKE_FAILURE`
OpenJDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 4.19	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 5.22	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 6.35	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
GnuTLS 3.1.8	`fatal` *Alert*	`PROTOCOL_VERSION`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Connection reset	(Response Protocol Version 00 00)
BigIp 8950 11.3.0	Connection reset	

Table 7.28.: Protocol version set to `00 00` in the `ClientHello` message

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`HANDSHAKE_FAILURE`
OpenSSL 1.0.1	`fatal` *Alert*	`HANDSHAKE_FAILURE`
OpenJDK 6.24	`fatal` *Alert*	`HANDSHAKE_FAILURE`
OpenJDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 4.19	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 5.22	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 6.35	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
GnuTLS 3.1.8	`fatal` *Alert*	`PROTOCOL_VERSION`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`

Table 7.29.: Protocol version set to `03 00` in the `ClientHello` message

All implementations recognize the, in this case, invalid protocol version (only TLS cipher suites were offered and additionally the protocol version in the *Record* header differs) and react in an adequate way.

Invalid Protocol Version `03 03` (TLS 1.2) The results of this test case are listed in Table 7.30.

Not all implementations recognize the invalid protocol version (differs from the one in the record header[16]) and react in an adequate way.

[16]The specification advices to use the highest available protocol version.

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenSSL 1.0.1	`fatal` *Alert*	`PROTOCOL_VERSION`
OpenJDK 6.24	`fatal` *Alert*	`HANDSHAKE_FAILURE`
OpenJDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 4.19	Ignored	
Oracle JDK 5.22	Ignored	
Oracle JDK 6.35	Ignored	
Oracle JDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
GnuTLS 3.1.8	`fatal` *Alert*	`PROTOCOL_VERSION`
Microsoft Schannel (IIS 6.0)	Ignored	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`BAD_RECORD_MAC`

Table 7.30.: Protocol version set to `03 03` in the `ClientHello` message

Overlong `session_ids` A `session_id` is defined to be of variable length between 0 and 32 bytes. This test simply checks the behaviour of a target when confronted with overlong `session_id` values. In this case the value of the `session_id` contains 256 bytes. The results of this test case are listed in Table 7.31.

Stack	Reaction	Error Message
OpenSSL 0.9.8	Ignored	
OpenSSL 1.0.1	Ignored	
OpenJDK 6.24	Ignored	
OpenJDK 7.07	Ignored	
Oracle JDK 4.19	Ignored	
Oracle JDK 5.22	Ignored	
Oracle JDK 6.35	Ignored	
Oracle JDK 7.07	Ignored	
GnuTLS 3.1.8	`fatal` *Alert*	`RECORD_OVERFLOW`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Connection reset	(Response Protocol Version 00 00)
BigIp 8950 11.3.0	Ignored	

Table 7.31.: Overlong `session_id` in the `ClientHello` message

Most implementations simply ignore the overlong `session_id` value, but GnuTLS and BigIP 8950 version 11.1.0 show an interesting behaviour. While the `RECORD_OVERFLOW` *Alert* may be explained with decoding issues the behaviour of the BigIP device remains not comprehensible.

Overlong `session_id`s with wrong length byte In this case the value of the `session_id` contains 256 bytes, but the length byte is set to value `00`. The results of this test case are listed in Table 7.32.

Stack	Reaction	Error Message
OpenSSL 0.9.8	Connection reset	
OpenSSL 1.0.1	Connection reset	
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`RECORD_OVERFLOW`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Connection reset	(Response Protocol Version 00 00)
BigIp 8950 11.3.0	`fatal` *Alert*	`DECODE_ERROR`

Table 7.32.: Overlong `session_id`s with wrong length byte in the `ClientHello` message

While the error of the BigIP device remains the same (version 11.1.0) all other implementations show behaviour likely to be related to decoding problems – which is the expected behaviour.

Record Compression Support The byte indicating compression support in the `ClientHello` message is set to the fictional value `a1` and the target's response is analyzed. The results of this test case are listed in Table 7.33.

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`DECODE_ERROR`
OpenSSL 1.0.1	`fatal` *Alert*	`DECODE_ERROR`
OpenJDK 6.24	Ignored	
OpenJDK 7.07	Ignored	
Oracle JDK 4.19	Ignored	
Oracle JDK 5.22	Ignored	
Oracle JDK 6.35	Ignored	
Oracle JDK 7.07	Ignored	
GnuTLS 3.1.8	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Ignored	
BigIp 8950 11.3.0	`fatal` *Alert*	`DECODE_ERROR`

Table 7.33.: Record compression set to `a1` in the `ClientHello` message

Since *Record* compression is only supported by a few implementations[17] the ignorant behaviour seems legal, but is invlaid according to RFC 2246: *"If the session_id field is not empty (implying a session resumption request) it must include the compression_method from that session. This vector must contain, and all implementations must support, compressionMethod.null."* – *Source: RFC 2246*. The `DECODE_ERROR` *Alert* is misleading but may be related to a check for this mandatory requirement.

Wrong Length For the Cipher Suite List The length of the contained cipher suite list was set to a wrong value (`01`). The results of this test case are listed in Table 7.34.

Stack	Reaction	Error Message
OpenSSL 0.9.8	Connection reset	
OpenSSL 1.0.1	Connection reset	
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`RECORD_OVERFLOW`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Connection reset	(Response Protocol Version 00 00)
BigIp 8950 11.3.0	Connection reset	

Table 7.34.: Wrong length of the cipher suite list in the `ClientHello` message

All implementations seem to be unable to decode the message correctly when confronted with an invalid length of the cipher suite list. Once again BigIP 8950 11.1.0 behaves unexpectedly.

Invalid Length for the Cipher Suite List The length of the contained cipher suite list was set to an invalid value (`00`). The results of this test case are listed in Table 7.35.

The results seem all to be caused by the incorrect length of the cipher suite list. BigIP 8950 11.1.0 behaves unexpectedly.

`ClientKeyExchange`

Invalid DH Exchange Parameters In case a cipher suite requiring (ephemeral) Diffie-Hellman Key Agreement (DHE/DH) [13] for key exchange is chosen, both

[17] For details please see `http://en.wikipedia.org/wiki/Comparison_of_TLS_implementations#Compression` .

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`DECODE_ERROR`
OpenSSL 1.0.1	`fatal` *Alert*	`DECODE_ERROR`
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 5.22	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`HANDSHAKE_VALUE`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Connection reset	(Response Protocol Version 00 00)
BigIp 8950 11.3.0	`fatal` *Alert*	`DECODE_ERROR`

Table 7.35.: Invalid length of the cipher suite list in the `ClientHello` message

sides have to compute a common secret based on mutually exchanged parameters which is then used as the *PreMasterSecret*. Invalid DH parameters may lead to computational errors which are handled differently by the tested stacks. To evaluate the target stack's behaviour the test sends a `00 00` as public DH/DHE value which leads to different behaviour on the target side. The results of this test case are listed in Table 7.36.

Stack	Reaction	Error Message
OpenSSL 0.9.8	Connection reset	
OpenSSL 1.0.1	Connection reset	
OpenJDK 6.24	`fatal` *Alert*	`INTERNAL_ERROR`
OpenJDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 4.19	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 5.22	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 6.35	`fatal` *Alert*	`INTERNAL_ERROR`
Oracle JDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
GnuTLS 3.1.8	`fatal` *Alert*	`INTERNAL_ERROR`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	Connection reset	

Table 7.36.: Public DH(E) value set to `00 00` in the `ClientKeyExchange` message

Some Java versions and GnuTLS seem to fail during public value processing causing the `Alert` message. These processing errors are definitely not expected and cause, as a last resort, an `INTERNAL_ERROR` `Alert`.

`ChangeCipherSpec`

The `ChangeCipherSpec` message requires only a single byte as payload: `01`. Differing values should trigger an `Alert` message. The stack behaviour was anlayzed for two different cases:

1. Payload is `ff` – which is different from the mandatory `01`
2. Payload contains more than a single byte (`02 01`.

The results of the analyzed stacks follow below.

Payload set to `ff` The results of this test case are listed in Table 7.37.

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`ILLEGAL_PARAMETER`
OpenSSL 1.0.1	`fatal` *Alert*	`ILLEGAL_PARAMETER`
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	Ignored	
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Ignored	
BigIp 8950 11.3.0	`fatal` *Alert*	`ILLEGAL_PARAMETER`

Table 7.37.: Payload set to `ff` in the `ChangeCipherSpec` message

Except from GnuTLS and the older release of the BigIP 8950 firmware all implementations detect the wrong payload of the `ChangeCipherSpec` message and sent an adequate response.

Payload set to `02 01` The results of this test case are listed in Table 7.38.
All implementations react validly to a wrong payload.

`Finished`

The `Finished` message is the first encrypted message if a cipher suite with encryption was selected.

Invalid Padding of Concatenated Payload and MAC This tests assume that the `Finished` message is about to be encrypted. It is worth reminding that SSL/TLS uses the *Mac-then-Pad-then-Encrypt* scheme. In this test case the padding applied after MACing, but before encrypting is manipulated. After

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`ILLEGAL_PARAMETER`
OpenSSL 1.0.1	`fatal` *Alert*	`ILLEGAL_PARAMETER`
OpenJDK 6.24	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
OpenJDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 4.19	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 5.22	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 6.35	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Oracle JDK 7.07	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
GnuTLS 3.1.8	`fatal` *Alert*	`UNEXPECTED_MESSAGE`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`ILLEGAL_PARAMETER`

Table 7.38.: Payload set to `02 01` in the `ChangeCipherSpec` message

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenSSL 1.0.1	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenJDK 6.24	`fatal` *Alert*	`HANDSHAKE_FAILURE`
OpenJDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 4.19	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 5.22	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 6.35	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
GnuTLS 3.1.8	`fatal` *Alert*	`BAD_RECORD_MAC`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Ignored	
BigIp 8950 11.3.0	Ignored	

Table 7.39.: Invalid padding value set in the `Finished` message

the manipulation the padding value is invalid (to be precicse, the last padding byte is removed). The results of this test case are listed in Table 7.39.

Very alarming are the results for the BigIp devices. The behaviour suggests that the padding of SSL/TLS *Records* is not checked at all. As this misbehaviour can possibly result in a major security vulnerability – padding verification is relevant to security – the issue was communicated to the vendor.

Invalid MAC The MAC value protecting the plain `Finished` message is invalidated in this test. For no special reason the sixth byte of the MAC is changed to `00`. The results of this test case are listed in Table 7.40.

As in the test case before, the result for the BigIp 8950 device with software release 11.1.0 is alarming. The behaviour suggests that the MAC of SSL/TLS

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenSSL 1.0.1	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenJDK 6.24	`fatal` *Alert*	`HANDSHAKE_FAILURE`
OpenJDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 4.19	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 5.22	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 6.35	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
GnuTLS 3.1.8	`fatal` *Alert*	`BAD_RECORD_MAC`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Ignored	
BigIp 8950 11.3.0	`fatal` *Alert*	`BAD_RECORD_MAC`

Table 7.40.: Invalid MAC value set in the `Finished` message

Records is not checked at all. Againn, the issue was communicated to the vendor.

Tampering with the Message Hash The hash value authenticating all messages of the handshake is destroyed in this test. For no special reason the sixth byte of the hash is changed to `00`. The results of this test case are listed in Table 7.41.

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`DECRYPT_ERROR`
OpenSSL 1.0.1	`fatal` *Alert*	`DECRYPT_ERROR`
OpenJDK 6.24	`fatal` *Alert*	`HANDSHAKE_FAILURE`
OpenJDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 4.19	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 5.22	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 6.35	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
GnuTLS 3.1.8	`fatal` *Alert*	`INTERNAL_ERROR`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`DECRYPT_ERROR`

Table 7.41.: Invalidated message hash set in the `Finished` message

All implementations detect the tampering, but the GnuTLS *Alert* is either misleading or points towards a software bug. It is assumed that the `DECRYPT_ERROR` *Alert* is simply a more general error message to hide the real cause of the problem.

Manipulate Verify Data The value of `verify_data` in the `Finished` message can be manipulated by destroying an input parameter of the pseudo-random function which is used for computing `verify_data`. The results of this test case are listed in Table 7.42.

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenSSL 1.0.1	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenJDK 6.24	`fatal` *Alert*	`HANDSHAKE_FAILURE`
OpenJDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 4.19	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 5.22	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 6.35	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
GnuTLS 3.1.8	`fatal` *Alert*	`BAD_RECORD_MAC`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	`fatal` *Alert*	`HANDSHAKE_FAILURE`
BigIp 8950 11.3.0	`fatal` *Alert*	`BAD_RECORD_MAC`

Table 7.42.: Invalid `verify_data` set in the `Finished` message

In this test case, every implementation behaves as expected.

Change Padding Length This test is very similar to the test invalidating the padding value, but modifies the padding length byte instead of removing it. The length is set to `00` signalizing no added padding. The results of this test case are listed in Table 7.43.

Stack	Reaction	Error Message
OpenSSL 0.9.8	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenSSL 1.0.1	`fatal` *Alert*	`BAD_RECORD_MAC`
OpenJDK 6.24	`fatal` *Alert*	`HANDSHAKE_FAILURE`
OpenJDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 4.19	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 5.22	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 6.35	`fatal` *Alert*	`HANDSHAKE_FAILURE`
Oracle JDK 7.07	`fatal` *Alert*	`HANDSHAKE_FAILURE`
GnuTLS 3.1.8	`fatal` *Alert*	`BAD_RECORD_MAC`
Microsoft Schannel (IIS 6.0)	Connection reset	
Microsoft Schannel (IIS 7.5)	Connection reset	
BigIp 8950 11.1.0	Ignored	
BigIp 8950 11.3.0	`fatal` *Alert*	`BAD_RECORD_MAC`

Table 7.43.: Padding length set to `00` in the `Finished` message

Again the BigIp devices with software release 11.1.0 do not seem to validate the padding at all. The issue was reported.

7.6.4. Check Handshake Message Stapling

In a TLS handshake several messages can either be sent in batch or seperately. As an example, regard the `ServerHello`, `Certificate` and `ServerHelloDone` messages. There are two ways to send these messages:

1. The simple one is to send each message in a separate *Record*.
2. To improve performance it is also possible to bundle the messages in a single *Record* (within a single handshake message).

The results of this test case are listed in Table 7.44.

Stack	Support
OpenSSL 0.9.8	
OpenSSL 1.0.1	
OpenJDK 6.24	X
OpenJDK 7.07	X
Oracle JDK 4.19	X
Oracle JDK 5.22	X
Oracle JDK 6.35	X
Oracle JDK 7.07	X
GnuTLS 3.1.8	
Microsoft Schannel (IIS 6.0)	X
Microsoft Schannel (IIS 7.5)	X
BigIp 8950 11.1.0	
BigIp 8950 11.3.0	

Table 7.44.: Message stapling support

7.6.5. Check for Extension Support

A `ClientHello` message signalizing support for all officially listed extensions[18]. The `ServerHello` in response is analyzed to determine the supported extensions. As the T.I.M.E. Framework only supports the extensions required for ECC, it is refered to a reguarly updated list at Wikipedia for details on extension support: `https://en.wikipedia.org/wiki/Comparison_of_TLS_implementations`

[18] `https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml`

"In cryptography, the algorithms can be strong, but the implementation can be wrong"

Unknown Wise Man

8 Breaking the Defense

As already discussed in Chapter 5, history revealed many attacks on SSL/TLS. Some of these attacks exploited implementation bugs, some of them were based on conceptual flaws. This chapter presents novel, yet unpublished, applications of Bleichenbacher's attack (see Subsection 5.2.7) and is based on an, at the time of writing, unpublished paper together with Juraj Somorovsky, Sebastian Schinzel, Eugen Weiss, Jörg Schwenk and Erik Tews.

As preconditions, the SSL/TLS session needs to use RSA key exchange and an attacker must have recorded the encrypted traffic for the session of interest. The attack can be performed completely from remote and can not be detected by the client of the recorded session.

8.1. Resurrecting Fixed Attacks

Bleichenbacher's attack (see Subsection 5.2.7) was believed to be fixed with the following countermeasure:

> "The best way to avoid vulnerability to this attack is to treat incorrectly formatted messages in a manner indistinguishable from correctly formatted RSA blocks. Thus, when it receives an incorrectly formatted RSA block, a server should generate a random 48-byte value and proceed using it as the premaster secret. Thus, the server will act identically whether the received RSA block is correctly encoded or not." – *Source: RFC 2246 [43]*

In simple words, the server is advised to create a random *PreMasterSecret* in case of problems during processing of the received, encrypted *PreMasterSecret* (structure violations, decryption errors, etc.). The server must continue the handshake with the randomly generated *PreMasterSecret* and perform all subsequent computations with this value. This leads to a `fatal` *Alert* when

checking the `Finished` message (because of different key material at client- and server-side), but it does not allow the attacker to distinguish valid from invalid (e.g. PKCS#1 v1.5 compliant and non-compliant) ciphertexts. In theory, an attacker gains no additional information on the ciphertext if this countermeasure is applied.

8.1.1. Bleichenbacher Oracles

This research relies on the original Bleichenbacher attack (see Subsection 5.2.7) together with algorithm optimizations (see Subsections 5.2.12 and 5.2.32) and investigates the chances of turning seemingly secure SSL/TLS servers into Bleichenbacher oracles. An oracle $\mathcal{O}$ responds with *true* if the decrypted ciphertext starts with 00 02, or $false$ otherwise. The attacker communicates with this oracle $\mathcal{O}$ and sends ciphertexts. The oracle in turn, sends the ciphertexts to the server by performing a TLS handshake, evaluates its responses and returns *true* or $false$ according to the message validity.

Such an oracle $\mathcal{O}$ can be based on different side channels. First, *noisy* SSL/TLS servers responding with different error messages or second, even if the server does not respond with meaningful error messages, its internal processing logic can cause timing differences while processing valid and invalid ciphertexts.

To be useful, an oracle $\mathcal{O}$ must fulfil the following preconditions:

1. $\mathcal{O}$ must not respond with false positives: ciphertexts falsely identified as valid are fatal to Bleichenbacher's algorithm.
2. $\mathcal{O}$ should respond with as little false negatives as possible: valid ciphertexts falsely identified as invalid slow down the attack performance.
3. $\mathcal{O}$ should make as few queries as possible to the server.

Original Bleichenbacher Oracles If the implementation responds with distinguishable error messages leaking information whether a decrypted *PreMasterSecret* starts with 00 02 or not, the oracle can easily be built by analyzing these error messages. In this case the server is *noisy*. Such an oracle was described and used by Bleichenbacher in [78] to attack SSL.

Oracles without Presence of Distinguishable Error Messages When the internal processing can no longer be observed by inspecting response messages (e.g. when no differing *Alert* messages can be triggered), an attacker is forced to find new strategies. A promising side channel is the processing time of different data. Drawing conclusions on internal processing, only by using timing side channels, is a challenging, but feasible task as attacks from the past have shown (see e.g. Subsections 5.2.10, 5.2.27, 5.2.36 and 5.2.38).

8.1.2. Methodology

The T.I.M.E. framework introduced in Chapter 6 was used to implement the attack scenarios. The Bleichenbacher attack logic was built directly upon the

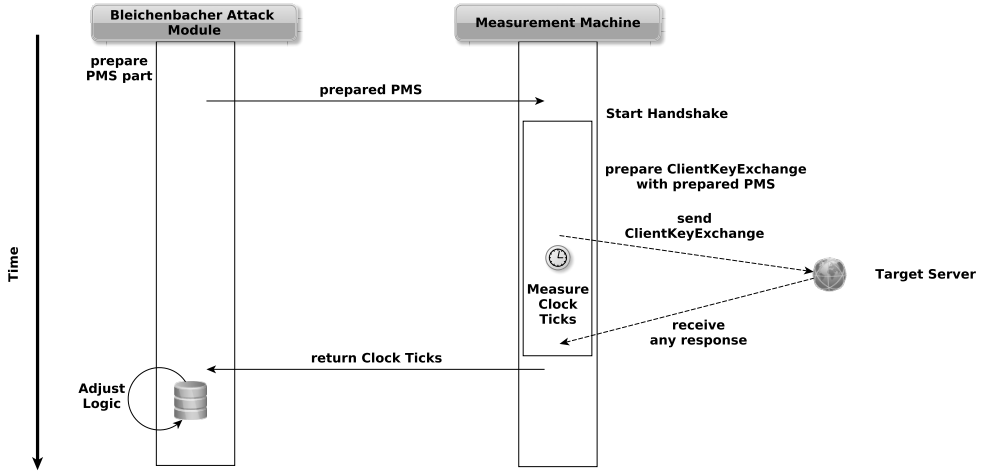

Figure 8.1.: Architecture for measuring timing differences. The enhanced T.I.M.E. framework is split into two parts: The Bleichenbacher Attack Module and the Measurement Module based on the MatrixSSL library. Each part is located on a separate machine to minimize noise during the time measurement process.

TLS stack of T.I.M.E. and can be used to modify messages in the TLS handshake. The modified handshake messages are used to trigger different server behavior. This makes it possible to automatically analyze whether a remote SSL/TLS server is vulnerable to Bleichenbacher's attack.

For timing based side channel attacks, the measurement setup was slightly different, because T.I.M.E. provides no reliable base for highly fine grained time measurement. The Java Virtual Machine causes too much noise to give reliable results. Thus, T.I.M.E. was located on a separate machine and communicates as a client with the remote SSL/TLS server. But, the messages are not sent directly to the target machine. Instead, the time measurement component was deployed on another separate machine and receives the modified messages from the Bleichenbacher logic built on top of T.I.M.E. A patched MatrixSSL - Open Source Embedded SSL and TLS (MatrixSSL) stack[1] injects these messages into a handshake and measures the time between sending the the last byte of the message and arrival of the first byte of the subsequent response. Figure 8.1 illustrates this measurement setup. The machine hosting the Bleichenbacher attack logic that triggers and executes the attack is located on the left. The Bleichenbacher logic generates new ciphertexts and sends them to the measurement machine. The measurement machine in the middle is repsonsible for executing the handshake and measuring time. This machine runs a stripped down Linux (disabled power management, disabled frequency scaling, etc.) and a Realtek 8139 NIC for optimal time measurement. Finally, the target machine (server) can be found on the right side. This architecture turned out to be reliable and accurate for highly fine grained timing measurements. To analyze

[1] `https://www.matrixssl.org/`

the timing datasets, the timing analysis tool FAUtimer[2] was used to evaluate the server response times. With this setup, timing differences of a few hundred nanoseconds over a switched network could be distinguished.

During the tests for Bleichenbacher side channels, three different variations of *ClientKeyExchange* payload (encrypted *PreMasterSecret*) were used to trigger different checks during the PKCS#1 v1.5 processing:

1. PKCS#1 v1.5 compliant ciphertexts containing valid *PreMasterSecret*s (see e.g. Subsection 2.2.3).

2. PKCS#1 v1.5 compliant messages which do not contain valid *PreMasterSecret*s (e.g. wrong protocol version or invalid length).

3. Non-PKCS#1 v1.5 compliant messages (errors in the predefined structure, e.g. messages starting with a non-zero byte or missing 00 separation byte after the random padding).

8.2. Exploiting PKCS#1 Processing in JSSE

The attack presented in the following is confirmed by Oracle and assigned CVE-2012-5081. The vulnerability is fixed with the *Oracle Java SE Critical Patch October 2012 – Java SE Development Kit 6, Update 37 (JDK 6u37).*

8.2.1. Hidden Errors During PKCS#1 Processing

It is very important to hide the current internal processing steps from the outside world. Bleichenbacher's attack (see Subsection 5.2.7) can only be used if information on the PKCS#1 processing is available. The following attack is caused by errors in the SSL/TLS implementation of Java. The PKCS#1 v1.5 processing can be forced to cause different error messages from which the following oracle can be build:

$$\mathcal{O}_{JSSE1}(c) = \begin{cases} true & \text{if } \textit{Alert} \text{ message is } \texttt{INTERNAL\_ERROR} \\ false & \text{otherwise} \end{cases}$$

In this case c denotes the candidate for an (RSA) encrypted (and hence PKCS#1 v1.5 encoded) *PreMasterSecret*. With the knowledge that the *Alert* of type `INTERNAL_ERROR` is only caused if the decrypted candidate c is of a particular form (leading 00 02 bytes), Bleichenbacher's attack is possible again. 00 bytes in specific positions of the unencrypted PKCS#1 v1.5 structure caused the stack to access invalid positions of an internal array, finally leading to an internal `ArrayIndexOutOfBoundsException`. If $\mathcal{O}_{JSSE1}$ responds with *true* the ciphertext represents a valid PKCS#1 v1.5 structure, whereas $false$ suggests an invalid candidate.

[2] `https://code.google.com/p/fau-timer/`

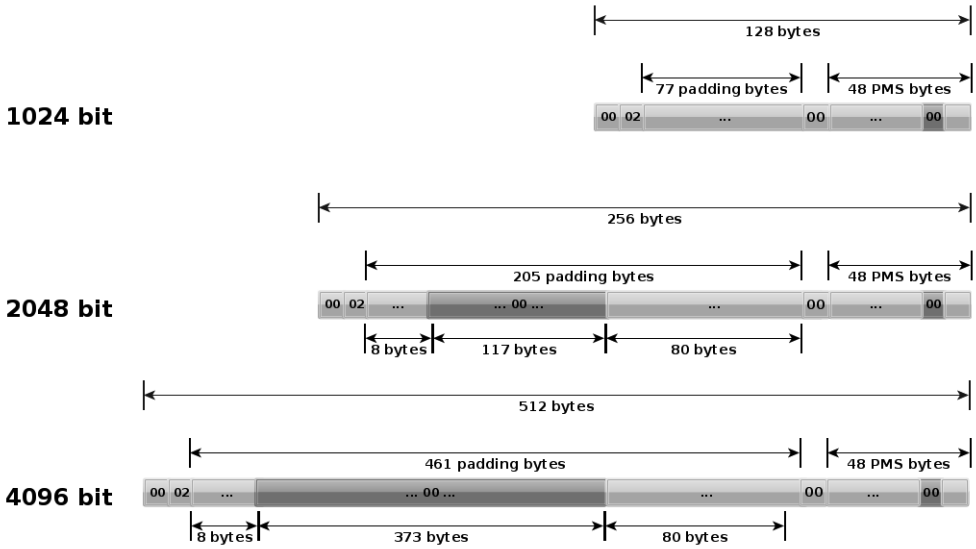

Figure 8.2.: If the decrypted candidate contains a 00 byte in one of the marked positions, JSSE responds with an `INTERNAL_ERROR` *Alert*.

Provoking the Errors The error which caused the side channel was found during code review and automated fuzzing using the T.I.M.E. Framework (see Chapter 6). The bug is caused by an implementation error in the routine responsible for the padding check of a PKCS#1 v1.5 structure. An improper padding check and the subsequent processing of the *PreMasterSecret* could lead to an array index violation (`ArrayIndexOutOfBoundsException`) under special conditions. These conditions are caused by 00 bytes in specific positions. Propagation of the thrown `Exception` to the surface lead to an `INTERNAL_ERROR` SSL/TLS *Alert*. Figure 8.2 illustrates the cases for different key sizes that trigger the desired behaviour (`INTERNAL_ERROR`).

8.2.2. Oracle Analysis

During code review the root cause of the `INTERNAL_ERROR` *Alert* was identified as an `ArrayIndexOutOfBoundsException`. The `Exception` is thrown in the `com.sun.crypto.provider.TlsPrfGenerator.expand(..)` function that performs XOR operations on byte arrays in a loop (see Listing 8.1). The pitfall is a missing length check on the array sizes. An array with length $<$ `secLen` triggers the exception. Such cases can be provoked as described above.

```
1 for (int i = 0; i < secLen; i++) {
2             pad1[i] ^= secret[i + secOff];
3             pad2[i] ^= secret[i + secOff];
4 }
```

Listing 8.1: Internal loop causing an `Exception` in special cases – *Source: com.sun.crypto.provider.TlsPrfGenerator*

The exception is finally caught at last stage in the SSL/TLS stack and thus leads to the `INTERNAL_ERROR` *Alert*.

Oracle Strength The following evaluates the probability for 2048 and 4096 bit random messages starting with 00 02 to contain a structure causing an `INTERNAL_ERROR` *Alert*. Let n be the byte size of the PKCS#1 v1.5 message and $|PMS|$ the length of the *PreMasterSecret*. The number of bytes provoking the desired alert can be derived as:

$$x = n - 3 - |PMS| - 8 - 80.$$

When considering that the first two message bytes are 00 02, the probability that the following 8 padding bytes are non-zero and at least one of the following x bytes becomes 00 (and thus the server responds with an `INTERNAL_ERROR` *Alert*) is:

$$P_{JSSE1}\left(1|A\right) = \left(\frac{255}{256}\right)^{8} \cdot \left(1 - \left(\frac{255}{256}\right)^{x}\right)$$

For key sizes of 2048 and 4096 bits (256 and 512 bytes) it results in:

$$P^{2048}_{JSSE1}\left(1|A\right) = \left(\frac{255}{256}\right)^{8} \cdot \left(1 - \left(\frac{255}{256}\right)^{117}\right) \approx 0.356$$

$$P^{4096}_{JSSE1}\left(1|A\right) = \left(\frac{255}{256}\right)^{8} \cdot \left(1 - \left(\frac{255}{256}\right)^{373}\right) \approx 0.744$$

This means that a server (with behaviour according to $\mathcal{O}_{JSSE1}$) using a 2048 bit RSA key responds with a probability of $P^{2048}_{JSSE1}(1|A) \approx 35.6\%$ with *true* (`INTERNAL_ERROR`), if the decrypted *PreMasterSecret* message starts with 00 02. In case of using 4096 bit keys, the oracle is even more permissive. It responds with a probability of $P^{4096}_{JSSE1}(1|A) \approx 74.4\%$ if the message starts with 00 02. These probabilities present a low number of false negatives leading to an application of an efficient Bleichenbacher's attack.

On the other hand, when applying 1024 bit RSA keys, $\mathcal{O}_{JSSE1}$ is much less permissive. It responds with the desired *Alert* only if 00 is positioned just before the last byte. Thus, the probability $P^{1024}_{JSSE1}(1|A)$ can be computed as:

$$P^{1024}_{JSSE1}\left(1|A\right) = \left(\frac{255}{256}\right)^{124} \cdot \left(\frac{1}{256}\right) \approx 0.0024$$

Attack Evaluation By using this new oracle $\mathcal{O}_{JSSE1}$ the *PreMasterSecret* of a previously recorded session could be successfully extracted. The attack took about 5h (`localhost` server) and nearly 460000 queries to the server (2048 bit key, unimproved Bleichenbacher algorithm, Java 1.6). Figure 8.3 shows the output of the attack.

The first byte (00) is suppressed by the output logic, the following byte 02 defines a PKCS#1 v1.5 structure used for encryption, the first byte surrounded by the rectangular (00) separates the *PreMasterSecret* from the preceeding padding bytes, the last bytes (03 01) surrounded by the rectangular represent the protocol version id for TLS 1.0, the following 46 bytes are the randomly

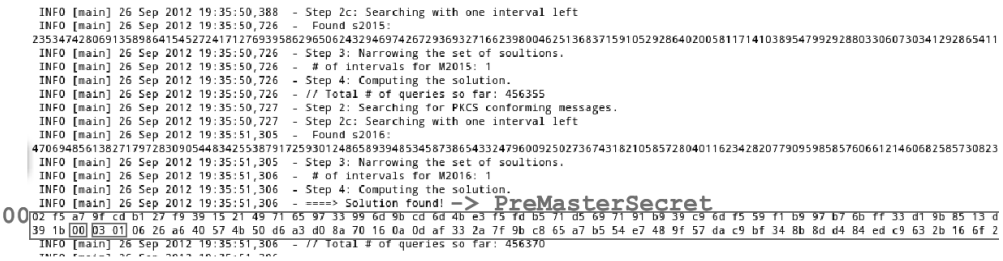

Figure 8.3.: Successful Bleichenbacher Attack on JSSE (about 5h duration and nearly 460000 queries).

	Queries Mean	Queries Median
2048 bit RSA key	176,797	37,399
4096 bit RSA key	73,710	27,744

Table 8.1.: Number of required queries for the optimized Bleichenbacher's attack on JSSE.

chosen bytes by the client. With the knowledge of the *PreMasterSecret* an attacker can easily recompute the keys used in this connection and decrypt the communication of this SSL/TLS session.

Performance Tweaks In order to enhance the attack performance the improvements to Bleichenbacher's algorithm as proposed by Bardou et al. [112] (explained in detail in Subsection 5.2.32) were implemented and applied to the given scenario. The number of queries could be significantly reduced (for details see Table 8.1). The case of 1024 bit keys was deliberately ommited as the oracle strength is too weak to be practical. Performance evaluation of the (highly restrictive) oracle using 1024 bit keys resulted in hundreds of millions of oracle queries.

8.3. Secret Dependent Processing Branches Lead to Timing Side Channels

This potential issue was confirmed by Oracle and assigned CVE-2014-0411. A fix is available since *Oracle Java SE Critical Patch January 2014 – Java SE Development Kit 7, Update 51 (JDK 7u51)*. Meanwhile, the OpenSSL library was patched independently during our research in April 2013 to provide a constant timing while handling RSA ciphertexts[3].

8.3.1. Timing Differences Caused by Random Number Generation

A conspicuousness with respect to the random *PreMasterSecret* generation was already obvious during the code analysis of JSSE for the previous attack (see Section 8.2): The random *PreMasterSecret* is only generated if problems oc-

[3] `https://github.com/openssl/openssl/commit/adb46dbc6dd7347750df2468c93e8c34bcb93a4b`

cured during PKCS#1 v1.5 decoding. Otherwise, no random bytes are generated. Listing 8.2 sketches the processing (no real source code).

```
1  byte preMasterSecret = new byte[48];
2  byte[] plainText = decrypt(clientKeyExchangePayload);
3
4  /*
5   * Plaintext is PKCS#1 v1.5 conform if its encoding is of the
        following form:
6   *  00 02 || Padding || 00 || PreMasterSecret
7   *
8   * Padding.length >= 8, PreMasterSecret.length = 48
9   */
10 if(!isPKCS15Conform(plainText) || plainText[plainText.length-
       preMasterSecret.length-1] != 0) {
11   preMasterSecret = generateRandomPMS();
12 } else {
13   System.arraycopy(plainText, plainText.length-preMasterSecret.
         length, preMasterSecret, 0, preMasterSecret.length);
14 }
```

Listing 8.2: Sketch of PreMasterSecret processing – for the sake of clarity additional logic is omitted and processing is simplified.

This represents the countermeasure as standardized in the TLS 1.0 [43] and TLS 1.1 [44] specifications. TLS 1.2 advises a different processing (cf. Chapter 7.4.7.1. of RFC 5246 [45]): Generate random bytes everytime a *PreMasterSecret* is decrypted and perform the following computations (e.g. *MasterSecret* derivation) either with the decrypted or the random value, depending on the validity of the received *PreMasterSecret*. With this countermeasure a random value is always generated *before* the processing of decrypted data. This puts secret dependent processing aside.

The countermeasure as defined by TLS 1.0 and 1.1 could, hypothetically, cause a timing side channel which may lead to an oracle suitable for Bleichenbacher's algorithm $\mathcal{O}_{RandomPMSGeneration}$. But, when calculating the strength of this oracle, it revealed to be unexploitable, since it is too weak[4]. If the oracle $\mathcal{O}_{RandomPMSGeneration}$ responds with $true$ the ciphertext represents a valid *PreMasterSecret*, whereas $false$ suggests an invalid *PreMasterSecret* candidate.

$$\mathcal{O}_{RandomPMSGeneration}(c) = \begin{cases} true & \text{expected normal response time} \\ false & \text{otherwise (random number generation)} \end{cases}$$

Provoking the Differences in Processing Time As can be seen in Listing 8.2 the random number generation is only performed if there are errors during the decryption/PKCS#1 v1.5 decoding process. Thus, to trigger random number generation, it is sufficient to invalidate the PKCS#1 v1.5 structure. This was done by changing the first byte (which is 00 by definition, see Subsection 2.2.3 for details) of the PKCS#1 v1.5 encoded *PreMasterSecret* to 08. The value of the modified first byte (08) was chosen for no particular reason.

[4]Remember: The timing side channel would - if at all - only be available if the PKCS#1 v1.5 structure can be recognized as such (starts with 00 02) which is very seldom the case.

The side channel, if existent, would be caused by the incorrect countermeasure description in the TLS 1.0 [43] and TLS 1.1 [44] specifications.

8.3.2. Oracle Analysis

The described behaviour was at first identified in Java's JSSE code (function `sun.security.ssl.Handshaker.calculateMasterSecret(...)`) where a random number generation is only performed in case of errors during the *PreMasterSecret* processing (see Listing 8.3).

```
1 try {
2     KeyGenerator kg = JsseJce.getKeyGenerator(masterAlg);
3     kg.init(spec);
4     masterSecret = kg.generateKey();
5 } catch (GeneralSecurityException e) {
6
7  ... code stripped ...
8
9     if (requestedVersion != null) {
10        preMasterSecret =
11            RSAClientKeyExchange.generateDummySecret(
                  requestedVersion);
12    } else {
13        preMasterSecret =
14            RSAClientKeyExchange.generateDummySecret(
                  protocolVersion);
15    }
16
17    // recursive call with new premaster secret
18    return calculateMasterSecret(preMasterSecret, null);
19 }
20
21  ... Code to check if the protocol version matches, returns
      directly the
22 masterSecret on success ...
23
24 preMasterSecret =
25     RSAClientKeyExchange.generateDummySecret(requestedVersion);
26
27 // recursive call with new premaster secret
28 return calculateMasterSecret(preMasterSecret, null);
```

Listing 8.3: Timing side channel causing Java code for processing a received *PreMasterSecret* (simplified).

In lines 11, 14 and 25 a random *PreMasterSecret* is possibly generated, depending on the received *PreMasterSecret* validity. This generation is either triggered by problems during decryption/decoding of the PKCS#1 v1.5 structure or if the protocol version of the *PreMasterSecret* does not match the one established during the Handshake.

Beyond JSSE, OpenSSL and GnuTLS revealed quite similar processing logic. The findings, especially the behaviour of OpenSSL, were independently discovered and criticized by Green in a tweet[5].

[5] `https://twitter.com/OpenSSLFact/status/253060773218222081`

During experimental tests, valid, encrypted *PreMasterSecret*s, as well as ciphertexts with an invalid PKCS#1 v1.5 structure (the first byte - which should be 00 - was changed to 08) were sent to an OpenSSL 0.98 based server (`openssl s_server`). While the first case follows the normal processing, the second case triggers the random number generation. Figure 8.4 shows the filtered results of the timing analysis, conducted over a LAN with 5000 measurements. The rectangles define the interval of timing differences that occured during the measurement. The Figure illustrates the results for valid and invalid PKCS#1 v1.5 structures (ciphertexts).

As can be seen, the measurement reveals a clear boundary between valid and invalid ciphertexts of about 1.5 μs. The result suggests that distinguishing a valid, PKCS#1 v1.5 encoded and encrypted *PreMasterSecret* from a ciphertext with invalidated PKCS#1 v1.5 structure is possible by using the timing side channel. Similar results could be observed for OpenSSL 1.01. But, even though the results clearly showed constant differences of about 1.5 μs, it is not clear if these differences were only caused by the additional random number generation or if other processing (such as e.g. error handling) was also involved in creating the differences in time. The analysis of this problem revealed to be very difficult and compiler/compile-flag dependent. As a hypothesis it can be concluded that different code blocks (of whatever kind) at different stages during the processing are responsible for this side channel. Despite this uncertainty, the presence of a side channel (of which kind ever, not necessarily only $\mathcal{O}_{RandomPMSGeneration}$) is clearly confirmed.

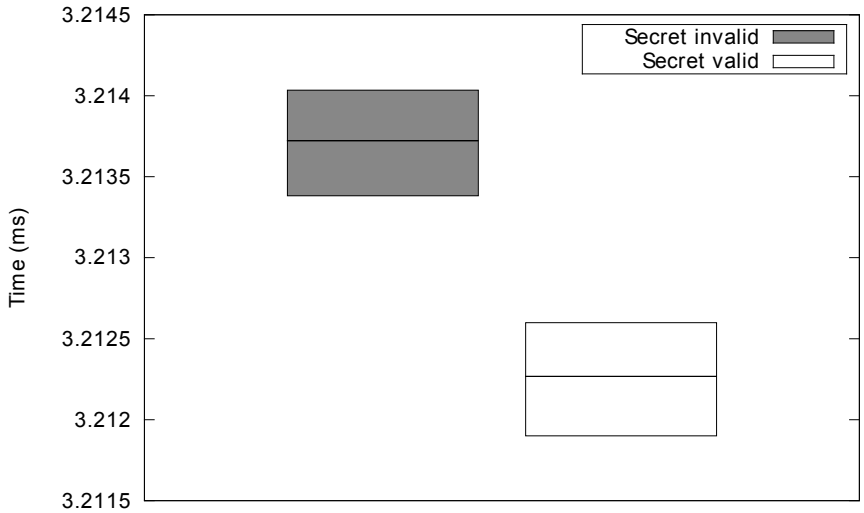

Figure 8.4.: Timing measurement results for OpenSSL 0.98. The valid secret refers to a TLS compliant ciphertext with correct *PreMasterSecret*. The invalid secret refers to a non-PKCS#1 v1.5 compliant ciphertext. In the non-PKCS#1 v1.5 compliant structure the first byte (which should be 00) was altered to 08 to provoke a random number generation on server-side.

Oracle Strength However, the findings did not lead to a practical attack. An oracle created from this timing leak is very "weak". It responds to an oracle request with *true* if, and only if, the decrypted request represents a valid *PreMasterSecret* with correct protocol version number contained in a valid PKCS#1 v1.5 structure (see Subsections A.1.2 and 2.2.3).

For a 2048 bit key, the probability that an oracle responds with *true* in case that the decrypted message starts with 00 02 is very low:

$$P^{2048}_{RandomPMSGeneration}(1|A) = \left(\frac{255}{256}\right)^{205} \cdot \left(\frac{1}{256}\right)^{3} \approx 2.7 \cdot 10^{-8}$$

The reason is that 205 padding bytes must be non-zero and the following bytes must contain $00||major||minor$ according to the predefined structure of PKCS#1 v1.5 (see Subsection A.1.2) and the *PreMasterSecret* (see Subsection 2.2.3) where *major* and *minor* are the parts of the protocol version.

Attack Evaluation $\mathcal{O}_{RandomPMSGeneration}$ is very "weak" and did not allow a practical attack, hence the number of oracle queries can only be estimated. According to Bleichenbacher [78] and Bardou et al. [112], the number of oracle queries for the complete attack can be computed as:

$$(2^{17} + 16 \cdot 256)/P_{RandomPMSGeneration} = 5 \cdot 10^{12}$$

8.4. Risks of Modern Software Design

This Section is based on the attack in Section 8.3 and takes the idea of measuring processing time one step further. The vulnerability was confirmed by Oracle and assigned CVE-2014-0411. A fix is available since *Oracle Java SE Critical Patch January 2014 – Java SE Development Kit 7, Update 51 (JDK 7u51)*. Oracle used a different solution (compared to the ones presented here) to fix this problem.

8.4.1. Timing Differences Caused by `Exception`s

Again, focus of interest are different processing times for valid and invalid PKCS#1 v1.5 payload. The vulnerability is based on timely expensive creation and handling of `Exception`s in Java. A common implementation pattern for RSA decryption is to provide a (generic) function to which the ciphertext is passed which returns the plaintext on success or an `Exception` otherwise. From a developers point of view this is good function design. But, as already mentioned, `Exception`s are timely expensive and thus are likely to introduce a side channel to the decryption function. A paradoxal result of this research is that this side channel is only caused, because of modular and structured software engineering.

To make things even worse, the object oriented architecture and especially the `Exception` handling of the JSSE implementation makes fixing the uncovered timing leak challenging. Since the generation of the `Exception` in Java creates a

detectable timing difference, but the handling as well, generating an `Exception` in advance, but throwing it only in the error case, results in a *smaller*, but not *equal* difference of processing time which might still be exploitable.

A possible propsal to fix this problem is the following: The PKCS#1 processing needs to be time constant for SSL/TLS. At first, the RSA decryption operation is applied. Then, a random *PreMasterSecret* is (always) generated. The plaintext (PKCS#1 structure), including the padding is passed together with the random *PreMasterSecret* bytes to a custom padding check function. This function also executes in constant time and returns either the real *PreMasterSecret* bytes, if the padding is correct, or the random bytes otherwise.

Exceptions in the PKCS#1 Processing Inspired by a post of James Manger on the JOSE (JSON Object Signing and Encryption) mailing list,[6] an additional side channel could arise from `Exception` handling. Research revealed a side channel in the Java code responsible for PKCS#1 v1.5 processing.

As already mentioned, the side channel is caused by a speciality of Java: `Exception`s. Throwing `Exception`s is very costly and takes much more time than regular `return`s. Measuring the time consumption of throwing `Exception`s is difficult, it depends on the runtime environment and its vendor, architecture, background processes and finally the code responsible for `Exception` handling (e.g. logging, ...). Without discussion of the real costs of throwing, catching and handling an `Exception` it is obvious that the processing times are very likely to differ (compared to a processing without `Exception`s). However, tests simulating the behaviour of JSSE revealed measurable timing differences (dependent on the keysize) between 5-10 μs. While these values are not representative (since they are measured in a simulated environment), they confirm the possible presence of a timing side channel. The vulnerable code can be found in Listing 8.4.

```
1  private byte[] unpadV15(byte[] padded) throws BadPaddingException {
2      int k = 0;
3      if (padded[k++] != 0) {
4        throw new BadPaddingException("Data must start with zero");
5      }
6      if (padded[k++] != type) {
7        throw new BadPaddingException("Blocktype mismatch: " + padded
             [1]);
8      }
9      while (true) {
10       int b = padded[k++] & ff;
11       if (b == 0) {
12         break;
13       }
14       if (k == padded.length) {
15         throw new BadPaddingException("Padding string not terminated
               ");
16       }
17       if ((type == PAD_BLOCKTYPE_1) && (b != ff)) {
18         throw new BadPaddingException("Padding byte not ff: " + b);
19       }
```

[6] http://www.ietf.org/mail-archive/web/jose/current/msg01936.html

```
20      }
21      int n = padded.length - k;
22      if (n > maxDataSize) {
23        throw new BadPaddingException("Padding string too short");
24      }
25      byte[] data = new byte[n];
26      System.arraycopy(padded, padded.length - n, data, 0, n);
27
28      return data;
29  }
```

Listing 8.4: Java's PKCS#1 v1.5 function for format check and padding removal – *Source:* `sun.security.rsa.RSAPadding`

As can be seen, any error in the PKCS#1 v1.5 structure leads directly to a `BadPaddingException` – with additional costs regarding processing time. This characteristic can be exploited as follows: Longer response times indicate that it is very likely that the message was not correctly PKCS#1 v1.5 formatted which implies possible 00 02 mismatches. This oracle $\mathcal{O}_{JSSE2}$ is much stronger than $\mathcal{O}_{RandomPMSGeneration}$ because an invalid PKCS#1 v1.5 structure is the most probable case when performing Bleichenbacher's attack.

$$\mathcal{O}_{JSSE2}(c) = \begin{cases} true & \text{expected response time of normal processing} \\ false & \text{otherwise (Exception handling)} \end{cases}$$

Provoking the Differences in Processing Time The vulnerable code is contained in the `sun.security.rsa.RSAPadding.unpadV15(...)` function. Listing 8.4 reveals that the PKCS#1 structure is strictly checked for correctness. The message must start with 00 02 (when PKCS#1 is used in encryption mode), contain at least eight non-zero padding bytes and a 00 byte indicating the end of the padding string. If this format is correct, the secret is extracted. Otherwise, a `BadPaddingException` is thrown.

When sending non-compliant PKCS#1 v1.5 structures as *PreMasterSecret* during a handshake, each of the `BadPaddingExceptions` could be tiggered when using a JSSE based server (JSSE utilizes the code of Listing 8.4 to decode a PKCS#1 structure). The `Exceptions` showed up in the log files each time the server was presented an invalid PKCS#1 structure. The `Exception` was correctly handled and did not result in a distinguishable error message. But, the additional `Exception` handling in Java (as well as in other object oriented languages) revealed to be timely expensive and thus slows down the whole application. Throwing, catching, and handling an `Exception` results in additional processing time. Therefor, this timing side channel could be confirmed.

8.4.2. Oracle Analysis

The timing differences between processing PKCS#1 v1.5 compliant and non-PKCS#1 v1.5 compliant messages on TLS servers running on Java 1.6 and 1.7 platforms were analyzed. Figure 8.5 shows the filtered results of the timing measurement with 5000 queries. Again, the rectangles define the interval of

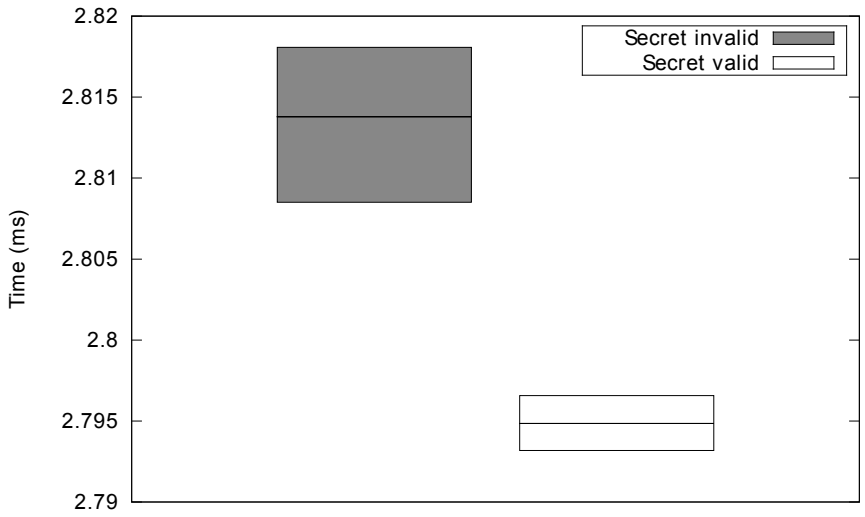

Figure 8.5.: Timing measurement results for Java 1.7 (JSSE). The valid secret refers to a PKCS#1 compliant ciphertext. The invalid secret refers to a non-PKCS#1 compliant ciphertext. In the non-PKCS#1 compliant structure the first byte (which should be 0x00) was altered to 0x08 to provoke an `BadPaddingException` on the server.

timing differences that occured during the measurement. The analysis revealed differences in the processing time of about 20 μs.

Oracle strength The new oracle $\mathcal{O}_{JSSE2}$ is very permissive and much stronger than $\mathcal{O}_{RandomPMSGeneration}$. When working with 2048-bit keys, this oracle responds with *true* (starting with 00 02) with the following probability:

$$P_{T-exc}^{2048}(1|A) = \left(\frac{255}{256}\right)^{8} \cdot \left(1 - \left(\frac{255}{256}\right)^{246}\right) \approx 0.6$$

Applying such an oracle results in much lesser queries and can thus be expected to be used for a practical Bleichenbacher attack.

Attack Evaluation $\mathcal{O}_{JSSE2}$ was used to perform a real (improved) Bleichenbacher attack (see Bardou et al. [112] for details) in a switched LAN and prove the practicability. The attack on OpenJDK 1.6 took about 19.5h and 18600 oracle queries. About 20% of PKCS#1 compliant messages were identified as non-PKCS#1 compliant (false negatives which slow down the attack). The attack on Java 1.7 took about 55h and 20662 queries. The larger number of queries and the longer processing time are caused by a higher value of false negatives (about 50%). The oracle identified about 467 PKCS#1 compliant messages incorrectly.

"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin."

John von Neumann

9 The Crux of Randomness

This Chapter presents flaws in multiple Java-based PRNGs that may cause concerns regarding the security of SSL/TLS stacks written in Java. The results are based on the paper *"Randomly Failed! The State of Randomness in Current Java Implementations"* [125] together with Kai Michaelis and Jörg Schwenk presented at the *RSA Conference 2013 (CT-RSA 2013)*, San Francisco – USA, February 2013.

9.1. Theoretical Weakness

Good random numbers are essential for the security of SSL/TLS. The *Handshake Phase* relies on unpredictable values, such as e.g. the nonces contained in the `ClientHello` and `ServerHello` messages, the randomly chosen *PreMasterSecret* or secret values of a DH/DHE key exchange. If these random numbers could be predicted with a high probability, there is a good chance to seriously harm the security goals of SSL/TLS.

The uncovered shortcomings and flaws of the analyzed libraries may influence the security of Java-based SSL/TLS in terms of predictable random values, but have not been successfully exploited yet.

9.1.1. Problems with Random Numbers in Practice

Because of the importance of good random numbers, multiple Common Weakness Enumeration (CWE)s[1] address problems related to poor randomness.

- CWE-330: Use of Insufficiently Random Values
- CWE-331: Insufficient Entropy

[1] `http://cwe.mitre.org`

- CWE-332: Insufficient Entropy in PRNG
- CWE-333: Failure to Handle Insufficient Entropy in PRNG
- CWE-334: Small Space of Random Values
- CWE-335: PRNG Seed Error
- CWE-336: Same Seed in PRNG
- CWE-337: Predictable Seed in PRNG
- CWE-338: Use of Cryptographically Weak PRNG
- CWE-339: Small Seed Space in PRNG
- CWE-340: Predictability Problems
- CWE-341: Predictable from Observable State
- CWE-342: Predictable Exact Value from Previous Values
- CWE-343: Predictable Value Range from Previous Values

The most prevalent vulnerability caused by predictable random numbers is undoubtedly the Debian OpenSSL Bug [93] where insufficient entropy caused random numbers to be chosen from an interval, small enough to be subject to brute force attacks (for details see Subsection 5.2.17).

Lenstra et al. analyzed in [126] millions of public keys of different types (RSA, DSA, ElGamal and ECDSA) and found keys violating basic principles for secure cryptographic parameters. According to the authors these weak keys are very likely to be caused by poorly seeded PRNGs.

At the USENIX Symposium 2012, Heninger et al. presented in [127] alarming results concerning the quality of keys. Missing entropy was identified as a root cause for many weak and vulnerable keys. Additionally, the authors identified entropy problems under certain conditions in the Linux Random Number Generator (RNG).

9.2. Random Numbers in Java

Java does not provide an API for direct access to real RNGs. Instead, PRNGs are used to derive random numbers by the help of deterministic algorithms from an initial seed. In Java, cryptographically strong *random* numbers are obtained through the `SecureRandom` class which is part of the Java Cryptography Architecture. The determinism of the algorithms implies that two instances of a PRNG (`SecureRandom`) initialized with identical seeding values will always create the same sequence of (pseudo-) *random* numbers. This is on the one hand useful if the sequence of bytes ever has to be correctly recreated, but on the other hand very disadvantageous when *real* randomness is needed. However, when using the PRNG without further parameterization, the PRNG's

implementation automatically selects an initial seed. It is essential how this seed is chosen, because once publicly known the sequences can be regenerated. Therefor, each implementation ships with an entropy collector responsible for getting or generating *random* numbers used for seeding. The strategy of these entropy collectors is library specific and not dictated by any specification.

9.2.1. Methodology

The PRNGs were analyzed as follows: Manual code review was performed with a special focus on checking the code for implementation flaws and obvious bugs. Aside from code review, blackbox tests on the output were performed to grade the entropy. These blackbox tests were taken from the Dieharder test suite[2] and the STS suite [124].

While these tests can not replace cryptanalysis, they still uncover bias and dependency in the pseudo random sequence. For every test exists an expected distribution of outcomes. Each test run produces a value that is compared to the theoretical outcome. A p-value, describing the probability that a real RNG would produce this outcome, between 0 and 1 is computed. A p-value below a fixed significance level α indicates a failure of the PRNG with probability $1 - \alpha$. Dieharder differs from this methodology as it relies on multiple p-values to evaluate the quality of a PRNG. It is possible (and expected) for a good RNG to produce "failing" p-values. Instead of grading a single outcome, 100 p-values are computed and the distribution of these values is compared to an uniform one. This generates a final p-value grading the quality of the PRNG.

Beyond code, algorithm and blackbox test evaluation, no cryptanalysis was performed.

9.2.2. Java Runtime Libraries with `SecureRandom` Implementations

The following paragraph lists the Java Runtime Libraries which were focus of research and presents the algorithms of these libraries. Please note that the listed algorithms are simplified pseudo code in order to facilitate understanding.

Apache Harmony Apache Harmony was an open source implementation of the Java Core Libraries published under the Apache License[3], introduced in 2005. The project became obsolete with the publication of SUN Microsystems' reference implementation in 2006. Although discontinued since 2011, parts of Harmony are used as core libraries of Google's Android platform.

```
1 // require cnt: counter >= 0, state: seed bytes and iv: previous
      output
2 iv = sha1(iv,concat(state,cnt));
3 cnt = cnt + 1;
4
5 return iv;
```

Listing 9.1: Apache Harmony's SecureRandom (Pseudo Code)

[2] `http://www.phy.duke.edu/~rgb/General/dieharder.php`
[3] `http://www.apache.org/licenses/`

The `SecureRandom` implementation of Android uses SHA-1 [128] to generate pseudo-random byte sequences and is – ideally – seeded with *real* random bytes of the OS's RNG. Pseudo-random numbers are generated by calculating the hash value of an internal state concatenated with a 64 bit integer counter and a padding (see the algorithm in Listing 9.1 for details).
Starting at zero, the counter is incremented each run. Additionally, a padding is required to fill remaining bits of an 85 bytes buffer. The padding follows the padding scheme of SHA-1: The last 64 bits hold the length *len* of the values to be hashed in bits. Any space between the end of the values and the length field is filled with a single '1' bit followed by zeros. The resulting hash values are returned as pseudo-random sequence. Figure 9.1 illustrates this state buffer.

	20 byte seed (5 * 32 bit words)					8 byte counter		57 byte padding	
i.	s_0	s_1	s_2	s_3	s_4	$0 \cdots 0$			
ii.	s_0	s_1	s_2	s_3	s_4	cnt_0	cnt_1	$10 \cdots 0$	len

Figure 9.1.: The seed bytes $s_0, .., s_4$ in row i concatenated with a 64 bit counter c_0, c_1 (two 32 bit words), padding bits and the length *len* as in row ii are hashed repeatedly to generate pseudo-random bytes.

Weaknesses. Apache Harmony suffers from two weaknesses caused by implementation bugs. One of these bugs directly addresses the Android platform, whereas the second one only targets Apache Harmony.

FIRST – When a `SecureRandom` instance is created without passing any arguments to the constructor, the implementation *seeds itself*. During this process a byte offset (pointer into the state buffer) is not properly adjusted after inserting a start value. This causes the 64 bit counter and the beginning of the padding (a 32 bit word) to overwrite parts of the seed instead of appending. The remaining 64 bits of entropy render the PRNG useless for cryptographic applications.
The bug was communicated to the Google Security Team and should be fixed in up-to-date Android versions.

	20 byte seed (5 * 32 bit words)					8 byte counter		57 byte padding	
ii.	s_0	s_1	s_2	s_3	s_4	cnt_0	cnt_1	$10 \cdots 0$	len
iii.	cnt_0	cnt_1	$10 \cdots 0$	s_3	s_4	$0 \cdots 0$			len

Figure 9.2.: Instead of appending (cf. row ii), the counter and the succeeding padding overwrite a portion of the seed, yielding row iii.

SECOND[4] – When running under a Unix-like OS a new `SecureRandom` instance is seeded with 20 bytes from the */dev/urandom* or */dev/random* device.

[4]This bug is not part of the Android Source Code

If both are unaccessible, the implementation provides a fall-back seeding facility (cf. Listing 9.2): Seed is gathered from the `random()` PRNG of the GNU C library, which is seeded it via `srandom()` with the UNIX-time, processor time and the pointer value of a heap-allocated buffer. After seeding, `random()` is used to generate seed bytes for `SecureRandom`. Before these bytes are returned the most significant bit is set to zero (mod 128) – this behavior is neither documented, expected nor explained.

```
1 char *seed = malloc(20);
2 srandom(clock() * time() * malloc()) % pow(2,31));
3 for(int i = 0; i < 20; i++)
4   seed[i] = random() % 128;
5
6 return seed;
```

Listing 9.2: Apache Harmony's `getUnixSystemRandom`

The missing entropy is not compensated (e.g. by requesting > 20 bytes). As a consequence, the effective seed of a `SecureRandom` instance is only 7/8 for each requested byte, degrading security (of only 64 bits due to the *first* bug) by another 8 bits to 56 bits (s_3 and s_4 are 2 * 32 bit words == 8 byte). Even worse, the argument of `srandom()` in the GNU C library is of type `unsigned int` while Harmony reduces the argument modulo `INT_MAX` (defined in *limits.h*) - the maximum value for *signed int*s. This limits the entropy of a single call to the seeding facility to 31 bits.

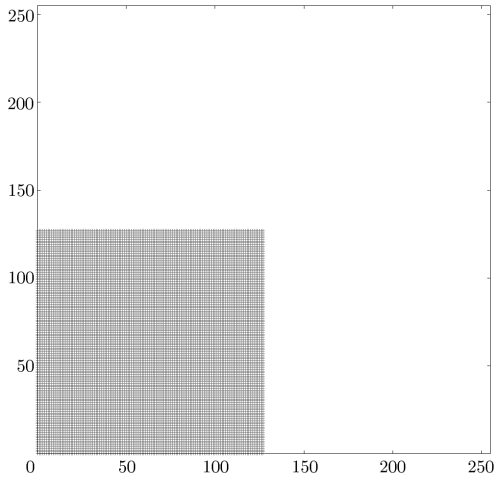

Figure 9.3.: Distribution of 2-tuples from Apache Harmonys integrated seeding facility.

Quality of entropy collectors. Generating 10MiB of seed – two consecutive bytes are interpreted as a single point – leads to the chart in Figure 9.3. It shows a lack of values above 127 in each direction. This test targets the *second* bug. The *first* bug limits the security to 64 or 56 bit, depending on the seeding source.

GNU Classpath GNU Classpath was started in 1998 as the first open source implementation of Java's core libraries. The library is licensed under GPL[5] and is e.g. partly used by the IcedTea[6] project.

```
1 // require state: <= 512 bit buffer, iv: current Initialzation
      Vector
2 byte[] output = sha1(iv,state);
3 state = concat(state,output);
4
5 if(state.length > 512) {     // in bits
6   iv = sha(iv,state[0:512]);  // first 512 bits
7   state = state[512:-1];      // rest
8   output = sha1(iv,state);
9 }
10
11 return output;
```

Listing 9.3: GNU Classpath's SecureRandom (Pseudo Code)

The `SecureRandom` implementation (algorithm in Listing 9.3) of GNU Classpath is powered by a class `MDGenerator` that is parameterized with an arbitrary hash algorithm. Each `SecureRandom` instance is seeded with 32 bytes yielding an internal state as shown in row i of Figure 9.4. This start value is hashed and the output gives the pseudo-random value r_0. r_0 in turn is concatenated with the former seed giving the new state in row ii. These bytes are hashed again yielding the second output r_1. Finally, the seed concatenated with the previous two hash values form the new state (cf. row iii) whose hash value is the third output value r_3. r_3 is again appended to the previous state resulting in the state illustrated in row iv.

Each fourth r_i value causes a block overflow, forcing the implementation to hash the full block and use this hash as IV for the new block. The only unknown value is the 32 byte long initial seed. All other information are known (as they are part of former r_i values).

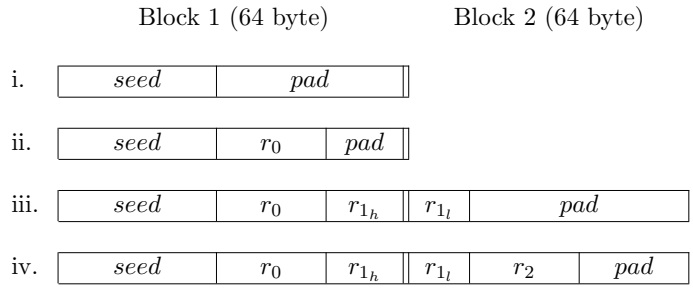

Figure 9.4.: Hashing the previous pseudo-random bytes concatenated with the fomer seed gives the next output value.

Part of the implementation is a "backup" seeding facility (algorithm in Listing 9.4) for Unix-like operating systems. The `VMSecureRandom` class is able

[5] `http://www.gnu.org/licenses/`
[6] `http://icedtea.classpath.org/wiki/Main_Page`

to harvest entropy from the OS's process scheduler. Every call to the seeding facility starts 8 threads, each one incrementing a looped one byte counter. The parent thread waits until at least one of the 8 threads is picked up by the scheduler. The seed value is calculated by XORing the counters. This gives the first seed byte. Accordingly, the next seed byte is generated the same way. When the requested amount of seeding material is gathered the threads are stopped.

```
1  int n = 0
2  byte[] S = new byte[8];
3  byte[] output = new byte[32];
4
5  for(int i = 0; i < 8; i++) {
6    S[i] = start_thread(n); // ''spinner'' incrementing a counter
         starting at n
7    n = (2*n) % pow(2,32);
8  }
9
10 while(!spinners_running())
11   wait();
12
13 for(int i = 0; i < 32; i++) {
14   output[i] = 0;
15
16   for(int j = 0; j < 8; j++)
17     output[i] = output[i] ^ counter(S[j]);
18 }
19
20 for(int i = 0; i < 8; i++)
21   stop_thread(S[i]);
22
23 return output;
```

Listing 9.4: GNU Classpath's Entropy Collector (Pseudo Code)

Weaknesses. The entropy collector revealed inconsistencies regarding normal distribution of the output bits which could result in vulnerabilities.

The `SecureRandom` implementation of GNU Classpath contains a significant weakness related to internal states: As long as a newly generated random value concatenated with the previous state does not overflow the current block, all hash computations are done with the same IV. States in rows i and ii of Figure 9.4 are both hashed with the SHA-1 standard IV. The following state in row iii overflows the first block and the hash value of the first block is used as the input for the hash computation of the second block in row iii, as well as IV for the succeeding computation of $r_{1_l}|r_2$ concatenated with r_3. The state is reduced from 32 bytes seed to only 20 unkown IV bytes.

Keeping in mind that the compression function of SHA-1, ignoring the final 32 bit additions, is invertible(cf. *Analysis of SHA-1 in Encryption Mode* [129]) for a given message, the addition of the secret IV prevents the whole algorithm from being completely broken.

Quality of entropy collectors. The seeding facility harvests entropy from multiple threads competing for CPU time. While the behaviour of the scheduler is difficult to predict it can be influenced. Only one of eight threads is expected to be scheduled, but during seed extraction this precondition is not checked.

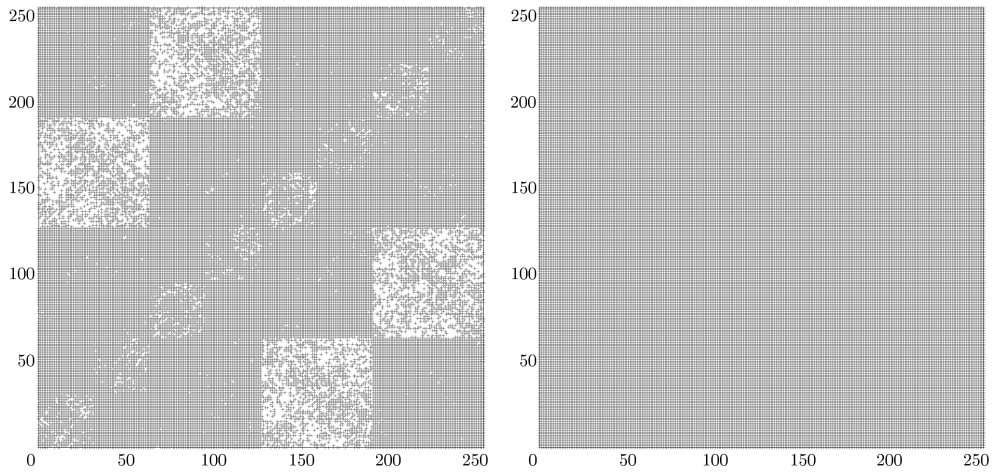

Figure 9.5.: Distribution of 2-tuples from `VMSecureRandom` under heavy and normal system workload.

This enables an attacker to fill (parts) of the output array with equal values by preventing threads to run (e.g. by creating high process load).

To test this construction under worst conditions, 11GiB of seed values were generated. To simulate high load, 8 processes were run simultaneously. Each process queried for 16384 bytes while iterating in a loop. At first inspection, the resulting random seed revealed large (up to 2800 bytes) "holes" where random bytes were equal. While the test conditions were extreme, they still expose a weakness in this entropy harvester.

The first 10MiB of seed are sampled on a graph (see Figure 9.5). As can be seen, Classpath was unable to fill the whole space, leaving 64 by 64 large patches in the second and forth quadrant, as well as 32 by 32 patches along the diagonal when running under heavy load. In contrast, under normal conditions the entropy collector produced a well-balanced pane.

OpenJDK OpenJDK is the official reference implementation of the Java Platform, Standard Edition (Java SE)[7] and open source licensed under GPL.

```
1 // require state: seed bytes, iv: SHA-1 standard IV
2 byte[] output = sha1(iv,state);
3 byte[] new_state = (state + output + 1) % pow(2,160);
4
5 if(state == new_state)
6   state = (new_state + 1) % pow(2,160);
7 else
8   state = new_state;
9
10 return output;
```

Listing 9.5: OpenJDK's SecureRandom (Pseudo Code)

[7] `http://www.oracle.com/technetwork/java/javase/overview/index.html`

OpenJDK uses the OS's (P)RNG in conjunction with a similar scheme as the previously mentioned libraries for its implementation of `SecureRandom` (algorithm in Listing 9.5). An initial 160 bit seed value is hashed using SHA-1. The result is XORed with the output of the OS specific (P)RNG and returned as pseudo-random bytes. This output is added to the initial seed plus 1 modulo 2^{160}. An additional check compares the new seed to the old one preventing the function from getting trapped in a fixed state where $s_i + \mathit{SHA\text{-}1}(s_i) + 1 \equiv s_i \bmod 2^{160}$.

The integrated entropy collector (cf. Listing 9.6) uses multiple threads to gather randomness. In contrast to GNU Classpath, only one thread increments a counter. Subsequently, new threads are started suspending five times for 50ms to keep the scheduler busy. Before continuing, the lower 8 bits of the current counter value pass an S-Box. The XOR sum of all 5 counters is returned as random byte. The entropy collector enforces mandatory runtime (250ms) and counter value (64000) to ensure proper execution. Even after enough seed is produced, the entropy collector continues to run. The seed bytes are hashed together with entries of `System.properties` and the result is used as seed for `SecureRandom`.

```
1  int counter = 0;
2  int quanta = 0;
3  int v = 0;
4
5  while(counter < 64000 && quanta < 6) {
6    start_thread()  // loops 5 times, sleeping 50ms each
7    latch = 0;
8    t = time();
9
10   while(time() - t < 250ms) // repeat for 250ms
11     latch = latch + 1;
12
13   counter = counter + latch;
14   v = v ^ SBox[latch % 255];
15   quanta = quanta + 1;
16 }
17
18 return v;
```

Listing 9.6: OpenJDK's Entropy Collector (Pseudo Code)

Weaknesses. The overall impression of `SecureRandom`'s reference implementation (expected to be used in most of today's Java applications running on server and desktop platforms), represents a thoughtful and mature implementation. Code review bared no obvious weaknesses. Still, if an attacker is able to learn any internal state (s_i) all following states $s_j \quad \forall j > i$ and outputs $o_j = \mathit{SHA\text{-}1}(s_j)$ can be predicted if the OS PRNG (*/dev/{u,}random*) is unavailable[8].

Quality of entropy collectors. OpenJDK uses a similar algorithm for seed generation in abstance of OS support like `VMSecureRandom` of GNU Classpath. The threaded entropy collector is supplemented by enforcing minimal

[8] Until the user manually reseeds the SecureRandom instance

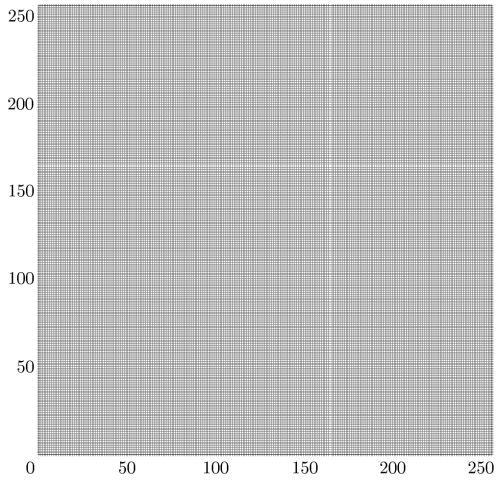

Figure 9.6.: Distribution of 2-tuples from the entropy collector of OpenJDK.

limits on runtime before extracting bytes and adding a substitution box. The unhashed[9] seed bytes were evaluated. The implementation revealed to be magnitudes slower, resulting in only 20MiB of seed generated by 8 processes running simultaneously, but anyway the resulting graph is filled in a very balanced way (cf. Figure 9.6).

The Legion of the Bouncy Castle. Bouncy Castle is not a complete core library, but a security framework for the Java platform. This includes an optional `SecurityProvider`, as well as cryptographic APIs for standalone usage. The project provides enhanced functionality and support for a broader range of cryptographic algorithms. Bouncy Castle implements various PRNGs as well as a threaded entropy collector akin to the one in (Open-) JDK.

The `DigestRandomGenerator` (see Listing 9.7) uses a cryptographic hash algorithm to generate a pseudo-random sequence by hashing an internal (secret) state of 160 bits together with a 64 bit *state counter*, producing 160 bits of pseudo-random output at once. The state counter is incremented after each hash operation. The initial secret state is received from a seeding facility. Every tenth hash operation on the state, a 64 bit *seed counter* is incremented and a new secret state is generated by hashing the current state concatenated with the seed counter. This new secret state replaces the old one, where the state counter remains at its previous value. When another pseudo-random value is requested from the `DigestRandomGenerator` instance, this new secret state is hashed with the state counter and returned to the caller as random byte array.

```
1 // require seedBuffer: 160bit seed, stateBuffer: 160bit array,
      seedCounter and stateCounter: 64bit integers
2 if(stateCounter % 10 == 0) {
3   stateBuffer = sha1(iv,concat(seedBuffer,seedCounter));
4   seedCounter += 1;
5 }
6
```

[9] Before hashing with additional `System.properties` values

```
byte[] output = sha1(iv,concat(stateBuffer,stateCounter));
stateCounter++;

return output;
```

Listing 9.7: DigestRandomGenerator (Pseudo Code)

The `VMPCRandomGenerator` is based on Bartosz Zoltak's Variable Modified Permutation Composition one-way function [130].

The `ThreadedSeedGenerator` implements a threaded entropy collector scheme. Only two threads are used: one thread increments a counter in a loop, whereas the other waits 1ms until the counter has changed. The new value is appended to an output array. The incrementing thread is teared down after all random bytes are collected. Two modes of operation are offered by the generator:

1. "slow" mode where only the least significant bit of every byte is used
2. "fast" mode where the whole byte is used

```
// require count: number of seed bytes needed, fast: enable "fast"
    mode
byte[] output = byte[count];
Thread t = start_thread() // increments a counter in a loop
int last = 0;

// use bits in "slow" mode
if(!fast)
  count *= 8;

for(int i = 0; i < count; i++) {
  while(counter(t) == last)
    sleep(1);

  last = counter(t);
  if(fast)
    output[i] = (byte)last;
  else
    output[i/8] = (last % 2) | (output[i/8] << 1);
}
stop_thread(t);

return output;
```

Listing 9.8: ThreadedSeedGenerator (Pseudo Code)

Weaknesses. Bouncy Castle's implementation is also a hash-based algorithm (`DigestRandomGenerator`). No obvious bugs were found during code review. In contrast, the entropy collector `ThreadedSeedGenerator` revealed difficulties to generate sufficient random bytes.

In `DigestRandomGenerator` the seed is modified every tenth call (cf. Listing 9.7). This may increase the period of the PRNG and hinders an attacker aware of the secret state to calculate previous outputs. Predicting all succeeding outputs is still possible as the counter values can be guessed by observing the amount of previously produced values.

The VMPC function used in `VMPCRandomGenerator` is known to be vulnerable to multiple distinguishing attacks (cf. *The Most Efficient Distinguishing Attack on VMPC and RC4A* [131], *Two Linear Distinguishing Attacks on VMPC and RC4A and Weakness of RC4 Family of Stream Ciphers (Corrected)* [132], *Improved Cryptanalysis of the VMPC Stream Cipher* [133]).

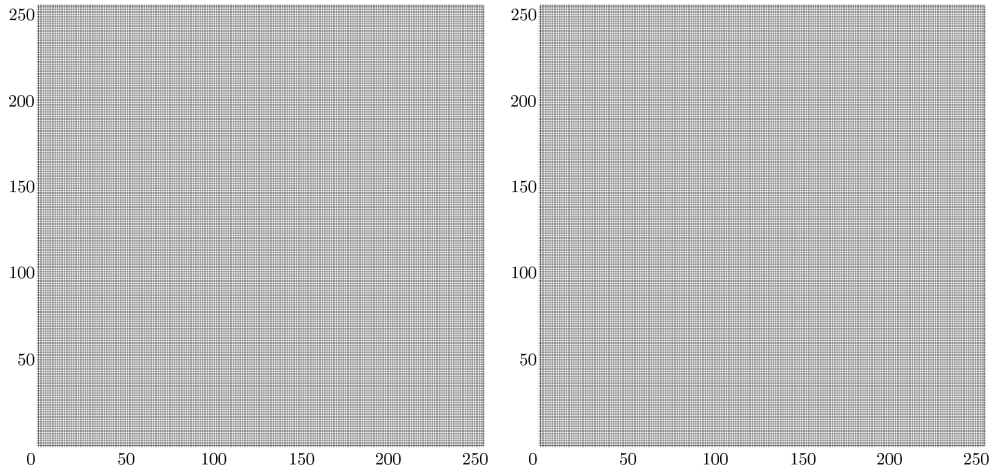

Figure 9.7.: Distribution of 2-tuples from the `ThreadedSeedGenerator` in slow and fast mode

Quality of entropy collectors. Thus, the entropy collector checks if a counter incremented in another thread has changed under heavy load the counter often differs only about 1 incrementation. Both modes were still able to fill the pane after 10MiB of one byte samples (cf. Figure 9.7).

9.2.3. Overall Impression

All inspected implementations of `SecureRandom` are bound to a limited internal state size. Alarming are the results for OpenJDK and GNU Classpath where additional entropy ($>$ 160bit) does not enhance security. The use of these libraries for the generation of *good* keys $>$ 160 bit (as e.g. in AES' case) remains therefor questionable. Only Apache Harmony relies on a 512 bit buffer, but suffers on the other hand from multiple striking weaknesses.

"Distrust and caution are the parents of security."
Benjamin Franklin

10 Future work

SSL/TLS provides a wide range of research opportunities for future work. Ranging from cryptography improvements and attacks on implementation pitfalls to the need for adaption of the newest specification revisions.

10.1. General

10.1.1. Cryptography

Since RC4 and block-ciphers operating in CBC mode are no longer considered as secure as believed (see Chapter 5) and even ECC suffers from distrust, since NSA was invloved in the specification (the security of ECC remains questionable regarding the standard sabotaged by the NSA[1] leading to a backdoor in an ECC based PRNG[2]), there is an urgent need for replacements. This stresses the importance of research in these areas and at the same time requires research on new algorithms and operation modes. Furthermore, from a theoretical point of view, it is advantageous to have security proofs for both, the cryptography and its usecases in SSL/TLS in realistic and practice related models.

10.1.2. Implementation

Many weaknesses are theoretically non-existent, but software still suffers from such vulnerabilities. This highlights the gap between theory and practice. Algorithms are only as strong as the cryptographic background **and** the implementation that maps the right assumptions flawlessly into code. If the code contains flaws or side channels, vulnerabilities may arise. Sometimes, this gap

[1] `http://nyti.ms/15hYlK1`
[2] `http://en.wikipedia.org/wiki/Dual_EC_DRBG`, `http://rump2007.cr.yp.to/15-shumow.pdf`

is caused by developer mistakes and sometimes the gap is caused by assumptions that cannot be mapped from theory into practice. As a consequence, both – theory and practice – have to work hand in hand to improve algorithms and implementations.

To give an example, Listing 10.1 shows Java's Optimal Asymmetric Encryption Padding (OAEP) processing function (`sun.security.rsa.RSAPadding.java`, see RFC 3447 [30] for details on the scheme).

```
1 private byte[] unpadOAEP(byte[] padded) throws
      BadPaddingException {
2     byte[] EM = padded;
3     int hLen = lHash.length;
4
5     if (EM[0] != 0) {
6         throw new BadPaddingException("Data must start with zero"
              );
7     }
8
9     int seedStart = 1;
10    int seedLen = hLen;
11
12    int dbStart = hLen + 1;
13    int dbLen = EM.length - dbStart;
14
15    mgf1(EM, dbStart, dbLen, EM, seedStart, seedLen);
16    mgf1(EM, seedStart, seedLen, EM, dbStart, dbLen);
17
18    // verify lHash == lHash'
19    for (int i = 0; i < hLen; i++) {
20        if (lHash[i] != EM[dbStart + i]) {
21            throw new BadPaddingException("lHash mismatch");
22        }
23    }
24
25    // skip over padding (0x00 bytes)
26    int i = dbStart + hLen;
27    while (EM[i] == 0) {
28        i++;
29        if (i >= EM.length) {
30            throw new BadPaddingException("Padding string not
                  terminated");
31        }
32    }
33
34    if (EM[i++] != 1) {
35        throw new BadPaddingException
36            ("Padding string not terminated by 0x01 byte");
37    }
38
39    int mLen = EM.length - i;
40    byte[] m = new byte[mLen];
41    System.arraycopy(EM, i, m, 0, mLen);
42
43    return m;
44 }
```

Listing 10.1: OAEP unpadding function of Java 7

Many problems related to the *old* PKCS#1 v1.5 are supposed to disappear with the introduction of OAEP. But, the code already reveals a new side channel. Lines 5-7 outline a conditional branch that could be used to apply an attack described by Manger in 2001 [134]. Further research and patching is needed here. This example shows that OAEP only helps if it is implemented correctly, i.e. without side channels.

10.1.3. Attacks

SSL/TLS has a long history, full of improvements and – on the other hand – attacks. This field is an ongoing area of research, because every improvement motivates for new or improved attacks and attack techniques. Future attacks will unoubtly focus on each possible attack vector (cryptography, interplay with other protocols, timings, implementation faults....). For example, the attacks on RC4 and timing based attacks are new and perhaps these attacks only scratch the surface, revealing the true potential only after intensive research. The currently most discussed field of research is perfect forward secrecy where no promising attacks are known yet.

10.1.4. Switching to Newest Revisions

According to SSLPulse[3] (see Figure 10.1, data of September 02, 2013) only 17.8% of the monitored sites offer TLS 1.2 support. An even more frigthening fact is that 26.9% of the monitored sites still offer support for the insecure SSL 2.0 protocol.

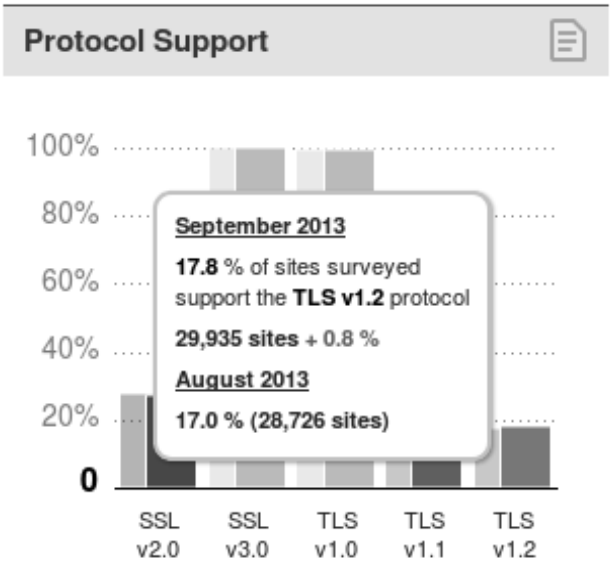

Figure 10.1.: SSL/TLS version support of monitored sites – *Source: SSLPulse*, data of September 02, 2013

[3] `https://www.trustworthyinternet.org/ssl-pulse/`

Additionally, the number of client software (browsers and libraries) supporting TLS 1.2 is not satisfying. As an example: neither Java's JSSE or the widely used Firefox browser (with a market share of 28% at the time of writing[4]) do not support TLS 1.2, yet.

Anyone relying on the security of SSL/TLS 1.2 is hereby highly encouraged to switch to the newest revision 1.2, since it counters many (not all) of the known attacks by design. Because of this, moving on to migrate from older SSL/TLS versions to the newest revision is definitely a remaining future task.

10.2. Specific to this Work

10.2.1. SSL Analyzer

As already mentioned, the SSL Analyzer introduced in Section 6.6 is still a proof of concept application and lacks for a comprehensive fingerprint database containing a significant number of different stacks, devices and patch levels. Therefore, it is necessary to collect more fingerprints over time.

Additionally, there are many other scenarios resulting in special behaviour that can be useful for the creation of fingerprints. The test suite has to be expanded and subsequently applied to create fingerprints. With an increasing number of fingerprints and tests, the scan results will at the same time be significantly more accurate and fine-grained.

10.2.2. T.I.M.E. Framework

The T.I.M.E. framework is currently limited to client side inspection only, but when extended with server capabilities it would also be possible to penetrate clients. As an example, the SSL Analyzer could easily be adjusted to fingerprint client instead of server behaviour. As a conclusion, it would be beneficial to add server functionality to the framework. Beyond this, adding support for more cipher suites is planned. This enhancement is not that complicated and would allow for a more flexible inspection of SSL/TLS connections.

A major adjustment would be to support more SSL/TLS revisions. At the time of writing, the framework supports only TLS 1.0. However, different revisons come along with major changes to the basic message and data structures, as well as changes to message processing and used algorithms. Supporting different revisons requires a major change in the framework's source code, but is definitely worth the effort. Adding support for at least all TLS versions is a future goal.

[4] Accoording to `http://www.w3schools.com/browsers/browsers_stats.asp`

"The gap between theory and practice is not as wide in theory as it is in practice."

Unknown Author, maybe thinking about the Random Oracle Model

11 Conclusion

SSL/TLS is the defintive foundation of secure communication over the internet at this time. The protocol suffers from some shortcomings leading to vulnerabilities, but provides – at least in the latest revision 1.2 – very good security with regards to the security goals confidentiality, integrity and optional authentication. Remaining challenges for the future are on the one hand reliable and future proof new algorithms with ideally proven security (with realistic and suitable assumptions) and on the other hand bug free and specification conform implementations. It remains questionable if both of these goals may ever be achieved by 100%. As seen in the previous chapters, security is not unbreakable and weaknesses can occur at various places, caused by various reasons. Confidentially, integrity and authentication are important security goals worth to be protected. These goals are essential for a reliable network that is used for various purposes.

In these times, when privacy is becoming increasingly important, it is also indispensable to provide the best possible protection, even for seemingly unimportant data. Every piece of information can be of relevance depending on what you are looking for. As a consequence, a development towards mandatory encrypted and integrity protected network traffic may be advantageous.

Another point to highlight is the dependence on an unflawed and trustworthy PKI. If the trust anchors can be manipulated the best efforts on enhancing and hardening TLS are futile. Therefore, advances in either monitoring the CAs or the whole attestation process are necessary.

The fingerprinting technique presented in Chapter 7 is still in the proof-of-concept phase, but gives amazing precise results when trying to identify the SSL/TLS stack running on a target machine. The tool is certainly helpful for statistical analysis, as well as for detailed and particular targeted analysis of stack behaviour. Some of the weaknesses presented in Chapter 5 and Chapter 8 were found during evaluation of the SSL Analyzer.

The lessons learned during research can after all be summarized in a few, but very important principles for more secure and reliable specifications and software:

1. Theoretical attacks can turn into practice
2. Unexploitability may not last forever.
3. Side channels may appear at different layers in different situations
4. Reliable cryptographic primitives are important
5. Processes must leak as little information as possible
6. Specifications have to be implemented without own improvements
7. Critical parts in specifications and source code have to be highlighted
8. Specifications have to be verbose, unambiguous and technically detailed
9. Details on requirements and preconditions are necessary
10. Data has to be protected (authenticated, integrity ensured, encrypted, etc.)
11. The interplay between different layers must be part of the security analysis
12. Flexibility mostly means additional risks
13. Always be careful and alarmed

Two additional – *must reads* – for anyone interested in creating better software and protocols are the following: *Lessons Learned in Implementing and Deploying Crypto Software* [135] by Peter Gutman and *Robustness principles for public key protocols* [136] by Ross Anderson and Roger Needham.

Finally, a closing adivce from the author of this thesis:

> *Scrutinize all and everything at any time and check for correctness – things often look correct at first sight, but reveal to be different!*

A Appendix

A.1. Basic Notation

A.1.1. RSA

The following notation is used for RSA operations and the required parameters.

Symbol	Meaning
C	Ciphertext
P	Plaintext
S	Signature
e	RSA public exponent
d	RSA private exponent
n	RSA modulus

Table A.1.: Formal notation of RSA encryption/signature

The basic RSA operations are defined in Equations A.1, A.2, A.3, A.4.
Encryption:

$$C \equiv P^e \mod n \tag{A.1}$$

Decryption:

$$P \equiv C^d \mod n \tag{A.2}$$

Signature of some data X:

$$S \equiv X^d \mod n \tag{A.3}$$

Verification of a Signature over some data X:

$$X \stackrel{?}{\equiv} S^e \mod n \tag{A.4}$$

Block Type	Padding	Separation Byte	Encapsulated Data
00 02	...	00	*PreMasterSecret*

Table A.2.: PKCS#1 v 1.5 encoded *PreMasterSecret*

A.1.2. PKCS#1 v1.5

This Subsection introduces the PKCS#1 v1.5 format (specified in RFC 2313 [65]). As an example application, the RSA encryption of the *PreMasterSecret* in SSL/TLS is considered. The payload, in this case the *PreMasterSecret*, is encapsulated in a strictly specified structure which is subsequently encrypted, in this example with RSA. The structure is illustrated in Table A.2 and contains at first a block type of fixed length (always 2 bytes), a padding string which is of variable length but requires at minimum 8 non-zero bytes (all bytes of the padding string are required to be non-zero), a single zero padding byte which separates the padding from the following encapsulated payload data.

It follows the description of each part of the structure illustrated in Table A.2. The block type identifies the usage type – what kind of data is wrapped:

- 00 01 : Wrapped data is the result of a private key operation – e.g. signature data
- 00 02 : Wrapped data is the result of a public key operation – e.g. encrypted data

The integrated (pseudo-random) padding aims at obfuscating the data and hiding its real content, beyond the fact that the padding adjusts the length of this structure to be of size of the RSA modulus. Thus, each wrapped data will look different each time, because of the pseudo-random padding.

The separation byte indicates the end of the padding data (which is of variable length) and the start of the payload data. The encapsulated data contains, in this case, the plain *PreMasterSecret* which is of a fixed length.

Table A.3 introduces the notation used in the following.

EB is converted into an integer and encrypted using the RSA scheme ($CB = int(EB)^e \mod n$, e is the public exponent, CB is the resulting ciphertext). Following the previously discussed structure, an EB consisting of multiple bytes $[EB_0, EB_1..., EB_{\#EB}]$ remains PKCS compliant if the structure fulfils the conditions of Table A.4.

A valid PKCS#1 v1.5 structure is graphically illustrated in Figure A.1.

Figure A.1.: Valid PKCS structure

Symbol	Meaning	Notes
n	RSA Modulus	$n = p * q, p, q \in \mathbb{P}$
$\#n$	byte length of n	$2^{8(k-1)} \leq n < 2^{8k}$
D	Data block	
$\#D$	byte length of D	$\#D \leq \#n-11$ (min. 8 padding bytes + 2 bytes block type + 1 separation byte = 11 bytes)
P	Padding block, non-zero random bytes	
$\#P$	byte length of P	$\#P = \#n - \#D - 3, \#P \geq 8$ (2 bytes block type + 1 separation byte = 3 bytes)
EB	Encryption block	$EB = 00\|02\|P\|00\|D$
$\#EB$	byte length of EB	

Table A.3.: Formal notation used for PKCS#1 v1.5 definition

Byte	Value
EB_0	00
EB_1	02
$EB_3..EB_{10}$	$\neq 00$
$EB_{11}..EB_{\#EB}$	$\exists!00$

Table A.4.: Valid PKCS structure

A.1.3. CBC Mode

The CBC mode of operation is illustrated in Figure A.2.

A ciphertext of a block cipher consists of a variable number l of blocks: $C = [C_0, ...C_l]$. The same holds for the plaintext: $P = [P_0, ...P_l]$, where mostly the first block contains an IV ($C_0 = IV$) otherwise the IV must be obtained from a different source, because it is essential for the encryption and decryption process. For the formalization it is assumed that the length of each block is of length of the block-ciphers internal block length and each C_i/P_i denote a block of this length. In CBC mode these blocks are chained together so that a manipulation of a block always has an effect on other blocks. The chaining of the encrypt and decrypt operations can be formalized as follows:

Encryption

$$C_i = Enc_{Key}(P_i \oplus C_{i-1}) \tag{A.5}$$

If C_0 contains the (unencrypted) IV the first plaintext block starts a t index $i = 1$ (P_1).

Decryption

$$P_i = Dec_{Key}(C_i) \oplus C_{i-1} \tag{A.6}$$

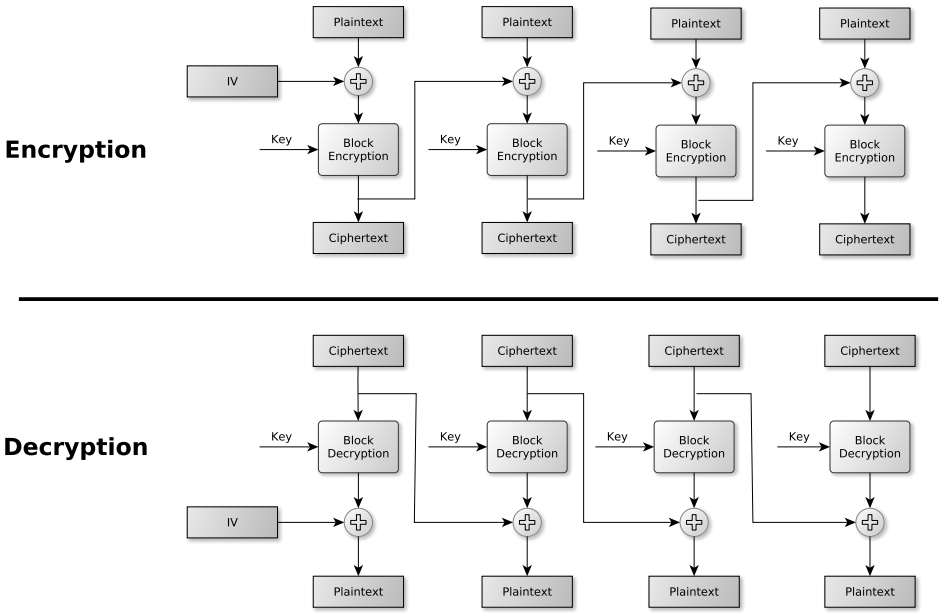

Figure A.2.: Encryption/Decryption in CBC Mode

B
Bibliography

[1] National Institute of Standards and Technology, "Federal Information Processing Standards Publication (FIPS 197), Advanced Encryption Standard (AES)," US Department of Commerce, Tech. Rep. FIPS PUB 197, Nov. 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[2] P. Rogaway, "Authenticated-Encryption with Associated-Data," in *Proceedings of the 9th ACM conference on Computer and communications security*, ser. CCS '02. ACM, Nov. 2002, pp. 98–107.

[3] ISO, *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Dec. 2011.

[4] M. Nystrom and B. Kaliski, "PKCS #10: Certification Request Syntax Specification Version 1.7," RFC 2986 (Informational), Internet Engineering Task Force, Nov. 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2986.txt

[5] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 5th ed. CRC Press, Inc., 2001.

[6] W. F. Ehrsam, C. H. W. Meyer, J. L. Smith, and W. L. Tuchman, "Message verification and transmission error detection by block chaining," Patent US4 074 066 A, Feb., 1978.

[7] National Institute of Standards and Technology, "Recommendations for Cipher Modes of Operation: CCM Mode for Authenticity and Confidentiality," US Department of Commerce, Tech. Rep., May 2004. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38C.pdf

[8] W. W. Peterson and D. T. Brown, "Cyclic Codes for Error Detection," in *Proceedings of the IRE*, vol. 49, Jan. 1961, pp. 228–235.

[9] National Institute of Standards and Technology, "Federal Information Processing Standards Publication (FIPS PUB 46-3), Data Encryption Standard (DES)," US Department of Commerce, Tech. Rep. FIPS PUB 46, Oct. 1999. [Online]. Available: http://www.itl.nist.gov/fipspubs/fips46-3.pdf

[10] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security," RFC 4347 (Proposed Standard), Internet Engineering Task Force, Apr. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4347.txt

[11] ——, "Datagram Transport Layer Security Version 1.2," RFC 6347 (Proposed Standard), Internet Engineering Task Force, Jan. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6347.txt

[12] M. Handley, E. Rescorla, and IAB, "Internet Denial-of-Service Considerations," RFC 4732 (Informational), Internet Engineering Task Force, Dec. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4732.txt

[13] E. Rescorla, "Diffie-Hellman Key Agreement Method," RFC 2631 (Proposed Standard), Internet Engineering Task Force, Jun. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2631.txt

[14] B. W. Diffie, M. E. Hellman, and R. C. Merkle, "Cryptographic apparatus and method," Patent US4 200 770 A, Apr., 1980.

[15] National Institute of Standards and Technology, "Federal Information Processing Standards Publication (FIPS 186-4), Digital Signature Standard (DSS)," US Department of Commerce, Tech. Rep. FIPS PUB 186, Jul. 2013. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

[16] M. D., K. Igoe, and S. M., "Fundamental Elliptic Curve Cryptography Algorithms," RFC 6090 (Informational), Internet Engineering Task Force, Feb. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6090.txt

[17] American National Standards Institute and American Bankers Association, *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA).* American Bankers Association, 1999.

[18] National Institute of Standards and Technology, "Recommendations for Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," US Department of Commerce, Tech. Rep., Nov. 2007. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf

[19] ——, "Federal Information Processing Standards Publication (FIPS 198), The Keyed-Hash Message Authentication Code (HMAC)," US Department of Commerce, Tech. Rep. FIPS PUB 198, Jul. 2008. [Online]. Available: http://csrc.nist.gov/publications/fips/fips198/fips-198.pdf

[20] T. Berners-Lee, R. Fielding, and H. Frysty, "Hypertext Transfer Protocol – HTTP/1.0," RFC 1945 (Informational), Internet Engineering Task Force, May 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1945.txt

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616 (Draft Standard), Internet Engineering Task Force, Jun. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2616.txt

[22] X. Lai and J. L. Massey, "A Proposal for a New Block Encryption Standard," in *Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology*, ser. EUROCRYPT '90. Springer-Verlag New York, Inc., 1991, pp. 389–404.

[23] M. Crispin, "Internet Message Access Protocol - Version 4rev1," RFC 3501 (Proposed Standard), Internet Engineering Task Force, Mar. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3501.txt

[24] National Institute of Standards and Technology, "Federal Information Processing Standards Publication (FIPS PUB 113), Computer Data Authentication," US Department of Commerce, Tech. Rep. FIPS PUB 113, May 1985.

[25] B. Kaliski, "The MD2 Message-Digest Algorithm," RFC 1319 (Historic), Internet Engineering Task Force, Apr. 1992. [Online]. Available: http://www.ietf.org/rfc/rfc1319.txt

[26] R. Rivest, "The MD4 Message-Digest Algorithm," RFC 1320 (Historic), Internet Engineering Task Force, Apr. 1992. [Online]. Available: http://www.ietf.org/rfc/rfc1320.txt

[27] ——, "The MD5 Message-Digest Algorithm," RFC 1321 (Informational), Internet Engineering Task Force, Apr. 1992. [Online]. Available: http://www.ietf.org/rfc/rfc1321.txt

[28] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP," RFC 2560 (Proposed Standard), Internet Engineering Task Force, Jun. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2560.txt

[29] A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed. Prentice Hall Press, 2007.

[30] J. Jonsson and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1," RFC 3447 (Informational), Internet Engineering Task Force, Feb. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3447.txt

[31] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[32] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 5280 (Proposed Standard), Internet Engineering Task Force, May 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5280.txt

[33] L. M. Adleman, R. L. Rivest, and A. Shamir, "Cryptographic communications system and method," Patent US4 405 829 A, Sep., 1983.

[34] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, 1978.

[35] R. Rivest, "A Description of the RC2(r) Encryption Algorithm," RFC 2268 (Informational), Internet Engineering Task Force, Mar. 1998. [Online]. Available: http://www.ietf.org/rfc/rfc2268.txt

[36] ——, "The RC5 Encryption Algorithm," in *Fast Software Encryption*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1995, vol. 1008, pp. 86–96.

[37] R. L. Rivest, M. Robshaw, R. Sidney, and Y. Yin, "The RC6 Block Cipher," 1998.

[38] National Institute of Standards and Technology, "Federal Information Processing Standards Publication (FIPS 180-2), Secure Hash Standard," US Department of Commerce, Tech. Rep. FIPS PUB 180, Aug. 2002. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf

[39] K. E. Hickman, "The SSL Protocol," Internet Draft, Internet Engineering Task Force, Apr. 1995. [Online]. Available: http://tools.ietf.org/html/draft-hickman-netscape-ssl-00

[40] A. Freier, P. Karlton, and P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0," RFC 6101 (Historic), Internet Engineering Task Force, Aug. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6101.txt

[41] J. Postel, "Domain Name System Structure and Delegation," RFC 1591 (Informational), Internet Engineering Task Force, Mar. 1994. [Online]. Available: http://www.ietf.org/rfc/rfc1591.txt

[42] ——, "Transmission Control Protocol," RFC 793 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 1981. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt

[43] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," RFC 2246 (Proposed Standard), Internet Engineering Task Force, Jan. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2246.txt

[44] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," RFC 4346 (Proposed Standard), Internet Engineering Task Force, Apr. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4346.txt

[45] ——, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[46] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax," RFC 3986 (Internet Standard), Internet Engineering Task Force, Jan. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc3986.txt

[47] T. Berners-Lee, L. Masinter, and M. McCahill, "Uniform Resource Locators (URL)," RFC 1738 (Proposed Standard), Internet Engineering Task Force, Dec. 1994. [Online]. Available: http://www.ietf.org/rfc/rfc1738.txt

[48] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM*, vol. 17, pp. 365–375, 1974.

[49] J. Postel, "User Datagram Protocol," RFC 768 (INTERNET STANDARD), Internet Engineering Task Force, Aug. 1980. [Online]. Available: http://www.ietf.org/rfc/rfc768.txt

[50] A. Tanenbaum and A. Todd, *Computer Networks*, 6th ed. Prentice Hall Professional Technical Reference, 2012.

[51] A. Adelsbach, S. Gajek, and J. Schwenk, "Visual Spoofing of SSL Protected Web Sites and Effective Countermeasures," in *Information Security Practice and Experience*, ser. Lecture Notes in Computer Science, R. Deng, F. Bao, H. Pang, and J. Zhou, Eds. Springer Berlin Heidelberg, 2005, vol. 3439, pp. 204–216.

[52] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor, "Crying wolf: an empirical study of SSL warning effectiveness," in *Proceedings of the 18th conference on USENIX security symposium*, ser. SSYM'09. USENIX Association, 2009, pp. 399–416.

[53] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov, "Transport Layer Security (TLS) Renegotiation Indication Extension," RFC 2246 (Proposed Standard), Network Working Group, Nov. 2009. [Online]. Available: http://tools.ietf.org/id/draft-rescorla-tls-renegotiation-01.txt

[54] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright, "Transport Layer Security (TLS) Extensions," RFC 3546 (Proposed Standard), Internet Engineering Task Force, June 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3546.txt

[55] ——, "Transport Layer Security (TLS) Extensions," RFC 4366 (Proposed Standard), Internet Engineering Task Force, April 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4366.txt

[56] W. Diffie and M. Hellman, "Special Feature Exhaustive Cryptanalysis of the NBS Data Encryption Standard," *Computer Magazine*, vol. 10, no. 6, pp. 74–84, Jun. 1977.

[57] S. Turner and T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0," RFC 6176 (Proposed Standard), Internet Engineering Task Force, Mar. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6176.txt

[58] C. Meyer and J. Schwenk, "SoK: Lessons Learned From SSL/TLS Attacks," in *Proceedings of the International Workshop on Information Security Applications (WISA2013)*, ser. WISA '13. Springer-Verlag, Aug. 2013.

[59] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk, "On the Security of TLS-DHE in the Standard Model," in *Advances in Cryptology – CRYPTO 2012*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7417, pp. 273–293.

[60] H. Krawczyk, K. Paterson, and H. Wee, "On the Security of the TLS Protocol: A Systematic Analysis," in *Advances in Cryptology – CRYPTO 2013*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 8042, pp. 429–448.

[61] J.-S. Coron, M. Joye, D. Naccache, and P. Paillier, "New attacks on PKCS#1 v1.5 encryption," in *Proceedings of the 19th international conference on Theory and application of cryptographic techniques*, ser. EUROCRYPT'00. Berlin, Heidelberg: Springer-Verlag, May 2000, pp. 369–381.

[62] J. Jonsson and B. S. Kaliski, Jr., "On the Security of RSA Encryption in TLS," in *Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '02. Springer-Verlag, Aug. 2002, pp. 127–142.

[63] E. Kiltz and K. Pietrzak, "On the Security of Padding-Based Encryption Schemes – or – Why We Cannot Prove OAEP Secure in the Standard Model," in *Advances in Cryptology - EUROCRYPT 2009*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Apr. 2009, vol. 5479, pp. 389–406.

[64] E. Kiltz, A. O'Neill, and A. Smith, "Instantiability of RSA-OAEP under Chosen-Plaintext Attack," in *Advances in Cryptology – CRYPTO 2010*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Aug. 2010, vol. 6223, pp. 295–313.

[65] B. Kaliski, "PKCS #1: RSA Encryption Version 1.5," RFC 2313 (Informational), Internet Engineering Task Force, Mar. 1998. [Online]. Available: http://www.ietf.org/rfc/rfc2313.txt

[66] H. Krawczyk, "The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?)," in *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '01. Springer-Verlag, Aug. 2001, pp. 310–331.

[67] J. C. Mitchell, "Finite-State Analysis of Security Protocols," in *Proceedings of the 10th International Conference on Computer Aided Verification*, ser. CAV '98. Springer-Verlag, Jun. 1998, pp. 71–76.

[68] K. Ogata and K. Futatsugi, "Equational Approach to Formal Analysis of TLS," in *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS '05. IEEE Computer Society, Jun. 2005, pp. 795–804.

[69] P. Morrissey, N. P. Smart, and B. Warinschi, "A Modular Security Analysis of the TLS Handshake Protocol," in *Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ser. ASIACRYPT '08. Springer-Verlag, Dec. 2008, pp. 55–73.

[70] C. He, M. Sundararajan, A. Datta, A. Derek, and J. C. Mitchell, "A modular correctness proof of IEEE 802.11i and TLS," in *Proceedings of the 12th ACM conference on Computer and communications security*, ser. CCS '05. ACM, Nov. 2005, pp. 2–15.

[71] L. C. Paulson, "Inductive analysis of the Internet protocol TLS," *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 3, pp. 332–351, Aug. 1999.

[72] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu, "Cryptographically verified implementations for TLS," in *Proceedings of the 15th ACM conference on Computer and communications security*, ser. CCS '08. ACM, Oct. 2008, pp. 459–468.

[73] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub, "Implementing TLS with Verified Cryptographic Security," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. IEEE Computer Society, May 2013, pp. 445–459.

[74] S. Chaki and A. Datta, "ASPIER: An Automated Framework for Verifying Security Protocol Implementations," in *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, ser. CSF '09. IEEE Computer Society, Jul. 2009, pp. 172–185.

[75] M. Bellare and C. Namprempre, "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm," *J. Cryptol.*, vol. 21, no. 4, pp. 469–491, Sep. 2008.

[76] Goldberg and Wagner, "Randomness and the Netscape browser," *Dr. Dobb's Journal*, Jan. 1996.

[77] D. Wagner and B. Schneier, "Analysis of the SSL 3.0 Protocol," *The Second USENIX Workshop on Electronic Commerce Proceedings*, pp. 29–40, 1996.

[78] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1," in *Advances in Cryptology — CRYPTO '98*, ser. Lecture Notes in Computer Science, H. Krawczyk, Ed. Springer Berlin / Heidelberg, 1998, vol. 1462, pp. 1–12.

[79] G. Davida, "Chosen Signature Cryptanalysis of the RSA (MIT)Public Key Cryptosystem," Dept. of EECS, University of Wisconsin, Tech. Rep., 1982.

[80] S. Vaudenay, "Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS..." in *Advances in Cryptology — EUROCRYPT 2002*, ser. Lecture Notes in Computer Science, L. Knudsen, Ed. Springer Berlin / Heidelberg, Apr. 2002, vol. 2332, pp. 534–545.

[81] R. Housley, "Cryptographic Message Syntax (CMS)," RFC 5652 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5652.txt

[82] M. Bellare, A. Boldyreva, L. Knudsen, and C. Namprempre, "On-Line Ciphers and the Hash-CBC constructions," in *Advances in Cryptology - CRYPTO 2000. Lecture Notes in Computer Science*. Springer-Verlag, Aug. 2001, pp. 292–309.

[83] J. Kelsey, "Compression and Information Leakage of Plaintext," in *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers*, ser. Lecture Notes in Computer Science, vol. 2365. Springer, Nov. 2002, pp. 263–276.

[84] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, Jun. 2003, pp. 1–1.

[85] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '96. London, UK, UK: Springer-Verlag, Aug. 1996, pp. 104–113.

[86] O. Aciicmez, W. Schindler, and C. Koç, "Improving Brumley and Boneh timing attack on unprotected SSL implementations," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, Nov. 2005, pp. 139–146.

[87] B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux, "Password Interception in a SSL/TLS Channel," in *Advances in Cryptology - CRYPTO 2003*, ser. Lecture Notes in Computer Science, D. Boneh, Ed. Springer Berlin / Heidelberg, Aug. 2003, vol. 2729, pp. 583–599.

[88] V. Klíma, O. Pokorný, and T. Rosa, "Attacking RSA-Based Sessions in SSL/TLS," in *Cryptographic Hardware and Embedded Systems - CHES 2003*, ser. Lecture Notes in Computer Science, C. Walter, c. Koç, and C. Paar, Eds. Springer Berlin / Heidelberg, Sep. 2003, vol. 2779, pp. 426–440.

[89] G. V. Bard, "The Vulnerability of SSL to Chosen Plaintext Attack." *IACR Cryptology ePrint Archive*, vol. 2004, p. 111, May 2004.

[90] A. Lenstra, X. Wang, and B. de Weger, "Colliding X.509 Certificates," Cryptology ePrint Archive, Report 2005/067, Mar. 2005.

[91] G. V. Bard, "A Challenging But Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL," in *SECRYPT 2006, Proceedings of the International Conference on Security and Cryptography, Setúbal*. INSTICC Press, Aug. 2006, pp. 7–10.

[92] M. Rosenfeld, "Internet Explorer SSL Vulnerability," May 2008. [Online]. Available: http://www.thoughtcrime.org/ie-ssl-chain.txt

[93] F. Weimer, "DSA-1571-1 openssl – predictable random number generator," Network Working Group, May 2008. [Online]. Available: http://lists.debian.org/debian-security-announce/2008/msg00152.html

[94] G. Danezis, "Traffic Analysis of the HTTP Protocol over TLS," unpublished manuscript.

[95] Q. Sun, D. R. Simon, Y.-M. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu, "Statistical Identification of Encrypted Web Browsing Traffic," in *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, ser. SP '02. IEEE Computer Society, May 2002, pp. 19–.

[96] M. Ray and S. Dispensa, "Renegotiating TLS," PhoneFactor, Inc., Tech. Rep., Nov. 2009.

[97] M. Rosenfeld, "Null Prefix Attacks Against SSL/TLS Certificates," Feb. 2009. [Online]. Available: http://www.thoughtcrime.org/papers/null-prefix-attacks.pdf

[98] ——, "Defeating OCSP With The Character '3'," Jul. 2009. [Online]. Available: http://www.thoughtcrime.org/papers/ocsp-attack.pdf

[99] R. Moore, "Multiple Browser Wildcard Certificate Validation Weakness," Westpoint Security Advisory, Jul. 2010.

[100] E. Rescorla, "HTTP Over TLS," RFC 2818 (Informational), Internet Engineering Task Force, May 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2818.txt

[101] Comodo CA Ltd., "Comodo Report of Incident - Comodo detected and thwarted an intrusion on 26-MAR-2011," Comodo CA Ltd., Tech. Rep., Mar. 2011.

[102] J. Rizzo and T. Duong, "Here Come The ⊕ Ninjas," May 2011.

[103] Fox-IT, "Black Tulip - Report of the investigation into the DigiNotar Certificate Authority breach," Fox-IT, Tech. Rep., Aug. 2012.

[104] B. Brumley and N. Tuveri, "Remote Timing Attacks Are Still Practical," in *Computer Security - ESORICS 2011*, ser. Lecture Notes in Computer Science, V. Atluri and C. Diaz, Eds. Springer Berlin / Heidelberg, Sep. 2011, vol. 6879, pp. 355–371.

[105] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *MC*, no. 48, 1987.

[106] J. López and R. Dahab, "Fast Multiplication on Elliptic Curves Over GF(2m) without precomputation," in *Cryptographic Hardware and Embedded Systems*, ser. Lecture Notes in Computer Science, c. Koç and C. Paar, Eds. Springer Berlin / Heidelberg, 1999, vol. 1717, pp. 724–724.

[107] N. A. Howgrave-Graham and N. P. Smart, "Lattice Attacks on Digital Signature Schemes," *Designs, Codes and Cryptography*, vol. 23, pp. 283–290, 2001.

[108] K. G. Paterson, T. Ristenpart, and T. Shrimpton, "Tag size does matter: attacks and proofs for the TLS record protocol," in *Proceedings of the 17th international conference on The Theory and Application of Cryptology and Information Security*, ser. ASIACRYPT'11. Berlin, Heidelberg: Springer-Verlag, Dec. 2011, pp. 372–389.

[109] D. E. 3rd, "Transport Layer Security (TLS) Extensions: Extension Definitions," RFC 6066 (Proposed Standard), Internet Engineering Task Force, Jan. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6066.txt

[110] N. AlFardan and K. Paterson, "Plaintext-Recovery Attacks Against Datagram TLS," in *Network and Distributed System Security Symposium (NDSS 2012)*, Feb. 2012.

[111] R. Seggelmann, M. Tuexen, and M. Williams, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension," RFC (Draft), Internet Engineering Task Force, Dec. 2011.

[112] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J.-K. Tsay, "Efficient Padding Oracle Attacks on Cryptographic Hardware," INRIA, Rapport de recherche RR-7944, Apr. 2012. [Online]. Available: http://hal.inria.fr/hal-00691958

[113] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software," in *ACM Conference on Computer and Communications Security*, 2012.

[114] T. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," RFC 1951 (Proposed Standard), Internet Engineering Task Force, May 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1951.txt

[115] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel, "A Cross-Protocol Attack on the TLS Protocol," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. ACM, Oct. 2012, pp. 62–72.

[116] N. AlFardan and K. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. IEEE Computer Society, Feb. 2013.

[117] S. R. Fluhrer, I. Mantin, and A. Shamir, "Weaknesses in the Key Scheduling Algorithm of RC4," in *Revised Papers from the 8th Annual International Workshop on Selected Areas in Cryptography*, ser. SAC '01. Springer-Verlag, Aug. 2001.

[118] N. AlFardan, D. Bernstein, K. Paterson, B. Poettering, and J. Schuldt, "On the security of RC4 in TLS," in *Proceedings of the 22th USENIX Security Symposium*, Aug. 2013.

[119] Y. W. Takanori Isobe, Toshihiro Ohigashi and M. Morii, "Full Plaintext Recovery Attack on Broadcast RC4," in *Proc. the 20th International Workshop on Fast Software Encryption (FSE 2013)*, Mar. 2013.

[120] T. Be'ery and A. Shulman, "A Perfect CRIME? Only TIME Will Tell," Black Hat Europe Conference 2013, Tech. Rep., Mar. 2013.

[121] Y. Gluck, N. Harris, and A. Prado, "BREACH: Reviving the CRIME Attack," Black Hat Conference 2013, Tech. Rep., Aug. 2013.

[122] E. Freeman, E. Freeman, B. Bates, and K. Sierra, *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.

[123] B. Amann, M. Vallentin, S. Hall, and R. Sommer, "Revisiting SSL: A large-scale study of the internet's most trusted protocol," Technical report, ICSI, Tech. Rep., 2012.

[124] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert1, J. Dray, S. Vo, and L. E. B. III, "A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications," http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf, National Institute of Standards and Technology (NIST), Tech. Rep., Apr. 2010.

[125] K. Michaelis, C. Meyer, and J. Schwenk, "Randomly Failed! The State of Randomness in Current Java Implementations," in *Topics in Cryptology – CT-RSA 2013*, ser. Lecture Notes in Computer Science, E. Dawson, Ed. Springer Berlin Heidelberg, Feb. 2013, vol. 7779, pp. 129–144.

[126] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter, "Public Keys," in *Advances in Cryptology – CRYPTO 2012*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Aug. 2012.

[127] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices," in *Proceedings of the 21st USENIX Security Symposium*, Aug. 2012.

[128] E.-H. Lee, J.-H. Lee, I.-H. Park, and K.-R. Cho, "Implementation of high-speed SHA-1 architecture," in *IEICE Electronics Express*. The Institute of Electronics, Information and Communication Engineers, Aug. 2009.

[129] H. Handschuh, L. Knudsen, and M. Robshaw, "Analysis of SHA-1 in Encryption Mode," in *Topics in Cryptology — CT-RSA 2001*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Feb. 2001.

[130] B. Zoltak, "VMPC One-Way Function and Stream Cipher," in *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, ser. Lecture Notes in Computer Science. Springer, Feb. 2004.

[131] Y. Tsunoo, T. Saito, H. Kubo, M. Shigeri, T. Suzaki, and T. Kawabata, "The Most Efficient Distinguishing Attack on VMPC and RC4A," 2005.

[132] A. Maximov, "Two Linear Distinguishing Attacks on VMPC and RC4A and Weakness of RC4 Family of Stream Ciphers (Corrected)." *IACR Cryptology ePrint Archive*, Feb. 2007.

[133] S. Li, Y. Hu, and Y. Z. Y. Wang, "Improved Cryptanalysis of the VMPC Stream Cipher," *Journal of Computational Information Systems*, 2012.

[134] J. Manger, "A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0," in *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2139. Springer, 2001, pp. 230–238.

[135] P. Gutmann, "Lessons Learned in Implementing and Deploying Crypto Software," in *Proceedings of the 11th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2002, pp. 315–325.

[136] R. J. Anderson and R. M. Needham, "Robustness Principles for Public Key Protocols," in *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '95. London, UK, UK: Springer-Verlag, 1995, pp. 236–247.

List of Tables

List of Figures

Curriculum Vitae

PERSONAL DETAILS

Name	Christopher Meyer
Nationality	German
Date of Birth	18.12.1984
Place of Birth	Marburg/Wehrda, Germany

EDUCATION

01/2011 – 02/2014	Ruhr-University Bochum, Bochum Degree : *Dr.-Ing.*
10/2004 – 01/2011	Ruhr-University Bochum, Bochum Degree: *Dipl.-Ing.*
09/1997 – 07/2004	Lahntalschule Biedenkopf, Biedenkopf Degree: *Abitur*

PROFESSIONAL EXPERIENCE - EXCERPT

since 01/2014	Safenet - SFNT Germany GmbH, Germering
01/2011 – 12/2013	Ruhr-University Bochum, Bochum
08/2009 – 11/2010	secunet Security Networks AG, Essen
11/2007 – 05/2009	escrypt Embedded Security GmbH, Bochum

Bochum, 9th February 2014