Disclaimer

Statement for open documents

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 780108. Any dissemination of results here presented reflects only the consortium view.

D6.1 Functional Encryption Toolset API Design

Document Identification			
Status	Final	**Due Date**	31/12/2018
Version	1.0	**Submission Date**	17/12/2018

Related WP	WP6	**Document Reference**	D6.1
Related Deliverable(s)	D4.1, D7.1	**Dissemination Level(*)**	PU
Lead Participant	XLAB	**Lead Author**	Miha Stopar (XLAB)
Contributors	ATOS, UEDIN	**Reviewers**	Hendrik Waldner (UEDIN) Norman Scaife (WALLIX) Henri Binsztok (WALLIX)

Keywords:
Functional Encryption, Implementation

This document is issued within the frame and for the purpose of the FENTEC project. This project has received funding from the European Union's Horizon2020 under Grant Agreement No. 780108. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.

(*) Dissemination level.-PU: Public, fully open, e.g. web; CO: Confidential, restricted under conditions set out in Model Grant Agreement; CI: Classified, Int = Internal Working Document, information as referred to in Commission Decision 2001/844/EC.

Document Information

List of Contributors	
Name	Partner
Miha Stopar	XLAB
Tilen Marc	XLAB
Manca Bizjak	XLAB
Jan Hartman	XLAB

Document History			
Version	Date	Change editors	Changes
0.1	13/11/2018	Miha Stopar (XLAB)	Skeleton
0.2	15/11/2018	Miha Stopar (XLAB)	Organization
0.3	20/11/2018	Miha Stopar (XLAB)	Obstacles
0.4	24/11/2018	Manca Bizjak (XLAB)	Best practices
0.5	26/11/2018	Jan Hartman (XLAB)	CiFEr
0.6	3/12/2018	Tilen Marc (XLAB)	Obstacles, benchmarks
0.7	6/12/2018	Miha Stopar (XLAB)	Wrap-up sections
0.8	14/12/2018	Miha Stopar (XLAB)	Inclusion of revision comments and minor corrections
1.0	17/12/2018	Miha Stopar (XLAB)	Submission to EC

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable Leader	Miha Stopar (XLAB)	14/12/2018
Technical Manager	Michel Abdalla (ENS)	14/12/2018
Quality Manager	Diego Esteban (ATOS)	14/12/2018
Project Coordinator	Francisco Gala (ATOS)	14/12/2018

Table of Contents

List of Figures

List of Acronyms

Acronym	Description
ABE	Attribute Based Encryption
API	Application Programming Interface
CP-ABE	Ciphertext Policy Attribute-Based Encryption
DDH	Decisional Diffie-Hellman
DMCFE	Decentralized Multi-Client Functional Encryption
FE	Functional Encryption
HD	High Definition
IND-CPA	INDistinguishability under Chosen Plaintext Attack
IoT	Internet of Things
KP-ABE	Key Policy Attribute-Based Encryption
LWE	Learning with Errors
RLWE	Ring-Learning With Errors
MSP	Monotone Span Program
LSSS	Linear Secret Sharing Scheme

Executive Summary

In this deliverable D6.1 “Functional Encryption Toolset API Design”, we discuss the software implementation of functional encryption schemes. The implementation has been guided by WP7 requirements analysis and WP4 specifications about which schemes are to be used to meet the use case requirements. The output of WP6 at the end of the first project year are two functional encryption libraries which are to be used by WP7 to build three different functional encryption based applications.

1 Introduction

Functional encryption is a generalization of public-key encryption, which allows to delegate to third parties the computation of certain functions of the encrypted data. This can be achieved by generating specific secret keys for these functions. A functional encryption scheme consists of a set of four algorithms. The setup algorithm takes as input a security parameter and generates a public key together with a master secret key. The key generation algorithm takes as input an index for the particular function f and generates a secret key sk for f. To encrypt a message x, the encryption algorithm on input x and the public key to obtain a ciphertext is to be run. Then, given the encryption of a message x, the holder of the secret key sk for the function f is able to compute the value of f(x) using the decryption algorithm but nothing else about the encrypted data.

The main objective of Work Package 6 is to implement state-of-the-art functional encryption library and thus enable functional encryption schemes to be easily integrated in real-world applications. As a proof-of-concept and for demonstration purposes the library will be used in three FENTEC use cases that require specific functionalities which can be provided by functional encryption.

This deliverable presents the implementation of the functional encryption library in two programming languages - Go and C.

1.1 Purpose of the document

The goal of this deliverable is to present the implementation of the functional encryption library provided by FENTEC in the first year of the project. This includes presentation of the internal blocks of the library, code organization, development best practices, licensing, and how to use the API. Furthermore, the performance of different schemes is evaluated and the differences are explained. Note that for most schemes different underlying cryptographic primitives can be used (for example pairings or lattices) and the performance can vary significantly. Also, different security parameters can be configured for each of the schemes which has a considerable effect on security and performance of the scheme. Configuration is especially complex when schemes use lattices as the underlying cryptographic primitive.

Finally, to demonstrate how the FENTEC library can be used we present a small project demonstrating how to apply a machine learning classification algorithm on the encrypted data.

1.2 Structure of the document

This deliverable is structured as follows. Section 2 discusses the high-level organization of the library and decisions taken to meet the use case requirements. Section 3 discusses the problems experienced when implementing functional encryption schemes. These are mostly due to the lack of stable and fully-fledged libraries providing mathematical machinery for lattices and pairings. Section 4 offers an initial report on the performance of the two libraries. More detailed analysis will be provided in future instances of this deliverable. Section 5 presents good software development practices that were followed when implementing functional encryption library. Section 6 describes a machine learning project demonstrating how machine learning classification algorithms can be applied on the encrypted data. The deliverable concludes with a summary and an outlook in Section 7.

2 Organization of the library

In this section we present the organization and main building blocks of the FENTEC library, more accurately of the two FENTEC libraries.

2.1 Requirements elicitation

User requirements analysis showed that two use cases (Kudelski, ATOS) need a library in the C language and one use case (WALLIX) needs a library in Go. Furthermore, the Kudelski and WALLIX use cases require inner-product encryption schemes and the ATOS use case requires an attribute-based encryption scheme. Finally, the Kudelski and ATOS use cases do not have any particular licensing requirements, while WALLIX requires a permissive licence to enable integration of the FENTEC library in the open source product Awless which is licensed under Apache License 2.0.

The choice of the programming languages to be supported in FENTEC was thus clear - to meet the requirements the libraries need to be provided in Go and C. In the first year of the project two libraries have been implemented and open-sourced, both are available on Github: GoFE is the Go version [1] and CiFEr is the C version [2].

GoFE has been licensed under the Apache License Version 2.0 to meet the WALLIX requirement (WA.08). It is written entirely in native Golang and mostly uses only the code provided by the Golang language itself. The only external library used is emmy - a library for zero-knowledge proofs implemented by XLAB [3] which is used for prime generation related functionality.

As no requirements regarding the license of the C library have been elicited, CiFEr has been released under GNU LGPL v3 and GNU GPL v2. This license was chosen because the most commonly used library for multiple precision arithmetic in the C world is GMP [4] which is licensed under this dual license: GNU LGPL v3 and GNU GPL v2.

In the first round of requirements gathering it was not entirely clear which schemes will be used by the use cases. What was known was that the WALLIX and Kudelski use cases would need some form of inner-product functional encryption schemes and the ATOS use case would need some form of attribute-based encryption (ABE). Furthermore, the WALLIX web analytics use case requires entities to participate in privacy-preserving polls which means a setting where entities neither trust a central authority nor trust each other. This can be provided by a decentralized multi-client functional encryption scheme such as [24].

Thus, the first schemes to be implemented in GoFE and CiFEr were chosen after examining the state-of-the-art of practical (efficient) inner-product encryption and ABE schemes.

Note that the goal was to choose efficient schemes. There exist other schemes for more general functionalities but at the expense of efficiency. The research on functional encryption can be divided into two categories:

- Concrete, efficient schemes for restricted functionalities (like inner-products).
- General functionality schemes based on indistinguishability obfuscation or multilinear maps.

The schemes of the second type rely on non-standard and ill-understood assumptions. Also, these schemes are in many cases extremely time-consuming. On the other hand the aim of FENTEC is to design and implement efficient schemes of restricted functionality but still of practical interest.

The first schemes implemented in the FENTEC library were inner-product schemes. These are schemes where decrypting an encrypted vector x with a key for a vector y will reveal only the inner product of x and y and nothing else. Despite the limitations of this functionality, it is still useful in many contexts like descriptive statistics.

Later, the library was extended with a scheme for quadratic polynomials which is much more powerful. This scheme is also used to demonstrate machine learning classification on the encrypted data.

Finally, an ABE scheme from a paper that introducted key-policy attribute-based encryption (KP-ABE) in 2006 was implemented [30]. Later, a more recent and highly efficient scheme for KP-ABE and ciphertext-policy attribute-based encryption (CP-ABE) will be added [18].

Now, we list the schemes that are currently supported by the FENTEC library (FAME is being developed):

- **Simple Functional Encryption Schemes for Inner Products [16].** The first efficient scheme for inner-products, albeit proved only for selective security. It can be instantiated either from DDH or LWE.
- **Fully Secure Functional Encryption for Inner Products, from Standard Assumptions [19].** Inner-product encryption scheme which is fully (adaptively) secure. In addition to DDH and LWE an instantiation from Paillier's composite residuosity assumption is possible which does not require computation of a discrete logarithm in the decryption phase and is thus highly efficient.
- **Multi-Input Functional Encryption for Inner Products: Function-Hiding Realizations and Constructions without Pairings [17].** Multi-input functional encryption scheme for inner-products. The scheme can be instantiated on MDDH, LWE and Paillier assumptions.
- **Decentralized Multi-Client Functional Encryption for Inner Product [24].** This scheme allows various senders to non-interactively generate ciphertexts which support inner-product evaluation, with functional decryption keys that can also be generated non-interactively.
- **Reading in the Dark: Classifying Encrypted Digits with Functional Encryption [38].** A scheme for quadratic multi-variate polynomials which enables efficient computation of quadratic polynomials on encrypted vectors.
- **Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data [30].** First scheme for KP-ABE which enables fine-grained sharing of encrypted data.
- **FAME: Fast Attribute-based Message Encryption [18].** The first fully secure KP-ABE and CP-ABE schemes based on a standard assumption on Type III pairing groups which do not put any restriction on policy type or attributes.

2.2 Selectively and adaptively secure schemes

Schemes can be proven selectively or adaptively secure. Selective security guarantees security only for messages that are fixed ahead of time (even before the adversary interacts with the system). Adaptive or full security guarantees security also for messages that are adaptively chosen at any point in time.

In selectively secure schemes (selective security under chosen-plaintext attacks denoted as s-IND-CPA), the adversary must begin by declaring the challenge vectors before the keys are generated. That means the adversary declares the challenge messages x_0 and x_1 before seeing the public keys.

In adaptively secure (IND-CPA) schemes the adversary can declare messages after seeing the public keys. Selective security is a weaker notion and whenever available, a fully secure scheme will be chosen to be implemented in the FENTEC library. Among the currently existing schemes only Abdalla *et al.* [17] is selectively secure. It was implemented because it meant a crucial step towards practical inner-product schemes and it contains building blocks used in other schemes as well.

2.3 Internal structure

Both the Go and C library are structured as follows:

- **Data package** Functionality for vectors and matrices. This includes generation of random vectors and matrices according to different distributions (uniform, Gaussian), component-wise operations, ordinary operations (multiplications), transpositions, inversion, and some more (FE) specific functions (like $x^T * M * y$ where x, y vectors and M matrix).
- **Internal package** Functions used internally in schemes for key generation and discrete logarithm computation.
- **Innerprod package** Schemes for inner-products (linear polynomials).
- **Quadratic package** Schemes for quadratic polynomials.
- **Sample package** Samplers for sampling random values from different probability distributions.
- **ABE package** Attribute-based encryption schemes.

2.4 API

All schemes offer methods for:

- **generation of (secret and public) master keys**
- **derivation of functional decryption keys**
- **encryption**
- **decryption.**

2.4.1 GoFE

All GoFE schemes are implemented as Go structs with similar APIs. So the first thing we need to do is to create a scheme instance by instantiating the appropriate struct. For this step, we need to pass in some configuration, e.g. values of parameters for the selected scheme.

Let us say we selected a simple.DDH scheme. We create a new scheme instance with:

```
scheme, _ := simple.NewDDH(5, 128, big.NewInt(1000))
```

The first argument is length of input vectors x and y, the second argument is bit length of prime modulus p (because this particular scheme operates in the Z_p group), and the last argument represents the upper bound for elements of input vectors.

Note that the configuration parameters for different FE schemes vary as the underlying cryptographic primitives require different and not-related parameters (for example configuration for pairing-based or lattice-based schemes).

All GoFE schemes use vectors (or matrices) of big integer (*big.Int) which are implemented in the data package. A Vector is a wrapper around a []*big.Int slice, while a Matrix wraps a slice of Vectors. They can be instantiated as follows:

```
x := []*big.Int{big.NewInt(0), big.NewInt(1), big.NewInt(2)}
xVec := data.NewVector(x)

vecs := make([]data.Vector, 3) // a slice of 3 vectors
vecs[0] := []*big.Int{big.NewInt(0), big.NewInt(1), big.NewInt(2)}
vecs[1] := []*big.Int{big.NewInt(2), big.NewInt(1), big.NewInt(0)}
vecs[2] := []*big.Int{big.NewInt(1), big.NewInt(1), big.NewInt(1)}
xMat := data.NewMatrix(vecs)
```

To generate random *big.Int values from different probability distributions, several implementations of random samplers are provided. Random vectors and matrices can be quickly constructed by:

```
s := sample.NewUniform(big.NewInt(100)) // will sample uniformly from [0,100)
x, _ := data.NewRandomVector(5, s) // creates a random vector with 5 elements
X, _ := data.NewRandomMatrix(2, 3, s) // creates a random 2x3 matrix
```

The example below demonstrates how to use single input scheme instances. Although the example shows how to use the DDH from package simple, the usage is similar for all single input schemes, regardless of their security properties (s-IND-CPA or IND-CPA) and instantiation (DDH or LWE).

Three DDH structs are instantiated to simulate the real-world scenarios where each of the three entities involved in FE are on separate machines.

```
// Instantiation of a trusted entity that
// will generate master keys and FE key
l := 2 // length of input vectors
bound := big.NewInt(10) // upper bound for input vector coordinates
modulusLength := 128 // bit length of prime modulus p
trustedEnt, _ := simple.NewDDH(l, modulusLength, bound)
msk, mpk, _ := trustedEnt.GenerateMasterKeys()

y := data.NewVector([]*big.Int{big.NewInt(1), big.NewInt(2)})
feKey, _ := trustedEnt.DeriveKey(msk, y)

// Simulate instantiation of encryptor
// Encryptor wants to hide x and should be given
// master public key by the trusted entity
enc := simple.NewDDHFromParams(trustedEnt.Params)
x := data.NewVector([]*big.Int{big.NewInt(3), big.NewInt(4)})
cipher, _ := enc.Encrypt(x, mpk)

// Simulate instantiation of decryptor that decrypts the cipher
// generated by encryptor.
```

```
dec := simple.NewDDHFromParams(trustedEnt.Params)
// decrypt to obtain the result: inner prod of x and y
// we expect xy to be 11 (e.g. <[1,2],[3,4]>)
xy, _ := dec.Decrypt(cipher, feKey, y)
```

More examples can be found on Github [1].

2.4.2 CiFEr

The CiFEr API is very similar to the GoFE API to make both libraries easily comparable and switching between both simple. The GoFE example of simple.DDH given above looks in CiFEr as follows:

```
size_t l = 3;
int modulus_len = 128;
int err = 1;

mpz_t bound, bound_neg, func_key, xy_check, xy;
mpz_inits(bound, bound_neg, xy_check, NULL);
mpz_set_ui(bound, 2);
mpz_pow_ui(bound, bound, 15);
mpz_neg(bound_neg, bound);

ddh s, encryptor, decryptor;
err = ddh_init(&s, l, modulus_len, bound);
munit_assert(err == 0);

vector msk, mpk, ciphertext, x, y;
vector_inits(l, &x, &y, NULL);
uniform_sample_range_vec(&x, bound_neg, bound);
uniform_sample_range_vec(&y, bound_neg, bound);
vector_dot(xy_check, &x, &y);

ddh_generate_master_keys(&msk, &mpk, &s);

err = ddh_derive_key(func_key, &s, &msk, &y);
munit_assert(err == 0);

ddh_copy(&encryptor, &s);
err = ddh_encrypt(&ciphertext, &encryptor, &x, &mpk);
munit_assert(err == 0);

ddh_copy(&decryptor, &s);
err = ddh_decrypt(xy, &decryptor, &ciphertext, func_key, &y);
munit_assert(err == 0);

munit_assert(mpz_cmp(xy, xy_check) == 0);

mpz_clears(bound, func_key, xy_check, xy, NULL);
```

```
vector_frees(&x, &y, &msk, &mpk, &ciphertext, NULL);

ddh_free(&s);
ddh_free(&encryptor);
ddh_free(&decryptor);
```

3 Implementation obstacles

Existing constructions of practical functional encryption schemes can be divided into three groups based on either:

- Modular arithmetic,
- Pairings, or
- Lattices.

Implementation of functional encryptions schemes based on modular arithmetic is relatively straightforward and we did not face any major obstacles when implementing these schemes. On the other hand the implementation of schemes based on pairings and lattices requires lower-level math artillery. Quite suprisingly the required math functionality is provided only by a handful of libraries. In this section we briefly present how this functionality (usually involving complex math operations) have been implemented in the FENTEC library.

3.1 Pairing schemes

Pairings were first used in 1991 to attack cryptosystems using supersingular elliptic curves. Ten years later, three seminal papers appeared where pairings were used to design novel (or improved) protocols: identity-based encryption [21], short signature scheme [22], and one round tripartite key exchange [31]. Since then pairings have enabled many new cryptographic protocols that had not previously been feasible.

A pairing is a function $e : G_1 \times G_2 \to G_T$ that maps two elements of an elliptic curve to an element of a finite field G_T. Function e is non-degenerate ($e(g_1, g_2) \neq 1$) and bilinear. For a detailed introduction to pairings, see e.g. [20, Ch. IX].

Numerous libraries for pairings are available but most lack at least some essential functionality or performance optimization. In what follows we list some of the existing libraries:

- **PBC [35].** While this is the first pairing-based crypto library (appeared in 2007), it is no longer maintained and lacks optimizations from recent years. It is written in C and published under Apache License 2.0.
- **RELIC [26].** A modern cryptographic meta-toolkit with emphasis on efficiency and flexibility. It can be used to build efficient and usable cryptographic toolkits tailored for specific security levels and algorithmic choices. Written in C, published under Apache-2.0 or LGPL-2.1.
- **Apache Milagro Cryptographic Library (AMCL) [5].** A self-contained (except for the requirement for an external entropy source for random number generation) cryptography library for pairings, providing many optimizations and improvements from recent years. It is written in C (it contains implementation for other languages too, but not as complete as for C) and published under Apache License 2.0.
- **BN256 [33].** Part of official Go crypto library providing Barreto-Naehrig pairings.
- **Cloudflare BN256 [6].** Go library providing Barreto-Naehrig pairings. Licensed under BSD 3.

AMCL was chosen as an underlying pairing library for CiFEr because it is portable (no assembly language), multi-lingual (besides C also Java, Javascript, Go and Swift, although these implementations lack the

completeness of the C library), small (less than 10 000 lines of code), optimized to fit into the smallest possible embedded footprint and particularly suitable for the Internet of Things. AMCL (the C version) only uses stack memory, and is thus natively multi-threaded. Also, we found AMCL to be easier to use compared to RELIC or PBC.

At first Cloudflare BN256 was chosen as the pairing library to be used for GoFE as it is an extended and optimized version of official BN256 implementation. However, we experienced some problems and switch to the official Go BN256 version. The problems we are mentioning are still under investigations and will be described in the next instance of this deliverable. If resolved we will eventually switched back to Cloudflare BN256 because of its speed.

Neither the Cloudflare BN256 nor the official BN256 provide hash functions for pairing groups. Indeed, properly implemented hash functions (for pairing groups) is a functionality rarely provided by pairing libraries (included in AMCL though).

An additional function most often not included in pairing libraries is encoding of messages into pairing groups. In inner-product and quadratic schemes a message is most often considered simply to be a vector of integers and thus does not need to be converted into some special format prior to encryption. In ABE schemes, the message is usually an arbitrary string. As ABE schemes (at least those implemented in FENTEC libraries) use pairings, the message needs to be encoded into one of the pairing groups.

In what follows we present how we addressed hashing and encoding issues. All extended functionality is being provided in a fork of the official BN256 library.

3.1.1 Hashing into G_1

Many pairing-based schemes involve hashing to either G_1 or G_2. In the security proofs of these schemes hash functions are modeled as random oracles. However, while random oracles to groups like Z_p can be easily constructed, it is not trivial to instantiate such functions in pairing groups. For Z_p groups, ordinary hash functions to fixed-length bit strings can be used. This approach is not directly applicable for pairing groups for numerous reasons. The most trivial encoding would be simply to hash a message into Z_r (r is order of a subgroup) and then simply multiply this value by a generator of the group G_1 or G_2. However, in many cases this completely breaks the security of the scheme.

For hashing into G_1 we implemented a classical construction based on the try-and-increment algorithm [23] which preserves random oracle proofs of security (it has the drawback of not running in constant time though). This function has already been implemented in AMCL (used by CiFEr). We implemented it in Go in order to have it available.

3.1.2 Hashing into G_2

Hashing into G_2 is much more complex compared to hashing into the G_1 group. To provide an efficient (hashing into G_2 is much slower compared to G_1 and is usually avoided in the construction of cryptographic schemes) hashing procedure we chose to implement the technique from [28] which is implemented also in AMCL.

3.1.3 Encoding into G_T

The ABE scheme provided by the FENTEC library requires encoding of messages into G_T which is a subgroup of a finite field. This is much easier compared to encoding messages into an elliptic curve group and can be done similarly as mapping messaging into a Z_p field. If a message is too large (depends on the field characteristic) to be encoded as a single element of G_T, the standard technique is to use a key encapsulation mechanism wherein a random element of G_T is ABE encrypted and hashed to get a session key. This key is then used to encrypt the message through a symmetric key scheme like AES.

3.2 Lattice schemes

The resistance of cryptographic protocols to post-quantum attacks is becoming ever more important as we get closer to the realization of quantum computers. Lattice-based cryptography is believed to be secure against quantum computers. Its cryptographic constructions are based on the presumed hardness of lattice problems (e.g. for example, the shortest vector problem). Currently the most used construction is the Regev cryptosystem [37] based on the Learning With Errors (LWE) problem. However, there are two main bottlenecks in LWE-based schemes. These are: sampling random values distributed according to the discrete Gaussian distribution and uniform distribution, and matrix modular multiplications.

Discrete Gaussian sampling is a problem of sampling values distributed according to Gaussian distribution but limited only to discrete values. This issue has been tackled by software implementations (see [32, 27]) as well as hardware implementations [25, 29]. However, even though some solutions are already implemented and available as open source libraries, they could not be used in the implementation of the FE schemes based on the LWE problem. The reason for this is that the standard deviation of the samples in FE is much bigger than in standard schemes based on the LWE problem. This implies that methods using precomputed tables or precomputed search trees are not feasible due to their sizes.

For this reason the FENTEC-provided implementation of a discrete Gaussian sampler is based on the algorithm from [27]. The sampler works using the following two part procedure. It starts by sampling a value from a discrete Gaussian distribution with some fixed (independent) standard deviation. Then using the uniform sampling from the obtained value it generates a candidate for the output which is accepted with certain probability. If it is not, then the procedure is repeated. As proven in [27], such a sampler is very efficient since the output is obtained by repeating the above procedure (on average taking less than 1.3 repeats). There are two hidden implementation challenges in this simple procedure. The first one is to create a discrete Gaussian sampler for some fixed standard deviation. In contrast to [27], where a special procedure that involves multiple sampling is suggested, we rather implemented an algorithm that samples from a Gaussian distribution by using a precomputed table for a small standard deviation. This choice was made since secure sampling of multiple random values can be computationally demanding and as few sampling as possible are preferred. The procedure works as follows: for a discrete Gaussian distribution with a small standard deviation we precompute the probabilities for each output with non-negligible probability and calculate their cumulative distribution. Then a random value from an interval $[0, 1]$ is sampled – in fact an approximation with an arbitrary precision float, statistically indistinguishable from the true value – which is then mapped back to the output. In this way only one uniformly distributed sampling is needed to obtain the result. The second challenge of the implementation was hidden in the fact that multiple calculations of probabilities involve evaluating whether a certain value is greater than a value of an exponential function of some other value. Since the exponential function is not a standard functionality of floats with arbitrary precision we implemented an algorithm that on one hand calculates the exponential function to arbitrary precision using Taylor polynomials and precomputed values, on the

other hand it does this only for the precision needed to provide a correct answer. All in all, this provides a discrete Gaussian sampler that efficiently samples with arbitrary precision from a discrete Gaussian distribution. We used GMP [4] for CiFEr and integrated the big.Int structure for GoFE as the source of arbitrary precision floats. The second bottleneck of FE schemes based on the LWE problem are matrix-vector and matrix-matrix multiplications. There are two reasons for these bottlenecks. On one hand the matrices generated in existing FE scheme are of sizes $n \times m$ where n is a parameter directly connected with the security of the scheme, while m is chosen depending on n and other metaparameters such as bounds on the input vector and dimensionality of the input vector space. For security reasons, in practice n must be chosen above 128 while m can be much greater leading to big matrices. Moreover, the entries of such matrices are integers whose size also depends on the metaparameters of the scheme for which the standard arithmetic operations are notably slower. We have implemented basic structures representing matrices and vectors of big numbers using standard libraries for dealing with big numbers: GMP [4] in CiFEr and integrated big.Int in GoFE. Moreover, basic algebraic operations such as multiplication, addition, finding inverses and determinants were implemented to simplify the implementation of the schemes. Nevertheless, computationally demanding operations as well as memory demands of storing public keys (matrices) could present a challenge for the schemes to become practical enough for the real-world use cases. However, note that for all lattice-based schemes CiFEr and GoFE provide more efficient instantiations, based for example on DDH or Paillier. There are ways of avoiding these problems but further research is needed. One way of avoiding costly operations and spacious public keys is by replacing LWE schemes with ring-LWE schemes [36]. The reason is that ring-LWE can be secure by using/sampling much smaller numbers as in LWE and replacing big matrices with (smaller) polynomials. This implies that matrix multiplications can be replaced by polynomial multiplications which are theoretically and practically much faster. We have implemented an experimental scheme using ring-LWE primitives and the improvement of the performance was considerable. Nevertheless, this replacement needs to be first proved theoretically to be secure before incorporating it into the use cases. Another possible approach for improvement in this direction is a development of a more efficient LWE-based schemes that supports only private key encryption. Lastly, hardware optimizations could also enable a speed up of the algorithms.

3.3 ABE schemes

ABE schemes provide a functionality where a client can access or not access the decryption data of a ciphertext based on a set of attributes that he or she possesses. FENTEC provided a KP-ABE scheme [30], while a recent and more efficient the FAME scheme [18] is being developed at the time of writing this deliverable. Many of the contemporary ABE schemes, including those chosen for the implementation, are based on similar cryptographic primitives: pairing groups and Linear Secret Sharing Scheme (LSSS) matrices. Thus, once the building blocks are prepared, new schemes can be added relatively quickly.

In particular, the FAME scheme requires complex operations such as hashing and encoding into the pairing groups which are described in the previous sections.

A part of every ABE scheme is a policy that defines which entity can decrypt the cipertext based on the attributes. A Monotone Span Program (MSP) is defined as a policy that accepts a subset of attributes as sufficient if a certain subset of chosen vectors spans a vector of ones. Hence to create a MSP policy one must carefully choose a set of vectors representing attributes in a way that they describe the desired rules of decryption. This set of vectors is also known as a Linear Secret Sharing Scheme (LSSS) matrix. On the other hand, expressing rules of decryption as a boolean expression is preferred for practical usage and interpretability. Hence we have implemented an algorithm that transforms a boolean expression into a MSP structure. We have chosen the Lewko-Waters Algorithm [34] for this task, due to its simplicity and

efficiency. The algorithm can transform an arbitrary boolean expression that does not include a "NOT" operation ($\neg$) into a set of vectors (a matrix) whose dimensions only depend on the number of "AND" operators ($\wedge$) and the number of variables in the expression.

4 Benchmarks

In the following section we focus on a practical evaluation of the implemented schemes, comparing the benefits and downsides of each one, and discussing their practicality for the use cases. For the latter we peformed various benchmarks that help to better understand the algorithms. All the benchmarks were run on a laptop with an Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz processor using the GoFE library to encrypt and decrypt random inputs for the schemes. The library CiFEr in fact gives better performance values within some small constant factor with similar benefits and downsides of the schemes.

4.1 Inner-product schemes

We start with the schemes allowing the inner-product functionality. Performance of the schemes depends on meta-parameters needed by particular use cases. Mainly, these parameters are:

- Security needed for the scheme: we will measure the security of the scheme by a parameter n, meaning that there is no known attack on the scheme running in significantly less than $O(2^n)$ operations.
- Dimensionality l of the input and inner-product vectors.
- Bounds on the entries of input and inner-product vectors: we will bound them with a general bound B, meaning that all the vectors in the schemes have entries with absolute value bellow B.

While the dimensionality and bound are straightforward parameters, comparing the security of different schemes is less trivial since the schemes depend on different cryptographic assumptions. For the purpose of this section we have made the following choices based on some general practices suggested in various academic research:

- In schemes based on the difficulty of computing a discrete logarithm (DDH assumption), we have considered security n as the bit length of a safe prime p which is the order of the underlying group that the scheme is based on.
- In schemes based on the LWE assumption we consider the security n as the dimensionality of the secret in the LWE problem.
- The Paillier type scheme is based on the DCR assumption which further depends on the assumption that factoring products of big prime numbers is hard. Since factoring seems to be a slightly easier problem than the discrete logarithm problem we considered security n as the difficulty of factoring a product of two safe primes with $4n$ bits.

We point out that the above choices were made only for the purpose of this section, while for a general use of the schemes these parameters can be specified at the initiation of the schemes.

In Table 1 we consider a basic use of all the implemented schemes for a commonly chosen security parameter $n = 128$ and a small message space of 20 dimensional vectors with a small bound on the entries $B = 10^4$. For this basic example the following can be observed:

- DDH-based schemes are the most appropriate choice in the case of small parameters.
- LWE-based schemes have a slow set up and key generation phase, reasonable performance of the encryption phase and the fastest decryption among the schemes. The reason for the slow-down

	Set up	KeyGen	KeyDerive	Encrypt	Decrypt
Simple DDH	7	< 1	< 1	< 1	95
Fully sec. DDH	6	< 1	< 1	< 1	118
Simple LWE	3498	10996	< 1	1104	< 1
Fully sec LWE	6286	52080	245	1101	5
Fully sec. Paillier	1586	219	< 1	8	18

Table 1: Time in milliseconds for parameters $n = 128$, $l = 20$, and $B = 10^4$.

comes from the fact that the set up includes a cryptographically secure sampling of big matrices, while the key generation involves a similar sampling and multiplication of matrices.

- The Paillier-based scheme involves a slow set up phase due to the generation of big safe prime numbers, a reasonably fast key generation and fast encryption and decryption calculations.

It seems from Table 1 that LWE and Paillier schemes suffer a major deficiency due to a slow set-up and key generation. This is not necessarily critical since for many application this phase is run only once compared to the encryption and decryption phases which are run repeatedly. A simple example of such a case is when a learned machine learning function is evaluated on an encrypted data from the same source, for example in the case of classifying encrypted images from a camera.

It is important to observe in Table 2 what happens when the bound on the vectors is increased. Due to the fact that the decryption process in the case of DDH-based schemes involves the computation of a discrete logarithm, the computation time rises exponentially with the number of bits of the bound, quickly making DDH schemes impractical. On the other hand, increasing the bound less drastically increases the set-up and key generation times in LWE and only slightly increases the encryption and decryption times. Moreover, the Paillier scheme is practically unaffected by such a change, as long as the bound is not extremely large.

	Set up	KeyGen	KeyDerive	Encrypt	Decrypt
Simple DDH	10	< 1	< 1	< 1	17647
Fully sec. DDH	6	< 1	< 1	< 1	17625
Simple LWE	7079	32449	1	1503	< 1
Fully sec LWE	11791	77108	277	1165	5
Fully sec. Paillier	1546	216	< 1	80	16

Table 2: Time in milliseconds for parameters $n = 128$, $l = 20$, and $B = 10^6$.

In Table 3 we can observe that when the dimensionality l of the vectors is increased the total computation times of the schemes in practice grow approximately linear with l.

	Set up	KeyGen	KeyDerive	Encrypt	Decrypt
Simple DDH	7	2	< 1	4	314
Fully sec. DDH	7	4	< 1	3	317
Simple LWE	8233	285619	2	3694	< 1
Fully sec LWE	6124	398716	1831	1160	5
Fully sec. Paillier	1559	1657	< 1	618	56

Table 3: Time in milliseconds for parameters $n = 128$, $l = 160$, and $B = 10^4$.

Finally, an increase in security is reflected in all the schemes slowing down as seen in Table 4. In the case of DDH-based schemes and the Paillier-based scheme, the computational times for encryption and decryption linearly depend on n, while the set-up phase depends on finding large safe primes which gives a greater slow-down. Nevertheless, a security parameter greater than $n = 256$ is not expected. On the other hand, LWE-based schemes quadratically depend on n due to the bigger matrices needed.

	Set up	KeyGen	KeyDerive	Encrypt	Decrypt
Simple DDH	56	< 1	< 1	1	108
Fully sec. DDH	50	1	< 1	1	109
Simple LWE	13499	37837	< 1	3472	< 1
Fully sec LWE	30211	204533	443	3978	10
Fully sec. Paillier	41175	1447	< 1	579	93

Table 4: Time in milliseconds for parameters $n = 256$, $l = 20$, and $B = 10^5$.

To sum up, the implemented schemes complement each other to support various practical use cases. DDH-based schemes are very efficient when the bounds on the inputs are small. In applications where this is not the case the Paillier-based scheme is a clear winner since after a relatively slow set up phase it offers a fast encryption and decryption practically independently of the bounds on the inputs. If a quantum security is needed, then LWE schemes can be used, having a slow set-up but reasonable encryption and decryption times even for inputs with greater bounds.

4.2 ABE

The ABE scheme implemented in the FENTEC library is based on a fixed elliptic curve group allowing bilinear pairing hence the only parameters affecting the performance are the number of attributes and the size of the decryption policy. For this reason we have tested the ABE scheme with increasing parameter l denoting the number of attributes in the scheme as well as the number of attributes used in a boolean expression defining the decryption policy. Note that here l denotes the number of attributes needed for expressing the decryption policy as well as the number of all attributes in the whole universe. If the number of all attributes is much bigger only the key generation phase is slower.

	Set up	KeysGen	Policy conversion	Encrypt	KeysDerive	Decrypt
$l = 10$	< 1	7	< 1	7	5	19
$l = 100$	< 1	61	40	61	55	212
$l = 1000$	< 1	589	6651	601	586	2059

Table 5: Time in milliseconds for increasing number of attributes

As one can observe in Table 5 the implemented scheme is very efficient with computational times growing only linear in the number of attributes. The bottleneck of the scheme is the decryption process caused by pairing operations and the policy conversion changing a boolean expression into a MSP structure later used in the scheme. Note that both these operations are demanding only if the boolean formula expressing the policy is large, which in most of the applications is not the case since there would be a large variety of attributes but a rather simple policy for decryption. Hence for practical use this scheme is very efficient.

4.3 Quadratic scheme

Similarly as for the ABE scheme, the quadratic scheme is based on a fixed elliptic curve group. It supports encrypting vectors x and y and distributing keys allowing decryption of any quadratic function of encrypted data of the form $x^T Ay$ for an arbitrary matrix A. Computational times depend on the parameters of x, y, A, i.e. on dimensionalitly l of the vectors and on the bound B of the entries of the latter. In Table 6 we present some tests for various parameters.

	Set up	KeysGen	KeyDerive	Encrypt	Decrypt
$l = 10, B = 100$	< 1	< 1	1	31	897
$l = 10, B = 1000$	< 1	< 1	1	31	16909
$l = 50, B = 100$	< 1	< 1	1	153	12360

Table 6: Time in milliseconds for various parameters

While on one hand the functionality of arbitrary quadratic functions offered by the scheme covers a wide range of possible applications, it is also a downside of the scheme. In particular, the computational bottleneck of the scheme is the decryption process which involves computing the discrete logarithm in an elliptic curve. Thus the computation time heavily depends on the size of the output. For example, already in the case $l = 10, B = 100$ the output of a quadratic function can be as big as $l^2B^3 = 10^8$, as big as 10^11 in the case $l = 10, B = 1000$, or $2.5 \cdot 10^9$ in the case $l = 50, B = 100$. Thus the full functionality of the scheme is possible only in the case of small parameters. Nevertheless, the scheme can still be used for many specific quadratic functions for which the maximal value of the output can be bounded. For example, if the scheme is used only for the inner-product functionality then its performance is comparable to performance of the DDH-based schemes presented in the tables above. Another example is presented in Section 6 evaluating the machine learning function.

5 Best practices

Functional encryption schemes in FENTEC are delivered in a form of software libraries in a variety of programming languages. Implementation of state-of-the-art cryptographic schemes is a challenge in itself, requiring collaboration of mathematicians/cryptographers and software engineers. However, to ensure that FENTEC libraries not only provide the envisioned functionality and work as expected, but are also *pleasant to work with* and *pleasant to work on*, we adhere to several software engineering practices that are considered good. Such practices have repeatedly seen success in the software engineering communities, either in the design or in implementation phases, or in the process of software development itself.

In the following sections, we present some of the good software development practices we follow in FENTEC. Section 5.1 focuses on the general approach to developing FENTEC software libraries. In contrast, Section 5.2 presents the good practices from a more technical perspective, as it takes into account specifics of programming languages in which FENTEC libraries are implemented.

5.1 Software development practices

5.1.1 Code review

To encourage learning within software development teams and to improve the overall quality of the libraries, every source code contribution to any of the FENTEC libraries is reviewed by at least one other contributor to the given library before a contribution is accepted to the codebase. Code review is an integral part of the software development process in all serious (closed- or open-source) software projects. Through code review, it is possible to catch all sorts of (potential) problems in the code, from minor ones (typos, inconsistencies, poor coding style) to more serious ones. In FENTEC, we encourage evolving the codebases of the libraries through small contributions, as they are easy to thoroughly review. Every contibution is subject to at least one cycle of code review. Proposed changes are reviewed, potential issues discussed, certain decisions clarified. As a result, the code under review is typically updated according to reviewer's comments, resulting in improved code. To support this process, we use GitHub (for already open-sourced libraries) or FENTEC GitLab (prior to the first open-source release), platforms that already integrate features to support the code review process.

5.1.2 Identifying with the user

For software libraries the aspect of user-friendliness is especially important, and may even be decisive when picking a specific software library among the alternatives. However, it does not come out-of-the box, and there is no universal recipe or guideline for achieving it. It is often just the opposite - user-friendliness requires careful design and implementation of a library. Perhaps the most crucial means for achieving it is the ability of developers to think like a user of the library. In code reviews certain implementation decisions are challenged with respect to how they affect ease of use. Further, we make little to no assumptions about the user's programming skills, and always provide clear setup instructions, usage documentation, and examples. When a user lands on a FENTEC library repository, he quickly learns what the library is about, who it is built for, what it is capable of, when and how to use it - aspects which are all too often overlooked.

5.1.3 Clean, readable code

While user satisfaction with the libraries is important, it is equally important to think about the aspects that attract external contributions to open-source projects. Once the FENTEC project concludes, it will likely be the users that will report the issues, deliver further improvements, bugfixes, or even new features, to FENTEC libraries. To encourage users of the libraries to submit contributions, it is important to develop and maintain clean, readable and understandable codebases, and elaborate on how certain pieces of code work (rather than just how to use them). Code reviews in FENTEC put a special focus on clean, intuitive organization of the code, and documentation in code comments. Thus, all FENTEC libraries strive towards readable code even for internal functionalities, e.g. the code users never directly interact with, but upon which they might stumble in case they are curious about how certain functionality is implemented.

5.1.4 (Automated) testing

Writing software tests is one of the key activities pertaining to software development. All FENTEC libraries are equipped with comprehensive test suites. Various parts of the libraries are tested either in:

- *unit tests* - to ensure individual functions or methods are working as expected, or in
- *integration tests* - to ensure collections of functions meant to be used together are working as expected. For instance, we run integration tests to confirm the entire sequence of setup, encryption, FE key derivation, and decryption works for all the implemented FE schemes.

Tests are used throughout the development, and allow for easier discovery of problems when new changes are introduced into the codebase. Further, when a new source code change is proposed in any of the FENTEC libraries (via GitHub pull request or GitLab merge request), tests are automatically run by a continuous integration (CI) service Travis CI [7]. It is mandatory that all tests pass before a contribution is accepted to any of the FENTEC libraries. In addition, tests are also meant to be run by the library user as a post-installation step, to check whether FENTEC libraries behave correctly on the target system.

5.2 Language-specific practices

As described in the previous Section, good practices certainly have their place in the software development process. However, on a more technical note, there are also (often community-driven) language-specific practices that have proven good, elegant or optimal through the years. A programming language may recommend or enforce a particular convention with respect to coding style (for instance, in C function names use the `snake_case`, while in Go they use `mixedCase` or `CamelCase`) or code organization. Further, a particular coding style or pattern might be optimal within the realm of one programming language, but considered sloppy, bad, or unconventional in another. This brings us to the notion of **idiomatic** code, which means following the convention of a given programming language.

Some programming language communities are very opinionated with respect to following language conventions, while others are more relaxed. Regardless, all FENTEC libraries can benefit from best practices coming from a particular programming language, both from the design and development perspective as well as from the user adoption perspective.

In the following sections we briefly present some examples of good practices for Go and C, respectively, and explain how they were applied in GoFE and CiFEr.

5.2.1 GoFE

Go is a relatively young programming language that was first released in 2009 [8]. Despite being less than a decade old, it has a large and fast growing user base. Go has many idioms that, when thoroughly followed, allow for simple, readable and mantainable code.

Naming and code organization

Go projects are organized into one or more *packages*. When a Go project imports types or functionality defined in another Go project, it accesses imported members by `packagename.ExportedMember`. Thus, package names, as well as functions and data types defined in them, are an important part of the Go libraries' APIs. Go favours short and expressive names over repetition and verbosity. For instance, in GoFE, we have packages and types like `sample.Normal` rather than `samplers.NormalSampler`.

When it comes to the question of how Go projects should be organized, the key guideline is that types and functionality that will be used close together should also be defined close together (for instance, in the same package, or in the same file). This greatly impacts how we partition data types and functionality across files and packages. One recommendation is to split Go projects into packages according to identified domains (following the practice of *domain-driven development*, or DDD). For GoFE, we initially partitioned the `innerprod` package to packages `lwe` (with schemes `Simple`, `FullySec` and `Ring`), `ddh` (with schemes `Simple` and `Damgard`) and `paillier`. However, we decided that it is more convenient to group inner product FE schemes by their security assumptions rather than how they are instantiated, thus we ended up with the `simple` and `fullysec` packages. The same approach was then followed in CiFEr.

Restricting access to data and functionality

In Go, packages expose data types and functionality to the user by exporting their members. Exported members' names start with an uppercase letter, while unexported ones start with a lowercase letter. For instance, a struct `lweParams` defined in package `simple` is unexported and thus only accessible from within the same package - the end user is not able to instantiate it. This also holds for access to fields of structs, as exported structs can (and sometimes should) restrict access to certain data or methods. In GoFE, we only export the data and behavior convenient or required for the user (in order to allow configuration of inputs and running the FE schemes), while hiding the rest.

Sometimes it makes sense to put library internals into a separate package, and refer to them from other parts of the library, which means they must be exported. Yet, we want to prevent the users from accessing the internals. In this case, we introduce package `internal`. Even if they are exported, members of `internal` package can only be imported by its parent package and parent's children. This makes the `internal` package unimportable by library users. In GoFE, the `internal` package comprises packages for computation of discrete logarithms, El Gamal key generation, error declarations, etc. This way we ensure GoFE is used for the envisioned purpose (its FE schemes), and prevent unexpected usage of library parts.

Using small interfaces

Interfaces allow us to decouple usage requirements from concrete implementations. Go favours small, ideally single-method interfaces. In GoFE, one such example is the `Sampler` interface in the `sample` package, defined as follows:

```
type Sampler interface {
        // Sample samples random big integer values,
```

```
        // possibly returning an error.
        Sample() (*big.Int, error)
}
```

GoFE provides implementations of various samplers, all of which satisfy the above interface by implementing the `Sample` method with the matching signature. This allows us to generate random vectors sampled in many different ways, since the `data.NewRandomVector()` function accepts the interface type `Sampler` as an argument. In addition, usage of such an interface allows us to re-use test code for various normal distribution samplers, as they are tested in the same manner, regardless of a specific implementation.

Error handling

Go has a built-in `error` data type. When a function or a method can fail, it typically returns a non-nil error (note that Go supports multiple return values). To reduce cyclic complexity and increase readability of the code, GoFE immediately checks and handles errors at every fallible point. Handling errors this way rather than resorting to the usage of `panic()` calls is idiomatic Go.

Further, all errors ocurring within/emitted from GoFE are pure Go `errors` with error strings set to descriptive messages indicating what went wrong. Despite its simplicity, this approach has proven expressive enough, and most importantly does not enforce the users of GoFE to depend on any custom error type implementations.

White-box and black-box testing

Go differentiates between so-called *white-box* and *black-box* testing. White-box tests are typically unit tests that may test internals and unexported functionalities of the package in addition to exported ones. They are located in the same package as the code under test (for instance `packagename`). On the other hand, we typically place black-box tests in the same directory as white-box tests, but name the package `packagename_test` to ensure test code can only access exported functionalities. Black-box testing can give us a better idea of what it is like to use our APIs. All the FE schemes in GoFE are black-box tested, while the majority of other library functionalities are tested in white-box tests.

Usage of linters

Throughout GoFE development we have made use of various Go tools for static code analysis. Most importantly, we have used *gometalinter* [9], which encapsulates numerous Go linters, allowing us to discover problems like dead code, untintentional variable shadowing, potential errors that compile, etc. Such linters have also proven to be a great support tool when conducting code reviews.

5.2.2 CiFEr

As the C programming language has been around since 1972 and is still very widely used, it has no "golden standard" of best practices. In some areas, the best choice is unclear and it is up to the programmer to choose one of the options and use it consistently.

Throughout all aspects of the CiFEr library, we tried to have simplicity and ease of use as our primary goals. This also helps with avoiding some of the common pitfalls one might encounter when programming in C.

Tooling

To build a non-trivial program in C, a tool called a build system should be used. There is no standardized build system and there are a lot of different ones to choose from. For CiFEr, the default build system is CMake [10]. CMake is a cross-platform build tool and although it is fairly modern, it is widely used in C/C++ projects. Its configuration consists of only one short file (*CMakeLists.txt*) and it can be used to create different build targets, which can be accessed by using the generated *Makefile*.

Dependency management is also quite difficult in C/C++ compared to modern programming languages since a standard package manager does not exist, thus the most common way to use external libraries is to download, compile and install them manually. CiFEr only uses two external libraries: GMP [4] for arbitrary precision integers and floating point numbers and Sodium [11] for cryptosecure random generation. Two other libraries are used in the code (a hash table implementation and a testing framework), but they are included in the repository and are automatically built together with CiFEr.

Documentation plays a critical part in a library's usability and popularity. CiFEr uses Doxygen [12], which is the most commonly used tool for generating documentation of C/C++ programs. It also supports other programming languages and multiple forms of output (HTML, XML, LaTeX...). The HTML option is a good choice because it can be easily published as a static website.

API

The API of CiFEr follows the well-known GMP's format:

```
int function(rop, op1, op2, ...)
```

where the return value indicates if the function encountered an error, `rop` is the actual value the function computes (passed as a pointer to a struct), and the remaining parameters, `op1, op2, ...` are the operands of the function.

This is a commonly used design as it can be very simple with some conventions. The memory allocation must be performed by the caller - this enables the called function to simply write its result to struct fields via the passed pointer.

Furthermore, all of the implemented cryptographic schemes' APIs are as similar to each other as possible. Functions with the same purpose have (apart from the name of the scheme) same names and their parameters are always in the same order (aligned with GMP's format, with `op1` always being a pointer to a scheme instance).

Naming and data types

Because C does not have custom defined namespaces, no two functions or data types can have the same name. In CiFEr, all function names and custom data types are prefixed with *cfe_* to prevent any possible name clashes. Additionally, functions are named based on what struct they operate on (e.g. functions in schemes are prefixed with `cfe_scheme_name_` and functions for vectors are prefixed with `cfe_vec_`).

A number of custom data types (structs) are defined in CiFEr. They may include other structs, which are directly nested inside. The other option was to use pointers, which adds a layer of indirection and does not convey directly if the struct member is a struct or an array. Since arrays are also used (via pointers) in some places, this is an additional benefit to clarity. The structs are not large, vectors and matrices (where most of the data is stored) are always allocated on the heap, thus stack overflows are unlikely to occur.

Memory management

Memory management (or lack thereof) is an important part of C. Memory on the heap must be manually allocated and freed (usually with functions `malloc` and `free`). As CiFEr uses GMP for virtually all computation, memory for big integers must be carefully managed. To make that easier, CiFEr follows the convention that each function should free all the memory it allocates. This makes debugging and fixing memory issues much easier. The well known Valgrind's [13] Memcheck tool is used for locating memory leaks.

However, there are two exceptions to this rule: 1.) allocation in `_init` functions and 2.) the crypto schemes' functions. Initialization functions initialize their respective structs in CiFEr (note that each defined struct has functions for initialization and freeing). Since an initialization function allocates memory for the struct's members which are later used, it cannot also deallocate that memory. Because of that, `_free` functions are used for cleanup at the end. The other exception, functions in crypto schemes, is due to convenience. Usually, if a function stores its result to a vector, CiFEr would require that the vector is initialized (i.e. the memory for it is allocated) and passed to the function via a pointer parameter. To do this, the caller would need to know the size of the vector precisely. This is inconvenient as the scheme should compute the size by itself. To enable that, CiFEr demands that the outputs of the schemes' functions are uninitialized and performs the allocations in them. To be consistent, the behavior is the same across all implemented schemes.

Error handling

The standard way of signaling errors from a function to its caller in C is to return a non-zero integer value. In CiFEr, an enum type is used for defining error types, which can improve debugging since the user knows what went wrong and can locate the issue with greater ease.

Memory management can be problematic if functions should immediately return in case of error. Usually, the memory is deallocated at the end of the function, but if we do not want to repeat deallocation code in every sad path (sad path is the opposite of happy path which is a default scenario where no exceptional or error condistions occur), this can lead to complicated control flow. To adhere to the DRY (*Do not Repeat Yourself*) principle, CiFEr uses the `goto` statement to jump to the cleanup section at the end of the function in case of error. Before jumping, an appropriate error code is set, which is then returned by the function. Usage of the `goto` statement is usually heavily discouraged, but as it is only used for forward jumps (even the Linux kernel uses it in this way), its use is justified.

6 Machine learning on encrypted data

Nowadays, machine learning classification is used in numerous settings such as medical or genomics predictions, face recognition, and financial predictions. Many of these applications handle sensitive data and for these it is highly beneficial that the data and the classifier remain confidential. However, tools that enable this are rare and in most cases extremely slow.

Functional encryption is one of the techniques that enable computation on encrypted data - besides homomorphic encryption, secret sharing and zero-knowledge-proofs. Currently, functional encryption does not allow training over encrypted data, but it is possible to build efficient classifiers once the model is trained. The classifier does not need any access to the plaintext, it works on ciphertexts alone.

FENTEC has provided a machine learning project [14] to demonstrate how a machine learning classifier can be built on the MNIST handwritten digit database and how functional encryption can be used to apply a classifier on the encrypted dataset. This means that an entity holding a functional encryption key for a classifier can classify encrypted images (which digit is under the ciphertext) but cannot see anything else about the image (for example some characteristics of the handwriting).

Figure 1: Classifying encrypted digits

The demonstrator uses a functional encryption scheme for quadratic multi-variate polynomials [38]. It uses the GoFE library provided by FENTEC and the widely-used machine learning library TensorFlow [15].

Each image is represented as 784 dimensional vector. A model consists of ten functions - one for each digit. The classifier returns digit with the greater value. For example the ten values returned by a model are:

```
[−99073763  −149651697  −114628671  83732640  −387336224
  130856071  −302672454  −126041027  −121102209  −111101930]
```

In the example above the classifier returns 5 (the highest value).

One can imagine for example a scenario where an entity possessing some valuable dataset of images builds a classifier, encrypts a dataset and stores it using some cloud storage service. The owner of a dataset does not want to reveal images in the clear but to enable some cloud storage functionality they are willing to provide a functional encryption key which will enable the cloud storage provider to execute computations needed for this particular functionality (for example classifying images or searching through the images).

6.1 Problems and limitations

Currently, advanced classifiers might not be possible (or might be too slow) due to the current limitations of functional encryption - efficient functional encryption schemes exist only for linear and quadratic polynomials.

Furthermore, functional encryption for evaluating a machine learning function on encrypted data is limited in the following way. Machine learning models are based on optimizing parameters that are considered to be real numbers, represented in computers as floats. Since functional encryption schemes are created around the assumption that the input values are integers, the parameters need to be discretized. If the machine learning function is relatively complex, the usual result of this procedure is that the function must operate with relatively big numbers. This presents a challenge, since in many schemes a computation of the discrete logarithm must be performed. This is extremely costly in terms of computation if the result is relatively large. Still, functional encryption is one of the most promising techniques to enable privacy-enhanced machine learning algorithms.

7 Conclusions

In this deliverable, we presented the library developed in the first year of the FENTEC project. The library comes in two flavours - in the C and Go programming languages. It contains a wide range of cryptographic primitives (lattices, pairings, discrete logarithm computations, matrice operations using modulo operator) and it offers an easy-to-use API of various functional encryption schemes. Documentation, tests and examples are provided for both flavours. The library is sufficient as a starting point for the implementation of the use case applications.

The library will be extended in the second year of the project with further schemes such as function-hiding schemes and multi-client / multi-input schemes. The latter enable a wide range of applications like running queries on encrypted databases, computation over encrypted data streams, and multi-client delegation of computation. However, the main focus in the second year will be supporting WP7 by providing any additional functions or performance optimizations required by the use cases. Also, the security will be thouroghly analysed and fixing any discovered vulnerabilities will be highly prioritized.

References

[1] https://github.com/fentec-project/gofe.

[2] https://github.com/fentec-project/cifer.

[3] https://github.com/milagro-crypto/amcl.

[4] https://gmplib.org.

[5] https://github.com/xlab-si/emmy.

[6] https://github.com/cloudflare/bn256.

[7] https://travis-ci.org.

[8] https://opensource.googleblog.com/2009/11/hey-ho-lets-go.html.

[9] https://github.com/alecthomas/gometalinter.

[10] https://cmake.org.

[11] https://github.com/jedisct1/libsodium.

[12] http://www.doxygen.nl.

[13] http://www.valgrind.org.

[14] https://github.com/fentec-project/fe-ml-example.

[15] https://www.tensorflow.org/.

[16] Michel Abdalla, Florian Bourse, Angelo De Caro, and David Pointcheval. Simple functional encryption schemes for inner products. In Jonathan Katz, editor, *Public-Key Cryptography - PKC 2015*, volume 9020 of *Lecture Notes in Computer Science*, pages 733–751. Springer, 2015.

[17] Michel Abdalla, Dario Catalano, Dario Fiore, Romain Gay, and Bogdan Ursu. Multi-input functional encryption for inner products: Function-hiding realizations and constructions without pairings. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018*, volume 10991 of *Lecture Notes in Computer Science*, pages 597–627. Springer, 2018.

[18] Shashank Agrawal and Melissa Chase. FAME: fast attribute-based message encryption. *IACR Cryptology ePrint Archive*, 2017:807, 2017.

[19] Shweta Agrawal, Benoît Libert, and Damien Stehlé. Fully secure functional encryption for inner products, from standard assumptions. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016*, volume 9816 of *Lecture Notes in Computer Science*, pages 333–362. Springer, 2016.

[20] Ian F. Blake, Gadiel Seroussi, and Nigel P. Smart. *Advances in Elliptic Curve Cryptography*. Cambridge University Press, 2005.

[21] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2001.

[22] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer, 2001.

[23] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *Journal of cryptology*, 17(4):297–319, 2004.

[24] Jérémy Chotard, Edouard Dufour Sans, Romain Gay, Duong Hieu Phan, and David Pointcheval. Decentralized multi-client functional encryption for inner product. *IACR Cryptology ePrint Archive*, 2017:989, 2017.

[25] Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Efficient software implementation of ring-LWE encryption. In Wolfgang Nebel and David Atienza, editors, *Design, Automation & Test in Europe- DATE 2015*, pages 339–344. ACM, 2015.

[26] Diego de Freitas Aranha, Conrado Porto Lopes Gouvea, and Tobias Markmann. RELIC. `https://github.com/dis2/bls12`.

[27] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2013.

[28] Laura Fuentes-Castaneda, Edward Knapp, and Francisco Rodriguez-Henriquez. Faster hashing to g2. In *International Workshop on Selected Areas in Cryptography*, pages 412–430. Springer, 2011.

[29] Norman Göttert, Thomas Feller, Michael Schneider, Johannes A. Buchmann, and Sorin A. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 512–529. Springer, 2012.

[30] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 89–98. Acm, 2006.

[31] Antoine Joux. A one round protocol for tripartite diffie-hellman. *J. Cryptology*, 17(4):263–276, 2004.

[32] Donald Knuth and Andrew Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press, 1976.

[33] Adam Langley, Kevin Burke, Filipo Valsorda, and David Symonds. Package bn256. `https://godoc.org/golang.org/x/crypto/bn256`, 2012.

[34] Allison Lewko and Brent Waters. Decentralizing attribute-based encryption. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 568–588. Springer, 2011.

[35] Ben Lynn. The Pairing Based Cryptography library. `https://crypto.stanford.edu/pbc/`.

[36] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.

[37] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *Theory of Computing - STOC 2005*, pages 84–93. ACM, 2005.

[38] Edouard Dufour Sans, Romain Gay, and David Pointcheval. Reading in the dark: Classifying encrypted digits with functional encryption. *IACR Cryptology ePrint Archive*, 2018:206, 2018.