Insert here your thesis' task.

Czech Technical University in Prague

Faculty of Information Technology

Department of Computer Systems

Master's thesis

Current Development of Authenticated Encryption and its Usage in the TLS Protocol

Bc. Jan Žák

Supervisor: prof. Ing. Róbert Lórencz, CSc.

May 3, 2015

Acknowledgements

My sincere thanks goes to prof. Wu Hongjun, Assistant Professor at Nangyang Technological University, Singapore, who served as my external supervisor, advised about my thesis and provided an invaluable insight into the field of authenticated encryption.

Declaration

In Prague on May 3, 2015 . . . .

Citation of this thesis

Žák, Jan. *Current Development of Authenticated Encryption and its Usage in the TLS Protocol.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Tato práce se zaměřuje na přidání nové šifrovací sady s autentizovaným šifrováním do OpenSSL implementace TLS protokolu použitím EVP API. Nová šifra byla vybrána z přihlášených algoritmů do CAESAR soutěže. Nová šifrovací sada byla úspěšně otestována analýzou TLS síťové komunikace mezi serverem a klientem.

Klíčová slova autentizované šifrování s přidruženými daty,caesar soutěž,openssl evp api,openssl knihovna,tls šifrovací sady,tls protokol

Abstract

This thesis focuses on adding a new authenticated encryption cipher suite in the OpenSSL implementation of the TLS protocol using the EVP API. The cipher was selected from CAESAR competition submissions. The new cipher suite was successfully tested by analysing TLS network communication between server and client.

Keywords authenticated encryption with associated data,caesar competition,openssl evp api,openssl library,tls cipher suite,tls protocol

Contents

Introduction **1**

1 Modern cryptography **3**

1.1 Kerckhoffs' principle . . . 3
1.2 Encryption . . . 3
1.3 Message authentication . . . 5
1.4 Key exchange . . . 6
1.5 Authenticated encryption . . . 7

2 Competition for Authenticated Encryption: Security, Applicability and Robustness **11**

2.1 Requirements . . . 12
2.2 Submissions . . . 15
2.3 Selection . . . 19

3 Transport Layer Security (TLS) protocol **21**

3.1 Standardization . . . 22
3.2 Records . . . 23
3.3 Handshake protocol . . . 24
3.4 Cipher suites . . . 28

4 OpenSSL **31**

4.1 Release schedule . . . 31
4.2 Source code . . . 32
4.3 Command-line tools . . . 33
4.4 libcrypto library . . . 40
4.5 libssl library . . . 49

5 Implementing a new TLS cipher suite in OpenSSL **51**

5.1 Cipher . . . 51

5.2 TLS cipher suite 58

Conclusion **67**

Bibliography **69**

A Symbols **71**

B Acronyms **73**

C TLS constants **75**

D Security indication in browsers **79**

E Content of attached CD **83**

List of Figures

1.1 How can Alice and Bob communicate securely? 4
1.2 Generic setting for encryption 4
1.3 How can Bob know who sent the message? 5
1.4 Generic setting for authentication 5
1.5 Generic compositions of authenticated encryption 8

2.1 Basic block modes 16
2.2 Sponge functions 17

3.1 Role of TLS in TCP/IP Reference Model 22
3.2 TLS record 23
3.3 TLS full handshake 26
3.4 TLS abbreviated handshake 27

D.1 Browser security indication - Chrome 40 80
D.2 Browser security indication - Firefox 35 80
D.3 Browser security indication - IE 11 81
D.4 Browser security indication - IE 11 (Metro) 81
D.5 Browser security indication - Opera 27 82
D.6 Browser security indication - Safari 5.1 82

List of Tables

2.1 CAESAR inputs 12

5.1 Packet capture 63

C.1 TLS record types 75
C.2 TLS versions 75
C.3 TLS handshake types 76
C.4 TLS alert levels 76
C.5 TLS alert types 77
C.6 Sample TLS cipher suites and their security properties 78

Introduction

At its core, the Internet is built on top of IP and TCP protocols, which are used to package data into small packets for transport. As these packets travel across the world, they cross many computer systems in many countries. Because the core protocols do not provide any security by themselves, anyone with access to the communication links can gain full access to the data as well as change the traffic without detection.

Over the last years, the Internet has grown into a major platform for the world's communication. The Internet's trustworthiness has become critical to its success. If a person cannot trust that they are communicating with the party they intend, they will not give out their confidential data. If they cannot be assured that delivered information is not modified in transit, they will not trust it as much.

Currently the TLS protocol uses a MAC-then-Encrypt generic composition of encryption and authentication algorithm to achieve both confidentiality and integrity. More recently, the idea of using a single cryptosystem has become accepted. In this concept, the MtE composition is replaced by a single authenticated encryption algorithm, such as AES-GCM.

This thesis focuses on the OpenSSL cryptographic library, which is the most frequently used implementation of the TLS protocol worldwide. It implements a new authenticated encryption algorithm into the TLS protocol.

Chapter 1

Modern cryptography

1.1 Kerckhoffs' principle

In the modern era of cryptography, crypto algorithms are preferred to be public. A well-known *Kerckhoffs' principle* roughly says, that the security of the cryptosystem must depend only on the secrecy of the key, and not on the secrecy of the algorithm.

There are very good reasons for this rule. Algorithms are hard to change. They are built into software or hardware, which can be difficult to upgrade. In real world, the same algorithm is used for a very long time. It is hard enough to keep the key secret, keeping the algorithm secret is far more difficult and expensive.

From past we know that it is very easy to make a small mistake and create cryptographic algorithm that is weak. If the algorithm is secret, nobody will find this fault until the attacker tries to break it. On the other hand, if the algorithm is public, researchers worldwide can participate in analyzing and improving the algorithm and its implementations. Thus if the algorithm is kept secret, for example by a private company saying that their algorithm is unbreakable, you should not trust it.

While the cipher is publicly known, the secret key still needs to be exchanged via another communication method, which prevents Eve from reading it. Alice and Bob can meet in person to exchange the key or Alice can mail it via public post service. The key exchange problem is covered more detailed in section 1.4.

1.2 Encryption

Encryption is the original goal of cryptography. It is the process of encoding messages in such a way that only authorized parties can read it. Encryption does not of itself prevent interception, but denies the message content to the

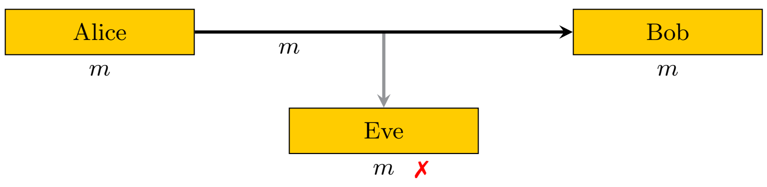


Figure 1.1: How can Alice and Bob communicate securely?

interceptor.

The generic use case is: Alice and Bob[1] want to communicate with each other. However, communication channels are assumed not to be secure. Eve is eavesdropping on the channel. Any message that Alice sends to Bob is also received by Eve. (The same applies for messages sent from Bob to Alice, but it is the same problem and the same solution will work for Bob's messages, so we concentrate to Alice messages.) How can Alice and Bob communicate without learning everything? (Figure 1.1) [2]

To prevent Eve from understanding the conversation, Alice and Bob want to use encryption. They first need to agree on a set of *encrypt* and *decrypt* function E, D (a cipher) and a secret encryption *key* K_e. Then they can use encryption in their communication channel in Figure 1.2.

So Alice wants to send a *plaintext* message m. She first encrypts it using the encrypt function $E(K_e, m)$ to get a *ciphertext* message c. It can be sent over the communication channel, because only Alice and Bob know how to decrypt it. When Bob receives the ciphertext, he can decrypt it using the decrypt function $D(K_e, c)$ to get the original plaintext m that Alice wanted to send to him.

[1]Alice, Bob and Eve are placeholder names commonly used when discussing cryptography, to identify an archetypal role of participant. Alice is a sender, Bob is a receiver and Eve is an eavesdropper. For the first time these names were used in Ron Rivest's paper introducing RSA public key cryptosystem. [1] Since then, a number of other names have entered cryptographic literature, such as Malory for malicious active attacker.

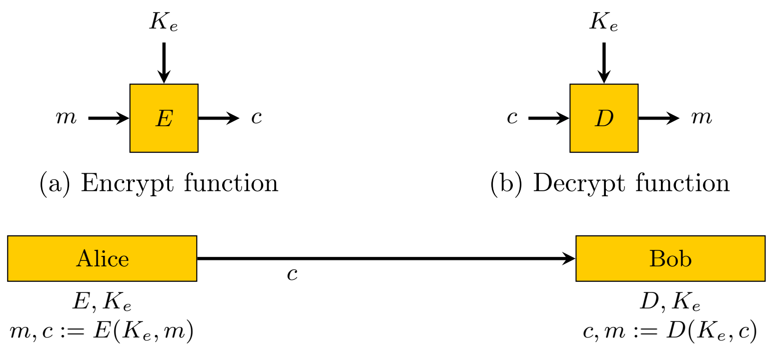


Figure 1.2: Generic setting for encryption

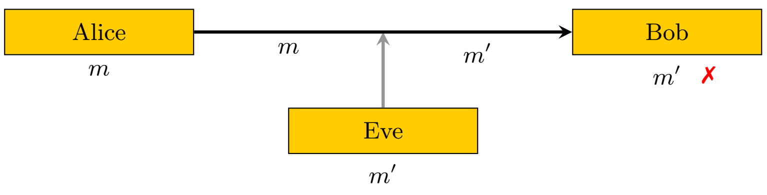


Figure 1.3: How can Bob know who sent the message?

Now Eve tries to listen to the message, but she receives only ciphertext message c. If we assume she does not know the encryption key K_e, she cannot decrypt it.

Example algorithms: RC4, DES, AES

1.3 Message authentication

Alice and Bob have another problem, as shown in Figure 1.3. If Eve has a bit more control over the communication channel, she can not only passively listen to messages, she can also actively interfere.

To prevent Eve from undetectably modifying or forging messages, Alice and Bob want to use authentication. They first need to agree on a set of *sign* and *verify* functions S, V (usually the verify function simply uses the sign function and compares its results) and a authentication key K_a (different from encryption key K_e). Then they can use authentication their communication channel in Figure 1.4.

So Alice wants to send a message m. She first computes a signature a using the sign function $S(K_a, m)$. This signature is also called *Message Authentication Code* (MAC). The message along with its signature can be sent over the communication channel, because only Alice and Bob know how to generate the signature. When Bob receives the message, he verifies the signature using

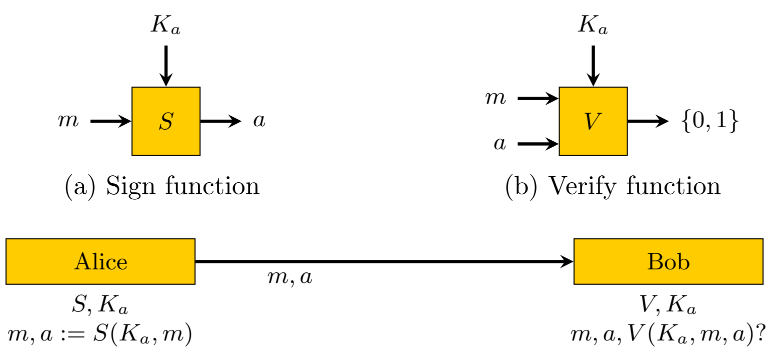


Figure 1.4: Generic setting for authentication

the verify function $V(K_a, m, a)$, if it passes, he can be sure that Alice sent the message.

Now Eve tries to modify the message m to a different message m'. If we assume that she does not know the authentication key K_a, she can only replace m with m'. Bob will try to verify it, but it fails, so Bob will recognize that the message is not correct and he will discard it.

Pure authentication is only a partial solution. Eve can still do a lot of other malicious actions. Imagine Alice sending to Bob a messages containing requests for bank transfer. Eve can record a message and then send it to Bob later again (replay it), reorder messages, or completely delete messages. Therefore, authentication is almost always combined with a message numbering scheme. If a message m contains such a message number, Bob is not fooled by Eve when she replays old messages. Bob will simply see that the message has correct signature, but the message number is from an old message, so he will discard it.

The best scheme of message numbering is a number sequence, incrementing by 1 for each message. Bob will accept only messages which passes the verification step and whose message number is strictly greater than the message number of the last message he accepted. So Bob receives a subsequence of messages of that Alice sent. A subsequence is simply the same sequence with growing message numbers with zero or more messages deleted.

Authentication with sequential message numbering solves most of the problem. Eve can still stop Alice and Bob from communicating by deleting or delaying messages. But that is all she can do. Alice and Bob can prevent the loss of information by using a scheme of resending messages that were lost, but that is more application-specific problem, and not part of cryptography.

Example algorithms: HMAC-MD5, HMAC-SHA1, HMAC-SHA256

1.4 Key exchange

Symmetric cryptographic primitives share a huge disadvantage. They rely on the existence of a shared secret value by Alice and Bob, which is used as a key to those algorithms. Without it, they do not work.

The secret value must be exchanged via a different, more secure communication channel, such as by handing it over in person, by public post service, or by cellphone. We cannot simply send the secret value in plaintext over the same channel as the message, which the secret value is protecting, because Eve could see it and use it to read or manipulate the protected content. If it would be safe to send the secret value via the channel, then it is useless to bother with encryption or authentication at all.

The problem of key distribution is often solved by technique called *One-Time Pad* (OTP). Alice and Eve can meet in person and exchange a lot of random values. Later when they are separated and they need to communicate securely, they just choose a single random value they exchanged before, eg. by its ID, and use it to secure further communication. After the communication ends, they will not use it again to prevent attacks which can take advantage from using the same key.

OTPs provide the most secure way to communicate securely, however it is very resource-consuming to exchange secret values this way. Thus an easier method is necessary to establish a secure communication channel. The solution lies in the modern field of asymmetric cryptography.

By using a well-designed key exchange cryptosystem, Alice and Bob can establish a shared secret value, without sending it over an insecure channel. In short, they agree on common parameters, which they use for generating their secret values. Then they transform them to a public parameters, they exchange them, and using their own secret value, and the value from the opposite party, they can compute the shared secred value. If thecomputed value is kept secret, Alice and Bob can use it as a key for all crypto algorithms that needs a shared key. Eve can listen to key exchange parameters, but she cannot compute the same value, because she does not know their secret exchange parameters.

Example algorithms: Diffie-Hellman, ElGamal

Asymmetric cryptography offers cryptographic primitives for encryption and message authentication as well, but these topics are out of scope of this thesis.

1.5 Authenticated encryption

Many applications desires to achieve both confidentiality and integrity goals by using some combination of encryption and authentication algorithm. However, securely combining them together is difficult, because it could lead to incorrect, error-prone combination.

There are a few researched combinations of encryption and authentication, called generic compositions: *Encrypt-and-MAC* (E&M), *Encrypt-then-MAC* (EtM), *MAC-then-Encrypt* (MtE). These combinations are successfully used in real-world software.

More recently, the idea of providing both confidentiality and integrity goals using a single cryptosystem has become accepted. In this concept, the combination of encryption and authentication algorithm is replaced by a single *Authenticated Encryption* (AE) or *Authenticated Encryption with Associated Data* (AEAD) algorithm. [3]

Authenticated Encryption is a form of encryption that, in addition to providing confidentiality for the plaintext that is encrypted, provides a way to check its integrity and authenticity. Decryption is combined in single step with integrity validation. These attributes are provided under a single, easy to use interface.

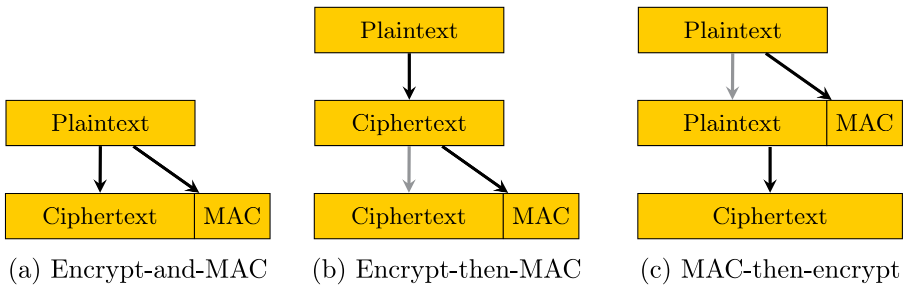


Figure 1.5: Generic compositions of authenticated encryption

Application designers can benefit from using AE by allowing them to focus on real project needs, instead of designing their own cryptosystem. More importantly, the security of an AE algorithm can be analyzed independent from its use in a particular application. This property frees the user of the AE of the need to consider security aspects such as the relative order of encryption and authentication and the security of the particular combination of cipher and MAC.

1.5.1 Generic compositions

The simpliest approach to design authenticated encryption schema is to combine a standard symmetric encryption algorithm with a MAC in some way. There are a few possible ways how to do it:

Encrypt-and-MAC The plaintext is encrypted to ciphertext, then a MAC is computed from plaintext as well. The ciphertext (containing encrypted plaintext) and plaintext's MAC are sent together.

Encrypt-then-MAC The plaintext is encrypted to ciphertext, then a MAC is computed from ciphertext. The ciphertext (containing encrypted plaintext) and ciphertext's MAC are sent together.

MAC-then-Encrypt A MAC is computed from the plaintext, then the plaintext and MAC are encrypted together to ciphertext. The ciphertext (containing encrypted plaintext and plaintext's MAC) is sent.

See Figure 1.5 for comparison.

All of them are used in real-world software - Encrypt-and-MAC in SSH, Encrypt-then-MAC in IPSec and MAC-then-Encrypt in TLS. However, recent research shows that only Encrypt-then-MAC schema is provably secure. [4] [5]

1.5.2 Associated data

Authenticated Encryption with Associated Data adds the ability to check the integrity and authenticity of some *Associated Data* (AD), also called *Additional*

Authenticated Data (AAD), that is not encrypted and sent side by side with the ciphertext.

Associated data can be for example TLS record header, to ensure that unencrypted data in the header cannot be tampered with, and TLS record sequence number, to ensure that the messages cannot be replayed.

Chapter 2

Competition for Authenticated Encryption: Security, Applicability and Robustness

In 2013, CAESAR was announced. It is a worldwide cryptographic competition, focused on finding new methods of authenticated encryption, that offer advantages against commonly used AES-GCM and will be suitable for widespread adoption. Submitted algorithms will be publicly evaluated by committee of researchers in fields of cryptography and cryptoanalysis.

This competition follows a long tradition of focused competitions in secret-key cryptography:

- In 1997, United States National Institute of Standards and Technology (NIST) announced an open competition for a new symmetric cupher, Advanced Encryption Standard (AES). This competition attracted 15 submissions from 50 cryptographers around the world. In the end, Rijndael was chosen as AES.
- In 2004, European Network of Excellence in Cryptology (ECRYPT) announced the ECRYPT Stream Cipher Project (eSTREAM), a call for new stream ciphers suitable for widespread adoption. This call attracted 34 submissions from 100 cryptographers around the world. In the end, the eSTREAM committee selected a portfolio containing several stream ciphers.
- In 2007, NIST announced an open competition for a new hash standard to Secure Hash Algorithm family (SHA-3). This competition attracted 64 submissions from 200 cryptographers around the world. In the end, Keccak was chosen as SHA-3.

	required	integrity	confidentiality	single-use
plaintext	**yes**	**yes**	**yes**	no
associated data	**yes**	**yes**	no	no
secret message number	no	**yes**	**yes**	**yes**
public message number	no	**yes**	no	**yes**

Table 2.1: CAESAR inputs

All past cryptographic competitions attracted many submissions from cryptographers around the world, and then even more security and performance evaluations from cryptanalysts. They are generally viewed as having provided a tremendous boost to the cryptographic research community's understanding of underlying concepts, and a tremendous increase in confidence in the security of some existing cryptosystems. Similar comments are expected to apply to CAESAR. [6]

2.1 Requirements

2.1.1 Functional requirements

For the purpose of CAESAR competition, an *authenticated cipher* is a pair of encrypt and decrypt functions, meeting the following specific requirements.

All inputs and outputs should be represented as opaque byte-strings (members of a set $\mathbb{Z}_{2^8}^*$), because they benefit from direct support of current computers to store and transmit them.

A cipher is permitted to be defined using objects other then byte-strings, nevertheless it must specify an unambiguous relationship between those objects and byte-strings (e.g. endianness of integers).

A cipher must specify a length of all fixed-length inputs. It is permitted to specify a maximum length of various-length inputs, but this limit must not be smaller than 65 kB and submissions are expected to include justification for any maximum length limits.

No other restrictions on their structure should be imposed, all inputs meeting the length restrictions must be accepted.

2.1.1.1 Inputs and outputs

A *plaintext* is a variable-length input/output, a piece of confidential information a sender wants to transmit to a receiver, as introduced in section 1.2.

A *ciphertext* is a variable-length input/output counterpart of the plaintext, that can be transmitted over an insecure channel. it is usually longer then the plaintext, because it contains an *authentication tag.* This length difference is permitted to be fixed constant, thus leaking the plaintext length via the ciphertext length. Designers are advised that minimizing ciphertext length is generally considered more valuable than hiding plaintext length.

A *key* is a fixed-length input, which determines the output of both encrypt and decrypt functions. The key must be shared between both communicating parties prior to encrypted communication. Without a key or with a different one then used in the encrypt function, the decrypt function produces no useful result. This follows the Kerckhoffs' principle as introduced in section 1.1.

An *associated data* is a variable-length input, a piece of information known by both communicating parties, which does not need to meet confidentiality requirement. However, its origin still needs to be verified by the receiving party. It can be for example some message metadata, such as version of used protocol.

A *nonce* (number used once) is a fixed-length input. It is a public value, which which is usually used as IV for the enclosed cipher. Such IVs should be unique for each encryption run, so it makes all ciphertexts undistinguishable even if the same key, message and associated data is used.

However, CAESAR call for submissions requests an unusual authenticated encryption interface. The user, who wants to encrypt, instead of providing the usual *four* arguments (the key, nonce, associated data, and message) for authenticated encryption, he needs to provide *five* arguments. The nonce has been transformed into a *public message number* and *secret message number*. [7]

A *public message number* is a fixed-length input. It is a public value with the same requirements as the nonce in the original definition of authenticated encryption.

A *secret message number* is a fixed-length input. It is a secret value, recoverable from the ciphertext, however it is not a part of the plaintext. Allowing both a secret message number and a public message number creates possibilities of different levels of their security requirements.

All inputs must meet various security purposes, as indicated by Table 2.1.

2.1.2 Software requirements

Each first-round submission must contain a portable reference software implementation to support public understanding of the cipher, cryptanalysis, verification of subsequent implementations, etc. The implementation must cover all recommended parameter sets, and must compute exactly the function specified in the submission. The reference implementation is expected to be optimized for clarity, not for performance. [6]

The submission must export the following constants:

- `CRYPTO_KEYBYTES` – the fixed length of key
- `CRYPTO_NSECBYTES` – the fixed length of secret message number (0 if not supported)

- `CRYPTO_NPUBBYTES` – the fixed length of public message number (0 if not supported)
- `CRYPTO_ABYTES` – the maximum (usually fixed) length difference between plaintext and ciphertext

```
#define CRYPTO_KEYBYTES 16
#define CRYPTO_NSECBYTES 0
#define CRYPTO_NPUBBYTES 12
#define CRYPTO_ABYTES 16
```

The submission must export the following `crypto_aead_encrypt` and `crypto_aead_decrypt` functions, which perform the encrypt and decrypt operation respectively.

```
int crypto_aead_encrypt(
    unsigned char *c, unsigned long long *clen,
    const unsigned char *m, unsigned long long mlen,
    const unsigned char *ad, unsigned long long adlen,
    const unsigned char *nsec,
    const unsigned char *npub,
    const unsigned char *k
) {
    // the code for the cipher implementation goes here,
    // generating a ciphertext c[0],c[1],...,c[*clen-1]
    // from a plaintext m[0],m[1],...,m[mlen-1]
    // and associated data ad[0],ad[1],...,ad[adlen-1]
    // and secret message number nsec[0],nsec[1],...
    // and public message number npub[0],npub[1],...
    // and secret key k[0],k[1],...
    return 0;
}
```

```
int crypto_aead_decrypt(
    unsigned char *m, unsigned long long *mlen,
    unsigned char *nsec,
    const unsigned char *c, unsigned long long clen,
    const unsigned char *ad, unsigned long long adlen,
    const unsigned char *npub,
    const unsigned char *k
) {
    // the code for the cipher implementation goes here,
    // generating a plaintext m[0],m[1],...,m[*mlen-1]
    // and secret message number nsec[0],nsec[1],...
    // from a ciphertext c[0],c[1],...,c[clen-1]
    // and associated data ad[0],ad[1],...,ad[adlen-1]
    // and public message number npub[0],npub[1],...
    // and secret key k[0],k[1],...
    return 0;
}
```

The output of functions must be determined entirely by the inputs in their arguments and must not be affected by any randomness or other hidden inputs.

The functions should perform the operation in constant time with regard to any input data (even invalid data) to prevent timing side-channel attacks.

The decrypt function must return -1 if the ciphertext is not valid, i.e. if the ciphertext does not equal the encryption of any (plaintext, secret message number) pair with this associated data, public message number, and secret key. The functions may return other negative numbers to indicate other failures, for example memory-allocation failures. [6]

2.1.3 Hardware requirements

Each submission selected for the second round will also be required to include a reference hardware design (i.e., a reference implementation in the VHDL or Verilog languages). Details of the hardware API have not yet been specified. [6]

2.2 Submissions

The competition was announced on 2013-01-15 at the Early Symmetric Crypto workshop in Mondorf-les-Bains, also announced online[2]. First-round submission papers must have been received till 2014-03-15, reference software implementations must have been received till 2014-04-15.

After passing the first-round deadline, all submissions were published. All submission papers can be downloaded on CAESAR homepage[3]. Submission source codes are bundled together with SUPERCOP benchmark application[4]. There is a website with SUPERCOP speed benchmark results[5].

Also there was Directions in Authenticated Ciphers (DIAC) 2014 conference, where a lot of sumbission authors presented their candidates. Talk slides are available for download on the DIAC website[6].

There were submitted 57 candidates for the CAESAR competition. A good insight into their classification is provided by Authenticated Encryption Zoo[7]. At the time of writing this thesis, 9 candidates (AES-COBRA, Calico, CBEAM, FASER, HKC, Marble, McMambo, PAES and PANDA) are considered broken and were withdrawn from the competition, because a cryptanalysis was published that broke the security claim made by the designers. [8] There are 48 candidates remaining. It is expected that the about a half of them will advance to the second round.

[2]`https://groups.google.com/d/forum/crypto-competitions`

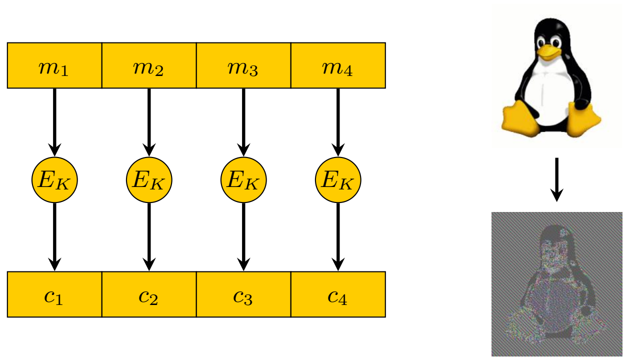
[3]`http://competitions.cr.yp.to/caesar-submissions.html`

[4]`http://bench.cr.yp.to/supercop.html`

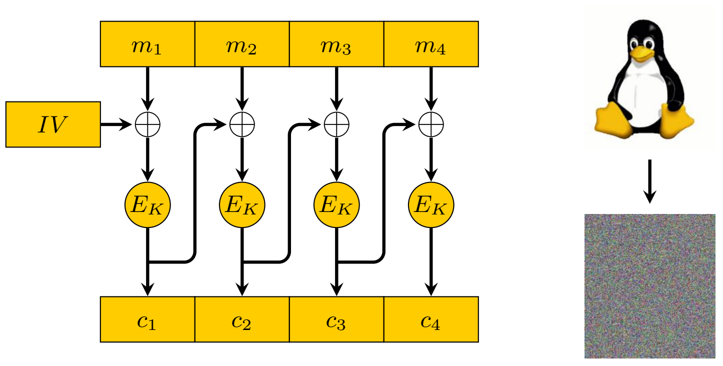
[5]`http://www1.spms.ntu.edu.sg/~syllab/speed/`

[6]`http://2014.diac.cr.yp.to/index.html`

[7]`https://aezoo.compute.dtu.dk/`



(a) ECB block mode



(b) CBC block mode

The ECB mode is the most basic block mode, which does not perform any block feedback. Note that the plaintext structure is still observable in the ciphertext.

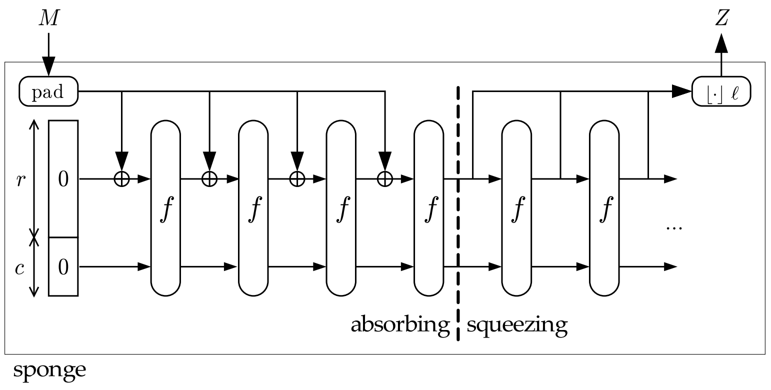
Figure 2.1: Basic block modes

Announcement of second-round candidates was initially scheduled for 2015-01-15. However it is a hard task to do a proper security review, analysis and comparison of all submissions. Currently in the time of writing this thesis, the second-round candidates announcement is being postponed every month.

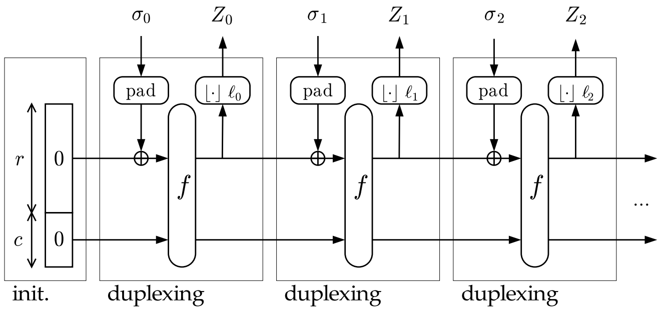
2.2.1 Overall construction

Most of first-round candidates can be classified according to their construction design approaches. [8]

Block Cipher A block cipher is a bijective keyed permutation $E : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n, E(K,P) = C$, parametrized by a secret key K of length k, and that takes as input a plaintext message p of length n, and outputs a ciphertext message c of length n. The permutation is used



(a) The sponge construction



(b) The duplex construction

Figure 2.2: Sponge functions

for encryption, an inverted permutation is used for decryption. A block mode is usually accompanied with a block mode.

Block Mode A block mode (also called mode of operation) is used by block ciphers for secure transformation of data larger than a single block. See Figure 2.1 for basic block modes. Some submissions defines only a new block mode and relies on existing block cipher (e.g. AES). [9]
Example candidates: COBRA, JAMBU, POET

Stream Cipher A stream cipher is a pseudo-random generator, that takes a fixed-length secret key and generates a keystream of variable length. The keystream is combined with the plaintext message to produce ciphertext message and vice versa. The combining operation is usually XOR.
Example candidates: ACORN, Morus

Key-Less Permutation A key-less permutation is a bijective permutation on fixed-length strings. The key is sent to the permutation alongside the input, thus changing the internal state, effectivelly encrypting the output and producing the MAC tag.

Sponge Function A sponge function is a generalization of both hash functions, which have a fixed output length, and stream ciphers, which have a fixed input length. It is a simple iterated construction for building a function with variable-length input and arbitrary-length output based on a fixed-length permutation. The inner permutation operates on a finite state of $b = r + c$ bits. The value r is called the bitrate and the value c the capacity. The sponge construction operates on the state by iteratively applying the inner permutation to it, interleaved with the entry of input or the retrieval of output, chunked by bitrate size. Literally, the sponge is said to *absorb* its inputs block by block first before it processes and *squeezes* it out afterwards. [10]
A duplex sponge construction is closely related to the sponges. However unlike sponges, which are stateless between calls, the duplex construction allows alternation of input and output blocks at the same rate, like a full-duplex communication. Which means that it requires only one call of the inner permutation per input block. [11] See Figure 2.2 for comparison.
Example candidates: Ascon, ICEPOLE, Keyak, NORX, STRIBOB

2.2.2 Underlying primitive

Under the overall construction usually hides a cryptographic primitive (e.g. inner permutation in sponge construction), which does the heavy lifting.

AES A lot of submissions use AES cipher or some of its parts (e.g. its round function), because during the years, AES have been enormously analysed in detail, and it is still believed to be secure. Moreover, starting with Intel's Westmere microarchitecture in 2011, current processors provide AES native instructions (AES-NI), that allow hardware-accelerated fast constant-time encryption and decryption.

Other named primitive e.g. SHA2, Keccak, ChaCha, Streebog, etc.

Generic primitive type e.g. Linear Feedback Shift Register (LSFR), Addition Rotation XOR (ARX), Logic Rotation XOR (LRX), Substition Permutation Network (SPN), etc.

2.2.3 Functional characteristics, selection criteria

The selection of second-round candidates will focus not only on general security of the scheme, but no less on important functional characteristics, which are good to have.

High Security The schema should be secure against all known kinds of cryptanalysis.

High Speed The schema should be fast enough to compete with existing ciphers. However it is arguable whether the speed can be achieved by hardware acceleration such as by AES-NI or SIMD (SSE2, AVX2) instruction sets, because it does not need to be available on all platforms.

Simplicity Clean design principles simplify cryptanalysis and allow a straightforward implementation in software and hardware.

Minimum Overhead The ciphertext will be always larger than the plaintext, because it needs to include the MAC tag. However, this length difference should be minimal.

Side-Channel Robustness The schema should resist against timing side-channel attacks, e.g. by avoiding data-dependant table look-ups (S-Boxes) or integer arithmetrics.

Parallelizable An operation is parallelizable if the processing an input block does not depend on the output of processing any other block. Parallelizable encryption and decryption is considered separately.

Online A cipher is called online if the processing an input block depends only on the output of processing of previous blocks and only constant size-state is used from the processing of one block to the next. It effectivelly means that the MAC tag must be computed during encryption, thus online schemes can be called one-pass. Such schemes can be faster in general. Schemes that are not online are called offline or two-pass.

Inverse-Free A scheme is called inverse-free if it does not require either its underlying primitive inverse operation, e.g. as does require the block cipher's decryption function. Such scheme can save precious memory and circuit area resources.

Nonce Misuse-Resistance States the robustness of the scheme when nonces are repeated. This property avoids maintaining a nonce generator.

2.3 Selection

Because in the time of writing this thesis the announcement of second-round candidates is still being postponed, I could not choose a qualified candidate, which I would implement into OpenSSL. So I decided to implement a cipher using the generic CAESAR API (see subsection 2.1.2).

As a reference cipher for my implementation part I chose the NORX cipher, because it have received no negative analysis. However it is not important which particular cipher I used, because the cipher can be easily switched for a different cipher complying with the CAESAR API, as soon as the the second-round candidate or final announcement will be made.

CHAPTER 3

Transport Layer Security (TLS) protocol

TLS is a security protocol used in almost 100% of secure Internet transactions. Essentially, TLS transforms a typical reliable transport protocol (such as TCP) into a secure communication channel suitable for sending sensitive messages. TLS does not dictate which cryptographic algorithms need to be used. Instead, TLS serves as a framework establishing and maintaining a secure comminucation channel, while new cryptographic algorithms can be implemented using a common interface.

Adding encryption to an existing protocol is best performed in a transparent way, so that applications using the protocol library do not need to change their code to support encryption. A perfect example is HTTP protocol. A HTTP library can support both plaintext HTTP and encrypted HTTPS, and an application using this library can select the protocol simply in an URL, by specifying *http://* or *https://* respectively. See Figure 3.1.

TLS has four main goals, listed here in the order of priority:

Cryptographic security This is the main issue: enable secure communication between any two parties who wish to exchange information.

Interoperability Independent programmers should be able to develop programs and libraries that are able to communicate with one another using common cryptographic parameters.

Extensibility TLS is effectively a framework for the development and deployment of cryptographic protocols. Its important goal is to be independent of the actual cryptographic primitives (e.g., ciphers and hashing functions) used, allowing migration from one primitive to another without needing to create new protocols.

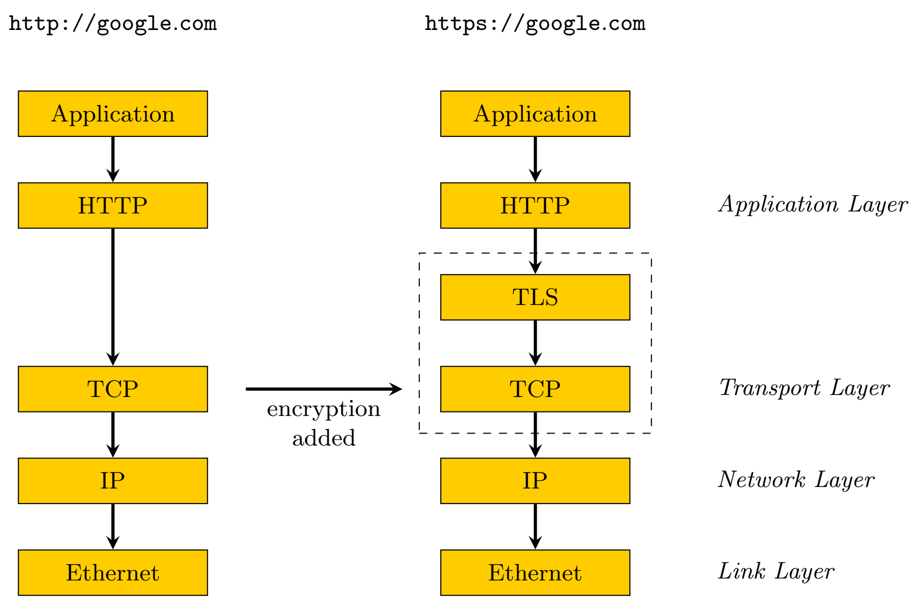


Figure 3.1: Role of TLS in TCP/IP Reference Model

Efficiency The final goal is to achieve all of the previous goals at an acceptable performance costreducing costly cryptographic operations down to the minimum and providing a session caching scheme to avoid them on subsequent connections. [12]

Whereas TLS provides security over reliable TLS communication, there also exists its variant, DTLS protocol. DTLS is deliberately designed to be as similar to TLS as possible, both to minimize new security invention and to maximize the amount of code and infrastructure reuse. [13] This thesis is about TLS only.

3.1 Standardization

The Internet is the result of a long-term collaboration between governments, academia, and businesses seeking to create a worldwide communication network. For the Internet to function correctly, it must be based upon standardized communication protocols.

Standards concerning the Internet are produced by the Internet Engineering Task Force (IETF) non-profit organization, where experts from around the world collaborates in work groups focused on specific area. IETF produces an informal series of documents known as Requests for Comments (RFCs). For a document to become an Internet standard, it is begins its life by being proposed as an RFC on the standardization track. RFCs in development are

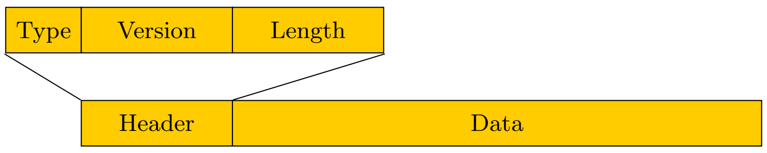


Figure 3.2: TLS record

temporarily available as *Internet Drafts*. After approval from IETF may be published as *Proposed Standard*. [14]

There are also other classes of RFCs, most notably experimental and informational RFCs. IETF RFCs cover all the topics of interest to an implementer working with the Internet, which would explain why there are so many of them[8] - over 7400 at the time of writing.

Many of IETF RFCs describe security algorithms, protocols, or recommendations. The most interesting for this thesis are these produced by TLS working group[9], such as:

RFC2246 The TLS Protocol Version 1.0

RFC4346 The Transport Layer Security (TLS) Protocol Version 1.1

RFC5246 The Transport Layer Security (TLS) Protocol Version 1.2

draft-ietf-tls-tls13 The Transport Layer Security (TLS) Protocol Version 1.3 (work in progress)

RFC5288 AES Galois Counter Mode (GCM) Cipher Suites for TLS

RFC6655 AES-CCM Cipher Suites for Transport Layer Security (TLS)

TLS implementations are typically written as a set of functions that generate and parse all TLS record messages, and perform the relevant cryptographic operations. The state machine that this process must implement, is currently not standardized, and differs between implementations. Allowing unexpected transitions in this state machine can lead to unexpected behavior. There is an effort to standardize the TLS state machine to allow formal verification of core components in cryptographic protocol libraries. [15]

3.2 Records

At a high level, TLS protocol specifies a structure of every record (packet). Each TLS record starts with a short header, which contains information about

[8]`http://www.rfc-editor.org/rfc-index.html`
[9]`https://tools.ietf.org/wg/tls/`

the record type (subprotocol), protocol version and data length. Message data follows the header. See Figure 3.2 for record structure.

The record type is identified in the record by 1-byte integer ID as specified in Table C.1. The protocol version can be either SSL 3.0 (deprecated), TLS 1.0, TLS 1.1, TLS 1.2 and it is identified in the record by 2-byte integer ID as specified in Table C.2. The data length field is 2-byte long and it specifies the message data length.

There are the following record types (subprotocols):

Handshake protocol The handshake protocol to negotiate connection parameters, such as the cipher suite, authenticate each other and verify that handhshake messages have not been modified by an attacker.

ChangeCipherSpec protocol The ChangeCipherSpec protocol contains a single message, which is a signal from the sending side that it obtained enough information to generate the connection parameters, such as the encryption keys, and is switching all further communication to encryption. Client and server both send this message when the time is right.

Alert protocol Alerts are intended to use a simple notification mechanism to inform the other side in the communication of exceptional circumstances. They're generally used for error messages, as listed in Table C.5.

Application protocol The Application protocol carries application messages, which are just opaque byte arrays as far as TLS is concerned. These messages are packaged, fragmented, and encrypted by the record layer, using the current connection security parameters, such as the negotiated cipher suite.

Heartbeat protocol The Heartbeat protocol extension allows a keep-alive functionallity without performing renegotiation. Its purpose is intended especially for DTLS, however it is implemented also in TLS.

This thesis focuses on negotiation of the cipher suite in the Handshake protocol and on application data encryption in the Application protocol.

3.3 Handshake protocol

When a client and server start communicating, they use the handshake protocol to negotiate connection parameters, such as the cipher suite, authenticate each other and verify that handhshake messages have not beed modified by an attacker. It is the most complex part of the TLS protocol, because it performs these tasks:

- exchange supported capabilities and agree on shared connection parameters (TLS protocol version, cryptographic algorithms)

- exchange necessary cryptographic parameters to agree on shared secret values (*master secret*) using public-key cryptography
- exchange certificates or other cryptographic information to authenticate one another
- verify that the handshake has not beed tampered by a third party
- verify that both parties have calculated the same secret values and they can be reliably used to transport application data via record protocol

This phase usually takes 6-13 messages (see Table C.3 for list of all message types) in 3-4 network flights, depending on which features are used. There can be many variations in the exchange, depending on the configuration and supported protocol extensions. In practice, we can see three common flows:

- full handshake with client and server authentication
- basic handshake with server authentication
- abbreviated handshake that resumes an earlier session

If a client and server has not previously communicated with each other, both parties will perform a full or basic handshake in order to establish a session. See Figure 3.3.

Full handshake requires client authentication, whereas basic handshake does not. Also it is possible to perform an anonymous handshake without any authentication, but it is not recommended, because it is sucpectible to MitM attacks.

A full handshake is completed after 4 network flights before the handshake is complete and protocol parties can begin to send application data. Thus, using TLS adds a latency penalty of 2 RTTs if the client sends application data first, such as in HTTP protocol.

1. *ClientHello* - client initiates a handshake, sends its capabilities to server
2. *ServerHello* - server selects the best connection parameters supported by both parties
3. *Certificate* - server sends its certificate chain (only if server authentication is required)
4. *ServerKeyExchange* - server sends additional information required to generate the master secret (only if it is required by selected cipher suite)
5. *CertificateRequest* - server requests client authentication and sends requirements for acceptable certificates (only if client authentication is required)

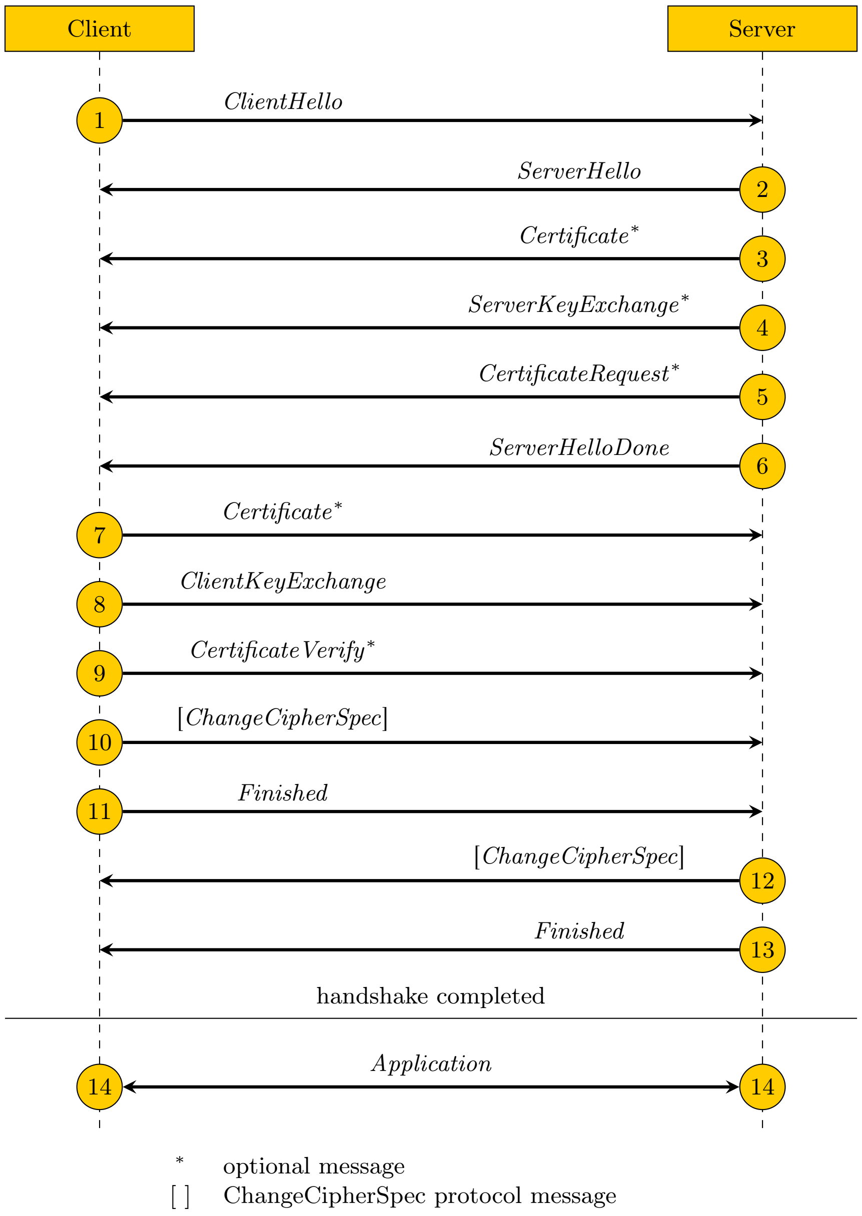


Figure 3.3: TLS full handshake

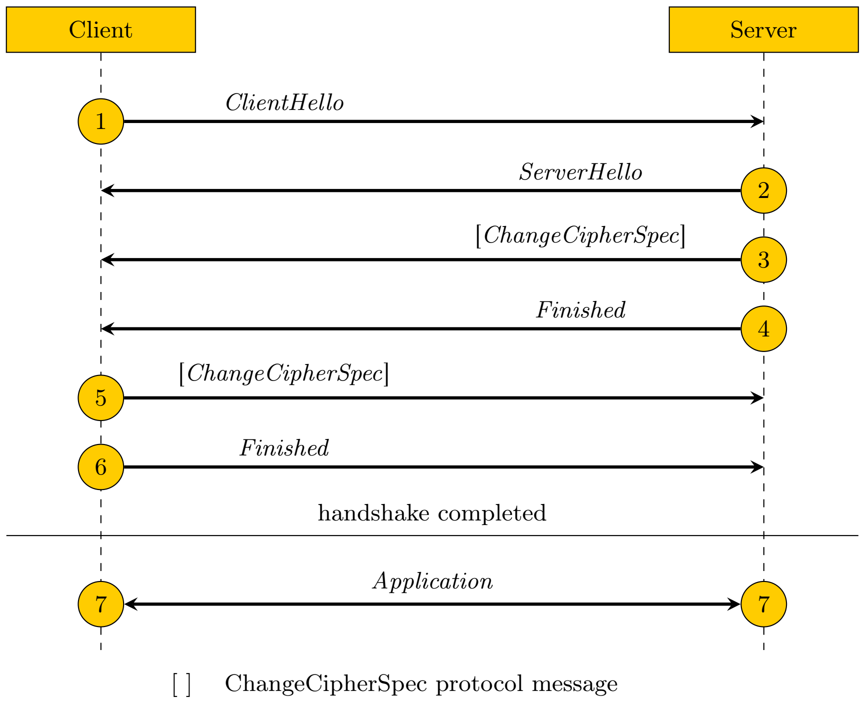


Figure 3.4: TLS abbreviated handshake

6. *ServerHelloDone* - server indicates completion of its side of negotiation

7. *Certificate* - client sends its certificate chain (only if client authentication is required)

8. *ClientKeyExchange* - client sends additional information required to generate the master secret

9. *CertificateVerify* - client proves the posession of private key corresponding to the previously sent client certificate (only if client authentication is required)

10. *ChangeCipherSpec* - client notifies server, that all following messages are encrypted

11. *Finished* - client sends a MAC of the handshake messages it sent and received

12. *ChangeCipherSpec* - server notifies client, that all following messages are encrypted

13. *Finished* - server sends a MAC of the handshake messages it sent and received

14. handshake is completed, secure communication channel is established, both parties can securely send application data

An abbreviated handshake is completed after 3 network flights, thus adding a latency penalty of just 1 RtT if the client sends application data first. See Figure 3.4. The session reuses previously exchanged secret values between the client and server, identified by either *Session Tickets* or *Session Cookies.*

3.4 Cipher suites

TLS is great in flexibility which provides for using various cryptographic primitives in a common framework. A selection of cryptographic primitives and their parameters is called *cipher suite.*

A cipher suite is defined by the following attributes:

- Key exchange algorithm
- Authentication algorithm
- Encryption algorithm
  - cipher algorithm
  - key size
  - cipher mode
- MAC algorithm (unless it is included in encryption algorithm)
- Pseudorandom function (since TLS 1.2)

Cipher suite names are usually long, descriptive and consistent. They are made from names of underlying used algorithms. See Table C.6 for sample common cipher suites.

3.4.1 Key exchange

Key exchange as introduced in section 1.4 is used in TLS to establish a shared value called *premaster secret.* The structure of premaster secret depends on the key exchange algorithm. From the premaster secret is constructed a 48-byte shared value called *master secret*, a value which is used in all subsequent operations affecting the security of the session.

Which key exchange is used depends on the negotiated suite. Once the suite is known, both sides know which algorithm to follow. In practice, there are four main key exchange algorithms:

RSA RSA is effectively the standard key exchange and authentication algorithm. More specifically, it is a key transport algorithm – the client generates the premaster secret and transports it to the server, encrypted with the server's public key. It is universally supported but suffers from one serious problem. Its design allows a passive attacker to decrypt all encrypted data, provided she has access to the server's private key. Because of this, the RSA key exchange is being slowly replaced with other algorithms, those that support so called *forward secrecy.*

DHE Ephemeral Diffie-Hellman (DHE) key exchange is a well-established algorithm. More specifically, it is a key agreement algorithm – the negotiating parties both contribute to the process and agree on a common key. It is liked because it provides forward secrecy but disliked because it's slow. In TLS, DHE is commonly used with RSA authentication.

ECDHE Ephemeral Elliptic Curve Diffie-Hellman (ECDHE) key exchange is based on elliptic curve cryptography, which is relatively new. Conceptually it is a key exchange algorithm similar to DHE. It's liked because it's fast and provides forward secrecy. It's well supported only by modern clients. In TLS, ECDHE can be used with either RSA or ECDSA authentication. [12]

3.4.2 Authentication

Authentication is tightly coupled with key exchange in order to avoid repetition of costly cryptographic operations. In most cases, the basis for authentication will be public key cryptography (most commonly RSA, but sometimes ECDSA) by X509 certificates.

The certificate is either compared to a preshared certificate or validated against a trusted root store managed by operating system. The trusted root store contains root certificates of default trusted *Certificate Authorites* (CAs), which are privileged to issue X509 certificates. If TLS is used in web browser in HTTPS protocol, the browser can show the state of certificate validation to the user. Trusted certificate is usually signalized by a green lock icon, while untrusted certificate displays a warning and the user can decide if he trusts the certificate or not. See Appendix D for browser behavior details.

Once the certificate is validated, the client has a known public key to work with. After that, it's down to the particular key exchange method to use the public key in some way to authenticate the other side.

During the RSA key exchange, the client generates a random value as the premaster secret and sends it encrypted with the server's public key. The server decrypts the message to obtain the premaster secret. The authentication is implicit. It is assumed that only the server in possession of the corresponding private key can retrieve the premaster secret.

During the DHE and ECDHE exchanges, the server contribute to the key exchange with its random parameters, signed with its private key. The client can validate it by the corresponding public key.

Do not confuse authentication with message authentication. Asymmetric authentication is used during handshake, while message authentication is used during application data transfer.

3.4.3 Encryption and message authenticaion

TLS can encrypt data (as introduced in section 1.2) using a variety of ways, using ciphers such as 3DES, RC4, AES or CAMELLIA. AES is by far the most popular cipher.

Integrity validation (as introduced in section 1.3) is part of the encryption process using an authenticated encryption scheme (as introduced in section 1.5). It is handled either explicitly at the protocol level using a MtE generic composition scheme (as introduced in subsection 1.5.1), example algorithms usually consist of HMAC and hash function such as MD5, SHA1, SHA256. However, recent research shows that only EtM schema is provably secure. TLS reacts to these results and introduced optional EtM support, which can be negotiated by a specific a TLS extension as defined in RFC 7366[10].

Or the integrity validation can be handled implicitly by the negotiated AEAD cipher. Currently it is recommended to prefer AEAD to generic composition cipher suites, because they can offer better performance and security. However there is only one widespread AEAD cipher, AES-GCM. The CAESAR competition as described in chapter 2 focuses on finding new AEAD ciphers.

The MAC tag is computed from plaintext and additional data, such as TLS record header, to ensure that unencrypted data in the header cannot be tampered with, and TLS record sequence number, to ensure that the messages cannot be replayed.

3.4.4 Pseudorandom function

A pseudorandom function is used in TLS to expand the 48-byte master secret into arbitrary-length blocks of data, which can be used as shared keys in encryption, message authentication and other algorithms requiring secred shared data.

Before TLS 1.2, a protocol-wide pseudorandom function was used, which was combined from HMAC-MD5 and HMAC-SHA1. Since TLS 1.2, it was defined a recommended PRF: HMAC-SHA256. New cipher suites must explicitly specify a PRF, the recommended PRF or stronger. Older cipher suites must use the recommended PRF when negotiated over TLS 1.2.

[10] `https://tools.ietf.org/html/rfc7366`

Chapter 4

OpenSSL

OpenSSL[11] is an opensource cryptographic toolkit, consisting of implementations of many cryptographic aslgorithms, all versions of TLS protocol and various command-line tools.

It is free to get and use for both non-commercial and commercial purposes, with some simple licence conditions[12].

OpenSSL provides two libraries, `libssl` and `libcrypto`. `libssl` is responsible for SSL/TLS protocol, also with other supportive functions such as parsing and validating X509 certificates.

4.1 Release schedule

Because SSL/TLS library is considered as part of critical infrastructure, it is important for developers and vendors to know which versions are supported to receive security fixes in future. Whenever a new version containing fixes for known security flaws is released, in production environment it is recommended to update the library as soon as possible, because not updating could cause dangerous leak of confidential information.

Since OpenSSL 1.0.0, the versioning policy was improved to clearly indicate the level of included changes[13]:

- Letter releases, such as 1.0.1**k**, contain bug and security fixes and no new features.
- Minor releases, such as 1.0.**2**, usually contain new features, but they does not break binary compatibility. Every application compiled with version 1.0.0 can be also compiled with any of future 1.0.x versions and get advantages of new implemented features.

[11] `https://www.openssl.org`
[12] `https://www.openssl.org/source/license.html`
[13] `https://www.openssl.org/about/releasestrat.html`

- Major releases, such as 1.**1**.0, can break binary compatibility.

Also support timelines were updated recently for all current and future releases:

- 0.9.8 and 1.0.0 will be supported until 2015-12-31. Security fixes only will be applied until then.
- 1.0.1 will be supported until 2016-12-31.
- 1.0.2 will be supported at least until 2016-12-31.
- Every future releases will be supported at least for two years, a LTS release will be supported at least for five years.

Version 1.1.0 will break binary compatibility because a major cleanup is necessary. A lot of recently found security bugs were caused by excessive complexity of the source code. The preview version is expected to be available in the middle of 2015 and to be released in the end of 2015.

4.2 Source code

Download the source code from the official OpenSSL homepage[14] and compare its hash fingerprints. The latest version in the time of writing is 1.0.2a.

Compile it with commands:

```
./config
make
```

It results into an all-in-one `apps/openssl` binary. You can run attached tests with `make test` command. If you wish to install it globally to your system, run `make install` command with root privileges.

While making changes into OpenSSL code, sometimes I needed to debug the binary with GDB. You can turn on debugging symbols with `./config -d` command and build again.

OpenSSL coding style in past was inconsistent, however in the recent stable version 1.0.2 it has been unified to conform a defined rules[15].

OpenSSL provides two primary libraries: libssl and libcrypto. The libcrypto library provides the fundamental cryptographic routines used by libssl. A user can however use libcrypto without using libssl.

OpenSSL source code contains a lot of various directories, for my purposes only the following are significant:

[14] `https://www.openssl.org/source/`

[15] `https://www.openssl.org/about/codingstyle.txt`

- `apps` – command-line tools
- `crypto` – libcrypto library
- `ssl` – libssl library
- `demos` – examples
- `docs` – man pages and howtos
- `include` – include header files
- `util` – perl scripts for C code generation

4.3 Command-line tools

OpenSSL is primarily a library that is used by developers to include support for strong cryptography in their programs. However it is also a tool that provides access to much of its functionality from command line. This way it can be used by shell scripts or programming languages that do not have native bindings, but can run shell commands. [16]

The `openssl` binary is an entry point for all commands. You call it following the pattern:

```
openssl command [command_opts] [command_args]
```

Alternatively you can call it without arguments to enter the interactive mode with an `OpenSSL>` prompt. Then you can directly type your commands. You can leave the interactive mode with Ctrl+C or Ctrl+D or by typing `quit`.

You can get a list of available commands by calling:

```
openssl list-standard-commands
```

OpenSSL binary provides command-line access to the following significant cryptographic operations and applications:

- `openssl dgst` – a message digest command, producing or verifying a digest of supplied file(s) using hash functions or digital signature algorithms
- `openssl enc` – a symmetric cipher command, allowing data to be encrypted or decrypted using various block and stream ciphers, using keys derivated from passwords or explicitly provided

- `openssl speed` – a benchmark to test the performance of all included cryptographic algorithms
- `openssl asn1parse` – a diagnostic parser of ASN.1 encoded structures; public and private keys, certificates and other cryptographic structures are usually stored in ASN.1 format
- `openssl x509` – a multi-purpose utility to operate with X509 certificates
- `openssl s_server` – a generic TCP+TLS server which listens on a given local port, it can operate either in plain text mode, or as a simple HTTP server, processing requests and responding with files in current directory
- `openssl s_client` – a generic TCP+TLS client which connects to a remote host, very useful diagnostic tool
- `openssl s_time` – a client which benchmarks the performance of a TLS connection

OpenSSL supports a lot of cryptographic algorithms, some of them also have their own aliases (pseudo-commands) for faster command line access. Supported algorithms and their corresponding pseudo-commands can be listed by the following commands:

- `openssl list-message-digest-algorithms` – list of message digest algorithms
- `openssl list-message-digest-commands` – list of message digest pseudo-commands
- `openssl list-cipher-algorithms` – list of symmetric encryption algorithms
- `openssl list-cipher-commands` – list of symmetric encryption pseudo-commands
- `openssl list-public-key-algorithms` – list of public key algorithms
- `openssl ciphers -v [`*`cipherlist`*`]` – list of TLS cipher suites, complying with the given cipherlist, or all by default
- `openssl ecparam -list_curves` – list of named elliptic curves

4.3.1 Symmetric encryption

The `openssl enc` command[16] encrypts or decrypts given data using various supported ciphers. By default, it reads the data from standard input, a writes to standard output.

It accepts the following significant options:

- `-`*`ciphername`* – the cipher name, it specifies the requirements on the length of key and IV
- `-d` – decrypt the input data
- `-K` *`hex`* – the key used in cipher, it must be represented as a string comprised only of hex digits
- `-iv` *`hex`* – the IV used in cipher, it must be represented as a string comprised only of hex digits

Example:

```
ENC="openssl enc -aes-128-cbc"
PLAINTEXT="Lorem ipsum dolor sit amet, consectetur adipiscing elit."
KEY=123456789abcdef03456789abcdef012
IV=00000000000000000000000000000000

CIPHERTEXT=$(echo -n "$PLAINTEXT" | $ENC -K $KEY -iv $IV | xxd -p)
echo "$CIPHERTEXT"

ef45d7ca2e7a1cb2b61e412767974b23af5b6532fc92373b9433029c8a30
7fbe7737aac0fe4435ac5a3919884195469038e7345c61cb3cc205e570d8
a10a1f9d

PLAINTEXT2=$(echo -n "$CIPHERTEXT" | xxd -r -p | $ENC -d -K $KEY -iv $IV)
echo "$PLAINTEXT2"

Lorem ipsum dolor sit amet, consectetur adipiscing elit.
```

4.3.2 Performance benchmarking of cryptographic algorithms

The `openssl speed` command[17] can run performance benchmarks of all included cryptographic algorithms - hash functions, symmetric ciphers, assymetric key exchanges and digital signatures.

It accepts the following significant options:

- `-evp` *`algorithmname`* – run benchmarks on an EVP algorithm

[16] `https://www.openssl.org/docs/apps/enc.html`
[17] `https://www.openssl.org/docs/apps/speed.html`

- *algorithmnames* – if any algorithms are given, speed tests those algorithms, otherwise all are tested

Example:

```
openssl speed -evp aes-128-gcm

Doing aes-128-gcm for 3s on 16 size blocks: 43106948 aes-128-gcm's in 3.00s
Doing aes-128-gcm for 3s on 64 size blocks: 30559247 aes-128-gcm's in 2.99s
Doing aes-128-gcm for 3s on 256 size blocks: 11774457 aes-128-gcm's in 3.00s
Doing aes-128-gcm for 3s on 1024 size blocks: 3428879 aes-128-gcm's in 3.00s
Doing aes-128-gcm for 3s on 8192 size blocks: 444651 aes-128-gcm's in 3.00s
...
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
aes-128-gcm     229903.72k   654110.97k  1004753.66k  1170390.70k  1214193.66k
```

4.3.3 Generic server

The `openssl s_server` command[18] emulates a generic TCP server, which uses TLS to ensure a secure communication channel. It listens for incoming connections and after a connection is established, it forwards standard input to the opposite party through TLS data protocol, and writes all received data to standard output.

It accepts the following significant options:

- `-accept` *port* – the port to listen on for connections, 4433 by default
- `-cert` *filename* – the certificate, a self-signed certificate can be used
- `-key` *filename* – the certificate private key
- `-cipher` *ciphernames* – the supported cipher list. When the client sends a list of supported ciphers, the first client cipher also included in the server list is chosen. Because the client specifies the preference order, the order of the server cipherlist irrelevant. This behavior can be overriden by `-serverpref` option.
- `-serverpref` – use the server's cipher preferences, rather than the client's preferences
- `-WWW` – emulates a simple web server. Resources will be resolved relative to the current directory, for example if the resource `/page.html` is requested, the file `./page.html` will be loaded.

Example, run in parallel with `s_client` example:

[18] https://www.openssl.org/docs/apps/s_server.html

```
openssl s_server -accept 4444 -cert selfsigned.crt -key selfsigned.key -cipher
 ↪  DHE-RSA-AES128-GCM-SHA256

Using default temp DH parameters
Using default temp ECDH parameters
ACCEPT
-----BEGIN SSL SESSION PARAMETERS-----
...
-----END SSL SESSION PARAMETERS-----
Shared ciphers:DHE-RSA-AES128-GCM-SHA256
CIPHER is DHE-RSA-AES128-GCM-SHA256
Secure Renegotiation IS supported
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
DONE
shutting down SSL
CONNECTION CLOSED
ACCEPT
```

After the connection is established, a string "Lorem ipsum..." is successfully transferred from the client to the server.

4.3.4 Generic client

The `openssl s_client` command[19] emulates a generic TCP client, which uses TLS to ensure a secure communication channel. After a connection is established, it forwards standard input to the opposite party through TLS data protocol, and writes all received data to standard output.

It accepts the following significant options:

- `-connect` *`host:port`* – the host and port to connect to, local host and port 4433 by default
- `-cipher` *`ciphernames`* – the supported cipher list. Although the server determines which cipher suite is used, it should take the first supported cipher in the list sent by the client.

Example, run in parallel with `s_server` example:

```
openssl s_client -connect 127.0.0.1:4444 -cipher DHE-RSA-AES128-GCM-SHA256

CONNECTED(00000003)
depth=0 C = XX, L = Default City, O = Default Company Ltd
verify error:num=18:self signed certificate
verify return:1
depth=0 C = XX, L = Default City, O = Default Company Ltd
verify return:1
```

[19] `https://www.openssl.org/docs/apps/s_client.html`

```
---
Certificate chain
 0 s:/C=XX/L=Default City/O=Default Company Ltd
   i:/C=XX/L=Default City/O=Default Company Ltd
---
Server certificate
-----BEGIN CERTIFICATE-----
...
-----END CERTIFICATE-----
subject=/C=XX/L=Default City/O=Default Company Ltd
issuer=/C=XX/L=Default City/O=Default Company Ltd
---
No client certificate CA names sent
Server Temp Key: DH, 1024 bits
---
SSL handshake has read 1704 bytes and written 289 bytes
---
New, TLSv1/SSLv3, Cipher is DHE-RSA-AES128-GCM-SHA256
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : DHE-RSA-AES128-GCM-SHA256
...
    Verify return code: 18 (self signed certificate)
---
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
DONE
```

After the connection is established, a string "Lorem ipsum..." is successfully transferred from the client to the server.

4.3.5 Performance benchmarking of TLS

The `openssl s_time` command[20] emulates a generic TCP client, which uses TLS to ensure a secure communication channel. It can request a page from the server and includes the time to transfer the payload data in its timing measurements. It measures the number of connections within a given timeframe, the amount of data transferred (if any), and calculates the average time spent for one connection.

It accepts the following significant options:

- `-connect` ***`host:port`*** – the host and port to connect to, local host and port 4433 by default

[20]`https://www.openssl.org/docs/apps/s_time.html`

- `-cipher` *`ciphernames`* – the supported cipher list. Although the server determines which cipher suite is used, it should take the first supported cipher in the list sent by the client.
- `-time` *`sec`* – specifies how long in seconds the benchmark should run
- `-new` – performs the timing test using a new session for each connection. If `-www` option is not used, this can be used to benchmark specifically the TLS handshake protocol.
- `-reuse` – performs the timing test using the same session for each connection. If `-www` option is not used, this can be used to benchmark specifically the TLS handshake protocol with session resume.
- `-www` *`filename`* – this specifies the resource to GET from the server. If this parameter is not specified, then it will only perform the TLS handshake to establish a connections, but not transfer any data.

```
openssl s_time -connect 127.0.0.1:4444 -cipher DHE-RSA-AES128-GCM-SHA256 -time
↪  3 -new

Collecting connection statistics for 3 seconds
...
857 connections in 1.47s; 582.99 connections/user sec, bytes read 0
857 connections in 4 real seconds, 0 bytes read per connection


openssl s_time -connect 127.0.0.1:4444 -cipher DHE-RSA-AES128-GCM-SHA256 -time
↪  3 -reuse

Collecting connection statistics for 3 seconds
...
11834 connections in 1.11s; 10661.26 connections/user sec, bytes read 0
11834 connections in 4 real seconds, 0 bytes read per connection


openssl s_time -connect 127.0.0.1:4444 -cipher DHE-RSA-AES128-GCM-SHA256 -time
↪  3 -reuse -www /test-8k.dat

Now timing with session id reuse.
starting
...
96 connections in 0.03s; 3200.00 connections/user sec, bytes read 790752
96 connections in 4 real seconds, 8237 bytes read per connection


openssl s_time -connect 127.0.0.1:4444 -cipher DHE-RSA-AES128-GCM-SHA256 -time
↪  3 -reuse -www /test-1M.dat

Now timing with session id reuse.
starting
```

```
...
84 connections in 0.41s; 204.88 connections/user sec, bytes read 88084164
84 connections in 4 real seconds, 1048621 bytes read per connection
```

4.4 libcrypto library

The libcrypto[21] library provides high-level and low-level interfaces for the implemented fundamental cryptographic algorithms.

For most uses, users should use the high-level interface that is provided for performing cryptographic operations. This is known as the EVP[22] interface (short for Envelope). This interface provides a suite of functions for performing encryption/decryption (both symmetric and asymmetric), signing/verifying, as well as generating hashes and MAC codes, across the full range of OpenSSL supported algorithms and modes. Working with the high-level interface means that a lot of the complexity of performing cryptographic operations is hidden from view. A single consistent API is provided. In the event that you need to change your code to use a different algorithm (for example), then this is a simple change when using the high-level interface. In addition low-level issues such as padding and encryption modes are all handled for you.

The EVP functions provide a high-level interface to OpenSSL cryptographic functions. They provide the following features:

- A single consistent interface regardless of the underlying algorithm or mode
- Support for an extensive range of algorithms
- Encryption/Decryption using both symmetric and asymmetric algorithms
- Sign/Verify
- Key derivation
- Secure Hash functions
- Message Authentication Codes
- Support for external crypto engines

In addition to the high-level interface, OpenSSL also provides low-level interfaces for working directly with the individual algorithms. These low-level interfaces are not recommended for the novice user, but provide a degree of control that may not be possible when using only the high-level interface.

[21]https://wiki.openssl.org/index.php/Libcrypto_API
[22]https://wiki.openssl.org/index.php/EVP

Where possible, the high-level EVP interface should be used in preference to the low-level interfaces. This is because the code then becomes transparent to the algorithm used and much more flexible. Additionally, the EVP interface will ensure the use of platform specific cryptographic acceleration such as AES-NI. The low-level interfaces do not provide the guarantee.

4.4.1 EVP API

A specific cipher or message digest algorithm is identified by an unique `EVP_CIPHER` or `EVP_MD` struct respectively. OpenSSL user is not expected to create these themself, but instead they can use a built-in function to return the struct of particular algorithm that they wish to use. In the following text I focus only on cipher algorithms.

An extract from `evp.h` header file listing some functions returning `EVP_CIPHER` struct is shown below. There is a specific function for every supported combination of cipher algorithm and its parameters (a key length, a block mode, etc.).

```
const EVP_CIPHER *EVP_enc_null(void);
const EVP_CIPHER *EVP_rc4(void);
const EVP_CIPHER *EVP_aes_128_cbc(void);
const EVP_CIPHER *EVP_aes_128_gcm(void);
const EVP_CIPHER *EVP_aes_256_cbc(void);
const EVP_CIPHER *EVP_aes_256_gcm(void);
```

The `EVP_CIPHER` is represented by the following signature:

```
struct evp_cipher_st {
  int nid;
  int block_size;
  int key_len;
  int iv_len;
  unsigned long flags;
  int (*init)(EVP_CIPHER_CTX *ctx, const unsigned char *key, const unsigned
   ↪  char *iv, int enc);
  int (*do_cipher)(EVP_CIPHER_CTX *ctx, unsigned char *out, const unsigned
   ↪  char *in, size_t inl);
  int (*cleanup)(EVP_CIPHER_CTX *);
  int ctx_size;
  int (*set_asn1_parameters)(EVP_CIPHER_CTX *, ASN1_TYPE *);
  int (*get_asn1_parameters)(EVP_CIPHER_CTX *, ASN1_TYPE *);
  int (*ctrl)(EVP_CIPHER_CTX *, int type, int arg, void *ptr);
  void *app_data;
} EVP_CIPHER;
```

The meaning of its important parameters follows:

- `nid` – integer; unique identifier
- `block_size` – integer; cipher block size
- `key_len` – integer; cipher key length
- `iv_len` – integer; cipher IV length
- `flags` – bit array, represented as integer; cipher flags specifying its capabilities
- `init` – function pointer; called during initialization, it can allocate the cipher context before the operation
- `do_cipher` – function pointer; called during sending the input data, it should perform the encrypt/decrypt operation
- `cleanup` – function pointer; called during finalization, it can free the cipher context after the operation
- `ctx_size` – integer; cipher-specific context size, can be used to store cipher-specific data during the operation
- `ctrl` – function pointer; used to invoke special actions, which do not have a specific field in the `EVP_CAESAR` struct

Prior to performing the encrypt/decrypt operation, a cipher context must be allocated and initialized to store setting and state during the operation. The cipher context is represented by `EVP_CIPHER_CTX` struct.

```
EVP_CIPHER_CTX *EVP_CIPHER_CTX_new(void);
```

Note the cipher context is different from cipher-specific context. The cipher-specific context is stored inside of the cipher context, however the cipher-specific context is allocated and freed by the cipher itself in its `init` and `cleanup` functions, which are called by EVP internally, and a user does not need to care about it.

So now we are prepared to perform the encrypt/decrypt operation. First, we need to set a cipher, a secret key and an initialization vector (IV). We can use an initialization function, specifically named `EVP_EncryptInit_ex` or `EVP_DecryptInit_ex`. There is also an universal `EVP_CipherInit_ex` function with `enc` parameter, which controls which operation is performed. However I recommend the first option, because usually we are sure which operation do we want to perform, so it should be hardcoded using the specific functions instead of depending of an integer parameter. It internally calls the cipher's `init` function from the `EVP_CIPHER` struct.

```
int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *cipher, ENGINE
↪  *impl, const unsigned char *key, const unsigned char *iv);
int EVP_DecryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *cipher, ENGINE
↪  *impl, const unsigned char *key, const unsigned char *iv);
```

Any input data should be sent to the cipher by an update function `EVP_EncryptUpdate` or `EVP_DecryptUpdate`. It accepts the input data, and it fills the provided output buffer with the encrypted/decrypted output data, and returns a number of written bytes. Usualy it can be called multiple times, if the cipher supports streaming.

The functions internally calls the cipher's `do_cipher` function with the same parameters.

```
int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl,
↪  const unsigned char *in, int inl);
int EVP_DecryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl,
↪  const unsigned char *in, int inl);
```

After the input data block ends, a finalization function `EVP_EncryptFinal_ex` or `EVP_DecryptFinal_ex` needs to be called. This function can append few more bytes to output, for example a padding or an authentication tag.

The functions internally call the cipher's `do_cipher` function with null parameters to signalize the end of operation. The cipher can react by appending few more bytes to output, for example a padding or an authentication tag.

```
int EVP_EncryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl);
int EVP_DecryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *outm, int *outl);
```

After all cipher operations were finished, the cipher context must be cleaned up and freed by the `EVP_CIPHER_CTX_free` function. It internally calls the cipher's `cleanup` function.

```
void EVP_CIPHER_CTX_free(EVP_CIPHER_CTX *a);
```

4.4.2 EVP API – Symmetric encryption and decryption

With the knowledge of EVP API as described in subsection 4.4.1, a user can perform symmetric encryption and decryption operations across a wide range of algorithms and modes. The following code shows how to use it to encrypt and decrypt a piece of confidential information.

Encryption using the EVP API consists of the following stages:

- Setting up a cipher context

- Initializing the encryption operation, providing key, IV
- Providing plaintext bytes to be encrypted
- Finalizing the encryption operation

The sample `encrypt` function uses AES-128 in CBC mode. It takes as arguments the plaintext, the length of the plaintext, the key, and the IV. Also it takes in a buffer to put the ciphertext in (which we assume to be long enough), and will return the length of the ciphertext that it writtes.

The length of plaintext is necessary, OpenSSL cannot use `strlen` function to determine its length, because it can contain any data, even null (`\0`) bytes. The length of key and IV is fixed, appropriate for the chosen cipher, which means both the key and IV 16 bytes long for AES-128 in CBC mode.

The source code follows. It misses the most of error handling code, which would be necessary in a real application.

```
int encrypt(
    unsigned char *plaintext, int plaintext_length,
    unsigned char *key,
    unsigned char *iv,
    unsigned char *ciphertext
) {
    EVP_CIPHER *cipher;
    EVP_CIPHER_CTX *ctx;
    int length;
    int ciphertext_length;

    // get the cipher handle
    cipher = EVP_aes_128_cbc();

    // initialize the cipher context
    ctx = EVP_CIPHER_CTX_new();

    // initialize the encryption operation
    // the key and IV length should be appropriate for the chosen cipher
    EVP_EncryptInit_ex(ctx, cipher, NULL, key, iv);

    // provide the plaintext to be encrypted, and receive the ciphertext
    // this can be called multiple times as required
    EVP_EncryptUpdate(ctx, ciphertext, &length, plaintext, plaintext_length);
    ciphertext_length = length;

    // finalize the encryption operation
    // further ciphertext bytes may be received
    EVP_EncryptFinal_ex(ctx, ciphertext + ciphertext_length, &length);
    ciphertext_length += length;

    // cleanup the cipher context
    EVP_CIPHER_CTX_free(ctx);

    return ciphertext_length;
}
```

Decryption using the EVP API consists of the following steps:

- Setting up a cipher context
- Initializing the decryption operation, providing key, IV
- Providing ciphertext bytes to be decrypted
- Finalizing the decryption operation

The sample `decrypt` function uses AES-128 in CBC mode. It takes almost the same arguments as the `encrypt` function, with the exception that the plaintext and the ciphertext are swapped.

The source code follows. It misses the most of error handling code, which would be necessary in a real application.

```
int decrypt(
    unsigned char *ciphertext, int ciphertext_length,
    unsigned char *key,
    unsigned char *iv,
    unsigned char *plaintext
) {
    EVP_CIPHER *cipher;
    EVP_CIPHER_CTX *ctx;
    int length;
    int plaintext_length;
    int ret;

    // get the cipher handle
    cipher = EVP_aes_128_cbc();

    // initialize the cipher context
    ctx = EVP_CIPHER_CTX_new();

    // initialize the decryption operation
    // the key and IV length should be appropriate for the chosen cipher
    EVP_DecryptInit_ex(ctx, cipher, NULL, key, iv);

    // provide the ciphertext to be decrypted, and receive the plaintext
    // this can be called multiple times as required
    EVP_DecryptUpdate(ctx, plaintext, &length, ciphertext, ciphertext_length);
    plaintext_length = length;

    // finalize the decryption operation
    // further plaintext bytes may be received
    ret = EVP_DecryptFinal_ex(ctx, plaintext + plaintext_length, &length);
    plaintext_length += length;

    // cleanup the cipher context
    EVP_CIPHER_CTX_free(ctx);

    if (ret > 0) {
        // decryption successful
        return plaintext_length;
    } else {
        // decryption failed
        return -1;
    }
}
```

4.4.3 EVP API – Authenticated encryption and decryption

Following the recent advances in AEAD, the EVP API of libcrypto library also supports the ability to perform authenticated encryption and decryption. It provides confidentiality by encrypting the data, and authenticity by creating a MAC tag over the encrypted data.

Using AEAD ciphers is nearly identical to using standard symmetric encryption ciphers. In addition, a user can optionally provide some Additional Authenticated Data (AAD). The AAD data is not encrypted, and is typically passed to the recipient in plaintext along with the ciphertext. An example of AAD is the IP address and port number in a IP header used with IPsec.

The output from the encryption operation will be the ciphertext, and a MAC tag. The MAC tag is subsequently used during the decryption operation to ensure that the ciphertext and AAD have not been tampered with.

Authenticated encryption using the EVP API in much the same way as for symmetric encryption as described in subsection 4.4.2. The main differences are:

- AAD data can be provided before encrypting the plaintext data
- after the encryption is finished, the MAC tag needs to be obtained

The sample `encrypt` function uses AES-128 in GCM mode. It takes as arguments the plaintext, the length of the plaintext, the key, and the IV. Also it takes in a buffer to put the ciphertext and the MAC tag in (which we assume to be long enough), and will return the length of the ciphertext that it writtes.

The length of plaintext is necessary, OpenSSL cannot use `strlen` function to determine its length, because it can contain any data, even null (`\0`) bytes. The length of key, IV and MAC tag is fixed, appropriate for the chosen cipher, which means the key 16 bytes long, the IV 12 bytes long and the MAC tag 16 bytes for AES-128 in GCM mode by default.

The source code follows. It misses the most of error handling code, which would be necessary in a real application.

```c
int encrypt(
    unsigned char *plaintext, int plaintext_length,
    unsigned char *aad, int aad_length,
    unsigned char *key,
    unsigned char *iv,
    unsigned char *ciphertext,
    unsigned char *tag
) {
    EVP_CIPHER *cipher;
    EVP_CIPHER_CTX *ctx;
    int length;
    int ciphertext_length;

    // get the cipher handle
    cipher = EVP_aes_128_gcm();

    // initialize the cipher context
    ctx = EVP_CIPHER_CTX_new();

    // initialize the encryption operation
    // the key and IV length should be appropriate for the chosen cipher
    EVP_EncryptInit_ex(ctx, cipher, NULL, key, iv);

    // provide the AAD data to be authenticated
    // this can be called zero or more times as required
    EVP_EncryptUpdate(ctx, NULL, &length, aad, aad_length);

    // provide the plaintext to be encrypted and authenticated, and receive
    ↪  the ciphertext
    // this can be called multiple times as required
    EVP_EncryptUpdate(ctx, ciphertext, &length, plaintext, plaintext_length);
    ciphertext_length = length;

    // finalize the encryption operation
    // further ciphertext bytes may be received
    EVP_EncryptFinal_ex(ctx, ciphertext + ciphertext_length, &length);
    ciphertext_length += length;

    // receive the MAC tag
    EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG, 16, tag);

    // cleanup the cipher context
    EVP_CIPHER_CTX_free(ctx);

    return ciphertext_length;
}
```

Authenticated decryption using the EVP API in much the same way as for symmetric decryption as described in subsection 4.4.2. The main differences are:

- AAD data can be provided before decrypting the ciphertext data

- before the decryption is finished, the expected MAC tag needs to be provided
- a return value should be considered as a possible failure to authenticate ciphertext and/or AAD

The sample `decrypt` function uses AES-128 in GCM mode. It takes almost the same arguments as the `encrypt` function, with the exception that the plaintext and the ciphertext are swapped, and the MAC tag is provided by a user.

The source code follows. It misses the most of error handling code, which would be necessary in a real application.

```
int decrypt(
    unsigned char *ciphertext, int ciphertext_length,
    unsigned char *aad, int aad_length,
    unsigned char *tag,
    unsigned char *key,
    unsigned char *iv,
    unsigned char *plaintext
) {
    EVP_CIPHER *cipher;
    EVP_CIPHER_CTX *ctx;
    int length;
    int plaintext_length;
    int ret;

    // get the cipher handle
    cipher = EVP_aes_128_gcm();

    // initialize the cipher context
    ctx = EVP_CIPHER_CTX_new();

    // initialize the decryption operation
    // the key and IV length should be appropriate for the chosen cipher
    EVP_DecryptInit_ex(ctx, cipher, NULL, key, iv);

    // provide the expected MAC tag value
    EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_TAG, 16, tag);

    // provide the AAD data to be verified
    // this can be called zero or more times as required
    EVP_DecryptUpdate(ctx, NULL, &length, aad, aad_length);

    // provide the ciphertext to be decrypted and verified, and receive the
    ↪  plaintext
    // this can be called multiple times as required
    EVP_DecryptUpdate(ctx, plaintext, &length, ciphertext, ciphertext_length);
    plaintext_length = length;

    // finalize the decryption operation
    // further plaintext bytes may be received
    ret = EVP_DecryptFinal_ex(ctx, plaintext + plaintext_length, &length);
    plaintext_length += length;

    // cleanup the cipher context
    EVP_CIPHER_CTX_free(ctx);

    if (ret > 0) {
        // decryption successful
        return plaintext_length;
    } else {
        // decryption failed
        return -1;
    }
}
```

Currently, there is no standardized way to get and set the MAC tag for different ciphers. All current implementations use the universal `EVP_CIPHER_CTX_ctrl` function, which allows various cipher specific actions. However, the action identifier is specific for each cipher, e.g. `EVP_CTRL_GCM_GET_TAG`. This should change in future version of OpenSSL, there are new universal action identifiers such as `EVP_CTRL_AEAD_GET_TAG` in development code, used by both currently implemented AEAD ciphers (AES-GCM, AES-CCM).

4.5 libssl library

The libssl[23] library is a part of OpenSSL source code in `ssl` directory and provides an open-source implementation of TLS protocol. It depends on the libcrypto library, as introduced in section 4.4.

It consists of both server and client code, which consists of functions that generate and parse all TLS record messages. It calls a common interface of the negotiated cipher suite to relevant generic cryptographic operations. The cipher suite calls libcrypto code using either low-level or high-level EVP API.

Cipher suites which use the EVP API can take advantage from the common EVP interface, so such cipher suites can share code as well. Implementing a new cipher suite with EVP algorithms is as easy as adding a few declarations and almost no code at all.

[23] `https://wiki.openssl.org/index.php/Libssl_API`

Chapter 5

Implementing a new TLS cipher suite in OpenSSL

All CAESAR candidates publish its encrypt/decrypt primitives through the same CAESAR API, see subsection 2.1.2. I decided to implement a generic TLS cipher suite into OpenSSL in such way, that any CAESAR candidate can be used in the new cipher suite.

This chapter documents my source code added to OpenSSL 1.0.2. Basically I implemented a bridge between OpenSSL EVP API and CAESAR API. I can compile my customized OpenSSL with any CAESAR cipher, and it works. For my testing purposes, I chose the NORX cipher, see section 2.3.

There is no public documentation about implementing a new cipher or a new TLS cipher suite into OpenSSL, so I had to read through the OpenSSL source code a lot, tracing the code of already implemented ciphers and cipher suites.

The OpenSSL code is a joint work of many security experts, verified by real-world production usage and it can contain a lot of undocumented hidden knowledge. Because of no public documentation about this topic, I consider my implementation as experimental and I'm sure that my implementation is not perfect, it can contain hidden security bugs. Having this in mind, **I do not recommend my code for production use under any circumstances.**

5.1 Cipher

EVP API provides an universal interface to symmetric encryption, here a source code using it is independent on the chosen cipher. This is the main reason why I chose to implement a new cipher into the OpenSSL EVP API.

EVP API is a high-level interface to OpenSSL cryptographic functions. While OpenSSL also has direct interfaces for cryptographic operations, the EVP interface separates the operations from the actual backend used. That way, the actual implementation that is used can be changed, and one can specify an engine to use for the operations.

5.1.1 Implementation

I implemented a new abstract cipher named CAESAR, which serves as a bridge to its real implementation behind CAESAR API. Source code shown here is stripped to the most important parts, for full source code see attached files of this thesis.

I defined a new function in `crypto/evp/evp.h` file returning a reference to my new `EVP_CIPHER` struct.

```
const EVP_CIPHER *EVP_caesar(void);
```

I implemented all cipher related code in `crypto/evp/caesar/e_caesar.c` file. The cipher is defined by an `EVP_CIPHER` struct, which holds all cipher-cpecific setting and pointers to functions performing related operations. See section 4.4 for more detailed description of the EVP API.

```
typedef struct {
  unsigned char *key;
  unsigned char *nsec;
  unsigned char *npub;
  unsigned char *ad;
  size_t ad_length;
  int is_tls;
} EVP_CAESAR_CTX;

static int caesar_init(EVP_CIPHER_CTX *ctx, const unsigned char *key, const unsigned char *iv, int enc);
static int caesar_set_ad(EVP_CIPHER_CTX *ctx, const unsigned char *in, size_t in_length);
static int caesar_encrypt(unsigned char *c, unsigned long long *clen, const unsigned char *m, unsigned long long mlen, const unsigned char *ad, unsigned long long adlen, const unsigned char *nsec, const unsigned char *npub, const unsigned char *k);
static int caesar_decrypt(unsigned char *m, unsigned long long *mlen, unsigned char *nsec, const unsigned char *c, unsigned long long clen, const unsigned char *ad, unsigned long long adlen, const unsigned char *npub, const unsigned char *k);
static int caesar_cipher(EVP_CIPHER_CTX *ctx, unsigned char *out, const unsigned char *in, size_t in_length);
static int caesar_cleanup(EVP_CIPHER_CTX *ctx);
static int caesar_ctrl(EVP_CIPHER_CTX *ctx, int type, int arg, void *ptr);

static const EVP_CIPHER caesar = {
  NID_caesar,
  1,
  CRYPTO_KEYBYTES,
  CRYPTO_NPUBBYTES,
  EVP_CIPH_CUSTOM_IV | EVP_CIPH_FLAG_CUSTOM_CIPHER |
  ↪  EVP_CIPH_FLAG_AEAD_CIPHER,
  caesar_init,
  caesar_cipher,
  caesar_cleanup,
  sizeof(EVP_CAESAR_CTX),
  NULL,
  NULL,
  caesar_ctrl,
  NULL
};

const EVP_CIPHER *EVP_caesar(void) {
  return &caesar;
}
```

The cipher's `init` function (specifically `caesar_init`) initializes the cipher context in `EVP_CAESAR_CTX` struct, and copies the key and the IV into the context, so it can be used later by the `caesar_cipher` function.

```
static int caesar_init_key(EVP_CIPHER_CTX *ctx, const unsigned char *key,
↪  const unsigned char *iv, int enc) {
  EVP_CAESAR_CTX *cipher_ctx = (EVP_CAESAR_CTX *)ctx->cipher_data;

  cipher_ctx->key = (unsigned char *)calloc(CRYPTO_KEYBYTES, sizeof(unsigned
  ↪  char));
  cipher_ctx->nsec = (unsigned char *)calloc(CRYPTO_NSECBYTES, sizeof(unsigned
  ↪  char));
  cipher_ctx->npub = (unsigned char *)calloc(CRYPTO_NPUBBYTES, sizeof(unsigned
  ↪  char));
  cipher_ctx->ad = (unsigned char *)calloc(1, sizeof(unsigned char));
  cipher_ctx->ad_length = 0;
  cipher_ctx->is_tls = 0;

  memcpy(cipher_ctx->key, key, CRYPTO_KEYBYTES);
  memcpy(cipher_ctx->npub, iv, CRYPTO_NPUBBYTES);

  return 1;
}
```

The cipher's `do_cipher` function (specifically `caesar_cipher`) is the main processing function. It applies the cipher to the input data, and writes the result of the encrypt/decrypt operation to the output buffer. If the function is called with no output buffer, the input data is considered as associated data, which contributes to MAC tag.

```
static int caesar_cipher(EVP_CIPHER_CTX *ctx, unsigned char *out, const
↪  unsigned char *in, size_t in_length) {
  int ret = 0;
  unsigned long long out_length = 0;
  EVP_CAESAR_CTX *cipher_ctx = (EVP_CAESAR_CTX *)ctx->cipher_data;

  if (in_length > 0) {
    if (out == NULL) {
      caesar_set_ad(ctx, in, in_length);
    } else {
      // correct length for AEAD tag
      if (cipher_ctx->is_tls && ctx->encrypt) {
        in_length -= CRYPTO_ABYTES;
      }

      // message
      if (ctx->encrypt) {
        ret = caesar_encrypt(out, &out_length, in, in_length, cipher_ctx->ad,
         ↪  cipher_ctx->ad_length, cipher_ctx->nsec, cipher_ctx->npub,
         ↪  cipher_ctx->key);
      } else {
        ret = caesar_decrypt(out, &out_length, cipher_ctx->nsec, in,
         ↪  in_length, cipher_ctx->ad, cipher_ctx->ad_length,
         ↪  cipher_ctx->npub, cipher_ctx->key);
      }
    }
  }


  return ret == 0 ? (int)out_length : ret;
}
```

```
static int caesar_set_ad(EVP_CIPHER_CTX *ctx, const unsigned char *in, size_t
 ↪  in_length) {
  EVP_CAESAR_CTX *cipher_ctx = (EVP_CAESAR_CTX *)ctx->cipher_data;

  if (cipher_ctx->ad) {
    free(cipher_ctx->ad);
  }

  cipher_ctx->ad = (unsigned char *)calloc(in_length, sizeof(unsigned char));
  cipher_ctx->ad_length = in_length;
  memcpy(cipher_ctx->ad, in, in_length);

  return 1;
}
```

```
static int caesar_encrypt(unsigned char *c, unsigned long long *clen, const
 ↪  unsigned char *m, unsigned long long mlen, const unsigned char *ad,
 ↪  unsigned long long adlen, const unsigned char *nsec, const unsigned char
 ↪  *npub, const unsigned char *k) {
  int ret = crypto_aead_encrypt(c, clen, m, mlen, ad, adlen, nsec, npub, k);
   ↪  // @see CAESAR API

  return ret;
}
```

```
static int caesar_decrypt(unsigned char *m, unsigned long long *mlen, unsigned
↪  char *nsec, const unsigned char *c, unsigned long long clen, const
↪  unsigned char *ad, unsigned long long adlen, const unsigned char *npub,
↪  const unsigned char *k) {
  int ret = crypto_aead_decrypt(m, mlen, nsec, c, clen, ad, adlen, npub, k);
  ↪  // @see CAESAR API

  return ret;
}
```

The cipher's `cleanup` function (specifically `caesar_cleanup`) is used to cleanup and free all memory allocated by init functions.

```
static int caesar_cleanup(EVP_CIPHER_CTX *ctx) {
  EVP_CAESAR_CTX *cipher_ctx = (EVP_CAESAR_CTX *)ctx->cipher_data;

  free(cipher_ctx->key);
  free(cipher_ctx->nsec);
  free(cipher_ctx->npub);
  free(cipher_ctx->ad);

  return 1;
}
```

The cipher's `ctrl` function (specifically `caesar_ctrl`) is used to invoke special actions, which do not have a specific field in the `EVP_CIPHER` struct. I needed only one specific action, `EVP_CTRL_AEAD_TLS1_AAD` for setting associated data from TLS library.

```
static int caesar_ctrl(EVP_CIPHER_CTX *ctx, int type, int arg, void *ptr) {
  EVP_CAESAR_CTX *cipher_ctx = (EVP_CAESAR_CTX *)ctx->cipher_data;

  switch (type) {
    case EVP_CTRL_AEAD_TLS1_AAD: ; // empty statement
      int in_length = arg;
      unsigned char *in = (unsigned char *)ptr;

      cipher_ctx->is_tls = 1;

      if (!ctx->encrypt) {
        // correct length for AEAD tag
        // @see e_aes.c
        unsigned int len = in[in_length - 2] << 8 | in[in_length - 1];
        len -= CRYPTO_ABYTES;

        in[in_length - 2] = len >> 8;
        in[in_length - 1] = len & 0xff;
      }

      caesar_set_ad(ctx, in, in_length);
      return CRYPTO_ABYTES; // AEAD tag length
    default:
      return -1;
  }
}
```

Finally I registered the new cipher in `crypto/evp/c_allc.c` file. From now on, the new cipher is available for use by its name, `caesar`.

```
void OpenSSL_add_all_ciphers(void) {
  ...
  EVP_add_cipher(EVP_caesar());
}
```

5.1.2 Testing

First I checked if the new cipher is registered properly in OpenSSL.

```
apps/openssl list-cipher-algorithms

...
CAESAR
...
```

I created a simple Shell script to test my new cipher. It uses a fixed sample plaintext, and it generates a random key and IV.

It uses OpenSSL CLI command `enc` as described in subsection 4.3.1. It encrypts and decrypts a sample plaintext with my new `caesar` cipher.

In order to verify that my new cipher implementation works as intended, the following criterias must pass:

- no error is thrown
  An error can be thrown because of various reasons:
  - the new cipher is not implemented correctly
  - invalid padding found during decryption
  - invalid MAC tag found during decryption
- the output plaintext from decrypt operation equals to the original plaintext

- if any of the key, the IV, and the input plaintext changes, the ciphertext output changes as well
- if the key and IV is set to a fixed value, the ciphertext output from encrypt operation should stay the same

```
ENC="apps/openssl enc -caesar"
PLAINTEXT="Lorem ipsum dolor sit amet, consectetur adipiscing elit."
KEY=$(openssl rand 16 | xxd -p)
IV=$(openssl rand 16 | xxd -p)


echo "$PLAINTEXT"
echo "key=$KEY"
echo "iv=$IV"

CIPHERTEXT=$(echo -n "$PLAINTEXT" | $ENC -K $KEY -iv $IV | xxd -p)
echo
echo "$CIPHERTEXT"

PLAINTEXT2=$(echo -n "$CIPHERTEXT" | xxd -r -p | $ENC -d -K $KEY -iv $IV)
echo
echo "$PLAINTEXT2"


echo
if [ "$PLAINTEXT" == "$PLAINTEXT2" ]; then
  echo "ok"
else
  echo "fail"
fi
```

The script's output follows.

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
key=7bd48e478c585ea9b43647b6e3d61a9e
iv=b556987d0755badb75b4178df8516b67

f42d77fd83ee1899b6ca0b852abcea796660f0be27ccefd1fd2ce220694b
a8fe154501fa0cde39332cb37099194a307203bb3b78294b9f4fc80f1c48
d478a7c611112982437f7a70

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

ok
```

Note that OpenSSL CLI `enc` command currently does not support any way to provide additional authenticated data. Actually it does not support AEAD ciphers at all, there is a condition checking for AEAD flag in the cipher's flags field, which throws an error. I commented off this condition, because I needed this command to test the new cipher. There is no harm in doing so, additional authenticated data can be empty, and the MAC tag is still checked by the cipher itself.

```
int MAIN(int argc, char **argv) {
  ...
  /*
  if (cipher && EVP_CIPHER_flags(cipher) & EVP_CIPH_FLAG_AEAD_CIPHER) {
    BIO_printf(bio_err, "AEAD ciphers not supported by the enc utility\n");
    goto end;
  }
  */
  ...
}
```

5.2 TLS cipher suite

Any TLS cipher suite consists of various selected cryptographic primitives, providing key exchange, authentication, encryption and MAC. My thesis focuses only on a new encryption algorithm, so I implemented a new TLS cipher suite, which consists of a secure choice of Diffie-Hellman ephemeral (DHE) key exchange, RSA authentication, CAESAR encryption (see section 5.1 for details about implementation) and SHA256 pseudorandom function. There is no MAC, because MAC is already provided by the CAESAR AEAD cipher itself.

I named the new cipher suite to be consistent with existing cipher suites, `TLS_DHE_RSA_WITH_CAESAR_SHA256` by IANA conventions and `DHE-RSA-CAESAR-SHA256` by OpenSSL conventions.

The new cipher suite which can be negotiated by client and server in TLS handshake (see section 3.3 for details). During the handshake, cipher suites are represented by IDs. Public cipher suites are registered by IANA organization[24] and they are assigned with unique IDs, which are known to all parties. I did not want to publish the new cipher suite to production, I consider it experimental, so I used a private space of IDs: *"All cipher suites whose first byte is 0xFF are considered private and can be used for defining local/experimental algorithms."* [17] So I chose the ID for the new cipher suite to be `FF81`.

5.2.1 Implementation

OpenSSL code related to TLS cipher suites is capable of using EVP API, thus it was really simple to implement the new cipher suite using the new cipher from section 5.1.

[24]`http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml`

First I added a few new constants to `ssl/tls1.h`, `ssl/ssl_locl.h` and `ssl/t1_trce.c` files, defining IDs and names of the new cipher suite.

```
#define TLS1_CK_DHE_RSA_WITH_CAESAR_SHA256 0x0300FF81
#define TLS1_TXT_DHE_RSA_WITH_CAESAR_SHA256 "DHE-RSA-CAESAR-SHA256"
```

```
#define SSL_CAESAR 0x00004000L
```

```
static ssl_trace_tbl ssl_ciphers_tbl[] = {
  ...
  {0xFF81, "TLS_DHE_RSA_WITH_CAESAR_SHA256"},
  ...
};
```

I added a new cipher suite definition to `ssl/s3_lib.c` file, where I referenced the new cipher using the new constants.

```
OPENSSL_GLOBAL SSL_CIPHER ssl3_ciphers[] = {
  ...
  {
    1,
    TLS1_TXT_DHE_RSA_WITH_CAESAR_SHA256,
    TLS1_CK_DHE_RSA_WITH_CAESAR_SHA256,
    SSL_kEDH,
    SSL_aRSA,
    SSL_CAESAR,
    SSL_AEAD,
    SSL_TLSV1_2,
    SSL_NOT_EXP | SSL_HIGH,
    SSL_HANDSHAKE_MAC_SHA256 | TLS1_PRF_SHA256,
    128,
    128,
  },
  ...
};
```

In `ssl/ssl_ciph.c` file I found every code responsible for finding a correct EVP cipher by the new constants, and I added there a condition branch returning the new cipher.

```
static const SSL_CIPHER cipher_aliases[] = {
  ...
  {0, SSL_TXT_CAESAR, 0, 0, 0, SSL_CAESAR, 0, 0, 0, 0, 0},
  ...
};

```

```
void ssl_load_ciphers(void) {
  ...
  ssl_cipher_methods[SSL_ENC_CAESAR_IDX] = EVP_get_cipherbyname(SN_caesar);
  ...
}

int ssl_cipher_get_evp(const SSL_SESSION *s, const EVP_CIPHER **enc, const
 ↪  EVP_MD **md, int *mac_pkey_type, int *mac_secret_size, SSL_COMP **comp) {
  ...
  switch (c->algorithm_enc) {
    ...
    case SSL_CAESAR:
      i = SSL_ENC_CAESAR_IDX;
      break;
    ...
  }
  ...
}

static void ssl_cipher_get_disabled(unsigned long *mkey, unsigned long *auth,
 ↪  unsigned long *enc, unsigned long *mac, unsigned long *ssl) {
  ...
  *enc |= (ssl_cipher_methods[SSL_ENC_CAESAR_IDX] == NULL) ? SSL_CAESAR : 0;
  ...
}

char *SSL_CIPHER_description(const SSL_CIPHER *cipher, char *buf, int len) {
  ...
  switch (alg_enc) {
    ...
    case SSL_CAESAR:
      enc = "CAESAR";
      break;
    ...
  }
  ...
}
```

5.2.2 Testing

I used OpenSSL CLI commands `s_server` and `s_client` as described in subsection 4.3.3 and subsection 4.3.4. I run them simultaneously, each command in a separate terminal on a single computer. Server and client are connected to each other byu specifying the same single port `4444`.

The `s_server` command creates a TCP server. It listens on a specified port (`4444`) and forces negotiation of the new ciphersuite `DHE-RSA-CAESAR-SHA256`. It authenticates itself by a self-signed certificate, which I created only for purpose of testing the new cipher suite.

The `s_client` command creates a TCP client. It connects to the specified host and port (`localhost:4444`) and forces negotiation of the new ciphersuite

`DHE-RSA-CAESAR-SHA256`.

In order to verify that my new cipher implementation works as intended, the following criterias must pass:

- no error is thrown
  An error can be thrown because of various reasons:
  - the new cipher is not implemented correctly (this was verified in section 5.1)
  - the new cipher suite is not implemented correctly
- a two-way **communication channel** between the server and client is established
  Any data that I enter on server will be sent **unaltered** to client and vice versa.
- a two-way **secure communication channel** between the server and client is established
  Any data that I send via the channel will be sent **encrypted** during transmission.

I entered a fixed message (*"Lorem ipsum dolor sit amet, consectetur adipiscing elit.\n"*) to client and I watched it appear on server. The output from server and client reports that it is using the new cipher suite. The communication channel was successfully established.

```
apps/openssl s_client -connect 127.0.0.1:4444 -cipher DHE-RSA-CAESAR-SHA256

CONNECTED(00000003)
depth=0 C = XX, L = Default City, O = Default Company Ltd
verify error:num=18:self signed certificate
verify return:1
depth=0 C = XX, L = Default City, O = Default Company Ltd
verify return:1
---
Certificate chain
 0 s:/C=XX/L=Default City/O=Default Company Ltd
   i:/C=XX/L=Default City/O=Default Company Ltd
---
Server certificate
-----BEGIN CERTIFICATE-----
...
-----END CERTIFICATE-----
subject=/C=XX/L=Default City/O=Default Company Ltd
issuer=/C=XX/L=Default City/O=Default Company Ltd
---
No client certificate CA names sent
Peer signing digest: SHA512
Server Temp Key: DH, 512 bits
```

```
---
SSL handshake has read 1568 bytes and written 217 bytes
---
New, TLSv1/SSLv3, Cipher is DHE-RSA-CAESAR-SHA256
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : DHE-RSA-CAESAR-SHA256
    ...
    Verify return code: 18 (self signed certificate)
---
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
DONE
```

```
apps/openssl s_server -accept 4444 -cert selfsigned.crt -key selfsigned.key
↪  -cipher DHE-RSA-CAESAR-SHA256

Using default temp DH parameters
ACCEPT
-----BEGIN SSL SESSION PARAMETERS-----
...
-----END SSL SESSION PARAMETERS-----
Shared ciphers:DHE-RSA-CAESAR-SHA256
CIPHER is DHE-RSA-CAESAR-SHA256
Secure Renegotiation IS supported
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
DONE
shutting down SSL
CONNECTION CLOSED
ACCEPT
```

I watched the communication using the Wireshark application, which is capable of monitoring network traffic. It shows the TLS Handshake protocol (as described in section 3.3) messages negotiating the new cipher suite identified by ID `FF81` and the TLS Application protocol messages carrying the encrypted data.

In Wireshark, I logged all communication on local loopback network interface and filtered all TLS communication. See the packet capture Table 5.1. For the purpose of verifying my implementation, there are following important packets with TLS protocol messages:

- Packet 4 – ClientHello
- Packet 6 – ServerHello
- Packet 7 – ChangeCipherSpec

No.	Source	Destination	Content
4	client	server	ClientHello
6	server	client	ServerHello, Certificate, ServerKeyExchange, ServerHelloDone
7	client	server	ClientKeyExchange, ChangeCipherSpec, Finished (encrypted)
8	server	client	NewSessionTicket, ChangeCipherSpec, Finished (encrypted)
11	client	server	ApplicationData

Table 5.1: Packet capture

- Packet 8 – ChangeCipherSpec
- Packet 11 – ApplicationData

When the client initiates the connection, it sends the ClientHello message. It contains client's capabilites. The most important for me is the list of supported cipher suites. There is an unknown cipher suite *"Unknown (0xff81)"*, which represents the new cipher suite.

```
Frame 4: 165 bytes on wire (1320 bits), 165 bytes captured (1320 bits) on
↪  interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
↪  00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1
↪  (127.0.0.1)
Transmission Control Protocol, Src Port: 35287 (35287), Dst Port: 4444 (4444),
↪  Seq: 1, Ack: 1, Len: 99
Secure Sockets Layer
    TLSv1.2 Record Layer: Handshake Protocol: Client Hello
        Content Type: Handshake (22)
        Version: TLS 1.0 (0x0301)
        Length: 94
        Handshake Protocol: Client Hello
            Handshake Type: Client Hello (1)
            Length: 90
            Version: TLS 1.2 (0x0303)
            Random
            Session ID Length: 0
            Cipher Suites Length: 4
            Cipher Suites (2 suites)
                Cipher Suite: Unknown (0xff81)
                Cipher Suite: TLS_EMPTY_RENEGOTIATION_INFO_SCSV (0x00ff)
            Compression Methods Length: 1
            Compression Methods (1 method)
            Extensions Length: 45
            Extension: SessionTicket TLS
            Extension: signature_algorithms
            Extension: Heartbeat
```

The server receives the ClientHello message, compares the client's capabilities with its own, selects the best connection parameters which are supported by both parties, and sends them back to the client in the ServerHello message. Again, there is an unknown cipher suite *"Unknown (0xff81)"*, which represents the new cipher suite.

```
Frame 6: 1416 bytes on wire (11328 bits), 1416 bytes captured (11328 bits) on
↪  interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
↪  00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1
↪  (127.0.0.1)
Transmission Control Protocol, Src Port: 4444 (4444), Dst Port: 35287 (35287),
↪  Seq: 1, Ack: 100, Len: 1350
Secure Sockets Layer
    TLSv1.2 Record Layer: Handshake Protocol: Server Hello
        Content Type: Handshake (22)
        Version: TLS 1.2 (0x0303)
        Length: 58
        Handshake Protocol: Server Hello
            Handshake Type: Server Hello (2)
            Length: 54
            Version: TLS 1.2 (0x0303)
            Random
            Session ID Length: 0
            Cipher Suite: Unknown (0xff81)
            Compression Method: null (0)
            Extensions Length: 14
            Extension: renegotiation_info
            Extension: SessionTicket TLS
            Extension: Heartbeat
    TLSv1.2 Record Layer: Handshake Protocol: Certificate
    TLSv1.2 Record Layer: Handshake Protocol: Server Key Exchange
    TLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
```

The client and server now performs a check if the secret communication channel was sucessfully established. They send a ChangeCipherSpec message signalizing that all subsequent communitaion is encrypted by the selected cipher suite and a Finished message containing some data known to both parties. If decryption succeeds, they can start transmitting ApplicationData messages.

There is a single ApplicationData message which does not contain any plaintext data. Nobody should be able to decrypt it, because the secret key is exchanged assymetrically by Diffie-Hellman algorithm. I can confirm that it contains my message by comparing plaintext and ciphertext lengths.

```
Frame 11: 144 bytes on wire (1152 bits), 144 bytes captured (1152 bits) on
↪  interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst:
↪  00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1
↪  (127.0.0.1)
Transmission Control Protocol, Src Port: 35287 (35287), Dst Port: 4444 (4444),
↪  Seq: 218, Ack: 1569, Len: 78
Secure Sockets Layer
    TLSv1.2 Record Layer: Application Data Protocol: Application Data
        Content Type: Application Data (23)
        Version: TLS 1.2 (0x0303)
        Length: 73
        Encrypted Application Data:
         ↪  63be9025dec52eb3fbae5c54560c5027de0a10af19eceb7e...

0000  00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00   ..............E.
0010  00 82 ff e4 40 00 40 06 3c 8f 7f 00 00 01 7f 00   ....@.@.<.......
0020  00 01 89 d7 11 5c c9 80 6f aa 2c 9d ee 1d 80 18   .....\..o.,.....
0030  05 6a fe 76 00 00 01 01 08 0a 01 95 d8 25 01 95   .j.v.........%..
0040  91 17 17 03 03 00 49 63 be 90 25 de c5 2e b3 fb   ......Ic..%.....
0050  ae 5c 54 56 0c 50 27 de 0a 10 af 19 ec eb 7e 3e   .\TV.P'.......~>
0060  d0 2f 58 6a 1f 30 ff d1 8e ec 2f 20 70 eb 12 64   ./Xj.0..../ p..d
0070  96 01 b2 0a 54 24 06 ec cb 39 ad 15 54 e0 e9 4f   ....T$...9..T..O
0080  7b 15 d3 cc ca 48 fb 88 ec a9 9d 3c 7f 5d fb 22   {....H.....<.]."
```

The plaintext message sent from client to server is 57 bytes long. The ciphertext as observed in packet capture is 73 bytes long. The cipher is an AEAD stream cipher with MAC tag 16 bytes long, thus the ciphertext length should equal the plaintext length + the MAC tag length. $L_{ciphertext} = L_{plaintext} + L_{MAC}$, $73 + 57 + 16$ It confirms that my implementation works as intended.

Conclusion

The TLS protocol consists of a complex structure of standardized RFC documents by IETF. I studied the most important ones to understand the flow of the TLS protocol over the network. I examined its implementation in the OpenSSL library, which consists of a almost two decades old, complicated C code. Nonetheless this library is massively used in the real world and it is an important part of critical infrastructure of the Internet.

I studied the field of authenticated encryption while following the course of the CAESAR competition and recently published relevant papers. I classified CAESAR submissions by their design principles, overall construction and underlying primitives. I presented functional requirements and selection criteria for the winning candidates.

During the time of writing this thesis, the announcement of second-round candidates keeps being postponed. I did not have enough information to make a quallified choice, so I decided to implement a cipher using the generic CAESAR API.

As a reference for my implementation I chose the NORX cipher, because it has received no negative analysis. However it is not important which particular cipher I used. It can be easily substituted for a different one complying with the CAESAR API, as soon as the second-round or the final candidates are announced.

I successfully tested my implementation by observing TLS network communication between server and client.

I consider my implementation as experimental. It can serve as a reference for future work. I do not recommend it for production deployment under any circumstances.

Bibliography

[1] Rivest, R. L.; Shamir, A.; Adleman, L. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM*, volume 21, no. 2, Feb. 1978: pp. 120–126, ISSN 0001-0782. Available from: `http://doi.acm.org/10.1145/359340.359342`

[2] Ferguson, N.; Schneier, B.; Kohno, T. *Cryptography Engineering: Design Principles and Practical Applications*. Indianapolis, IN: Wiley Pub., Inc, 2010, ISBN 978-0470474242.

[3] Rogaway, P. Authenticated-Encryption with Associated-Data. In *ACM Conference on Computer and Communications Security (CCS'02)*, ACM press, 2002, pp. 98–107.

[4] Bellare, M.; Namprempre, C. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In *Advances in Cryptology – ASIACRYPT 2000*, *Lecture Notes in Computer Science*, volume 1976, edited by T. Okamoto, Springer Berlin Heidelberg, 2000, ISBN 978-3-540-41404-9, pp. 531–545. Available from: `http://dx.doi.org/10.1007/3-540-44448-3_41`

[5] Krawczyk, H. The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?). In *Advances in Cryptology – CRYPTO 2001*, *Lecture Notes in Computer Science*, volume 2139, edited by J. Kilian, Springer Berlin Heidelberg, 2001, ISBN 978-3-540-42456-7, pp. 310–331. Available from: `http://dx.doi.org/10.1007/3-540-44647-8_19`

[6] Crypto competitions. Accessed: 2015-03-14. Available from: `http://competitions.cr.yp.to/`

[7] Namprempre, C.; Rogaway, P.; Shrimpton, T. AE5 Security Notions: Definitions Implicit in the CAESAR Call. Cryptology ePrint Archive, Report 2013/242, 2013. Available from: `http://eprint.iacr.org/2013/242`

[8] Farzaneh Abed, S. L., Christian Forler. General Overview of the First-Round CAESAR Candidates for Authenticated Ecryption. Cryptology ePrint Archive, Report 2014/792, 2014. Available from: `http://eprint.iacr.org/2014/792`

[9] Bogdanov, A.; Lauridsen, M. M.; Tischhauser, E. AES-Based Authenticated Encryption Modes in Parallel High-Performance Software. Cryptology ePrint Archive, Report 2014/186, 2014. Available from: `http://eprint.iacr.org/2014/186`

[10] Bertoni, G.; Daemen, J.; Peeters, M.; et al. Sponge functions. Ecrypt Hash Workshop 2007, May 2007.

[11] Bertoni, G.; Daemen, J.; Peeters, M.; et al. Duplexing the sponge: single-pass authenticated encryption and other applications. In *Selected Areas in Cryptography (SAC)*, 2011.

[12] Ristic, I. *Bulletproof SSL and TLS: Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications*. London: Feisty Duck, 2014, ISBN 978-1907117046.

[13] Modadugu, N.; Rescorla, E. Datagram Transport Layer Security Version 1.2. RFC 6347, IETF, January 2012. Available from: `https://tools.ietf.org/html/rfc6347`

[14] Dent, A. *User's guide to cryptography and standards*. Boston, MA: Artech House, 2004, ISBN 1-58053-530-5.

[15] Beurdouche, B.; Bhargavan, K.; Delignat-Lavaud, A.; et al. A Messy State of the Union: Taming the Composite State Machines of TLS. In *IEEE Security & Privacy 2015*, 2015, accessed: 2015-05-02. Available from: `https://www.smacktls.com/smack.pdf`

[16] Viega, J. *Network security with OpenSSL*. Sebastopol, CA: O'Reilly, 2002, ISBN 978-0-596-00270-1.

[17] Dierks, T.; Allen, C. The TLS Protocol Version 1.0. RFC 2246, IETF, Januar 1999. Available from: `https://tools.ietf.org/html/rfc2246`

Appendix A

Symbols

m	plaintext message
c	ciphertext message
a	message authentication code
E	encrypt function
D	decrypt function
H	hash function
S	sign function
V	verify function
K	secret shared key
PK	public key
SK	secret (private) key

Appendix B

Acronyms

AAD Additional Authenticated Data

AE Authenticated Encryption

AEAD Authenticated Encryption with Associated Data

AES Anvanced Encryption Standard

AES-NI Advanced Encryption Standard - New Instructions

AVX Advanced Vector Extensions

CAESAR Competition for Authenticated Encryption: Security, Applicability and Robustness

DTLS Datagram Transport Layer Security

EtM Encrypt-then-MAC

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IETF Internet Engineering Task Force

IP Internet Protocol

ISO International Organization for Standardization

IV Initialization Vector

MAC Message Authentication Code

MitM Man-in-the-Middle

M&E MAC-and-Encrypt

MtE	MAC-then-Encrypt
OTP	One-Time Pad
PRF	Pseudorandom Function
RFC	Request for Comments
RTT	Round-Trip Time
SIMD	Single Instruction Multiple Data
SSE	Streaming SIMD Extensions
SSH	Secure Shell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol

Appendix C

TLS constants

Dec	Hex	Name
20	`0x14`	ChangeCipherSpec
21	`0x15`	Alert
22	`0x16`	Handshake
23	`0x17`	Application
24	`0x18`	Heartbeat

Table C.1: TLS record types

Major	Minor	Hex	Name
3	0	`0x0300`	SSL 3.0
3	1	`0x0301`	TLS 1.0
3	2	`0x0302`	TLS 1.1
3	3	`0x0303`	TLS 1.2

Table C.2: TLS versions

Dec	Hex	Name
0	`0x00`	HelloRequest
1	`0x01`	ClientHello
2	`0x02`	ServerHello
4	`0x04`	NewSessionTicket
11	`0x0b`	Certificate
12	`0x0c`	ServerKeyExchange
13	`0x0d`	CertificateRequest
14	`0x0e`	ServerHelloDone
15	`0x0f`	CertificateVerify
16	`0x10`	ClientKeyExchange
20	`0x14`	Finished

Table C.3: TLS handshake types

Dec	Hex	Name
1	`0x01`	warning
2	`0x02`	fatal

Table C.4: TLS alert levels

Dec	Hex	Name	Level
20	`0x14`	Bad record MAC	fatal
21	`0x15`	Decryption failed	fatal
22	`0x16`	Record overflow	fatal
41	`0x29`	No certificate	warning/fatal
43	`0x2b`	Unsupported certificate	warning/fatal
45	`0x2d`	Certificate expired	warning/fatal
48	`0x30`	Unknown CA	fatal
49	`0x31`	Access denied	fatal
50	`0x32`	Decode error	fatal
51	`0x33`	Decrypt error	warning/fatal
60	`0x3c`	Export restriction	fatal
70	`0x46`	Protocol version	fatal
71	`0x47`	Insufficient security	fatal
80	`0x50`	Internal error	fatal
90	`0x5a`	User canceled	fatal
100	`0x64`	No renegotiation	warning
110	`0x6e`	Unsupported extension	warning
111	`0x6f`	Certificate unobtainable	warning
113	`0x71`	Bad certificate status response	fatal
114	`0x72`	Bad certificate hash value	fatal
115	`0x73`	Unknown PSK identity	fatal
120	`0x78`	No Application Protocol	fatal

Table C.5: TLS alert types

Value	Name	Key exchange	Authentication	Encryption	MAC	PRF
0004	TLS_RSA_WITH_RC4_128_MD5	RSA	RSA	RC4-128	MD5	protocol
000A	TLS_RSA_WITH_3DES_EDE_CBC_SHA	RSA	RSA	3DES-EDE-CBC	SHA1	protocol
0033	TLS_DHE_RSA_WITH_AES_128_CBC_SHA	DHE	RSA	AES-128-CBC	SHA1	protocol
C02F	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	ECDHE	RSA	AES-128-GCM	AEAD	SHA256
C02C	TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	ECDHE	ECDSA	AES-256-GCM	AEAD	SHA384

Table C.6: Sample TLS cipher suites and their security properties

Appendix D

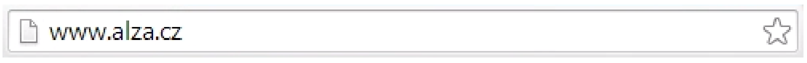
Security indication in browsers

If TLS is used in web browser in HTTPS protocol, the browser can show the state of certificate validation to the user. A certificate trusted by system is usually signalized by a green lock icon, while untrusted certificate displays a warning and the user can decide if he trusts the certificate or not manually.

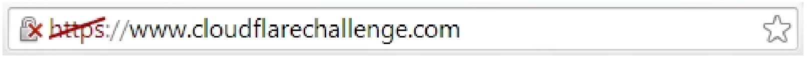
Additionally browsers can display a green bar with company name if the certificate is EV validated, in order to increase users' confidence of the server identity. It is useful in case of communication where the identity must be verfied with the lowest false positive possible, such as with banks or government.

We can divide browsers' indication of connection security to five categories:

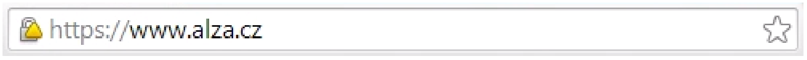
- No security
- Encrypted, failed authentication
  - self-signed certificate
  - expired certificate
  - forged certificate
- Encrypted, authenticated, with warning
  - mixed content from secured and unsecured sources
  - deprecated cryptographic primitives
- Encrypted, authenticated
  - domain ownership is validated
- Encrypted, authenticated with *Extended Validation* (EV)
  - domain ownership and company existence is validated


www.alza.cz

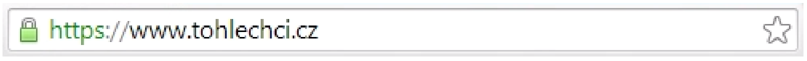
(a) No security


~~https~~://www.cloudflarechallenge.com

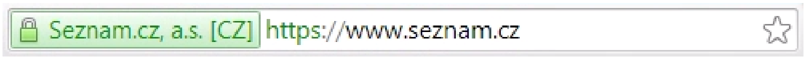
(b) Encrypted, failed authentication


https://www.alza.cz

(c) Encrypted, authenticated, with warning

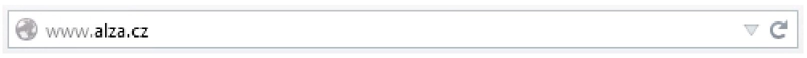

https://www.tohlechci.cz

(d) Encrypted, authenticated

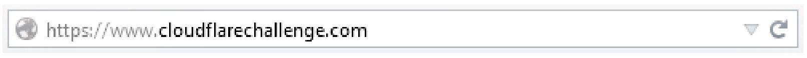

Seznam.cz, a.s. [CZ] https://www.seznam.cz

(e) Encrypted, authenticated with EV

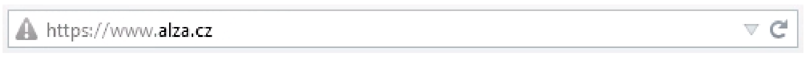
Figure D.1: Browser security indication - Chrome 40


www.alza.cz

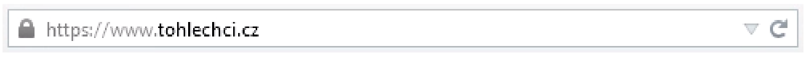
(a) No security


https://www.cloudflarechallenge.com

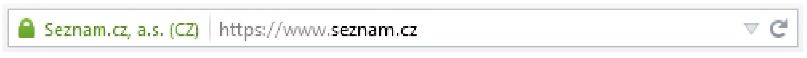
(b) Encrypted, failed authentication


https://www.alza.cz

(c) Encrypted, authenticated, with warning

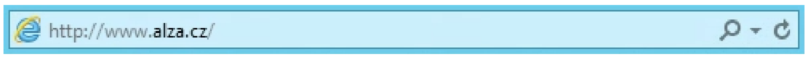

https://www.tohlechci.cz

(d) Encrypted, authenticated

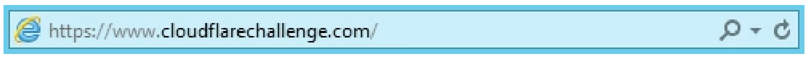

Seznam.cz, a.s. (CZ) https://www.seznam.cz

(e) Encrypted, authenticated with EV

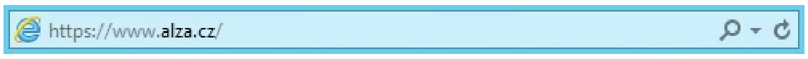
Figure D.2: Browser security indication - Firefox 35


http://www.alza.cz/

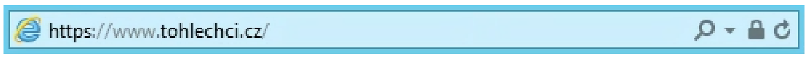
(a) No security


https://www.cloudflarechallenge.com/

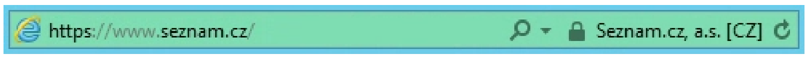
(b) Encrypted, failed authentication


https://www.alza.cz/

(c) Encrypted, authenticated, with warning

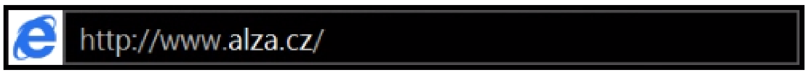

https://www.tohlechci.cz/

(d) Encrypted, authenticated

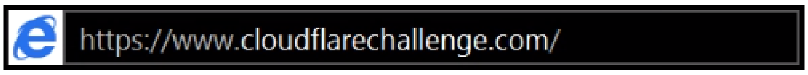

https://www.seznam.cz/ Seznam.cz, a.s. [CZ]

(e) Encrypted, authenticated with EV

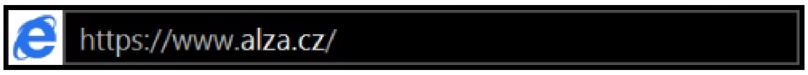
Figure D.3: Browser security indication - IE 11


http://www.alza.cz/

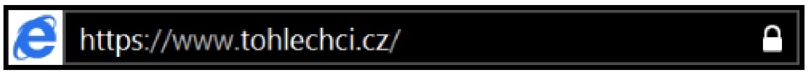
(a) No security


https://www.cloudflarechallenge.com/

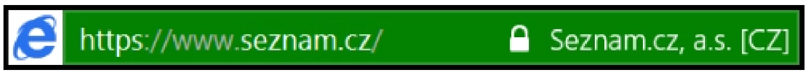
(b) Encrypted, failed authentication


https://www.alza.cz/

(c) Encrypted, authenticated, with warning

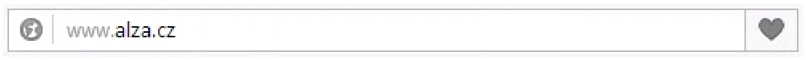

https://www.tohlechci.cz/

(d) Encrypted, authenticated

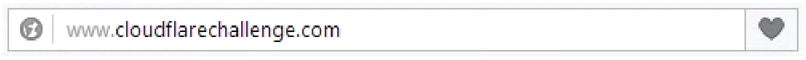

https://www.seznam.cz/ Seznam.cz, a.s. [CZ]

(e) Encrypted, authenticated with EV

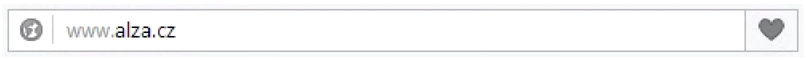
Figure D.4: Browser security indication - IE 11 (Metro)



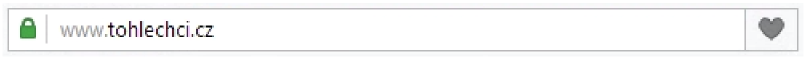
(a) No security



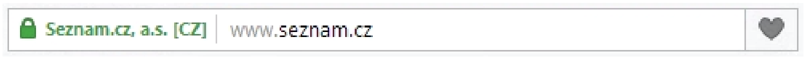
(b) Encrypted, failed authentication



(c) Encrypted, authenticated, with warning

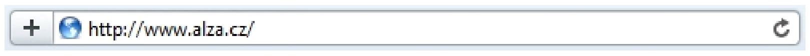


(d) Encrypted, authenticated

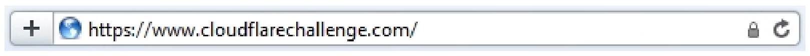


(e) Encrypted, authenticated with EV

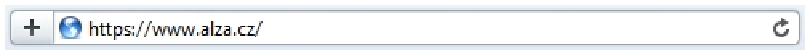
Figure D.5: Browser security indication - Opera 27



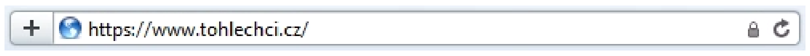
(a) No security



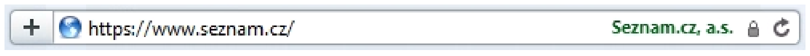
(b) Encrypted, failed authentication



(c) Encrypted, authenticated, with warning



(d) Encrypted, authenticated



(e) Encrypted, authenticated with EV

Figure D.6: Browser security indication - Safari 5.1

Appendix E

Content of attached CD

`README.md`	the file with CD contents description
`DP_Zak_Jan_2015.pdf`	the thesis text in PDF format
`DP_Zak_Jan_2015.tar.gz`	the archive of LaTeX source codes of the thesis
`openssl-1.0.2.tar.gz`	the archive of original OpenSSL 1.0.2
`openssl-1.0.2-caesar.tar.gz`	the archive of OpenSSL 1.0.2 with the new cipher suite, testing scripts and network capture between server and client
`supercop-20141124.tar.bz2`	the archive of SUPERCOP benchmarking suite, with the CAESAR candidates' source codes in `crypto_aead` directory