Masaryk University
Faculty of Informatics

Security and cryptography in GO

Bachelor's Thesis

Lenka Svetlovská

Brno, Spring 2017

Replace this page with a copy of the official signed thesis assignment and a copy of the Statement of an Author.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Lenka Svetlovská

Advisor: RNDr. Andriy Stetsko, Ph.D.

Acknowledgement

I would like to thank RNDr. Andriy Stetsko, Ph.D. for the management of the bachelor thesis, valuable advice, and comments. I would also like to thank the consultant from Y Soft Corporation a.s., Mgr. Lenka Bačinská, for useful advice, dedicated time and patience in consultations and application development.

Likewise, I thank my family and friends for their support throughout my studies and writing this thesis.

Abstract

The aims of the thesis are twofold.

First, the thesis overviews and compares cryptographic libraries available for Go programming language.

Second, the thesis contains the implementation, tests and evaluation of a prototype of a client application able to generate asymmetric keys and certificate signing requests, connect to a server application using TLS_PSK (pre-shared key) cipher suite.

Keywords

Certificate, Cgo, Language C, Language Go, OpenSSL, Pre-shared key, Security, TLS, TLS_PSK

Contents

1 Introduction

Programming language Go belongs to languages which a main idea is providing the best tools for programmers even though it is not always the easiest or the most effective way. Language Go was made by Google Inc. company in 2007, and it has syntax derived from language C. Safety, speed and concurrency are its main characters. Go provides extensive options for cryptography such as encryption, hashing and work with certificates.

The aims of the thesis are twofold. First, it is necessary to examine features of language Go related security and cryptography, analyze libraries which Go contains and also search for available third-party libraries. The thesis contains the overview, evaluation and comparison existing cryptographic libraries based on their license, usability, supported cryptographic functions, such as symmetric and asymmetric encryption and hash functions. Next based on possibilities for the generation of cryptographic keys and certificates and support for certificate signing requests and different formats. It also compares support of higher level protocols such as SSL/TLS.

Second, the thesis contains the implementation in language Go, tests and evaluation of a prototype of a client application able to generate asymmetric keys and certificate signing requests and connect to a server application using pre-shared key.

The bachelor thesis is consulted and realized in collaboration with Y Soft Corporation, a.s.

2 Background

In this chapter, I will define the basic terms which are necessary for understanding of a text presented in the following sections. First I will describe a structure of certificates, standard X.509, certificate authority and signing process. I will explain self-signed certificate and formats of certificate view. Finally, I will describe protocols SSL/TLS and TLS pre-shared key (TLS_PSK) algorithm used in the application.

2.1 Certificate

The certificate is digitally signed data structure which contains a public key of certificate owner. There are some standards of the certificate structure (X.509 (see subsection 2.1.1), EDI - Electronic Data Interchange [1], WAP - Wireless Application Protocol [2], etc.). On the Internet the mostly used standard is X.509 version 3, which was issued by ITU [3].

2.1.1 Standard X.509

Standard X.509 was originally designed as an authentication framework for X.500 directories. They have a hierarchical structure, and the individual attributes can be assigned to specific computers of companies or individual printers. Each entity can be clearly identified, and each has its private and public key [4].

An X.509 certificate, see figure 2.1, contains information about the identity to which a certificate is issued and the identity that issued it.

The first version was very simple and nowadays it is already inadequate. It contained 7 fields:

- certificate version;
- certificate series number;
- algorithm which signed certificate;
- name of certificate authority which released the certificate;
- identity of certificate owner;
- public key of certificate owner;
- certificate validity.

The second version contained only two new fields:

- unique identifier of the certificate owner;
- unique identifier of the certificate authority.

The third version was introduced in 1996. Its biggest and most important benefit is support of the expansion. It eliminates the majority of shortcomings of previous versions [5].

An expansion field permits any number of additional fields to be added to the certificate. Certificate extensions provide a way of adding information such as alternative subject names and usage restrictions to certificates [6].

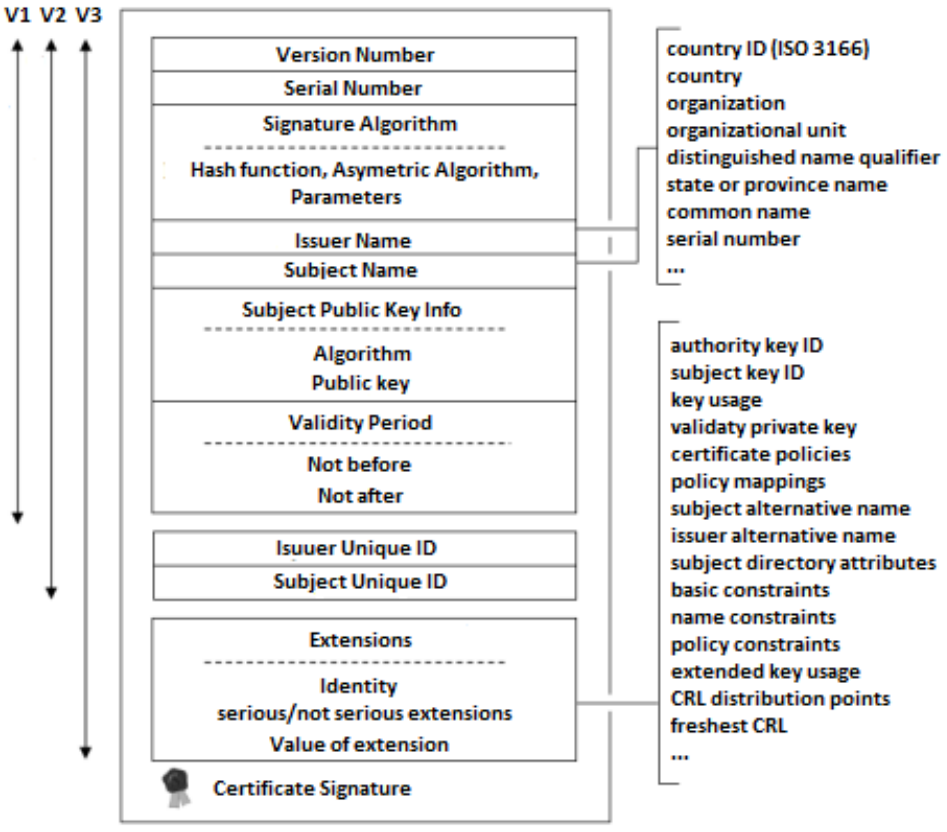

Figure 2.1: Structure of X.509 Certificate; translated from [7]

2.1.2 Certificate Authority

The Certificate Authority (CA) is an independent third party which issues certificates.

The CA arose to ensure secure encryption through a certified public key and guarantee the validity of a digital signature [8]. The CA also inspects a certificate request and provides validity revocation of certificate. In order to be truly credible, the CA must be impartial and, if it is possible, subordinate to some higher CA. Subordination means that it identifies the usage of certificates signed by a higher certificate authority [3].

Root CAs, certificate authorities at the highest level, use a certificate signed by themselves (self-signed certificate).

2.1.3 Self-signed Certificate

The self-signed certificate is a certificate which the owner issues by himself. The self-signed certificate has a same data structure as a regular certificate issued by CA. This certificate can be recognized by equality of Subject and Issuer items.

As proof of a private key possession, the self-signed certificate uses a digital signature which was made using the private key belonging to a public key in the certificate. Verification of the signature is carried out through a public key in the actual certificate [3].

2.1.4 Certificate Signing Request

The certificate applicant can request CA to sign a certificate by submitting a data structure called a certificate request (CSR). The most common format is PKCS#10 [3]. The CSR should include:

- Applicant ID
- Public key
- Evidence of the possession of the private key
- Other information that the owner wishes to insert in certificate
- May contain evidence of the data generation
- Data necessary for billing (in case issuance of certificates is paid)
- Passwords for communication with CA:
  - One-time password for issuing the certificate

- One-time password for certificate revocation
- Permanent password for personal (non-electronic) communication between owner and CA
- Phrase (useful in case losing all passwords; for example mother's maiden name)

A certificate authority must fill individual fields of the certificate before it signs a result digitally. The CA takes CSR, checks completed fields, and fills two important fields called validity period - the time while the certificate is valid - which are shown as: not before, not after. The CA can also add extensions, such as Subject Alternative Name, information about CA, etc. (see figure 2.1).

2.1.5 Formats PEM and DER

The PEM (Privacy Enhanced Mail) is file format for storing and sending cryptography keys, certificates, and other data. It is designed to be secure for inclusion in ASCII or even rich-text documents. This means that it is easy to copy and paste a content of a PEM file to another document and back [9]. The following figure, 2.2, is a sample PEM file containing a private key and a certificate.

Figure 2.2: A sample example of certificate view

A certificate or key must start with a header where the number of dashes is important and must be correct. The certificate starts with: "- - - - -BEGIN CERTIFICATE- - - - -" and ends with: "- - - - -END CERTIFICATE- - - - -". The key, for example RSA private key, starts with: "- - - - -BEGIN RSA PRIVATE KEY- - - - -" and ends with: "- - - - -END RSA PRIVATE KEY- - - - -". A single PEM file can contain a number of certificates and a key, for example, a single file with a public certificate, intermediate certificate, root certificate or private key.

The DER (Distinguished Encoding Rules) is a binary form of ASCII PEM format certificate. DER is used to represent keys, certificates and such in a portable format. All types of certificates and private keys can be encoded in DER format. Its information is stored in a binary DER for ASN.1 [10]. Applications providing RSA, SSL and TLS should handle DER encoding to read an information [11]. DER can not contain more than one certificate or one key.

2.2 Protocol SSL/TLS

Protocol SSL/TLS is used for secure communication between a client and a server. Protocol SSL was developed by Netscape Communications which published three versions. The first version was only a test; the second has been used in practice. However, it still contained security vulnerabilities; the most important was susceptibility to attack *man in the middle* [12]. The third version was introduced in 1996, and its specification could be found in the document *The SSL Protocol Version 3.0* (SSLv3.0) [13].

TLS protocol was created from *SSLv3.0* and it is currently the most widespread and supported protocol [12]. There are currently three versions which have only minimal differences (for more details see [14]). *TLS version 1.3* (TLSv1.3) is a draft already used in multiple locations [15]. For example, the forthcoming *OpenSSL 1.1.1* release will include support for *TLSv1.3* [16].

Protocol SSL/TLS provides authentication of two communicating parties by using asymmetric encryption, message integrity by using MAC and confidentiality by encrypting all communications by selected symmetric cipher.

Protocol SSL/TLS is located between the application and a transport layer reference ISO/OSI model and consists of two main parts. They are *The Record Layer Protocol* and *Handshake Protocol* [12].

Record Layer Protocol (RLP)

RLP processes application data, performs fragmentation, compression and data encryption. On the other hand, it decrypts data again and verifies the checksums. RLP does not care about a type of encryption algorithm or encryption key setting. This information is from HP [12].

Handshake Protocol (HP)

HP is activated immediately after establishment of a connection and provides identification of communicating parties, negotiation of cryptographic algorithms, compression algorithms and other attributes. Then it creates *a master secret* from which are derived encryption keys, initiation vectors, and the MAC [12]. In overview, the steps involved in the TLS handshake are as follows:

1. The client sends a *ClientHello* message to a server, along with TLS version, a client's random value and supported cipher suites.

2. The server responds by sending a *ServerHello* message that contains the cipher suite chosen by the server from the list provided by the client, the session ID, and server's random value.

3. The server sends its digital certificate to the client for authentication and may request a certificate from the client. The server sends the *ServerHelloDone* message.

4. If the server requires a digital certificate for client authentication, the server sends a "client certificate request"; the client sends it.

5. (a) The client creates a random *Pre-Master Secret* and encrypts it with the public key from the server's certificate, sending the encrypted *Pre-Master Secret* to the server.

   (b) The server and client each generate the *Master Secret* and session keys based on the *Pre-Master Secret*.

6. The client sends *ChangeCipherSpec* notification to server to indicate that the client will start using the new session keys for hashing and encrypting messages. Client also sends *ClientFinished* message.

7. Server receives *ChangeCipherSpec* and switches its record layer security state to symmetric encryption using the session keys. Server sends *ServerFinished* message to the client.

Client and server can now exchange application data over the secured channel they established. All messages sent from client to server and from server to client are encrypted using session key [17, 18].

2.2.1 PSK Key Exchange Algorithm

Using pre-shared keys can, depending on the cipher suite, avoid the need for public key operations. This is useful if TLS is used in performance-constrained environments with limited CPU power.

Pre-shared keys may be more convenient from a key management point of view. For instance, in closed environments where the connections are mostly configured manually in advance, it may be easier to configure a PSK than to use certificates. Another case is when the parties already have a mechanism for setting up a shared secret key, and that mechanism could be used to "bootstrap" a key for authenticating a TLS connection.

The cipher suites with PSK key exchange algorithm use only symmetric key algorithms. Figure 2.3 illustrates the pre-shared key algorithm.

1. The client indicates to use pre-shared key authentication by including one or more PSK cipher suites in the *ClientHello* message.

2. If the TLS server also wants to use pre-shared keys, it selects one of the PSK cipher suites and places it in the *ServerHello* message.

3. The server can provide a PSK identity hint[1] in the *ServerKeyExchange* message. If no hint is provided, the *ServerKeyExchange* message is omitted.

4. The client indicates which key to use by including a PSK identity in the *ClientKeyExchange* message.

5. A *Certificate* and *CertificateRequest* payloads are omitted from the response.

6. The TLS handshake is authenticated using the *Finished* messages.

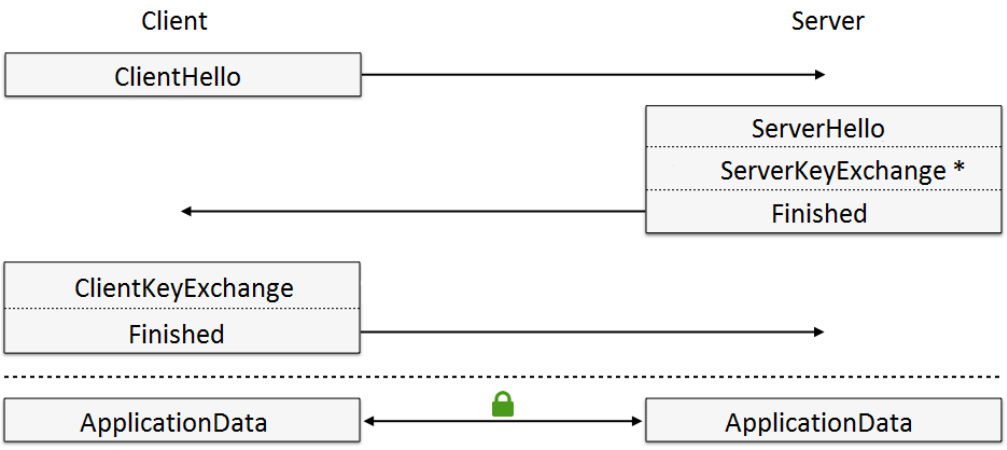

Figure 2.3: TLS_PSK Key Exchange; edited from [20]

If the server does not recognize the PSK identity, it may respond with an *unknown_psk_identity* alert message. Alternatively, if the server wishes to hide the fact that the PSK identity was not known, it may continue the protocol as if the PSK identity existed, but the key was incorrect: that is, respond to an *decrypt_error alert* [19].

The cipher suites with DHE_PSK key exchange algorithm use a PSK to authenticate a Diffie-Hellman exchange. These cipher suites

1. In a PSK exchange algorithm, both the client and the server have to be able to derive the same set of crypto keys. The identity hint is something the server provides to tell the client how to derive the key [19].

give some additional protection against dictionary attacks by passive eavesdroppers (but not active attackers) and also provide Perfect Forward Secrecy [21].

When these cipher suites are used, the *ServerKeyExchange* and *ClientKeyExchange* messages also include the Diffie-Hellman parameters. The PSK identity and identity hint fields have the same meaning as in the previous algorithm (note that the *ServerKeyExchange* message is always sent, even if no PSK identity hint is provided) [19].

DHE_PSK are used from *OpenSSL version 1.1.0*.

3 Language Go

In this chapter, I will explain the main characteristics of language Go and show how to write the code. Next, I will describe Go libraries called packages and the tool Cgo.

3.1 Introduction

Go is a programming language developed by *Google* in the year 2007 and announced in November 2009. Many companies have started using Go because of its performance, simplicity, ease of use and powerful tooling [22]. Go programming language is a statically-typed language with advanced features and a clean syntax. It combines the performance and security benefits associated with using a compiled language like *C++* with the speed of a dynamic language like *Python* [23]. It provides:

- garbage collector - memory is cleaned up automatically when nothing refers to it anymore,
- fast compilation time - through effective work with individual parts of a program and simple grammar,
- light-weight processes (via go-routines), channels,
- a rich standard library,
- easy testing - incorporated directly into the core language,
- clear documentation full of examples.

Go excluded some features intentionally to keep language simple and concise. There is no support for type inheritance, method or operator overloading, circular dependencies among packages, pointer arithmetic, assertions nor for generic programming [22, 24].

3.2 How to Start

This section contains short instructions how to start with Go programming language, what install, what is needed to set and how to compile and run programs.

3.2.1 Installation

Golang [1], the Go official website, provides an installer of Go to download free binaries release suitable for Windows, Linux, Mac OS X and FreeBSD. On Windows system, there is downloaded MSI file which is needed to open and follow the prompts to install the Go tools. By default, the installer puts the Go distribution in $C:\backslash Go$.

On Linux system, there is easier to install Go via package management, for example:

```
sudo apt-get install golang
```

Upgrading from an older version of Go must be predecessed by removing an existing version.

3.2.2 Environment

After installation, it is necessary to set Go to a *$PATH* environment variable and set a *$GOROOT* environment variable to point to a directory in which it was installed. The Go toolset uses an environment variable called *$GOPATH* to find Go source code.

```
export GOROOT=/path/to/go1.X
export PATH=$PATH:$GOROOT/bin
export GOPATH=$HOME/Projects/go
```

Go typically keep all code in a single workspace. A workspace contains many version control repositories (managed by Git, for example). Each repository contains one or more packages and each package consists of one or more Go source files in a single directory. The path to a package's directory determines its import path [2]. A workspace is a directory hierarchy with three directories: *src* contains Go source files, *pkg* contains package objects, and *bin* contains executable commands [25].

---

1. https://golang.org/dl/
2. This differs from other programming environments in which every project has a separate workspace and workspaces are closely tied to version control repositories [25].

3.2.3 Compilation and Arguments

Go is a compiled language, and like many languages, it makes heavy use of the command line. The following command compiles and runs program:

```
go run <file-name>.go
```

The following command parameterizes execution of programs. First must be used `go build` command to create a binary program.

```
go build <file-name>.go
./<file-name> <arguments>
```

In code, *os.Args* provides access to raw command-line arguments. The first value is a path to the program, and *os.Args[1:]* holds the arguments to the program.

```
argsWithProg := os.Args
argsWithoutProg := os.Args[1:]
arg := os.Args[3]
```

Here is necessary to import package *os*.

3.3 Packages

In Go, source files are organized into system directories called packages. To develop software applications, writing maintainable and reusable pieces of code is very important. Go provides the modularity and code reusability through its package system. Go encourages programmers to write small pieces of software components through packages, and compose their applications with these small packages. The packages from a standard library are available at the *pkg* subdirectory of the $*GOROOT* directory. The Go developer community is very enthusiastic about developing third-party Go packages [26].

When programmers develop executable programs, they will use the package *main* for making the package as an executable program. The package *main* tells the Go compiler that the package should compile as an executable program instead of a shared library. When programmers build shared libraries, they will not have any *main* package and *main* function in the package [26].

To download third-party Go packages, the following command is used:

```
go get example/exampleLib
```

After installing *exampleLib*, the import statement is put in programs for reusing the code, as shown below:

```
package db
import (
  "example/exampleLib"
  "example/exampleLib/sub"
)
func init {
  // initialization code here
}
```

Third-party Go packages are installed by using `go install`. This command will build the package *"sub"* which will be available at the *pkg* subdirectory of $*GOPATH*.

```
package main
import (
  "fmt"
  "example/exampleLib/sub"
)
func main() {
    sub.Add("dr","Dart")
    fmt.Println(sub.Get("dr"))
    // code here
}
```

3.4 C? Go? Cgo!

Cgo allows Go to interoperate with C code. Go code contains `import "C"`, and it is immediately preceded by comment lines which may contain any C code, including function and variable declarations and definitions. It is necessary to say that there must be no blank lines in between the Cgo comment and the import statement [27].

3.4.1 Own C library in Go

Cgo allows including own C libraries. The C library is located in the same directory as the Go wrapper. For example:

```
mkdir -p $HOME/go/src/hello
```

The header file *foo.h* defines functions.

```
typedef struct message_struct {
  char message[255];
  int shown;
} Message;

Message * create_msg(char * msg);
void show_msg(Message * message);
void free_msg(Message * message);
```

The library implementation *foo.c* performs the work.

```
#include "hello.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
Message * create_msg(char * msg) {
  Message * message = (Message *) malloc(sizeof(Message));
  if (message != NULL) {
    strcpy(message->message, msg);
    message->shown = 0;
  }
  return message;
}
void show_msg(Message * message) {
  printf("%s\n", message->message);
  message->shown = 1;
}
void free_msg(Message * message) {
  free(message);
}
```

The C library is compiled and it is up to programmer if he creates shared library object or not. He should write a short test (*test.c*) of the library.

The next step is to create the Go native library wrapper *foo.go* in the same location as C library. It is necessary to set *CFLAGS* and *LDFLAGS* to *#cgo* because of an environment issue where library *hello* could not be found by the Go linker.

```
package hello
/*
#cgo CFLAGS: -I$HOME/go/src/hello
#cgo LDFLAGS: -L$HOME/go/src/hello -lhello
#include "hello.h"
*/
```

```
import "C"
type Message C.Message
func CreateMsg(msg string) *C.Message {
  cMsg := C.CString(msg)
  return C.create_msg(cMsg)
}
func ShowMsg(msg *C.Message) {
  C.show_msg(msg)
}
func FreeMsg(msg *C.Message) {
  C.free_msg(msg)
}
```

It is needed to install the package for Go wrapper for the library to make it available to other packages.

```
go install hello
```

By installing the package, everything is prepared for use in Go. Package is called in Go code through `import "hello"`.

```
package main
import "hello"
func main() {
  msg := hello.CreateMsg("Hello, world!")
  hello.ShowMsg(msg)
  hello.FreeMsg(msg)
}
```

In order to use Cgo, it is firstly needed to install a gcc compiler and have *gcc.exe* in $*PATH* environment variable before compiling with Cgo will work [28, 29].

3.5 Testing

Writing tests for code is a good way to ensure quality and improve reliability. Go includes a special program that makes writing tests easier. The file included test has to have the same name as tested file with suffix "_test.go". For file *hello.go* the test file is *hello_test.go* [30, 31].

```
package main
import "testing"
func TestSomething(t *testing.T) {
  t.Error("Error message")
}
```

The file has to be in the same package and include `import`. The test function starts with "Test" and receives a single parameter `*testing.T`. A message why to test fail is shown by `t.Error()`.

Test runs using *go test* in that directory. The output is:

```
=== RUN TestSomething
--- FAIL: TestSomething (0.00s)
    hello_test.go:4: Error message
FAIL
exit status 1
FAIL    path/to/hello 0.004s
```

Tests aslo run using following commands:

```
//test all file in the directory
go test *.go
//test all file in the directory with run information
go test -v *.go
//run simple test
go test -run <test-name>
```

4 Go Packages Analysis

Before implementation of an application, I need to find suitable Go package which can connect to a server using TLS_PSK, work with cryptography and X.509 certificates. The package has to be established, tested and has user's community which uses its. The advantages are having issue tracker system and well-processed documentation.

Before analyzing libraries, I had to learn Go language, install it and try to programming. All important information about this language and working with it is described in the chapter 3.

I examine features of language Go related to the security and cryptography, analyze libraries which Go contains and also search for available third-party libraries. This chapter contains an overview, evaluation and comparison existing cryptographic libraries.

4.1 First step of analysis

The Internet provides a big amount of Go packages created by third parties. To the thesis, I selected from more than one hundred packages working with cryptographic functions and hash operations. It was necessary to remove packages from untrusted and unsupported sources.

Table 4.1 shows Go package *Crypto* created by *Golang* and other Go packages created by third-parties. Individually items mean:

- Name - founder nickname (almost all packages are searched on *https://github.com/* where is not published real name of founder or contributors) and name of package.
- Cryptographic functions - support symmetric encryption, asymmetric encryption and hash functions (all/sym/asym/hash/unknown).
- Start of project - a year of creating project with package.
- Is still opened - an important measure is if contributors still work on a project (Yes/No).
- Issue tracker system - support of solving bugs, feature requests, tracking todos, and more (Yes/No).
- Documentation quality - assessed on a scale of 0-5 points (5 is the highest) according to own experience to work with a pack-

age. I evaluate description of package and functions, work instructions, etc.

- Downloads - a number of downloads. In case a package is saved on *https://github.com/*, an information about downloads presents a value of Fork. Downloads of package *Crypto* created by *Golang* is unknown; this package is automatically installed with installing program Go.
- License type - type of licenses:
  - BSD - BSD 3-clause "New" or "Revised" License,
  - Apache - Apache License 2.0,
  - MIT - MIT License,
  - Mozilla - Mozilla Public License 2.0,
  - GNU - GNU Lesser General Public License v3.0,

  some packages do not provide this information ("none").

Table 4.1: Table of cryptographic libraries in Go

Name	Crypto functions	Start of project	Is still opened	Issue tracker system	Doc. quality	Downloads	License type
golang/crypto	all	2009	Y	Y	5	-	BSD
ory-am/fosite	asym, hash	2015	Y	Y	4	39	Apache
square/go-jose	asym, sym	2014	Y	Y	4	70	Apache
Shopify/ejson	hash	2014	Y	Y	3	22	MIT
mitchellh/go-mruby	asym	2014	Y	Y	4	19	MIT
Sermo Digital/jose	all	2015	Y	Y	4	28	MIT

Continued on next page

Table 4.1 – continued from previous page

Name	Crypto functions	Start of project	Is still opened	Issue tracker system	Doc. quality	Downloads	License type
mozilla/tls-observatory	hash	2014	Y	Y	4	31	Mozilla
square/certigo	hash	2016	Y	Y	3	12	Apache
docker/libtrust	all	2014	N	Y	4	26	Apache
dedis/crypto	sym, hash	2010	Y	Y	4	17	GNU
dchest/blake2b	hash	2012	N	Y	4	6	none
enceve/crypto	hash	2016	N	Y	4	1	GNU
go-libp2p-crypto	all	2015	Y	Y	4	5	MIT
dchest/blake2s	hash	2012	N	N	3	3	none
benburkert/openpgp	sym, hash	2012	N	N	3	0	none
minio/sha256-simd	hash	2016	Y	Y	3	14	Apache
avelino/awesome-go	unknown	2014	Y	Y	2	2287	MIT
ethereum/go-ethereum	all	2013	Y	Y	4	1028	GNU

Continued on next page

Table 4.1 – continued from previous page

Name	Crypto functions	Start of project	Is still opened	Issue tracker system	Doc. quality	Downloads	License type
chain/chain	hash	2015	Y	Y	4	113	GNU
lightning network/lnd	sym, hash	2015	Y	Y	4	50	MIT
square/go-jose	asym, hash	2014	Y	Y	4	70	Apache
dgrijalva/jwt-go	asym, hash	2012	Y	Y	4	226	MIT
golang/oauth2	asym, hash	2014	Y	Y	4	279	BSD

The table was created based on an information available on the following websites [33, 34, 35, 36]. The table does not include packages which do not support any cryptographic functions. Also, the table does not contain information from untrusted sources which are not up to date or supported.

4.2 Second step of analysis

The next step of analysis was to select packages to the other analysis. I selected four libraries: *Crypto*, *jose*, *go-lib-p2p-crypto* and *go-ethereum* because of their support of symmetric and asymmetric functions, hash operands, because the contributors still work on them and they have support of solving bugs and feature requests. I thought about taking number of downloads as a relevant parameter but the packages can be available on several sources.

The other libraries, I do not include to the second analysis because they does not fulfill all the relevant parameters appointed in previous paragraph together.

In a table 4.2, there are shown those four libraries which are compared to the base of the other measures:

- Generation crypto. key - possibilities for generation of cryptographic keys (Yes/No),
- Key size min. - if package could generate cryptographic key, it shows minimal key size in bytes (some packages do not provide this information or can not generate keys),
- Key size max. - if package could generate cryptographic key, it shows maximal key size in bytes (some packages do not provide this information or can not generate keys),
- Self-signed certificate - generation of self-signed certificates (Yes/No/unknown),
- CSR - support for certificate signing requests (Yes/No),
- Formats - support for different formats and extensions (Yes/No/unknown),
- SSL/TLS - support of TLS protocol (Yes/No),
- TLS_PSK - support communication with a server by using pre-shared key (Yes/No).

Table 4.2: Filtered table of packages with expansion measures

Package	crypto [37]	jose [38]	go-libp2p-crypto [39]	go-ethereum [40]
Founder	Golang	SermoDigital	libp2p	ethereum
Crypto. functions	all	all	all	all
Start of project	2009	2015	2015	2013
Is still opened	Y	Y	Y	Y
Issue tracker system	Y	Y	Y	Y

Continued on next page

Table 4.2 – continued from previous page

Package	crypto	jose	go-libp2p-crypto	go-ethereum
Doc. accessability	5	4	4	4
Downloads	-	28	5	1028
License	BSD	MIT	MIT	GNU
Generation crypto. keys	Y	N	Y	Y
Key size min	2	unknown	unknown	unknown
Key size max	8192	unknown	unknown	unknown
Self-signed certificate	Y	Y	Y	Y
CSR	Y	N	unknown	N
Formats	Y	unknown	N	unknown
SSL/TLS	Y	N	N	N
TLS_PSK	N	N	N	N

The analysis shows that until today, a package that is suitable with all the requirements does not exist. The generation of cryptographic keys (within 2 to 8192 bytes, as suggests table 1 in [41]) and support of TLS protocol at the same time is only in package *Crypto* by *Golang*. This is the main reason why i chose it to use it in the implementation. *Crypto* by *Golang*(from version 1.5) also works with X.509 certificates what means generation of self-signed certificates, support for certificate signing requests, different formats and extensions.

During the analysis I found package *Raff/tls-psk* using TLS_PSK cipher suites, I describe it in subsection 5.2.1.

5 Implementation

The practical part of my bachelor thesis is to implement console application to connect to a server using TLS_PSK, create keys and certificates which are signed by a certificate authority. In this chapter, I will explain an assignment, the complications and a process of implementation.

5.1 Assignment

For functionality, there are needed several parameters, such as:

- address of a server with certificate authority,
- port,
- pre-shared key to connect to server,
- path to the key store,
- IP address,
- domain name and
- identity of a system for what the certificate is created.

The application connects to a server with a certificate authority through TLS_PSK, then it generates an asymmetric RSA key pair and a certification signing request which is sent to an authority. Server sends signed certificate back and it is along with private key saved to a designated key store.

5.2 TLS_PSK

For application I decided to use packages created by *Golang*. There was a problem because this package does not support TLS with pre-shared key. During the previous analysis, explained in the chapter 4, I knew that there is one third-party library which is adapted to use TLS_PSK cipher suites [42].

5.2.1 Raff

Raff/tls-psk library was created 3 years ago and it has not been maintained yet. It does not have any downloads, issue tracker nor active contributors.

For application needed I decided to try this library. Connecting to *OpenSSL version 1.1.0e* server was correct but connecting to server with CA was not possible because of unsupported cipher suites. *Raff/tls-psk* library supports only 4 versions of cipher suites:

```
TLS_PSK_WITH_RC4_128_SHA,
TLS_PSK_WITH_3DES_EDE_CBC_SHA,
TLS_PSK_WITH_AES_128_CBC_SHA,
TLS_PSK_WITH_AES_256_CBC_SHA
```

To solving the issue, one of the supported cipher suites was added to the test server. It turned out that package *Raff/tls-psk* has also some other issues whose I have not examined yet.

The other solution how to implement TLS_PSK to Go is using Cgo. Cgo allows Go to interoperate code written in C language. How Cgo works I describe in section 3.4.

5.3 Language C in Go

I created methods using *OpenSSL* in language C for initialization, connection, reading from and writing to the server. *OpenSSL* contains an open-source implementation of the SSL/TLS protocols. The libraries, written in the C programming language, implements basic cryptographic functions and provides various utility functions [43].

5.3.1 Initialization

Initialization consists of four methods from libraries `openssl/ssl.h` and `openssl/err.h`. Methods load the strings used for error messages, and set up the algorithms needed for TLS.

```
void init() {
  SSL_load_error_strings();
  ERR_load_BIO_strings();
  OpenSSL_add_all_algorithms();
  SSL_library_init();
}
```

5.3.2 Connection

Connection to the server is realized using TLS_PSK cipher suite. Connection method creates a new `SSL_CTX` context. This is created using the `TLSv1_client_method` which despite its name actually creates a client that will negotiate the highest version of TLS supported by the server it is connecting to.

Next it creates a new `BIO` chain consisting of an SSL `BIO` (using `SSL_CTX`) via `BIO_new_ssl_connect`. The connection object inherits from the context object, and can override the settings on the context. The connection object is tuned with `BIO_set_conn_hostname`. The address consists of host name and port. `BIO_get_ssl` is used to fetch the SSL connection object.

The pre-shared key is set through `SSL_set_psk_client_callback` with SSL and a callback function as arguments. The purpose of the callback function is to select the PSK identity and the pre-shared key to use during the connection setup phase. The variable with the pre-shared key has to be global.

```
BIO * conn(char* add) {
  SSL_CTX *ctx;
  BIO *bio;
  SSL *ssl;
  ctx = SSL_CTX_new(TLSv1_client_method());
  bio = BIO_new_ssl_connect(ctx);
  BIO_get_ssl(bio, &ssl);
  if (!ssl) {
    printf("Can't locate SSL pointer\n");
    return NULL;
  }
  BIO_set_conn_hostname(bio, add);
  BIO_get_ssl(bio, ssl);
  SSL_set_mode(ssl, SSL_MODE_AUTO_RETRY);
  SSL_set_psk_client_callback(ssl, psk_client_cb);
  int e = BIO_do_connect(bio);
  printf("BIO_do_connect(bio): %d\n", e);
  if (e == -1) return NULL;
  return bio;
}
```

It is necessary to note that a client's pre-shared key is hexadecimal number without leading 0x unlike a server's pre-shared key, it is ASCII. Interesting is that this information is not included in any

documentation. *OpenSSL* manual to a client[1] and to a server[2] mention only a hexadecimal number without leading 0x.

5.3.3 Reading from and writing to server

I solved question how to read from a server requisite amount of bytes but server ends every message by null terminator. So, reading from the server I realized via `BIO_read` method. This method reads 1 byte from `BIO` and saves value to the temp variable in the loop which ends when temp variable contains the null terminator $'\backslash 0'$.

Writing to the server I realized via created method `BIO_write`. I had to add the null terminator $'\backslash 0'$ to the end of message (string) because of conversions between C string and Go string.

C does not have an explicit string type and strings are represented by a null-terminated array of chars. In Go, a string is a read-only slice of bytes without $'\backslash 0'$. Conversion between Go and C strings is done with the `C.CString`, `C.GoString`, and `C.GoStringN` functions. These conversions make a copy of the string data [28].

```
1 // Go string to C string
2 func C.CString(string) *C.char
3
4 // C string to Go string
5 func C.GoString(*C.char) string
6
7 // C data with explicit length to Go string
8 func C.GoStringN(*C.char, C.int) string
```

5.4 Communication protocol

Communication between a client and a server containing a certificate authority (below only server) is based on null-terminated strings. Methods written in C language have to be called in Go code with prefix `'C.'` and have the correct types of arguments. The protocol looks as follows:

1. https://wiki.openssl.org/index.php/Manual:S_client(1)
2. https://wiki.openssl.org/index.php/Manual:S_server(1)

5.4.1 Connection using TLS_PSK

The application connects to the server using given pairing key and default identity.

As I wrote in the subsection 5.3.2, the pre-shared key has to be the global variable in C code. I used the following commands to set key from Go code where application takes the pre-shared key as an argument. Pre-shared key has to be a string. Next, I call C method `conn` with preffix `'C.'` to create `BIO` and establish connection.

```
//in C
char *psk_key;

//in Go
arg := os.Args
private_key := arg[3]

C.psk_key = C.CString(private_key)

bio *C.BIO = C.conn(C.CString(server + ":" + port))
```

The server validates the application. Client and server exchanged *ClientHello - ServerHello*, and *ClientKeyExchange - ServerKeyExchange*. They agreed to use common cipher suite.

5.4.2 Protocol version exchange

The application sends supported protocol version. Currently is used only protocol version 1. It is assumed that the application will be developed and version will be changed. The protocol is sent as string to server via C writing method.

```
protocol_version := C.CString("1")
err_write := C.C_bio_write(bio, protocol_version, 1)
server_protocol_version := C.C_bio_read(bio)
if *server_protocol_version != *protocol_version {
  return -1
}
```

The server chooses the highest protocol version possible and sends it back. The application reads that version and compare to version which was sent in the beginning (currently version 1).

5.4.3 Key length

The application sends the system identification and certificate type whose received as arguments. The server returns the information with key length for the key generation. I used standard library function `strconv.Atoi` to convert string to the number from package `"strconv"` and function `rsa.GenerateKey` to generate key from package `"crypto/rsa"`.

```
key_len, err := strconv.Atoi(C.GoString(key_length))
checkErr(err)

var key interface{}
key, err = rsa.GenerateKey(rand.Reader, key_len)
```

I had to create the generated key as `interface` because of using in temp function for creating PEM format of key. This function is not included in standard Go library and looks following:

```
func pemBlockForKey(priv interface{}) *pem.Block {
  switch k := priv.(type) {
  case *rsa.PrivateKey:
    return &pem.Block{Type: "RSA PRIVATE KEY", Bytes: x509.
    MarshalPKCS1PrivateKey(k)}
  case *ecdsa.PrivateKey:
    b, err := x509.MarshalECPrivateKey(k)
    if err != nil {
      fmt.Println(os.Stderr, "Unable to marshal ECDSA
    private key: %v", err)
      os.Exit(2)
    }
    return &pem.Block{Type: "EC PRIVATE KEY", Bytes: b}
  default:
    return nil
  }
}
```

5.4.4 Certificate signing

The application creates structure `tmp` of certificate request based on observed IP address (`ip`) and `domain_name` from arguments.

```
tmp := &x509.CertificateRequest {
  Subject      : pkix.Name{CommonName : ip},
  DNSNames     : []string{domain_name},
```

```
    IPAddresses : []net.IP{net.ParseIP(ip)},
}
```

Following, I used the function `x509.CreateCertificateRequest` from package `crypto/x509`. The function takes structure `tmp` and generated key (from previous subsection) to create certificate request. The function returns bytes array, so to sending the request to the server, it has to be converted to PEM string.

PEM is created using standard library function `pem.Encode` from package `encoding/pem`. It takes arguments file (`csr_out`)and PEM encoded structure with type (in this case `"CERTIFICATE REQUEST"`) and the decoded bytes of certificate request (`csr`) - typically a DER encoded ASN.1 structure.

```
csr,err:=x509.CreateCertificateRequest(rand.Reader,tmp,key)

csr_out, err := os.Create("csr.pem")
pem.Encode(csr_out, &pem.Block{Type: "CERTIFICATE REQUEST",
    Bytes: csr})
```

The application sends generated PEM encoded certificate signing request (with domain name in Common Name field) and additionally IP address to the server. When the file with request is not needed, it is deleted through:

```
os.Remove("csr.pem")
```

The server responses with the whole chain signed certificates (PEM encoded). Signed certificate is saved to file and together with generated key they are saved to the key store.

5.4.5 Achieving trust

The connection ends when the server sends trust anchor as a CA certificate. After that, it is not possible to write to or read from the server.

6 Installation, compilation and testing

This chapter contains information about installing programs whose are necessary to install to correct application run. The chapter also contains example of build and run commands. Next, I will describe application unit tests and test tools.

6.1 Necessity before compilation

The application makes use of x.509 certificates. The Go from version 1.5 has full support for Cgo and the package `crypto/x509`, as well as a number of other fixes and improvements. How to install it (or upgrade) I describe in section 3.2.

When the Go tool sees that Go file uses the special `import "C"`, it needs to be invoked to generate the C to Go and Go to C thunks and stubs. C compiler has to be invoked for every C file in the package. The individual compilation units are combined together into a single file. The resulting file take a trip through the system linker for fix-ups against shared objects they reference [44].

The C code uses *OpenSSL* libraries, I tried to compile application on different versions. The oldest version was OpenSSL version 1.0.1e.

Before correct compilation, I encountered following problem:

```
# command-line-arguments
/tmp/go-build029926020/command-line-arguments/_obj/app.cgo2.
    o: In function 'psk_client_cb':
./app.go:37: undefined reference to 'BIO_snprintf'
/tmp/go-build029926020/command-line-arguments/_obj/app.cgo2.
    o: In function 'conn':
./app.go:59: undefined reference to 'TLSv1_client_method'
./app.go:59: undefined reference to 'SSL_CTX_new'
./app.go:61: undefined reference to 'BIO_new_ssl_connect'
./app.go:62: undefined reference to 'BIO_ctrl'
...
./app.go:123: undefined reference to 'BIO_test_flags'
/tmp/go-build029926020/command-line-arguments/_obj/app.cgo2.
    o: In function 'init':
./app.go:86: undefined reference to 'SSL_library_init'
collect2: ld returned 1 exit status
```

It means that compiler can not find *libcrypto.a* and *libssl.a*. This dynamic libraries can be found in package *libssl-dev* (or *openssl-devel* on Fedora), for example. This package is part of the *OpenSSL* project's implementation of the SSL and TLS cryptographic protocols for secure communication over the Internet. It contains development libraries, header files, and manpages for *libssl*[1] and *libcrypto*[2] [45, 46].

LDFLAGS and CFLAGS are defined with pseudo #*cgo* directives within comments to tweak the behavior of the C compiler.

```
#cgo CFLAGS: -Iaddress/to/openssl/include
#cgo LDFLAGS: -Laddress/to/openssl -lcrypto -lssl
```

Application is built and run through:

```
go build <name>.go
./<name> <server> <port> <PSK> <pathToKeyStore> <subjectIP>
    <subjectDomainName> <subjectID>
```

For example:

```
go build app.go
./app localhost 10443 aaaa /tmp 192.0.2.0 example.com END
```

6.2 Testing

The necessary step of implementation is testing multiple parts of application and all of the system acting together. I test whether each component fulfills its contract, but also whether they are composed and configured correctly and interact as expected.

6.2.1 Connectivity test

Connectivity test checks behavior of application for the wrong server, port or key. For the wrong server or port, the application returns *BIO_do_connect:* -1 what means the connection failed. For the wrong key, the application writes checking identity and key, *BIO_do_connect:* -1 what means that the connection failed for the wrong key.

1. *libssl* provides the client and server-side implementations for SSLv3 and TLS.
2. *libcrypto* provides general cryptographic and X.509 support needed by SSL/TLS.

6.2.2 Step by step testing of communication

I created tests for checking communication in certain moments. Each test has to contain connection and the previous steps of communication to correct test of given step.

For example test of protocol version exchange contains the correct connection to the server.

```
func Test_version_exchange_OK(t *testing.T) {
  t.Log("Testing protocol version exchange... ")

  bio := create_connection("localhost", "10443", "aaaa")
  if bio == nil {
    t.Errorf("Error while connecting to server")
  }

  if ret := protocol_version_exchange(bio); ret == -1 {
    t.Errorf("Error: different version")
  } else if ret == -2 {
    t.Errorf("Error: problem with connection")
  }
  t.Log("Test OK")
}
```

Tests of generation certificate request and private key also contain generation with the correct and wrong connection, arguments and key size.

6.2.3 File test

These tests check saving certificates and keys without connection and previous communication with the server to the key store with the existing or non-existing path, correct key or empty interface.

6.2.4 Unit Test Evaluation

I wrote 18 unit tests for application:

```
$ go test -v *.go
--- PASS: Test_connection_OK (0.02s)
--- PASS: Test_connection_NOK_port (0.00s)
--- PASS: Test_connection_NOK_key (0.00s)
--- PASS: Test_version_exchange_OK (0.00s)
--- PASS: Test_version_exchange_NOK_server (0.00s)
```

```
--- PASS: Test_generate_key_OK_CA (3.67s)
--- PASS: Test_generate_key_OK_END (0.09s)
--- PASS: Test_generate_key_NOK_server (0.00s)
--- PASS: Test_generate_key_NOK_type (0.01s)
--- PASS: Test_create_csr_OK (0.16s)
--- PASS: Test_create_csr_NOK (0.09s)
--- PASS: Test_send_csr_OK (0.24s)
--- PASS: Test_save_cert_OK (0.00s)
--- PASS: Test_save_cert_NOK_path (0.00s)
--- PASS: Test_save_key_OK (0.04s)
--- PASS: Test_save_key_NOK_key (0.00s)
--- PASS: Test_save_key_NOK_path (0.08s)
PASS
ok      command-line-arguments  4.422s
```

The all tests pass with average time 3.682 second from 10 tests where minimal time was 2.374 second and maximal time was 5.114 second.

The other testing makes testing tools.

Kolko pamate, CPU, zdrojov?

6.3 Valgrind

Valgrind is an instrumentation framework for building dynamic analysis tools. Valgrind tools can automatically detect many memory management and threading bugs, and profile programs in detail [47].

I checked application via Valgrind with following result:

```
HEAP SUMMARY:
    in use at exit: 1,864 bytes in 5 blocks
  total heap usage: 8 allocs, 3 frees, 1,968 bytes allocated

LEAK SUMMARY:
    definitely lost: 0 bytes in 0 blocks
    indirectly lost: 0 bytes in 0 blocks
      possibly lost: 1,824 bytes in 3 blocks
    still reachable: 40 bytes in 2 blocks
         suppressed: 0 bytes in 0 blocks
Rerun with --leak-check=full to see details of leaked memory

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from
    0)
```

The result indicates that there is possibly lost 1,824 bytes in 3 blocks and still reachable 40 bytes in 2 blocks. When I tried to run it again, the number of blocks was different what is unusual behavior. There was possibly lost 1,824 bytes in 3 blocks and still reachable 16 bytes in 1 blocks.

I checked all allocations in C code - if I deallocated all memory and I checked all allocations in Go. Go has garbage collector, but C variables created in Go code has to be deallocated manually. Following example shows allocations in Cgo:

```
1 //memory allocation in C:
2 char * buf = (char*) malloc(size * sizeof(char));
3 //memory deallocation in C:
4 free(buf);
5
6 //memory allocation C variable in Go:
7 var bio * C.BIO = <some-function-which-returns-bio>
8 //or: bio := <some-function-which-returns-bio>
9 //memory deallocation C variable in Go:
10 C.free(unsafe.Pointer(bio))
```

When I was sure that all of the memory was correct deallocated, I run Valgrind. The result showed some memory leaks again. I tried to run Valgrind on "Hello" code - code containing only print with "Hello World!". There were no memory leaks. I assumed that memory leaks are caused by C code. I added the whole C code from the application to "Hello" code; Valgrind showed memory leaks as I supposed to. I wanted to detect which function deallocated memory incorrectly. The Valgrind showed memory leaks in empty C code with different numbers of blocks again.

Research this issue showed that Go programs are not compatible with Valgrind. Issue is still opened in *Golang* issue tracker[3].

6.4 Other analysis tools

I tried to find the other tool for application analysis. I found the post [44] which says that Go has great tools - the race detector, pprof for profiling code, coverage, fuzz testing, and source code analysis tools. But none of those work across the Cgo.

3. https://github.com/golang/go/issues/782

7 Conclusion

This bachelor thesis deals with cryptographic tools of Go programming language. The aim was to evaluate Go cryptographic libraries and implement the prototype of a client application able to generate asymmetric keys and certificate signing requests and connect to a server application using TLS_PSK cipher suite. The thesis was consulted and realized in cooperation with Y Soft Corporation, a.s.

The Go libraries were overviewed and compared based on their support of symmetric and asymmetric cryptographic algorithms and hash operations. In pursuance of analysis, the four libraries were selected and subsequently examined according to methods for generation of cryptographic keys and self-signed certificates, support for certificate signing requests, different formats, and SSL/TLS. From the report, one library was selected for the implementation.

The prototype of a client application is implemented in language Go which interoperates with methods in language C. The source code contains the both languages written according to rules of Cgo. C methods serve the connection to the server and the communication between the client and the server using OpenSSL. Go methods serve subjects of communication concretely. During implementation, I solved problems with non-supported library and OpenSSL integration.

The Bachelor thesis had a great personal benefit for me. I expanded my knowledge of Go programming language, OpenSSL, and the certificates. I also gain a great asset in designing and developing the application and I improved my programming skills in the languages C and Go.

Bibliography

1. GXS, Inc. *What is EDI (Electronic Data Interchange)?* 2017. Available also from: `http://www.edibasics.com/what-is-edi/`. [Online; 20-05-2017].
2. SHARMA, Chetan; NAKAMURA, Yasuhisa. *Wireless data services: Technologies, business models and global markets*. Cambridge University Press, 2003.
3. DOSTÁLEK, Libor; VOHNOUTOVÁ, Marta. *Velký průvodce infrastrukturou PKI*. Computer Press, Albatros Media as, 2016.
4. SCHMEH, Klaus. *Cryptography and public key infrastructure on the Internet*. John Wiley & Sons, 2006.
5. HOUSLEY, Russell; POLK, William; FORD, Warwick; SOLO, David. *Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile*. 2002. Technical report.
6. RED HAT, Inc. *B.3. Standard X.509 v3 Certificate Extension Reference*. 2017. Available also from: `https://access.redhat.com/documentation/en-US/Red_Hat_Certificate_System/8.0/html/Admin_Guide/Standard_X.509_v3_Certificate_Extensions.html`. [Online; 20-05-2017].
7. OŠŤÁDAL, Radim. *Práce aplikací s certifikáty veřejných klíčů*. 2010. Available also from: `https://is.muni.cz/auth/th/255508/fi_b/Prace_aplikaci_s_certifikaty_verejnych_klicu-el.pdf`. Bachelor's thesis. Faculty of Informatics Masaryk University. [Online; 20-05-2017].
8. SINGH, Simon; KOUBSKÝ, Petr; ECKHARDTOVÁ, Dita. *Kniha kódů a šifer: tajná komunikace od starého Egypta po kvantovou kryptografii*. Dokořán, 2003.
9. HOW TO SSL. *PEM Files*. 2017. Available also from: `http://how2ssl.com/articles/working_with_pem_files/`. [Online; 21-02-2017].
10. JR., Burton S. Kaliski. *A Layman's Guide to a Subset of ASN.1, BER, and DER*. 1993. Available also from: `http://luca.ntop.org/Teaching/Appunti/asn1.html`. [Online; 21-05-2017].

11. BAKKER, Paul. *ASN.1 key structures in DER and PEM*. 2014. Available also from: https://tls.mbed.org/kb/cryptography/asn1-key-structures-in-der-and-pem. [Online; 21-02-2017].

12. OPPLIGER, Rolf. *Security technologies for the world wide web*. Artech House, 2003.

13. FREIER, Alan; KARLTON, Philip; KOCHER, Paul. *The secure sockets layer (SSL) protocol version 3.0*. 2011. Technical report.

14. WOLFSSL. *Differences between SSL and TLS Protocol Versions*. 2010. Available also from: https://www.wolfssl.com/wolfSSL/Blog/Entries/2010/10/7_Differences_between_SSL_and_TLS_Protocol_Versions.html. [Online; 21-05-2017].

15. RESCORLA, E. *The Transport Layer Security (TLS) Protocol Version 1.3 draft-ietf-tls-tls13-20*. 2017. Technical report.

16. CASWELL, Matt. *Using TLS1.3 With OpenSSL*. 2017. Available also from: https://www.openssl.org/blog/blog/2017/05/04/tlsv1.3/. [Online; 05-05-2017].

17. CORPORATION, IBM. *IBM Knowledge Center*. 2014. Available also from: https://www.ibm.com/support/knowledgecenter/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10660_.htm. [Online; 21-05-2017].

18. MICROSOFT. *TLS Handshake Protocol*. 2017. Available also from: https://msdn.microsoft.com/en-us/library/windows/desktop/aa380513(v=vs.85).aspx. [Online; 08-05-2017].

19. ERONEN, Pasi; TSCHOFENIG, Hannes. *Pre-shared key ciphersuites for transport layer security (TLS)*. 2005. Technical report.

20. TAUBERT, Tim. *More Privacy, Less Latency*. 2015. Available also from: https://timtaubert.de/images/tls13-hs-resumption.png. [Online; 10-05-2017].

21. HELME, Scott. *Perfect Forward Secrecy - An Introduction*. 2015. Available also from: https://scotthelme.co.uk/perfect-forward-secrecy/. [Online; 24-05-2017].

22. DOXSEY, Caleb. *Introducing Go*. O'Reilly Media, 2016.

23. HARRIS, Alan. *Go: Up and Running*. O'Reilly Media, 2015.

24. HARA, Mikio. *golang/go*. 2017. Available also from: https://github.com/golang/go/wiki. [Online; 27-02-2017].

25. GOLANG. *How to Write Go Code*. 2017. Available also from: https://golang.org/doc/code.html. [Online; 21-05-2017].

26. VARGHESE, Shiju. *Understanding Golang Packages*. 2014. Available also from: https://thenewstack.io/understanding-golang-packages/. [Online; 27-02-2017].

27. GOLANG. *Command cgo*. 2017. Available also from: https://golang.org/cmd/cgo/. [Online; 24-03-2017].

28. GERRAND, Andrew. *The Go Blog*. 2011. Available also from: https://blog.golang.org/c-go-cgo. [Online; 25-03-2017].

29. DOGAN, Jaana Burcu. *golang/go*. 2016. Available also from: https://github.com/golang/go/wiki/cgo. [Online; 25-03-2017].

30. CHISNALL, David. *The Go programming language phrasebook*. Addison-Wesley, 2012.

31. SMARTYSTREETS. *Testing in Go by example: Part 1*. 2015. Available also from: https://smartystreets.com/blog/2015/02/go-testing-part-1-vanillla. [Online; 22-05-2017].

32. KOZYRA, Nathan. *Mastering concurrency in go*. Packt Publishing, 2014.

33. GOLANGLIBS. *Security*. 2017. Available also from: https://golanglibs.com/top?q=security. [Online; 27-02-2017].

34. GOLANGLIBS. *Cryptography - Go libraries and apps*. 2017. Available also from: https://golanglibs.com/category/cryptography?sort=top. [Online; 27-02-2017].

35. GO DOC. *crypto - GoDoc*. 2017. Available also from: https://godoc.org/golang.org/x/crypto. [Online; 27-02-2017].

36. GOLANG. *Packages*. 2017. Available also from: https://golang.org/pkg/. [Online; 27-02-2017].

37. GOLANG. *Package crypto*. 2012. Available also from: https://golang.org/pkg/crypto/. [Online; 18-11-2016].

38. SERMODIGITAL. *SermoDigital/jose*. 2015. Available also from: https://github.com/SermoDigital/jose. [Online; 18-11-2016].

39. LIBP2P. *libp2p/go-libp2p-crypto*. 2015. Available also from: https://github.com/libp2p/go-libp2p-crypto. [Online; 18-11-2016].

40. ETHEREUM. *ethereum/go-ethereum*. 2013. Available also from: `https://github.com/ethereum/go-ethereum`. [Online; 18-11-2016].

41. HINEK, Jason. On the security of multi-prime RSA. *Journal of Mathematical Cryptology*. 2008, vol. 2, no. 2, pp. 117–147.

42. RAFF. *raff/tls-psk*. 2014. Available also from: `https://github.com/raff/tls-psk`. [Online; 27-02-2017].

43. FOUNDATION, Inc. OpenSSL. *Manpages for 1.0.2*. 2017. Available also from: `https://www.openssl.org/docs/man1.0.2/`. [Online; 02-05-2017].

44. CHENEY, Dave. *Cgo is not Go*. 2016. Available also from: `https://dave.cheney.net/2016/01/18/cgo-is-not-go`. [Online; 24-05-2017].

45. OPENSSL. *openssl/openssl*. 2016. Available also from: `https://github.com/openssl/openssl`. [Online; 02-05-2017].

46. DIE.NET. *Section 3: library functions - Linux man pages*. 2017. Available also from: `https://linux.die.net/man/3/`. [Online; 02-05-2017].

47. DEVELOPERS, Valgrind. *Valgrind Home*. 2016. Available also from: `http://valgrind.org/`. [Online; 24-05-2017].