D6.3 Final Functional Encryption Toolset API

Document Identification			
Status	Final	**Due Date**	30/6/2020
Version	1.0	**Submission Date**	30/6/2020

Related WP	WP6	**Document Reference**	D6.3
Related Deliverable(s)	D6.1, D6.2	**Dissemination Level(*)**	PU
Lead Participant	XLAB	**Lead Author**	Miha Stopar (XLAB)
Contributors	UEDIN, FUAS	**Reviewers**	Hendrik Waldner (UEDIN) Clement Gentilucci (FUAS)

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.

Document Information

List of Contributors	
Name	Partner
Miha Stopar	XLAB
Tilen Marc	XLAB
Miguel Angel Mateo	ATOS

Document History			
Version	Date	Change editors	Changes
0.1	11/5/2020	Miha Stopar (XLAB)	Inner-Product schemes
0.2	15/5/2020	Miha Stopar (XLAB)	Function-Hiding and Quadratic schemes
0.3	19/5/2020	Miha Stopar (XLAB)	ABE schemes
0.4	23/5/2020	Miha Stopar (XLAB)	Showcases
0.5	26/5/2020	Miha Stopar (XLAB)	Showcases
1.0	30/7/2019	Miha Stopar (XLAB)	Finalization

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable Leader	Miha Stopar (XLAB)	30/7/2019
Technical Manager	Michel Abdalla (ENS)	30/7/2019
Quality Manager	Diego Esteban (ATOS)	30/7/2019
Project Coordinator	Francisco Gala (ATOS)	30/7/2019

Table of Contents

Document Information . . . 1
Table of Contents . . . 3
List of Figures . . . 4
List of Acronyms . . . 5
Executive Summary . . . 6
1 Introduction . . . 7
1.1 Purpose of the Document . . . 7
1.2 Structure of the Document . . . 7
2 Single-Input Inner-Product Schemes . . . 8
2.1 Simple Functional Encryption Schemes for Inner Products – DDH based . . . 10
2.2 Simple Functional Encryption Schemes for Inner Products – LWE based . . . 14
2.3 Simple Functional Encryption Schemes for Inner Products – RLWE based . . . 16
2.4 Fully Secure Functional Encryption for Inner Products, from Standard Assumptions – DDH based . . . 19
2.5 Fully Secure Functional Encryption for Inner Products, from Standard Assumptions – DCR based . . . 22
2.6 Fully Secure Functional Encryption for Inner Products, from Standard Assumptions – LWE based . . . 25
3 Multi-Input Inner-Product Schemes . . . 28
3.1 Multi-Input Functional Encryption for Inner Products: Function-Hiding Realizations and Constructions without Pairings . . . 29
4 Quadratic Schemes . . . 32
4.1 Reading in the Dark: Classifying Encrypted Digits with Functional Encryption . . . 32
4.2 A New Paradigm for Public-Key Functional Encryption for Degree-2 Polynomials . . . 34
5 Decentralized Inner-Product . . . 36
5.1 Decentralized Multi-Client Functional Encryption for Inner Product . . . 36
5.2 Decentralizing Inner-Product Functional Encryption . . . 38
6 Function-Hiding Single-Input Inner-Product . . . 40
6.1 Function-Hiding Inner Product Encryption is Practical . . . 40
7 Function-Hiding Multi-Input Inner-Product . . . 43
7.1 Full-Hiding (Unbounded) Multi-Input Inner Product Functional Encryption from the k-Linear Assumption . . . 43
8 Attribute-Based Encryption . . . 46
8.1 Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data (KP-ABE) . . . 46
8.2 FAME: Fast Attribute-based Message Encryption (CP-ABE) . . . 49
8.3 Decentralized Policy-Hiding Attribute-Based Encryption with Receiver Privacy . . . 51
9 Showcases . . . 54
9.1 Privacy-Friendly Prediction of Cardiovascular Diseases . . . 54
9.2 Underground Anonymous Heatmap . . . 63

9.3 Selective Access to Clinical Data 66
9.4 Neural Networks on Encrypted MNIST Dataset 74
10 Conclusions 78
References 79

List of Figures

1 GoFE logo . . . 8
2 Simple inner-product . . . 8
3 Without Key Generator subject . . . 9
4 Without Key Generator subject . . . 10
5 GoFE – insecure schemes? . . . 19
6 Multi-input inner-product . . . 28
7 Evil Data Analytics subject . . . 29
8 Quadratic scheme . . . 32
9 Decentralized scheme . . . 36
10 Generation of key shares . . . 37
11 Function-Hiding scheme . . . 40
12 Function-Hiding Multi-Input scheme . . . 43
13 KP-ABE scheme . . . 46
14 CP-ABE scheme . . . 49
15 MA-ABE Policy-Hiding scheme . . . 51
16 Parameters for cardiovascular disease risk computation . . . 54
17 Computation of cardiovascular disease risk . . . 55
18 App following the User's path . . . 63
19 Users sending encrypted paths to the Service . . . 64
20 Interactions between clinical data components . . . 67
21 Decryptor module execution . . . 72
22 Clinical history file example . . . 72
23 Clinical history encrypted file . . . 73
24 Data fetched by a doctor . . . 73
25 Can you classify encrypted digits? . . . 74
26 Two-layer neural network . . . 74

List of Acronyms

Acronym	Description
ABE	Attribute Based Encryption
API	Application Programming Interface
CP-ABE	Ciphertext Policy Attribute-Based Encryption
DCR	Decisional Composite Residuosity
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
MSP	Monotone Span Program
LWE	Learning With Errors
RLWE	Ring-Learning With Errors

Executive Summary

In this deliverable D6.3 “Final Functional Encryption Toolset API”, we present a functional encryption library that offers multiple schemes for attribute-based encryption, inner-product functional encryption, and quadratic functional encryption. To support various use cases and to ease the integration, the library comes in two flavors – in Go and C programming languages. We present the API of both flavors and give examples of how the library can be integrated into real-world applications. The examples aim to showcase the FENTEC toolset API and to simplify the adoption of libraries.

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 is a set of four algorithms. The setup algorithm takes as input a security parameter and generates a public key for the system together with a master secret key. The key derivation algorithm generates a secret key sk_f for a particular function f. To encrypt a message x, the encryption algorithm on input x is to be run. Then, given the encryption of a message x, the holder of the secret key sk_f for the function f is able to compute the value of $f(x)$ using the decryption algorithm. Nothing else can be learned from the encrypted x.

This deliverable presents the functional encryption library developed in the FENTEC project. FENTEC provides functional encryption library in two flavours: GoFE implemented in Go language, CiFEr in C language. Both flavors contain a variety of inner-product, quadratic, and attribute-based encryption (ABE) schemes. The document presents the API, as well as demonstrations and explanations on how to use it.

1.1 Purpose of the Document

The goal of this deliverable is to present the readiness of the FENTEC library to be included in real-world applications. The document can serve as a manual on how to use GoFE and CiFEr.

1.2 Structure of the Document

This deliverable is structured as follows. Section 2 presents the API for single-input inner-product schemes. Section 3 presents the API for multi-input inner-product schemes. Section 4 presents the API for quadratic polynomial schemes. Section 5 presents the API for decentralized inner-product schemes. Section 6 presents the API for function-hiding single-input inner-product schemes. Section 7 presents the API for function-hiding multi-input inner-product schemes. Section 8 presents the API for attribute-based encryption schemes. Section 9 presents real-world demonstrations on how to use the FENTEC library. The deliverable concludes with a summary in Section 10.

2 Single-Input Inner-Product Schemes

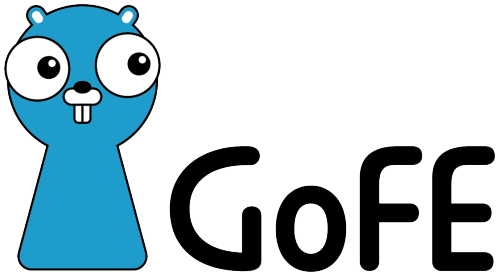

Figure 1: GoFE logo

This section presents single-input inner-product schemes implemented in GoFE and CiFEr. It describes the intended usage for single-input inner-product schemes and gives examples of how to use them with GoFE and CiFEr API.

What are the subjects involved in the single-input inner-product schemes?

The subjects are:

- Users which encrypt their data and send encrypted data to the Data Analytics subject.
- Data Analytics collects encrypted data from Users and is able to compute some functions on the encrypted data.
- Any subject which would like to compute functions on the encrypted data needs functional decryption keys. These keys are provided by a Key Generator.

What do these schemes enable?

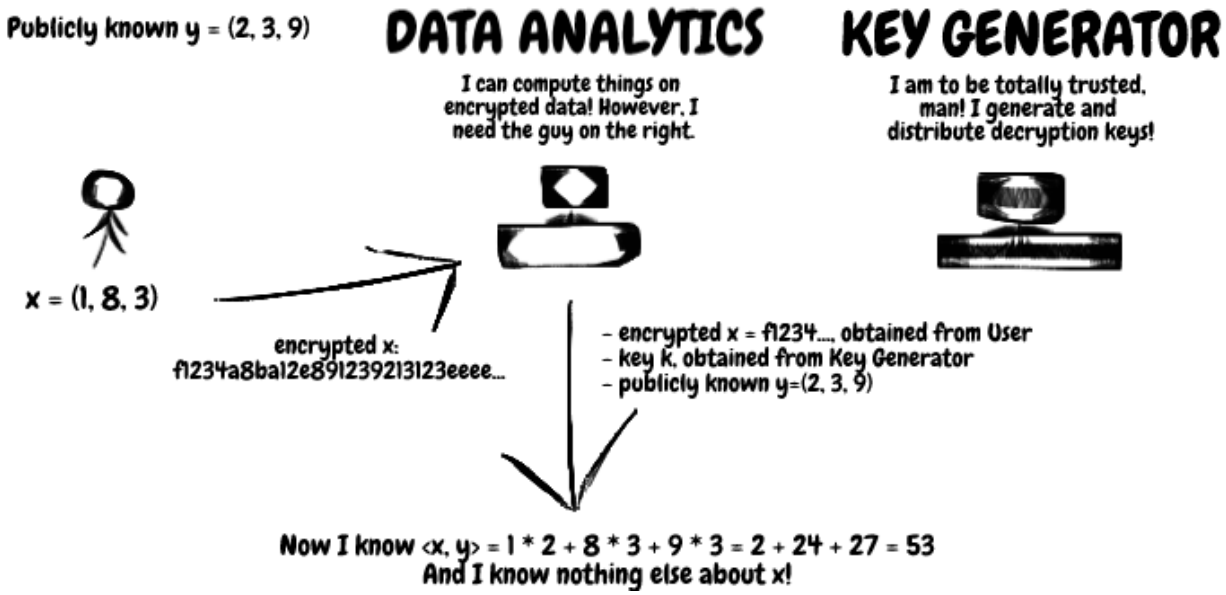

Figure 2: Simple inner-product

Given a publicly known vector y, Key Generator can provide a functional decryption key k_y, which enables

the key holder to compute the inner-product of x and y: $< x, y >$.

A subject having the encryption of x and a key k_y can compute $< x, y >$.

Note that with k_y, it is possible to compute $< x, y >$ for every x. Also, if Data Analytics subject wants to compute the inner-product for some other vector, let us say $< x, z >$, a new functional decryption key k_z needs to be generated by the Key Generator, however, the same encryption of x can be used for the computation of $< x, y >$ and $< x, z >$.

Is Key Generator really needed?

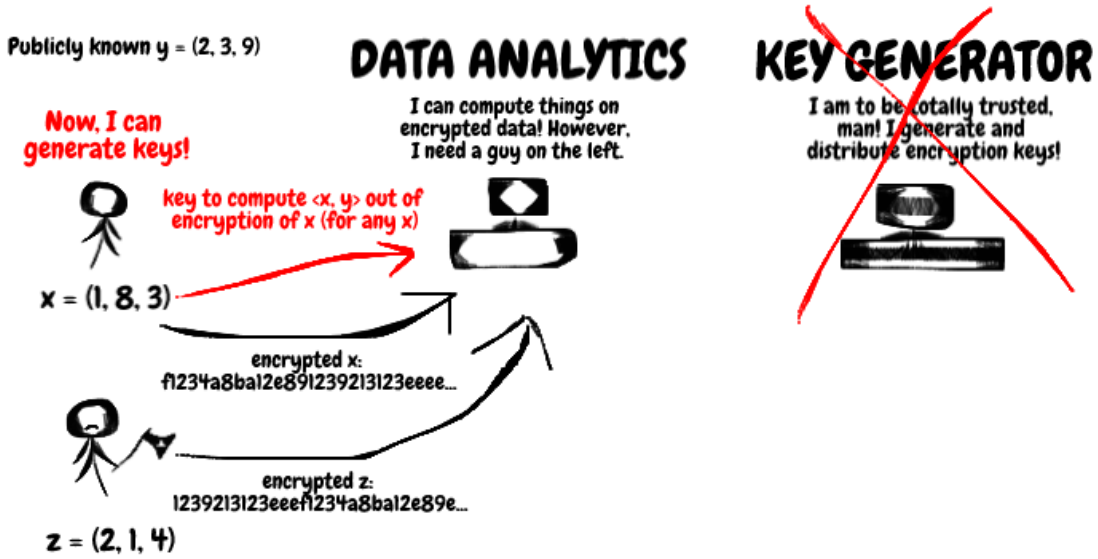

Figure 3: Without Key Generator subject

Generally, it is good to avoid the need for a trusted third party, because it presents a single point of failure.

We could pass the responsibility of key generation to one of the Users, but this would not solve the problem – there would still be a single point of failure. Also, other Users might not want to tolerate one of them having the power to decrypt all messages.

If there is only one User, the latter is not a problem. Indeed, if only one User is involved, an independent Key Generator is not needed. But in this case, functional encryption is not needed – Users can, for example, simply encrypt $< x, y >$ using the Data Analytics public key (using traditional public-key encryption) and send the ciphertext to Data Analytics.

So why not always use traditional public-key encryption? Because if Data Analytics later needs to compute $< x, y >$ for some other y, the Users will need to compute $< x, y >$ again for the new y, encrypt it and send it to the Data Analytics. By using functional encryption, no additional encryption and sending of data is needed, Data Analytics just need to obtain a new functional decryption key.

One of the first practical functional encryption schemes was "Simple Functional Encryption Schemes for Inner Products" by Abdalla et al. [11]. The paper presents instantiations from DDH and LWE. Additionally, FENTEC partners designed a scheme based on RLWE which requires smaller parameters than the LWE version and is thus significantly faster. In what follows we present the API for DDH, LWE, and RLWE instantiations.

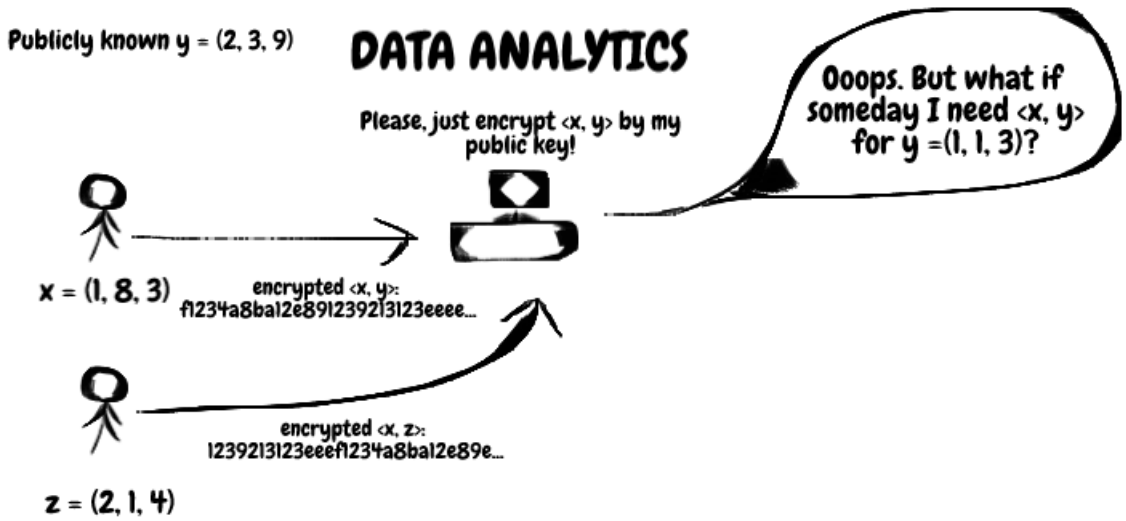

Figure 4: Without Key Generator subject

2.1 Simple Functional Encryption Schemes for Inner Products – DDH based

2.1.1 GoFE API

Generator

Let us first show how to instantiate a *sampler*, which is used to generate random numbers, vectors, and matrices. GoFE provides multiple samplers, all of them are provided in the *sample* package and all implement *sample.Sampler* interface.

```
l := 3
bound := new(big.Int).Exp(big.NewInt(2), big.NewInt(10), nil)
sampler := sample.NewUniformRange(new(big.Int).Add(new(big.Int).Neg(bound), big.
    ↪ NewInt(1)), bound)
```

Vector y is usually defined by Data Analytics (y determines the function – what kind of computation it will be able to compute), however, for the demonstration purposes, here we generate a random vector y:

```
y, err := data.NewRandomVector(l, sampler)
if err != nil {
       t.Fatalf("Error during random generation: %v", err)
}
```

The scheme is implemented in the *innerprod/simple* package. It can be instantiated using the *simple.NewDDH* or *simple.NewDDHPrecomp* functions.

The difference between the two is that the latter uses a precomputed group (based on the precomputed prime numbers and group generators). This allows a fast scheme initialization for realistic parameters, i.e. prime numbers with bit length 2048 or more.

```
var simpleDDH *simple.DDH
var err error
if param.precomputed {
	simpleDDH, err = simple.NewDDHPrecomp(l, param.modulusLength, bound)
} else {
	simpleDDH, err = simple.NewDDH(l, param.modulusLength, bound)
}
if err != nil {
	t.Fatalf("Error during simple inner product creation: %v", err)
}
```

What are the other parameters?

Parameter l specifies the length of vectors x and y.

Parameter *modulusLength* specifies the bit length of p (operations take place in Z_p group), it should be one of the following values 1024, 1536, 2048, 2560, 3072, or 4096.

Parameter *bound* specifies the bound of the coordinates of the vectors x and y. Note that vector coordinates need to be bounded, otherwise the scheme might not be sufficiently fast as the decryption phase requires the computation of the discrete logarithm which is slow for big numbers.

Once the scheme is instantiated, the Key Generator can generate master keys:

```
masterSecKey, masterPubKey, err := simpleDDH.GenerateMasterKeys()
```

Note that *masterSecKey* will be needed only by the Key Generator to derive functional decryption keys. On the other hand, *masterPubKey* will be needed by the User to encrypt vector x.

When asked for a functional decryption key for vector y, Key Generator executes:

```
funcKey, err := simpleDDH.DeriveKey(masterSecKey, y)
if err != nil {
	t.Fatalf("Error during key derivation: %v", err)
}
```

The *funcKey* is to be given to the requester (Data Analytics in our case). Checking whether a requester is entitled to the key is different from use case to use case and is to be implemented separately.

User

Vector x might, for example, represent some IoT measurements. Here, we generate a random vector x. Note that the User needs to instantiate a *sampler* in the same way it was shown for the Key Generator.

```
x, err := data.NewRandomVector(l, sampler)
if err != nil {
	t.Fatalf("Error during random generation: %v", err)
}
```

To encrypt vector x, User needs to instantiate *DDH* first. The same scheme parameters need to be used as for the Key Generator. This can be done most easily by using one of the precomputed groups.

```
encryptor := simple.NewDDHPrecomp(l, param.modulusLength, bound)
ciphertext, err := encryptor.Encrypt(x, masterPubKey)
if err != nil {
       t.Fatalf("Error during encryption: %v", err)
}
```

The ciphertext is then sent to the Data Analytics subject.

Data Analytics

Data Analytics subject needs to instantiate the scheme with the same parameters as Key Generator and User. Once it obtains the functional decryption key for y from Key Generator, it can pass ciphertext of x, functional decryption key, and y to the *Decrypt* operation to compute the inner-product $< x, y >$.

```
decryptor := simple.NewDDHFromParams(simpleDDH.Params)
xy, err := decryptor.Decrypt(ciphertext, funcKey, y)
if err != nil {
       t.Fatalf("Error during decryption: %v", err)
}
```

2.1.2 CiFEr API

Generator

The scheme *cfe_ddh* is implemented in the *innerprod/simple* package. It can be instantiated using the *cfe_ddh_init* or *cfe_ddh_precomp_init* functions.

As in GoFE, the difference between the two is that the latter uses a precomputed group (based on the precomputed prime numbers and generators). This allows a fast scheme initialization for realistic parameters, i.e. prime numbers with bit length 2048 or more.

```
size_t l = 3;

mpz_t bound, bound_neg;
mpz_inits(bound, bound_neg, NULL);
mpz_set_ui(bound, 2);
mpz_pow_ui(bound, bound, 10);
mpz_neg(bound_neg, bound);

cfe_ddh s;
cfe_error err;

size_t modulus_len;
const char *precomp = munit_parameters_get(params, "parameters");
if (strcmp(precomp, "precomputed") == 0) {
    modulus_len = 2048;
    err = cfe_ddh_precomp_init(&s, l, modulus_len, bound);
} else if (strcmp(precomp, "random") == 0) {
    modulus_len = 512;
```

```
    err = cfe_ddh_init(&s, l, modulus_len, bound);
} else {
    err = CFE_ERR_INIT;
}
```

What are the other parameters?

Parameter l specifies the length of vectors x and y.

Parameter *modulus_len* specifies the bit length of p (operations take place in Z_p group), it should be one of the following values 1024, 1536, 2048, 2560, 3072, or 4096.

Parameters *bound* and *bound_neg* specifies the bound of the coordinates of the vectors x and y. Note that vector coordinates need to be bounded, otherwise the scheme might not be sufficiently fast as the decryption phase requires the computation of the discrete logarithm.

Vector y is usually defined by Data Analytics subject (y determines the function – what kind of computation it will be able to compute), however, for the demonstration purposes, here we generate a random vector y by using *cfe_uniform_sample_range_vec*:

```
cfe_vec y;
cfe_vec_inits(&y, l);
cfe_uniform_sample_range_vec(&y, bound_neg, bound);
```

Once the scheme is instantiated, Key Generator can generate the master keys:

```
cfe_vec msk, mpk;
cfe_ddh_master_keys_init(&msk, &mpk, &s);
cfe_ddh_generate_master_keys(&msk, &mpk, &s);
```

Note that the master secret key *msk* will be needed only by the Key Generator to derive functional decryption keys. On the other hand, the master public key *mpk* will be needed by the User to encrypt vector x.

When asked for a functional decryption key for vector y, Key Generator executes:

```
mpz_t fe_key;
mpz_init(fe_key);
err = cfe_ddh_derive_fe_key(fe_key, &s, &msk, &y);
```

The *fe_key* is to be given to the requester (Data Analytics subject in our case). Checking whether a requester is entitled to the key is different from use case to use case and needs to be implemented separately.

User

Vector x might, for example, represent some IoT measurements. Here, we generate a random vector x.

```
cfe_vec x;
cfe_vec_init(&x, l);
cfe_uniform_sample_range_vec(&x, bound_neg, bound);
```

To encrypt vector x, User needs to instantiate *cfe_ddh* first. The same scheme parameters need to be used as for the Key Generator. This can be most easily achieved by using the same precomputed group for all subjects.

```
cfe_ddh encryptor;
size_t modulus_len;
modulus_len = 2048;
err = cfe_ddh_precomp_init(&encryptor, l, modulus_len, bound);
```

Vector x is then encrypted as:

```
cfe_vec ciphertext;
cfe_ddh_ciphertext_init(&ciphertext, &encryptor);
err = cfe_ddh_encrypt(&ciphertext, &encryptor, &x, &mpk);
```

The ciphertext is then sent to the Data Analytics subject.

Data Analytics

Data Analytics subject needs to instantiate the scheme with the same parameters as Key Generator and User. Once it obtains a functional decryption key for y from the Key Generator, it can pass ciphertext of x, functional decryption key, and y to the *Decrypt* operation to compute the inner-product $< x, y >$.

```
cfe_ddh decryptor;
err = cfe_ddh_precomp_init(&decryptor, l, modulus_len, bound);

mpz_t xy;
mpz_init(xy);
err = cfe_ddh_decrypt(xy, &decryptor, &ciphertext, fe_key, &y);
```

2.2 Simple Functional Encryption Schemes for Inner Products – LWE based

2.2.1 GoFE API

Generator

The scheme is implemented in the *innerprod/simple* package. It can be instantiated using the *simple.NewLWE* function.

```
l := 4
n := 128
b := big.NewInt(10000)

simpleLWE, err := simple.NewLWE(l, b, b, n)
```

Parameter l specifies the length of vectors x and y.

Parameter n is the main security parameter of the scheme.

Parameter b specifies the bound of the coordinates of the vectors x and y.

Once the scheme is instantiated, Key Generator can generate master keys:

```
SK, err := simpleLWE.GenerateSecretKey()
PK, err := simpleLWE.GeneratePublicKey(SK)
```

Note that *SK* will be needed only by the Key Generator to derive functional decryption keys. On the other hand, *PK* will be needed by the User to encrypt vector x.

When asked for a functional decryption key for vector y, Key Generator executes:

```
skY, err = simpleLWE.DeriveKey(y, SK)
```

The *skY* is to be given to the requester (Data Analytics in our case). Checking whether a requester is entitled to the key is different from use case to use case and is to be implemented separately.

User

Vector x might, for example, represent some IoT measurements. To encrypt vector x, User needs to instantiate *LWE* first.

```
cipher, err := simpleLWE.Encrypt(x, PK)
```

The ciphertext is then sent to Data Analytics.

Data Analytics

Once Data Analytics obtains functional decryption key for y from Key Generator, it can pass ciphertext of x, functional decryption key, and y to the *Decrypt* operation to compute the inner-product $< x, y >$.

```
xyDecrypted, err := simpleLWE.Decrypt(cipher, skY, y)
```

2.2.2 CiFEr API

Generator

The scheme is implemented in the *innerprod/simple* package. It can be instantiated using the *cfe_lwe_init* function.

```
size_t l = 4;
size_t n = 128;

mpz_t B, B_neg;
mpz_init_set_ui(B, 10000);
mpz_init(B_neg);
mpz_neg(B_neg, B);

cfe_lwe s;
cfe_error err = cfe_lwe_init(&s, l, B, B, n);
```

Parameter l specifies the length of vectors x and y.

Parameter n is the main security parameter of the scheme.

Parameter B specifies the bound of the coordinates of the vectors x and y.

Once the scheme is instantiated, Key Generator can generate master keys:

```
cfe_mat SK, PK; // secret and public keys
cfe_lwe_sec_key_init(&SK, &s);
cfe_lwe_generate_sec_key(&SK, &s);
cfe_lwe_pub_key_init(&PK, &s);
err = cfe_lwe_generate_pub_key(&PK, &s, &SK);
```

Note that *SK* will be needed only by Key Generator to derive functional decryption keys. On the other hand, *PK* will be needed by the User to encrypt vector x.

When asked for a functional decryption key for vector y, Key Generator executes:

```
cfe_vec fe_key;
cfe_lwe_fe_key_init(&fe_key, &s);
err = cfe_lwe_derive_fe_key(&fe_key, &s, &SK, &y);
```

The *skY* is to be given to the requester (Data Analytics in our case). Checking whether a requester is entitled to the key is different from use case to use case and is to be implemented separately.

User

Vector x might, for example, represent some IoT measurements. To encrypt vector x, User needs to instantiate *LWE* first.

```
cfe_lwe_ciphertext_init(&ct, &s);
err = cfe_lwe_encrypt(&ct, &s, &x, &PK);
```

The ciphertext is then sent to the Data Analytics subject.

Data Analytics

Once Data Analytics subject obtains functional decryption key for y from Key Generator, it can pass ciphertext of x, functional decryption key, and y to the *Decrypt* operation to compute the inner-product $< x, y >$.

```
mpz_t res;
mpz_init(res);
err = cfe_lwe_decrypt(res, &s, &ct, &fe_key, &y);
```

2.3 Simple Functional Encryption Schemes for Inner Products – RLWE based

2.3.1 GoFE API

Generator

The scheme is implemented in the *innerprod/simple* package. It can be instantiated using the *simple.NewRingLWE* function.

```
l := 100
n := 256
b := big.NewInt(1000000)
p, _ := new(big.Int).SetString("10000000000000000", 10)
q, _ := new(big.Int).SetString("90346868817997361638783029959 9", 10)
sigma := big.NewFloat(20)
```

Parameter l specifies the length of vectors x and y.

Parameter n is the main security parameter of the scheme.

Parameter b specifies the upper bound for coordinates of input vectors x and y.

Parameter p specifies modulus for the resulting inner product.

Parameter q specifies modulus for ciphertext and keys.

Parameter *sigma* specifies standard deviation – settings for a discrete gaussian sampler.

Once the scheme is instantiated, Key Generator can generate master keys:

```
SK, err := ringLWE.GenerateSecretKey()
PK, err := ringLWE.GeneratePublicKey(SK)
```

Note that *SK* will be needed only by Key Generator to derive functional decryption keys. On the other hand, *PK* will be needed by the User to encrypt vector x.

When asked for a functional decryption key for vector y, Key Generator executes:

```
skY, err := ringLWE.DeriveKey(y, SK)
```

The *skY* is to be given to the requester (Data Analytics in our case). Checking whether a requester is entitled to the key is different from use case to use case and is to be implemented separately.

User

Vector x might, for example, represent some IoT measurements. To encrypt vector x, User needs to instantiate *LWE* first.

```
cipher, err := ringLWE.Encrypt(X, PK)
```

The ciphertext is then sent to Data Analytics.

Data Analytics

Once Data Analytics subject obtains functional decryption key for y from Key Generator, it can pass ciphertext of x, functional decryption key, and y to the *Decrypt* operation to compute the inner-product $< x, y >$.

```
xyDecrypted, err := ringLWE.Decrypt(cipher, skY, y)
```

2.3.2 CiFEr API

Generator

The scheme is implemented in the *innerprod/simple* package. It can be instantiated using the *cfe_ring_lwe_init* function.

```
size_t l = 100;
size_t n = 256;

mpz_t B, B_neg;
mpz_inits(B, B_neg, NULL);
mpz_set_si(B, 1000000);
mpz_neg(B_neg, B);

mpf_t sigma;
mpf_init_set_ui(sigma, 20);

mpz_t p, q;
mpz_init_set_str(p, "10000000000000000", 10);
mpz_init_set_str(q, "90346868817997361638783029959 9", 10);

cfe_ring_lwe s;
cfe_error err = cfe_ring_lwe_init(&s, l, n, B, p, q, sigma);
```

Parameter l specifies the length of vectors x and y.

Parameter n is the main security parameter of the scheme.

Parameter B specifies the bound of the coordinates of the vectors x and y.

Parameter p specifies modulus for the resulting inner product.

Parameter q specifies modulus for ciphertext and keys.

Parameter *sigma* specifies standard deviation – settings for a discrete gaussian sampler.

Once the scheme is instantiated, Key Generator can generate master keys:

```
cfe_mat SK, PK; // secret and public keys
cfe_ring_lwe_sec_key_init(&SK, &s);
cfe_ring_lwe_generate_sec_key(&SK, &s);
cfe_ring_lwe_pub_key_init(&PK, &s);
cfe_ring_lwe_generate_pub_key(&PK, &s, &SK);
```

Note that *SK* will be needed only by Key Generator to derive functional decryption keys. On the other hand, *PK* will be needed by User to encrypt vector x.

When asked for a functional decryption key for vector y, Key Generator executes:

```
cfe_vec fe_key;
cfe_ring_lwe_fe_key_init(&fe_key, &s);
err = cfe_ring_lwe_derive_fe_key(&fe_key, &s, &SK, &y);
```

The *fe_key* is to be given to the requester (Data Analytics in our case). Checking whether a requester is entitled to the key is different from use case to use case and is to be implemented separately.

User

Vector x might, for example, represent some IoT measurements. To encrypt vector x, User needs to instantiate *LWE* first.

```
cfe_mat CT;
cfe_ring_lwe_ciphertext_init(&CT, &s);
err = cfe_ring_lwe_encrypt(&CT, &s, &X, &PK);
```

The ciphertext is then sent to the Data Analytics subject.

Data Analytics

Once Data Analytics subject obtains functional decryption key for y from Key Generator, it can pass ciphertext of x, functional decryption key, and y to the *Decrypt* operation to compute the inner-product $< x, y >$.

```
cfe_vec res;
cfe_ring_lwe_decrypted_init(&res, &s);
err = cfe_ring_lwe_decrypt(&res, &s, &CT, &fe_key, &y);;
```

2.4 Fully Secure Functional Encryption for Inner Products, from Standard Assumptions – DDH based

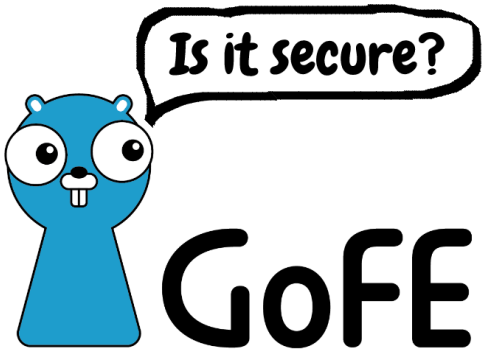

Figure 5: GoFE – insecure schemes?

In the previous sections, we presented the first practical inner-product schemes: Simple Functional Encryption Schemes for Inner Products by Abdalla et al. [11].

Later on, further practical inner-product schemes have been designed, some of them being presented in a paper "Fully Secure Functional Encryption for Inner Products, from Standard Assumptions" by Agrawal et al. [15]. The paper presented multiple new schemes, in this section, we describe the API for the DDH variant.

Why is it named fully secure? Does this imply schemes by Abdalla et al. [11] are not secure?

Not really.

However, they are showed to be only selectively secure. That means, in the proof of the construction the attacker must declare upfront, before seeing the public parameters, what the challenge ciphertexts will be. On the other hand, achieving adaptive security (also called full security), where security is guaranteed even for messages that are adaptively chosen at any point in time, is significantly more challenging.

But even if the scheme is proven to be only selectively secure, it might be adaptively secure as well. Still, the selectively secure schemes are usually seen as an intermediary step. Thus, whenever inner-products are needed, [15] API should be preferred over [11] API.

We summarize the [15] API below. Note that the roles and workflow are the same as in [11] API.

2.4.1 GoFE API

Generator

The scheme is implemented in the *innerprod/fullysec* package. It can be instantiated using the *fullysec.NewDamgard* or *fullysec.NewDamgardPrecomp* functions.

The difference between the two is that the latter uses a precomputed group (based on the precomputed prime numbers and generators). This allows a fast scheme initialization for realistic parameters, i.e. prime numbers with bit length 2048 or more.

```
var damgard *fullysec.Damgard
var err error
if param.precomputed {
	damgard, err = fullysec.NewDamgardPrecomp(l, param.modulusLength, bound)
} else {
	damgard, err = fullysec.NewDamgard(l, param.modulusLength, bound)
}
if err != nil {
	t.Fatalf("Error during simple inner product creation: %v", err)
}
```

Once the scheme is instantiated, Key Generator can generate master keys:

```
masterSecKey, masterPubKey, err := damgard.GenerateMasterKeys()
```

Note that *masterSecKey* will be needed only by the Key Generator to derive functional decryption keys. On the other hand, *masterPubKey* will be needed by User to encrypt vector x.

When asked for a functional decryption key for vector y, Key Generator executes:

```
funcKey, err := damgard.DeriveKey(masterSecKey, y)
if err != nil {
	t.Fatalf("Error during key derivation: %v", err)
}
```

The *funcKey* is to be given to the requester. Checking whether a requester is entitled to the key is different from use case to use case and is to be implemented separately.

User

To encrypt vector x, User needs to instantiate *Damgard* first. The same scheme parameters need to be used as for Key Generator.

```
encryptor, err = fullysec.NewDamgardPrecomp(l, param.modulusLength, bound)
ciphertext, err := encryptor.Encrypt(x, masterPubKey)
if err != nil {
       t.Fatalf("Error during encryption: %v", err)
}
```

The ciphertext is then sent to Data Analytics.

Data Analytics

Data Analytics needs to instantiate the scheme with the same parameters as Key Generator and User. Once it obtains a functional decryption key for y from Key Generator, it can pass ciphertext of x, functional decryption key, and y to the *Decrypt* operation to compute the inner-product $< x, y >$.

```
decryptor, err = fullysec.NewDamgardPrecomp(l, param.modulusLength, bound)
xy, err := decryptor.Decrypt(ciphertext, funcKey, y)
if err != nil {
       t.Fatalf("Error during decryption: %v", err)
}
```

2.4.2 CiFEr API

Generator

The scheme is implemented in the *innerprod/fullysec* package. It can be instantiated using the *cfe_damgard_init* or *cfe_damgard_precomp_init* functions.

The difference between the two is that the latter uses a precomputed group (based on the precomputed prime numbers and generators). This allows a fast scheme initialization for realistic parameters, i.e. prime numbers with bit length 2048 or more.

```
cfe_damgard s;
cfe_error err;

const char *precomp = munit_parameters_get(params, "parameters");
if (strcmp(precomp, "precomputed") == 0) {
    modulus_len = 2048;
    err = cfe_damgard_precomp_init(&s, l, modulus_len, bound);
} else if (strcmp(precomp, "random") == 0) {
    modulus_len = 512;
    err = cfe_damgard_init(&s, l, modulus_len, bound);
} else {
    err = CFE_ERR_INIT;
}
```

Once the scheme is instantiated, Key Generator can generate master keys:

```
cfe_vec mpk;
cfe_damgard_sec_key msk;
cfe_damgard_sec_key_init(&msk, &s);
cfe_damgard_pub_key_init(&mpk, &s);
cfe_damgard_generate_master_keys(&msk, &mpk, &s);
```

Note that *msk* will be needed only by Key Generator to derive functional decryption keys. On the other hand, *mpk* will be needed by User to encrypt vector x.

When asked for a functional decryption key for vector y, Key Generator executes:

```
cfe_damgard_fe_key key;
cfe_damgard_fe_key_init(&key);
err = cfe_damgard_derive_fe_key(&key, &s, &msk, &y);
```

The *funcKey* is to be given to the requester. Checking whether a requester is entitled to the key is different from use case to use case and is to be implemented separately.

User

To encrypt vector x, User needs to instantiate *cfe_damgard* first. The same scheme parameters need to be used as for Key Generator.

```
cfe_damgard encryptor;
err = cfe_damgard_precomp_init(&encryptor, l, modulus_len, bound);
cfe_vec ciphertext;
cfe_damgard_ciphertext_init(&ciphertext, &encryptor);
err = cfe_damgard_encrypt(&ciphertext, &encryptor, &x, &mpk);
```

The ciphertext is then sent to Data Analytics.

Data Analytics

Data Analytics needs to instantiate the scheme with the same parameters as Key Generator and User. Once it obtains a functional decryption key for y from Key Generator, it can pass ciphertext of x, functional decryption key, and y to the *Decrypt* operation to compute the inner-product $< x, y >$.

```
cfe_damgard decryptor;
err = cfe_damgard_precomp_init(&decryptor, l, modulus_len, bound);
err = cfe_damgard_decrypt(xy, &decryptor, &ciphertext, &key, &y);
```

2.5 Fully Secure Functional Encryption for Inner Products, from Standard Assumptions – DCR based

In the previous section, we presented the Decisional Diffie-Hellman based scheme from the paper "Fully Secure Functional Encryption for Inner Products, from Standard Assumptions" by Agrawal et al. [15]. In this section, we describe the API for the scheme based on Decisional Composite Residuosity (DCR) assumption.

The API for both schemes is very similar. However, the DCR variant has the advantage of having a much faster decryption operation. This is because there is no need to compute a discrete logarithm as part of the decryption process. It needs to be noted that other operations, especially initialization, are slightly slower in the DCR variant.

2.5.1 GoFE API

Generator

The scheme is implemented in the *innerprod/fullysec* package.

```
paillier, err := fullysec.NewPaillier(l, lambda, bitLength, boundX, boundY)
```

Parameter *l* specifies the length of data vectors.

Parameter *lambda* is a security parameter (the bigger it is, the more secure scheme is).

Parameter *bitLength* specifies the bit length of prime numbers to be used for a group where operations take place.

It should be big enough so that factoring two primes with such a bit length takes at least 2^{lambda} operations. For example, values *bitLength* = 1024 and *lambda* = 128 can be used.

Parameter *boundX* specifies a bound on the entries of the input vector.

Parameter *boundY* specifies a bound on the entries of y.

Once the scheme is instantiated, Key Generator can generate master keys:

```
masterSecKey, masterPubKey, err := paillier.GenerateMasterKeys()
```

Note that *masterSecKey* will be needed only by Key Generator to derive functional decryption keys. On the other hand, *masterPubKey* will be needed by User to encrypt vector x.

When asked for a functional decryption key for vector y, Key Generator executes:

```
key, err := paillier.DeriveKey(masterSecKey, y)
```

The *key* is to be given to the requester. Checking whether a requester is entitled to the key is different from use case to use case and is to be implemented separately.

User

To encrypt a vector x, User needs to instantiate *Paillier* first. The same scheme parameters need to be used as for Key Generator.

```
encryptor := fullysec.NewPaillierFromParams(paillier.Params)
ciphertext, err := encryptor.Encrypt(x, masterPubKey)
```

The ciphertext is then sent to Data Analytics.

Data Analytics

Data Analytics needs to instantiate the scheme with the same parameters as Key Generator and User. Once it obtains a functional decryption key for y from Key Generator, it can pass ciphertext of x, functional decryption key, and y to the *Decrypt* operation to compute the inner-product $< x, y >$.

```
decryptor := fullysec.NewPaillierFromParams(paillier.Params)
xy, err := decryptor.Decrypt(ciphertext, key, y)
```

2.5.2 CiFEr API

Generator

The scheme is implemented in the *innerprod/fullysec* package.

```
cfe_paillier s;
cfe_error err = cfe_paillier_init(&s, l, lambda, bit_len, bound_x, bound_y);
```

Once the scheme is instantiated, Key Generator can generate master keys:

```
cfe_vec msk, mpk;
cfe_paillier_master_keys_init(&msk, &mpk, &s);
err = cfe_paillier_generate_master_keys(&msk, &mpk, &s);
```

Note that *msk* will be needed only by Key Generator to derive functional decryption keys. On the other hand, *mpk* will be needed by the User to encrypt vector x.

When asked for a functional decryption key for vector y, Key Generator executes:

```
mpz_t fe_key;
mpz_init(fe_key);
err = cfe_paillier_derive_fe_key(fe_key, &s, &msk, &y);
```

The *fe_key* is to be given to the requester. Checking whether a requester is entitled to the key is different from use case to use case and is to be implemented separately.

User

To encrypt a vector x, User needs to instantiate *cfe_paillier* first. The same scheme parameters need to be used as for Key Generator.

```
cfe_paillier encryptor;
cfe_error err = cfe_paillier_init(&encryptor, l, lambda, bit_len, bound_x,
    ↪ bound_y);
cfe_vec ciphertext;
cfe_paillier_ciphertext_init(&ciphertext, &encryptor);
err = cfe_paillier_encrypt(&ciphertext, &encryptor, &x, &mpk);
```

The ciphertext is then sent to Data Analytics.

Data Analytics

Data Analytics subject needs to instantiate the scheme with the same parameters as Key Generator and User. Once it obtains a functional decryption key for y from Key Generator, it can pass ciphertext of

x, functional decryption key, and y to the *cfe_paillier_decrypt* operation to compute the inner-product $< x, y >$.

```
cfe_paillier decryptor;
cfe_error err = cfe_paillier_init(&decryptor, l, lambda, bit_len, bound_x,
    ↪ bound_y);
err = cfe_paillier_decrypt(xy, &decryptor, &ciphertext, fe_key, &y);
```

2.6 Fully Secure Functional Encryption for Inner Products, from Standard Assumptions – LWE based

In the previous two sections, we presented the DDH and DCR based schemes from the paper "Fully Secure Functional Encryption for Inner Products, from Standard Assumptions" by Agrawal et al. [15]. In this section, we describe the API for the LWE based scheme.

2.6.1 GoFE API

Generator

The scheme is implemented in the *innerprod/fullysec* package.

```
l := 4
n := 64
boundX := big.NewInt(1000)
boundY := big.NewInt(1000)
fsLWE, err := fullysec.NewLWE(l, n, boundX, boundY)
```

Parameter l specifies the length of data vectors.

Parameter n is the main security of the scheme.

Parameter *boundX* specifies a bound on the entries of the input vector. Parameter *boundY* specifies a bound on the entries of y.

Once the scheme is instantiated, Key Generator can generate master keys:

```
Z, err := fsLWE.GenerateSecretKey()
U, err := fsLWE.GeneratePublicKey(Z)
```

When asked for a functional decryption key for vector y, Key Generator executes:

```
zY, err := fsLWE.DeriveKey(y, Z)
```

The zY is to be given to the requester. Checking whether a requester is entitled to the key is different from use case to use case and is to be implemented separately.

User

To encrypt a vector x, User needs to instantiate *Paillier* first. The same scheme parameters need to be used as for Key Generator.

```
cipher, err := fsLWE.Encrypt(x, U)
```

The ciphertext is then sent to Data Analytics.

Data Analytics

Data Analytics needs to instantiate the scheme with the same parameters as Key Generator and User. Once it obtains a functional decryption key for y from Key Generator, it can pass ciphertext of x, functional decryption key, and y to the *Decrypt* operation to compute the inner-product $< x, y >$.

```
xyDecrypted, err := fsLWE.Decrypt(cipher, zY, y)
```

2.6.2 CiFEr API

Generator

The scheme is implemented in *innerprod/fullysec* package.

```
size_t l = 4;
size_t n = 64;
mpz_t bound_x, bound_x_neg, bound_y, bound_y_neg;
mpz_inits(bound_x, bound_x_neg, bound_y, bound_y_neg, NULL);
mpz_set_ui(bound_x, 1000);
mpz_set_ui(bound_y, 1000);
mpz_neg(bound_x_neg, bound_x);
mpz_neg(bound_y_neg, bound_y);
cfe_lwe_fs s;
cfe_error err = cfe_lwe_fs_init(&s, l, n, bound_x, bound_y);
```

Once the scheme is instantiated, Key Generator can generate master keys:

```
cfe_mat SK;
cfe_lwe_fs_sec_key_init(&SK, &s);
cfe_lwe_fs_generate_sec_key(&SK, &s);

cfe_mat PK;
cfe_lwe_fs_pub_key_init(&PK, &s);
cfe_lwe_fs_generate_pub_key(&PK, &s, &SK);
```

When asked for a functional decryption key for vector y, Key Generator executes:

```
cfe_vec fe_key;
cfe_lwe_fs_fe_key_init(&fe_key, &s);
err = cfe_lwe_fs_derive_fe_key(&fe_key, &s, &y, &SK);
```

The *fe_key* is to be given to the requester. Checking whether a requester is entitled to the key is different from use case to use case and is to be implemented separately.

User

To encrypt a vector x, User needs to instantiate *cfe_paillier* first. The same scheme parameters need to be used as for Key Generator.

```
cfe_vec ciphertext;
cfe_lwe_fs_ciphertext_init(&ciphertext, &s);
err = cfe_lwe_fs_encrypt(&ciphertext, &s, &x, &PK);
```

The ciphertext is then sent to Data Analytics.

Data Analytics

Data Analytics subject needs to instantiate the scheme with the same parameters as Key Generator and User. Once it obtains a functional decryption key for y from Key Generator, it can pass ciphertext of x, functional decryption key, and y to the *cfe_paillier_decrypt* operation to compute the inner-product $< x, y >$.

```
mpz_t res;
mpz_init(res);
err = cfe_lwe_fs_decrypt(res, &s, &ciphertext, &fe_key, &y);
```

3 Multi-Input Inner-Product Schemes

Single-Input Inner-Product schemes enable analytics over the (encrypted) data of a single User. What if Data Analytics subject is interested in computing the function that takes as an input data from multiple Users?

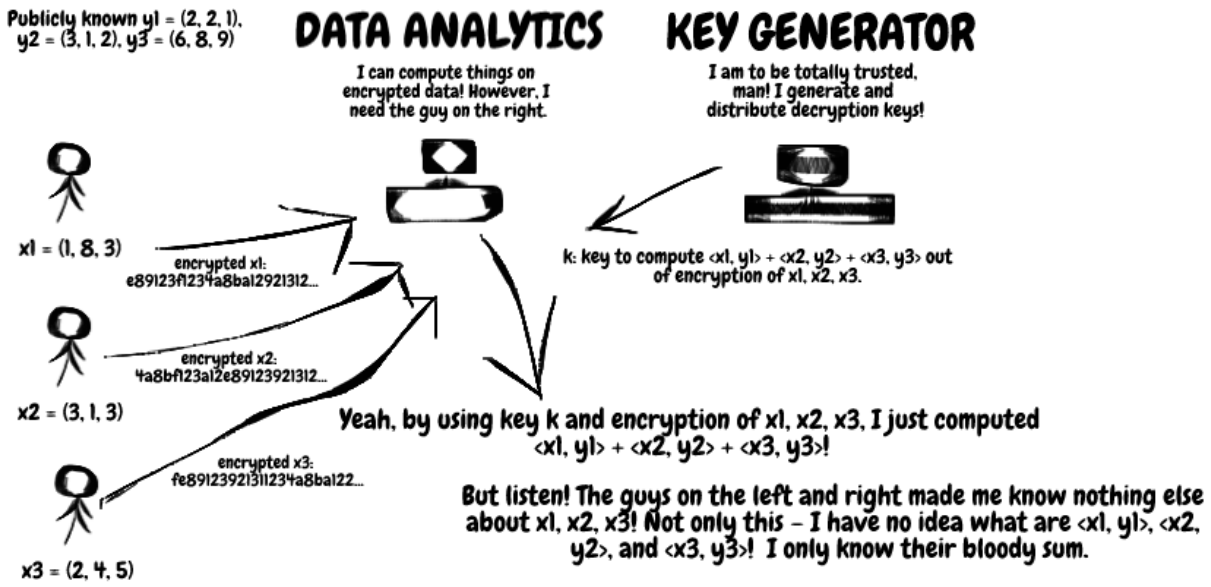

Figure 6: Multi-input inner-product

Such scenarios are addressed by multi-input inner-product schemes. Multiple Users send encrypted data to Data Analytics. Let us say we have three Users. For publicly known vectors y_1, y_2, y_3, Data Analytics can compute $< x_1, y_1 > + < x_2, y_2 > + < x_3, y_3 >$ by having only a functional decryption key (for these particular known vectors) and ciphertexts of x_1, x_2, and x_3.

The important detail is that Data Analytics does not learn anything about $< x_1, y_1 >, < x_2, y_2 >, < x_3, y_3 >$, except their sum: $< x_1, y_1 > + < x_2, y_2 > + < x_3, y_3 >$.

Multi-input inner product schemes enable mining encrypted datasets. For example, our three Users could encrypt electricity consumption information and send it to Data Analytics. In this case, vector x_i presents electricity consumption of household appliances for i-th User (Home). Each User sends an encrypted vector each hour.

For each hour, Data Analytics can compute the aggregated consumption of all Users (by simply using $y_i = (1, 1, 1)$). Data Analytics can thus know the overall peak hours, but does not know, for example, the peak hours for each particular User.

Private-key setting

Single-input inner-product schemes presented in the previous section are public-key schemes. For encryption, the Users use the master public key generated by Key Generator. On the contrary, all multi-input inner-product schemes need to be private-key schemes – each User needs a private key to be able to encrypt the messages.

Private-key setting is necessary as otherwise Data Analytics could use public key to encrypt, for example, x_1 and x_3. By having a functional decryption key for y_1, y_2, y_3, it could compute $< x_1, y_1 > + < x_2, y_2 >$

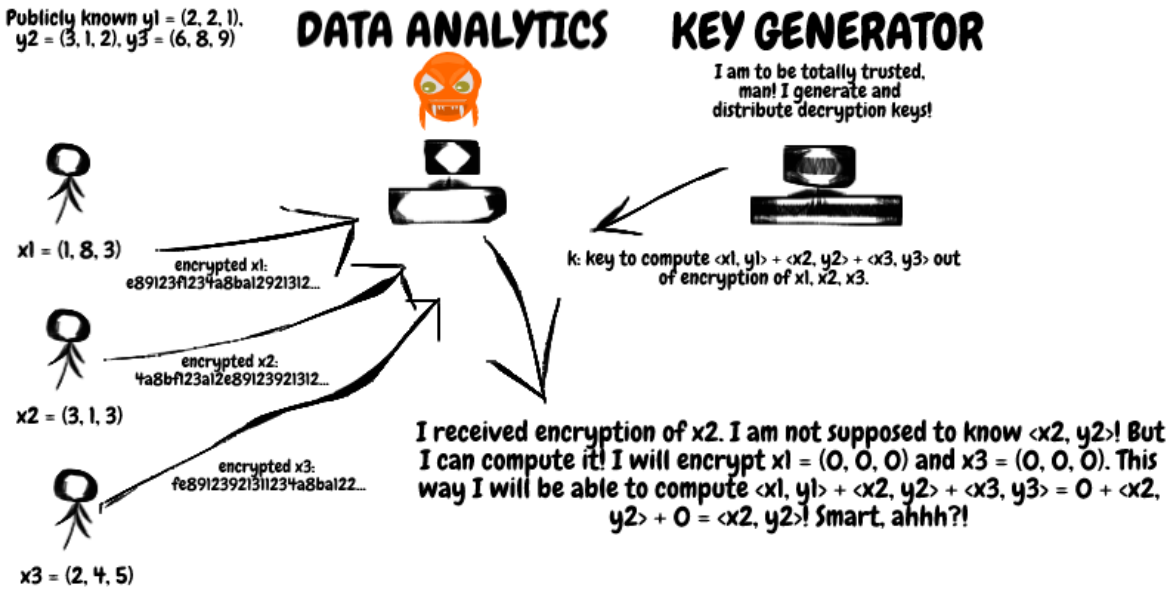

Figure 7: Evil Data Analytics subject

$+ < x_3, y_3 >= 0 + < x_2, y_2 > +0 =< x_2, y_2 >$. This way, Data Analytics would be able to extract information about each particular User, which it should not.

3.1 Multi-Input Functional Encryption for Inner Products: Function-Hiding Realizations and Constructions without Pairings

One of the first practical multi-input inner-product schemes was designed in the paper "Multi-Input Functional Encryption for Inner Products: Function-Hiding Realizations and Constructions without Pairings" by Abdalla et al. [12]. The paper presented how different single-input inner-product schemes can be turned into multi-input inner-product schemes. Similarly, GoFE and CiFEr implementation of the multi-input scheme can be instantiated by using different underlying single-input inner-product implementations (*simple.DDHMulti* uses *simple.DDH*, *fullysec.DamgardMulti* uses *fullysec.Damgard*). As the API is almost identical, we present only *fullysec.DamgardMulti*.

3.1.1 GoFE API

Generator

First, Generator needs to instantiate the scheme and generate the master keys:

```
var damgardMulti *fullysec.DamgardMulti
damgardMulti, err = fullysec.NewDamgardMultiPrecomp(numClients, l, param.
    ↪ modulusLength, bound)
secKeys, err := damgardMulti.GenerateMasterKeys()
```

Note that *secKeys* contains secret keys for all Users. However, the i-th User should be given only the i-th part: *secKeys.Mpk[i]* and *secKeys.Otp[i]*.

Vectors y_i are to be represented in a matrix.

```
y, err := data.NewRandomMatrix(numClients, l, sampler)
```

The functional decryption key is derived as:

```
derivedKey, err := damgardMulti.DeriveKey(secKeys, y)
```

User

User instantiates the scheme with the same parameters as Key Generator and encrypts vector x:

```
encryptor, err = fullysec.NewDamgardMultiPrecomp(numClients, l, param.
    ↪ modulusLength, bound)
c, err := encryptor.Encrypt(x, secKeys.Mpk[i], secKeys.Otp[i])
```

Data Analytics

Data Analytics instantiates the scheme with the same parameters as Key Generator and User. It decrypts value $< x_1, y_1 > + ... < x_n, y_n >$ as:

```
encryptor, err = fullysec.NewDamgardMultiPrecomp(numClients, l, param.
    ↪ modulusLength, bound)
xy, err := decryptor.Decrypt(ciphertexts, derivedKey, y)
```

Note that *ciphertexts* contains all ciphertexts from all Users.

3.1.2 CiFEr API

Generator

First, Generator needs to instantiate the scheme and generate the master keys:

```
cfe_damgard_multi m;
err = cfe_damgard_multi_precomp_init(&m, num_clients, l, modulus_len, bound);
cfe_mat mpk;
cfe_damgard_multi_sec_key msk;
cfe_damgard_multi_master_keys_init(&mpk, &msk, &m);
cfe_damgard_multi_generate_master_keys(&mpk, &msk, &m);
```

Note that *msk* contains secret keys for all Users. However, the i-th User should be given only the i-th part.

Vectors y_i are to be represented in a matrix.

```
cfe_uniform_sample_mat(&y, bound);
```

The functional decryption key is derived as:

```
cfe_damgard_multi_fe_key fe_key;
cfe_damgard_multi_fe_key_init(&fe_key, &m);
err = cfe_damgard_multi_derive_fe_key(&fe_key, &m, &msk, &y);
```

User

User instantiates the scheme with the same parameters as Key Generator and encrypts vector x:

```
cfe_damgard_multi encryptor;
err = cfe_damgard_multi_precomp_init(&encryptor, num_clients, l, modulus_len,
    ↪ bound);
err = cfe_damgard_multi_encrypt(&ciphertext, &encryptor, x, pub_key, otp);
```

Data Analytics

Data Analytics instantiates the scheme with the same parameters as Key Generator and User. It decrypts value $< x_1, y_1 > + ... < x_n, y_n >$ as:

```
cfe_damgard_multi decryptor;
err = cfe_damgard_multi_precomp_init(&decryptor, num_users, l, modulus_len,
    ↪ bound);
err = cfe_damgard_multi_decrypt(xy, &decryptor, ciphertext, &fe_key, &y);
```

Note that *ciphertexts* contains all ciphertexts from all Users.

4 Quadratic Schemes

Single-input inner-product and multi-input inner-product schemes have been presented in the previous two sections. But – is inner-product all we can do with functional encryption?

It is not. Practical functional encryption schemes for quadratic polynomials exist too.

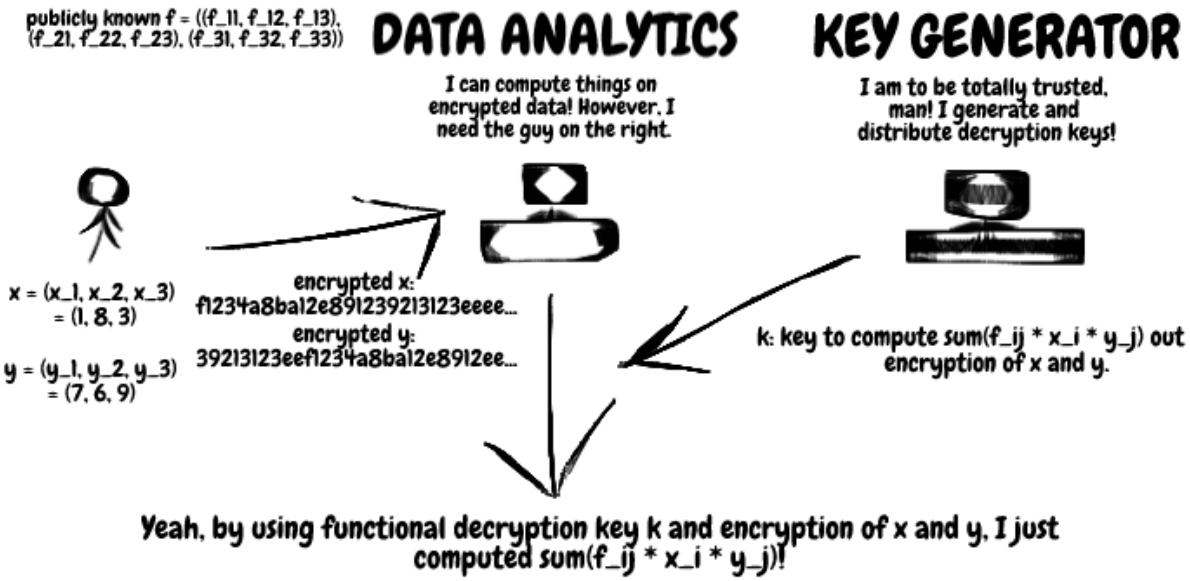

Figure 8: Quadratic scheme

Inner-products are linear polynomials. We take a vector x and compute the function $f(x) = y_1 \cdot x_1 + y_2 \cdot x_2 + y_3 \cdot x_3$, where y is a constant vector.

In the case of quadratic polynomials, we take a vector x and compute the function $f(x) = \sum f_{ij} \cdot x_i \cdot x_j$, where f is a constant matrix.

Also, the function above can be generalized: we take vectors x and y, and compute the function $f(x, y) = \sum f_{ij} \cdot x_i \cdot y_j$, where f is a constant matrix.

In quadratic polynomial schemes, the holder of the functional decryption key can compute $f(x, y)$ by having encryptions of x and y.

4.1 Reading in the Dark: Classifying Encrypted Digits with Functional Encryption

One of the first practical quadratic schemes was designed in the paper "Reading in the Dark: Classifying Encrypted Digits with Functional Encryption" by Dufour Sans et al. [25]. The paper also presents how to use the quadratic polynomial scheme for privacy-enhanced machine learning. In particular, how two-layer neural network model can be applied on the encrypted data (we cover this later in the document, see 9.4).

4.1.1 GoFE API

Generator

First, Generator needs to instantiate the scheme and generate the master keys:

```
q := quadratic.NewSGP(n, bound)
msk, err := q.GenerateMasterKey()
```

The parameter n defines the length of vectors x and y. Matrix f is of dimensions nxn. The parameter *bound* defines the value by which elements of vectors x, y, and matrix f are bounded.

Note that the scheme uses bilinear pairings and the same group is always used. Thus the subjects involved in the protocol always use the same group and there is no need to take special care for specifying the scheme group.

The functional decryption key is derived as:

```
key, err := q.DeriveKey(msk, f)
```

User

User instantiates the scheme and encrypts vector x:

```
encryptor := quadratic.NewSGP(n, bound)
c, err := encryptor.Encrypt(x, y, msk)
```

Data Analytics

Data Analytics instantiates the scheme and decrypts the value $< x_1, y_1 > + ... < x_n, y_n >$ as:

```
decryptor := quadratic.NewSGP(n, bound)
dec, err := decryptor.Decrypt(c, key, f)
```

The value dec is $f(x, y) = \sum f_{ij} \cdot x_i \cdot y_j$.

4.1.2 CiFEr API

Generator

First, Generator needs to instantiate the scheme and generate the master keys:

```
cfe_sgp s;
err = cfe_sgp_init(&s, l, b);
cfe_sgp_sec_key msk;
cfe_sgp_sec_key_init(&msk, &s);
cfe_sgp_generate_sec_key(&msk, &s);
```

The parameter l defines the length of vectors x and y. Matrix f is of dimensions lxl. The parameter b defines the value by which elements of vectors x, y, and the matrix f are bounded.

Note that the scheme uses bilinear pairings andthe same group is always used. Thus the subjects involved in the protocol always use the same group and there is no need to take special care for specifying the scheme group.

The functional decryption key is derived as:

```
cfe_mat f;
cfe_mat_init(&f, l, l);
cfe_uniform_sample_range_mat(&f, b_neg, b);
ECP2_BN254 key;
err = cfe_sgp_derive_fe_key(&key, &s, &msk, &f);
```

Note that for demonstration purposes a random matrix f is used.

User

User instantiates the scheme and encrypts vector x:

```
cfe_sgp encryptor;
err = cfe_sgp_init(&encryptor, l, b);
cfe_sgp_cipher cipher;
cfe_sgp_cipher_init(&cipher, &encryptor);
err = cfe_sgp_encrypt(&cipher, &encryptor, &x, &y, &msk);
```

Data Analytics

Data Analytics instantiates the scheme and decrypts the value $< x_1, y_1 > + \ldots < x_n, y_n >$ as:

```
cfe_sgp decryptor;
err = cfe_sgp_init(&decryptor, l, b);
mpz_t dec;
mpz_init(dec);
cfe_sgp_decrypt(dec, &decryptor, &cipher, &key, &f);
```

The value dec is $f(x, y) = \sum f_{ij} \cdot x_i \cdot y_j$.

4.2 A New Paradigm for Public-Key Functional Encryption for Degree-2 Polynomials

A New Paradigm for Public-Key Functional Encryption for Degree-2 Polynomials by Gay [20] exhibits a new paradigm to build quadratic schemes. It introduces and uses partially function-hiding schemes for inner-products, a primitive that bypasses impossibility results of public-key function-hiding functional encryption schemes. This gives stronger, desirable security guarantees that were previously not achieved. Also, partially function-hiding schemes can be used in GoFE and CiFEr as standalone schemes too.

4.2.1 GoFE API

Generator

First, Generator needs to instantiate the scheme and generate the master keys:

```
q, err := quadratic.NewQuad(n, m, bound)
pubKey, secKey, err := q.GenerateKeys()
```

The parameter n defines the length of vectors x, m defines the length of vectors y. Matrix f is of dimensions nxn. The parameter *bound* defines the value by which elements of vectors x, y, and the matrix f are bounded.

The functional decryption key is derived as:

```
f, err := data.NewRandomMatrix(n, m, sampler)
feKey, err := q.DeriveKey(secKey, f)
```

User

User instantiates the scheme and encrypts the vector x:

```
encryptor := quadratic.NewQuadFromParams(q.Params)
c, err := encryptor.Encrypt(x, y, pubKey)
```

Data Analytics

Data Analytics instantiates the scheme and decrypts the value $< x_1, y_1 > + ... < x_n, y_n >$ as:

```
decryptor := quadratic.NewQuadFromParams(q.Params)
dec, err := decryptor.Decrypt(c, feKey, f)
```

The value dec is $f(x, y) = \sum f_{ij} \cdot x_i \cdot y_j$.

5 Decentralized Inner-Product

In Section 2, we briefly discussed why Key Generator is needed and some ways to avoid the necessity of it. In particular, we could delegate Key Generator responsibility to one of the Users. However, this might not be accepted by the other Users. Could the responsibility be divided among all Users?

In fact, it can be. The two papers presented below apply the techniques to enable Users to generate key shares by themselves. Only a subject which obtains key shares from all Users can combine them into a functional decryption key. Thus, the authority is removed and the Users work together to generate appropriate functional decryption keys. Note that the authority is not simply distributed to a larger number of parties, but that the resulting protocol is indeed decentralized: each User has complete control over their individual data and the functional keys they authorize the generation of.

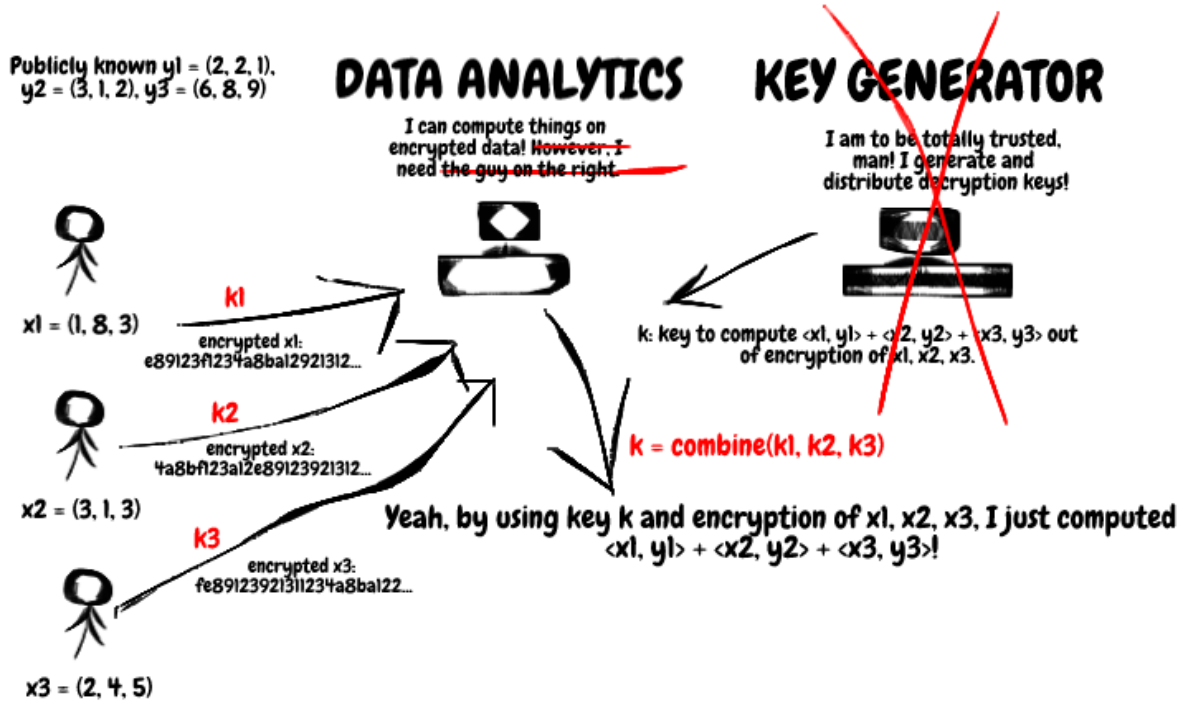

Figure 9: Decentralized scheme

5.1 Decentralized Multi-Client Functional Encryption for Inner Product

One of the first practical decentralized schemes was designed in the paper "Decentralized Multi-Client Functional Encryption for Inner Product" by Chotard et al. [16].

A special technique enables the generation of key shares without heavy communication between the involved Users. Users only need to know public keys of each other and then use their own secret keys to generate the shares.

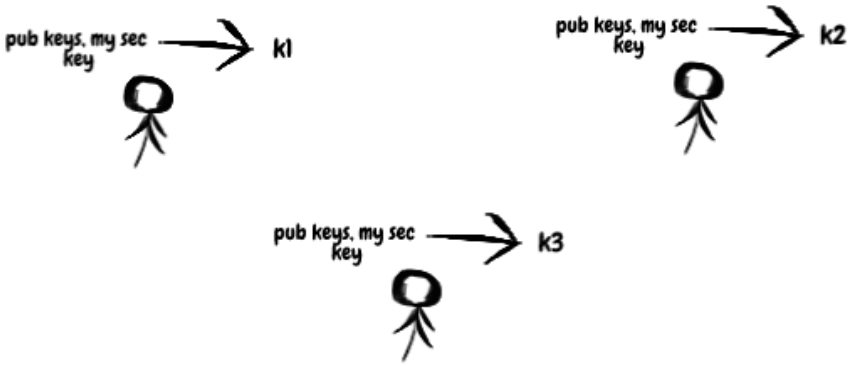

Figure 10: Generation of key shares

5.1.1 GoFE API

User

User first instantiates the *DMCFEClient*. Note that each Users has its own index i, which is passed into the constructor.

```
c, err := fullysec.NewDMCFEClient(i)
```

When the scheme is instantiated, a public key is generated as well. Each User needs to collect all public keys.

Public keys are passed into *SetShare* method to prepare the parameters that will be needed to compute the key shares for different vectors y.

```
err := c.SetShare(pubKeys)
```

User then encrypts the vector x under some label (of type string) and derives the key share for vector y (to compute the inner-product of x and y) to be passed to Data Analytics.

```
c, err := c.Encrypt(x, label)
keyShare, err := c.DeriveKeyShare(y)
```

Data Analytics

Data Analytics calls the function *DMCFEDecrypt*. All ciphertexts and key shares need to be passed in. Also, the label under which the vectors have been encrypted is needed.

```
d, err := fullysec.DMCFEDecrypt(ciphers, keyShares, y, label, bound)
```

5.1.2 CiFEr API

User

User first instantiates *cfe_dmfce_client*. Note that each Users has its own index i.

```
cfe_dmcfe_client c;
cfe_dmcfe_client_init(&c, i);
```

Public keys are passed into the *cfe_dmcfe_set_share* method to prepare the parameters that will be needed to compute the key shares for different vectors y.

```
cfe_dmcfe_set_share(&c, pub_keys, num_clients);
```

User then encrypts the vector x under some label (of type string) and derives the key share for vector y (to compute the inner-product of x and y) to be passed to Data Analytics.

```
cfe_dmcfe_encrypt(&cipher, &clients, x, label, label_len);
cfe_dmcfe_fe_key_part_init(&fe_key);
cfe_dmcfe_derive_fe_key_part(&fe_key, &c, &y);
```

Data Analytics

Data Analytics calls the function *cfe_dmcfe_decrypt*. All ciphertexts and key shares (*fe_key*) need to be passed in. Also, the label under which the vectors have been encrypted is needed.

```
cfe_error err = cfe_dmcfe_decrypt(xy, ciphers, fe_key, label, label_len, &y,
    ↪ bound);
```

5.2 Decentralizing Inner-Product Functional Encryption

In this section, we present a scheme designed in the paper "Decentralizing Inner-Product Functional Encryptiont" by Abdalla et al. [10]. This scheme does not require bilinear pairings and is thus faster than the scheme described in the previous section.

5.2.1 GoFE API

User

First, the *DamgardMulti* needs to be instantiated.

```
damgardMulti, err = fullysec.NewDamgardMultiPrecomp(numOfClients, l, param.
    ↪ modulusLength, bound)
```

DamgardMulti represents a multi-input variant of the underlying scheme based on [12].

User then instantiates the *DamgardDecMultiClient*. Note that each Users has its own index i, which is passed into the constructor.

```
c, err = fullysec.NewDamgardDecMultiClient(i, damgardMulti)
```

Each User prepares a key share out of the public keys and creates its own secret key for the encryption of a vector.

```
err = c.SetShare(pubKeys)
secKey, err = c.GenerateKeys()
```

User then encrypts the vector x and derives the key share for vector y (to compute the inner-product of x and y) to be passed to Data Analytics.

```
c, err := c.Encrypt(x, secKey)
partKey, err = c.DeriveKeyShare(secKey, y)
```

Data Analytics

Data Analytics instantiates *DamgardDecMultiDec*. All ciphertexts and key shares need to be passed in the decryption method.

```
decryptor := fullysec.NewDamgardDecMultiDec(damgardMulti)
xy, err := decryptor.Decrypt(ciphertexts, partKeys, y)
```

5.2.2 CiFEr API

User

First, *damgard_multi* needs to be instantiated:

```
cfe_damgard_multi damgard_multi;
err = cfe_damgard_multi_precomp_init(&damgard_multi, num_clients, l, modulus_len
    ↪ , bound);
```

Struct *damgard_multi* represents a multi-input variant of the underlying scheme based on [12].

User then instantiates *cfe_damgard_dec_multi_client*. Note that each Users has its own index i, which is passed into the constructor.

```
cfe_damgard_dec_multi_client c;
cfe_damgard_dec_multi_client_init(&c, &damgard_multi, i);
```

Each User prepare a share key out of the public keys and creates its own secret key for the encryption of a vector.

```
cfe_damgard_dec_multi_client_set_share(&c, pub_keys);
cfe_damgard_dec_multi_generate_keys(&sec_key, &c);
```

User then encrypts the vector x and derives the key share for vector y (to compute the inner-product of x and y) to be passed to Data Analytics.

```
err = cfe_damgard_dec_multi_encrypt(&cipher, &x, &sec_key, &c);
cfe_damgard_dec_multi_fe_key_share_init(&fe_key);
err = cfe_damgard_dec_multi_client_derive_fe_key_part(&fe_key, &y, &sec_key, &c)
    ↪ ;
```

Data Analytics

Data Analytics instantiates *cfe_damgard_dec_multi_dec*. All ciphertexts and key shares need to be passed in the decryption method.

```
cfe_damgard_dec_multi_dec decryptor;
cfe_damgard_dec_multi_dec_init(&decryptor, &damgard_multi);
err = cfe_damgard_dec_multi_decrypt(xy, ciphers, fe_key, &y, &decryptor);
```

6 Function-Hiding Single-Input Inner-Product

Message confidentiality is not always sufficient. Sometimes, functions contain sensitive information and require to be hidden too.

An example scenario could be a hospital that uses an external cloud server for storing medical records of the patients. To enable computations on the records, the hospital can delegate functional decryption keys to the cloud server. Thus, the data remains confidential. But what about the confidentiality of the function? The function could reveal sensitive data too. For example, the function could compute the list of all patients who are receiving treatment for a certain disease. Without function confidentiality, the cloud server could see the list in the clear.

Function-hiding schemes guarantee the privacy of the function as well as the privacy of the data. In what follows, we present one of the first practical function-hiding schemes.

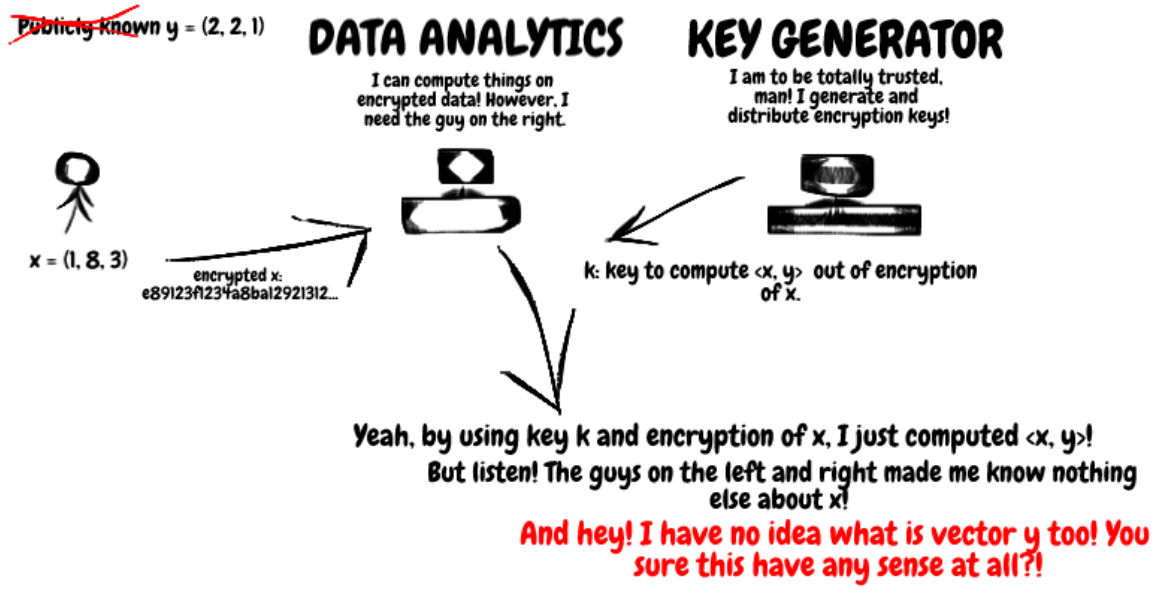

Figure 11: Function-Hiding scheme

6.1 Function-Hiding Inner Product Encryption is Practical

The paper "Function-Hiding Inner Product Encryption is Practical" by Kim et al. [22] presents a practical function-hiding single-input inner-product scheme. That means, vector y that defines the function remains hidden to the Data Analytics subject.

6.1.1 GoFE API

Key Generator

Key Generator first instantiates the scheme.

The parameter l presents the length of vectors.

The parameters *boundX* and *boundY* define the bounds of vector coordinates.

Key Generator then establishes the master keys.

```
fhipe, err := fullysec.NewFHIPE(l, boundX, boundY)
masterSecKey, err := fhipe.GenerateMasterKey()
```

Finally, it derives the functional key for vector y:

```
key, err := fhipe.DeriveKey(y, masterSecKey)
```

User

User instantiates the scheme and encrypts the vector x:

```
encryptor := fullysec.NewFHIPEFromParams(fhipe.Params)
ciphertext, err := encryptor.Encrypt(x, masterSecKey)
```

Data Analytics

Data Analytics decrypts the inner-product of x and y without knowing x and y:

```
decryptor := fullysec.NewFHIPEFromParams(fhipe.Params)
xy, err := decryptor.Decrypt(ciphertext, key)
```

6.1.2 CiFEr API

Key Generator

Key Generator first instantiates the scheme.

The parameter *l* presents the length of vectors.

The parameters *bound_x* and *bound_y* define the bounds of vector coordinates.

Key Generator then establishes the master keys.

```
cfe_fhipe fhipe;
cfe_error err= cfe_fhipe_init(&fhipe, l, bound_x, bound_y);
cfe_fhipe_sec_key sec_key;
cfe_fhipe_master_key_init(&sec_key, &fhipe);
err = cfe_fhipe_generate_master_key(&sec_key, &fhipe);
```

Finally, it derives the functional key for vector y:

```
cfe_fhipe_fe_key FE_key;
cfe_fhipe_fe_key_init(&FE_key, &fhipe);
err = cfe_fhipe_derive_fe_key(&FE_key, &y, &sec_key, &fhipe);
```

User

User instantiates the scheme and encrypts the vector x:

```
cfe_fhipe encryptor;
cfe_fhipe_copy(&encryptor, &fhipe);
cfe_fhipe_ciphertext cipher;
cfe_fhipe_ciphertext_init(&cipher, &encryptor);
err = cfe_fhipe_encrypt(&cipher, &x, &sec_key, &encryptor);
```

Data Analytics

Data Analytics decrypts the inner-product of x and y without knowing x and y:

```
cfe_fhipe decryptor;
cfe_fhipe_copy(&decryptor, &fhipe);
err = cfe_fhipe_decrypt(xy, &cipher, &FE_key, &decryptor);
```

7 Function-Hiding Multi-Input Inner-Product

The number of possible applications of functional encryption greatly increases if functions over data from multiple Users are enabled. This way, different aggregation functions over multiple Users are enabled for Data Analysis and at the same time, the privacy of each particular User is provided.

The previous section presented a function-hiding single-input inner-product scheme. In what follows, we present a multi-input function-hiding scheme.

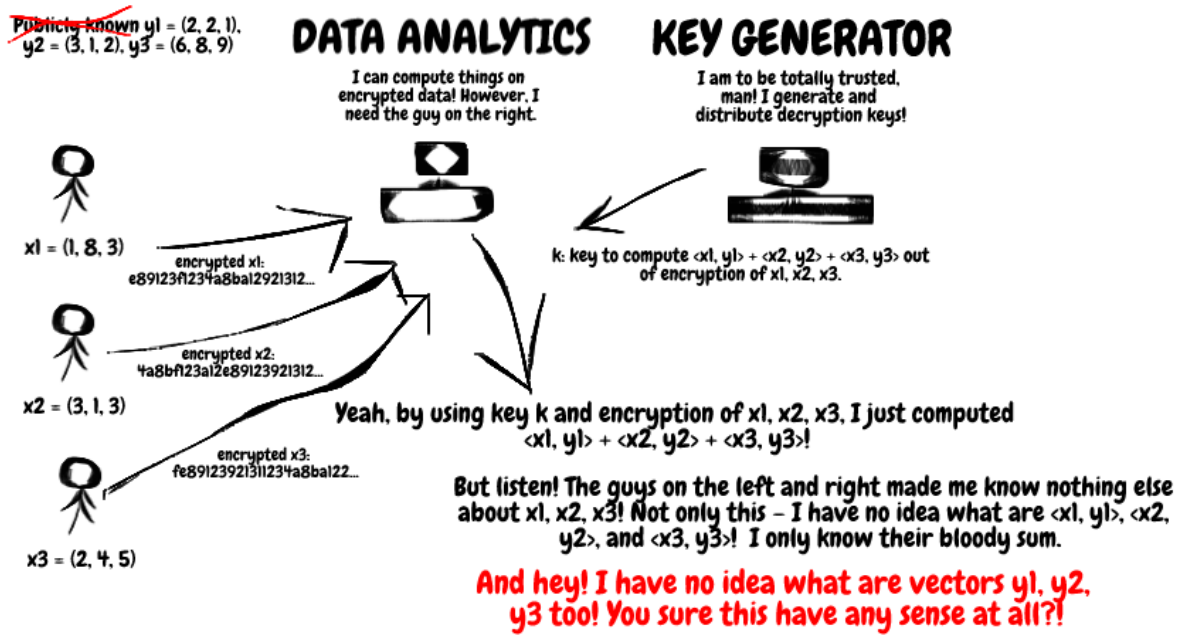

Figure 12: Function-Hiding Multi-Input scheme

7.1 Full-Hiding (Unbounded) Multi-Input Inner Product Functional Encryption from the k-Linear Assumption

The paper "Full-Hiding (Unbounded) Multi-Input Inner Product Functional Encryption from the k-Linear Assumption" by Datta et al. [18] presents two non-generic and practically efficient private key multi-input functional encryption (MIFE) schemes for the multi-input inner-product functionality that are the first to achieve simultaneous message and function privacy.

7.1.1 GoFE API

Key Generator

Key Generator first instantiates the scheme.

The parameter *secLevel* defines the level of security.

The parameter *numClient* defines the number of Users.

The parameter *vecLen* defines the length of the vectors.

The parameters *boundX* and *boundY* define the bounds of vector coordinates.

Key Generator then establishes the master keys.

```
fhmulti := fullysec.NewFHMultiIPE(secLevel, numClient, vecLen, boundX, boundY)
masterSecKey, pubKey, err := fhmulti.GenerateKeys()
```

A functional key for y is then derived which stores the vectors defining the function:

```
key, err := fhmulti.DeriveKey(y, masterSecKey)
```

User

Each Users has its own index i. User instantiates the scheme and encrypts the vector x:

```
encryptor = fullysec.NewFHMultiIPEFromParams(fhmulti.Params)
cipher, err = encryptor.Encrypt(x, masterSecKey.BHat[i])
```

Data Analytics

Data Analytics decrypts the inner-product by passing the ciphertexts from all Users (contained in matrix *cipher*) to the *Decrypt* method.

```
decryptor := fullysec.NewFHMultiIPEFromParams(fhmulti.Params)
xy, err := decryptor.Decrypt(cipher, key, pubKey)
```

7.1.2 CiFEr API

Key Generator

Key Generator first instantiates the scheme.

The parameter *sec_level* defines the level of security.

The parameter *num_clients* defines the number of Users.

The parameter *vec_len* defines the length of the vectors.

The parameters *bound_x* and *bound_y* define the bounds of vector coordinates.

Key Generator then establishes the master keys.

```
cfe_fh_multi_ipe fh_multi_ipe;
cfe_error err= cfe_fh_multi_ipe_init(&fh_multi_ipe, sec_level, num_clients,
    ↪ vec_len, bound_x, bound_y);
```

Finally, it derives the functional key for y which stores the vectors defining the function:

```
cfe_fh_multi_ipe_sec_key sec_key;
cfe_fh_multi_ipe_master_key_init(&sec_key, &fh_multi_ipe);
FP12_BN254 pub_key;
err = cfe_fh_multi_ipe_generate_keys(&sec_key, &pub_key, &fh_multi_ipe);
```

User

Each Users has its own index i. User instantiates the scheme and encrypts the vector x. Note that the parameters used for instantiation need to be the same as for Key Generator. If both subjects run in the same process, *cfe_fh_multi_ipe_copy* can be used.

```
cfe_fh_multi_ipe c;
cfe_fh_multi_ipe_copy(&c, &fh_multi_ipe);
err = cfe_fh_multi_ipe_encrypt(&cipher, &x, &sec_key.B_hat[i], &c);
```

Data Analytics

Data Analytics decrypts the inner-product by passing the ciphertexts from all Users (contained in matrix *ciphers*) to the *cfe_fh_multi_ipe_decrypt* method.

```
cfe_fh_multi_ipe decryptor;
cfe_fh_multi_ipe_copy(&decryptor, &fh_multi_ipe);
err = cfe_fh_multi_ipe_decrypt(xy, ciphers, &FE_key, &pub_key, &decryptor);
```

8 Attribute-Based Encryption

Attribute-based encryption (ABE) extends the concept of public-key cryptography. In traditional public-key cryptography, a message is encrypted for a specific receiver using the receiver's public-key. ABE defines identity as a set of attributes and messages can be encrypted with respect to subsets of attributes. A subject should be able to decrypt a ciphertext only if it holds a key with "matching attributes". User keys are always issued by some trusted party.

There are two types of ABE. In Key-Policy ABE (KP-ABE), the ciphertexts are associated with sets of attributes, whereas user secret keys are associated with policies. In Ciphertext-Policy ABE (CP-ABE), user keys are associated with sets of attributes, whereas ciphertexts are associated with policies.

The next two subsections present the implementation of a KP-ABE and a CP-ABE scheme in GoFE and CiFEr. The last subsection presents the implementation of the multi-authority policy-hiding ABE scheme.

8.1 Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data (KP-ABE)

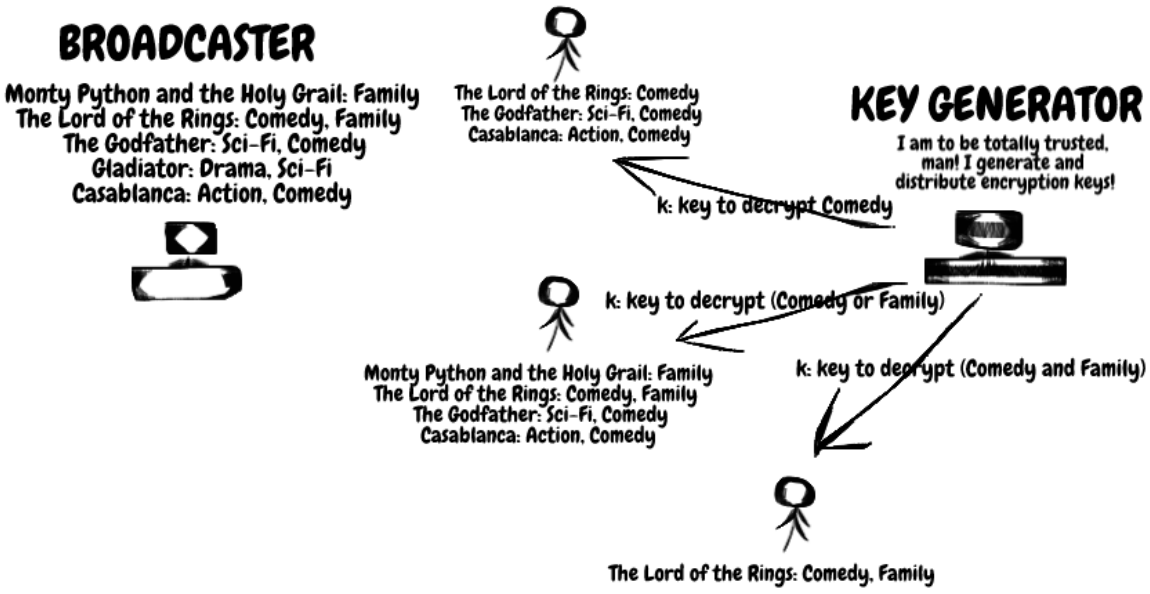

Figure 13: KP-ABE scheme

The paper "Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data" by Goyal et al. [21] presented a first KP-ABE scheme. Also, the paper demonstrates the applicability of the scheme to broadcast encryption. There, a Broadcaster broadcasts a sequence of different items (like movies), each one labeled with a set of attributes describing the item. For instance, movies might be labeled with attributes such as "Family", "Drama", "Comedy", "Action".

Each User is subscribed to a different "package". The package describes an access policy, which along with the set of attributes describing any particular item being broadcast, determines whether or not the User should be able to access the item. For example, a User may want to subscribe to a package that

allows to view movies (“Crime” and (“Action” or “Sci-Fi”)). That means the User can view movies that are labeled with either "Crime, Action" or "Crime, Sci-Fi".

8.1.1 GoFE API

Key Generator

Key Generator first instantiates the scheme. The parameter l defines the number of attributes. Each attribute needs to bo mapped to a number between 0 and $l - 1$.

Key Generator then establishes the master keys.

```
l := 10
a := abe.NewGPSW(l)
pubKey, secKey, err := a.GenerateMasterKeys()
```

Let us say that User paid to watch movies "Drama or Comedy". If attribute "Drama" is mapped to 1 and attribute "Comedy" is mapped to 3, Key Generator executes the following:

```
msp, err := abe.BooleanToMSP("(1 OR 3)", true)
abeKey, err := abe.GenerateKey(msp, secKey)
```

BooleanToMSP takes as an input a boolean expression (without a NOT gate) and outputs a MSP (Monotone Span Program) structure representing the expression. The MSP structure is then passed to the *GenerateKey* function.

The User is then given the *abeKey*.

Encryptor

Encryptor instantiates the scheme and encrypts the item. For example, Broadcaster encrypts the movie using the attributes describing it (3 = "Comedy", 4 = "Sci-Fi")

```
l := 10
a := abe.NewGPSW(l)
gamma := []int{3, 4}
cipher, err := a.Encrypt(movie, gamma, pubKey)
```

User

User decrypts the item using the key obtained from Key Generator. For example, the User that possesses the key for "Drama or Comedy" can decrypt the movie that was encrypted by attributes "Drama", "Sci-Fi".

```
l := 10
a := abe.NewGPSW(l)
movie, err := a.Decrypt(cipher, abeKey)
```

8.1.2 CiFEr API

Key Generator

Key Generator first instantiates the scheme. The parameter *l* defines the number of attributes. Each attribute needs to bo mapped to a number between *0* and *l-1*.

Key Generator then establishes the master keys.

```
cfe_gpsw gpsw;
cfe_gpsw_init(&gpsw, 10);
cfe_gpsw_pub_key pk;
cfe_vec sk;
cfe_gpsw_master_keys_init(&pk, &sk, &gpsw);
cfe_gpsw_generate_master_keys(&pk, &sk, &gpsw);
```

Let us say that User paid to watch movies "Drama or Comedy". If attribute "Drama" is mapped to 1 and attribute "Comedy" is mapped to 3, the Key Generator executes the following:

```
cfe_gpsw_cipher cipher;
cfe_gpsw_cipher_init(&cipher, 10);
char bool_exp[] = "(1 OR 3)";
size_t bool_exp_len = 8; // length of the boolean expression string
cfe_msp msp;
cfe_error err = cfe_boolean_to_msp(&msp, bool_exp, bool_exp_len, true);

cfe_gpsw_keys abe_key;
cfe_gpsw_keys_init(&abe_key, &msp);
cfe_gpsw_generate_key(&abe_key, &msp, &sk);
```

The User is then given the *abe_key*.

Encryptor

Encryptor instantiates the scheme and encrypts the item. For example, Broadcaster encrypts the movie using the attributes describing it (3 = "Comedy", 4 = "Sci-Fi")

```
cfe_gpsw gpsw;
cfe_gpsw_init(&gpsw, 10);
cfe_gpsw_cipher cipher;
cfe_gpsw_cipher_init(&cipher);
int gamma[] = {3, 4};
cfe_gpsw_encrypt(&cipher, &gpsw, &movie, gamma, &pk);
```

User

User decrypts the item using the key obtained from Key Generator. For example, the User that possesses the key for "Drama or Comedy" can decrypt the movie that was encrypted by attributes "Drama", "Sci-Fi".

```
cfe_gpsw gpsw;
cfe_gpsw_init(&gpsw, 10);
```

```
FP12_BN254 decrypted;
cfe_error check = cfe_gpsw_decrypt(&decrypted, &cipher, &abe_key, &gpsw);
```

8.2 FAME: Fast Attribute-based Message Encryption (CP-ABE)

In CP-ABE schemes, the policy who can decrypt the item is set on a ciphertext. Imagine an Army encrypting a message that is intended only for soldiers that are either Lieutenants or they served in Atropia. Such a message is encrypted with a policy "Lieutenant or Atropia". Only persons who possess keys with attributes "Lieutenant" or "Atropia" (or both) can decrypt the message.

Or, an Army could encrypt a message with a policy "General and Gorgas". This would mean only generals that served in Gorgas can decrypt the message.

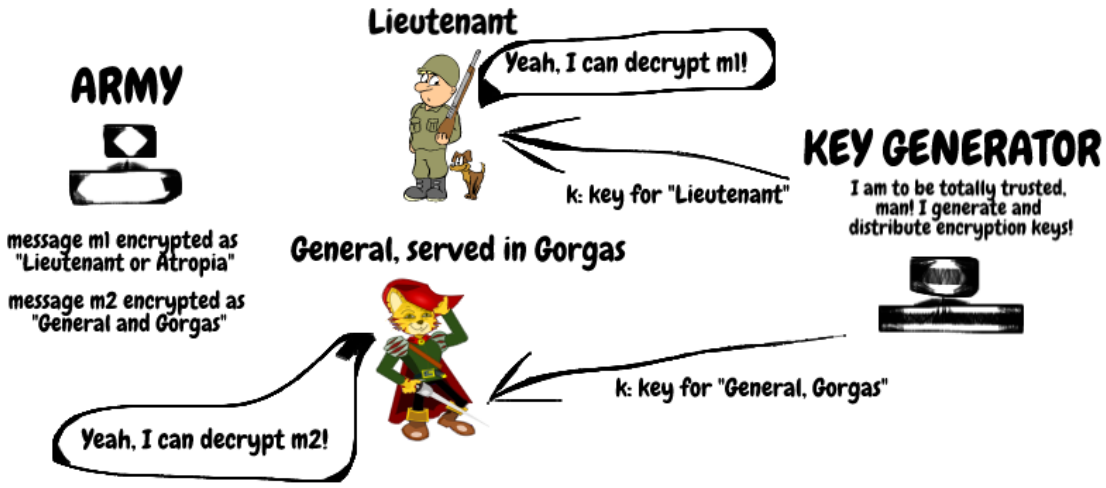

Figure 14: CP-ABE scheme

The paper "FAME: Fast Attribute-based Message Encryption" by Agrawal et al. [14] proposed the first fully secure CP-ABE and KP-ABE schemes based on a standard assumption on pairing groups, which do not put any restriction on policy type or attributes. GoFE and CiFEr offers API to the CP-ABE FAME version.

8.2.1 GoFE API

Key Generator

Key Generator first instantiates the scheme and generates the master keys.

```
a := abe.NewFAME()
pubKey, secKey, err := a.GenerateMasterKeys()
```

It then generates keys for the Users. For example, if User possesses attributes 2 and 5:

```
gamma := []int{2, 5}
abeKey, err := a.GenerateAttribKeys(gamma, secKey)
```

Encryptor

Encryptor creates a *msp* structure out of a boolean expression representing the policy specifying which attributes are needed to decrypt the ciphertext. It then encrypts the message with the decryption policy specified by the *msp* structure.

```
a := abe.NewFAME()
msp, err := abe.BooleanToMSP("((0 AND 1) OR (2 AND 3)) AND 5", false)
cipher, err := a.Encrypt(msg, msp, pubKey)
```

User

User holding a proper key can decrypt the message:

```
a := abe.NewFAME()
msg, err := a.Decrypt(cipher, abeKey, pubKey)
```

8.2.2 CiFEr API

Key Generator

Key Generator first instantiates the scheme and generates the master keys.

```
cfe_fame fame;
cfe_fame_init(&fame);
cfe_fame_pub_key pk;
cfe_fame_sec_key sk;
cfe_fame_sec_key_init(&sk);
cfe_fame_generate_master_keys(&pk, &sk, &fame);
```

It then generates keys for the Users. For example, if User possesses attributes 2 and 5:

```
int owned_attrib[] = {2, 5};
cfe_fame_attrib_keys keys;
cfe_fame_attrib_keys_init(&keys, 2); // the number of attributes needs to be
    ↪ specified
cfe_fame_generate_attrib_keys(&keys, owned_attrib, 2, &sk, &fame);
```

Encryptor

Encryptor creates a *msp* structure out of a boolean expression representing the policy specifying which attributes are needed to decrypt the ciphertext. It then encrypts the message with the decryption policy specified by the *msp* structure.

```
cfe_fame fame;
cfe_fame_init(&fame);
char bool_exp[] = "(5 OR 3) AND ((2 OR 4) OR (1 AND 6))";
size_t bool_exp_len = 36; // length of the boolean expression string
cfe_msp msp;
cfe_error err = cfe_boolean_to_msp(&msp, bool_exp, bool_exp_len, false);
```

```
cfe_fame_cipher cipher;
cfe_fame_cipher_init(&cipher, &msp);
cfe_fame_encrypt(&cipher, &msg, &msp, &pk, &fame);
```

User

User holding a proper key can decrypt the message:

```
cfe_fame fame;
cfe_fame_init(&fame);
FP12_BN254 decryption;
err = cfe_fame_decrypt(&decryption, &cipher, &keys, &fame);
```

8.3 Decentralized Policy-Hiding Attribute-Based Encryption with Receiver Privacy

What if multiple subjects share the items and each of them wants to manage the keys and policies?

For example, what if two intelligence agencies, like CIA and MI6, agreed on sharing some of the files, but they both want to control who is being given the keys and what these keys enable to decrypt?

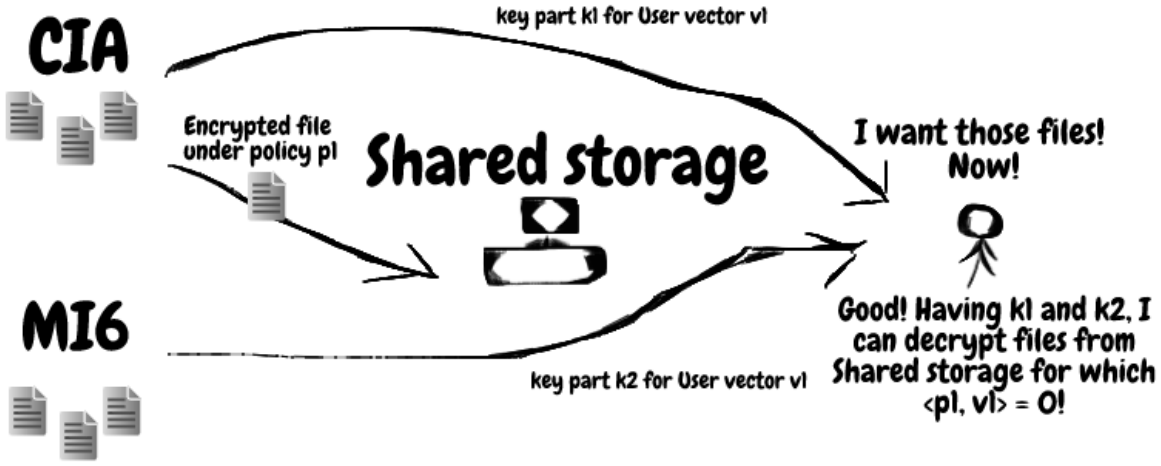

Figure 15: MA-ABE Policy-Hiding scheme

This could be solved by using "Decentralized Policy-Hiding Attribute-Based Encryption with Receiver Privacy" by Michalevsky and Joye [23].

Only User that obtains key parts from all of the authorities (CIA and MI6) can decrypt the file. Also, each key is generated using a policy which specifies the items that can be decrypted. In particular, the policy in [23] is given by a vector. User is assigned another vector (User's vector) when the key is derived; User is then able to decrypt only the items for which the policy vectors are orthogonal to User's vector. For example, by applying the policy, CIA and MI6 can control which files can be decrypted by the key.

Furthermore, by observing the ciphertext, it is not possible to determine what was the policy used when encrypting. That means, unauthorized parties cannot extract meta information, for example, for whom the item is intended or what it is about.

8.3.1 GoFE API

Authority

First, Authority instantiates the scheme. For each item to be encrypted, Authority defines the policy vector.

```
d, err := abe.NewDIPPE(3) // 3 is security parameter
auth, err = d.NewDIPPEAuth(i)
pubKeys = &auth.Pk
policyVec := data.Vector([]*big.Int{big.NewInt(1), big.NewInt(-1),
             big.NewInt(1), big.NewInt(0), big.NewInt(0)})
cipher, err := d.Encrypt(msg, policyVec, pubKeys)
```

When deriving the key for User, a unique ID needs to be chosen. To control which items will be decryptable by this key, User's vector is defined: only the items with policy vector orthogonal to the User's vector can be decrypted.

```
userGID := "someGID"
userVec := data.Vector([]*big.Int{big.NewInt(0), big.NewInt(1),
             big.NewInt(1), big.NewInt(-3), big.NewInt(4)})
userKey := make([]data.VectorG2, vecLen)
userKey, err = auth.DeriveKeyShare(userVec, pubKeys, userGID)
```

Note that User needs to obtain such a key (*userKey*) from each Authority.

User

The User possessing keys from all Authorities, can decrypt the item:

```
dec, err := d.Decrypt(cipher, userKeys, userVec, userGID)
```

8.3.2 CiFEr API

Scheme [23] can be transformed into an ABE scheme with the conjunction policy, where User must have all the specified attributes to be able to decrypt the item. In what follows, we give an example how to do it in CiFEr.

Authority

First, Authority instantiates the scheme.

```
cfe_dippe dippe;
cfe_dippe_init(&dippe, 2);

cfe_dippe_pub_key pk;
cfe_dippe_sec_key sk;
cfe_dippe_pub_key_init(&(pk), &dippe);
cfe_dippe_sec_key_init(&(sk), &dippe);
cfe_dippe_generate_master_keys(&(pk), &(sk), &dippe);
```

The length of attribute and policy vectors needs to be defined (*vlen*). Vector defining the conjunction policy needs to be defined. For example, pattern = 0, 1, 4 means 0-th, 1-st, and 4-th attributes are required. Policy vector is generated by using the *pattern*.

```
size_t vlen = 6;
size_t pattern[] = {0, 1, 4};
cfe_vec pv;
err = cfe_dippe_conjunction_policy_vector_init(&pv, &dippe, (vlen - 1), pattern,
    ↪  (sizeof(pattern) / sizeof(size_t)));
```

The item is encrypted with the chosen conjunction policy:

```
cfe_dippe_cipher cipher;
cfe_dippe_cipher_init(&cipher, &dippe, pv.size);

err = cfe_dippe_encrypt(&cipher, &dippe, pks, (sizeof(pks) / sizeof(
    ↪ cfe_dippe_pub_key *)), &pv, &msg);
```

Finally, the key is generated. Note that keys derived by patterns like 0, 1, 3, 4 and 0, 1, 4 will decrypt the message, while patterns like 0, 4 will not.

```
cfe_dippe_user_sec_key key;
cfe_dippe_user_sec_key_init(&sec_key, &dippe);
cfe_vec av;
size_t ap = {0, 1, 3, 4}
err = cfe_dippe_attribute_vector_init(&av, (vlen - 1), ap, aps_len[i]);
char gid[9];
printf(gid, "TESTGID%zu", i);
size_t gid_len = 8;
err = cfe_dippe_keygen(&key, &dippe, j, pks, (sizeof(pks) / sizeof(
    ↪ cfe_dippe_pub_key *)), &sk[(j & 1)], &av, gid, gid_len);
```

Note that User needs to obtain such a key (*userKey*) from each Authority.

User

The User possessing keys from all Authorities, can decrypt the item:

```
err = cfe_dippe_decrypt(&result, &dippe, (cfe_dippe_user_sec_key *) &usks, (
    ↪ sizeof(usks) / sizeof(cfe_dippe_user_sec_key)), &cipher, &av, gid,
    ↪ gid_len);
```

9 Showcases

In this section, we give real-world examples of how to build privacy-friendly services by using the FENTEC library.

9.1 Privacy-Friendly Prediction of Cardiovascular Diseases

Let us say Jim Edmonds would like to know how likely it is for him to develop cardiovascular diseases in the next thirty years. The risk can be computed by the Framingham Heart Study algorithm [19]. The Framingham algorithm takes parameters like sex, age, total cholesterol, high-density lipoprotein cholesterol, systolic blood pressure, treatment for hypertension, smoking, and diabetes status.

However, Jim does not want to reveal any of these parameters to some external services running the Framingham algorithm.

But there is a Company which offers all kinds of privacy-friendly prediction services. By connecting to their services, Jim can send his encrypted health status and obtain the risk evaluation. The Company never sees Jim's health parameters in the clear.

The source code of the demonstrator can be found at [1].

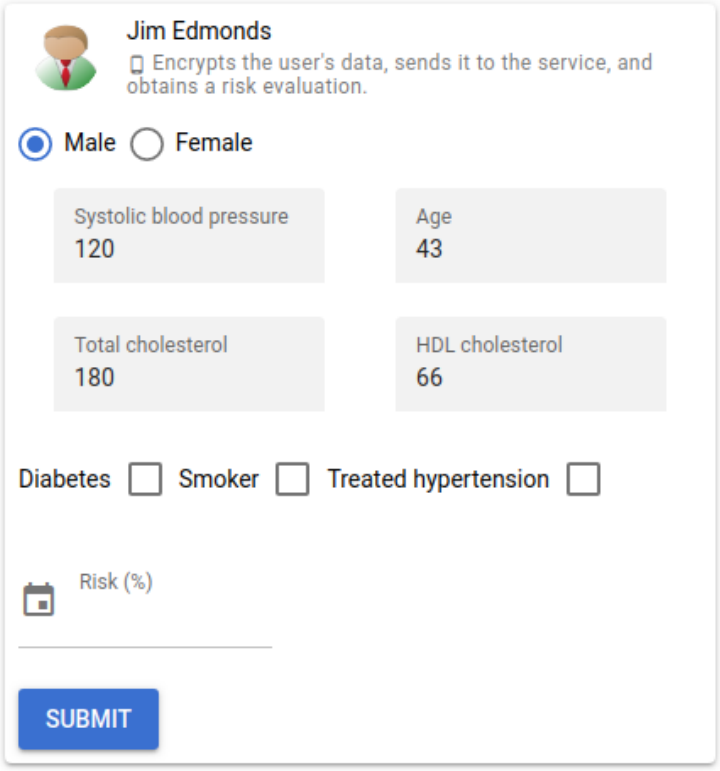

Figure 16: Parameters for cardiovascular disease risk computation

The demonstrator comprises the following components:

- Key Generator: central authority component which executes the setup procedure and generates functional keys.
- Company: running privacy-friendly prediction services: User sends encrypted data to the Company and obtains the risk of CVD. The Company obtains functional decryption keys from the Key Generator.
- User: obtains the public key from the Key Generator, encrypts parameters with the public key and sends it to the Company.

The Framingham heart study [19] followed roughly 5,000 patients from Framingham, Massachusettes, for many decades starting in 1948. In 1971, their offspring and spouses were included in the study, and several other cohorts of 5,000 since then. The risk models they computed are publicly available. These are multivariable risk algorithms used to assess the risk of specific atherosclerotic cardiovascular disease events, i.e., coronary heart disease, cerebrovascular disease, peripheral vascular disease, and heart failure. Algorithms most often estimate the 10-year or 30-year CVD risk of an individual. If a group of 100 persons all have a 20% 10-year risk of CVD it means that we should expect that 20 of these 100 individuals will develop CVD (for example coronary heart disease or stroke) in the next 10 years.

The demonstrator uses a functional encryption scheme based on the Paillier cryptosystem [15] due to its fast decryption operation. This way, Company is able to compute the risk score using only the encrypted values of the input parameters. User specifies the parameters, these are encrypted and sent to Company. Company computes the 30-year risk [24] and returns it to User.

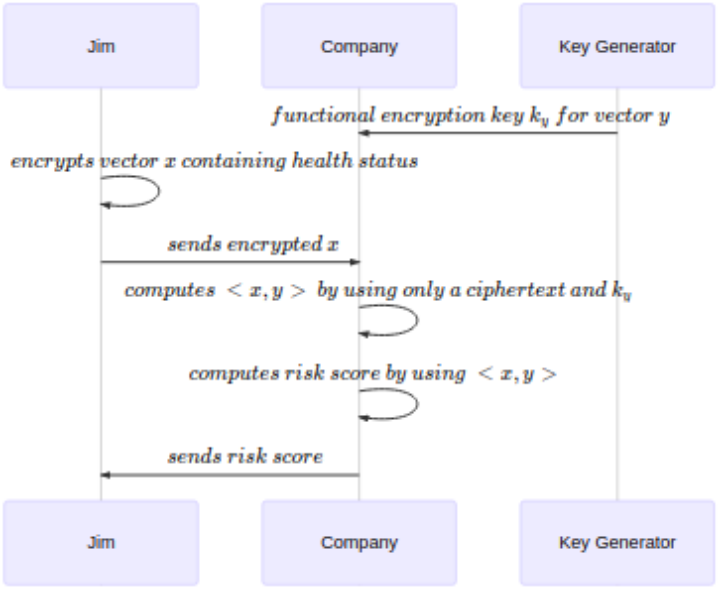

Figure 17: Computation of cardiovascular disease risk

Key Generator

The scheme parameters and master keys are generated by Key Generator at the setup phase:

```
l := 8
boundX := new(big.Int).Exp(big.NewInt(2), big.NewInt(64), nil)
boundY := new(big.Int).Exp(big.NewInt(2), big.NewInt(64), nil)
```

```
bitLength := 512
lambda := 128

paillier, err := fullysec.NewPaillier(l, lambda, bitLength, boundX, boundY)
if err != nil {
	fmt.Errorf("Error during simple inner product creation: %v", err)
}

masterSecKey, masterPubKey, err := paillier.GenerateMasterKeys()
if err != nil {
	fmt.Errorf("Error during master key generation: %v", err)
}

serialization.WriteGob("secKey.gob", masterSecKey)
serialization.WriteGob("pubKey.gob", masterPubKey)

serialization.WriteGob("paillier.gob", paillier.Params)
```

The same scheme parameters need to be used by all three components (Key Generator, Company, User) and thus need to be made public (below serialized in a file by using Golang encoding/gob package).

```
params := new(fullysec.PaillierParams)
err := serialization.ReadGob("paillier.gob", params)
if err != nil {
	fmt.Errorf("Error during Paillier params reading: %v", err)
}
paillier := fullysec.NewPaillierFromParams(params)
```

The Key Generator demonstrator code provides the REST API function which takes one parameter – vector y. It returns the functional decryption key which enables to compute the inner-product of x and y for an arbitrary vector x.

```
package keys

import (
	"encoding/json"
	"fmt"
	"net/http"

	"github.com/fentec-project/gofe/data"
	"github.com/fentec-project/gofe/innerprod/fullysec"
	"github.com/fentec-project/private-predictions/serialization"
	"github.com/go-chi/chi"
	"github.com/go-chi/render"
)

func Routes() *chi.Mux {
	router := chi.NewRouter()
	router.Post("/paillier", DerivePaillierKey)
```

```
	return router
}

func DerivePaillierKey(w http.ResponseWriter, r *http.Request) {
	params := new(fullysec.PaillierParams)
	err := serialization.ReadGob("paillier.gob", params)
	if err != nil {
		fmt.Errorf("Error during Paillier params reading: %v", err)
	}
	paillier := fullysec.NewPaillierFromParams(params)

	masterSecKey := new(data.Vector)
	err = serialization.ReadGob("secKey.gob", masterSecKey)
	if err != nil {
		fmt.Errorf("Error during key reading: %v", err)
	}

	y1 := new(data.Vector)
	err = json.NewDecoder(r.Body).Decode(&y1)
	if err != nil {
		panic(err)
	}

	key1, err := paillier.DeriveKey(*masterSecKey, *y1)
	if err != nil {
		fmt.Errorf("Error during key derivation: %v", err)
	}

	render.JSON(w, r, key1)
}
```

For the demonstrator, the Paillier scheme is used. However, Key Generator might provide key derivation functions for other schemes if needed. Also, note that the authentication and authorization process is not included – in real-world applications, this would need to be added to check whether an entity asking for the functional keys is entitled to them.

User

The component that runs on User's device prepares vector x which contains eight input parameters, encrypts it, and sends it to the privacy-friendly service:

```
var age, systolicBP, totalCh, hdlCh, factor float64
factor = 100000
factorInt := 100000

age = 43
systolicBP = 120 // systolic blood pressure
totalCh = 180 // total cholesterol
hdlCh = 66 // HDL cholesterol
```

```
ageLog := math.Log(age)
systolicBPLog := math.Log(systolicBP)
totalChLog := math.Log(totalCh)
hdlChLog := math.Log(hdlCh)

ageLog *= factor
systolicBPLog *= factor
totalChLog *= factor
hdlChLog *= factor

isMale := 1
smoker := 0
treatedBP := 0 // treated blood pressure
diabetic := 0

ageInt := big.NewInt(int64(math.Round(ageLog)))
systolicBPInt := big.NewInt(int64(math.Round(systolicBPLog)))
totalChInt := big.NewInt(int64(math.Round(totalChLog)))
hdlChInt := big.NewInt(int64(math.Round(hdlChLog)))

isMaleInt := big.NewInt(int64(isMale * factorInt))
smokerInt := big.NewInt(int64(smoker * factorInt))
treatedBPInt := big.NewInt(int64(treatedBP * factorInt))
diabeticInt := big.NewInt(int64(diabetic * factorInt))

x := data.NewVector([]*big.Int{isMaleInt, ageInt, systolicBPInt, totalChInt,
    ↪ hdlChInt, smokerInt, treatedBPInt, diabeticInt})

params := new(fullysec.PaillierParams)
err := serialization.ReadGob("paillier.gob", params)
if err != nil {
       fmt.Errorf("Error during Paillier params reading: %v", err)
}
paillier := fullysec.NewPaillierFromParams(params)

masterPubKey := new(data.Vector)
err = serialization.ReadGob("pubKey.gob", masterPubKey)
if err != nil {
       fmt.Errorf("Error during key reading: %v", err)
}

ciphertext, err := paillier.Encrypt(x, *masterPubKey)
if err != nil {
       fmt.Errorf("Error during encryption: %v", err)
}

jsonValue, _ := json.Marshal(ciphertext)
response, err := http.Post("http://localhost:8081/v1/api/framingham/30", "
```

```
    ↪ application/json", bytes.NewBuffer(jsonValue))
```

Note that the parameters have been multiplied by a factor – this is due to the fact that the risk computation uses vector y which contains real numbers (not integers). More details follow in the next section.

Company

Framingham risk score algorithms are based on Cox proportional hazards model [17]. Part of it is a multiplication of the input parameters by regression factors which are real numbers. In the 30-year algorithm, the vector x containing the user's health parameters is multiplied by two vectors (scalar or inner-product):

```
y1 = (0.34362, 2.63588, 1.8803, 1.12673, -0.90941, 0.59397, 0.5232,
0.68602)
y2 = (0.48123, 3.39222, 1.39862, -0.00439, 0.16081, 0.99858,
0.19035, 0.49756)
```

Regression factors need to be converted into integers because cryptographic schemes operate with integers. This is straight-forward in functional encryption schemes: we multiply factors by a power of 10 to obtain whole numbers. A factor of 100 000 is used in our case. Consequently, we multiply the input parameters by the same factor (see the previous section). For example, boolean parameters which are presented as 1 (true) or 0 (false) thus become 100 000 or 0.

Company needs to obtain two functional decryption keys from the Key Generator: a key to compute the inner-product of x and $y1$, and a key to compute the inner-product of x and $y2$:

```
r2 := big.NewInt(34362)
r3 := big.NewInt(263588)
r4 := big.NewInt(188030)
r5 := big.NewInt(112673)
r6 := big.NewInt(-90941)
r7 := big.NewInt(59397)
r8 := big.NewInt(52320)
r9 := big.NewInt(68602)
y1 := data.NewVector([]*big.Int{r2, r3, r4, r5, r6, r7, r8, r9})

jsonValue, _ := json.Marshal(y1)
response, err := http.Post("http://localhost:8080/v1/api/paillier",
       "application/json", bytes.NewBuffer(jsonValue))
if err != nil {
       fmt.Printf("The HTTP request failed with error %s\n", err)
}
data1, err := ioutil.ReadAll(response.Body)
if err != nil {
       fmt.Printf("Accessing response body failed with error %s\n", err)
}
key1 := string(data1)

t2 := big.NewInt(48123)
t3 := big.NewInt(339222)
```

```
t4 := big.NewInt(139862)
t5 := big.NewInt(-439)
t6 := big.NewInt(16081)
t7 := big.NewInt(99858)
t8 := big.NewInt(19035)
t9 := big.NewInt(49756)
y2 := data.NewVector([]*big.Int{t2, t3, t4, t5, t6, t7, t8, t9})

jsonValue, _ = json.Marshal(y2)
response, err = http.Post("http://localhost:8080/v1/api/paillier",
       "application/json", bytes.NewBuffer(jsonValue))
if err != nil {
       fmt.Printf("The HTTP request failed with error %s\n", err)
}
data2, err := ioutil.ReadAll(response.Body)
if err != nil {
       fmt.Printf("Accessing response body failed with error %s\n", err)
}
key2 := string(data2)

key1 = strings.TrimSpace(key1)
key2 = strings.TrimSpace(key2)

key1Int, _ := new(big.Int).SetString(key1, 10)
key2Int, _ := new(big.Int).SetString(key2, 10)

serialization.WriteGob("framingham30-FE-y1-key.gob", key1Int)
serialization.WriteGob("framingham30-FE-y2-key.gob", key2Int)
```

Once Company gets the encrypted parameters, it can compute the risk:

```
func Risk(w http.ResponseWriter, r *http.Request) {
       ciphertext := new(data.Vector)
       err := json.NewDecoder(r.Body).Decode(&ciphertext)
       if err != nil {
              panic(err)
       }

       y1Key := new(big.Int)
       err = serialization.ReadGob("framingham30-FE-y1-key.gob", y1Key)
       if err != nil {
              fmt.Errorf("Error during key reading: %v", err)
       }

       y2Key := new(big.Int)
       err = serialization.ReadGob("framingham30-FE-y2-key.gob", y2Key)
       if err != nil {
              fmt.Errorf("Error during key reading: %v", err)
```

```
}

// paillier.gob generated by key-server needs to be used
params := new(fullysec.PaillierParams)
err = serialization.ReadGob("paillier.gob", params)
if err != nil {
	fmt.Errorf("Error during Paillier params reading: %v", err)
}
paillier := fullysec.NewPaillierFromParams(params)

r2 := big.NewInt(34362)
r3 := big.NewInt(263588)
r4 := big.NewInt(188030)
r5 := big.NewInt(112673)
r6 := big.NewInt(-90941)
r7 := big.NewInt(59397)
r8 := big.NewInt(52320)
r9 := big.NewInt(68602)
y1 := data.NewVector([]*big.Int{r2, r3, r4, r5, r6, r7, r8, r9})

t2 := big.NewInt(48123)
t3 := big.NewInt(339222)
t4 := big.NewInt(139862)
t5 := big.NewInt(-439)
t6 := big.NewInt(16081)
t7 := big.NewInt(99858)
t8 := big.NewInt(19035)
t9 := big.NewInt(49756)
y2 := data.NewVector([]*big.Int{t2, t3, t4, t5, t6, t7, t8, t9})

xy1, err := paillier.Decrypt(*ciphertext, y1Key, y1)
if err != nil {
	fmt.Errorf("Error during decryption")
}

xy2, err := paillier.Decrypt(*ciphertext, y2Key, y2)
if err != nil {
	fmt.Errorf("Error during decryption")
}

var factor float64
factor = 100000

xy1Actual := float64(xy1.Int64())
xy1Actual = xy1Actual / (factor * factor)

xy2Actual := float64(xy2.Int64())
xy2Actual = xy2Actual / (factor * factor)
```

```
        w2 := 21.29326612
        w5 := math.Exp(xy1Actual - w2)

        x2 := 20.12840698
        x5 := math.Exp(xy2Actual - x2)

        F := make([]float64, 1340)
        G := make([]float64, 1340)
        K := make([]float64, 1340)
        M := make([]float64, 1340)

        for i := 0; i < len(E); i++ {
                F[i] = math.Pow(E[i], w5)
        }

        for i := 0; i+1 < len(E); i++ {
                G[i] = math.Log(E[i]) - math.Log(E[i+1])
        }

        for i := 0; i < len(K); i++ {
                K[i] = math.Pow(J[i], x5)
        }

        M[0] = w5 * (-math.Log(E[0]))
        for i := 0; i+1 < len(M); i++ {
                M[i+1] = F[i] * K[i] * w5 * G[i]
        }

        var fullRisk = 0.0
        for i := 0; i < len(M); i++ {
                fullRisk = fullRisk + M[i]
        }

        risk := math.Round(100.0 * fullRisk)

        fmt.Println("risk:")
        fmt.Println(risk)

        render.JSON(w, r, risk)
}
```

Note that the computation of the two inner-products is only a small part of the risk algorithm. To obtain the risk score, the algorithm computes e raised to the inner-product value (for both inner-products). In the next step, 1340 * 1340 power functions, 1340 * 3 multiplications, and 1340 additions are computed using the values derived from both inner-products by an exponential function. For details please refer to [24]. These operations are executed by Company and the result is returned to User.

User thus does not need to know anything about the algorithm to obtain the personal CVD risk score and at the same time, Company does not know anything about User's parameters (except the inner-products of x with vectors y1 and y2).

9.2 Underground Anonymous Heatmap

The demonstrator in the previous section uses a centralized Key Generator. In this section, we show how such a centralized trustworthy component can be avoided by using the usage of a decentralized scheme.

Let us say Ljubljana has a brand new metro. A mayor would like to collect some traffic data by using a metro app. Well, citizens do not want to reveal their data. Luckily, there are types of encryption that allow to run statistical analyses on the encrypted data. The goal is to protect (encrypt) Users' paths and still being able to compute some statistics on the mayor's side.

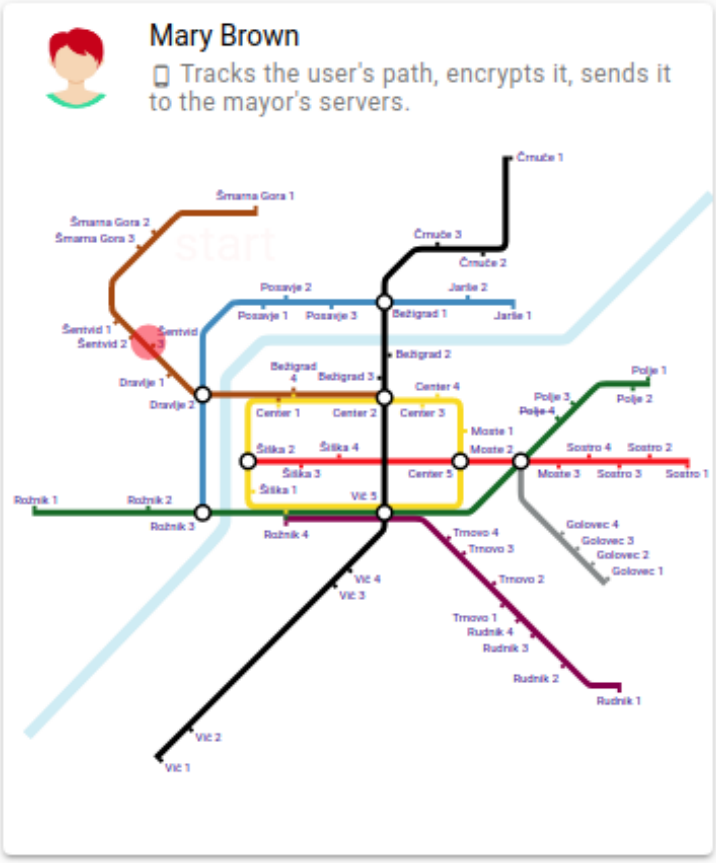

Figure 18: App following the User's path

We demonstrate how traffic heatmap can be generated based on the encrypted data. Given the encrypted information about Users of the underground, the mayor's Service can measure the traffic density at each particular station. Thus, congestions and potential increases in traffic can be detected while the User data is encrypted and remains private.

The demonstrator [2] uses "Decentralized Multi-Client Functional Encryption for Inner Product scheme" [16].

This scheme allows each User to encrypt the data in a decentralized way. Neither Service nor the other individuals know anything about the location or private key of the User. The only information that Service can obtain is how many Users traveled through each particular station in a given time frame, thus preserving the privacy of each individual.

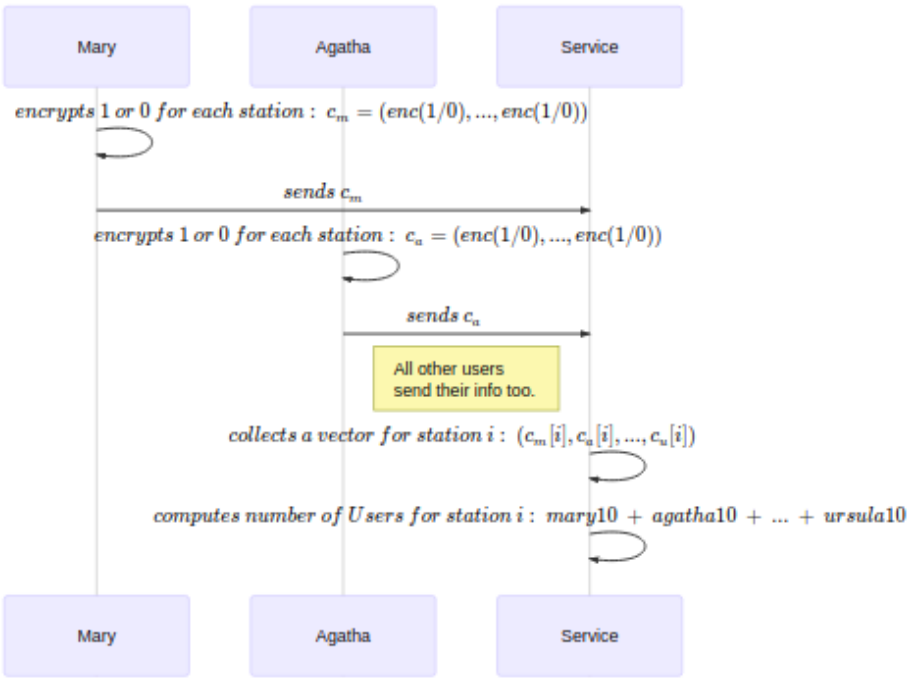

Figure 19: Users sending encrypted paths to the Service

Each User encrypts the vector specifying the path that was traveled. The length of the vector is the same as the number of stations. It consists of 0s and 1s: 1s for the stations were the User traveled. More precisely, each coordinate (0 or 1) is encrypted separately to provide an encryption vector. In GoFE the code looks as presented below. First, path vectors are loaded – note that the paths have been generated randomly and are stored in one file. In a real-world application, the app of each User would generate the path and encrypt it locally. Just for demonstration purposes, the code below performs all the computations in a single execution.

```
pathVectors, stations, err := readMatFromFile("london_paths.txt")
numClients := len(pathVectors)
vecDim := len(pathVectors[0])
fmt.Println("reading the data; numer of clients:", numClients)

clients := make([]*fullysec.DMCFEClient, numClients)
pubKeys := make([]*bn256.G1, numClients)
```

Each User needs to instantiate the *DMCFEClient*, and then the secret keys are generated.

```
// create clients and make a slice of their public values
for i := 0; i < numClients; i++ {
	c, err := fullysec.NewDMCFEClient(i)
	if err != nil {
		panic(errors.Wrap(err, "could not instantiate fullysec"))
```

```
        }
        clients[i] = c
        pubKeys[i] = c.ClientPubKey
}

// based on public values of each client create private matrices T_i summing to
    ↪ 0
for i := 0; i < numClients; i++ {
        err = clients[i].SetShare(pubKeys)
        if err != nil {
                panic(errors.Wrap(err, "could not create private values"))
        }
}
fmt.Println("clients agreed on secret keys")
```

Finally, each User encrypts the path:

```
// pathVec[i] is the value of i-th station, label its name,
// c[i] is its encryption
label = station[i]
c[i], _ := client.Encrypt(pathVec[i], label)
```

In the decentralized scheme [16], the functional decryption keys are generated by Users (no trusted authority is needed). Users thus provide a functional key to a central service component. In our case, a functional key for a vector y of 1s is provided (the vector length is the number of users). This is because the central authority will decrypt the sum of all the Users that traveled through that station, i.e., a value that can be represented as an inner-product of y and a vector x of 0s and 1s indicating which Users traveled through that station. Each User provides a key share.

The subject that gets all the secrets can thus combine them to obtain a functional key and then decrypt the heatmap.

```
// each client gives his key share corresponding to the vector of
// ones; only knowing all the key shares one can decrypt the
// sum of all locations of the clients
keyShares := make([]data.VectorG2, numClients)
oneVec := data.NewConstantVector(numClients, big.NewInt(1))
for k := 0; k < numClients; k++ {
        keyShare, err := clients[k].DeriveKeyShare(oneVec)
        if err != nil {
                panic(errors.Wrap(err, "could not generate key shares"))
        }
        keyShares[k] = keyShare
}
fmt.Println("clients created keys for decrypting heatmap")

heatmap := make([]*big.Int, vecDim)
for i := 0; i < vecDim; i++ {
        label := stations[i]
```

```
        heatmap[i], err = fullysec.DMCFEDecrypt(ciphers[i], keyShares, oneVec,
            ↪ label, big.NewInt(int64(numClients)))
        if err != nil {
                panic(errors.Wrap(err, "could not decrypt"))
        }
}
```

Note that the subject holding the functional key could decrypt only the heatmap (density for each station) and no other information (like individual paths).

While we use randomly generated User data for this demonstration, one can easily imagine a smartphone app which tracks the User's path, generates a vector, encrypts it (all operations performed locally), and finally sends it to Service.

Note that the computed value for the i-th station represents

$$x \cdot y = x_1 \cdot 1 + x_2 \cdot 1 + \ldots + x_\ell \cdot 1,$$

where x_k is ether 0 or 1 depending if the k-th User traveled through it and y the vector of 1s. Thus, the number of all Users that traveled through a particular station is computed.

Using the described approach, a variety of other analysis services can be built on the encrypted data, for example, the power consumption of a group of houses in a neighborhood, measurements from IoT devices, etc. In the former case, the power consumption could be encrypted for each hour and sent to the central component. The Service could then compute (decrypt) the overall consumption (across all houses) for each particular hour. Based on such privacy-enhanced computations, various prediction services can be built using only encrypted data.

9.3 Selective Access to Clinical Data

One of the main advantages of attribute-based encryption (ABE) schemes is the ability to manage who can access each particular set of data. In the following demonstration, we follow the typical functional encryption deployment with a central authority as a Key Generator, an Encryptor which possesses data and encrypts it, and a Decryptor which decrypts some parts of the data (according to its role).

Patients are owners of their clinical histories, therefore clinical services such as hospitals should provide tools to manage them according to privacy policies. To this end, the Encryptor component is a tool that enables clinical services to set policies to access clinical histories. Later, the personnel of a hospital will use the Decryptor component to get access to clinical histories.

Key Generator is similar to the demonstrators in the previous sections – a trustworthy component which at the setup phase generates master keys and is later able to derive functional keys for each of the Decryptors.

The source code at [3] provides a demonstrator from health domain where clinical histories are encrypted to enable privacy for patients while clinical personnel still have access to the data needed for their work.

The demonstrator comprises the following components:

- Key Generator: central authority component which generates keys.

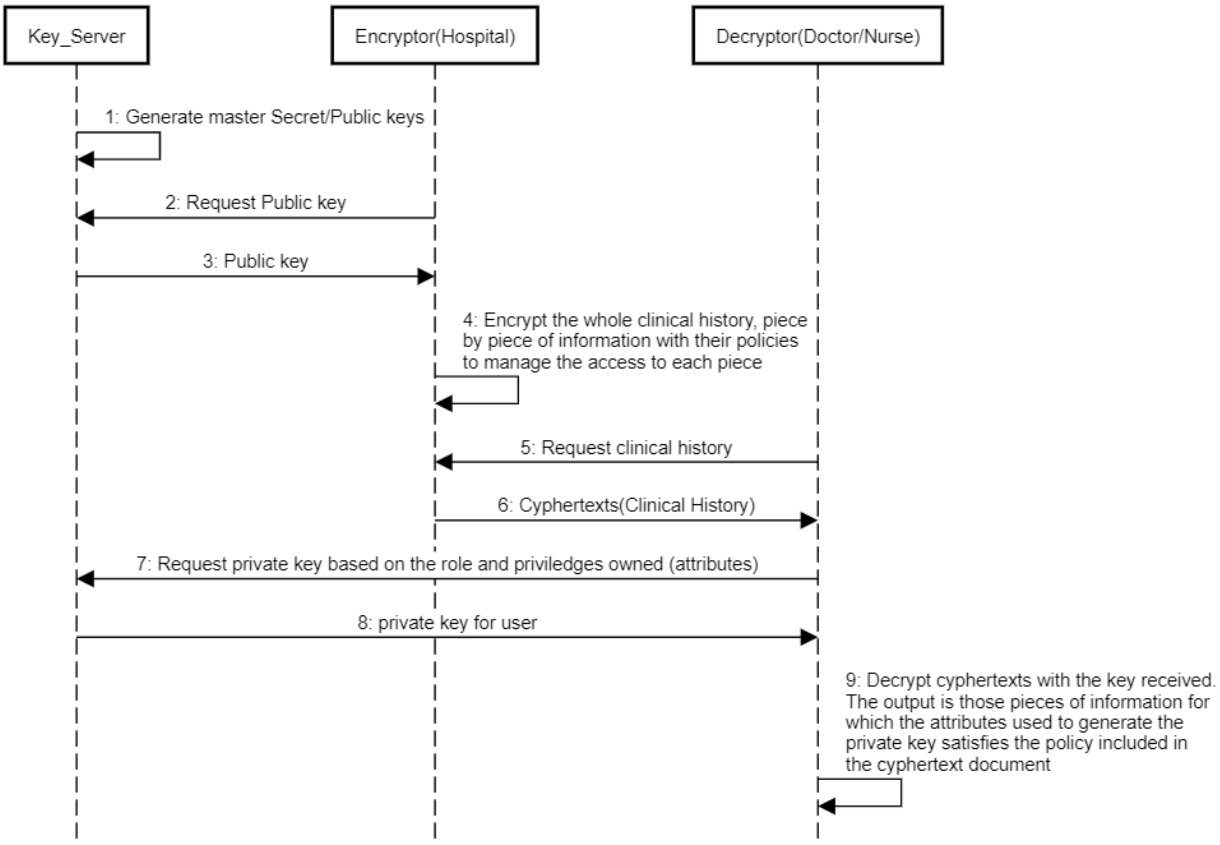

Figure 20: Interactions between clinical data components

- Decryptor: this component is used by the hospital personnel; doctors, nurses, etc. The component obtains functional decryption keys from the Key Generator according to the profile of the requester. It decrypts only those pieces of information that have a policy which matches with the attributes of the key.
- Encryptor: this component is used by the hospital to encrypt the clinical history according to access policies.

The Health Level Seven International (HL7, [4]) provides a set of standards and framework for the management of health information. "Level Seven" refers to the seventh level of the International Organization for Standardization (ISO) seven-layer communications model for Open Systems Interconnection (OSI) - the application level. One of the resources provided by HL7 is Fast Health Interoperability Resource (FHIR, [5]), which provides a set of data structure in several formats (JSON,cXML, etc) to represent different types of clinical information. For instance, sickness and the process of diagnosis and treatment for a patient is defined by the structure "Condition" [6]. This labeled format is perfectly suitable to be encrypted by pieces as each label of the document can constitute a piece of information.

The demonstrator manages a set of diseases expressed as FHIR Conditions, although in this case each of them is managed through an Excel table (Figure, 22) to provide the information in a more readable format. Each row of the table corresponds to a patient, and each column corresponds to different levels of confidentiality.

The demonstrator uses the CP-ABE [13] scheme which allows to encrypt a message based on a boolean expression, defining a policy with the attributes needed for the decryption. The scheme uses positive integer numbers as attributes, which makes it necessary to associate each meaningful attribute with a number.

```
number 0 and 1: attribute values corresponds to clinical service (0=Saint
    ↪ Charles Hospital, 1=Ramon y Cajal Hospital).
numbers 2 up to 4: attribute values corresponds to departments in a hospital (2=
    ↪ emergency, 3=administration, 4=cardiology).
numbers 5 and 6: attribute values corresponds to personnel roles (5=doctors, 6=
    ↪ nurses).
```

Key Generator

The Key Generator creates a master secret key which is used to generate functional decryption keys for each of the Decryptors based on their attributes. The keys are stored in files which are later used by Decryptors.

In this demo we have three Decryptors:

```
Decryptor 1: it works in Saint Charles Hospital, emergency and administration
    ↪ departments (0, 2, 3)
Decryptor 2: it works in Saint Charles Hospital, cardiology department (0, 4)
Decryptor 3: it works in Saint Charles Hospital, emergency department and is a
    ↪ doctor (0, 2, 5)
```

```
// generate a public key and a secret key for the scheme
abe.GenerateMasterKeys(path)
```

```
/* Decryptor 1: it works in Saint Charles Hospital, emergency and administration
    ↪ departments*/
if debug {fmt.Println("Key for decryptor 1: {0, 2, 3}")}
// define a set of attributes (a subset of the universe of attributes)
// that an entity possesses
gamma := []int{0, 2, 3}
// generate keys for decryption for an entity with
// attributes gamma
abe.GenerateAttribKeys(path, "keyDecryptor1.gob", gamma)

/* Decryptor 2: it works in Saint Charles Hospital, cardiology department*/
if debug {fmt.Println("Key for decryptor 2: {0, 4}")}
// define a set of attributes (a subset of the universe of attributes)
// that an entity possesses
gamma = []int{0, 4}
// generate keys for decryption for an entity with
// attributes gamma
abe.GenerateAttribKeys(path, "keyDecryptor2.gob", gamma)

/* Decryptor 3: it works in Saint Charles Hospital, emergency deparment and is a
    ↪ doctor*/
if debug {fmt.Println("Key for decryptor 3: {0, 2, 5}")}
// define a set of attributes (a subset of the universe of attributes)
// that an entity possesses
gamma = []int{0, 2, 5}
// generate keys for decryption for an entity with
// attributes gamma
abe.GenerateAttribKeys(path, "keyDecryptor3.gob", gamma)
```

Encryptor/Hospital The Encryptor uses a predefined set of policies in the form of boolean expresions.

```
msp1, _ := abe.BooleanToMSP("(0 AND 2) OR 6", false)
msp2, _ := abe.BooleanToMSP("(0 AND 4) OR 5", false)
msp3, _ := abe.BooleanToMSP("(0 AND 2) AND 5", false)
```

The msp structure represents the policies which will be used to encrypt each column. They can be translated as:

- Column one can be seen by anyone in the emergency department of the Saint Charles Hospital, or by any nurse.
- Column two can be seen by anyone in the cardiology department of the Saint Charles Hospital, or by any doctor.
- Column three can be only seen by doctors working in the emergency department of the Saint Charles Hospital.

Decryptor The Decryptor uses the functional decryption keys created by the Key Generator to decrypt the clinical history file. Each of the Decryptors will only get the pieces of information which could be

decrypted with the key provided.

For the test, the Decryptor provides a command line interface to choose between the three Decryptors which correspond to three decryption keys created before.

```
package main

import (
	"fmt"
	"bufio"
	"os"
	"strings"
	"io/ioutil"
	"encoding/base64"
	"github.com/fentec-project/gofe-v2/abe"
	"github.com/tealeg/xlsx"
)

func main() {
	keyPath := ""
	outputFile := ""
	/***** SELECT DECRYPTOR *****/
	menu :=

	`Select decryptor
	[1] Decryptor1 - {0, 2, 3}
	[2] Decryptor2 - {0, 4}
	[3] Decryptor3 - {0, 2, 5}`

	fmt.Println(menu)

	reader := bufio.NewReader(os.Stdin)

	in, _ := reader.ReadString('\n')
	selection := strings.TrimRight(in, "\r\n")

	switch selection{
		case "1":
			fmt.Println("Decryptor1 - {0, 2, 3}")
			keyPath = "keyDecryptor1.gob"
			outputFile = "decryptor1_data.xlsx"
		case "2":
			fmt.Println("Decryptor2 - {0, 4}")
			keyPath = "keyDecryptor2.gob"
			outputFile = "decryptor2_data.xlsx"
		case "3":
			fmt.Println("Decryptor3 - {0, 2, 5}")
			keyPath = "keyDecryptor3.gob"
			outputFile = "decryptor3_data.xlsx"
```

```
        default:
                fmt.Println("ERROR: Undefined decryptor")
    }
    /***** SELECT DECRYPTOR *****/

    path := "key-material/"

    var outputExcelFile *xlsx.File
    var sheet *xlsx.Sheet
    var row *xlsx.Row
    var cell *xlsx.Cell

    outputExcelFile = xlsx.NewFile()
    inputExcelFileName := "data/encryptor_data.xlsx"
    inputExcelFile, _ := xlsx.OpenFile(inputExcelFileName)

    for _, _sheet := range inputExcelFile.Sheets {
        sheet, _ = outputExcelFile.AddSheet("1")
        for _, _row := range _sheet.Rows {
            row = sheet.AddRow()
            i := 0
            for _, _cell := range _row.Cells {
                cell = row.AddCell()
                encoded_enc_text := _cell.String() // information to
                    ↪  be decrypted
    //Columns whose data will be encrypted. In this case: 1, 2 and 3 (
        ↪ starting from 0)
                if i == 1 || i == 2 || i == 3 {
                    enc_text, _ := base64.StdEncoding.
                        ↪ DecodeString(encoded_enc_text)
                    ioutil.WriteFile(path+"aux_encrypt.gob", []
                        ↪ byte(enc_text), 0644)
    // decrypt the ciphertext with the keys of an entity
    // that has sufficient attributes
                    err := abe.Decrypt(path, keyPath)
                    if err != nil {
    //fmt.Printf("Failed to decrypt: %v\n", err)
                        text := _cell.String()
                        cell.Value = text
                    }else {
                        text, _ := ioutil.ReadFile(path+"
                            ↪ aux_decrypt.gob")
                        cell.Value = string(text)
                    }
                }else{
                    text := _cell.String()
                    cell.Value = text
                }
```

```
                        i++;
                }
            }
            break
        }
            outputExcelFile.Save("data/"+outputFile)
            fmt.Println("\n---------- DECRYPTOR finished ----------\n")
}
```

```
Select decryptor
        [1] Decryptor1 - {0, 2, 3}
        [2] Decryptor2 - {0, 4}
        [3] Decryptor3 - {0, 2, 5}
1
Decryptor1 - {0, 2, 3}

---------- DECRYPTOR finished ----------
```

Figure 21: Decryptor module execution

Once the Encryptor has encrypted the clinical history, doctors and nurses can fetch the data they are permitted to see. For the demonstrator, the clinical history file is a table, where each cell contains a piece of information of a FHIR conditions structure:

Patient	Access level: Clinical org 0 **and** Department 2, **or** personnel level 6	Access level: Clinical org 0 **and** Department 4, **or** personnel level 5	Access level: Clinical org 0 **and** Department 2 **and** personnel level 5
Thomas	<coding> <system value="http://snomed.info/sct"/> <code value="38266002"/> <display value="Entire body as a whole"/> </coding>	<target> <reference value="Procedure/f201"/> <display value="TPF chemokuur"/> </target>	<target> <reference value="Condition/f205"/> <display value="bacterial infection"/> </target>
James	<coding> <system value="http://snomed.info/sct"/> <code value="24484000"/> <display value="Severe"/> </coding>	<detail> <reference value="DiagnosticReport/f201"/> <display value="Erasmus' diagnostic report of Roel's tumor"/> </detail>	<coding> <system value="http://snomed.info/sct"/> <code value="361355005"/> <display value="Entire head and neck"/> </coding>
Martha	<coding> <system value="http://snomed.info/sct"/> <code value="371924009"/> <display value="Moderate to severe"/> </coding>	<detail> <reference value="DiagnosticReport/f202"/> <display value="Diagnostic report for Roel's sepsis"/> </detail>	<target> <reference value="Substance/f202"/> <display value="Staphylococcus Aureus"/> </target>
Julia	<coding> <system value="http://snomed.info/sct"/> <code value="14803004"/> <display value="Temporary"/> </coding>	<coding> <system value="http://snomed.info/sct"/> <code value="24484000"/> <display value="Severe"/> </coding>	<target> <reference value="Procedure/f201"/> <display value="TPF chemotherapy for the sphenoid bone"/> </target>
Robert	<coding> <system value="http://snomed.info/sct"/> <code value="368009"/> <display value="Heart valve disorder"/> </coding>	<coding> <system value="http://snomed.info/sct"/> <code value="6736007"/> <display value="Moderate"/> </coding>	<coding> <system value="http://snomed.info/sct"/> <code value="40768004"/> <display value="Left thorax"/> </coding>

Figure 22: Clinical history file example

Once the file is encrypted, it looks like in Figure 23.

When a Decryptor decrypts the clinical history file, it will only obtain the information that it is allowed to see according to its attributes (role, department).

According to the Decryptor roles (attributes sets) and the policies, Decryptor role 1 (a person who works

Patient	Access level: Clinical org 0 **and** Department 2, **or** personnel level 6	Access level: Clinical org 0 **and** Department 4, **or** personnel level 5	Access level: Clinical org 0 **and** Department 2 **and** personnel level 5
Thomas	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2
James	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2
Martha	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2
Julia	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2
Robert	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2

Figure 23: Clinical history encrypted file

Patient	Access level: Clinical org 0 **and** Department 2, **or** personnel level 6	Access level: Clinical org 0 **and** Department 4, **or** personnel level 5	Access level: Clinical org 0 **and** Department 2 **and** personnel level 5
Thomas	<coding> <system value="http://snomed.info/sct"/> <code value="38266002"/> <display value="Entire body as a whole"/> </coding>	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2
James	<coding> <system value="http://snomed.info/sct"/> <code value="24484000"/> <display value="Severe"/> </coding>	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2
Martha	<coding> <system value="http://snomed.info/sct"/> <code value="371924009"/> <display value="Moderate to severe"/> </coding>	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2
Julia	<coding> <system value="http://snomed.info/sct"/> <code value="14803004"/> <display value="Temporary"/> </coding>	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2
Robert	<coding> <system value="http://snomed.info/sct"/> <code value="368009"/> <display value="Heart valve disorder"/> </coding>	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2	P/+BAwEBCkZBTUVDaXBoZXIB/4IAAQQBA0N0MAH/jAABAkN0Af+UAA EHQ3RQcmItZQH/IgABA01zcAH/nAAAAB3/iwEBAQxbM10qYm4yNTYu RzIB/4wAAf+EAQYAABP/gwMBAv+EAAEBAQFQAf+GAAAANP+FAwEBCn R3aXN0UG9pbnQB/4YAAQQBAVgB/4gAAQFZAf+IAAEBWgH/iAABAVQB /4gAAAAg/4cDAQEEZ2ZQMgH/iAABAgEBWAH/igABAVkB/4oAAAAK/4k FAQL/pAAAAB3/kwIBAQ5bXVszXSpibjI1Ni5HMQH/IAAB/5IAAA//kQEB Av+SAAH/jgEGAAAT/40DAQL/jgABAQEBUAH/kAAAADT/jwMBAQpjdXJ2

Figure 24: Data fetched by a doctor

in emergency and administration departments of the Saint Charles Hospital) will only obtain information from the column 1.

9.4 Neural Networks on Encrypted MNIST Dataset

We saw above how to implement privacy-friendly predictive services using inner-product schemes. Using inner-products (linear functions) many efficient machine learning models can be built based on linear regression or linear logistic.

However, linear models in many cases do not suffice. One of such tasks is image classification where linear classifiers mostly achieve significantly lower accuracy compared to the higher-degree classifiers. For example, classifiers for the well-known MNIST dataset where handwritten digits need to be recognized. A linear classifier on MNIST dataset is reported to have 92% accuracy (TensorFlows tutorial [7]), while more complex classifiers achieve over 99% accuracy.

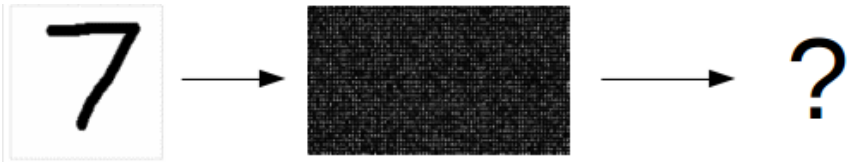

Figure 25: Can you classify encrypted digits?

GoFE and CiFEr include a scheme [25] for quadratic multi-variate polynomials which enables computation of quadratic polynomials on encrypted vectors. This enables richer machine learning models and even basic versions of neural networks. We provided a code [8] to demonstrate how an accurate neural network classifier can be built on the MNIST dataset and how functional encryption can be used to apply a classifier on the encrypted dataset. This means that a subject holding a function decryption key for a classifier can classify encrypted images, i.e., can classify each image depending on the digit in the encrypted image, but cannot see anything else within the image (for example, some characteristics of the handwriting).

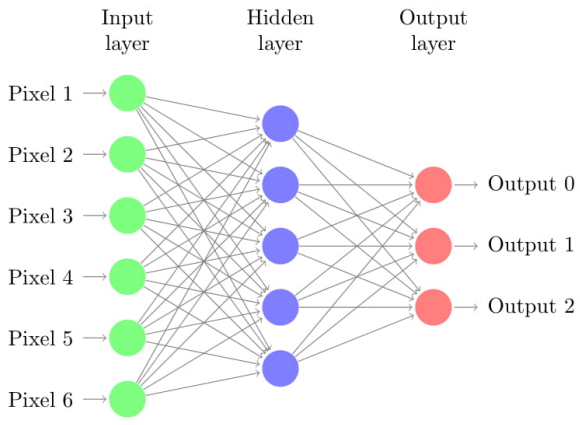

Figure 26: Two-layer neural network

The demonstrator uses the GoFE library and the widely-used machine learning library Tensor-Flow [9]. MNIST dataset consists of 60 000 images of handwritten digits. Each image is a 28×28 pixel array, where each pixel is represented by its gray level. The model we used is a 2-layer neural network with quadratic

function as non-linear activation function. Training of the model needs to be done on unencrypted data, while prediction is done on encrypted images. The images have been presented as 785-coordinate vectors ($28 \cdot 28 + 1$ for bias). We achieved the accuracy of 97%. The decryption of one image (applying the trained model on the encrypted image) takes under 20 seconds.

First, the machine learning model for recognizing the digits needs to be trained on the data in the clear. For this, *mnist.py* can be run. The model parameters are stored in *mat_diag.txt* and *mat_proj.txt*. File *mat_valid.txt* contains the vector to be encrypted.

```
proj, err := readMatFromFile("testdata/mat_proj.txt")
if err != nil {
	panic(errors.Wrap(err, "error reading projection matrix"))
}
nVecs := proj.Rows()
vecSize := proj.Cols()

// Diagonal matrix
// number of rows of this matrix represents the number of classes.
// The function will predict one of these classes.
diag, err := readMatFromFile("testdata/mat_diag.txt")
if err != nil {
	panic(errors.Wrap(err, "error reading diagonal matrix"))
}
nClasses := diag.Rows()
if diag.Cols() != nVecs {
	panic(fmt.Sprintf("diagonal matrix must have %d columns", nVecs))
}

// Valid matrix
// number of rows of this matrix represents the number of examples.
valid, err := readMatFromFile("testdata/mat_valid.txt")
if err != nil {
	panic(errors.Wrap(err, "error reading valid matrix"))
}
if valid.Cols() != vecSize {
	panic(fmt.Sprintf("valid matrix must have %d columns", vecSize))
}
```

The scheme is then initialized and a vector encrypted:

```
// We know that all the values in the matrices are in the
// interval [-bound, bound].
bound := big.NewInt(100)

// q is an instance of the FE scheme for quadratic multi-variate
// polynomials constructed by Sans, Gay, Pointcheval (SGP)
q := quad.NewSGP(vecSize, bound)

// we generate a master secret key that we will need for encryption
// of our data.
```

```
fmt.Println("Generating master secret key...")
msk, err := q.GenerateMasterKey()
if err != nil {
	panic(errors.Wrap(err, "error when generating master keys"))
}

// First, we encrypt the data from mat_valid.txt
// with our master secret key.
// x = first row of matrix valid
// y = also the first row of matrix valid
fmt.Println("Encrypting...")
c, err := q.Encrypt(valid[0], valid[0], msk)
if err != nil {
	panic(errors.Wrap(err, "error when encrypting"))
}

// Then, we manipulate the encryption to be the encryption of the
// projected data.
// Note that this can also be done without knowing the secret key.
fmt.Println("Manipulating encryption...")
projC := projectEncryption(c, proj)

fmt.Println("Manipulating secret key...")
projSecKey := projectSecKey(msk, proj)
```

The subject that holds a functional decryption key can then compute the classification of the encrypted vector (image):

```
// We create a new (projected) scheme instance for decrypting
newBound := big.NewInt(1500000000)
fmt.Println("Creating new (projected) scheme instance for decrypting...")
qProj := quad.NewSGP(nVecs, newBound)

res := make([]*big.Int, nClasses)
maxValue := new(big.Int).Set(newBound)
maxValue = maxValue.Neg(maxValue)

fmt.Println("Predicting...")
predictedNum := 0 // the predicted number
for i := 0; i < nClasses; i++ {
	// We construct a diagonal matrix D that has the elements in the
	// current row of matrix diag on the diagonal.
	D := diagMat(diag[i])

	// We derive a feKey for obtaining the prediction from the encryption.
	// We will use this feKey for decrypting the final result,
	// e.g. x^T * D * y.
	feKey, err := qProj.DeriveKey(projSecKey, D)
```

```
        if err != nil {
                panic(errors.Wrap(err, "error when deriving FE key"))
        }

        // We decrypt the encryption with the derived key feKey.
        // The result of decryption holds the value of x^T * D * y,
        // which in our case predicts the number from the handwritten
        // image.
        dec, err := qProj.Decrypt(projC, feKey, D)
        if err != nil {
                panic(errors.Wrap(err, "error when decrypting"))
        }
        res[i] = dec
        if dec.Cmp(maxValue) > 0 {
                maxValue.Set(dec)
                predictedNum = i
        }
}

fmt.Println("Prediction vector:", res)
fmt.Println("The model predicts that the number on the image is", predictedNum)
```

10 Conclusions

In this deliverable, we presented the API of the FENTEC library. We gave a brief overview of the intended usage for each of the schemes. Finally, we demonstrated using four real-world applications how to use the FENTEC library. Soon, a website will be released which will contain similar content as this document, but with much more interactivity and visual representations. Hopefully, the website will help to bring the exciting, but still somewhat lesser-known techniques of functional encryption closer to the developers' and IT architects' communities.

References

[1] https://github.com/fentec-project/privacy-friendly-analyses.

[2] https://github.com/fentec-project/FE-anonymous-heatmap.

[3] https://github.com/fentec-project/Selective-Access-to-Clinical-Data.

[4] https://www.hl7.org/.

[5] http://www.fhir.org/.

[6] http://hl7.org/fhir/condition.html.

[7] https://www.tensorflow.org/tutorials#evaluating_our_model.

[8] https://github.com/fentec-project/neural-network-on-encrypted-data.

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

[10] Michel Abdalla, Fabrice Benhamouda, Markulf Kohlweiss, and Hendrik Waldner. Decentralizing Inner-Product Functional Encryption. In Dongdai Lin and Kazue Sako, editors, *Public-Key Cryptography – PKC 2019*, volume 11443 of *Lecture Notes in Computer Science*, pages 128–157, Beijing, China, April 2019.

[11] Michel Abdalla, Florian Bourse, Angelo De Caro, and David Pointcheval. Simple functional encryption schemes for inner products. In *IACR International Workshop on Public Key Cryptography*, pages 733–751. Springer, 2015.

[12] 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.

[13] Melissa Chase Agrawal. FAME: Fast Attribute-based Message Encryption. https://acmccs.github.io/.

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

[15] Shweta Agrawal, Benoît Libert, and Damien Stehlé. Fully secure functional encryption for inner products, from standard assumptions. In *Annual International Cryptology Conference*, pages 333–362. Springer, 2016.

[16] 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.

[17] David R Cox. Regression models and life-tables. *Journal of the Royal Statistical Society: Series B (Methodological)*, 34(2):187–202, 1972.

[18] Pratish Datta, Tatsuaki Okamoto, and Junichi Tomida. Full-hiding (unbounded) multi-input inner product functional encryption from the k-linear assumption. In *IACR International Workshop on Public Key Cryptography*, pages 245–277. Springer, 2018.

[19] Ralph B D'agostino, Ramachandran S Vasan, Michael J Pencina, Philip A Wolf, Mark Cobain, Joseph M Massaro, and William B Kannel. General cardiovascular risk profile for use in primary care. *Circulation*, 117(6):743–753, 2008.

[20] Romain Gay. A new paradigm for public-key functional encryption for degree-2 polynomials. In *IACR International Conference on Public-Key Cryptography*, pages 95–120. Springer, 2020.

[21] 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.

[22] Sam Kim, Kevin Lewi, Avradip Mandal, Hart Montgomery, Arnab Roy, and David J Wu. Function-hiding inner product encryption is practical. In *International Conference on Security and Cryptography for Networks*, pages 544–562. Springer, 2018.

[23] Yan Michalevsky and Marc Joye. Decentralized policy-hiding attribute-based encryption with receiver privacy. *IACR Cryptology ePrint Archive*, 2018:753, 2018.

[24] Michael J Pencina, Ralph B D'Agostino Sr, Martin G Larson, Joseph M Massaro, and Ramachandran S Vasan. Predicting the thirty-year risk of cardiovascular disease: the framingham heart study. *Circulation*, 119(24):3078, 2009.

[25] 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.