FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Custom OpenSSL provider based on CNG
Student:	Ladislav Marko
Supervisor:	Ing. Josef Kokeš, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2023/2024

Instructions

1) Study the OpenSSL cryptographic library. Describe its evolution, features, structure.
2) Focus on the concept of providers available in OpenSSL since version 3: Explain their purpose, limitations, how they interact with the rest of the library. Explore the built-in as well as third-party providers available in the wild.
3) Design the structure of a new provider that would eventually be able to offload certificate work in the SSL/TLS protocols to an alternative cryptographic implementation.
4) Create a proof-of-concept of such a provider based on Microsoft's Crypto New Generation API.
5) Discuss your results.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 13 February 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Custom OpenSSL provider based on CNG

Ladislav Marko

Department of Computer Systems
Supervisor: Ing. Josef Kokeš, Ph.D.

May 10, 2023

Acknowledgements

I would like to thank Ing. Josef Kokeš, Ph.D., for his guidance and light-speed responses. Huge thanks also go to my parents and my significant other for their endless support. I would also like to show gratitude to everyone that has helped me with this project from spell checking to code reviews.

Declaration

In Prague on May 10, 2023 . . . .

Citation of this thesis

Marko, Ladislav. *Custom OpenSSL provider based on CNG*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Tato práce rozebírá OpenSSL poskytovatele (providers) a jak je implementovat. Práce prochází procesem implementace poskytovatele, který přenechává kryptografické operace ohledně certifikátů jiným implementacím algoritmů než těm z OpenSSL. Konkrétní zvolenou implementací jsou algoritmy Windows Cryptography API: Next Generation. Výsledný poskytovatel umožňuje TLS 1.3 spojení s klientským certifikátem načteným ze systémového úložiště certifikátů operačního systému Windows.

Klíčová slova OpenSSL, provider, CNG

Abstract

This thesis takes a closer look at OpenSSL providers and how to implement them. The thesis goes through the process of implementing a provider that offloads certificate operations to other algorithm implementations then OpenSSL ones. The selected implementation of algorithms is the Windows Cryptography API: Next Generation. The final provider allows for TLS 1.3 connection using client certificate loaded from the system certificate store of operating system Windows.

Keywords OpenSSL, provider, CNG

Contents

Introduction **1**

1 OpenSSL **3**

1.1 Features . . . 3
1.2 Evolution . . . 4
1.2.1 LibreSSL . . . 4
1.2.2 QuicTLS . . . 5
1.2.3 BoringSSL . . . 5
1.2.4 Current state . . . 6
1.3 Structure . . . 6
1.3.1 libcrypto library . . . 6
1.3.2 libssl library . . . 8
1.3.3 openssl . . . 9
1.3.4 Other command line tools . . . 9

2 OpenSSL providers **11**

2.1 Purpose . . . 11
2.2 Limitations . . . 12
2.3 Interactions . . . 12
2.4 Provider infrastructure . . . 13
2.4.1 Special data types . . . 14
2.4.2 Operations . . . 16
2.4.3 Key management . . . 16
2.4.4 Store management . . . 17
2.4.5 Digests . . . 17
2.4.6 Symmetric ciphers . . . 18
2.4.7 Message authentication codes . . . 18
2.4.8 Key derivation . . . 18
2.4.9 Key exchange . . . 19

2.4.10 Asymmetric ciphers 19
2.4.11 Asymmetric key encapsulation 19
2.4.12 Encoding and Decoding 19
2.4.13 Random number generation 20
2.4.14 Signatures 20
2.4.15 Algorithm selection 21
2.5 Built-in providers 22
2.5.1 Base provider 22
2.5.2 Default provider 23
2.5.3 Legacy provider 23
2.5.4 FIPS provider 23
2.5.5 Null provider 24
2.6 Third party providers 24
2.6.1 PKCS#11 provider 24
2.6.2 TPM 2.0 provider 25

3 Provider implementation 27
3.1 Our provider's core 27
3.1.1 Initialization function 27
3.1.2 Provider context 29
3.1.3 Teardown function 29
3.2 Information exchange 29
3.2.1 Dispatch table 30
3.2.2 Parameter tables and parameter getters 30
3.2.3 Query operation function 31
3.3 Key management 31
3.3.1 Key management implementation 32
3.4 Store 32
3.4.1 Store context 32
3.4.2 Opening the store 33
3.4.3 Loading from the store 33
3.4.4 Closing the store 33
3.4.5 Setting parameters to store context 33
3.5 Signatures 34
3.5.1 Signature context 34
3.5.2 Setting parameters to signature context 34
3.5.3 Signing 34
3.5.4 Demonstration of the provider 35
3.5.5 Results 35
3.5.6 Possible improvements 35
3.6 Practical usage 36
3.6.1 Building and installing the CNG provider 36
3.6.2 Loading the provider 36

Conclusion **39**

Bibliography **41**

A Acronyms **45**

B Contents of enclosed media **47**

List of Figures

1.1 Parts of OpenSSL 6
1.2 Parts of libcrypto 7
1.3 Parts of libssl 8
1.4 Command line tools 10

List of Listings

2.1 An example dispatch array for provider's base 14
2.2 OSSL_PARAM definition from the documentation 15
2.3 An example usage of OSSL_PARAM array 15
2.4 OSSL_ALORITHM definition 16
2.5 Example algorithm array 16
2.6 Example property names 21
2.7 Example property queries 22
2.8 Example file scheme usage 22
3.1 Example parameter table for providers core 30
3.2 Example part of parameter getter for providers core 31
3.3 Building and installing of the provider 36
3.4 Using provider from commandline 37
3.5 Using provider from code 38

Introduction

Cryptography is present in our lives on a day-to-day basis. It has gained its importance over the years and there is no sign of it becoming obsolete any time soon. As of 2023, there are estimates that more than 5 billion people are online[1] and their communication needs to be secured. And not only their communication but their stored personal data as well. Thus the need for cryptography.

Nevertheless, there is a slight caveat with it. It needs to be simple and easy to use for the end user to actually be useful.

OpenSSL library and its accompanying tools are one such solution giving users relative simplicity and ease of use when dealing with cryptography. Although they solve many problems and common scenarios, there are areas in which they are lacking.

One of these issues is cryptography on dedicated devices such as hardware tokens. The OpenSSL library is designed to be as self-sufficient as possible to achieve a high level of security. However, sometimes we as users or developers may also want to relay part of the trust to other sources as well. One of these sources might be considered our operating system, particularly its cryptographic storage.

Fortunately, OpenSSL has ways of dealing with such shortcomings. These were called engines in the earlier versions and have been superseded by a concept called providers in version 3.0.0 of OpenSSL. Through them, more functionalities can be added and easily shared with others by simply loading those providers into the OpenSSL binaries or integrating them into the OpenSSL code.

In this thesis, we will take a closer look at these providers, then discuss and implement a provider able to offload certificate work to an alternative cryptographic implementation. Specifically Cryptograpy API: Next Generation. This will allow us to take advantage of the strengths of both pieces of software: general availability, common usage, transparent access to secure hardware elements and operating system managed crypto storage.

Chapter 1

OpenSSL

OpenSSL describes itself as cryptography and SSL/TLS[1] toolkit[2]. It is a mature piece of software used in a variety of contexts, from generating RSA[2] keys for users to serving as a cryptographic backend to Nginx or Apache web servers.

1.1 Features

It comprises two main parts (`libcrypto` and `libssl` libraries) and additional tools. An excellent example of the accompanying tools would be one for visualizing the outputs of cryptographic data or a full-fledged HTTP(S)[3] client. A common, yet not primary use case for OpenSSL is printing human-readable content of X.509 certificates to the command line interface.

The libcrypto part is a library with a broad spectrum of algorithm implementations. Those range anywhere from symmetric cryptography to certificate handling. Algorithms implemented in this library are usually used to facilitate secure online communication. That has to do with the fact that the libssl library (the second major part of OpenSSL) strongly depends on this one.

This library can have multiple implementations of different algorithms. Usually, there are at least two versions of the most important algorithms: the *default* implementation available in the toolkit (used by default) and the *FIPS* certified version (that needs to be loaded explicitly). FIPS versions of algorithms are not that common since FIPS being a certification (of the US National Institute of Standards and Technology) puts additional requirements on the implementation that are usually unnecessary. Furthermore, complying with these requirements takes a lot of time and capital since there is a mandatory certification process.

[1]Secure Sockets Layer/Transport Layer Security
[2]Rivest–Shamir–Adleman
[3]Hyper Text Transfer Protocol (Secure)

The second part, the libssl library, is also a library but focused on providing Secure Sockets Layer, Transport Layer Security and Datagram Transport Layer Security (DTLS) capabilities to programmers. It allows for creating secure communication channels over insecure network connections. As have the standards for online data transfer evolved, so did the OpenSSL library. Today it offers anything from SSLv3 to TLSv1.3 for packets and DTLS 1.0 to DTLS 1.2 for datagrams.

As stated previously, libssl is dependent on libcrypto. Each communication protocol that libssl supports needs different algorithms to fulfill its function. More often than not, those are bundled in cipher suites. We can take a look at the smallest modern protocol TLSv1.3, which supports just five cipher suites, yet needs seven different algorithms for them. Considering that TLSv1.2 has 37 cipher suites, it is clear that it has been an excellent choice to separate those implementations into a standalone library.

1.2 Evolution

OpenSSL has been in development for many years, starting in 1998 as a fork of SSLeay. It has been developed under a double license for a long time: the OpenSSL license and the original SSLeay license[3], both considered open-source licenses. From version 3.0.0, it is now licensed under the Apache-2.0 license [4][5], thus simplifying its licensing yet maintaining its open-source status.

The project is not always perfect for every use case and carries with itself a significant technical debt. However, the functionalities are one of the broadest available, and the project has solved many problems that usually arise when creating cryptography software.

OpenSSL's imperfections lead to the creation of a few big forks. These were created for various reasons, such as a better and simpler user experience, minimizing attack surface and legacy code, or QUIC support. Names of these forks are LibreSSL[6], QuicTLS[7] and BoringSSL[8]. While QuicTLS is relatively new (created in 2021), LibreSSL and BoringSSL were both forked in 2014 after the critical Heartbleed vulnerability was discovered.

1.2.1 LibreSSL

LibreSSL's primary goal was to modernize the codebase to make it easier to audit, understand and repair, to apply best practices during the development process (such as code review and faster release times) and remove obsolete and broken features and support[9]. It has succeeded with its goals and has been the primary SSL/TLS library for significant distributions such as OpenBSD and Alpine Linux for some time.

Unfortunately, some design choices led to it being replaced by OpenSSL again. As it was supposed to be a drop-in replacement at the beginning, the library names and SONAMEs[4] were kept the same as in OpenSSL. This was not an issue at the start. However, even with little changes to the application binary interface (ABI) over time, the differences accumulated so much that developers of distributions needed to create patches to keep the ABI the same. So this relatively small and understandable design choice led to poor maintainability over time.

Also, because it is developed by a smaller team and without a long-lasting brand, LibreSSL has suffered to keep up as fast as OpenSSL to the introduction of new standards such as TLS 1.3, not to mention that OpenSSL has extensive support of platform-specific optimizations outside of the x86 architecture, which is something with which a small team cannot compete.

1.2.2 QuicTLS

Another fork is QuicTLS, which aims to add QUIC support to OpenSSL. It has mainly been a collaboration between Akamai and Microsoft. Its goal was not to create a standalone project as with LibreSSL; quite the contrary. Being compatible and allowing for quicker implementation of QUIC into OpenSSL has played a considerable role in this project.

As of today, there exists a QUIC minimal viable product from QuicTLS that should be merged into OpenSSL. That should happen in major version 3.2 [10] of OpenSSL, with the current version being 3.1.

1.2.3 BoringSSL

The last mentioned fork is BoringSSL. In comparison to QuicTLS, it is supposed to be its own product. Although it is open source due to the OpenSSL licenses, it is not intended for general use. It is a collection of patches and tweaks from Google for OpenSSL rather than an independent project.

The project is still in active development and is in a symbiotic relationship with both OpenSSL and LibreSSL. Bugs found and their fixes are generally shared among these three in advance (albeit it has not always been this way [11]) since they are all based on the same code. BoringSSL code has even been relicensed to an ISC license[12].

With all these forks being around to this date and still being actively developed, it is apparent that even after nine years, there are changes to OpenSSL that can be made to better suit the needs of some big interest groups.

[4]SONAME is basically a shared object's name.

1.2.4 Current state

Despite all this, the toolkit is still widely used. The implementations even adhere to very high standards such as FIPS 140-2[13][14]. There are currently plans to validate OpenSSL to meet the requirements of the FIPS-140-3 standard[15]. However, this certification is not expected to be completed before 2024, although the FIPS 140-3 compliant provider has been released a version before it was initially planned to[15][16]. It is currently undergoing the certification procedure.

1.3 Structure

As mentioned before, OpenSSL comprises multiple parts. The two main ones are `libssl` and `libcrypto` C libraries. The other parts are command line tools like `openssl` as seen in figure 1.1. Unfortunately, there are no graphical user interface options as a part of the project, but there have been some attempts at creating them by the community [17].

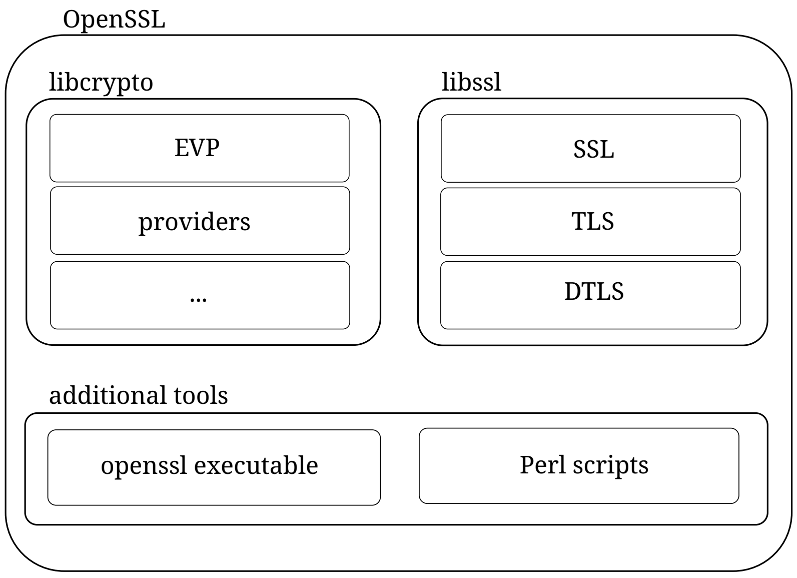

Figure 1.1: Parts of OpenSSL

1.3.1 libcrypto library

The part of OpenSSL responsible for cryptography is the `libcrypto`. It stores all of the implementations of cryptographic algorithms available in OpenSSL by default. It complements the SSL/TLS library also available in OpenSSL but can be used on its own.

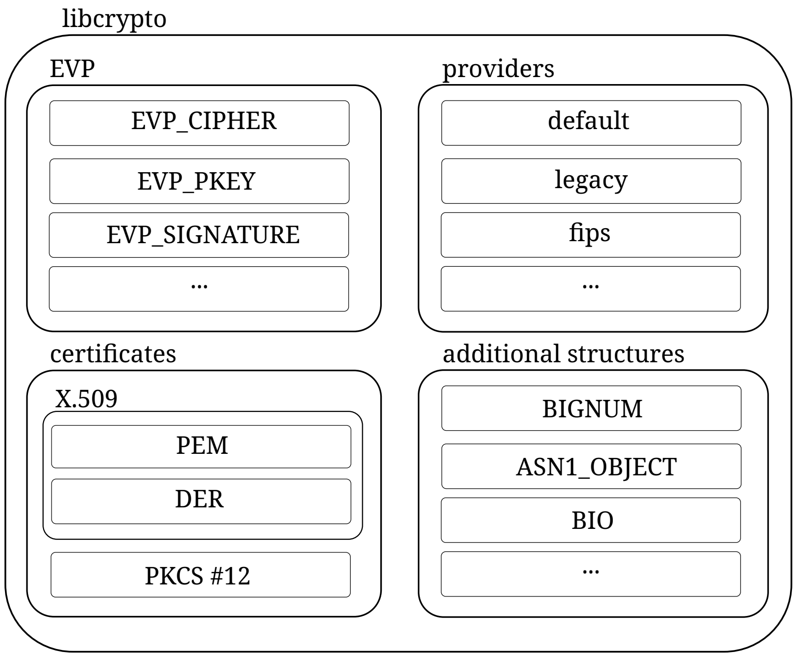

Figure 1.2: Parts of libcrypto - comparable to OpenSSL 1.1.1 version in [18]

To unify the terminology going forward, these are the relevant definitions from the OpenSSL documentation:

Operation something one wants to do, such as encryption and decryption, key derivation, message authentication code (MAC) calculation, signing and verification, and many other options. Operations are discussed in detail in section 2.4.2.

Algorithm a named method to perform an operation. The algorithms very often revolve around cryptographic operations but may also revolve around other types of operations, such as managing certain types of objects.

Most of the functionality of `libcrypto`, such as public key cryptography, random number generation or message authentication codes, is implemented in *providers*. We will discuss them later in greater detail. The rest of the library is an envelope providing APIs[5] for using platform resources and providers in a transparent manner, for example, fetching said algorithm implementations.

When fetching algorithms explicitly (which is more common), one might use functions such as `EVP_MD_fetch()`, `EVP_CIPHER_fetch()` or `EVP_MAC_fetch()`. These functions take an algorithm name as a parameter and return the re-

[5]application programming interface

quested implementation. There is a possibility to specify this implementation further with a concept called *provider queries*, which will be discussed later.

Implicit algorithm fetching happens when the generic operation functions are called with an object that already has a property query associated with it. This is a very common scenario for `EVP_PKEY` type structures. These hold the asymmetric key pair, and when used in conjunction with `EVP_DigestInit_ex()`, their associated property query is automatically used along with information about the provider from which the `PKEY` structure comes.

OpenSSL uses contexts to allow concurrent use in threads, and scopes,. All actions in `libcrypto` take action in a context; actions taken in different contexts should not affect each other. These contexts are defined by context structures that serve as arguments to many API calls where multi-threading or other concurrent use might be a concern.

1.3.2 libssl library

On top of `libcrypto` stands the `libssl` library, which serves as a collection of Secure Sockets Layer, Transport Layer Security and Datagram Transport Layer Security protocols. The `libcrypto` library is its very important dependency since these protocols depend on cryptography algorithms to work properly.

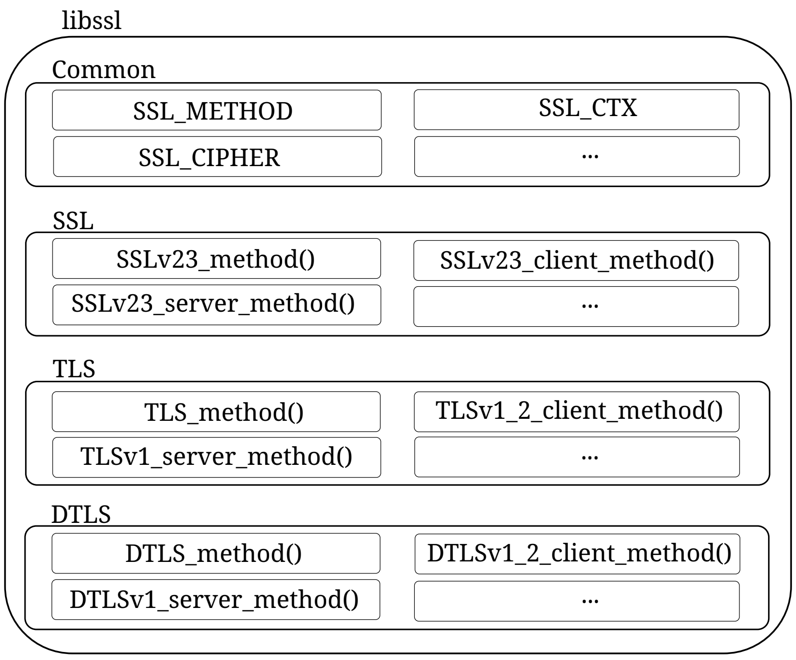

Figure 1.3: Parts of libssl

This library has its own object called `SSL_CTX`, which serves as a way to keep track of TLS/SSL sessions (`SSL_SESSION` objects), connections (`SSL` objects), ciphers (`SSL_CIPHER`) and methods (`SSL_METHOD`, for example, TLS1.3). Not only does this object keep track of the state the connection is in, but it also influences the progression of the connection, as details like whether to use a client certificate or not are also stored in it. Most of these can be set with `SSL_CTX_set_options()`.

1.3.3 openssl

The executable `openssl` is the main command line utility (which has the same name as the project itself). It holds most of the functionality that can be achieved with code using both `libcrypto` and `libssl`. It is divided into many smaller parts, each one solving problems regarding most common online cryptography problems.

These range anywhere from RSA and DSA[6] key generation, through password hashing, to full-fledged SSL/TLS clients and servers, not to mention more niche functionalities like prime number calculations, S/MIME[7] support and SSL server benchmarking.

The most common usage of this utility is key generation and translation. It can convert between PEM[8] and DER[9] and multiple cryptography key formats. Another common task is certificate manipulation. This command can create certificate authorities, sign certificates, revoke them and perform all the other necessary tasks when dealing with certificates.

1.3.4 Other command line tools

OpenSSL comes equipped with at least three other tools out of the box: `CA.pl`, `c_rehash.pl` and `tsget.pl`. They are all little known, and two of them have been, for the most part, reworked into the `openssl` command line utility.

Perl script `CA.pl` is a front-end to the `openssl` command for dealing with certificates. It allows easy creation, signing, cross-signing, conversion and revocation. Its abilities are very limited but suffice for simple tasks. This utility is only available when compiling OpenSSL and copying the Perl script from the build directory, as it does not get installed automatically and usually does not come with OpenSSL in any package manager. There are manual pages available for this script, however.

The script `c_rehash.pl` is also written in Perl. This script has been largely adopted as a sub-command of `openssl` called `openssl-rehash`. It creates symbolic links to certificates with the hash of said certificate as the link's

[6]Digital Signature Algorithm
[7]Secure/Multipurpose Internet Mail Extensions
[8]Privacy-Enhanced Mail
[9]Distinguished Encoding Rules

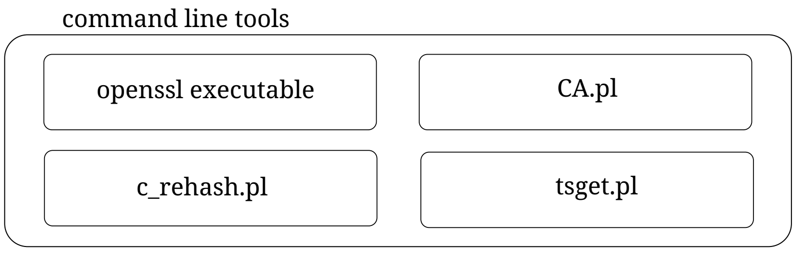

Figure 1.4: Command line tools

name. This is mainly due to the fact that many programs that use OpenSSL expect this type of hash-based file structure to find certificates. This tool gets installed automatically with OpenSSL 3.1.0.

The third utility is `tsget.pl`, again hidden like `CA.pl`. This program's purpose is to send and receive timestamp requests. It has mostly been adopted as a sub-command into `openssl-ts`.

CHAPTER 2

OpenSSL providers

With version 3.0 of OpenSSL came *providers*. They are a tool to enhance the library with alternative and new algorithm implementations. They are similar to *engines*, a concept that has been available since version 0.9.6 (although they only appeared in the mainstream from 0.9.7). As with engines, their purpose is to allow third parties to implement new algorithms into OpenSSL. This is very helpful when developing with algorithms found to be obsolete, for example, `RIPEMD160` (even though this particular algorithm has an exception [19]) since they are not necessarily present in the OpenSSL library.

Although OpenSSL maintains a provider for algorithms that were widely used but are now deprecated. It is called the Legacy provider. We will discuss this provider in more depth later in section 2.5.3.

Providers offer an advantage over engines in the sense that they are interwoven into the library in a tighter way. It is possible to specify which calls should be made through which provider in a particular application. That can be achieved either directly through provider queries or via a configuration file. This gives the programmer a more granular and predictable way to interact with the OpenSSL APIs when using providers, in contrast to engines.

Providers are also designed to be high-level tools. So, for example, when implementing an AES encryption, the user will have `Init()`, `Update()` and `Final()` style functions exposed but nothing like `AddRoundKey()`. This is in compliance with the OpenSSL team's efforts to streamline and simplify the library's APIs.

2.1 Purpose

As mentioned before, the main goal of providers is to allow third parties to provide alternative algorithm implementations. The advantage of using providers is in the fact that the user can use the rest of the toolkit without the need to implement those functionalities themselves.

Providers can be both compiled into OpenSSL for custom builds or can be dynamically loaded. Compiling them directly and distributing a custom version of OpenSSL, for example, in the form of packages, can be the easiest way to share a version that exactly meets the needs of a large company with specific requirements. On the other hand, the possibility to load the providers dynamically allows all users of OpenSSL versions that support providers (even the custom ones!) to use them without the need to recompile the whole source code, thus allowing for a quick and easy deployment for new algorithm implementations and functionalities.

There are many use cases for this. One in particular is adding support for some custom hardware device to OpenSSL. The device manufacturer releases drivers for their device for an operating system but will not support OpenSSL, nor will OpenSSL support that device. Using providers, this problem can be easily solved, as discussed by Pešek in [18]. Another one might be when developing a tool using proprietary (or non-standardized) algorithms. OpenSSL will not support them, but all the benefits of OpenSSL infrastructure can be taken advantage of if they are implemented with providers.

2.2 Limitations

Unfortunately, providers are not the perfect solution to extensibility. While other software provides plugins, modules or extensions for a broader interaction with it, OpenSSL's only way to extend the software is to change its source code or use just the limited scope of providers.

To illustrate: many developers want to take advantage of the QUIC protocol. It is a transport layer network protocol designed to replace TCP. OpenSSL, by design, works with TCP to implement TLS to secure such communications. However, adding support for QUIC to OpenSSL is impossible without modifying the source code. Providers can only be used to implement algorithms used for cryptography. That includes key exchange, key management, random number generation, key derivation functions and many others, but not transport layer protocols.

As of today, programmers wanting to use developing standards must wait for official releases tackling these issues or use alternative solutions and forks such as BoringSSL.

2.3 Interactions

Although providers seem to be self-contained pieces of software, they do not need to be. There are a few possibilities for interaction and cooperation between them. It is rare since functionalities that a provider would provide are often tied to a narrow use case. Still, the OpenSSL libraries allow for it.

One possible way that providers can interact is through decoders. Decoding functions are special functions used to transform data. Usually, they are given data loaded from outside the application and are then supposed to convert it to either something that is understandable by their provider or an intermediate object that can be processed by another decoder (either from the same provider or a different one).

This creates a dynamic chain of loaded providers for a particular input data type. The OpenSSL core will pass the input data format to the first applicable decoder and then process the format returned from it further with the rest of the loaded decoders until it can process it no more or has reached the desired output format[10]. The intermediate formats can be anything from well-known formats like PEM or proprietary formats specific to that particular provider.

Another option are property queries. When developing a program using OpenSSL, the developer can ask for certain implementations of algorithms. That is done with property queries which are essentially a string with key-value pairs (although some other options are available, as discussed later in section 2.4.15). Any provider can specify its own properties, even ones that are used by other providers. For example, a custom provider might state that its algorithms are FIPS compliant and OpenSSL might use those in preference to the actual FIPS provider ones.

That is not a vulnerability, as only trusted providers should be loaded. Quite the contrary, the FIPS provider might not provide all the necessary algorithms one might desire but implement a great amount of them. Adding a custom provider that implements all of them would be unnecessary and resource-consuming. But implementing just the missing ones, marking them as FIPS compliant and letting OpenSSL interchange information between the FIPS provider and the custom one is most definitely a desired feature.

2.4 Provider infrastructure

In the OpenSSL documentation, providers are defined as follows:

Provider a unit of code that provides one or more implementations for various operations for diverse algorithms that one might want to perform.

We will refer to the OpenSSL libraries as the OpenSSL core as that is the terminology for public types used to communicate between the OpenSSL libraries and providers. To the functionalities of the provider, excluding the operation functions, we will refer to as the provider base.

[10]or more precisely, finds the desired algorithm input. More on that later in section 2.4.12

2.4.1 Special data types

The OpenSSL libraries implement a few new data types, especially for providers. Their main function is to be a multi-purpose, self-sufficient way for passing data between providers and the core and sometimes even between providers themselves.

OSSL_DISPATCH

The first data type is an array of `OSSL_DISPATCH`. This is a tuple of a function pointer and an integer ID describing the function that the function pointer points to. These IDs are available in `<openssl/core_dispatch.h>` as macros; their names start with `OSSL_FUNC_`.

Each function ID is only usable with the associated operation (which have been mentioned in section 1.3.1). That is because the actual integer values can be reused in between operations. `OSSL_FUNC_KDF_RESET` has the same value as `OSSL_FUNC_MAC_INIT`, yet it is perfectly fine since the dispatch arrays should always be used in the context of a particular operation known to both sides passing the array.

There is a stopping element defined in the form of {`0, NULL`}[11] that always needs to be present in the array even if it were to be the only element in it. The core often implements enumerating this array with a `while` loop waiting for this element to show up to cease.

Listing 2.1: An example dispatch array for provider's base

```
static const OSSL_DISPATCH cng_dispatch_table[] = {
 {OSSL_FUNC_PROVIDER_GETTABLE_PARAMS, cng_gettable_params_voidp},
 {OSSL_FUNC_PROVIDER_GET_PARAMS, cng_get_params_voidp},
 {OSSL_FUNC_PROVIDER_QUERY_OPERATION, cng_query_operation_voidp},
 {OSSL_FUNC_PROVIDER_TEARDOWN, cng_teardown_voidp},
 {0, NULL}
};
```

The expected behavior when reading a dispatch array is to ignore anything not recognized, thus ensuring backwards compatibility.

When there is the need for this array to be passed, the provider needs to make sure that the pointer passed back to the core is properly allocated and that the last item of the array is actually the stopping element. The core trusts the provider and often does not perform checks, which can lead to anything from a silent fail to an outright crash.

[11] This thesis has sparked the idea to implement an `OSSL_DISPATCH_END`. Its adoption is undergoing: `https://github.com/openssl/openssl/issues/20710`

OSSL_PARAM

The second custom data type is the `OSSL_PARAM` array. Its purpose is to transfer parameters. Since parameters themselves can have many different data types, `OSSL_PARAM` is a way to encapsulate those data types along with the information that is carried in them.

Listing 2.2: OSSL_PARAM definition from the documentation [20]

```c
typedef struct ossl_param_st OSSL_PARAM;
struct ossl_param_st {
  const char *key;             /* the name of the parameter */
  unsigned char data_type;     /* kind of content in data */
  void *data;                  /* value being passed in or out */
  size_t data_size;            /* data size */
  size_t return_size;          /* returned size */
};
```

There are four main data types supported: integers, floating point numbers, UTF-8 encoded strings and octet strings. The last one is the most useful since it allows passing arbitrary data just labeled as an octet string. Though unsigned integer and pointer variants for both types of strings do exist as well, they are less common.

There are many predefined constants that can be used in these parameter arrays, and they are all defined in `<openssl/core_names.h>`.

Listing 2.3: An example usage of OSSL_PARAM array

```c
static const int object_type_cert = OSSL_OBJECT_CERT;
OSSL_PARAM cert_params[] = {
 OSSL_PARAM_int(OSSL_OBJECT_PARAM_TYPE,(int *)&object_type_cert),
 OSSL_PARAM_octet_string(OSSL_OBJECT_PARAM_DATA,
  store_ctx->prevCertCtx->pbCertEncoded,
  store_ctx->prevCertCtx->cbCertEncoded),
 OSSL_PARAM_END
};
```

OSSL_ALGORITHM

The third data type is intended for passing algorithm implementations between the libraries and providers. This data type contains information about the algorithm names, their common property definitions, `OSSL_DISPATCH` table pointer with the actual implementation, and optionally a description. Its definition can be found in listing 2.4.

It is used when passing information about supported operations from the providers `query_operation()` function to the OpenSSL core. An example of such an array that might be passed is shown in listing 2.5.

The `algorithm_names` should be a colon-separated list of names to identify the desired algorithm. Those can even be object identifiers as per the RFC 8017 appendix C[21] or their canonical decimal text form.

The last item in such an array should always have its `algorithm_names` set to a `NULL` pointer to signal the end of the array; failure to do so might lead to unexpected behavior.

Listing 2.4: OSSL_ALORITHM definition

```c
struct ossl_algorithm_st {
  const char *algorithm_names;     /* key */
  const char *property_definition; /* key */
  const OSSL_DISPATCH *implementation;
  const char *algorithm_description;
};
```

Listing 2.5: Example algorithm array

```c
static const OSSL_ALGORITHM cng_keymgmt[] = {
  {"RSA:rsaEncryption", CNG_DEFAULT_ALG_PROPERTIES,
    cng_keymgmt_functions, "CNG␣Provider␣RSA␣Implementation"},
  {NULL, NULL, NULL}
};
```

2.4.2 Operations

As stated before, OpenSSL providers can consist of multiple operation groups. These operations are just bundles of functions performing related tasks to each other. Some are standalone, though some interchange information between each other.

The functions present in an operation can be further divided into logical groups, which we shall refer to as function bundles. Usually, not all bundles are necessary to implement, yet it is recommended to achieve full functionality and avoid unexpected refusal of OpenSSL to use your provider.

2.4.3 Key management

The operation that interacts the most with the others. The key management operation with the operation number `OSSL_OP_KEYMGMT` is intended primarily for the internal use of the provider. Its purpose is to manage the private-public key pairs. It does it completely transparently to OpenSSL core. The core's only concern is the `EVP_PKEY` structure which gets filled with the relevant information about the key.

The basic bundle contains functions to create, duplicate and destroy a provider-side key object. Those three are almost always needed. The next bundle are key generation functions that allow the user to customize and generate the provider-side key objects. Then there are bundles for import and export, and finally, a bundle for comparison and one for querying information about the keys.

There are special constants defined for dealing with parts of keys since a private-public key pair can consist not only of its private and public parts but also can contain domain parameters or even other additional data. Those constants all bear the name of `OSSL_KEYMGMT_SELECT_*` and are defined in `<openssl/core_dispatch.h>`. The common combinations of these constants with the same naming scheme are also available.

2.4.4 Store management

Store management operation is a provider-based backend to the `ossl_store` API. That means that this operation is designed to return keys, certificates, certificate revocation lists and additional information in the form of names and parameters of applicable objects. It has an operation number of `OSSL_OP_STORE`

Its intended usage is the same as with the `ossl_store`. The user is expected to either open a store based on a Uniform Resource Indentifier (URI) or attach to an existing BIO structure of OpenSSL (which serves as an abstraction for any input/output method), then enumerate through the whole store with a loading function and end when you reach the end of the store. Additionally, there are functions to set some parameters (such as issuers, fingerprints or aliases) when searching for data in the store and a function to export a given loaded object to a foreign supported provider format.

When the loading function is called, it is given a callback function into which it should pass the loaded object in an `OSSL_PARAM` array. In that array, a parameter with the key `OSSL_OBJECT_PARAM_TYPE` should be defined to tell the callback what is actually being passed into it. An array that could be passed into the callback is shown in listing 2.3.

Stores can also return provider-side objects. For example, when loading a key, the key can be a provider-side key object, and the OpenSSL core will deal with it accordingly. That means that it will try to find operations it wants to perform in the loaded providers and use them. It must be marked so in the `OSSL_PARAM` array by setting the `OSSL_OBJECT_PARAM_REFERENCE` key. When the `OSSL_OBJECT_PARAM_VALUE` is set, it is assumed that it is DER encoded data and thus processable by the OpenSSL libraries.

2.4.5 Digests

Implementations of digests are collected in the `OSSL_OP_DIGEST` operation. These implementations serve as a provider backend to APIs like `EVP_Digest` or `EVP_Q_digest`. Their main purpose is to allow the creation of digests.

This operation's bundles are the base bundle which creates, duplicates and frees a provider-side digest context. Another one allows for the actual digest processing, which means initializing, updating and finalizing a digest (or doing all of that with one one-shot function). The last bundle allows for setting and getting parameters from the digest itself or the digest context.

2.4.6 Symmetric ciphers

Operation with the OSSL_OP_CIPHER operation number. The functions here serve as a provider backend to OpenSSLs API for encryption and decryption, mainly functions like EVP_Cipher, EVP_EncryptInit_ex and its opposite EVP_DecryptInit_ex. Here are all the different implementations of encryption and decryption algorithms stored.

Cipher operation bundles are similar to the digest bundles. The base bundle that creates, duplicates and frees a provider-side cipher context, the actual cipher bundle, which allows full progress of en/decryption and the last bundle, which allows for setting and getting parameters from both the actual cipher or the cipher context.

The last bundle allows for customizing (for applicable ciphers) values like padding type, number of rounds to be used, or effective key bits.

2.4.7 Message authentication codes

This operation with number OSSL_OP_MAC is the provider backend to functions creating message authentication codes like EVP_Q_mac and EVP_MAC_init. It is very similar to the Digest operation in the manner it operates in.

It also consists of three bundles. One for context manipulation, another one for message authentication code processing and the last one for setting and getting parameters from either the context or the message authentication code itself.

2.4.8 Key derivation

Operation for deriving keys with the number OSSL_OP_KDF. The included functions serve as provider backend to EVP_KDF_derive and similar functions. They are used to implement additional algorithms for creating one or more secret keys from a secret value passed as an input to them.

There are three bundles present. The base bundle, which manipulates the key derivation context, the derivation bundle, which initializes and derives the actual key and the parameter bundle, which gets and sets parameters to the context or the derivation itself.

This operation should not be confused with the Key exchange operation. They perform similar tasks, but both serve as backends to different APIs.

2.4.9 Key exchange

The Key exchange operation with the number OSSL_OP_KEYEXCH takes care of the provider backend of key exchange algorithms. Its functions are part of the EVP_PKEY_derive function set.

The bundles present are for context manipulation, the key exchange itself, and for parameter manipulation, as with previous operations. This operation can easily be confused with the Key derivation operation since it performs similar tasks yet belongs to different OpenSSL APIs.

2.4.10 Asymmetric ciphers

OSSL_OP_ASYM_CIPHER is very similar to its symmetric counterpart version. This operation implements the provider backend to asymmetric cipher algorithms for functions like EVP_PKEY_encrypt and EVP_PKEY_decrypt.

There are, once again, three bundles. One for context manipulation, one for encryption and decryption, and one for getting and setting parameters from the asymmetric cipher context.

2.4.11 Asymmetric key encapsulation

The Asymmetric key encapsulation operation identified with OSSL_OP_KEM serves as provider backend to EVP_PKEY_encapsulate, its opposite function EVP_PKEY_decapsulate and other related functions.

Functions belonging to this operation can be divided into three distinct bundles: the context manipulation bundle that allows for creating, duplicating and freeing a key encapsulation context, the actual en- and decapsulating functions, and the getters and setters for the encapsulation context.

2.4.12 Encoding and Decoding

Those are two operations but are closely tied together. Their respective numbers are OSSL_OP_ENCODER and OSSL_OP_DECODER. The functions from these operations are not accessible directly from the normal user space. Instead, they are used by the OpenSSL core to change the encoding of different objects.

A typical use case is transforming a PEM-encoded object into its DER form or vice versa. But these functions can even be used to transform encoding only one provider understands into an encoding another provider can use. Even more, the OpenSSL core can chain these encoders and decoders and create the desired output from a given input with many intermediate encoded objects. That implies that a conversion from PEM to DER to provider-side object and the other way around is possible.

Moreover, there can exist a provider whose whole job is just converting between different encodings and thus serving as an intermediate layer between two other providers without doing any other operations at all, essentially becoming a structural pattern known as a *wrapper* or an *adapter*.

These two operations have similar bundles. Both have a bundle for context manipulation, a bundle for getters and setters of parameters of those contexts, a bundle for selection (which allows selection of different parts of processed objects if applicable), and finally, each of them has their respective encoding and decoding bundle, each containing their respective import and export functions.

2.4.13 Random number generation

The generation of random numbers is handled by the operation with number `OSSL_OP_RAND`. It is the provider backend for functions in the `EVP_RAND` API. It allows not only for the implementation of pseudo-random number generation algorithms but true random number sources as well.

It does not have a function for copying its context, unlike all the other operations, since such behavior is undesirable. This operation's context manipulation bundle contains functions for creating and destroying the context as usual but also has a function for locking and unlocking the context and thus providing exclusive access.

The random number generating bundle is split into two parts, the NIST part, which corresponds to NIST SP 800-90A and SP 800-90C, and additional functions like nonce generation and seed extraction.

The last bundle is dedicated to parameter setting and getting from both the random number generator and its context.

2.4.14 Signatures

This operation is the most fragmented one. Its number is `OSSL_OP_SIGNATURE`, and it serves as a provider backend to functions like `EVP_PKEY_sign` and its counterpart `EVP_PKEY_verify`. Its purpose is to allow the implementation of various signature schemes (and in conjunction with key management operation) with various keys.

Into its bundles belong the common context manipulation, the signatures, signature verification, recoverable signature verification and recovery, digest signature, digest verification and parameter manipulation twice — once for the signature itself and once for the message digest.

2.4.15 Algorithm selection

Since all the functions actually implemented by a provider are called through a higher-level API and multiple providers can implement the same algorithms, there is a need to be able to distinguish between them and only select the appropriate implementation. That is where property queries come into play.

Properties in OpenSSL 3.0.0 and higher are basically text representations of key-value pairs. They are separated into two categories, *reserved* and *user defined* properties. Both are case-insensitive, but OpenSSL uses only lowercase.

Reserved are the ones defined by OpenSSL. They consist of a single C-style identifier without a leading underscore. The *reserved* properties defined at the writing of this thesis are `provider` (which should have the provider's name as its value), `fips` (which should refer to whether or not the algorithm implementation is FIPS certified), `type` (which refers to the type of data to be encoded and is defined only for the base provider, more about it in section 2.5.1) and `format` (which refers to the output format of the base provider's encoding).

User defined properties are similar to *reserved* but consist of two or more C-style identifiers separated by periods. This creates a kind of namespace architecture. The recommended usage can be found in lines one and two in listing 2.6 and an example in lines three and four.

Listing 2.6: Example property names

```
1  <provider_name>.<property_name>
2  <provider_name>.<algorithm_name>.<property_name>
3  cng_provider.sha256.padding_type
4  cng_provider.deprecated
```

When using a higher level API function of OpenSSL and trying to perform a certain operation, the user can, either directly or through a context, set the desired properties. The queries can be either an equality test, inequality test, mark of preference or a discard mark to invalidate a previous (higher level) query. They are represented by `=`, `!=`, `?` and `-` respectively.

The queries themselves are comma-separated property query clauses. No property name can be in one query multiple times, and in case it is, it's considered an error. In listing 2.7, we can see in line one a correct query that asks that the algorithm being fetched has the `provider` parameter set to `cng_provider` and does not have the `fips` parameter set to `yes`.

The query in the second line says that the algorithm preferably has the key `cng_provider.deprecated` set to `no` and the selection mechanism should not consider anything previously set to `cng_provider.sha256.padding_type`.

In the third line, there is an invalid query since it contains the same parameter name twice, even though it is the same key-value pair.

Listing 2.7: Example property queries

```
"provider=cng_provider,fips!=yes"
"?cng_provider.deprecated=no,-cng_provider.sha256.padding_type"
"provider=cng_provider,fips=yes,provider=cng_provider"
```

2.5 Built-in providers

There are five built-in providers available in the standard build of OpenSSL. Those are the default, base, legacy, null and FIPS providers. They all serve a specific purpose to the core of OpenSSL.

2.5.1 Base provider

This provider is quite simple, offering only algorithms for encoding and decoding OpenSSL keys. Its property query contains `provider=base`. Even though encoding and decoding have little to do with FIPS certification, their functions do have the `fips=yes` property set to allow usage with the FIPS provider.

It has two extra properties defined: the `type` parameter, which can have values `parameters`, `private` and `public`, referring to parts of keys, and `format`, with values `der`, `pem` and `text` referring to the output format of the providers encoding.

Operations supported by this provider are thus `Encoding` and `Decoding`. The user can use this provider with the `STORE` API using the scheme `file`. Some valid example calls are in listing 2.8. The first two are for the Windows environment, and the last two are for the Linux environment. This scheme demands the scheme name and a slash after the colon to be valid. The normal URI variant of writing a scheme works as well, but the starting slash still needs to be present. Only absolute paths are supported.

Listing 2.8: Example file scheme usage

```
OSSL_STORE_open_ex("file:/C:/Users/Ladislav␣Marko/privkey.pem",
  NULL, NULL, NULL, NULL, NULL, NULL, NULL);
OSSL_STORE_open_ex("file:///C:/Users/Ladislav␣Marko/pubkey.pem"
  , NULL, NULL, NULL, NULL, NULL, NULL, NULL);
OSSL_STORE_open_ex("file:/home/Ladislav␣Marko/privkey.pem",
  NULL, NULL, NULL, NULL, NULL, NULL, NULL);
OSSL_STORE_open_ex("file:///home/Ladislav␣Marko/privkey.pem",
  NULL, NULL, NULL, NULL, NULL, NULL, NULL);
```

This provider's functionality is also implemented in the default provider.

2.5.2 Default provider

As the name suggests, it is the provider that is automatically loaded on any OpenSSL usage unless defined otherwise by explicitly loading another provider. At that stage, this provider must be loaded explicitly to be used as well. It contains all of the most commonly used algorithm implementations. Its property query is `provider=default`.

Example algorithms implemented in this provider are `SHA2`, `AES`, `HMAC`, `ECDH` and `RSA`. All of the other non-obsolete algorithms are implemented here as well.

It implements all of the available operations despite the documentation saying otherwise[12].

2.5.3 Legacy provider

This provider serves as an archive of algorithms. It contains algorithms deemed legacy or deprecated. That includes algorithms that are insecure, not used commonly anymore, or superseded by other algorithms.

These obsolete algorithms include `MD2` (as the Internet Engineering Task Force has moved it to *historic* status in RFC 6149[22]), `DES` (which has been cracked by the Electronic Frontier Foundation in 1998[23]), `RC4` (deprecated in TLS in RFC 7465 [24]) or `PBKDF1` (superseded by `PBKDF2` in RFC 2898[25]).

To use the implementations of this provider specifically, the property query is `provider=legacy`.

2.5.4 FIPS provider

As stated previously, OpenSSL 3.1.0 has FIPS 140-2 certification. To maintain this certification easily, all the necessary code is condensed into the FIPS provider. Thus when auditing, all the code is in one place. It also means that changes made to the rest of OpenSSL do not result in the need for re-certification since the boundaries are well set.

This provider has two properties defined, `provider=fips` and `fips=yes`. That is needed because not all algorithms implemented in the FIPS provider are actually FIPS certified (at least yet; for example, the Triple DES ECB or EdDSA). So to be FIPS compliant, the property `fips=yes` should be in the user's property query.

There is a slight caveat with this provider as for it to be available, OpenSSL needs to be built in a special way. During the configuration phase of the build, the `enable-fips` flag must be present. This will ensure that the FIPS shared library will be built and the `fips.so` (on Unix-based distributions) or `fips.dll` will be present and loadable into the OpenSSL libraries and tools.

[12]The author of this thesis has submitted tickets to the GitHub page of OpenSSL with numbers 20742 and 20743 regarding this issue.

The build process even allows the FIPS provider to be built and installed separately with `install_fips` *make* command. This allows backporting new implementations to the OpenSSL system without changing any of the APIs.

The FIPS provider also includes (as the certification mandates) self-testing. With a series of integrity checks, known answer tests (KAT) and additional testing, the provider validates itself on each startup. However the KATs can be disabled with configuration files after being run once during installation. There is even a callback option so the user can receive the test results directly with `OSSL_SELF_TEST_set_callback`.

2.5.5 Null provider

Because it supports no operations and algorithms, this provider might seem like a useless one. Yet it is not. When used, it ensures that no accidental automatic loading of the default provider will happen. It serves only as a guarantee that the user will not have algorithms from the built-in providers available by default.

2.6 Third party providers

The community has been slow to adopt providers, and there are not many available publicly. This has to do with the fact that providers themselves are not a very mature concept and are a fairly new addition to OpenSSL. Additionally, the OpenSSL itself may provide enough functionality that the use of providers is just not necessary. However, there are at least two notable mentions of third-party providers.

2.6.1 PKCS#11 provider

There is a maintained version of PKCS#11 provider[26], allowing for access to hardware and software tokens via the PKCS#11 interface. This provides OpenSSL with the ability to access compatible storage in suitable environments.

Unfortunately, the documentation is very bare and not much information apart from installing and configuring this provider is given.

It is licensed under the Apache 2.0 license and a special license based on OASIS Intellectual Property Rights policy[13].

[13] https://www.oasis-open.org/policies-guidelines/ipr/

2.6.2 TPM 2.0 provider

Another third-party provider is the Trusted Platform Module 2.0 integrating one[27]. Its purpose is to allow OpenSSL access to data through TPM 2.0.

It is based on the tpm2-tss-engine, which implemented this functionality through the now-obsolete engines API of OpenSSL prior to version 3.0.0. Currently, it is licensed under the BSD 3-clause revised license.

This provider has well-written documentation about the possibilities of its operations, the installation and the general usage. Additionally, there is also a mailing list for this provider.

Chapter 3

Provider implementation

In this chapter, we will take a closer look at what needs to be implemented in order to achieve the goal of this thesis: a provider able to offload certificate work to Cryptograpy API: Next Generation. The main goal will be to load a certificate and its private key from the CNG store and perform the necessary digest signing to establish a TLS 1.3 connection via OpenSSL. To simplify matters, we will only focus on RSA keys and assume that all certificates in our CNG store have their private keys attached and are set to be able to sign.

From the chapter about operations, we know that we will need the `STORE` operation for loading the certificates, the `KEYMGMT` operation to have our own representation of keys (because CNG will not give us the actual private key, just its handle) and the `SIGNATURE` operation to sign the digest to prove we own the private key for the certificate being sent during the TLS handshake.

From CNG, we will need to load a client certificate and its public and private keys (with the private key represented by its CNG handle). Then we will need to sign a digest with the loaded private key.

3.1 Our provider's core

First, we will examine the necessary functionality of what the provider needs to implement and then take a look at how we are going to approach the implementation. The implementation of this section can be found in the header file `cng_provider/cng_provider.h` and its `cng_provider/cng_provider.c`.

3.1.1 Initialization function

A provider, being a piece of code loadable by the main OpenSSL libraries, needs to have a joint meeting point with them. That is achieved with the initialization function. It is a function defined by the provider developer that serves as the first exchange spot for the core and the actual provider implementation.

Its purpose is to initialize the communication between the core and the provider and allow the provider to set up its internal structures. The core hands the provider two items:

- a dispatch array of *base functions* that the OpenSSL core provides to the provider, such as thread, error and memory functions, and
- its *handle*, which is an object that describes that particular provider to the core. The handle is, by design, transparent to the provider and only useful in some cases when the provider calls *base functions* received from the core with the *handle* as their argument.

The provider also passes back two items:

- its own dispatch array of *base functions*, including functions that allow querying the provider about useful information, a self-test function of the provider and a teardown function, and
- the *provider-side context*, which is meant to allow multiple instances of a provider running simultaneously. It will often be passed as an argument from the core to the provider's own functions. It is transparent to the core, which will not modify it or try to extract information from it directly.

The initialization function does not need to be named anything in particular when the final provider is being compiled directly with OpenSSL. The only thing it needs to have is the correct signature of `OSSL_provider_init_fn` declared in `<openssl/core.h>`. However, when the provider is supposed to be dynamically loadable, its initialization function specifically needs to be named `OSSL_provider_init` and be exported.

While not explicitly stated in the documentation, the return value is supposed to be zero when the initialization was successful and non-zero (preferably 1) otherwise.

Also, care should be taken regarding the outgoing dispatch array. It has to be a pointer to valid memory and a valid `OSSL_DISPATCH` array. That means that the last element of the array always has to be {`0, NULL`}.

Initialization function implementation

Practically, we should do self-testing in this stage and create the necessary structures that will last over the whole lifespan of the provider. These structures should be saved in an easily transferable structure which we will call the provider context.

Since we are essentially building a wrapper around CNG and it does not need self-tests, we will only allocate memory for the provider context structure and set its values. More on them in the following sections.

3.1.2 Provider context

To OpenSSL core, provider context is just a `void` pointer that is being passed to all functions the core thinks will need it, e.g., functions that initialize operations. It is also passed to functions that are supposed to tell the OpenSSL core information about the provider.

In case our provider had allocated some structures in the initialization function, they should be accessible to us via this context in subsequent calls. Even these calls can modify the context and allocate and free the structures we deem necessary. As this context will be available to us in the `teardown`, it can later be used to free any of the structures we allocated in it during the provider's existence.

Provider context implementation

For us, the context is going to be just a structure with one pointer to the functions passed to us by the OpenSSL core so that we can use them in later stages. The implementation is going to be called `T_CNG_PROVIDER_CTX`.

The context is a good place to carry mutexes, results from self-tests and other additional data, but since we do not need those, we will simply leave them out.

3.1.3 Teardown function

In the teardown function we should free the allocated memory, and if we saved some critical information, we should destroy it in a secure manner. This is where closing open sockets and file descriptors should happen because it is the last place where we have control.

Teardown function implementation

For us, this function, which we will call `cng_teardown()`, just frees the context we allocated in the initialization function. Since the only object in the provider context is a constant pointer to `OSSL_DISPATCH` array which we are not supposed to free, this is enough.

3.2 Information exchange

Since the OpenSSL core wants some information back from the provider, it is necessary to establish functions and structures that will implement this exchange.

3.2.1 Dispatch table

We talked about dispatch arrays earlier in section 2.4.1. A minimal provider needs to implement a dispatch table that looks like the one in listing 2.1 (which is our actual implementation). That means it passes the core information about which functions to use to ask for what parameters it can get, to actually ask for the parameters, to ask which operations are supported and how to access them, and how to tear down the provider itself.

3.2.2 Parameter tables and parameter getters

In general, parameter tables and their respective getter are used throughout the entire provider, and there is one for the provider itself and one for every supported operation. It is an array of `OSSL_PARAM` structures discussed in section 2.4.1.

Parameter tables and parameter getters implementation for the providers core

Typical values that the provider gives to the core are its printable name, version, information about how it was built (parameters, platform and other interesting information), and its status. Ours will look like the one in listing 3.1, although it is possible to write the `OSSL_PARAM` structures directly or use their respective types for different data types like `OSSL_PARAM_utf8_ptr()`. So it will tell the OpenSSL core that it can expect the typical values stated previously in their typical data types.

Listing 3.1: Example parameter table for providers core

```c
static const OSSL_PARAM cng_param_types[] = {
  OSSL_PARAM_DEFN(OSSL_PROV_PARAM_NAME, OSSL_PARAM_UTF8_PTR, NULL
    , 0),
  OSSL_PARAM_DEFN(OSSL_PROV_PARAM_VERSION, OSSL_PARAM_UTF8_PTR,
    NULL, 0),
  OSSL_PARAM_DEFN(OSSL_PROV_PARAM_BUILDINFO, OSSL_PARAM_UTF8_PTR,
     NULL, 0),
  OSSL_PARAM_DEFN(OSSL_PROV_PARAM_STATUS, OSSL_PARAM_INTEGER,
    NULL, 0),
  OSSL_PARAM_END
};
```

The getter will simply search the given parameter array, and if it finds a parameter it understands, it will fill the parameter with corresponding data. There are helper functions available for each data type supported by OpenSSL. An example part of a function passed to the OpenSSL core in `OSSL_FUNC_PROVIDER_GET_PARAMS` parameter is shown in listing 3.2.

Listing 3.2: Example part of parameter getter for providers core

```c
p = OSSL_PARAM_locate(params, OSSL_PROV_PARAM_NAME);
if (p != NULL && !OSSL_PARAM_set_utf8_ptr(p,
    CNG_PROVIDER_NAME_STR))
  return 0;
```

These values passed back to the OpenSSL core are only information for the users; the core does not act upon them, so they can be anything we find appropriate. We have decided to pass static strings back to the OpenSSL core, and for the `status`, we have implemented a function that checks the status of the provider as it is done this way in the OpenSSL providers. However, we have not found much functionality for it, so the function `cng_prov_is_running()` is, for now, a dummy one that always returns `true`. It serves as future-proofing, as when the need arises to tell the core our status, the necessary structures are in place.

The whole getter's implementation can be found in `cng_get_params()`.

3.2.3 Query operation function

This is a function passed to the OpenSSL core in a dispatch array mentioned earlier. It allows the core to ask the provider for an operation. This function receives an operation ID and returns a pointer to a `OSSL_ALGORITHM` array with all the functions implemented in this operation.

Query operation function implementation

For our implementation, `cng_query_operation()` is going to be a simple switch statement as we do not need to use any additional information passed to us other than the `operation_id`. If we have such an operation implemented, we pass it. Otherwise, we return `NULL`.

3.3 Key management

As stated previously, OpenSSL allows us to have our own key representation. But for it to be able to do its job, it needs to be able to extract some necessary information about the key. The least amount of information the OpenSSL core needs about a key is its raw public key, the length of its parts and the number of security bits as defined in SP800-57, and the maximum size that can be allocated for a signature.

We will keep this operation separate from the rest of the provider's code in the `cng_provider/keymgmt/` directory.

3.3.1 Key management implementation

To implement the minimal provider, we need to provide functions to create, copy and destroy key objects, be able to load a key from something given to us by `STORE` operation (we control what it is and will discuss it later), and be able to retrieve information from the provider.

The provider-side key object for us is going to be just a `NCRYPT_KEY_HANDLE`, which is how CNG represents keys.

This will cause an unfortunate collision as these handles cannot be duplicated directly, and OpenSSL expects a duplication function for key objects. To solve it, we need to extract all information about the key and about where it came from (which key storage provider) and use that in `NCryptOpenKey()` to open another copy of that key and gain its handle.

For key object manipulation and parameter exchange, we will use code very similar to what we used in our provider's core for context manipulation and parameter exchange, respectively. What is new is the need for a `OSSL_FUNC_KEYMGMT_LOAD` function and `OSSL_FUNC_KEYMGMT_HAS` function along with functions to export parts of the keys.

Our load function (`cng_keymgmt_load()`) will be actually very simple. We will construct our key object in our `STORE` operation and just pass it directly to our `KEYMGMT`. Here we will duplicate the key, pass it to the OpenSSL core and let the `STORE` manage the original.

For the `OSSL_FUNC_KEYMGMT_HAS`, we need to implement answers for questions about what parts the key object has. Since the function can return success even when OpenSSL core asks for domain parameters that do not make sense in the context of RSA keys, we always return `true`.

3.4 Store

This operation is easier to implement. We need to be able to open a store and close it, set parameters for search and have functions to access the store. Information about the store will be carried in the store-specific context as in previous cases.

3.4.1 Store context

Our store context, called `T_CNG_STORE_CTX`, will have to hold information about the current place in the enumeration of keys and of certificates, information whether we have reached the end of enumerated items, and for ease of use, we will hold additional information like the current key and certificate. We will also want to store the parameter query sent to us by the OpenSSL core in case we implement multiple algorithms. There is also the option to store the expected parameter type information that the OpenSSL core might sometimes give us.

3.4.2 Opening the store

When opening the provider's store, we actually want to open the NCrypt key storage provider of CNG and the certificate store to be able to enumerate both keys and certificates.

Because during development, we weren't able to receive all private keys and associate them with certificates with `NCryptEnumKeys()`, we resorted to using `CertEnumCertificatesInStore()` to both enumerate the certificates and to enumerate their associated keys by enumerating the certificates and extracting the keys directly.

To prepare for the OpenSSL core asking for keys and certificates, we will try to load one of each and set the appropriate EOF flags if we don't find any.

3.4.3 Loading from the store

We need to implement the loading part of the `STORE`. When the OpenSSL core asks for a type of object (in our case, a certificate or a key), we need to return an appropriate object back. That is done through the callback functions that are passed as arguments.

For certificates, we do not need to do much as we can simply return the already DER-encoded certificates returned from the CNG certificate store and prepare by pre-loading another certificate.

For keys, we enumerate the certificates as previously and extract the private keys from them. Since CNG won't allow private key extraction, we save only the key's CNG handle and use it to gain access to the key later.

When the CNG store returns EOF, we simply set the EOF to the appropriate provider store part. Our provider will only return EOF when all possible objects from all stores were depleted.

3.4.4 Closing the store

Closing the store is as easy as using the CNG close function and freeing our own store context. Since this function can return errors, we should check for return values of the functions we call and respond accordingly.

3.4.5 Setting parameters to store context

Setting parameters is done in the same way it was done with our provider's core. We have a function that returns a table of parameters that our provider's store operation can support and another function in which the actual values are being set.

3.5 Signatures

For signatures, we need to implement methods to create, duplicate and free a signature context, a way to set the signature context parameters and a method to actually sign a digest. That is the bare minimum for having our private keys of client certificates stored in CNG.

3.5.1 Signature context

The manipulation of the context is going to be similar to the previous cases. What is going to differ is its content. We will store the provider's context, property query, our provider's key object, CNG handle of a hash, CNG algorithm identifier for the hash function, flags for the hash and optionally the PSS salt length.

3.5.2 Setting parameters to signature context

This task is done in the same way as in the previous operations. The OpenSSL documentation is misleading about what data types the parameters being passed can have; they have been addressed in an issue on GitHub by the author of this thesis[14].

3.5.3 Signing

To keep the operation's code separate, our implementation will be located in `cng_provider/signature/` directory.

In TLS 1.3, to prove that we own a private key, we only need to sign a digest with it. Because OpenSSL will give us data from which the digest needs to be created in the signing operation, we also need to create the digest before we can sign it.

As the signing process is split into three stages (the initialization, updating and finalization), we need to implement all three functions.

In the initialization stage, we convert the OpenSSL message digest name to one that CNG can process and create a CNG object with it. In the update function, we keep hashing the data we receive, and at the end, in the final function, we perform the actual signature.

Nevertheless, we need to implement a particular case when the caller does not give our final function a buffer to write the output into. OpenSSL documentation demands that we return the maximum size the given signature can be.

We could try to ask `NCrypt` for the maximum signature size, but it would require us to convert the `NCrypt` key to its `BCrypt` variant and then query that version of the key. It is, therefore, easier and more future-proof to do

[14]Issue number 20707.

it through the actual signing function. For that, we need to perform a dry run of our signature function (that means run `NCryptSignHash` with `NULL` as its `pbSignature` argument) and only return the size that will be needed for allocation.

In the final function, we shall also destroy and free all data that has been allocated during signature initialization.

3.5.4 Demonstration of the provider

OpenSSL provides a simple TLS client. We have modified it to be able to run on Windows by initializing the `winsock` API by `WSAStartup()`. Another modification was to add support for client certificates by using the OpenSSL SSL API, in particular, the `SSL_CTX_use_certificate()` and its accompanying `SSL_CTX_use_PrivateKey()`. Additionally, we have added support for providers into the code. All of these modifications can be found as an attachment to this thesis.

3.5.5 Results

Running the aforementioned code against a server that accepts client certificates with an appropriate client certificate in our CNG store gives us the desired response from the server, thus proving our provider's implementation to be a success. To receive a `HTTP 200` status code with the desired content, we must have enumerated certificates in the system store, choose the desired one, find a corresponding private key in the store and sign the digest with the given key with the appropriate signature algorithm given to us by OpenSSL.

3.5.6 Possible improvements

We have limited ourselves to just RSA keys and TLS 1.3. This scope can be broadened to include support for keys based on elliptic curves by appropriately distinguishing between these types in `KEYMGMT` operation and setting parameters passed to the OpenSSL core with the given keys. The OpenSSL manual page about `EVP_KEYMGMT-EC` is a very good start for that. Then the parameters must be passed in `SIGNATURE` operation to the corresponding CNG functions. A good reference for this is the CNG algorithm identifiers manual page[15].

To add more functionality, the appropriate parameters must be passed to the CNG signature functions through the OpenSSL APIs. However, CNG does not support all the algorithms; for example, the `SHA-224` (used in TLS 1.2) is unsupported, so the functionality will be limited to the scope of CNG. For most use cases, the capabilities of CNG should be enough, as it was developed with Windows in mind.

[15]https://learn.microsoft.com/en-us/windows/win32/seccng/cng-algorithm-identifiers

Another good way to improve the provider is to use OpenSSL's error handling. Right now the provider uses a `debug_printf` function that handles errors and debug information but the same can be achieved with a pure OpenSSL solution with better support from the library itself in regards to tracing and established workflows.

3.6 Practical usage

To be able to use this provider, there are a few steps that need to be taken first. It needs to be built, installed, loaded into the appropriate code and finally get executed. This provider is ready to be compiled together with OpenSSL, but that is out of the scope of this thesis; we will focus on dynamic loading of providers as it provides more flexibility.

3.6.1 Building and installing the CNG provider

This project has two dependencies, CNG and OpenSSL version 3.0.0 and higher. CNG is available in Windows Vista and higher versions of Microsoft's operating system. OpenSSL needs to be compiled and installed in a standard location. Our build process relies on the `C:`
`Program Files`
`OpenSSL`
directory as the root folder of OpenSSL. It expects the static version of OpenSSL.

Other than these dependencies, Microsoft Visual C++ build tools need to be installed, as well as `CMake`.

In the project's code root folder, a `CMakeLists.txt` file can be found, which is a configuration file for `CMake`. To build the provider, the commands from lines one and two in listing 3.3 should be used. The command in the third line installs the provider into a location where OpenSSL searches for providers by default. If this is not done, specifying the path to the provider will be necessary. This step requires administrator privileges.

The last step builds the example client application.

Listing 3.3: Building and installing of the provider

```
1 cmake -S . -B ./custom-build-directory
2 cmake --build ./custom-build-directory --target cng_provider
3 cmake --build ./custom-build-directory --target install
4 cmake --build ./custom-build-directory --target client
```

3.6.2 Loading the provider

Fortunately, OpenSSL allows dynamic loading of providers. That means that if we have a version of OpenSSL that supports them, we can load a provider into it without having to recompile OpenSSL.

Command line

This can either be done with the command line utilities that support providers with a special flag `-provider`. For example, the `s_client` command can be invoked as shown in listing 3.4.

Listing 3.4: Using provider from commandline

```
openssl s_client -provider cng_provider -provider default  -
    connect cng.ladislavmarko.cz:443 -cert cng:example-key
```

To find keys and certificates in the CNG store, we need to tell OpenSSL that it should look for them there. To do this, we need to add a special *scheme* to the parameter specifying the certificate location. For us, that is the `cng://` scheme (we have registered it to the OpenSSL core in our `STORE` operation in `algorithm_names` field of `OSSL_ALGORITHM` array). After the scheme, a system store recognizable by CNG should be specified. We have implemented the `MY`, `CA` and `ROOT` options mirroring the naming convention of CNG.

All that implies that a valid URI will be, for example, `cng://MY`, which will prompt OpenSSL to use our provider to access the `MY` system store.

In code

Loading a provider into an application using OpenSSL is as easy as with the command line option. We can simply use the `OSSL_PROVIDER_load()` function to add our provider to the OpenSSL library context, and from that time forward, OpenSSL will use our provider when appropriate API calls are issued until our provider is unloaded with `OSSL_PROVIDER_unload()`. A special case is the `NULL` context, as it is the default SSL context. An example usage can be found in the code from OpenSSL documentation in listing 3.5.

Listing 3.5: Using provider from code - OpenSSL documentation [28]

```c
#include <stdio.h>
#include <stdlib.h>
#include <openssl/provider.h>

int main(void)
{
  OSSL_PROVIDER *legacy;
  OSSL_PROVIDER *deflt;

  /* Load Multiple providers into the default library context */
  legacy = OSSL_PROVIDER_load(NULL, "legacy");
  if (legacy == NULL) {
    printf("Failed␣to␣load␣Legacy␣provider\n");
    exit(EXIT_FAILURE);
  }
  deflt = OSSL_PROVIDER_load(NULL, "default");
  if (deflt == NULL) {
    printf("Failed␣to␣load␣Default␣provider\n");
    OSSL_PROVIDER_unload(legacy);
    exit(EXIT_FAILURE);
  }

  /* Rest of application */

  OSSL_PROVIDER_unload(legacy);
  OSSL_PROVIDER_unload(deflt);
  exit(EXIT_SUCCESS);
}
```

Conclusion

The goal of this thesis was to study the concept of providers introduced in OpenSSL 3.0.0 and summarize the learned information. Then try to design and create a provider that would be able to offload certificate work to CNG in a TLS 1.3 communication.

The first chapter examined OpenSSL parts and how they interact and discussed the history of the project.

Then we investigated the purpose of providers, what are their capabilities and how they interact. We explored their infrastructure in great detail — from new data types, through operations, to their built-in and publicly available versions.

In the third chapter we have gone through the process of examining the actual possibilities of implementation and considered how to implement our case. First, the basic functionality of a provider and, later, the concrete operations. Ultimately, we have demonstrated that the implementation was correct and functioning accordingly.

Overall the implementation in its limited scope was a success. It allowed for a client certificate to be loaded from the CNG store and the said certificate to be used to create a TLS 1.3 connection. The provider's structure allowed for a straightforward extension of available operations and additional functions.

The greatest obstacle in this thesis was the insufficient documentation of OpenSSL. It was fragmented, without examples and not united in relevant sections. As this project was bleeding-edge, it is understandable (even though OpenSSL does not have a good record of coherent documentation). As a side-effect, this thesis has led to improvements in the OpenSSL documentation regarding providers and has helped with the documentation of CNG as well.

The main takeaway from this thesis should be the condensed information about the OpenSSL providers and how to implement one.

The created provider is now available on GitHub as an open-source project under the name *cng-openssl-provider* [29].

Bibliography

[1] Roser, M.; Ritchie, H.; et al. WORLD INTERNET USAGE AND POPULATION STATISTICS 2023 Year Estimates. (accessed: 08.04.2023). Available from: `https://www.internetworldstats.com/stats.htm`

[2] OpenSSL Project. OpenSSL official website. (accessed: 24.02.2023). Available from: `https://www.openssl.org`

[3] OpenSSL project and Young, E. OpenSSL and SSLeay licenses. (accessed: 04.03.2023). Available from: `https://www.openssl.org/source/license-openssl-ssleay.txt`

[4] Caswell, M. The Holy Hand Grenade of Antioch. (accessed: 04.03.2023). Available from: `https://www.openssl.org/blog/blog/2018/11/28/version/`

[5] Apache Software Foundation. Apache License Version 2.0. (accessed: 04.03.2023). Available from: `https://www.apache.org/licenses/LICENSE-2.0.txt`

[6] OpenBSD Project. LibreSSL official website. (accessed: 24.02.2023). Available from: `https://www.libressl.org`

[7] Akamai; Microsoft. QuicTLS repository. (accessed: 16.03.2023). Available from: `https://github.com/quictls/openssl`

[8] Google. BoringSSL repository. (accessed: 24.02.2023). Available from: `https://boringssl.googlesource.com/boringssl`

[9] The OpenBSD project. LibreSSL goals. (accessed: 16.03.2023). Available from: `http://www.libressl.org/goals.html`

[10] Koci, M. OpenSSL Presentation at ICMC22 Conference. (accessed: 16.03.2023). Available from: `https://www.openssl.org/blog/blog/2022/09/21/OpenSSL-presentation-at-ICMC/`

[11] de Raath, T. Re: new OpenSSL flaws. (accessed: 16.03.2023). Available from: `https://marc.info/?l=openbsd-tech&m=140199655122732`

[12] Internet Systems Consortium. ISC License. (accessed: 16.03.2023). Available from: `https://www.isc.org/licenses/`

[13] Caswell, M. NIST FIPS 140-2 certification for OpenSSL FIPS provider announcment. (accessed: 16.03.2023). Available from: `https://www.openssl.org/blog/blog/2022/08/24/FIPS-validation-certificate-issued/`

[14] National Institute of Standards and Technology. NIST FIPS 140-2 certification for OpenSSL FIPS provider. (accessed: 04.03.2023). Available from: `https://csrc.nist.gov/projects/cryptographic-module-validation-program/certificate/4282`

[15] OpenSSL Management Committee. OpenSSL FIPS 140-3 provider plans. (accessed: 16.03.2023). Available from: `https://www.openssl.org/blog/blog/2022/09/30/fips-140-3/`

[16] Caswell, M. OpenSSL FIPS 140-3 provider release. (accessed: 16.03.2023). Available from: `https://www.openssl.org/blog/blog/2022/09/30/fips-140-3/`

[17] Patric; The Gitter Badger. YAOG. (accessed: 9.5.2023). Available from: `https://github.com/patrickpr/YAOG`

[18] Pešek, J. *Podpora CryptoAPI Next Generation v OpenSSL*. Czech Technical University in Prague, Faculty of Information Technology, 2020.

[19] OpenSSL Management Committee. RIPEMD160 and the Legacy Provider. (accessed: 16.3.2023). Available from: `https://www.openssl.org/blog/blog/2022/10/18/rmd160-and-the-legacy-provider/`

[20] OpenSSL Project. OSSL_PARAM documentation. (accessed: 11.4.2023). Available from: `https://www.openssl.org/docs/man3.1/man3/OSSL_PARAM.html`

[21] Internet Engineering Task Force. PKCS #1: RSA Cryptography Specifications Version 2.2. (accessed: 17.4.2023). Available from: `https://www.rfc-editor.org/rfc/rfc8017#appendix-C`

[22] Internet Engineering Task Force. MD2 to Historic Status. (accessed: 19.4.2023). Available from: `https://www.rfc-editor.org/rfc/rfc6149`

[23] Electronic Frontier Foundation and Loukides, M.; Gilmore, J. *Cracking DES: Secrets of Encryption Research, Wiretap Politics and Chip Design*. USA: O'Reilly & Associates, Inc., 1998, ISBN 1565925203.

[24] Internet Engineering Task Force. Prohibiting RC4 Cipher Suites. (accessed: 19.4.2023). Available from: https://www.rfc-editor.org/rfc/rfc7465

[25] Network Working Group. PKCS #5: Password-Based Cryptography Specification Version 2.0. (accessed: 19.4.2023). Available from: https://www.rfc-editor.org/rfc/rfc2898

[26] Sorce, S. PKCS#11 provider for OpenSSL. (accessed: 17.4.2023). Available from: https://github.com/latchset/pkcs11-provider

[27] Gotthard, P. Provider for integration of TPM 2.0 to OpenSSL 3.x. (accesssed: 17.4.2023). Available from: https://github.com/tpm2-software/tpm2-openssl

[28] Pierre, M. S. README-PROVIDERS.md. (accessed: 9.5.2023). Available from: https://github.com/openssl/openssl/blob/3868807d2fe5a72aa897ce5f7f7ba7e9cc3c09cb/README-PROVIDERS.md

[29] Marko, L. Openssl provider using Cryptography API: Next Generation. (accessed: 10.5.2023). Available from: https://github.com/Lipovlan/cng-openssl-provider/

Appendix **A**

Acronyms

ABI application binary interface.

API application programming interface.

BIO basic input output.

CNG Cryptograpy API: Next Generation.

DER Distinguished Encoding Rules.

DSA Digital Signature Algorithm.

DTLS Datagram Transport Layer Security.

EOF end of file.

FIPS Federal Information Processing Standards.

HTTP(S) Hyper Text Transfer Protocol (Secure).

MAC message authentication code.

PEM Privacy-Enhanced Mail.

PSS probabilistic signature scheme.

RSA Rivest–Shamir–Adleman.

S/MIME Secure/Multipurpose Internet Mail Extensions.

SSL Secure Sockets Layer.

TCP	Transmission Control Protocol.
TLS	Transport Layer Security.
URI	Uniform Resource Indentifier.

Appendix B

Contents of enclosed media

```
readme.txt ....................the file with media contents description
bin................................the directory with compiled code
  provider_x86.zip........................provider in 32-bit version
  provider_x64.zip........................provider in 64-bit version
src ....................................the directory of source codes
  openssl ....................................OpenSSL source code
  implementation ..........................implementation sources
    client ................modified OpenSSL TLS 1.3 client example
    cng_provider..........................provider implementation
  original_client ..................OpenSSL TLS 1.3 client example
  thesis .............the directory of LaTeX source codes of the thesis
text ........................................the thesis text directory
  thesis.pdf.........................the thesis text in PDF format
```