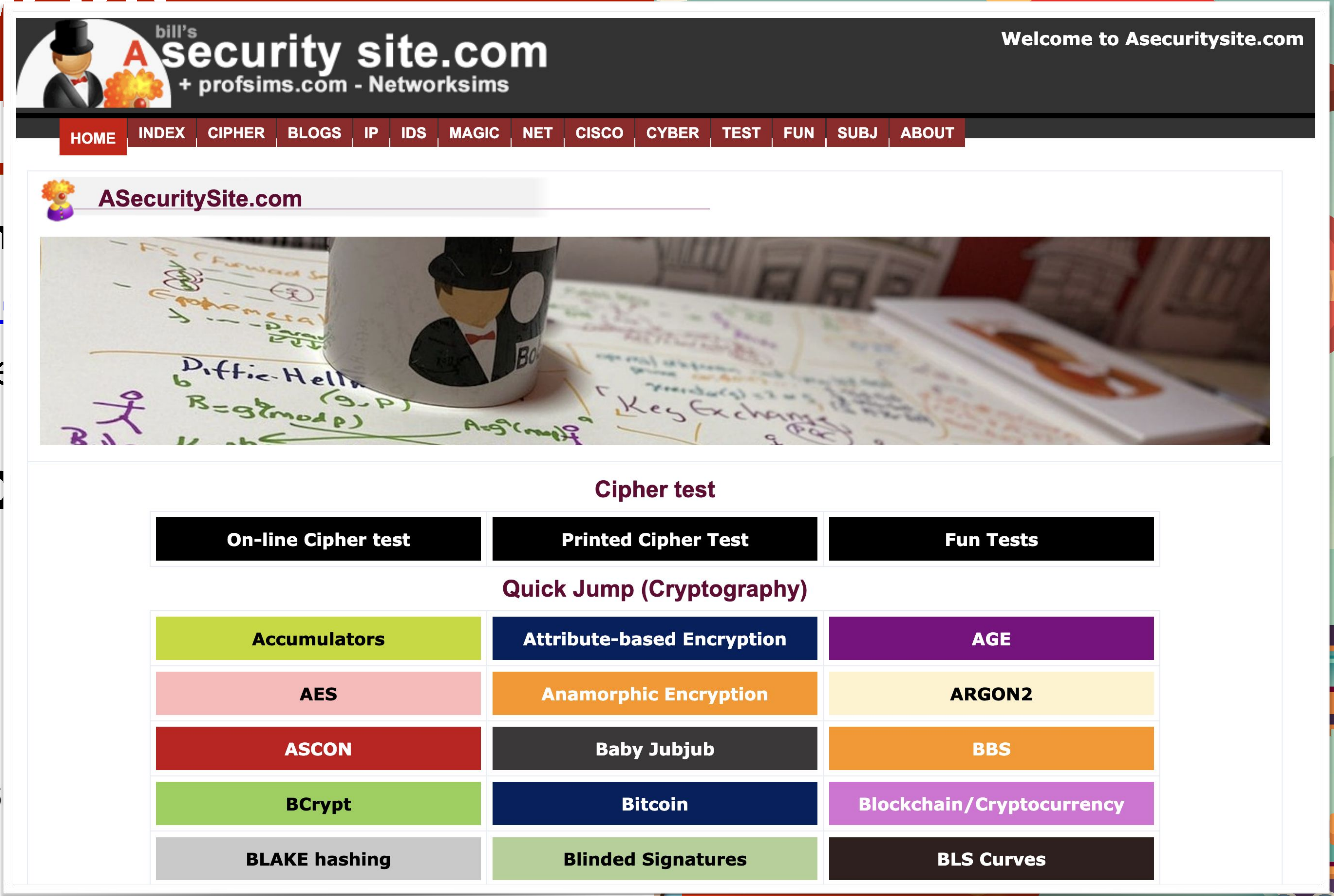
Privacy-Aw
Cryptogra

Prof Bill Buchanan

http://asecuritysit

Twitter: billatnapie

Homomorp
Encryption
OpenFHE



bill's A security site.com

+ profsims.com - Networksims

Welcome to Asecuritysite.com

ASecuritySite.com

Cipher test

On-line Cipher test	Printed Cipher Test	Fun Tests

Quick Jump (Cryptography)

Accumulators	Attribute-based Encryption	AGE
AES	Anamorphic Encryption	ARGON2
ASCON	Baby Jubjub	BBS
BCrypt	Bitcoin	Blockchain/Cryptocurrency
BLAKE hashing	Blinded Signatures	BLS Curves

BLOCKPASS IDENTITY LAB

World-leading Collaboration between Blockpass IDN and Edinburgh Napier University

PHE and FHE

Prof Bill Buchanan OBE, FRSE

http://asecuritysite.com

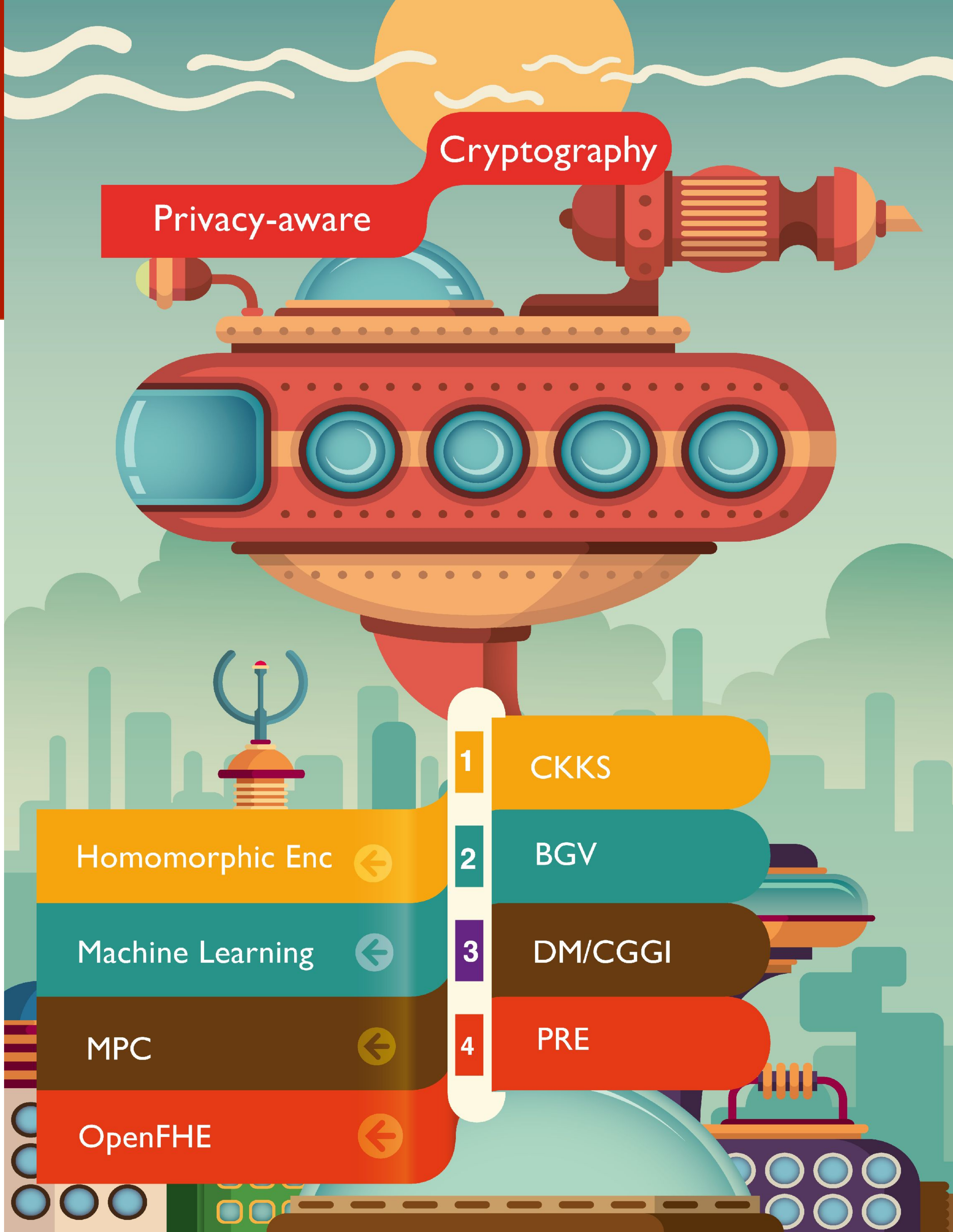
Twitter: billatnapier

Homomorphic Encryption with OpenFHE

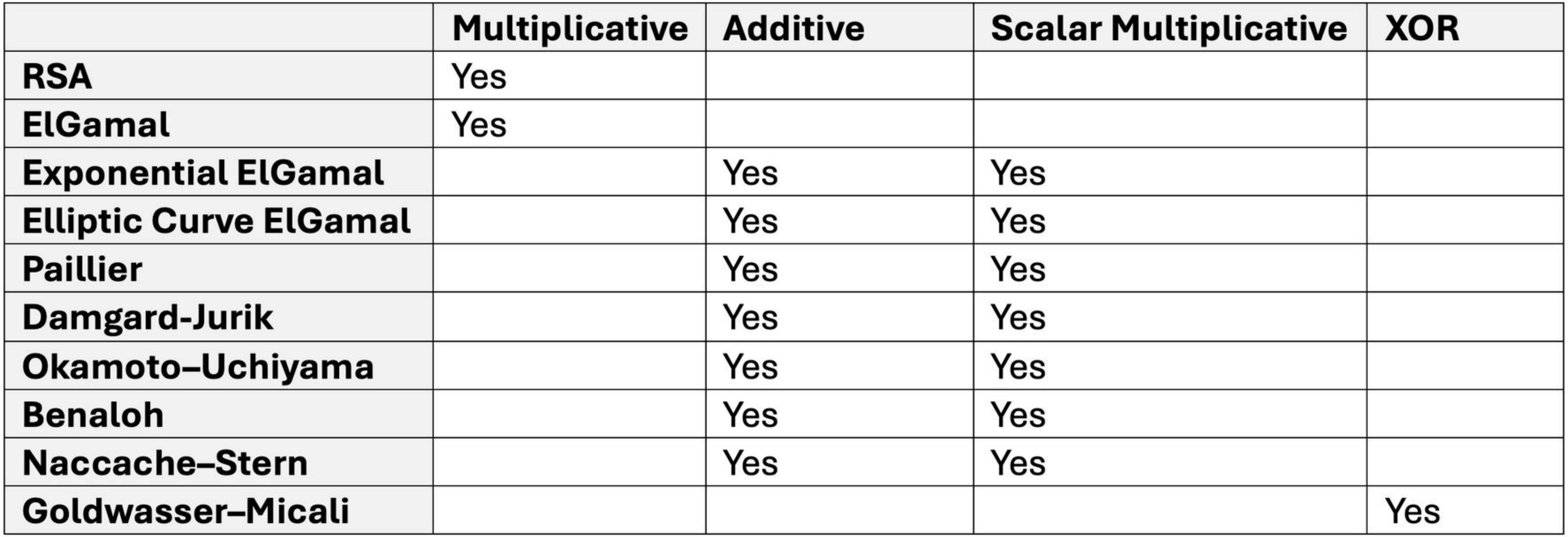
OpenFHE

BLOCKPASS IDENTITY LAB

World-leading Collaboration between Blockpass IDN and Edinburgh Napier University



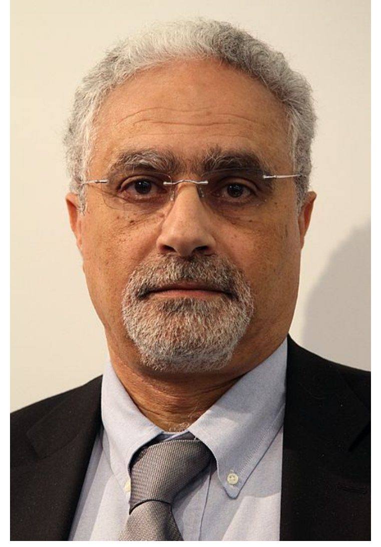
Partial Homorphic Encryption

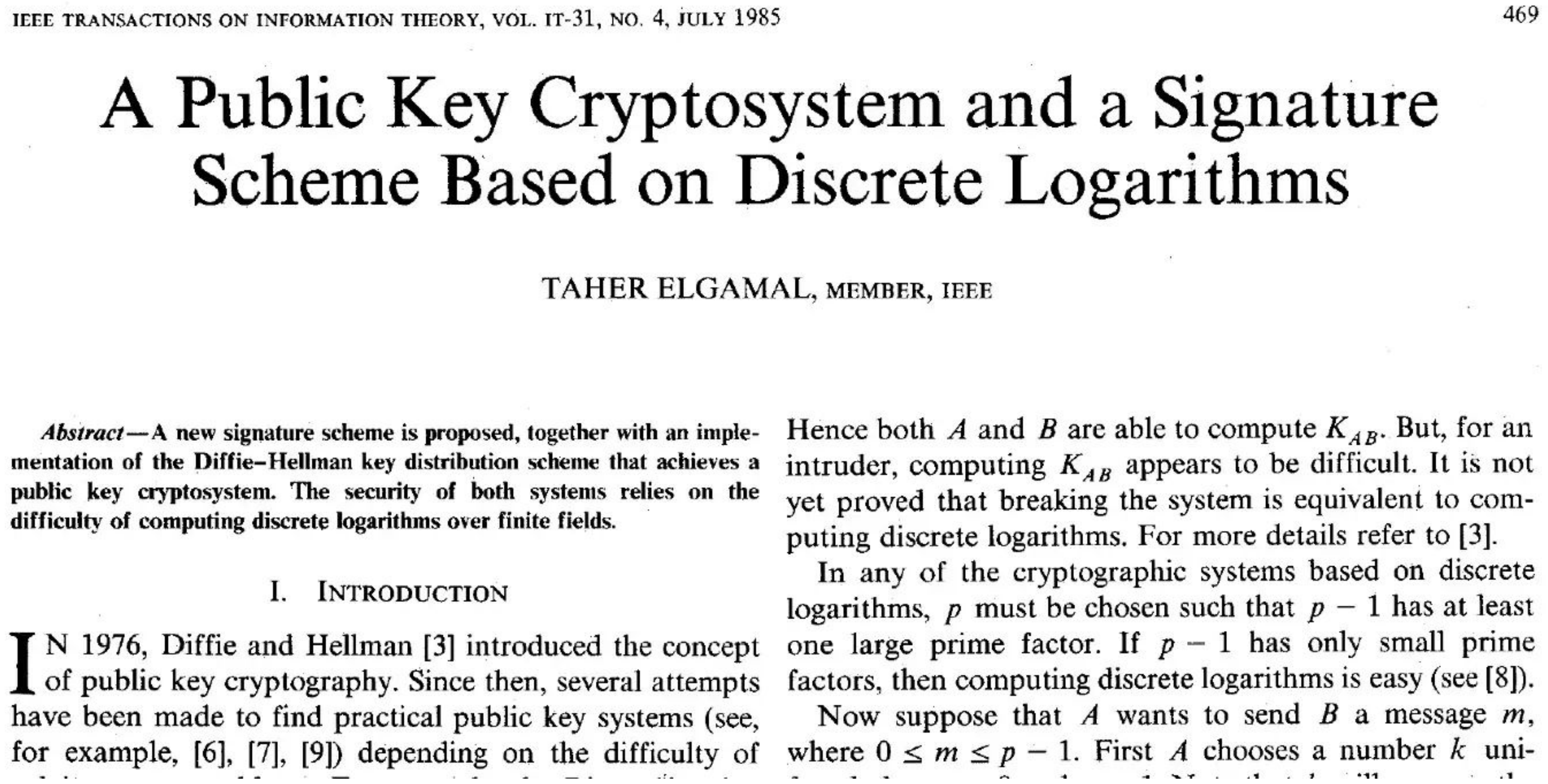


	Multiplicative	Additive	Scalar Multiplicative	XOR
RSA	Yes			
ElGamal	Yes			
Exponential ElGamal		Yes	Yes	
Elliptic Curve ElGamal		Yes	Yes	
Paillier		Yes	Yes	
Damgard-Jurik		Yes	Yes	
Okamoto–Uchiyama		Yes	Yes	
Benaloh		Yes	Yes	
Naccache–Stern		Yes	Yes	
Goldwasser–Micali				Yes

https://asecuritysite.com/homomorphic_partial/homo_types

Homomorphic Encryption (ElGamal - Multiply/Divide)





IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. IT-31, NO. 4, JULY 1985 469

A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms

TAHER ELGAMAL, MEMBER, IEEE

Abstract—A new signature scheme is proposed, together with an implementation of the Diffie–Hellman key distribution scheme that achieves a public key cryptosystem. The security of both systems relies on the difficulty of computing discrete logarithms over finite fields.

I. Introduction

In 1976, Diffie and Hellman [3] introduced the concept of public key cryptography. Since then, several attempts have been made to find practical public key systems (see, for example, [6], [7], [9]) depending on the difficulty of

Hence both A and B are able to compute K_{AB}. But, for an intruder, computing K_{AB} appears to be difficult. It is not yet proved that breaking the system is equivalent to computing discrete logarithms. For more details refer to [3].

In any of the cryptographic systems based on discrete logarithms, p must be chosen such that $p-1$ has at least one large prime factor. If $p-1$ has only small prime factors, then computing discrete logarithms is easy (see [8]).

Now suppose that A wants to send B a message m, where $0 \le m \le p-1$. First A chooses a number k uni-

$$Y = g^x \pmod{p}$$

$$a = g^k \pmod{p}$$

$$b = y^k M \pmod{p}$$

$$M = \frac{b}{a^x} \pmod{p}$$

$$a_1 = g^{k_1} \pmod{p}$$

$$a_2 = g^{k_2} \pmod{p}$$

$$b_1 = y^{k_1} M_1 \pmod{p}$$

$$b_2 = y^{k_2} M_2 \pmod{p}$$

$$a = a_1 \times a_2 = g^{k_1} \times g^{k_2} = g^{k_1+k_2} \pmod{p}$$

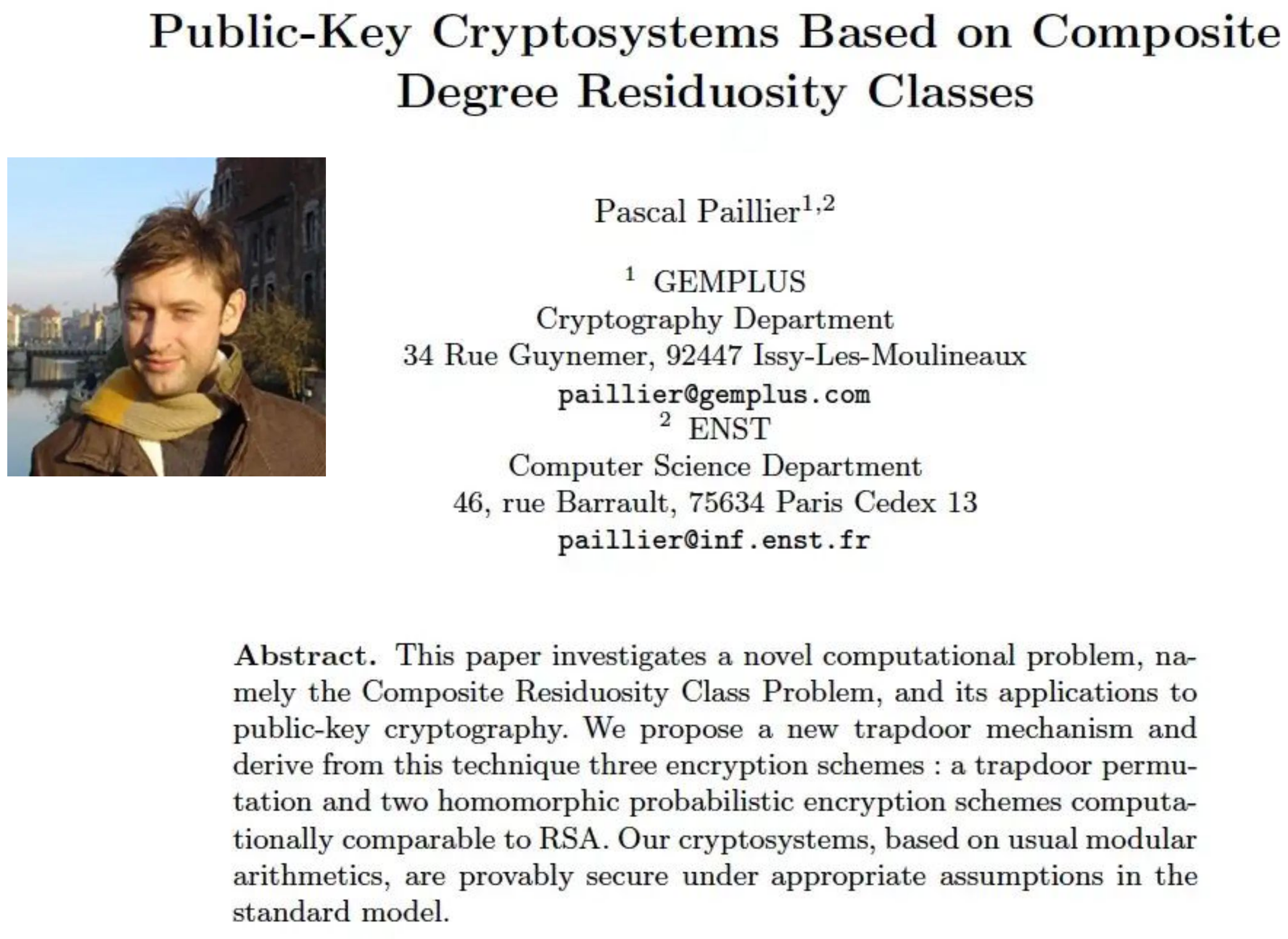
$$b = b_1 \times b_2 = Y^{k_1} M_1 \times Y^{k_2} M_2 = Y^{k_1+k_2} M_1 M_2 \pmod{p}$$

$$M = \frac{b}{a^x} = \frac{Y^{k_1+k_2} M_1 M_2}{(g^{(k_1+k_2)})^x} = \frac{Y^{k_1+k_2} M_1 M_2}{g^{(k_1+k_2)x}} = \frac{Y^{k_1+k_2} M_1 M_2}{(g^x)^{(k_1+k_2)}} = \frac{Y^{(k_1+k_2)} M_1 M_2}{Y^{(k_1+k_2)}} = M_1 M_2 \pmod{p}$$

ElGamal Multiply

ElGamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, *31*(4), 469-472.

Homomorphic Encryption (Add/Subtract)



Public-Key Cryptosystems Based on Composite Degree Residuosity Classes

Pascal Paillier[1,2]

[1] GEMPLUS
Cryptography Department
34 Rue Guynemer, 92447 Issy-Les-Moulineaux
paillier@gemplus.com
[2] ENST
Computer Science Department
46, rue Barrault, 75634 Paris Cedex 13
paillier@inf.enst.fr

Abstract. This paper investigates a novel computational problem, namely the Composite Residuosity Class Problem, and its applications to public-key cryptography. We propose a new trapdoor mechanism and derive from this technique three encryption schemes : a trapdoor permutation and two homomorphic probabilistic encryption schemes computationally comparable to RSA. Our cryptosystems, based on usual modular arithmetics, are provably secure under appropriate assumptions in the standard model.

$$g \in \mathbb{Z}^*_{N^2}$$

$$\mu = (L(g^\lambda \pmod{n})^2))^{-1} \pmod{N}$$

$$c = g^m \cdot r^N \pmod{N^2}$$

$$m = L(c^\lambda \pmod{N})^2) \cdot \mu \pmod{N}$$

$$C_1 = g^{m_1} \cdot r_1^N \pmod{N^2}$$

$$C_2 = g^{m_2} \cdot r_2^N \pmod{N^2}$$

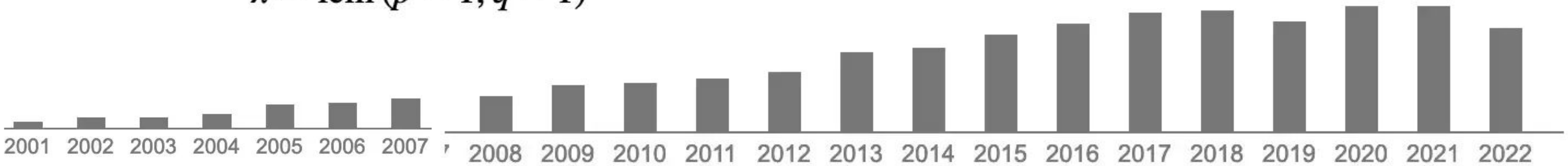
$$C_1 \cdot C_2 = g^{m_1} \cdot r_1^N \cdot g^{m_2} \cdot r_2^N \pmod{N^2}$$

$$C_1 \cdot C_2 = g^{m_1+m_2} \cdot r_1^N \cdot r_2^N \pmod{N^2}$$

$$N = pq$$

$$PHI = (p-1)(q-1)$$

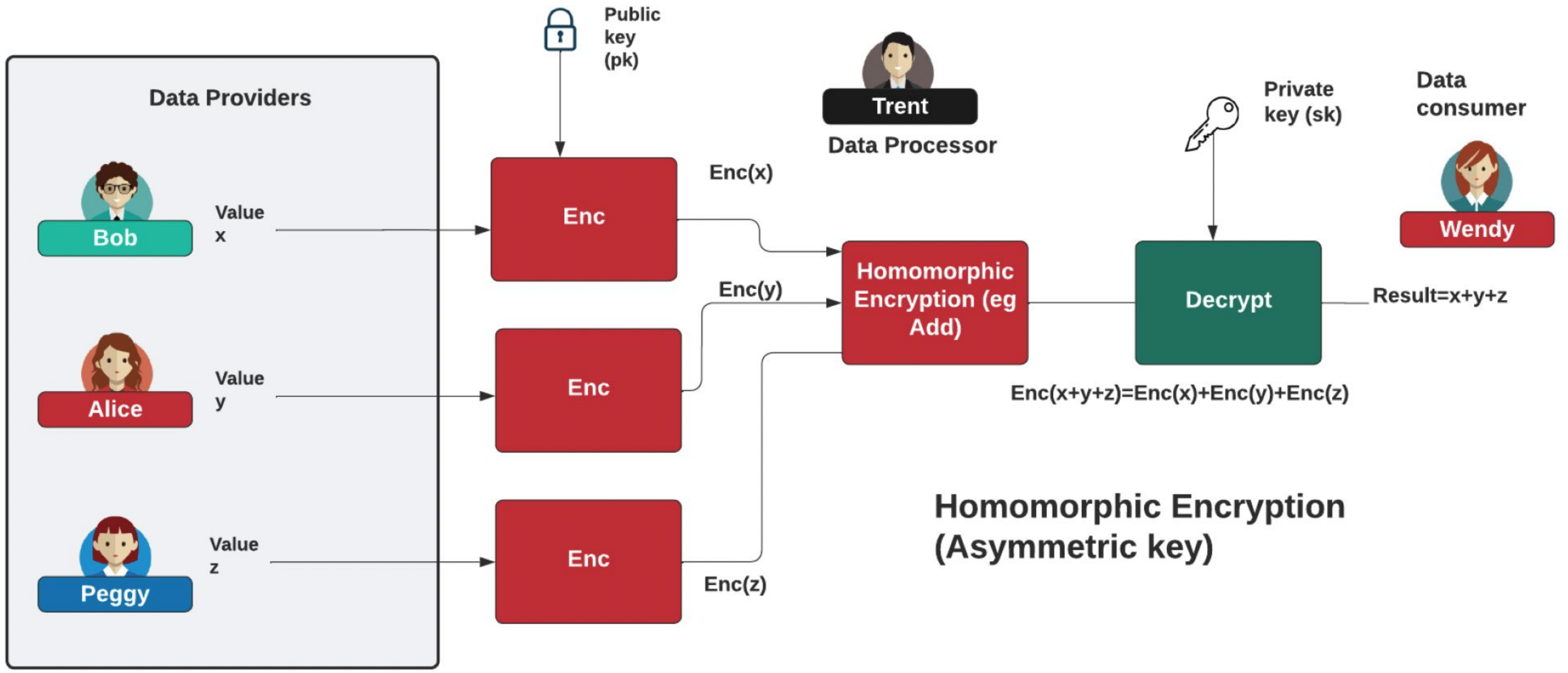
$$\lambda = \operatorname{lcm}(p-1, q-1)$$



Paillier with Kryptology

Paillier, P. (1999, May). Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques* (pp. 223-238). Springer, Berlin, Heidelberg.

FHE



Public key (pk)

Trent

Data Processor

Private key (sk)

Data consumer

Data Providers

Bob

Value x

Alice

Value y

Peggy

Value z

Enc

Enc

Enc

Enc(x)

Enc(y)

Enc(z)

Homomorphic Encryption (eg Add)

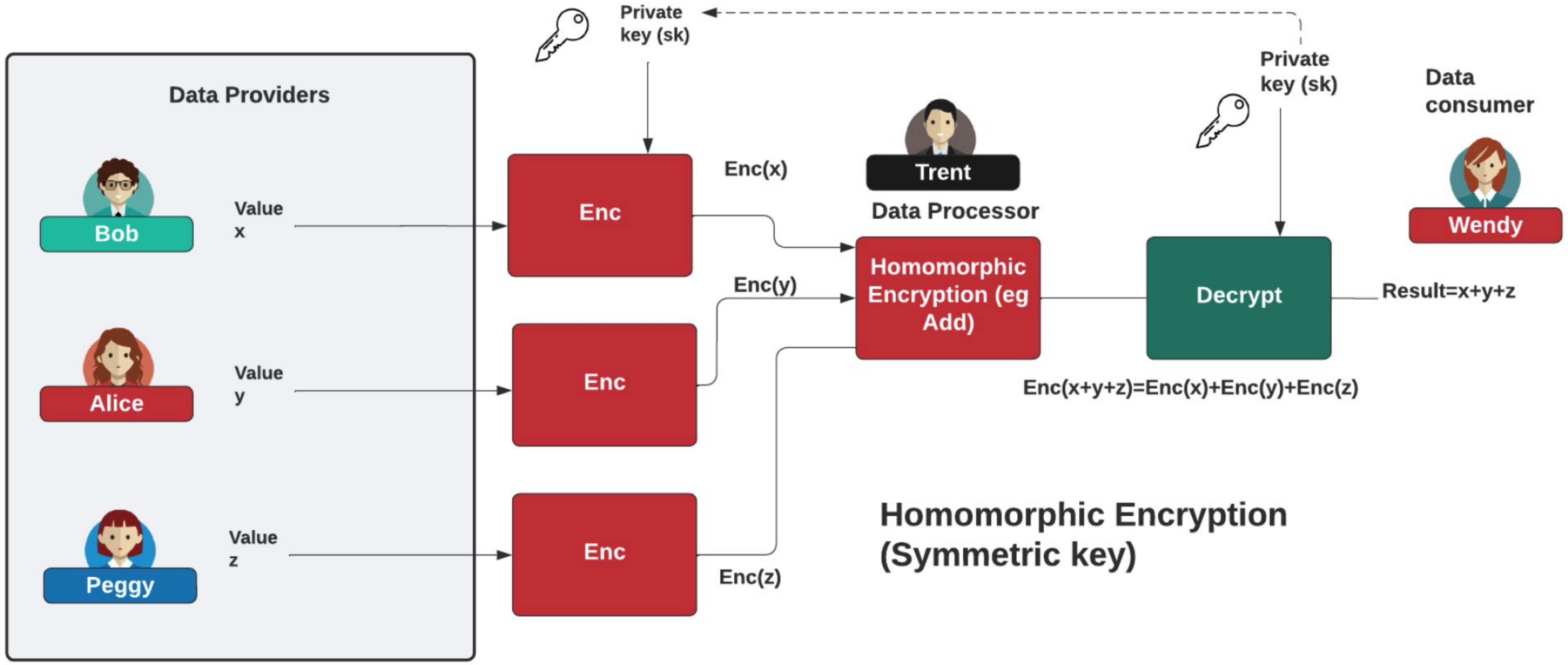
Decrypt

Wendy

Result=x+y+z

Enc(x+y+z)=Enc(x)+Enc(y)+Enc(z)

Homomorphic Encryption (Asymmetric key)



Private key (sk)

Private key (sk)

Data consumer

Data Providers

Trent

Data Processor

Bob

Value x

Alice

Value y

Peggy

Value z

Enc

Enc

Enc

Enc(x)

Enc(y)

Enc(z)

Homomorphic Encryption (eg Add)

Decrypt

Wendy

Result=x+y+z

Enc(x+y+z)=Enc(x)+Enc(y)+Enc(z)

Homomorphic Encryption (Symmetric key)

Polynomials

$$Enc_k(A \circ B) = Enc_k(A) \circ Enc_k(B)$$

With lattices, we use polynomials to represent our multi-dimensional spaces. In general, a polynomial can be represented by a number of coefficients (a_n ... a_0) and polynomial powers:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots a_1 x + a_0 \tag{2.33}$$

Within a lattice, we can have a point at (9, 5, 16), and then represent it with a quadratic equation of:

$$16x^2 + 5x + 9 \tag{2.34}$$

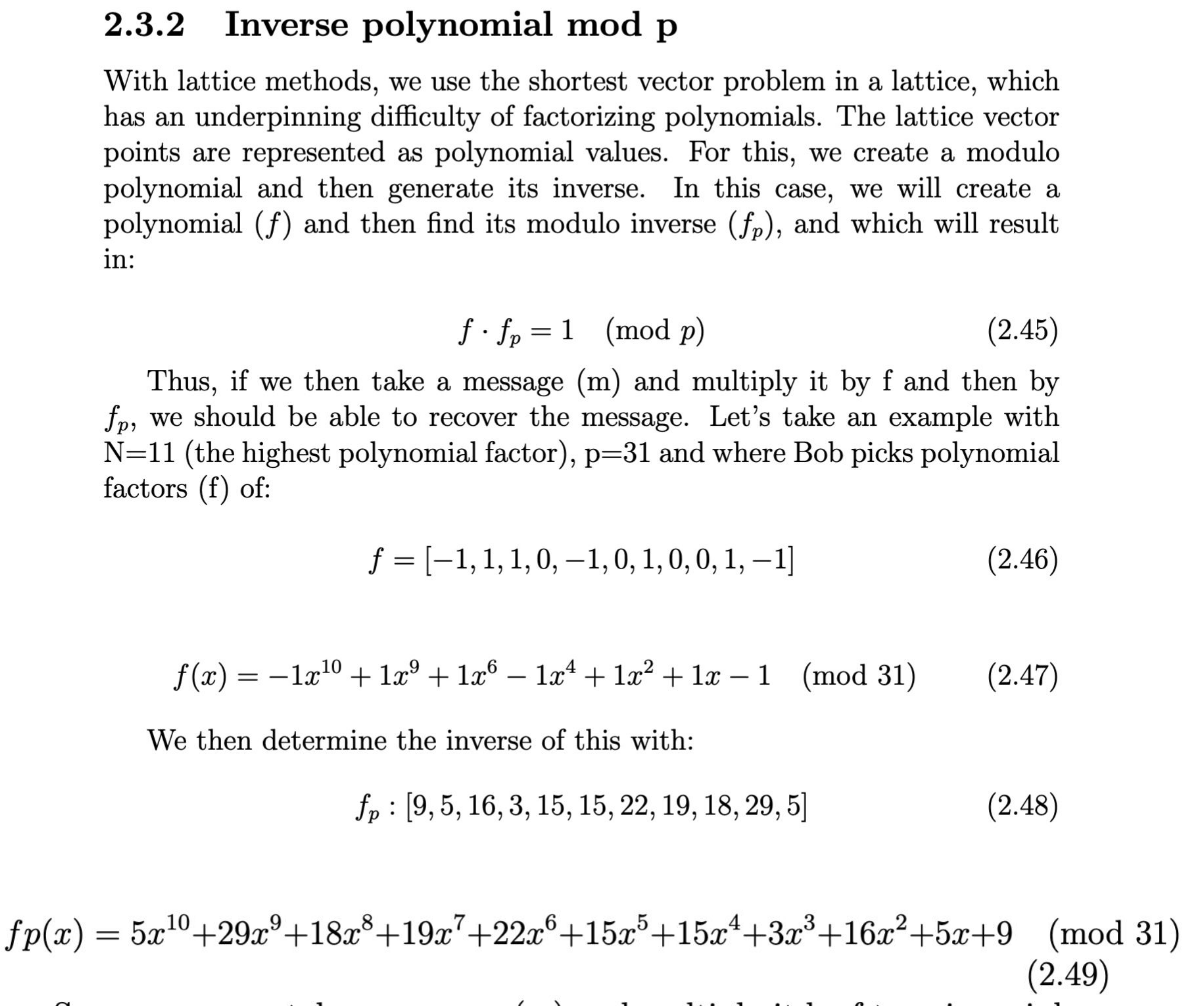
$$f = (16x^2 + 5x + 9)(2x^2 + x + 7) = 32x^4 + 26x^3 + 135x^2 + 44x + 63 \quad (2.35)$$

$$\begin{aligned} A \times B &= (10x^2 + 6x + 2) \times (12x^2 + 3x + 2) \\ &= 120x^4 + 30x^3 + 20x^2 + 72x^3 + 180x^2 + 12x + 24x^2 + 6x + 4 \\ &= 120x^4 + 102x^3 + 62x^2 + 18x + 4 \end{aligned}$$

And now, we can apply a $\pmod{13}$ operation to each of the coefficients:

$$A \times B = 120x^4 + 102x^3 + 62x^2 + 18x + 4 = 3x^4 + 11x^3 + 10x^2 + 5x + 4 \pmod{13}$$

Inv Mod p



2.3.2 Inverse polynomial mod p

With lattice methods, we use the shortest vector problem in a lattice, which has an underpinning difficulty of factorizing polynomials. The lattice vector points are represented as polynomial values. For this, we create a modulo polynomial and then generate its inverse. In this case, we will create a polynomial (f) and then find its modulo inverse (f_p), and which will result in:

$$f \cdot f_p = 1 \pmod{p} \tag{2.45}$$

Thus, if we then take a message (m) and multiply it by f and then by f_p, we should be able to recover the message. Let's take an example with N=11 (the highest polynomial factor), p=31 and where Bob picks polynomial factors (f) of:

$$f = [-1, 1, 1, 0, -1, 0, 1, 0, 0, 1, -1] \tag{2.46}$$

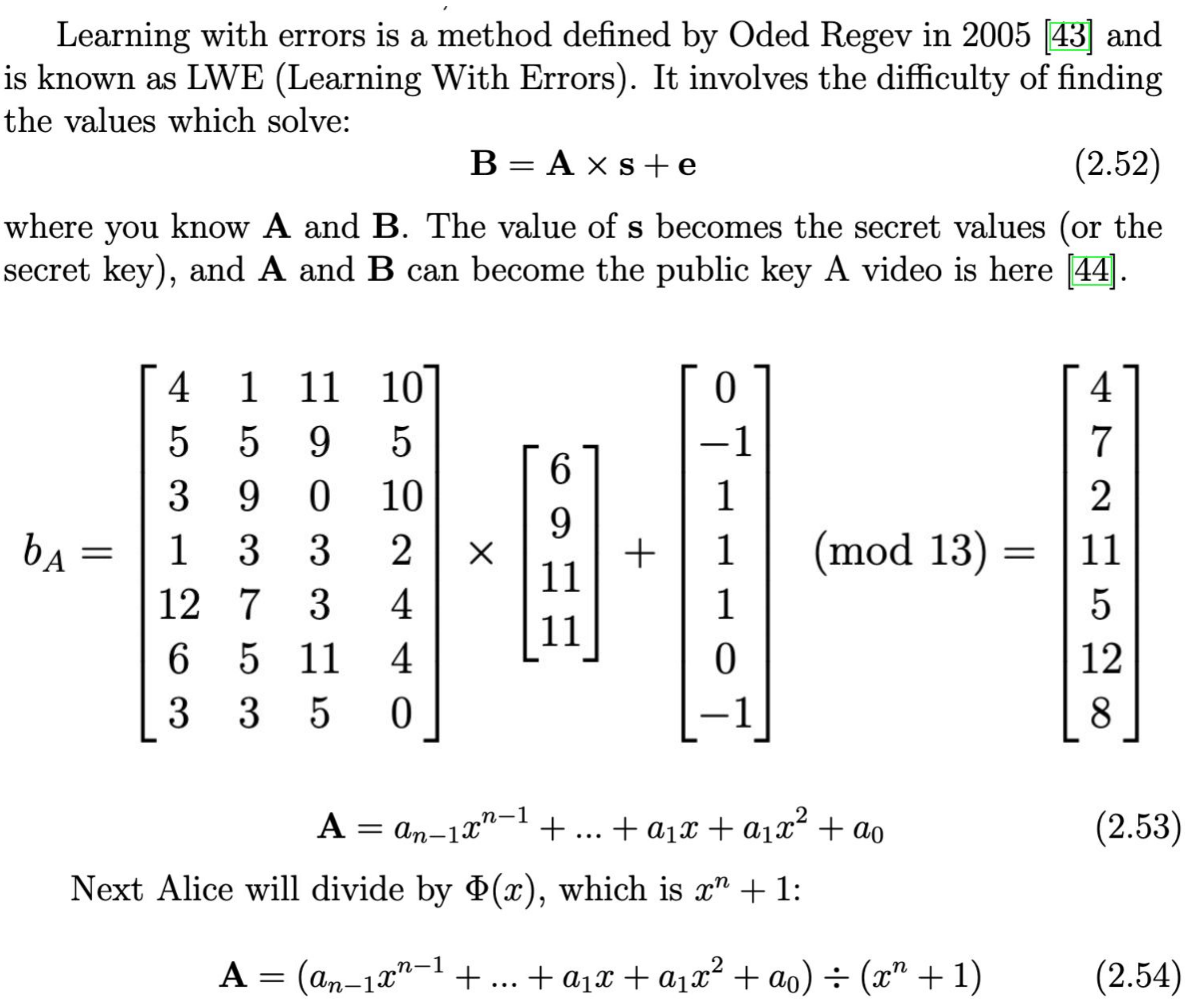
$$f(x) = -1x^{10} + 1x^9 + 1x^6 - 1x^4 + 1x^2 + 1x - 1 \pmod{31} \tag{2.47}$$

We then determine the inverse of this with:

$$f_p : [9, 5, 16, 3, 15, 15, 22, 19, 18, 29, 5] \tag{2.48}$$

$$fp(x) = 5x^{10}+29x^9+18x^8+19x^7+22x^6+15x^5+15x^4+3x^3+16x^2+5x+9 \pmod{31} \tag{2.49}$$

RLWE



Learning with errors is a method defined by Oded Regev in 2005 [43] and is known as LWE (Learning With Errors). It involves the difficulty of finding the values which solve:

$$\mathbf{B} = \mathbf{A} \times \mathbf{s} + \mathbf{e} \tag{2.52}$$

where you know **A** and **B**. The value of **s** becomes the secret values (or the secret key), and **A** and **B** can become the public key A video is here [44].

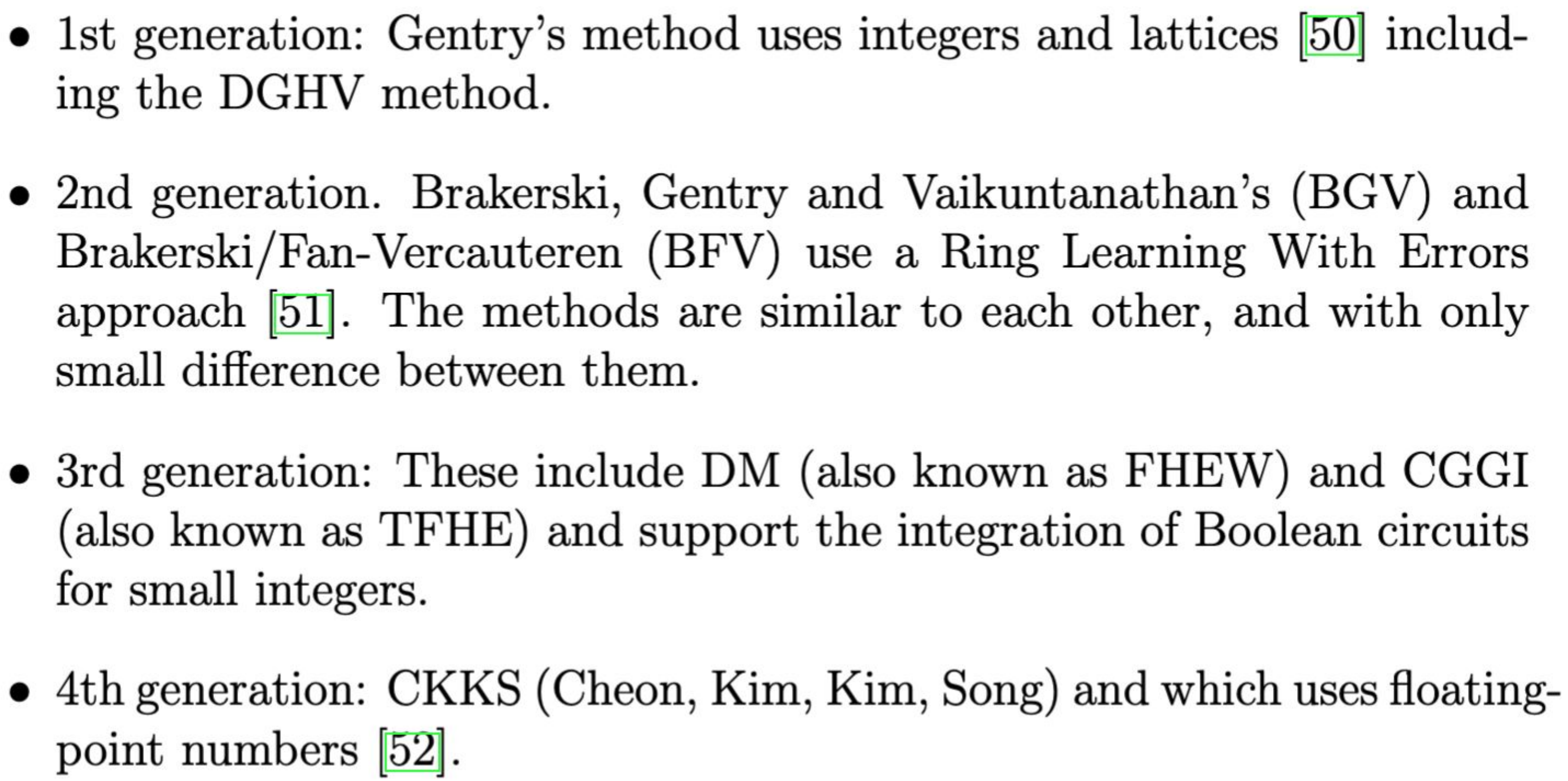
$$b_A = \begin{bmatrix} 4 & 1 & 11 & 10 \\ 5 & 5 & 9 & 5 \\ 3 & 9 & 0 & 10 \\ 1 & 3 & 3 & 2 \\ 12 & 7 & 3 & 4 \\ 6 & 5 & 11 & 4 \\ 3 & 3 & 5 & 0 \end{bmatrix} \times \begin{bmatrix} 6 \\ 9 \\ 11 \\ 11 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \\ 1 \\ 1 \\ 1 \\ 0 \\ -1 \end{bmatrix} \pmod{13} = \begin{bmatrix} 4 \\ 7 \\ 2 \\ 11 \\ 5 \\ 12 \\ 8 \end{bmatrix}$$

$$\mathbf{A} = a_{n-1}x^{n-1} + ... + a_1x + a_1x^2 + a_0 \tag{2.53}$$

Next Alice will divide by $\Phi(x)$, which is $x^n + 1$:

$$\mathbf{A} = (a_{n-1}x^{n-1} + ... + a_1x + a_1x^2 + a_0) \div (x^n + 1) \tag{2.54}$$

FHE



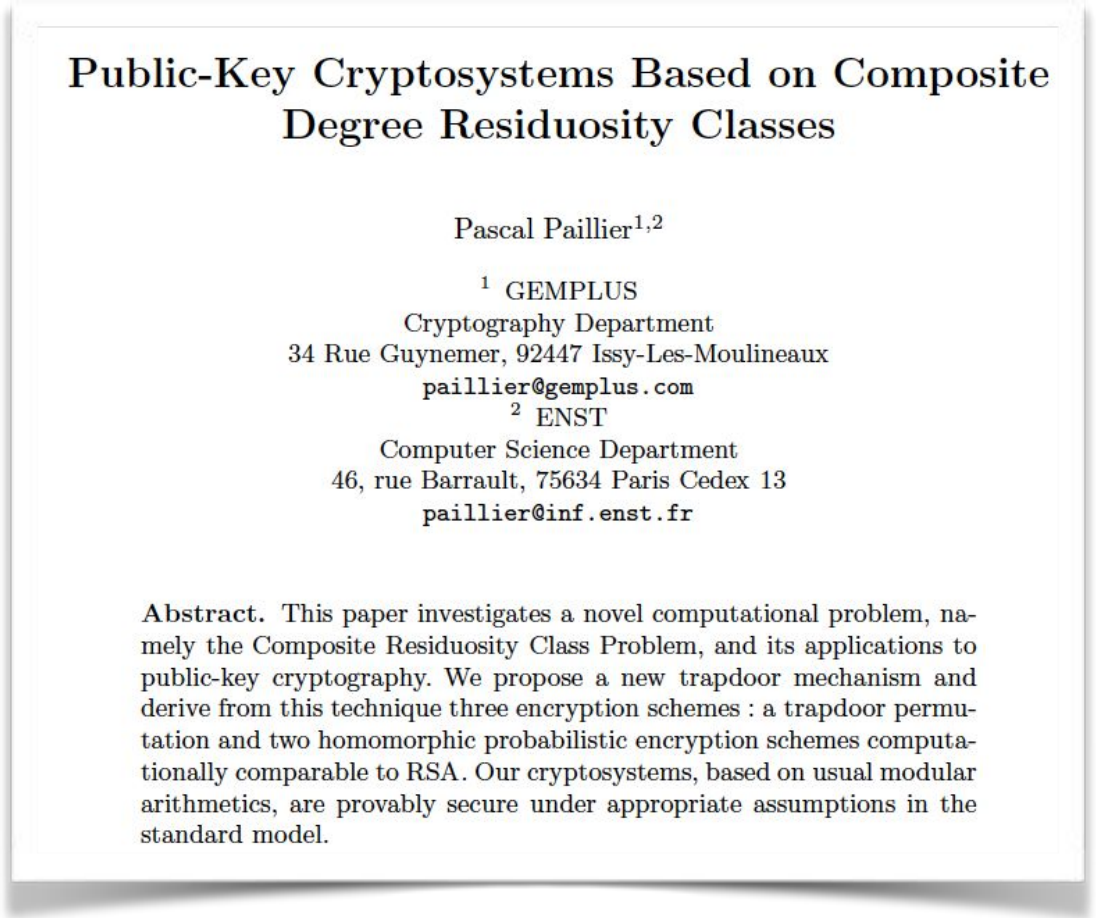
- 1st generation: Gentry's method uses integers and lattices [50] including the DGHV method.

- 2nd generation. Brakerski, Gentry and Vaikuntanathan's (BGV) and Brakerski/Fan-Vercauteren (BFV) use a Ring Learning With Errors approach [51]. The methods are similar to each other, and with only small difference between them.

- 3rd generation: These include DM (also known as FHEW) and CGGI (also known as TFHE) and support the integration of Boolean circuits for small integers.

- 4th generation: CKKS (Cheon, Kim, Kim, Song) and which uses floating-point numbers [52].

Four Generations of FHE

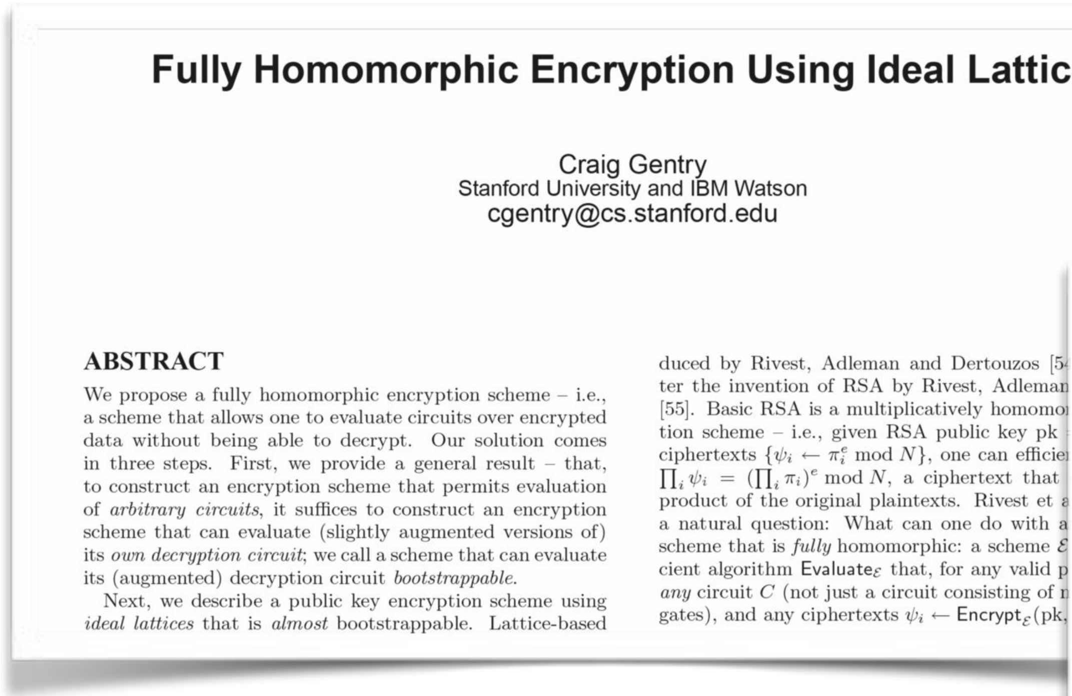


Public-Key Cryptosystems Based on Composite Degree Residuosity Classes

Pascal Paillier[1,2]

[1] GEMPLUS
Cryptography Department
34 Rue Guynemer, 92447 Issy-Les-Moulineaux
paillier@gemplus.com
[2] ENST
Computer Science Department
46, rue Barrault, 75634 Paris Cedex 13
paillier@inf.enst.fr

Abstract. This paper investigates a novel computational problem, namely the Composite Residuosity Class Problem, and its applications to public-key cryptography. We propose a new trapdoor mechanism and derive from this technique three encryption schemes : a trapdoor permutation and two homomorphic probabilistic encryption schemes computationally comparable to RSA. Our cryptosystems, based on usual modular arithmetics, are provably secure under appropriate assumptions in the standard model.



Fully Homomorphic Encryption Using Ideal Lattices

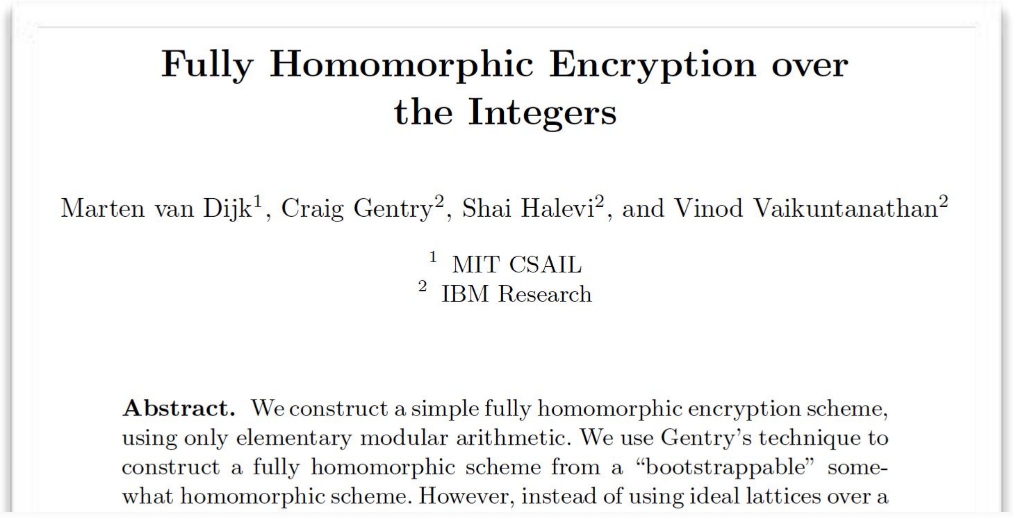
Craig Gentry
Stanford University and IBM Watson
cgentry@cs.stanford.edu

ABSTRACT

We propose a fully homomorphic encryption scheme – i.e., a scheme that allows one to evaluate circuits over encrypted data without being able to decrypt. Our solution comes in three steps. First, we provide a general result – that, to construct an encryption scheme that permits evaluation of *arbitrary circuits*, it suffices to construct an encryption scheme that can evaluate (slightly augmented versions of) its *own decryption circuit*; we call a scheme that can evaluate its (augmented) decryption circuit *bootstrappable*.

Next, we describe a public key encryption scheme using *ideal lattices* that is *almost* bootstrappable. Lattice-based

duced by Rivest, Adleman and Dertouzos [5 ter the invention of RSA by Rivest, Adleman [55]. Basic RSA is a multiplicatively homomo tion scheme – i.e., given RSA public key pk ciphertexts $\{\psi_i \leftarrow \pi_i^e \bmod N\}$, one can efficie $\prod_i \psi_i = (\prod_i \pi_i)^e \bmod N$, a ciphertext that product of the original plaintexts. Rivest et a a natural question: What can one do with a scheme that is *fully* homomorphic: a scheme $\mathcal{E}$ cient algorithm $\mathsf{Evaluate}_{\mathcal{E}}$ that, for any valid p *any* circuit C (not just a circuit consisting of n gates), and any ciphertexts $\psi_i \leftarrow \mathsf{Encrypt}_{\mathcal{E}}(\mathrm{pk},$

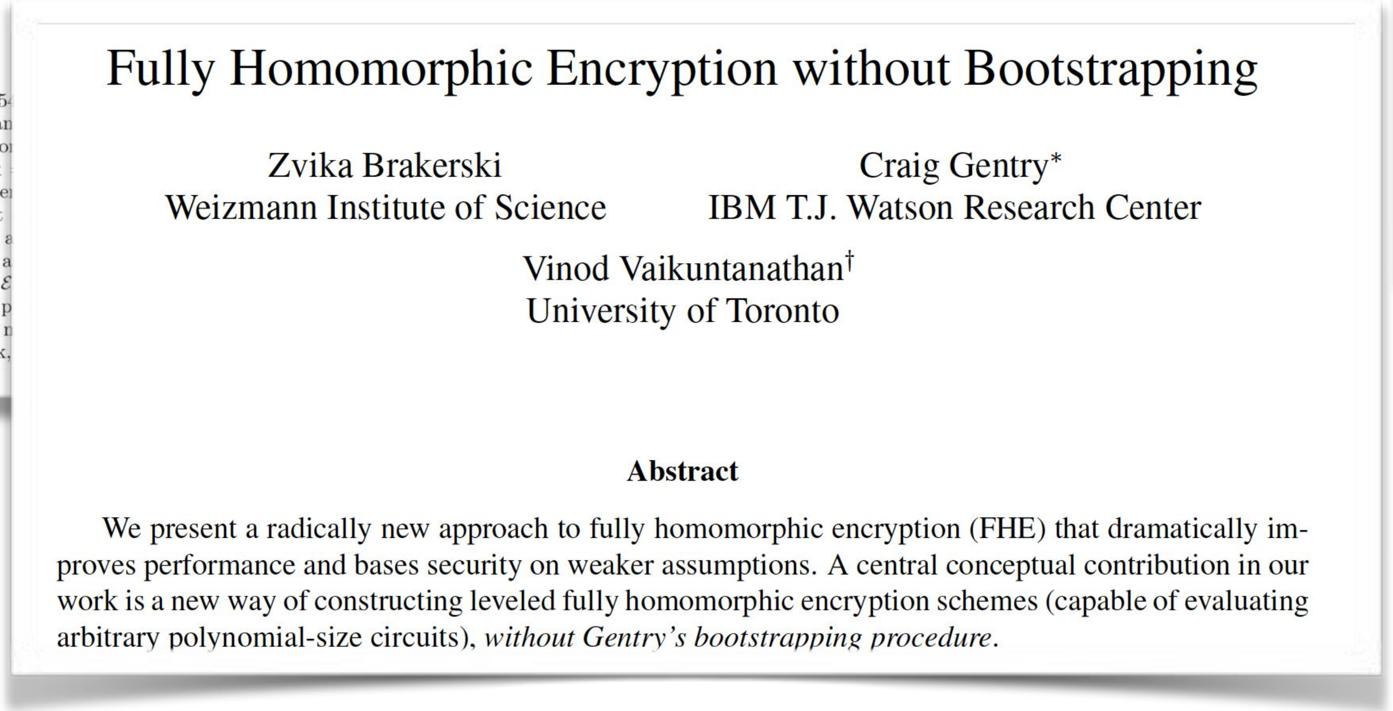


Fully Homomorphic Encryption over the Integers

Marten van Dijk[1], Craig Gentry[2], Shai Halevi[2], and Vinod Vaikuntanathan[2]

[1] MIT CSAIL
[2] IBM Research

Abstract. We construct a simple fully homomorphic encryption scheme, using only elementary modular arithmetic. We use Gentry's technique to construct a fully homomorphic scheme from a "bootstrappable" somewhat homomorphic scheme. However, instead of using ideal lattices over a



Fully Homomorphic Encryption without Bootstrapping

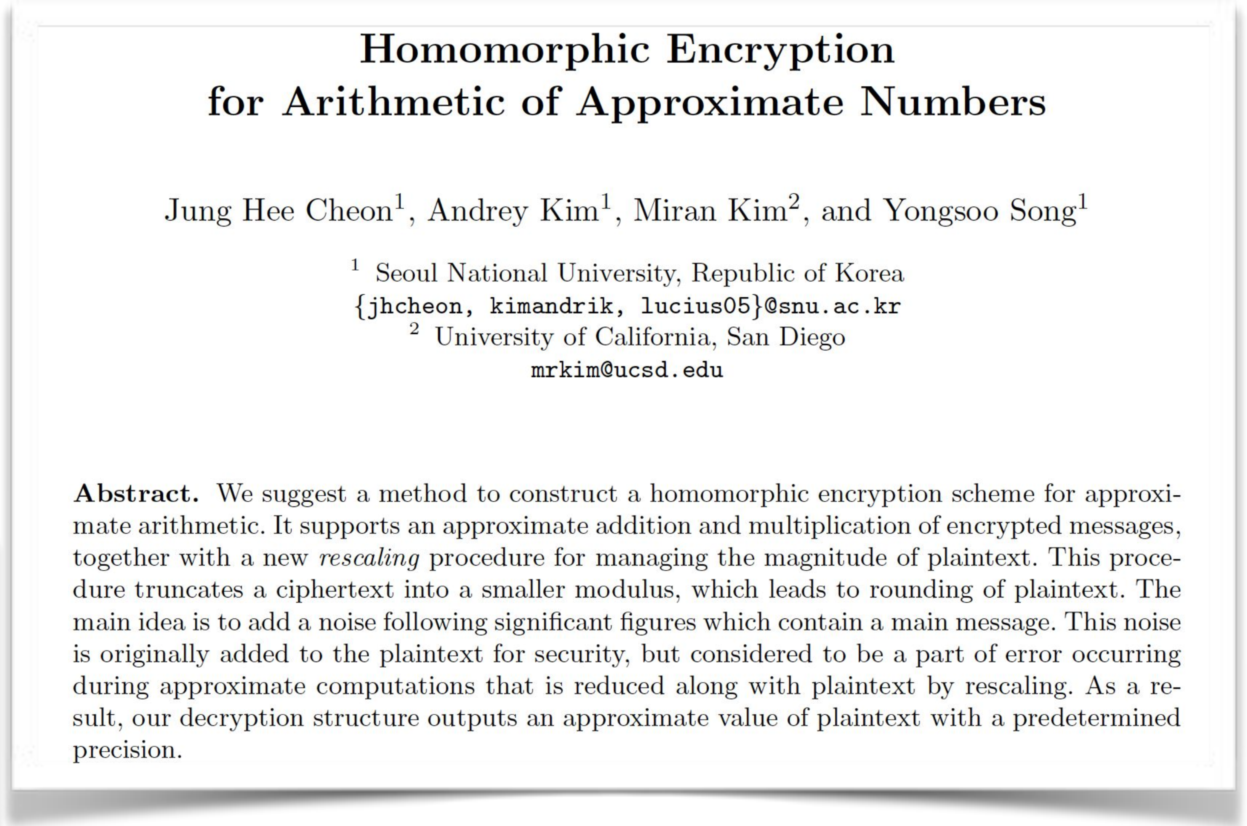
Zvika Brakerski
Weizmann Institute of Science

Craig Gentry*
IBM T.J. Watson Research Center

Vinod Vaikuntanathan†
University of Toronto

Abstract

We present a radically new approach to fully homomorphic encryption (FHE) that dramatically improves performance and bases security on weaker assumptions. A central conceptual contribution in our work is a new way of constructing leveled fully homomorphic encryption schemes (capable of evaluating arbitrary polynomial-size circuits), *without Gentry's bootstrapping procedure*.



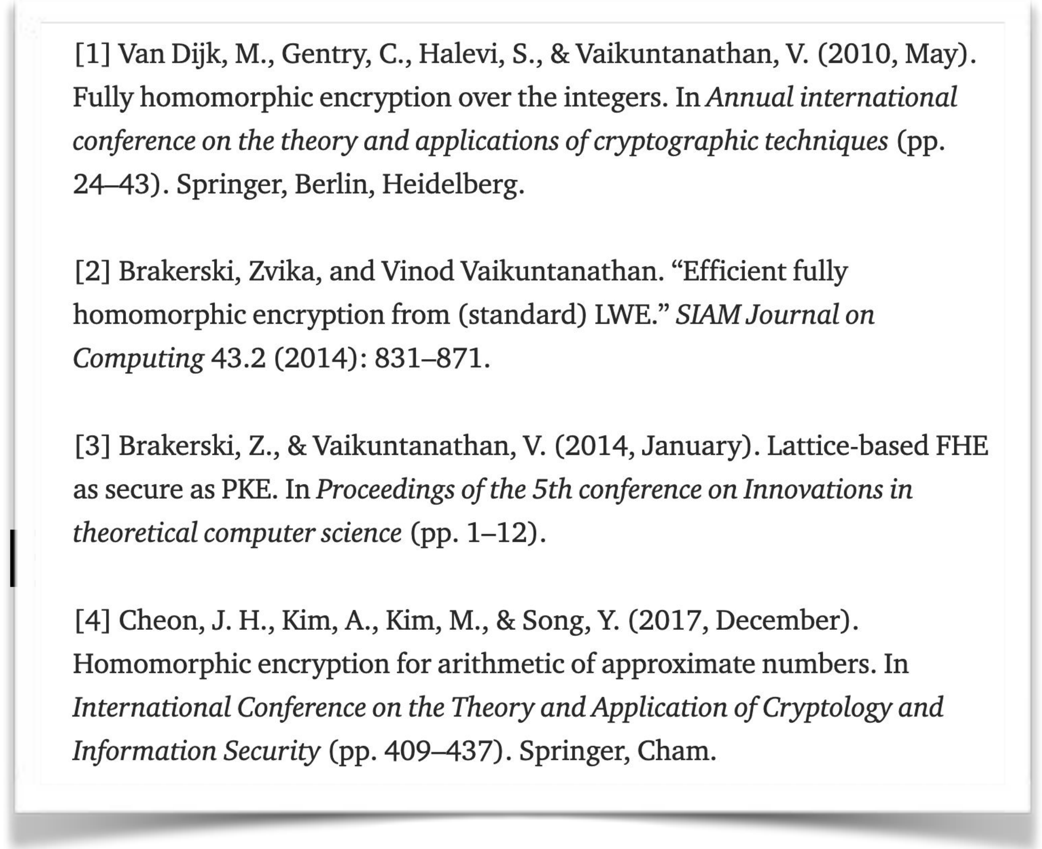
Homomorphic Encryption for Arithmetic of Approximate Numbers

Jung Hee Cheon[1], Andrey Kim[1], Miran Kim[2], and Yongsoo Song[1]

[1] Seoul National University, Republic of Korea
{jhcheon, kimandrik, lucius05}@snu.ac.kr
[2] University of California, San Diego
mrkim@ucsd.edu

Abstract. We suggest a method to construct a homomorphic encryption scheme for approximate arithmetic. It supports an approximate addition and multiplication of encrypted messages, together with a new *rescaling* procedure for managing the magnitude of plaintext. This procedure truncates a ciphertext into a smaller modulus, which leads to rounding of plaintext. The main idea is to add a noise following significant figures which contain a main message. This noise is originally added to the plaintext for security, but considered to be a part of error occurring during approximate computations that is reduced along with plaintext by rescaling. As a result, our decryption structure outputs an approximate value of plaintext with a predetermined precision.

- 1st generation: Gentry's method uses integers and lattices [1] including the **DGHV method**.
- 2nd generation. Brakerski, Gentry and Vaikuntanathan's (**BGV**) work in 2014 for FHE using Learning With l
- 3rd generation: Lattice-based methods as defined by Brakerski and Vaikuntanathan [3].
- 4th generation: **CKKS** (Cheon, Kim, Kim, Song) and which uses floating-point numbers [4]. Here.

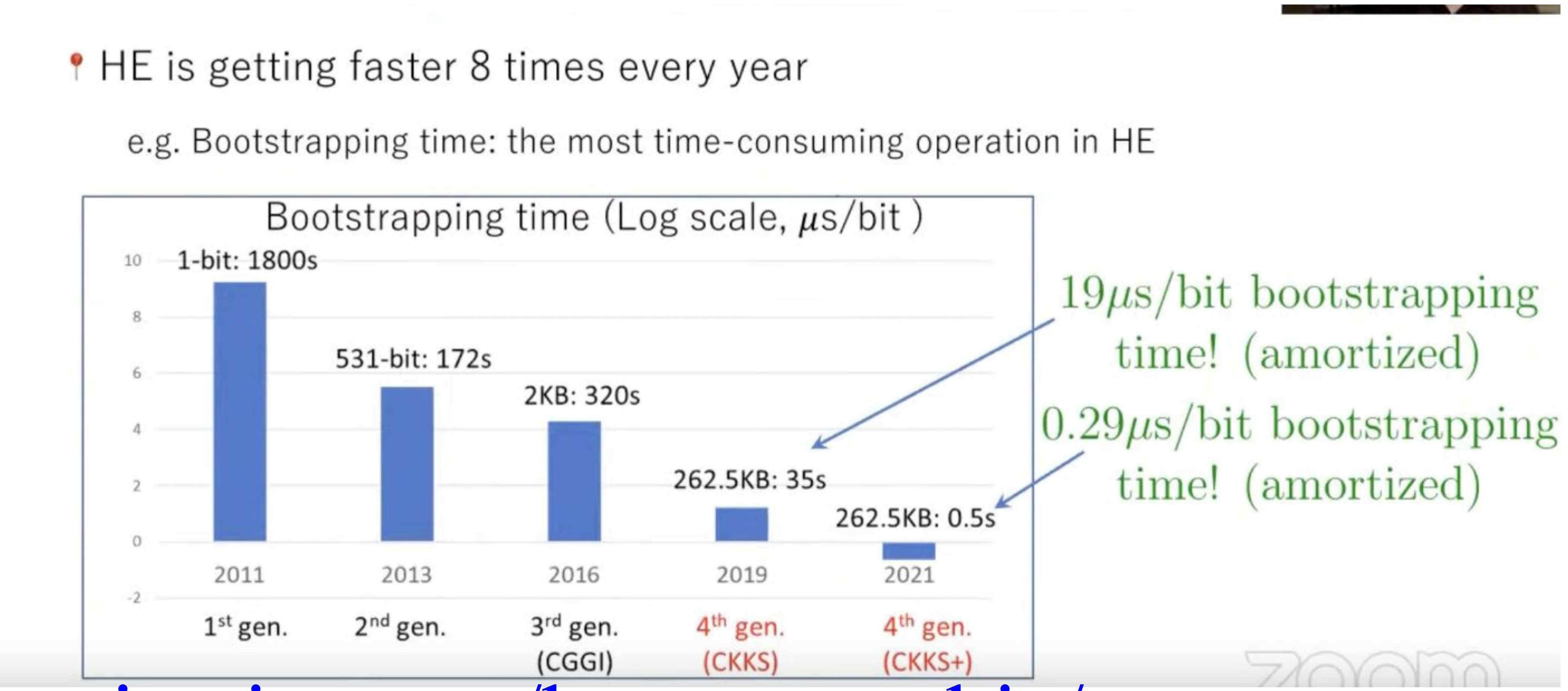


[1] Van Dijk, M., Gentry, C., Halevi, S., & Vaikuntanathan, V. (2010, May). Fully homomorphic encryption over the integers. In *Annual international conference on the theory and applications of cryptographic techniques* (pp. 24–43). Springer, Berlin, Heidelberg.

[2] Brakerski, Zvika, and Vinod Vaikuntanathan. "Efficient fully homomorphic encryption from (standard) LWE." *SIAM Journal on Computing* 43.2 (2014): 831–871.

[3] Brakerski, Z., & Vaikuntanathan, V. (2014, January). Lattice-based FHE as secure as PKE. In *Proceedings of the 5th conference on Innovations in theoretical computer science* (pp. 1–12).

[4] Cheon, J. H., Kim, A., Kim, M., & Song, Y. (2017, December). Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security* (pp. 409–437). Springer, Cham.



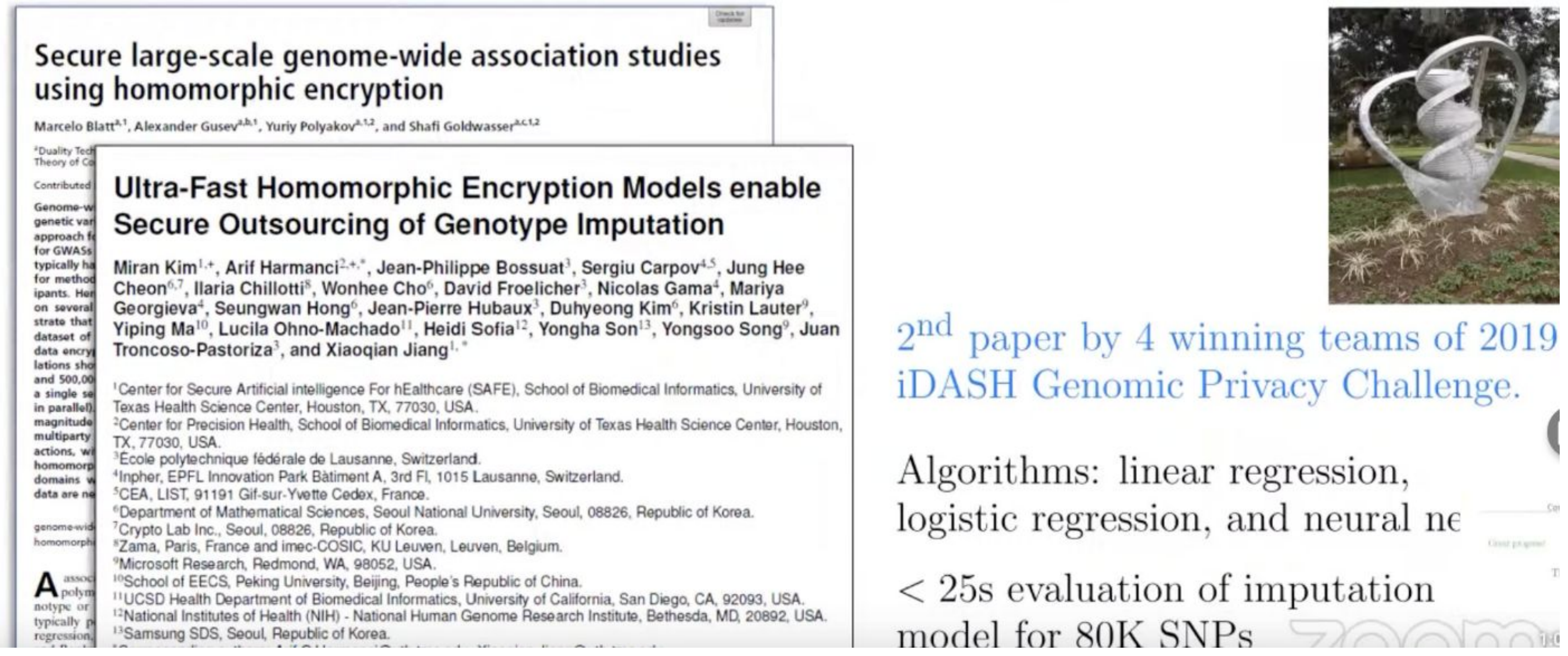
HE is getting faster 8 times every year

e.g. Bootstrapping time: the most time-consuming operation in HE

Bootstrapping time (Log scale, μs/bit)

$19\mu s$/bit bootstrapping time! (amortized)

$0.29\mu s$/bit bootstrapping time! (amortized)



Secure large-scale genome-wide association studies using homomorphic encryption

Marcelo Blatt[a,1], Alexander Gusev[a,b,1], Yuriy Polyakov[a,1,2], and Shafi Goldwasser[a,c,1,2]

Ultra-Fast Homomorphic Encryption Models enable Secure Outsourcing of Genotype Imputation

Miran Kim[1,+], Arif Harmanci[2,+,*], Jean-Philippe Bossuat[3], Sergiu Carpov[4,5], Jung Hee Cheon[6,7], Ilaria Chillotti[8], Wonhee Cho[6], David Froelicher[3], Nicolas Gama[4], Mariya Georgieva[4], Seungwan Hong[6], Jean-Pierre Hubaux[3], Duhyeong Kim[6], Kristin Lauter[9], Yiping Ma[10], Lucila Ohno-Machado[11], Heidi Sofia[12], Yongha Son[13], Yongsoo Song[9], Juan Troncoso-Pastoriza[3], and Xiaoqian Jiang[1,*]

[1] Center for Secure Artificial intelligence For hEalthcare (SAFE), School of Biomedical Informatics, University of Texas Health Science Center, Houston, TX, 77030, USA.
[2] Center for Precision Health, School of Biomedical Informatics, University of Texas Health Science Center, Houston, TX, 77030, USA.
[3] École polytechnique fédérale de Lausanne, Switzerland.
[4] Inpher, EPFL Innovation Park Bàtiment A, 3rd Fl, 1015 Lausanne, Switzerland.
[5] CEA, LIST, 91191 Gif-sur-Yvette Cedex, France.
[6] Department of Mathematical Sciences, Seoul National University, Seoul, 08826, Republic of Korea.
[7] Crypto Lab Inc., Seoul, 08826, Republic of Korea.
[8] Zama, Paris, France and imec-COSIC, KU Leuven, Leuven, Belgium.
[9] Microsoft Research, Redmond, WA, 98052, USA.
[10] School of EECS, Peking University, Beijing, People's Republic of China.
[11] UCSD Health Department of Biomedical Informatics, University of California, San Diego, CA, 92093, USA.
[12] National Institutes of Health (NIH) - National Human Genome Research Institute, Bethesda, MD, 20892, USA.
[13] Samsung SDS, Seoul, Republic of Korea.

2nd paper by 4 winning teams of 2019 iDASH Genomic Privacy Challenge.

Algorithms: linear regression, logistic regression, and neural ne

< 25s evaluation of imputation model for 80K SNPs

https://asecuritysite.com/homomorphic/

Cryptography

Prof Bill Buchanan OBE, FRSE

http://asecuritysite.com

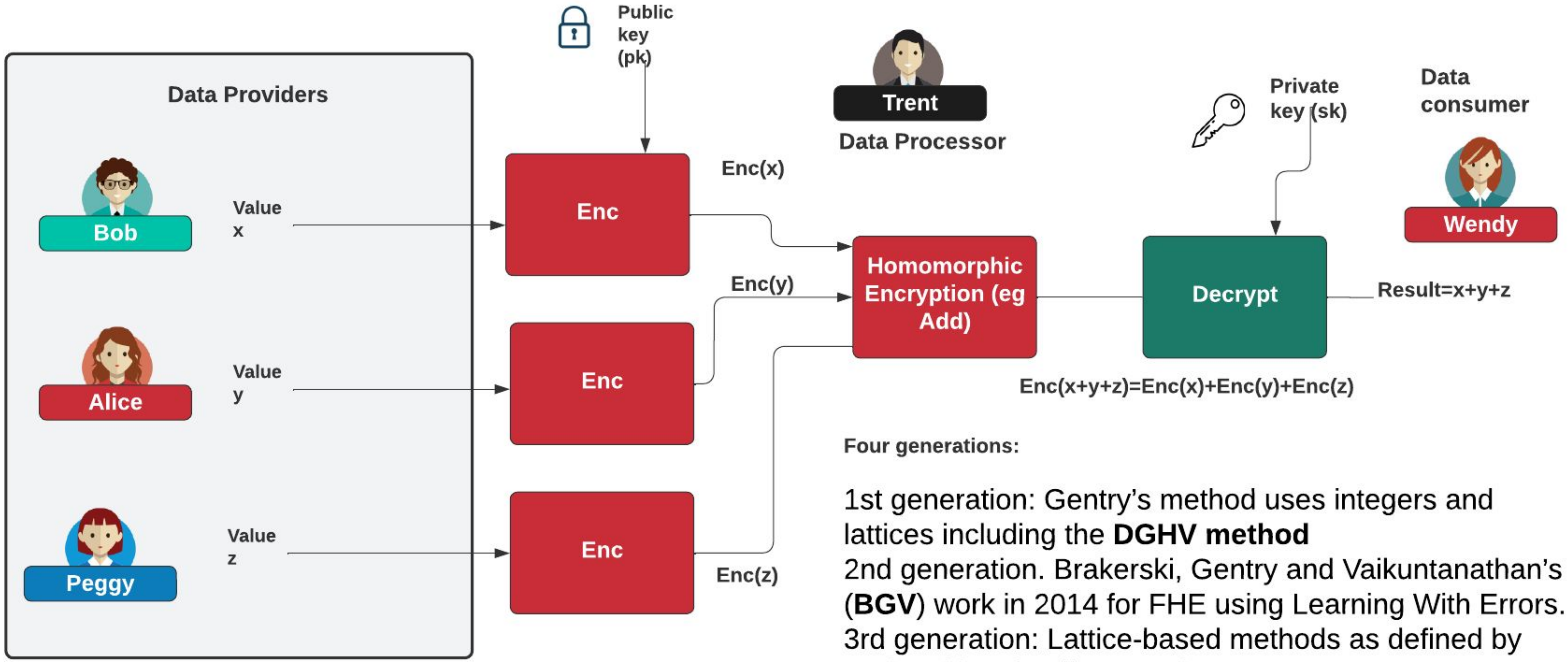
Twitter: billatnapier

Homomorphic Encryption with BFV, BGV and CKKS

BLOCKPASS IDENTITY LAB

World-leading Collaboration between Blockpass IDN and Edinburgh Napier University

Homomorphic Encryption

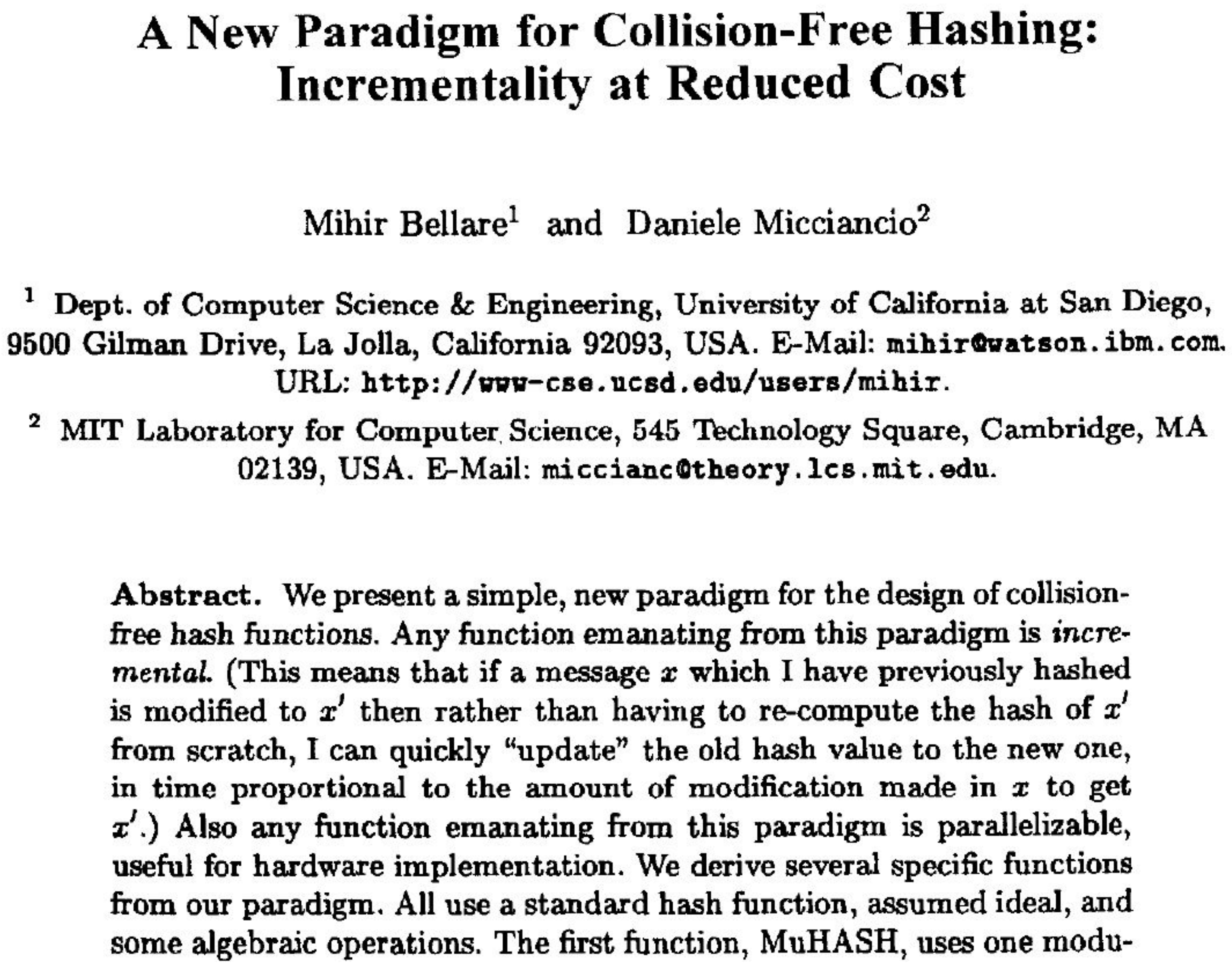


Four generations:

1st generation: Gentry's method uses integers and lattices including the **DGHV method**
2nd generation. Brakerski, Gentry and Vaikuntanathan's (**BGV**) work in 2014 for FHE using Learning With Errors.
3rd generation: Lattice-based methods as defined by Brakerski and Vaikuntanathan.
4th generation: **CKKS** (Cheon, Kim, Kim, Song) and which uses floating-point numbers.

https://asecuritysite.com/homomorphic/

Homomorphic Hashing



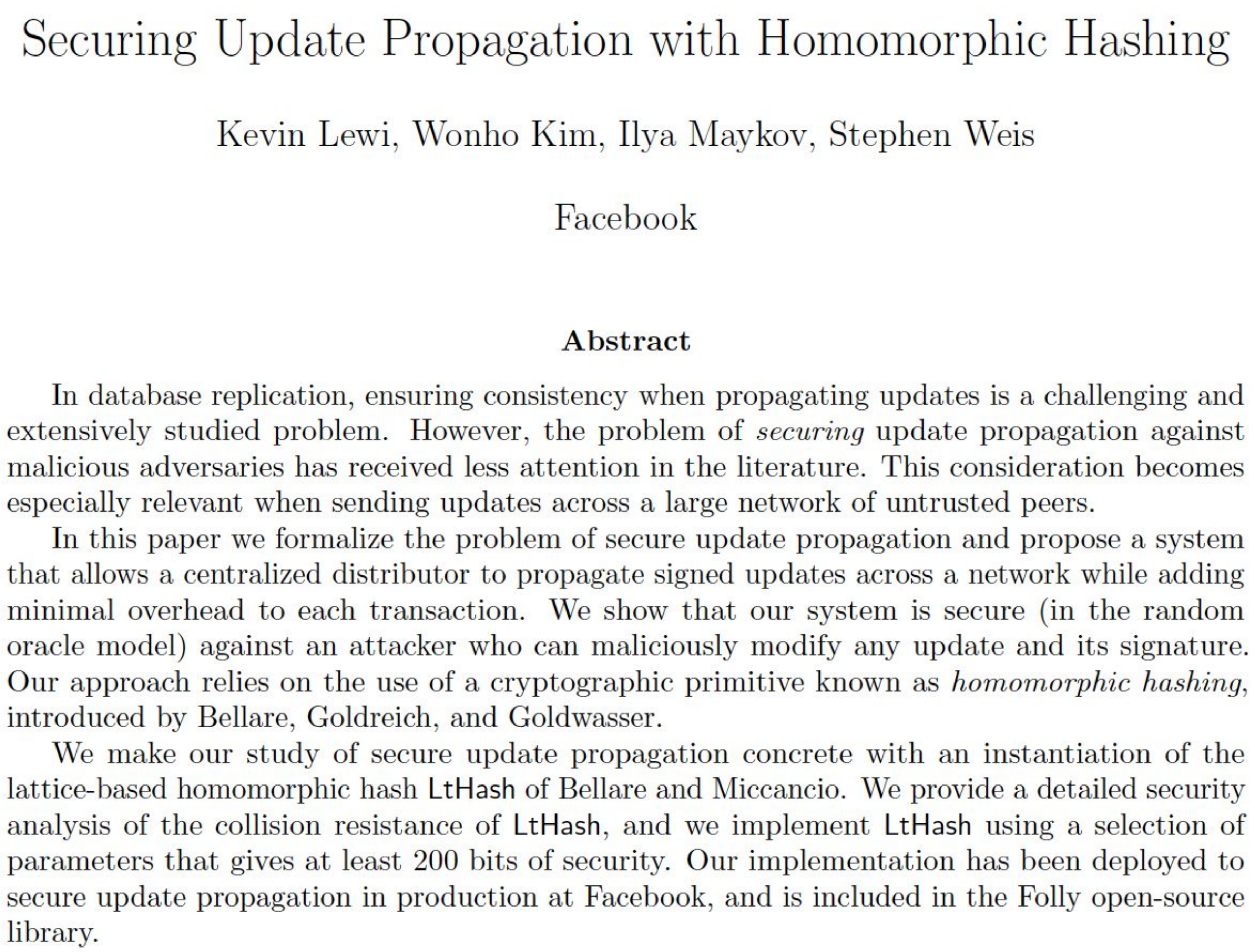
A New Paradigm for Collision-Free Hashing: Incrementality at Reduced Cost

Mihir Bellare[1] and Daniele Micciancio[2]

[1] Dept. of Computer Science & Engineering, University of California at San Diego, 9500 Gilman Drive, La Jolla, California 92093, USA. E-Mail: mihir@watson.ibm.com. URL: http://www-cse.ucsd.edu/users/mihir.

[2] MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA. E-Mail: miccianc@theory.lcs.mit.edu.

Abstract. We present a simple, new paradigm for the design of collision-free hash functions. Any function emanating from this paradigm is *incremental.* (This means that if a message x which I have previously hashed is modified to x' then rather than having to re-compute the hash of x' from scratch, I can quickly "update" the old hash value to the new one, in time proportional to the amount of modification made in x to get x'.) Also any function emanating from this paradigm is parallelizable, useful for hardware implementation. We derive several specific functions from our paradigm. All use a standard hash function, assumed ideal, and some algebraic operations. The first function, MuHASH, uses one modu-



Securing Update Propagation with Homomorphic Hashing

Kevin Lewi, Wonho Kim, Ilya Maykov, Stephen Weis

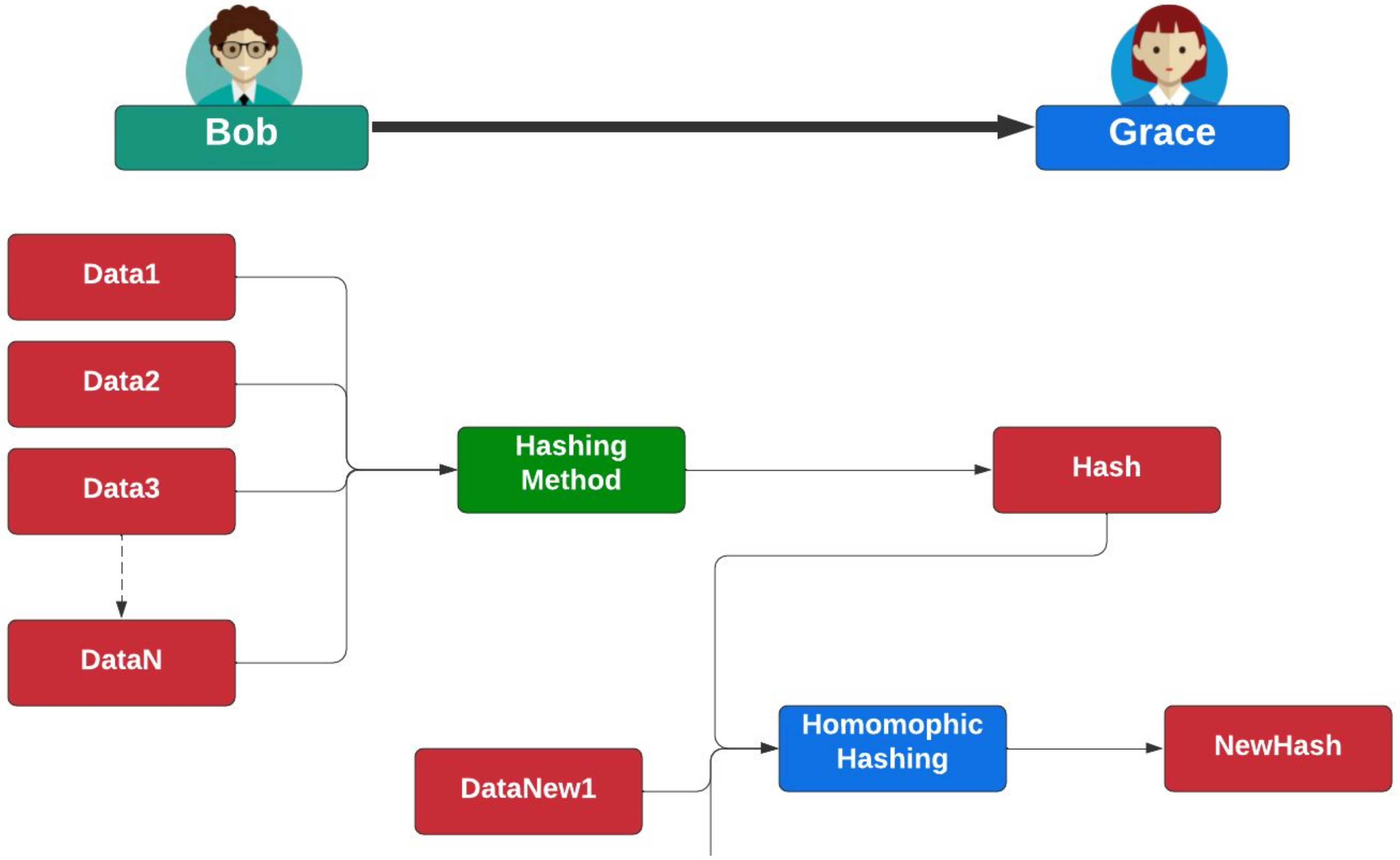
Facebook

Abstract

In database replication, ensuring consistency when propagating updates is a challenging and extensively studied problem. However, the problem of *securing* update propagation against malicious adversaries has received less attention in the literature. This consideration becomes especially relevant when sending updates across a large network of untrusted peers.

In this paper we formalize the problem of secure update propagation and propose a system that allows a centralized distributor to propagate signed updates across a network while adding minimal overhead to each transaction. We show that our system is secure (in the random oracle model) against an attacker who can maliciously modify any update and its signature. Our approach relies on the use of a cryptographic primitive known as *homomorphic hashing*, introduced by Bellare, Goldreich, and Goldwasser.

We make our study of secure update propagation concrete with an instantiation of the lattice-based homomorphic hash LtHash of Bellare and Miccancio. We provide a detailed security analysis of the collision resistance of LtHash, and we implement LtHash using a selection of parameters that gives at least 200 bits of security. Our implementation has been deployed to secure update propagation in production at Facebook, and is included in the Folly open-source library.

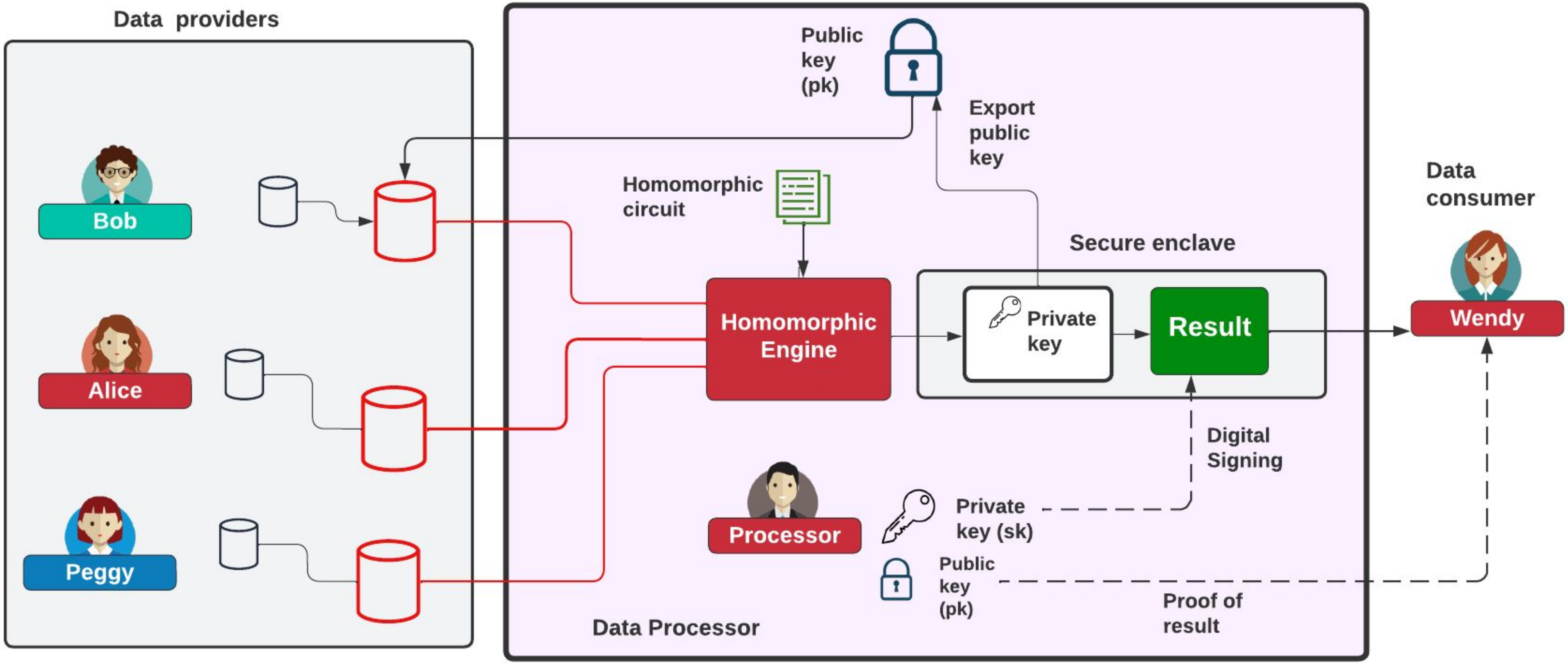


[1] Bellare, M., & Micciancio, D. (1997, May). A new paradigm for collision-free hashing: Incrementality at reduced cost. In International Conference on the Theory and Applications of Cryptographic Techniques (pp. 163–192). Springer, Berlin, Heidelberg.

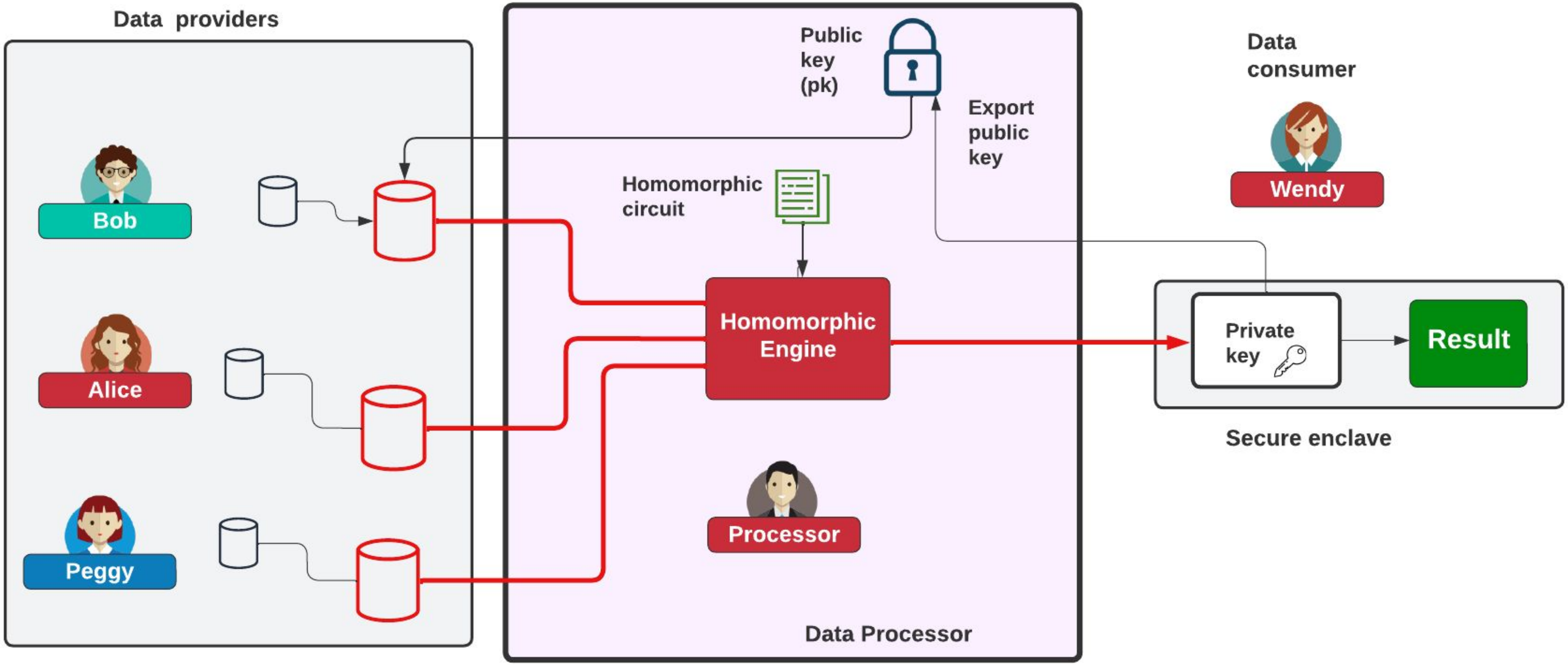
[2] Lewi, K., Kim, W., Maykov, I., & Weis, S. (2019). Securing Update Propagation with Homomorphic Hashing. Cryptology ePrint Archive.

[Homomorphic Hashing]

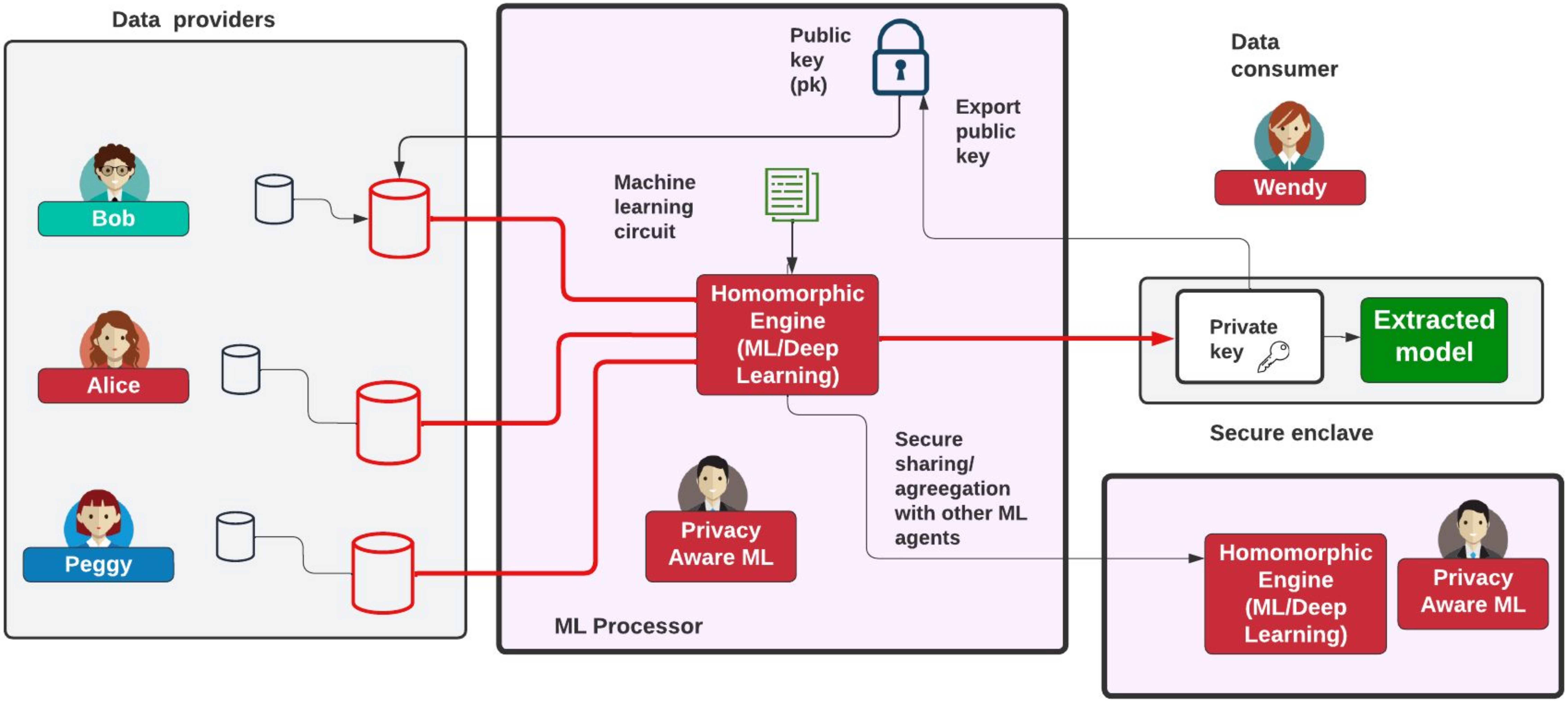
Homomorphic Processing



Homomorphic Processing



Homomorphic ML



Data providers

Bob

Alice

Peggy

ML Processor

Public key (pk)

Export public key

Machine learning circuit

Homomorphic Engine (ML/Deep Learning)

Privacy Aware ML

Secure sharing/ agreegation with other ML agents

Data consumer

Wendy

Private key

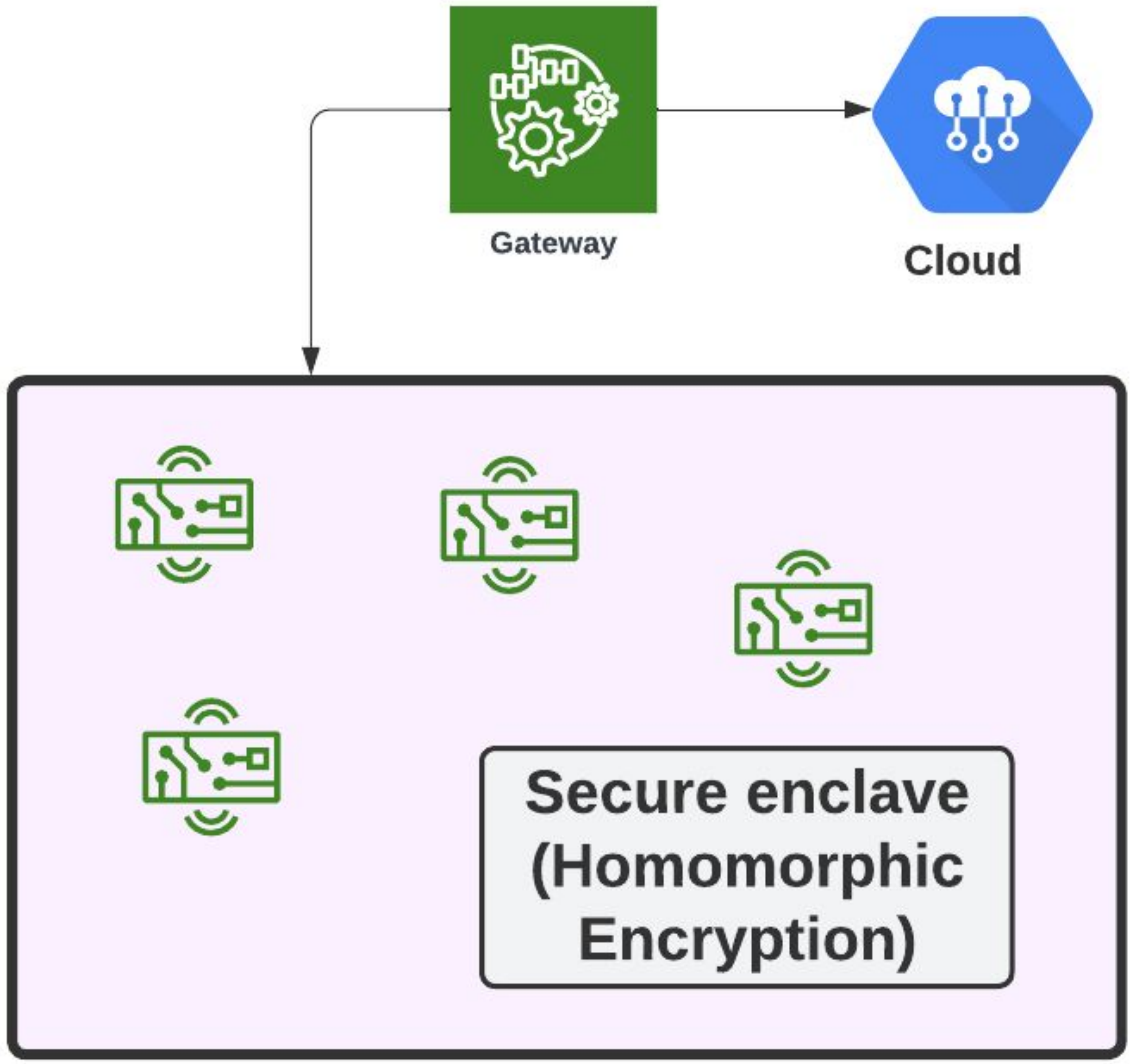
Extracted model

Secure enclave

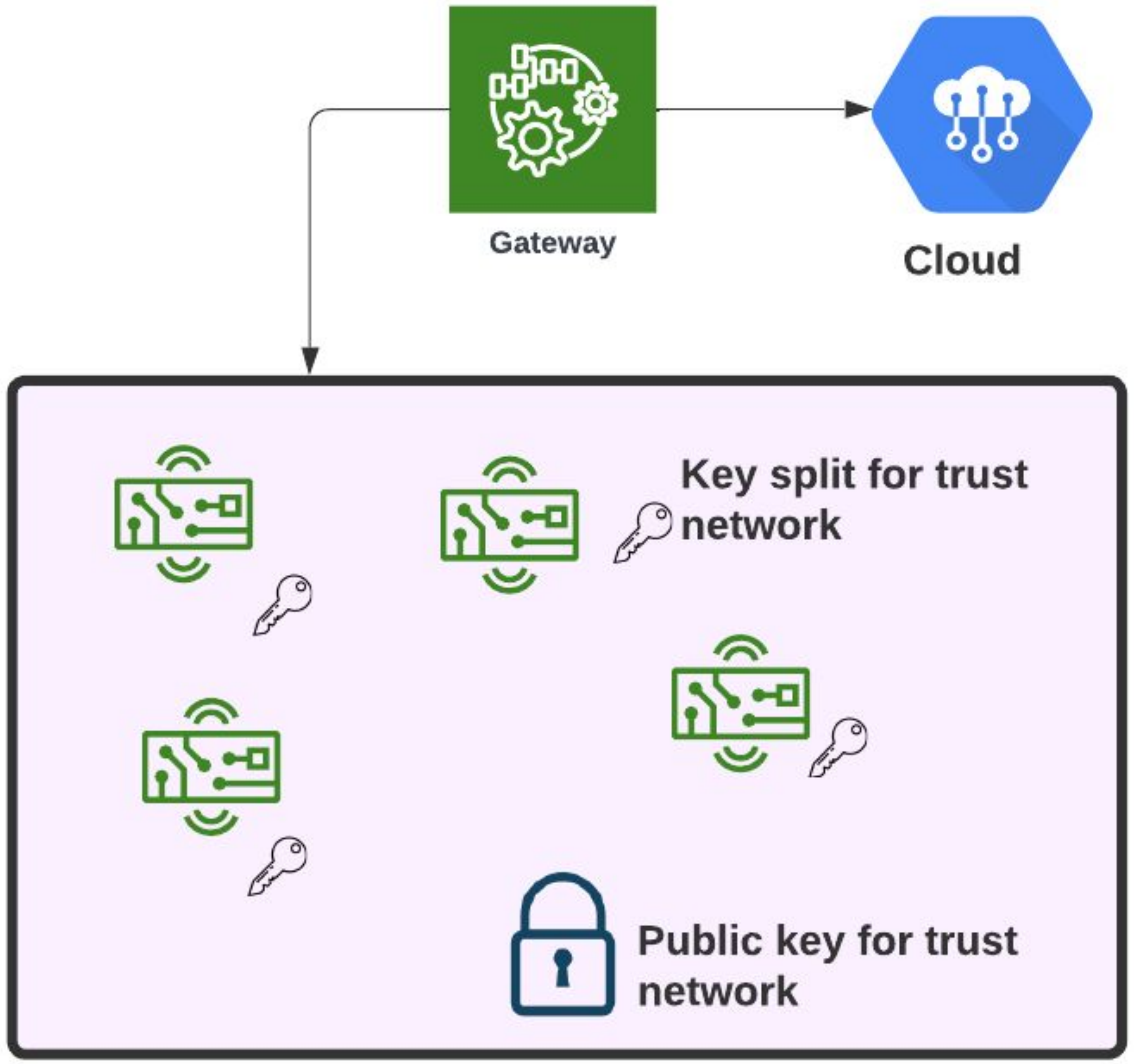
Homomorphic Engine (ML/Deep Learning)

Privacy Aware ML

Edge Computing/Trusted Environment



Edge computing



Distributed signing with consensus

FHE Methods

Prof Bill Buchanan OBE, FRSE

http://asecuritysite.com

Twitter: billatnapier

BFV/BGV

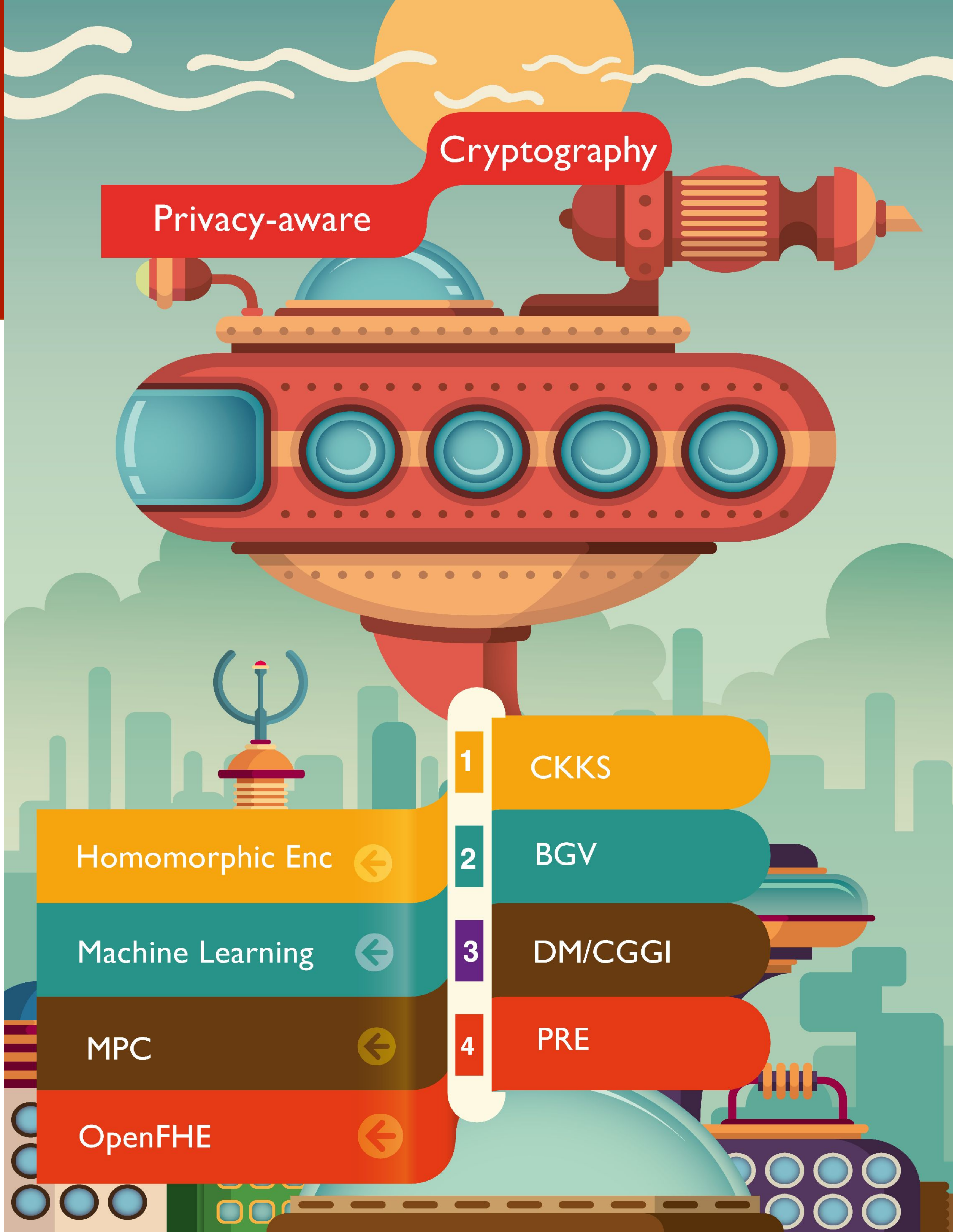
CKKS

DM/FHEW

OpenFHE

BLOCKPASS IDENTITY LAB

World-leading Collaboration between Blockpass IDN and Edinburgh Napier University



BFV (Brakerski/Fan-Vercauteren) Ring Learning With Errors (RLWE)

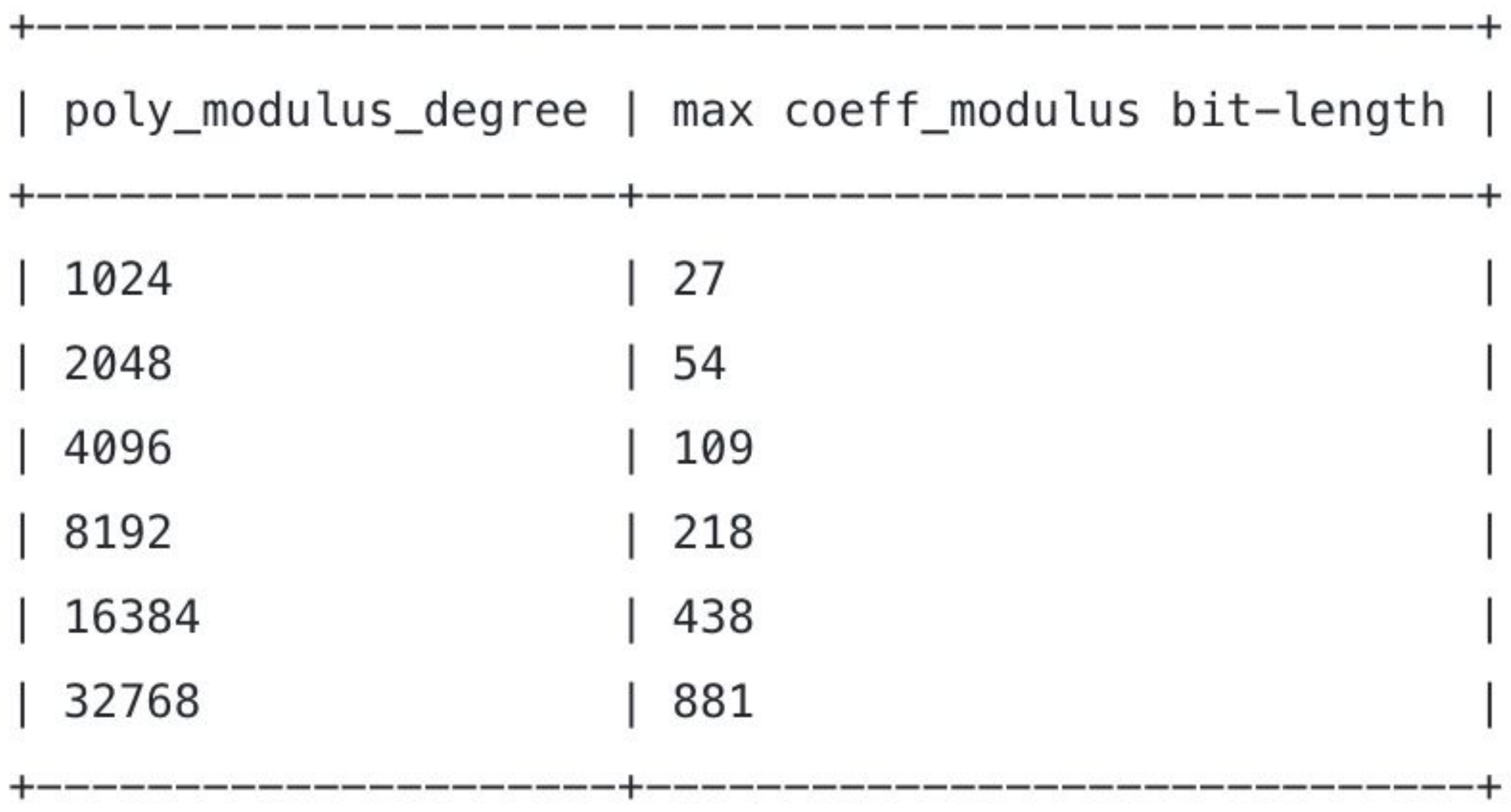
$$B = A.s + e \pmod{Q}$$

$$a_i = A_i \pmod{Q}$$

$$b_i = B_i + \frac{q}{2}.M_i \pmod{Q}$$

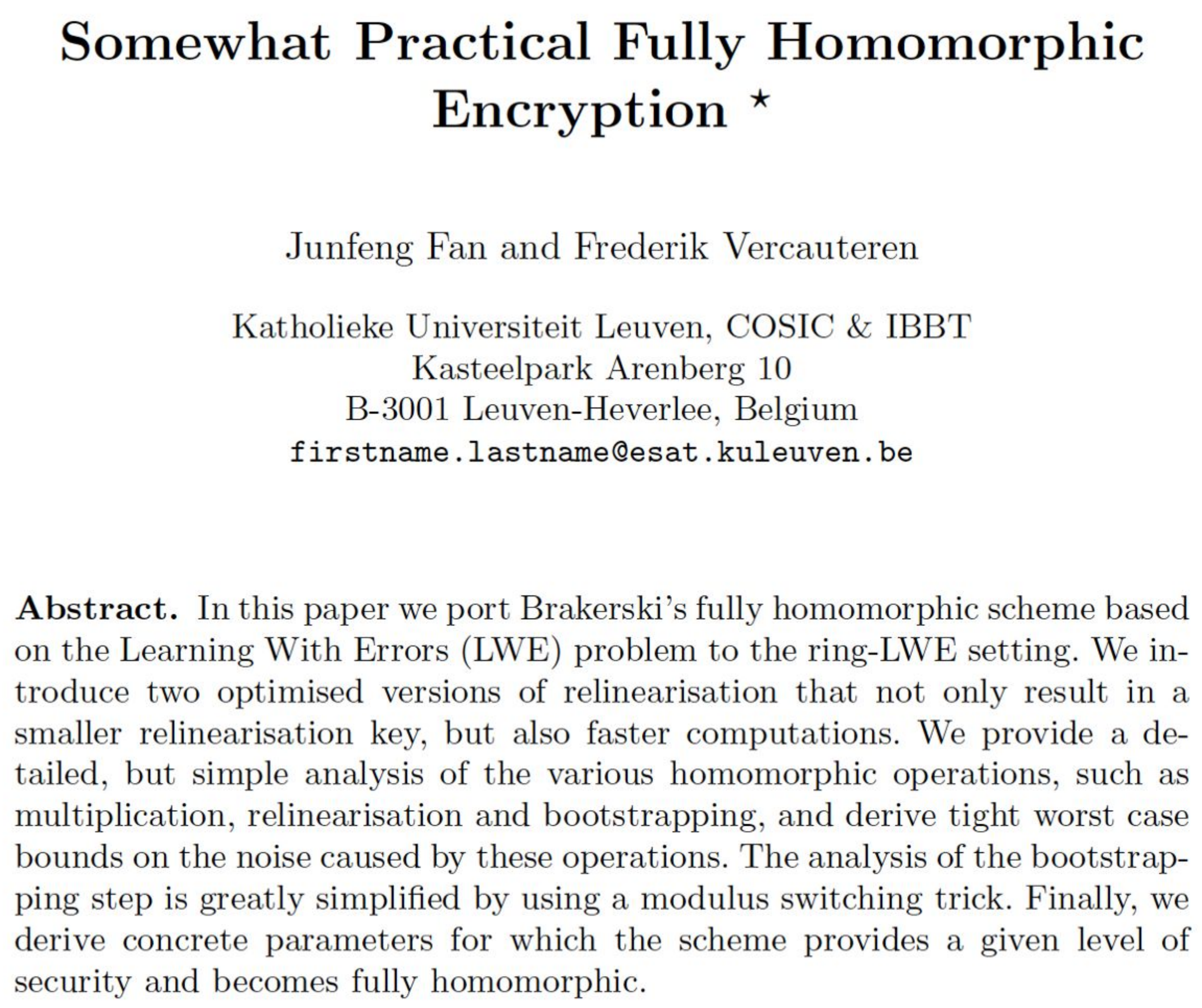
$$m = b - s.a \pmod{Q}$$

$$m = B + \frac{q}{2}.M - s.A = (A.s + e + \frac{q}{2}.M) - s.A = e + \frac{q}{2}.M \pmod{Q}$$



```
+----------------------------------------------------+
| poly_modulus_degree | max coeff_modulus bit-length |
+---------------------+------------------------------+
| 1024                | 27                           |
| 2048                | 54                           |
| 4096                | 109                          |
| 8192                | 218                          |
| 16384               | 438                          |
| 32768               | 881                          |
+---------------------+------------------------------+
```

The Microsoft SEAL (Simple Encrypted Arithmetic Library) library can support a range of homomorphic encryption methods, and which use Learning With Errors (LWE). In this case we will implement with the BFV (Brakerski/Fan-Vercauteren) method. Overall, BFV is a ring LWE method, and where we have a public modulus of Q. We then have a secret key of s (and which is between 0 and Q -1), and a random polynomial value of A. We also have an error polynomial of e.



Somewhat Practical Fully Homomorphic Encryption *

Junfeng Fan and Frederik Vercauteren

Katholieke Universiteit Leuven, COSIC & IBBT
Kasteelpark Arenberg 10
B-3001 Leuven-Heverlee, Belgium
firstname.lastname@esat.kuleuven.be

Abstract. In this paper we port Brakerski's fully homomorphic scheme based on the Learning With Errors (LWE) problem to the ring-LWE setting. We introduce two optimised versions of relinearisation that not only result in a smaller relinearisation key, but also faster computations. We provide a detailed, but simple analysis of the various homomorphic operations, such as multiplication, relinearisation and bootstrapping, and derive tight worst case bounds on the noise caused by these operations. The analysis of the bootstrapping step is greatly simplified by using a modulus switching trick. Finally, we derive concrete parameters for which the scheme provides a given level of security and becomes fully homomorphic.

Microsoft SEAL with Node.js

Cryptography

Prof Bill Buchanan OBE, FRSE

http://asecuritysite.com

Twitter: billatnapier

CKKS

OpenFHE

BLOCKPASS
IDENTITY
LAB

World-leading Collaboration between
Blockpass IDN and Edinburgh Napier University

CKKS (Cheon, Kim, Kim and Song)

HEAAN (Homomorphic Encryption for Arithmetic of Approximate Numbers) uses a rescaling procedure for the size of the plaintext. It then produces an approximate rounding due to the truncation of the ciphertext into a smaller modulus. The method is especially useful in that it can be applied to carry-out encryption computations in parallel. Unfortunately, the ciphertext modulus can become too small, and where it is not possible to carry out any more operations. The HEAAN (CKK) method uses approximate arithmetic over complex numbers (ℂ), and is based on Ring Learning With Errors (RLWE). It focuses on defining an encryption error within the computational error that will happen within approximate computations. We initially take a message (M) and convert to a cipher message (ct) using a secret key (sk). To decrypt ([⟨*ct*,*sk*⟩] q), we produce an approximate value along with a small error (e).

The main parameters:

- logN. Number of slots of plaintext values. This must be less than logP.
- logQ. The ciphertext modulus.
- logP. The scaling factor. The larger this is, the more accurace the answer will be.

Initially Alice and Bob agree on a complexity value of n, and which is the highest co-efficient power to be used

$$\mathbf{A} = a_{n-1}x^{n-1} + \ldots + a_1x + a_1x^2 + a_0$$

$$\mathbf{A} = (a_{n-1}x^{n-1} + \ldots + a_1x + a_1x^2 + a_0) \div (x^n + 1)$$

Cryptography

Prof Bill Buchanan OBE, FRSE

http://asecuritysite.com

Twitter: billatnapier

DM/FHEW

OpenFHE

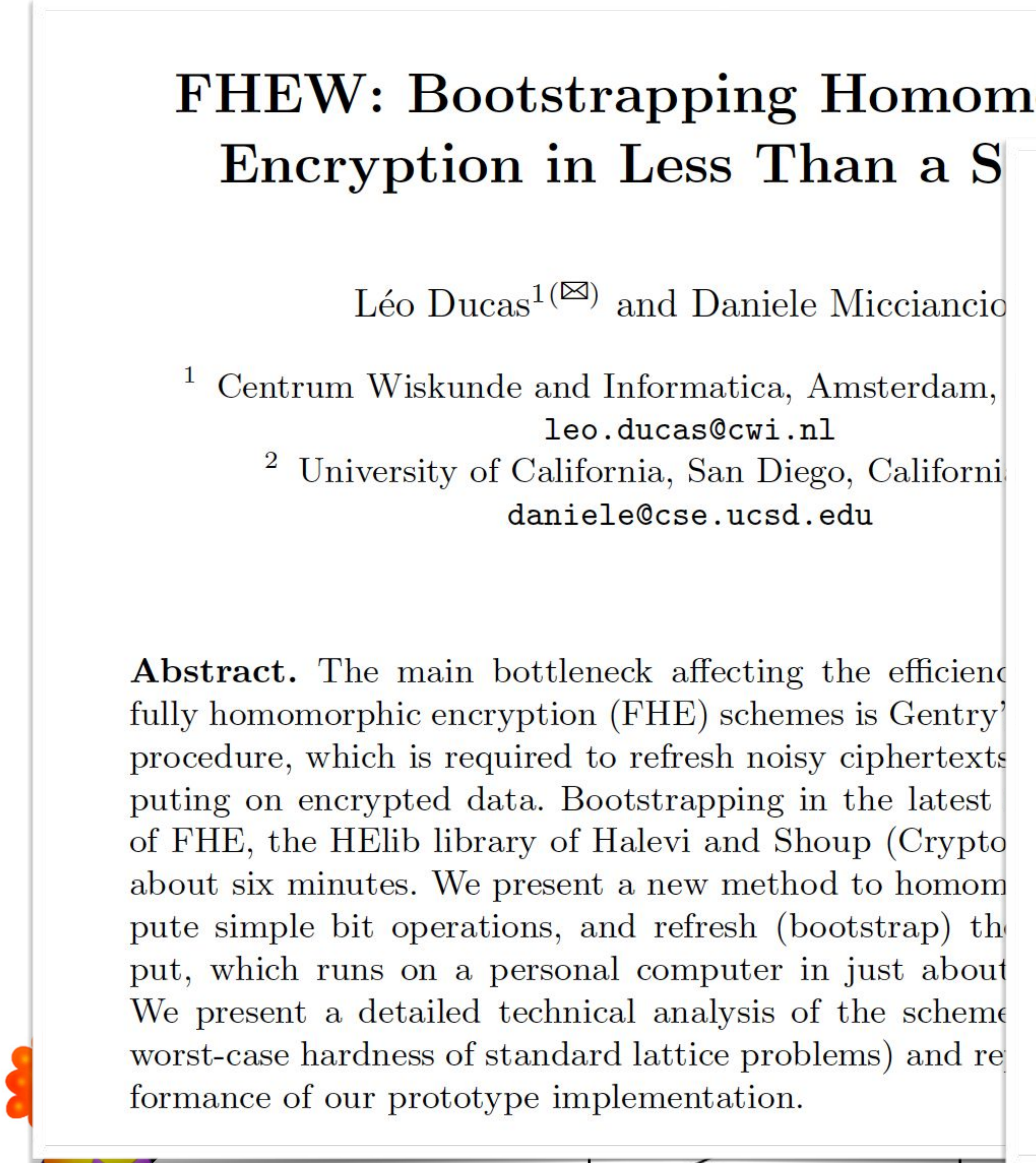
BLOCKPASS
IDENTITY
LAB

World-leading Collaboration between
Blockpass IDN and Edinburgh Napier University

DM/FHEW

that provides fully homomorphic encryption (FHE). It

n.

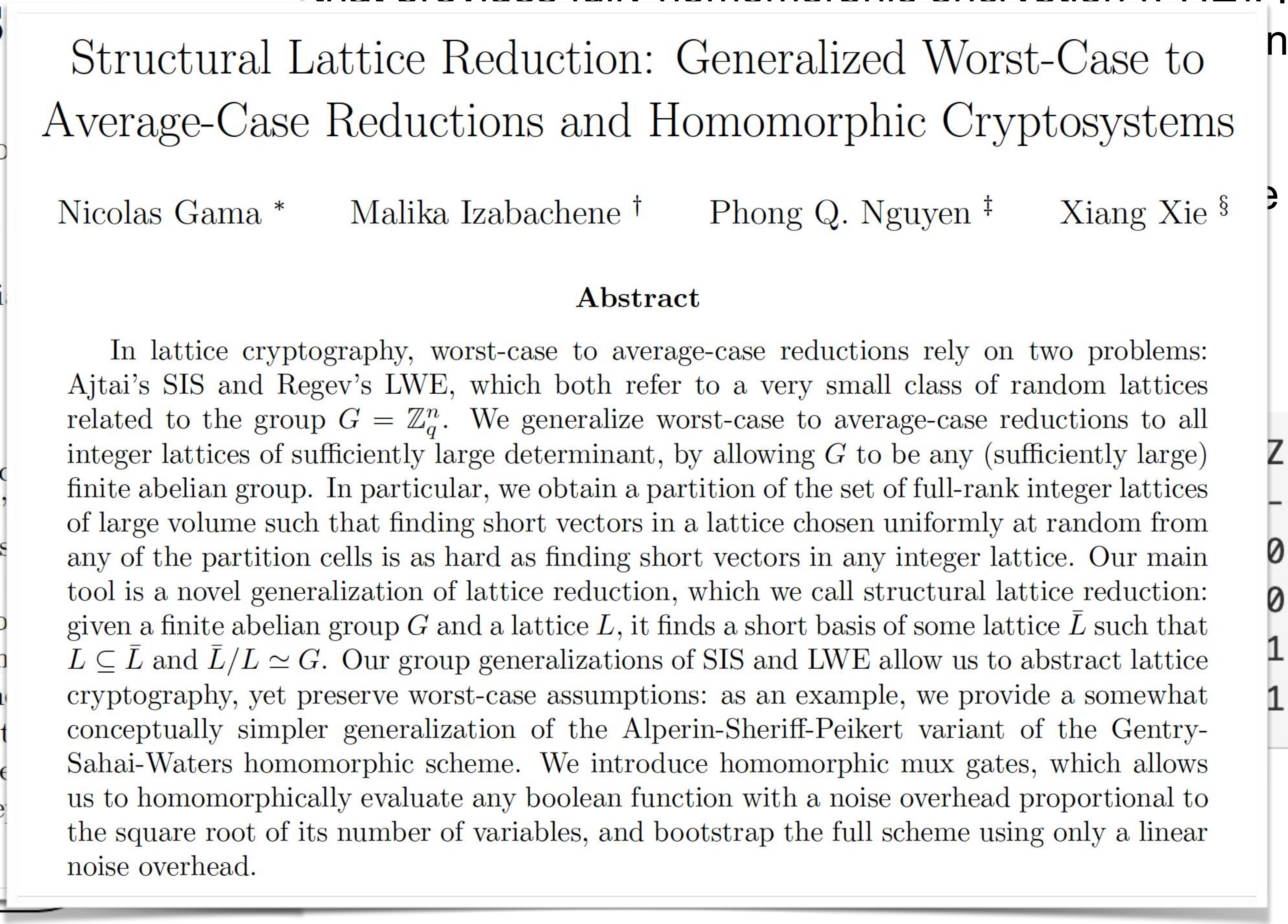


FHEW: Bootstrapping Homomorphic Encryption in Less Than a S

Léo Ducas[1](✉) and Daniele Micciancio

[1] Centrum Wiskunde and Informatica, Amsterdam,
leo.ducas@cwi.nl
[2] University of California, San Diego, Californi
daniele@cse.ucsd.edu

Abstract. The main bottleneck affecting the efficienc fully homomorphic encryption (FHE) schemes is Gentry' procedure, which is required to refresh noisy ciphertexts puting on encrypted data. Bootstrapping in the latest of FHE, the HElib library of Halevi and Shoup (Crypto about six minutes. We present a new method to homom pute simple bit operations, and refresh (bootstrap) th put, which runs on a personal computer in just about We present a detailed technical analysis of the scheme worst-case hardness of standard lattice problems) and re formance of our prototype implementation.



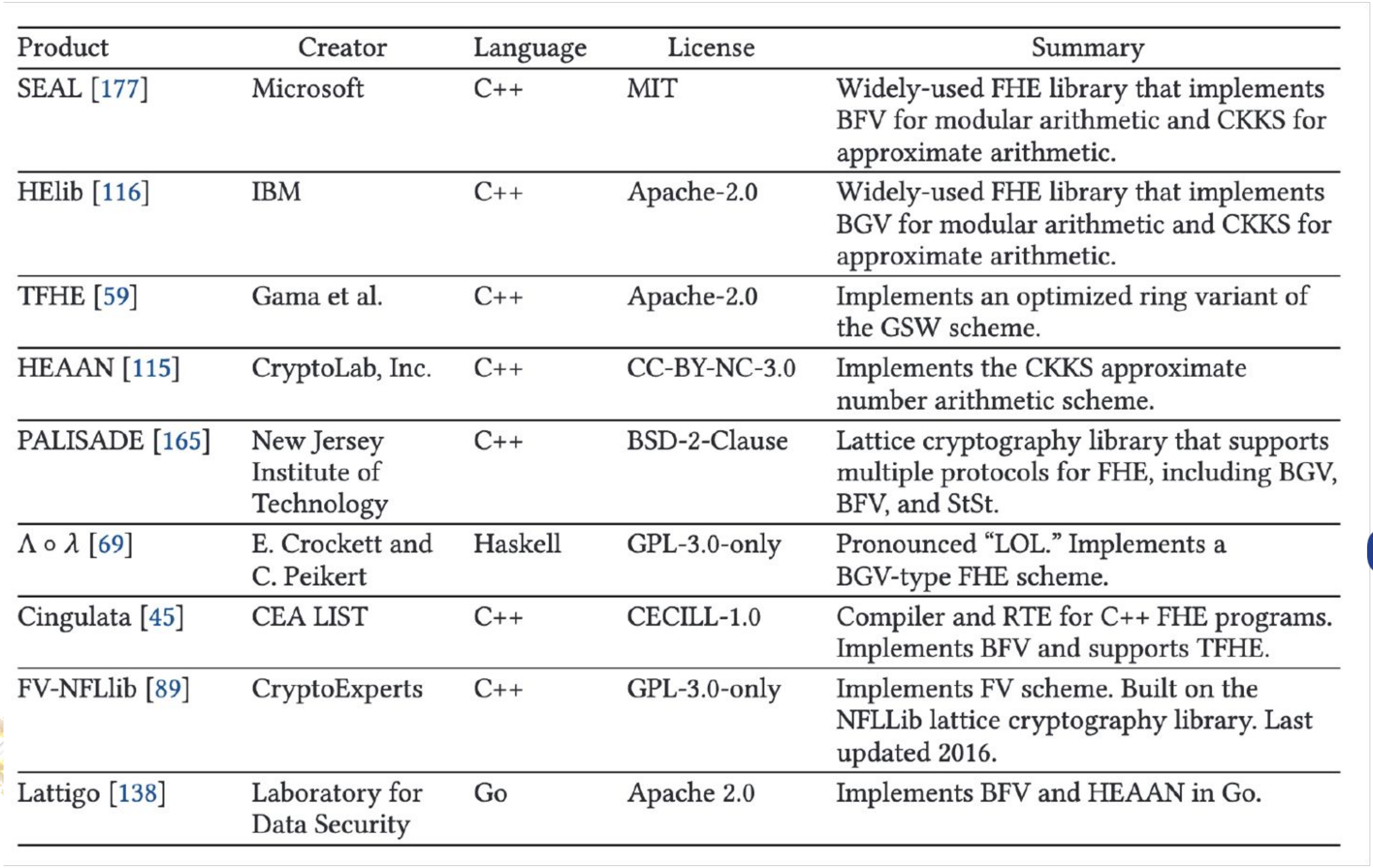
Structural Lattice Reduction: Generalized Worst-Case to Average-Case Reductions and Homomorphic Cryptosystems

Nicolas Gama * Malika Izabachene † Phong Q. Nguyen ‡ Xiang Xie §

Abstract

In lattice cryptography, worst-case to average-case reductions rely on two problems: Ajtai's SIS and Regev's LWE, which both refer to a very small class of random lattices related to the group $G = \mathbb{Z}_q^n$. We generalize worst-case to average-case reductions to all integer lattices of sufficiently large determinant, by allowing G to be any (sufficiently large) finite abelian group. In particular, we obtain a partition of the set of full-rank integer lattices of large volume such that finding short vectors in a lattice chosen uniformly at random from any of the partition cells is as hard as finding short vectors in any integer lattice. Our main tool is a novel generalization of lattice reduction, which we call structural lattice reduction: given a finite abelian group G and a lattice L, it finds a short basis of some lattice $\bar{L}$ such that $L \subseteq \bar{L}$ and $\bar{L}/L \simeq G$. Our group generalizations of SIS and LWE allow us to abstract lattice cryptography, yet preserve worst-case assumptions: as an example, we provide a somewhat conceptually simpler generalization of the Alperin-Sheriff-Peikert variant of the Gentry-Sahai-Waters homomorphic scheme. We introduce homomorphic mux gates, which allows us to homomorphically evaluate any boolean function with a noise overhead proportional to the square root of its number of variables, and bootstrap the full scheme using only a linear noise overhead.

Libraries



Product	Creator	Language	License	Summary
SEAL [177]	Microsoft	C++	MIT	Widely-used FHE library that implements BFV for modular arithmetic and CKKS for approximate arithmetic.
HElib [116]	IBM	C++	Apache-2.0	Widely-used FHE library that implements BGV for modular arithmetic and CKKS for approximate arithmetic.
TFHE [59]	Gama et al.	C++	Apache-2.0	Implements an optimized ring variant of the GSW scheme.
HEAAN [115]	CryptoLab, Inc.	C++	CC-BY-NC-3.0	Implements the CKKS approximate number arithmetic scheme.
PALISADE [165]	New Jersey Institute of Technology	C++	BSD-2-Clause	Lattice cryptography library that supports multiple protocols for FHE, including BGV, BFV, and StSt.
$\Lambda \circ \lambda$ [69]	E. Crockett and C. Peikert	Haskell	GPL-3.0-only	Pronounced “LOL.” Implements a BGV-type FHE scheme.
Cingulata [45]	CEA LIST	C++	CECILL-1.0	Compiler and RTE for C++ FHE programs. Implements BFV and supports TFHE.
FV-NFLlib [89]	CryptoExperts	C++	GPL-3.0-only	Implements FV scheme. Built on the NFLLib lattice cryptography library. Last updated 2016.
Lattigo [138]	Laboratory for Data Security	Go	Apache 2.0	Implements BFV and HEAAN in Go.

OpenFHE

Bootstrapping and Slots

Prof Bill Buchanan OBE, FRSE

http://asecuritysite.com

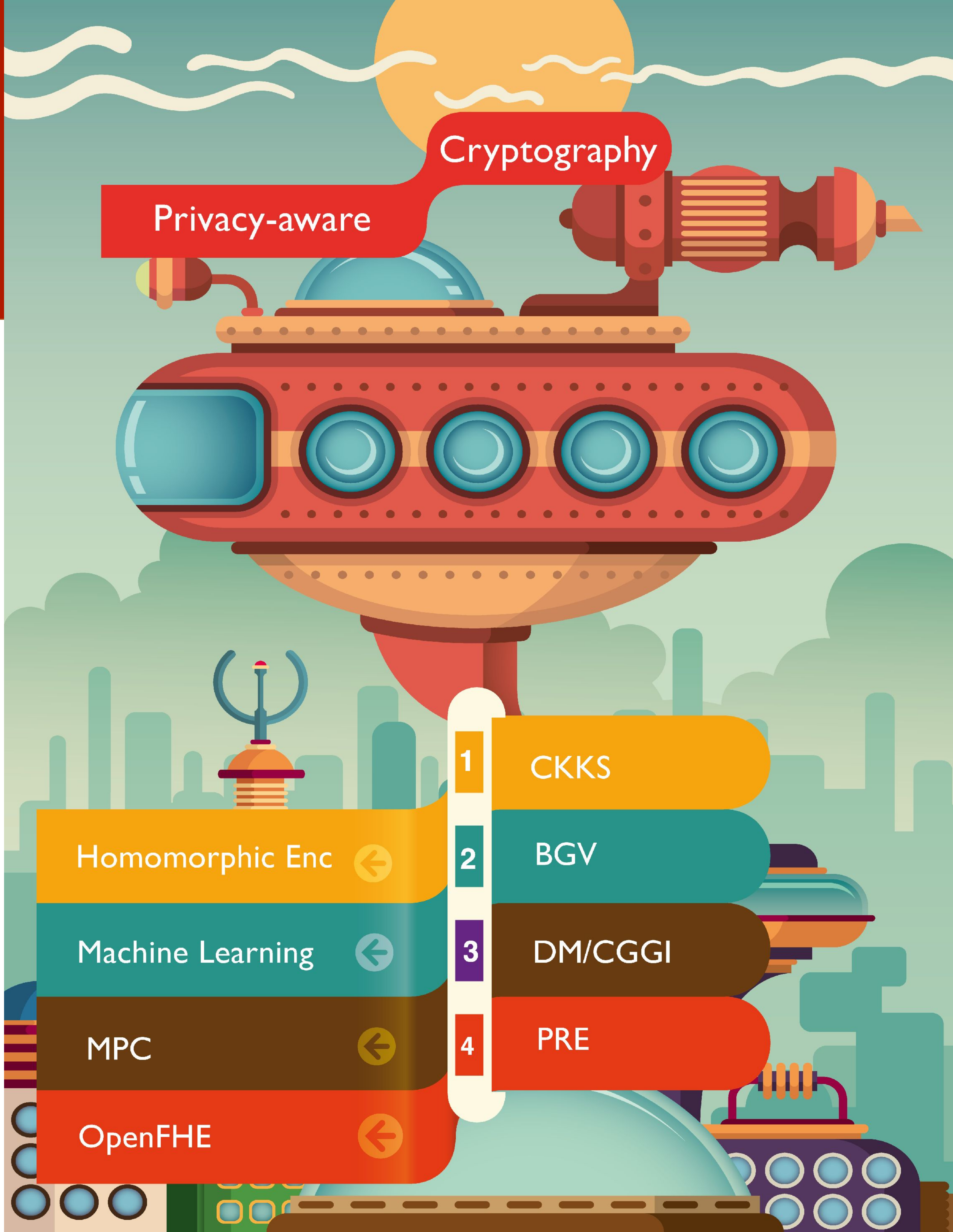
Twitter: billatnapier

Bootstrapping and Slots

OpenFHE

BLOCKPASS IDENTITY LAB

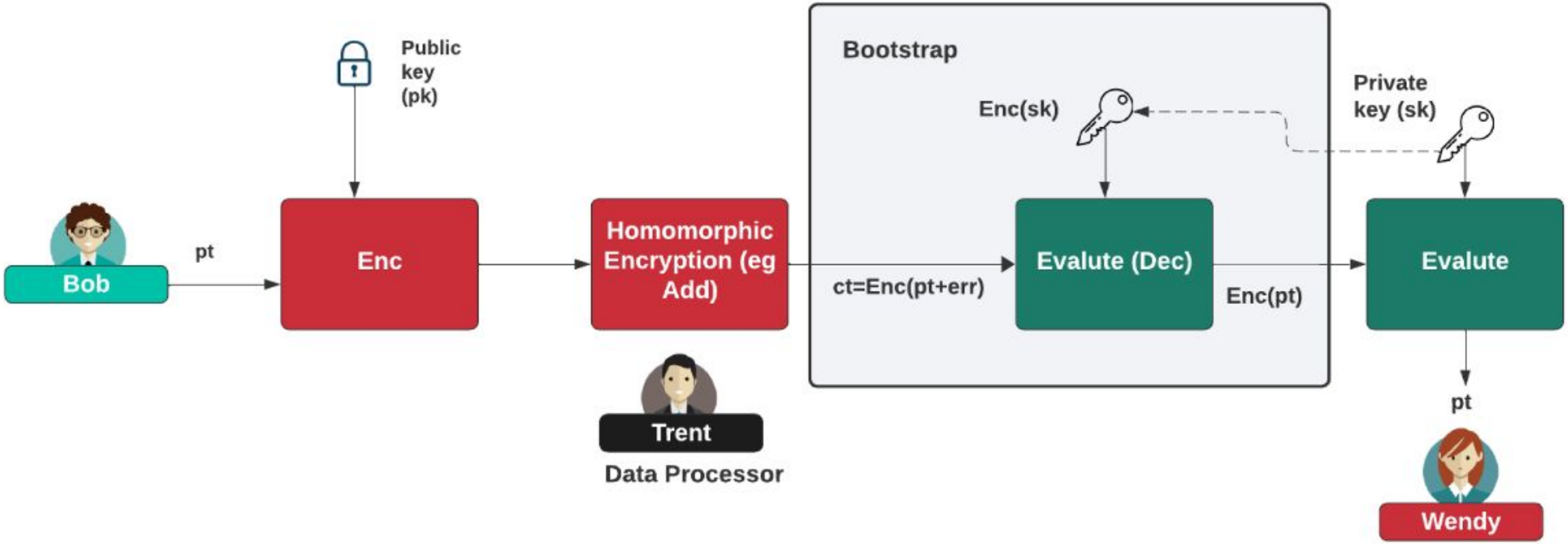
World-leading Collaboration between Blockpass IDN and Edinburgh Napier University

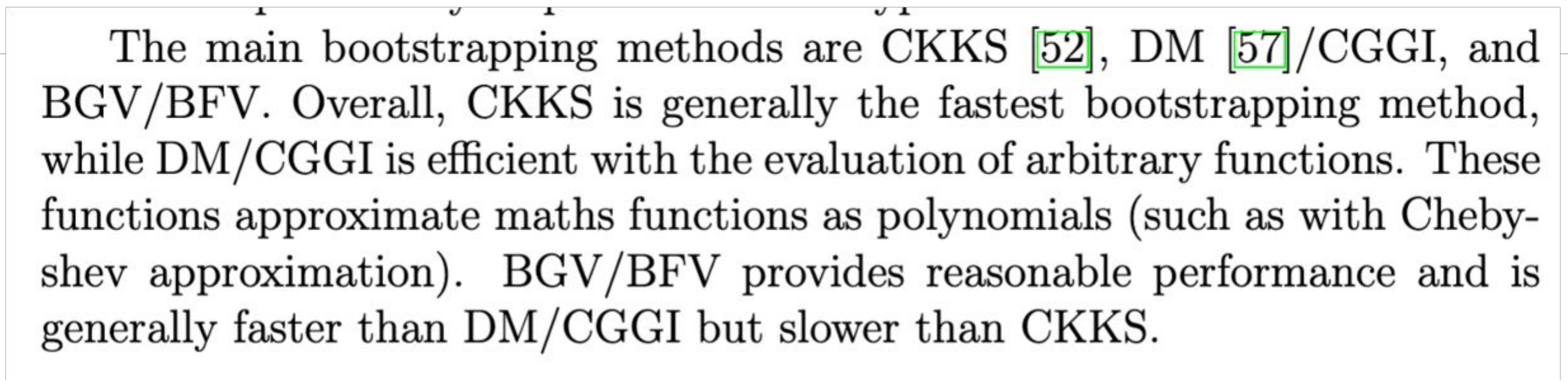


Bootstrapping

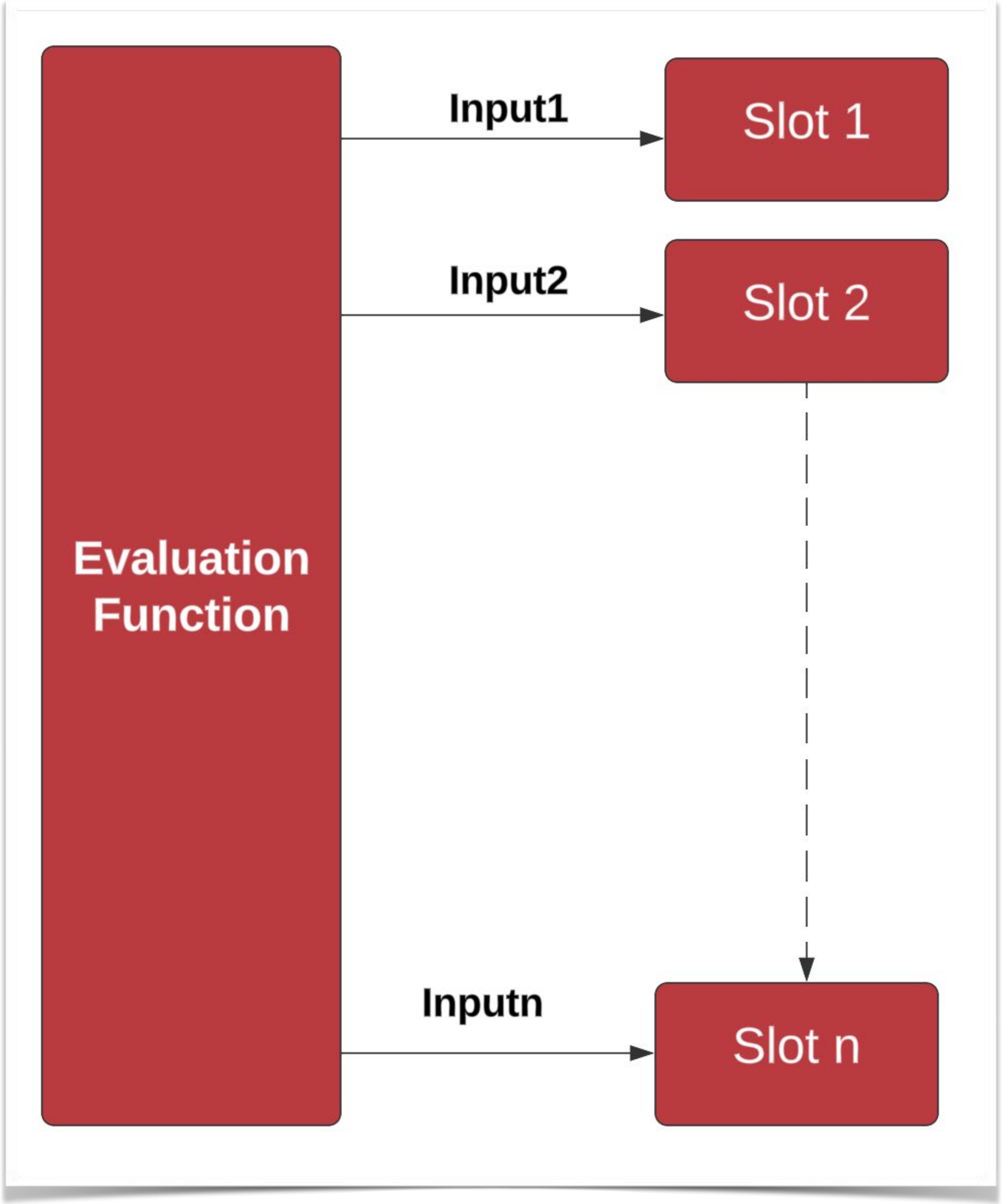
Each ciphertext can have an associated "level" and a value of "noise".

We have various levels. One multiplication consumes a level, and adds noise.




The main bootstrapping methods are CKKS [52], DM [57]/CGGI, and BGV/BFV. Overall, CKKS is generally the fastest bootstrapping method, while DM/CGGI is efficient with the evaluation of arbitrary functions. These functions approximate maths functions as polynomials (such as with Chebyshev approximation). BGV/BFV provides reasonable performance and is generally faster than DM/CGGI but slower than CKKS.

Slots


Evaluation Function
Input1
Slot 1
Input2
Slot 2
Inputn
Slot n

Cryptography

Prof Bill Buchanan OBE, FRSE

http://asecuritysite.com

Twitter: billatnapier

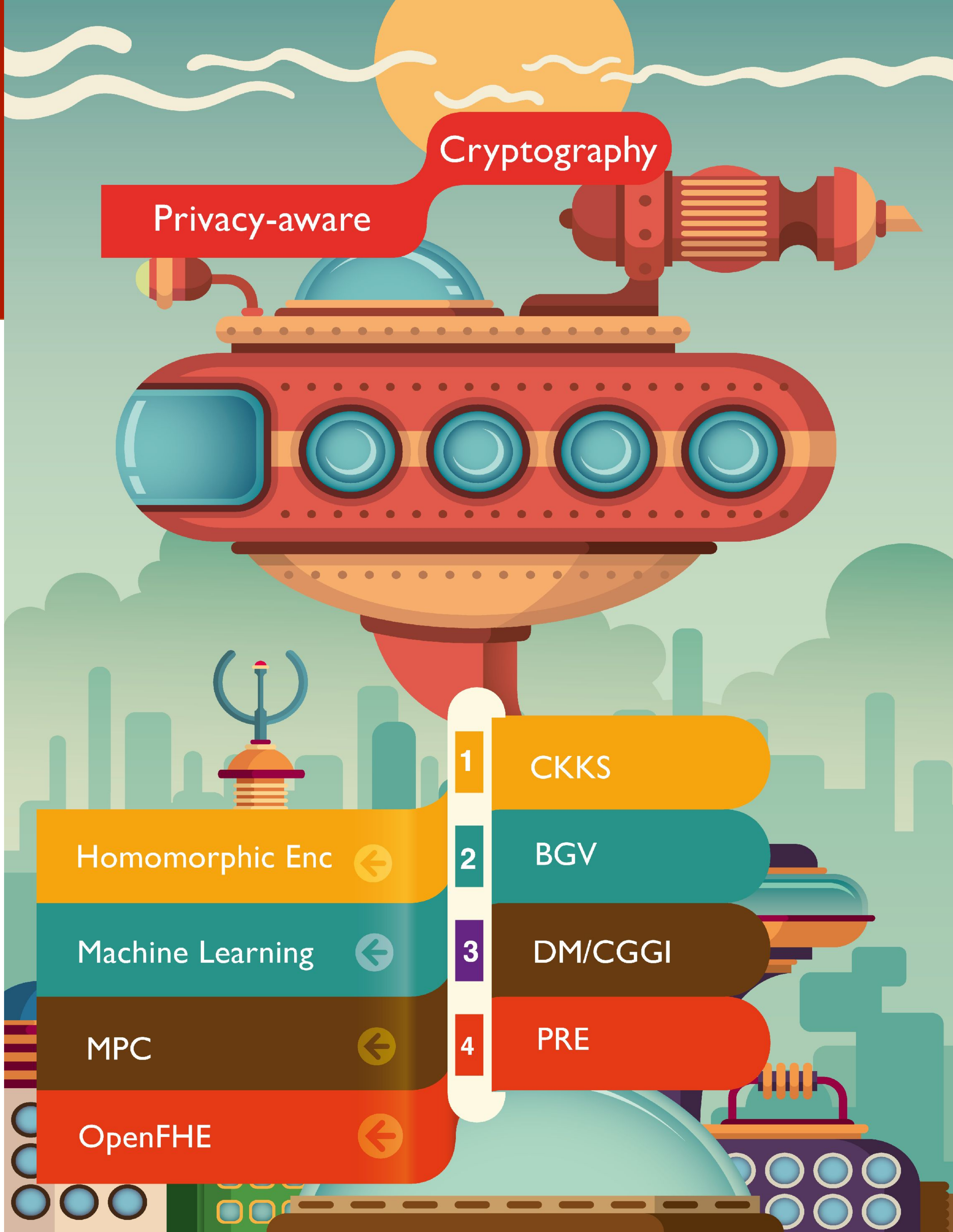
OpenFHE

Coding

OpenFHE

BLOCKPASS
IDENTITY
LAB

World-leading Collaboration between
Blockpass IDN and Edinburgh Napier University



Privacy-aware

Cryptography

Homomorphic Enc

Machine Learning

MPC

OpenFHE

1 CKKS

2 BGV

3 DM/CGGI

4 PRE

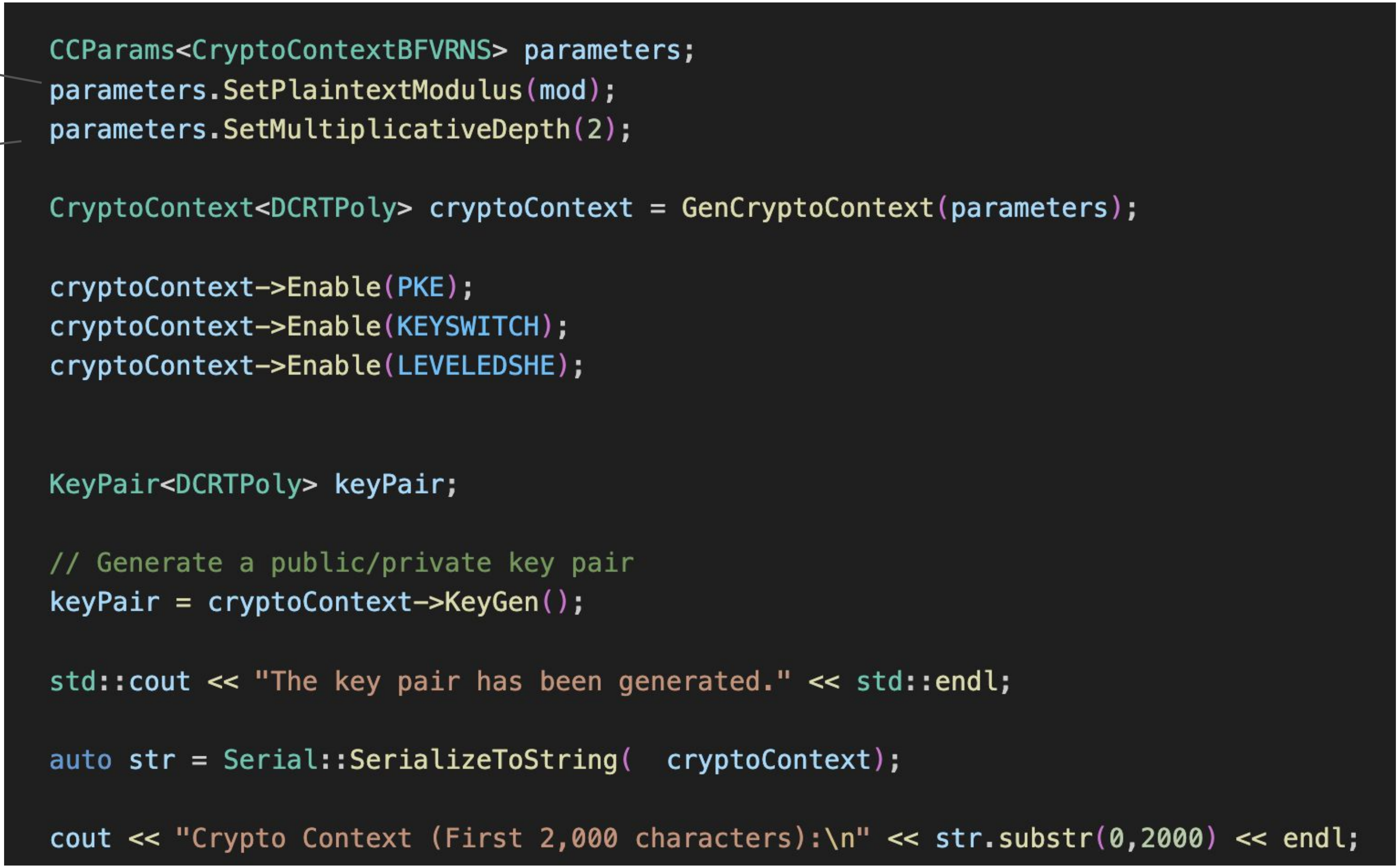
CryptoContext and Parameters (BFV)

Plaintext Modulus

Multiplicative Depth

PKE Scheme Features

Generate key pair (sk, pk)



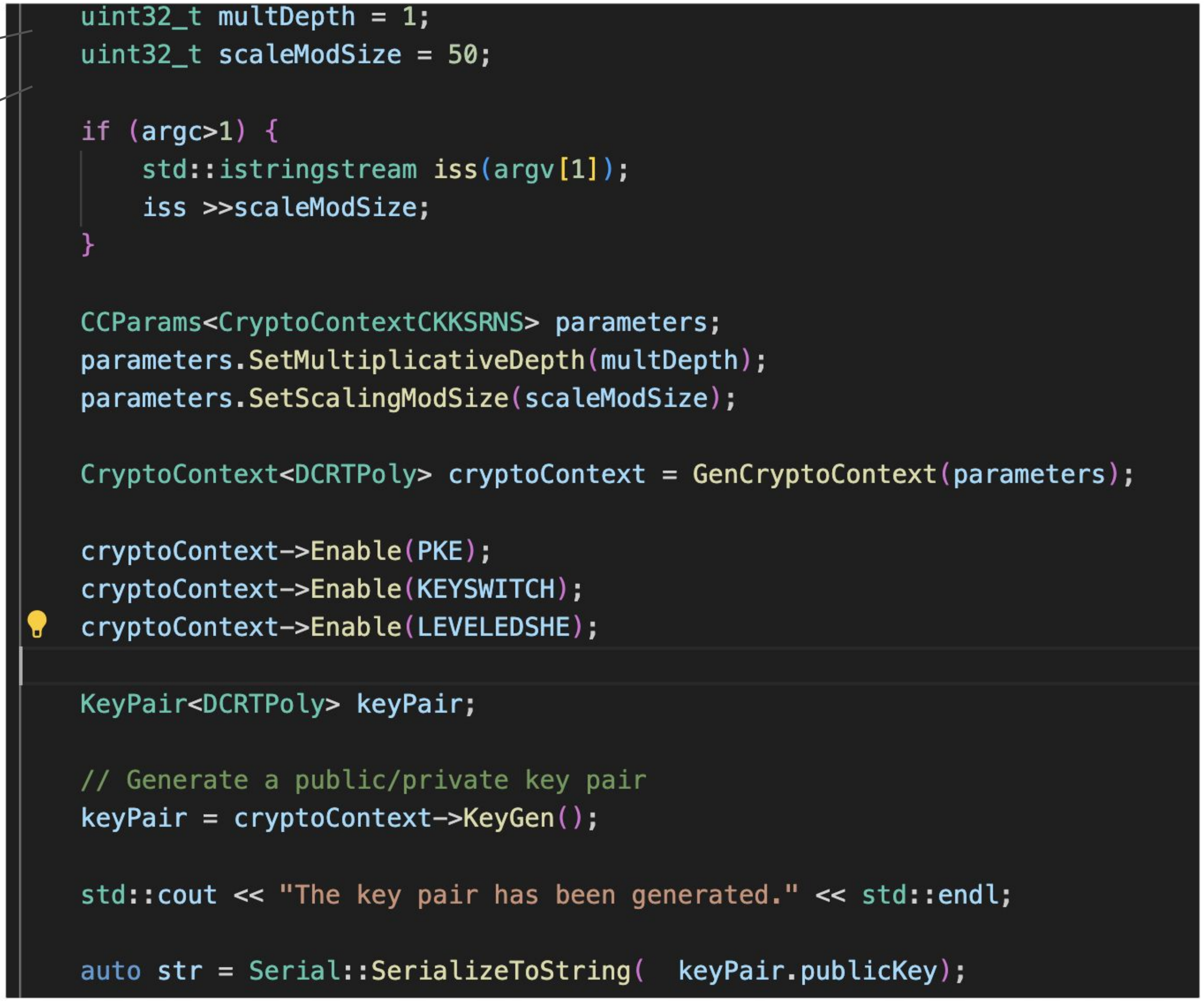
```
CCParams<CryptoContextBFVRNS> parameters;
parameters.SetPlaintextModulus(mod);
parameters.SetMultiplicativeDepth(2);

CryptoContext<DCRTPoly> cryptoContext = GenCryptoContext(parameters);

cryptoContext->Enable(PKE);
cryptoContext->Enable(KEYSWITCH);
cryptoContext->Enable(LEVELEDSHE);


KeyPair<DCRTPoly> keyPair;

// Generate a public/private key pair
keyPair = cryptoContext->KeyGen();

std::cout << "The key pair has been generated." << std::endl;

auto str = Serial::SerializeToString(  cryptoContext);

cout << "Crypto Context (First 2,000 characters):\n" << str.substr(0,2000) << endl;
```

https://asecuritysite.com/openfhe/openfhe_00cpp

CryptoContext and Parameters (CKKS)

Multiplication depth

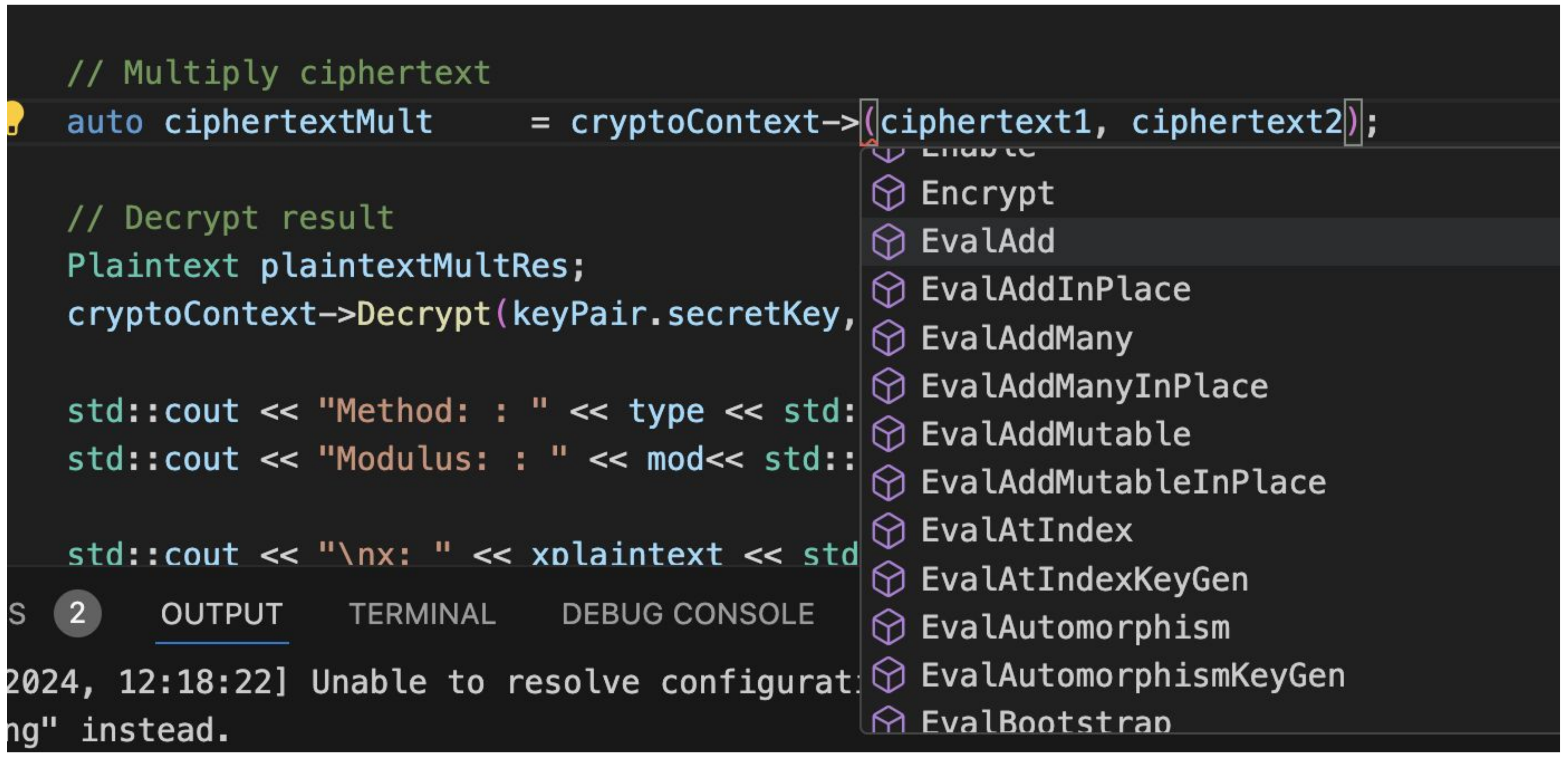
Scale Mod Size



```
uint32_t multDepth = 1;
uint32_t scaleModSize = 50;

if (argc>1) {
    std::istringstream iss(argv[1]);
    iss >>scaleModSize;
}

CCParams<CryptoContextCKKSRNS> parameters;
parameters.SetMultiplicativeDepth(multDepth);
parameters.SetScalingModSize(scaleModSize);

CryptoContext<DCRTPoly> cryptoContext = GenCryptoContext(parameters);

cryptoContext->Enable(PKE);
cryptoContext->Enable(KEYSWITCH);
cryptoContext->Enable(LEVELEDSHE);

KeyPair<DCRTPoly> keyPair;

// Generate a public/private key pair
keyPair = cryptoContext->KeyGen();

std::cout << "The key pair has been generated." << std::endl;

auto str = Serial::SerializeToString(  keyPair.publicKey);
```

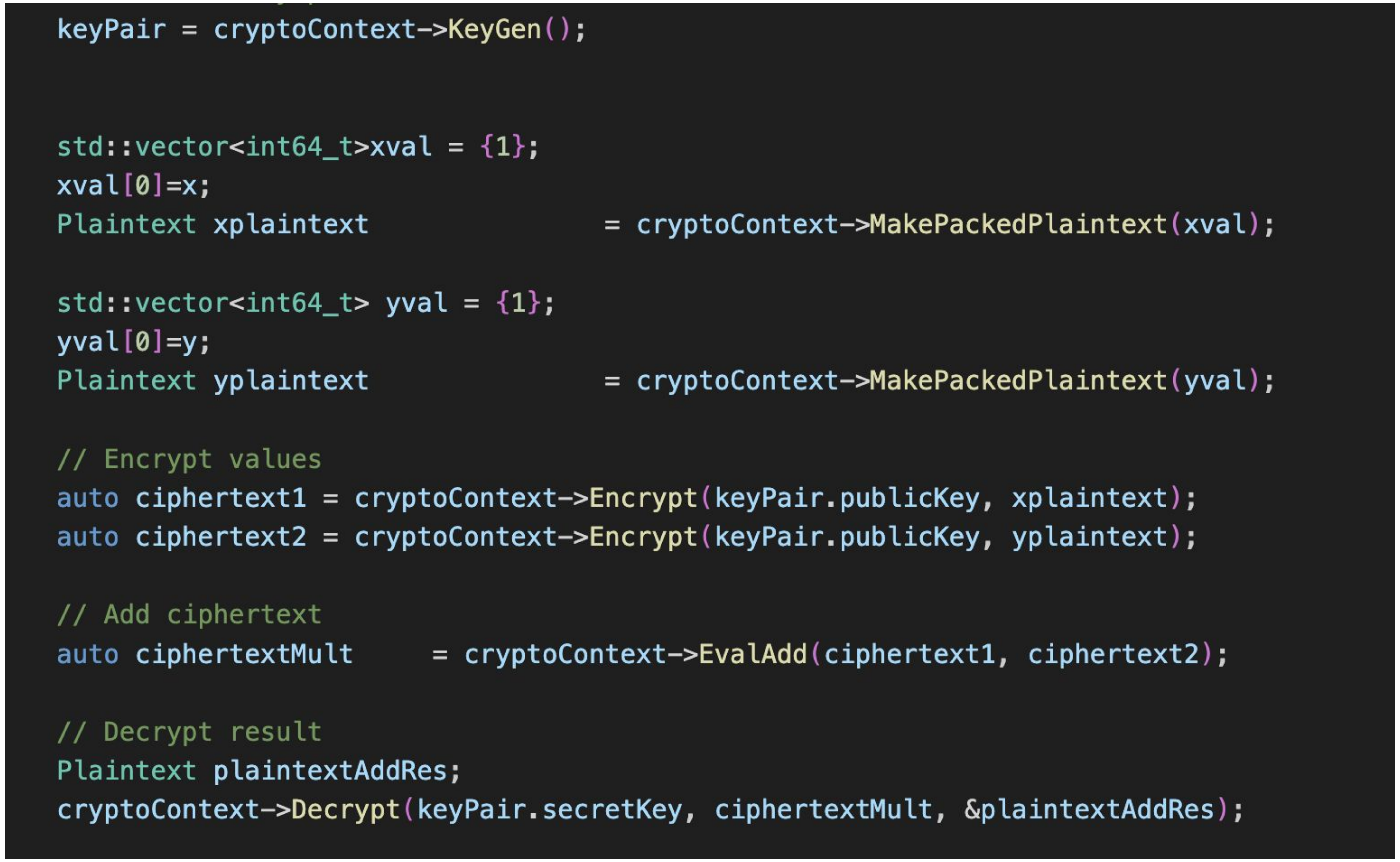
https://asecuritysite.com/openfhe/openfhe_00cpp_ckks

BFV - Adding/Multiplying Two Numbers



```
// Multiply ciphertext
auto ciphertextMult     = cryptoContext->(ciphertext1, ciphertext2);

// Decrypt result
Plaintext plaintextMultRes;
cryptoContext->Decrypt(keyPair.secretKey,

std::cout << "Method: : " << type << std:
std::cout << "Modulus: : " << mod<< std::

std::cout << "\nx: " << xplaintext << std
```

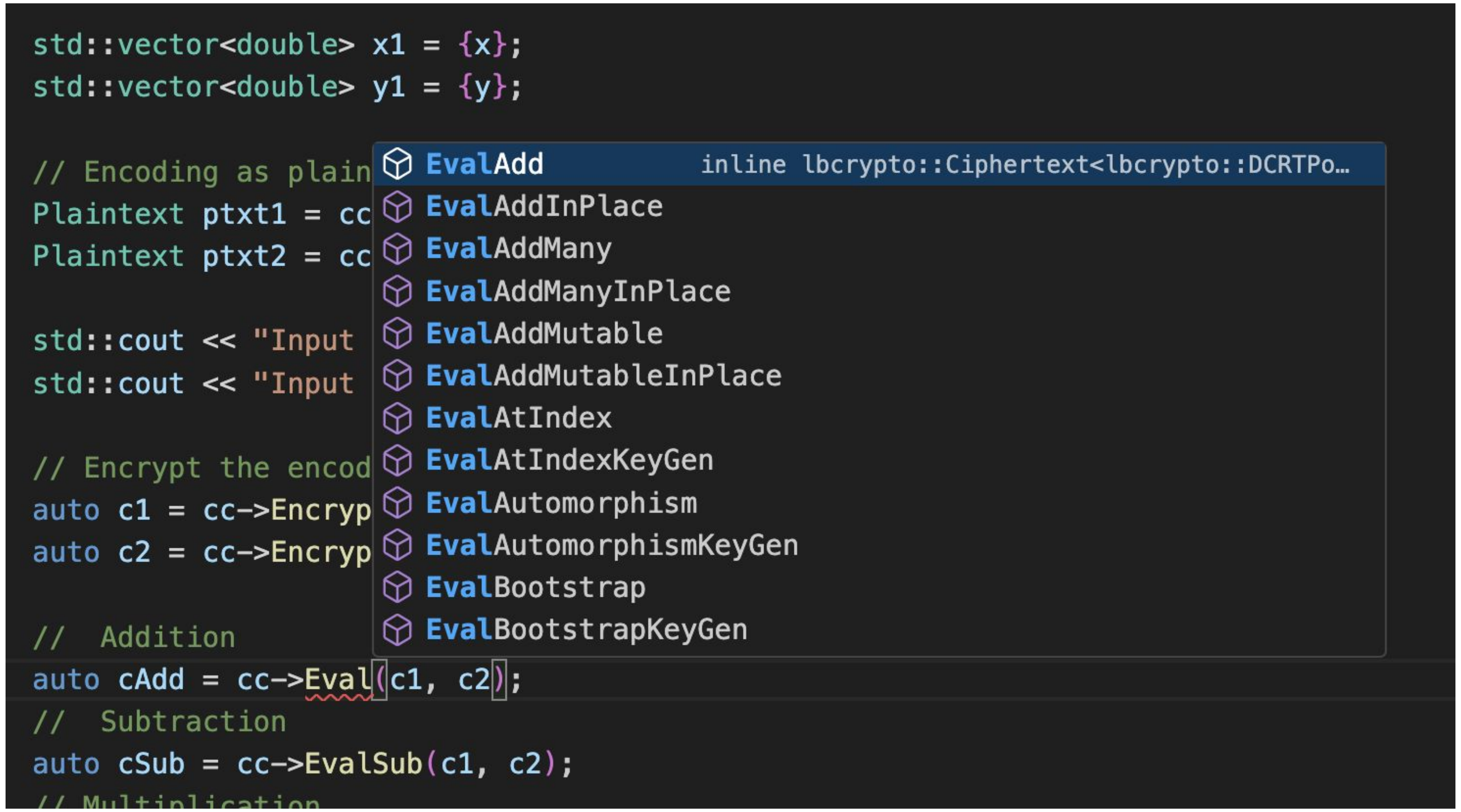
```
2024, 12:18:22] Unable to resolve configurat
ng" instead.
```



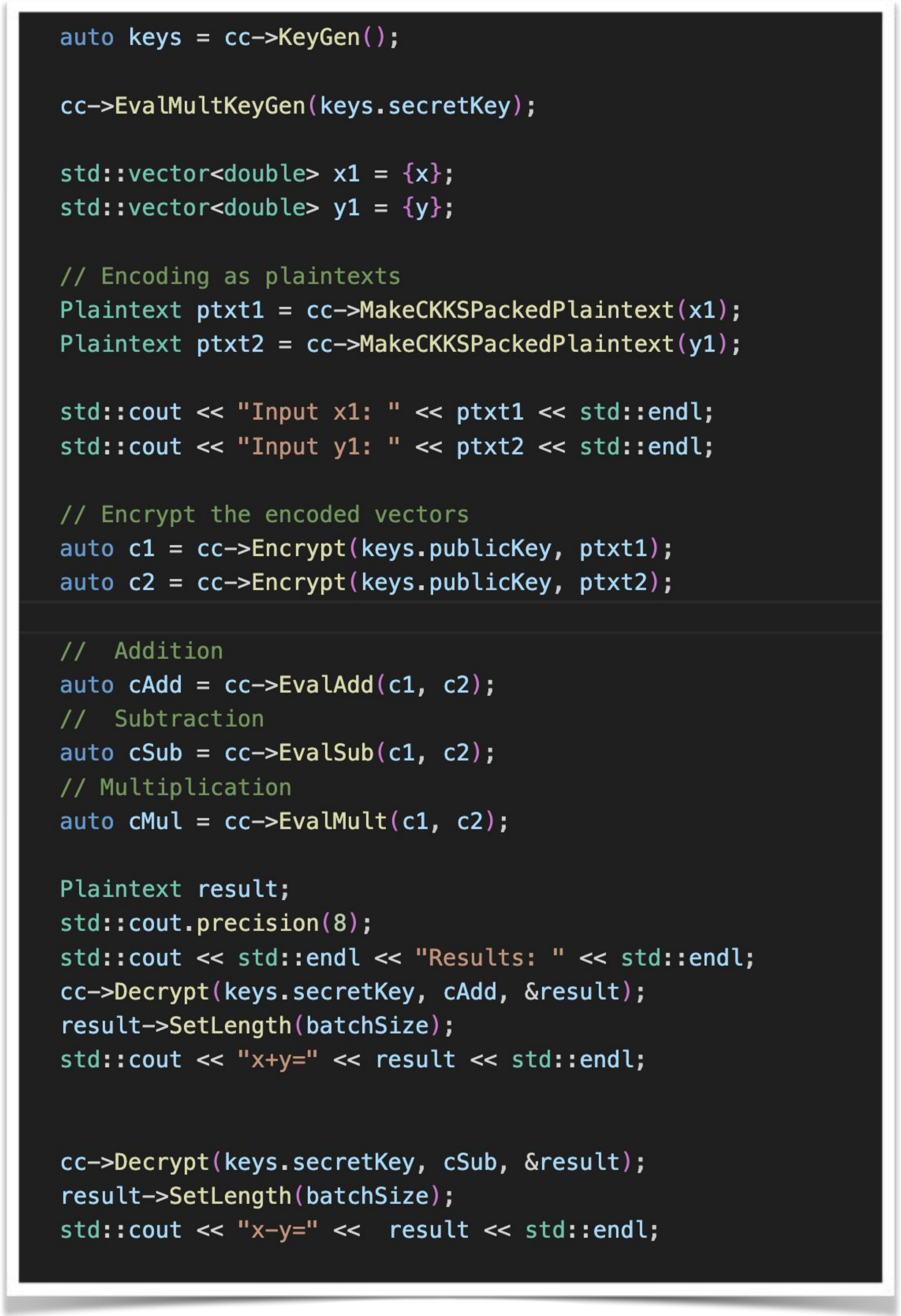
```
keyPair = cryptoContext->KeyGen();


std::vector<int64_t>xval = {1};
xval[0]=x;
Plaintext xplaintext              = cryptoContext->MakePackedPlaintext(xval);

std::vector<int64_t> yval = {1};
yval[0]=y;
Plaintext yplaintext              = cryptoContext->MakePackedPlaintext(yval);

// Encrypt values
auto ciphertext1 = cryptoContext->Encrypt(keyPair.publicKey, xplaintext);
auto ciphertext2 = cryptoContext->Encrypt(keyPair.publicKey, yplaintext);

// Add ciphertext
auto ciphertextMult     = cryptoContext->EvalAdd(ciphertext1, ciphertext2);

// Decrypt result
Plaintext plaintextAddRes;
cryptoContext->Decrypt(keyPair.secretKey, ciphertextMult, &plaintextAddRes);
```

https://asecuritysite.com/openfhe/openfhe_02cpp

CCKS - Adding/Multiplying Two Numbers



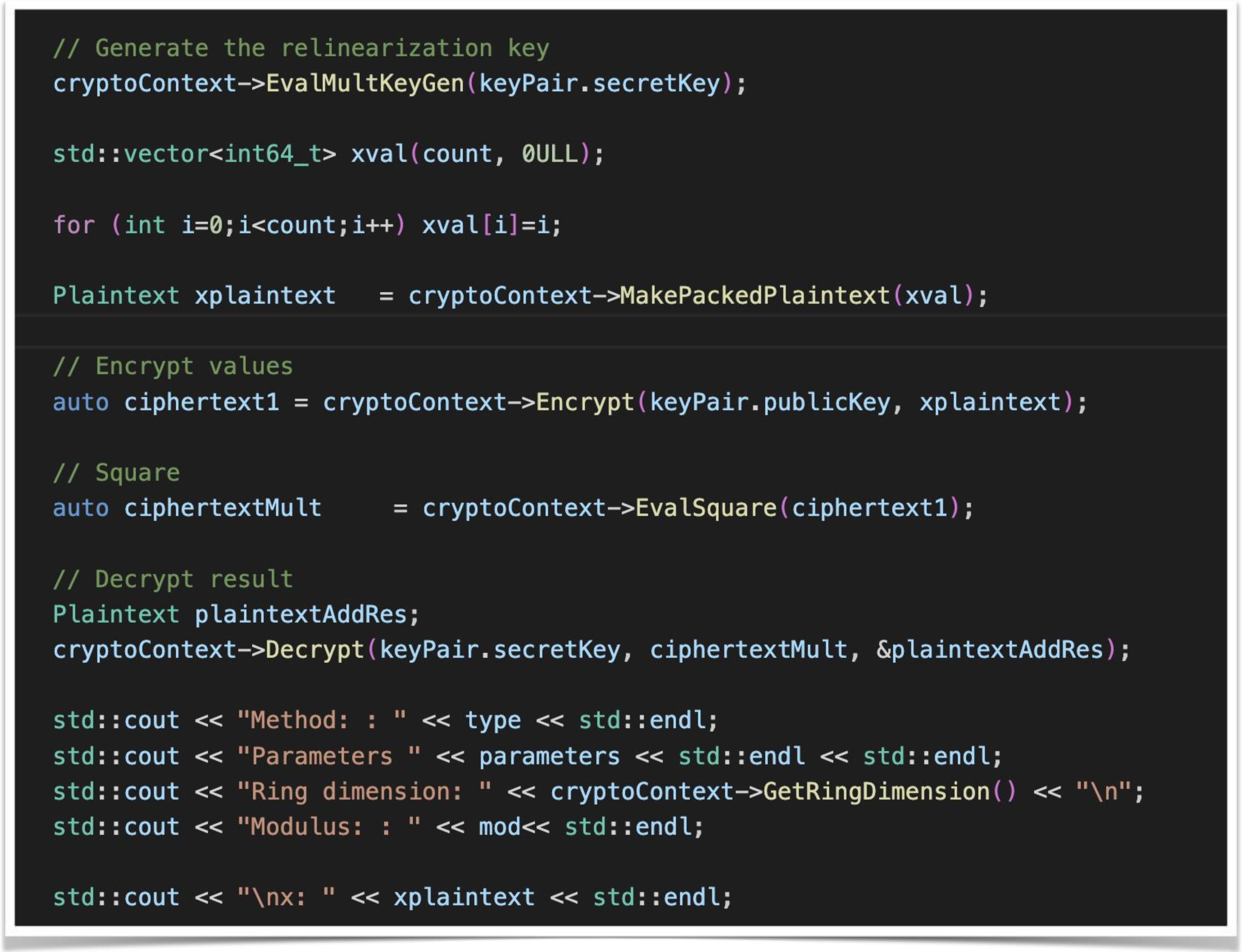
```
std::vector<double> x1 = {x};
std::vector<double> y1 = {y};

// Encoding as plain
Plaintext ptxt1 = cc
Plaintext ptxt2 = cc

std::cout << "Input
std::cout << "Input

// Encrypt the encod
auto c1 = cc->Encryp
auto c2 = cc->Encryp

//  Addition
auto cAdd = cc->Eval(c1, c2);
//  Subtraction
auto cSub = cc->EvalSub(c1, c2);
// Multiplication
```



```
auto keys = cc->KeyGen();

cc->EvalMultKeyGen(keys.secretKey);

std::vector<double> x1 = {x};
std::vector<double> y1 = {y};

// Encoding as plaintexts
Plaintext ptxt1 = cc->MakeCKKSPackedPlaintext(x1);
Plaintext ptxt2 = cc->MakeCKKSPackedPlaintext(y1);

std::cout << "Input x1: " << ptxt1 << std::endl;
std::cout << "Input y1: " << ptxt2 << std::endl;

// Encrypt the encoded vectors
auto c1 = cc->Encrypt(keys.publicKey, ptxt1);
auto c2 = cc->Encrypt(keys.publicKey, ptxt2);

//  Addition
auto cAdd = cc->EvalAdd(c1, c2);
//  Subtraction
auto cSub = cc->EvalSub(c1, c2);
// Multiplication
auto cMul = cc->EvalMult(c1, c2);

Plaintext result;
std::cout.precision(8);
std::cout << std::endl << "Results: " << std::endl;
cc->Decrypt(keys.secretKey, cAdd, &result);
result->SetLength(batchSize);
std::cout << "x+y=" << result << std::endl;


cc->Decrypt(keys.secretKey, cSub, &result);
result->SetLength(batchSize);
std::cout << "x-y=" <<  result << std::endl;
```

https://asecuritysite.com/openfhe/openfhe_05cpp

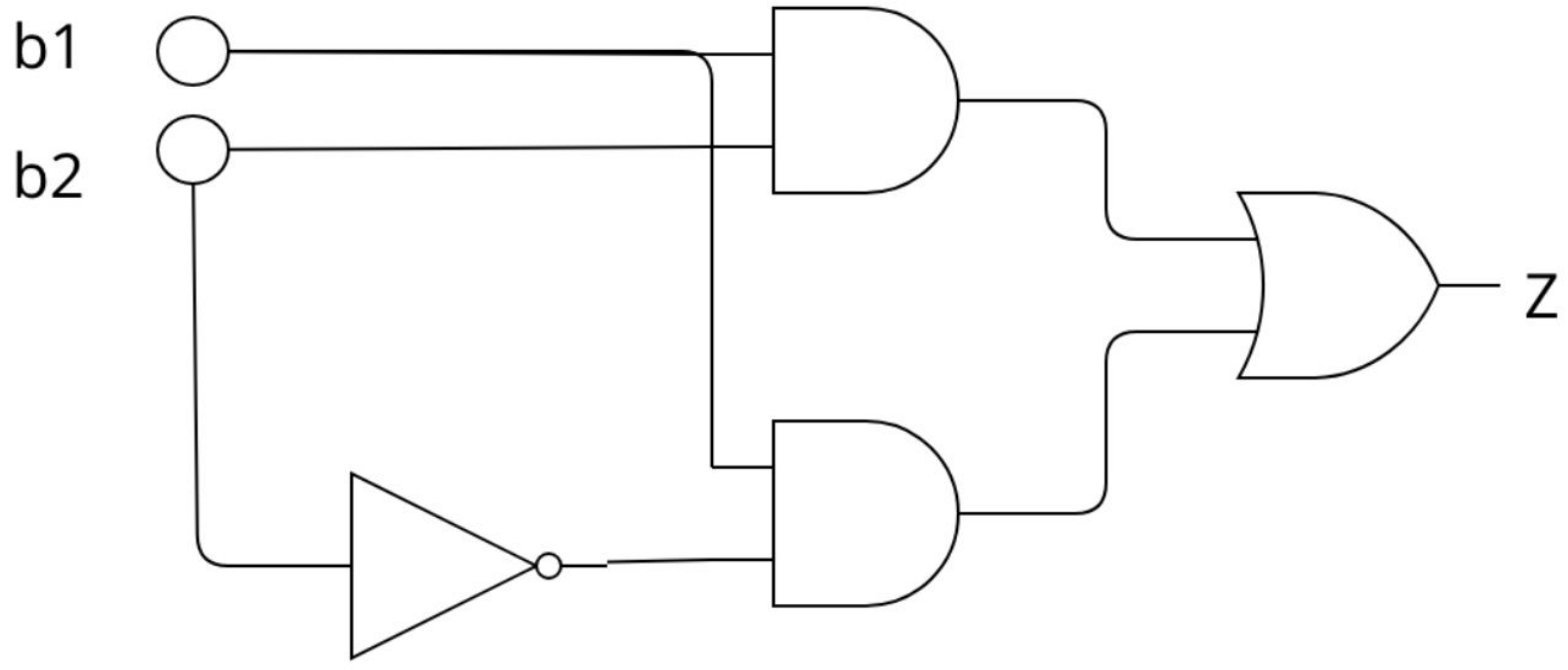
BFV - Batch Processing



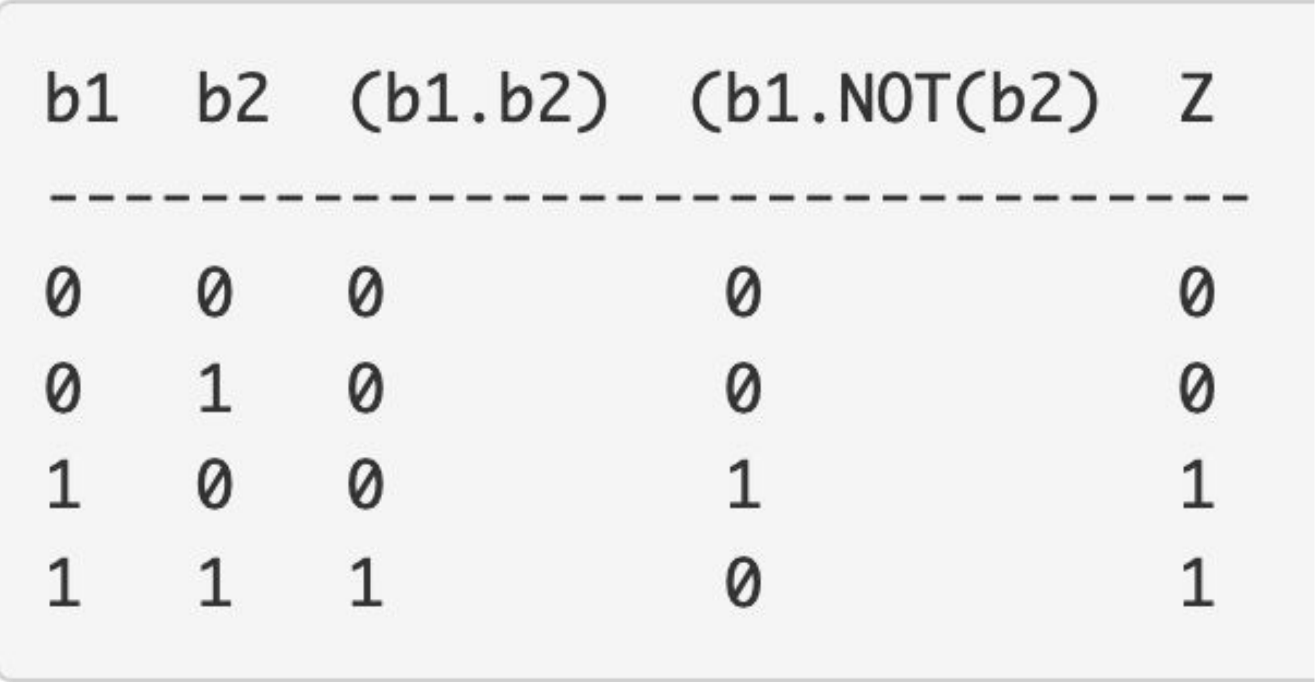
```
// Generate the relinearization key
cryptoContext->EvalMultKeyGen(keyPair.secretKey);

std::vector<int64_t> xval(count, 0ULL);

for (int i=0;i<count;i++) xval[i]=i;

Plaintext xplaintext   = cryptoContext->MakePackedPlaintext(xval);

// Encrypt values
auto ciphertext1 = cryptoContext->Encrypt(keyPair.publicKey, xplaintext);

// Square
auto ciphertextMult     = cryptoContext->EvalSquare(ciphertext1);

// Decrypt result
Plaintext plaintextAddRes;
cryptoContext->Decrypt(keyPair.secretKey, ciphertextMult, &plaintextAddRes);

std::cout << "Method: : " << type << std::endl;
std::cout << "Parameters " << parameters << std::endl << std::endl;
std::cout << "Ring dimension: " << cryptoContext->GetRingDimension() << "\n";
std::cout << "Modulus: : " << mod<< std::endl;

std::cout << "\nx: " << xplaintext << std::endl;
```

https://asecuritysite.com/openfhe/openfhe_08cpp

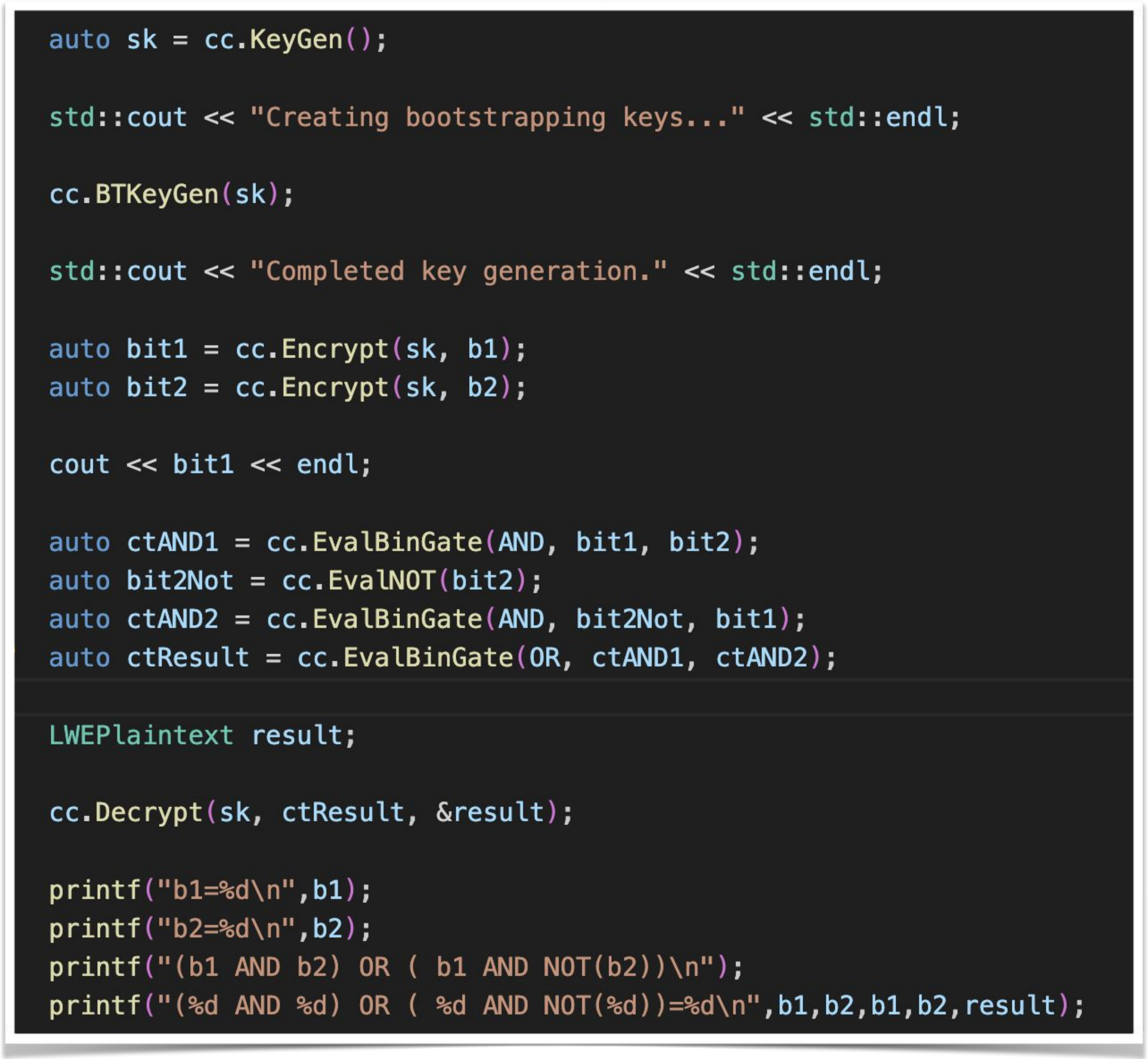
MD/FHEW



$$(b_1 . b_2) + (b_1 . \bar{b_2})$$



b1	b2	(b1.b2)	(b1.NOT(b2)	Z
0	0	0	0	0
0	1	0	0	0
1	0	0	1	1
1	1	1	0	1

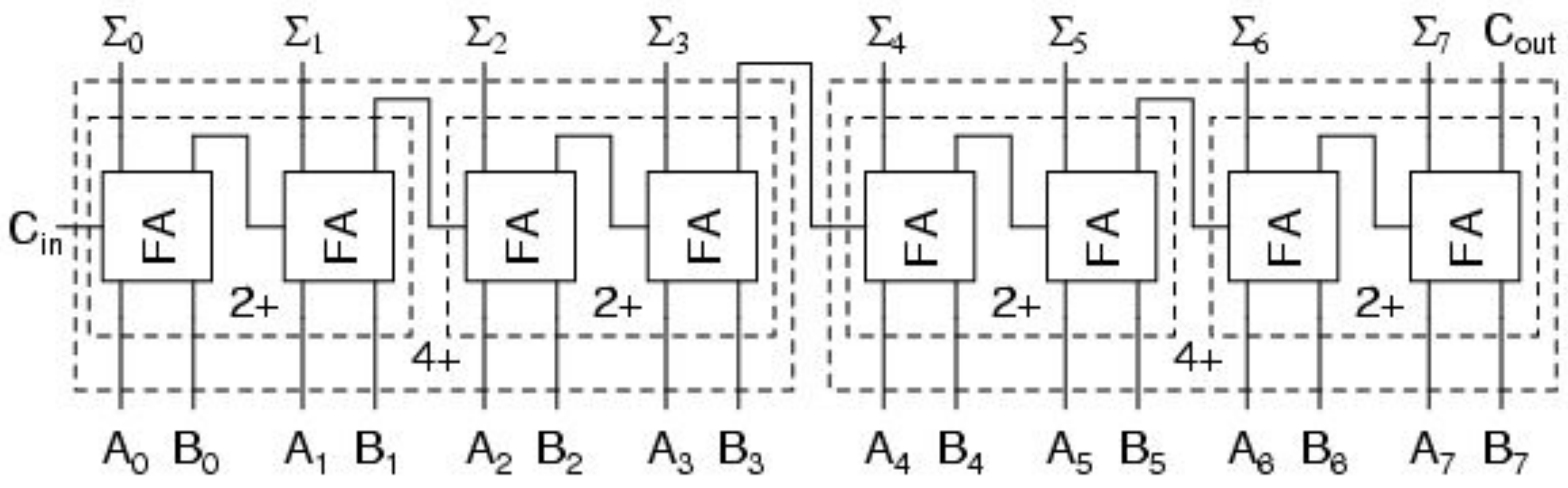


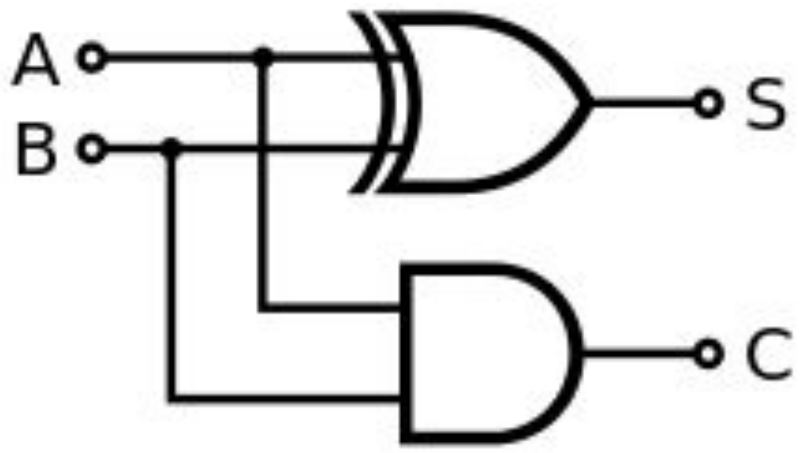
```
auto sk = cc.KeyGen();

std::cout << "Creating bootstrapping keys..." << std::endl;

cc.BTKeyGen(sk);

std::cout << "Completed key generation." << std::endl;

auto bit1 = cc.Encrypt(sk, b1);
auto bit2 = cc.Encrypt(sk, b2);

cout << bit1 << endl;

auto ctAND1 = cc.EvalBinGate(AND, bit1, bit2);
auto bit2Not = cc.EvalNOT(bit2);
auto ctAND2 = cc.EvalBinGate(AND, bit2Not, bit1);
auto ctResult = cc.EvalBinGate(OR, ctAND1, ctAND2);


LWEPlaintext result;

cc.Decrypt(sk, ctResult, &result);

printf("b1=%d\n",b1);
printf("b2=%d\n",b2);
printf("(b1 AND b2) OR ( b1 AND NOT(b2))\n");
printf("(%d AND %d) OR ( %d AND NOT(%d))=%d\n",b1,b2,b1,b2,result);
```

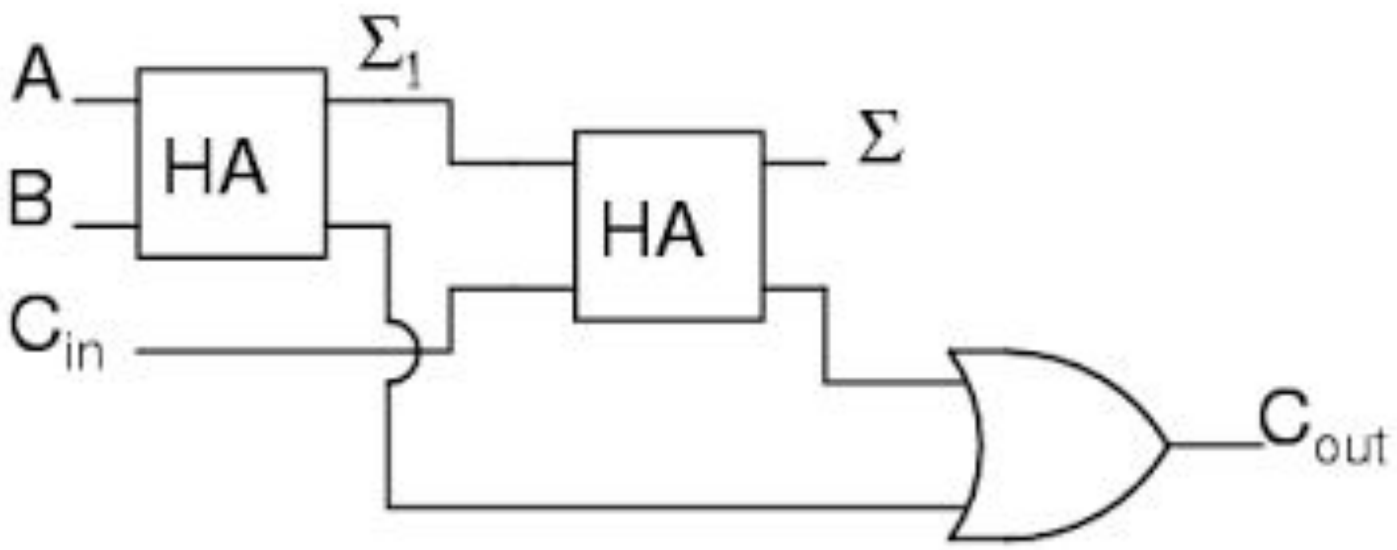
https://asecuritysite.com/openfhe/openfhe_09cpp

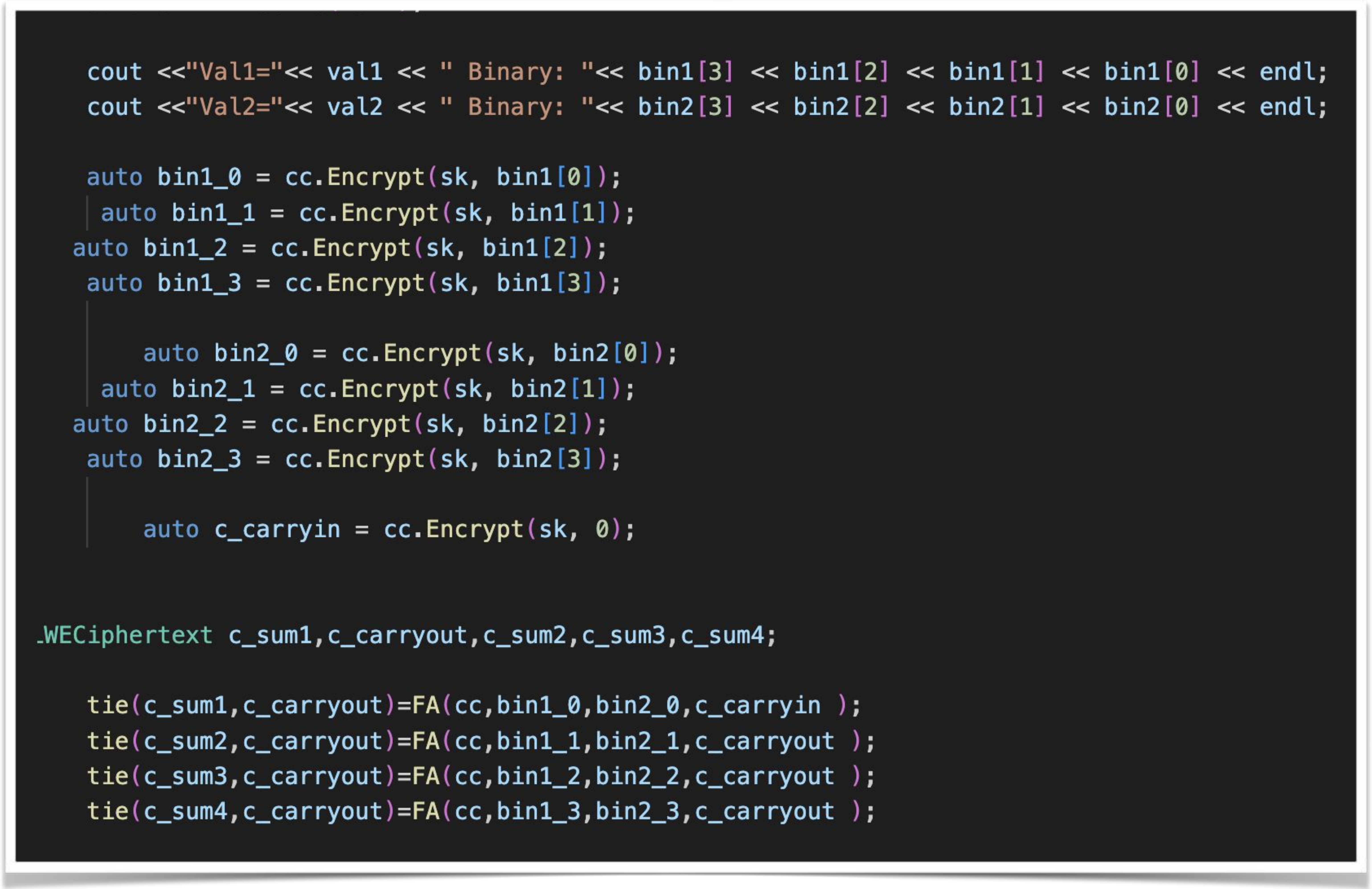
https://asecuritysite.com/openfhe/openfhe_09cpp_pke

MD/FHEW







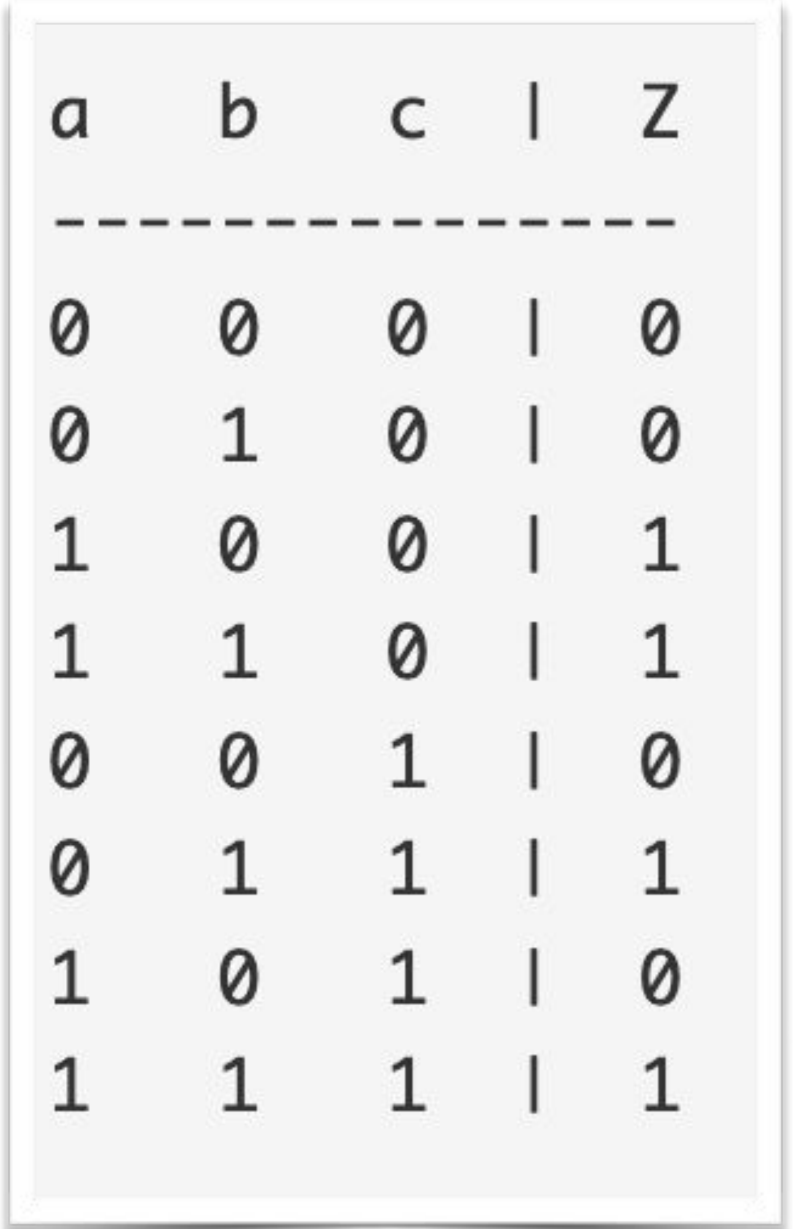


```
    cout <<"Val1="<< val1 << " Binary: "<< bin1[3] << bin1[2] << bin1[1] << bin1[0] << endl;
    cout <<"Val2="<< val2 << " Binary: "<< bin2[3] << bin2[2] << bin2[1] << bin2[0] << endl;

    auto bin1_0 = cc.Encrypt(sk, bin1[0]);
     auto bin1_1 = cc.Encrypt(sk, bin1[1]);
   auto bin1_2 = cc.Encrypt(sk, bin1[2]);
    auto bin1_3 = cc.Encrypt(sk, bin1[3]);

        auto bin2_0 = cc.Encrypt(sk, bin2[0]);
     auto bin2_1 = cc.Encrypt(sk, bin2[1]);
   auto bin2_2 = cc.Encrypt(sk, bin2[2]);
    auto bin2_3 = cc.Encrypt(sk, bin2[3]);

        auto c_carryin = cc.Encrypt(sk, 0);


WECiphertext c_sum1,c_carryout,c_sum2,c_sum3,c_sum4;

    tie(c_sum1,c_carryout)=FA(cc,bin1_0,bin2_0,c_carryin );
    tie(c_sum2,c_carryout)=FA(cc,bin1_1,bin2_1,c_carryout );
    tie(c_sum3,c_carryout)=FA(cc,bin1_2,bin2_2,c_carryout );
    tie(c_sum4,c_carryout)=FA(cc,bin1_3,bin2_3,c_carryout );
```

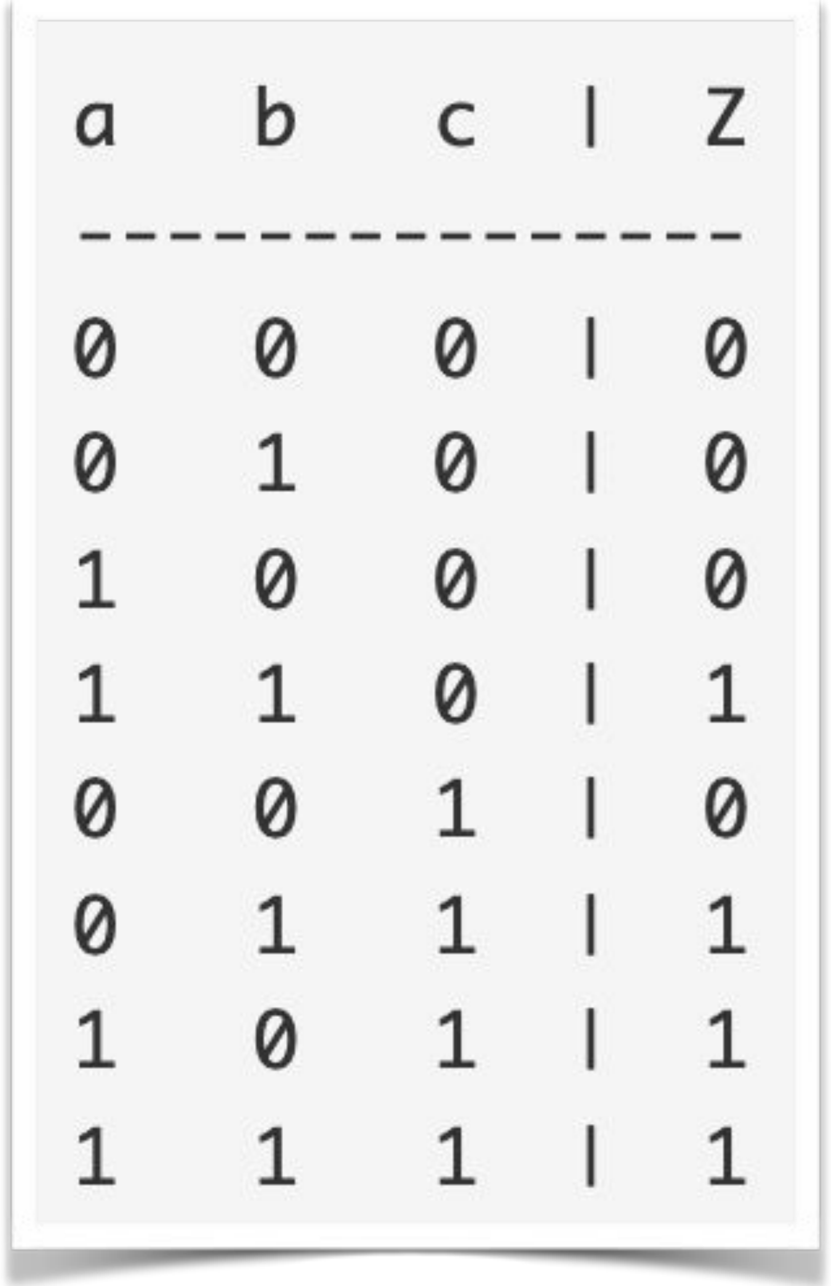
https://asecuritysite.com/openfhe/openfhe_11cpp

https://asecuritysite.com/openfhe/openfhe_11cpp_pke

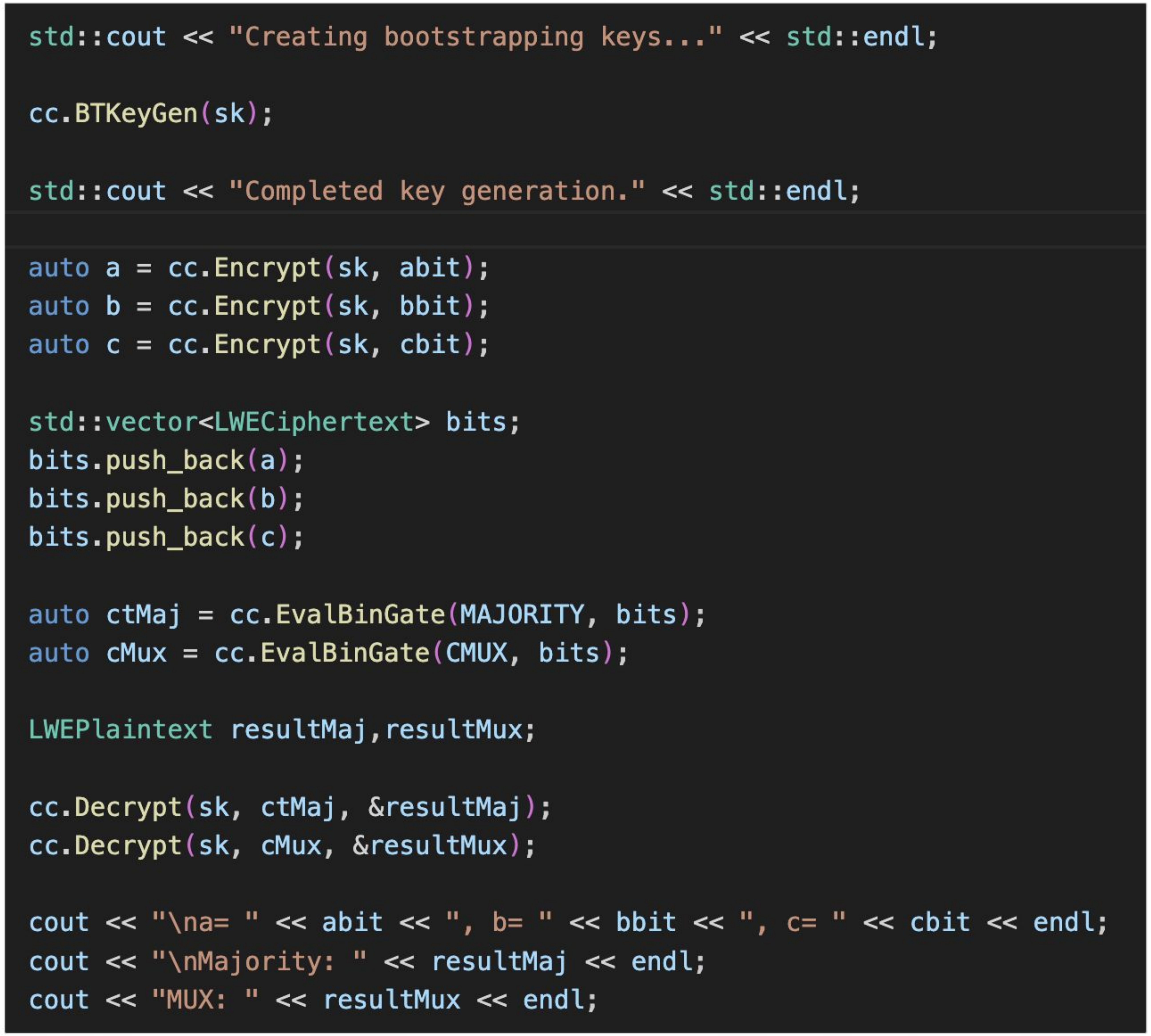
MD/FHEW - MUX and Majority



a	b	c	Z
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1



a	b	c	Z
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	1



```
std::cout << "Creating bootstrapping keys..." << std::endl;

cc.BTKeyGen(sk);

std::cout << "Completed key generation." << std::endl;

auto a = cc.Encrypt(sk, abit);
auto b = cc.Encrypt(sk, bbit);
auto c = cc.Encrypt(sk, cbit);

std::vector<LWECiphertext> bits;
bits.push_back(a);
bits.push_back(b);
bits.push_back(c);

auto ctMaj = cc.EvalBinGate(MAJORITY, bits);
auto cMux = cc.EvalBinGate(CMUX, bits);

LWEPlaintext resultMaj,resultMux;

cc.Decrypt(sk, ctMaj, &resultMaj);
cc.Decrypt(sk, cMux, &resultMux);

cout << "\na= " << abit << ", b= " << bbit << ", c= " << cbit << endl;
cout << "\nMajority: " << resultMaj << endl;
cout << "MUX: " << resultMux << endl;
```

https://asecuritysite.com/openfhe/openfhe_11cpp

https://asecuritysite.com/openfhe/openfhe_11cpp_pke

Cryptography

Prof Bill Buchanan OBE, FRSE

http://asecuritysite.com

Twitter: billatnapier

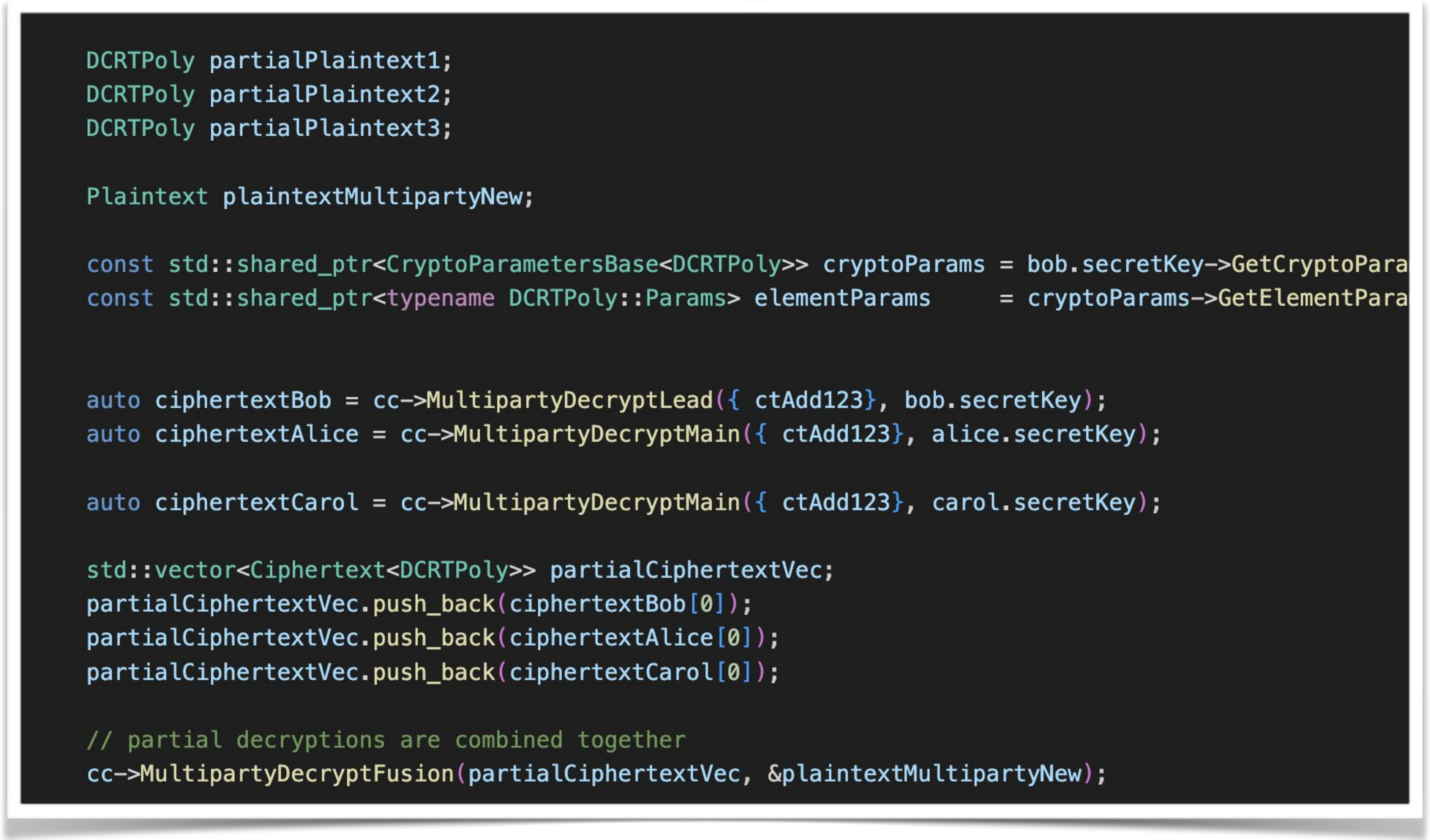
Threshold Encryption

OpenFHE

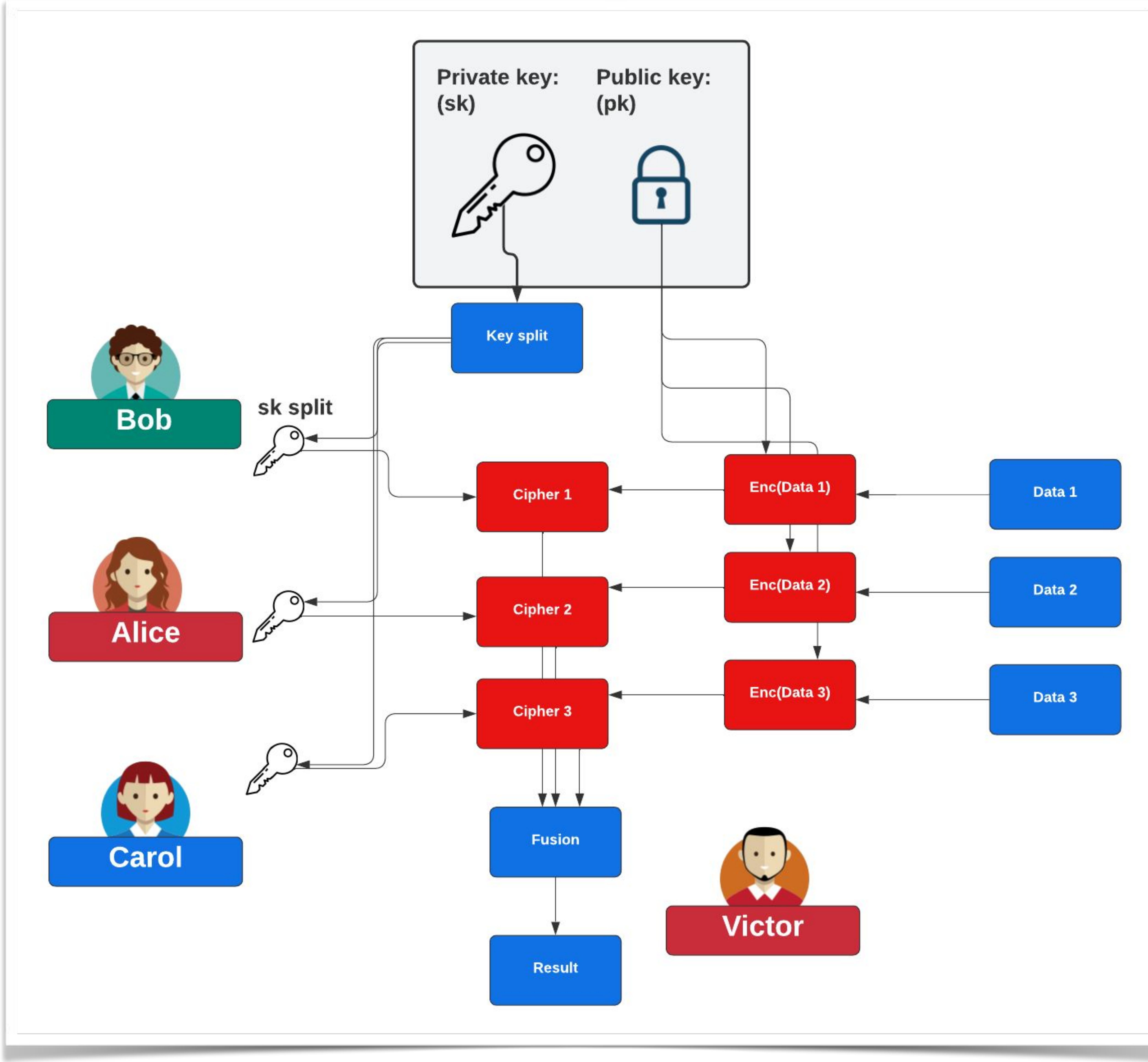
BLOCKPASS IDENTITY LAB

World-leading Collaboration between Blockpass IDN and Edinburgh Napier University

Threshold Encryption



```
DCRTPoly partialPlaintext1;
DCRTPoly partialPlaintext2;
DCRTPoly partialPlaintext3;

Plaintext plaintextMultipartyNew;

const std::shared_ptr<CryptoParametersBase<DCRTPoly>> cryptoParams = bob.secretKey->GetCryptoPara
const std::shared_ptr<typename DCRTPoly::Params> elementParams     = cryptoParams->GetElementPara


auto ciphertextBob = cc->MultipartyDecryptLead({ ctAdd123}, bob.secretKey);
auto ciphertextAlice = cc->MultipartyDecryptMain({ ctAdd123}, alice.secretKey);

auto ciphertextCarol = cc->MultipartyDecryptMain({ ctAdd123}, carol.secretKey);

std::vector<Ciphertext<DCRTPoly>> partialCiphertextVec;
partialCiphertextVec.push_back(ciphertextBob[0]);
partialCiphertextVec.push_back(ciphertextAlice[0]);
partialCiphertextVec.push_back(ciphertextCarol[0]);

// partial decryptions are combined together
cc->MultipartyDecryptFusion(partialCiphertextVec, &plaintextMultipartyNew);
```



https://asecuritysite.com/openfhe/openfhe_16cpp

https://asecuritysite.com/openfhe/openfhe_17cpp

Cryptography

Prof Bill Buchanan OBE, FRSE

http://asecuritysite.com

Twitter: billatnapier

Chebyshev Function

OpenFHE

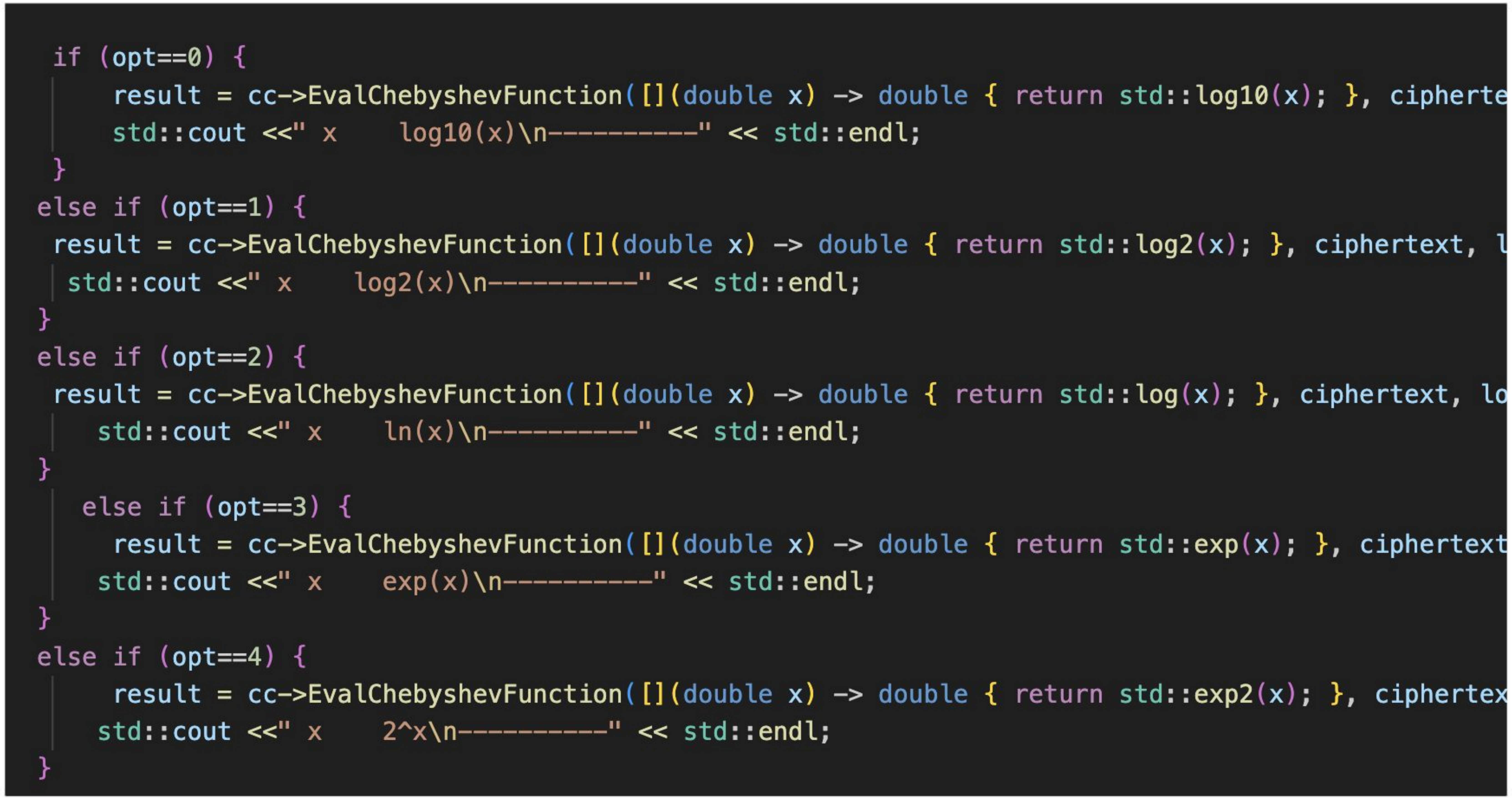
BLOCKPASS
IDENTITY
LAB

World-leading Collaboration between
Blockpass IDN and Edinburgh Napier University

Chebyshev Functions

With approximation theory, it is possible to determine an approximate polynomial p(x) that is approximate to a function f(x).

$T_0(x) = 1$
$T_1(x) = x$
$T_2(x) = 2x^2 - 1$
$T_3(x) = 4x^3 - 3x$
$T_4(x) = 8x^4 - 8x^2 + 1$
$T_5(x) = 16x^5 - 20x^3 + 5x$
$T_6(x) = 32x^6 - 48x^4 + 18x^2 - 1$
$T_7(x) = 64x^7 - 112x^5 + 56x^3 - 7x$
$T_8(x) = 128x^8 - 256x^6 + 160x^4 - 32x^2 + 1$
$T_9(x) = 256x^9 - 576x^7 + 432x^5 - 120x^3 + 9x$
$T_{10}(x) = 512x^{10} - 1280x^8 + 1120x^6 - 400x^4 + 50x^2 - 1$
$T_{11}(x) = 1024x^{11} - 2816x^9 + 2816x^7 - 1232x^5 + 220x^3 - 11x$



```
 if (opt==0) {
     result = cc->EvalChebyshevFunction([](double x) -> double { return std::log10(x); }, cipherte
     std::cout <<" x     log10(x)\n----------" << std::endl;
 }
else if (opt==1) {
 result = cc->EvalChebyshevFunction([](double x) -> double { return std::log2(x); }, ciphertext, l
  std::cout <<" x     log2(x)\n----------" << std::endl;
}
else if (opt==2) {
 result = cc->EvalChebyshevFunction([](double x) -> double { return std::log(x); }, ciphertext, lo
    std::cout <<" x     ln(x)\n----------" << std::endl;
}
   else if (opt==3) {
     result = cc->EvalChebyshevFunction([](double x) -> double { return std::exp(x); }, ciphertext
    std::cout <<" x     exp(x)\n----------" << std::endl;
}
else if (opt==4) {
     result = cc->EvalChebyshevFunction([](double x) -> double { return std::exp2(x); }, ciphertex
    std::cout <<" x     2^x\n----------" << std::endl;
}
```

https://asecuritysite.com/openfhe/openfhe_18cpp

Cryptography

Prof Bill Buchanan OBE, FRSE

http://asecuritysite.com

Twitter: billatnapier

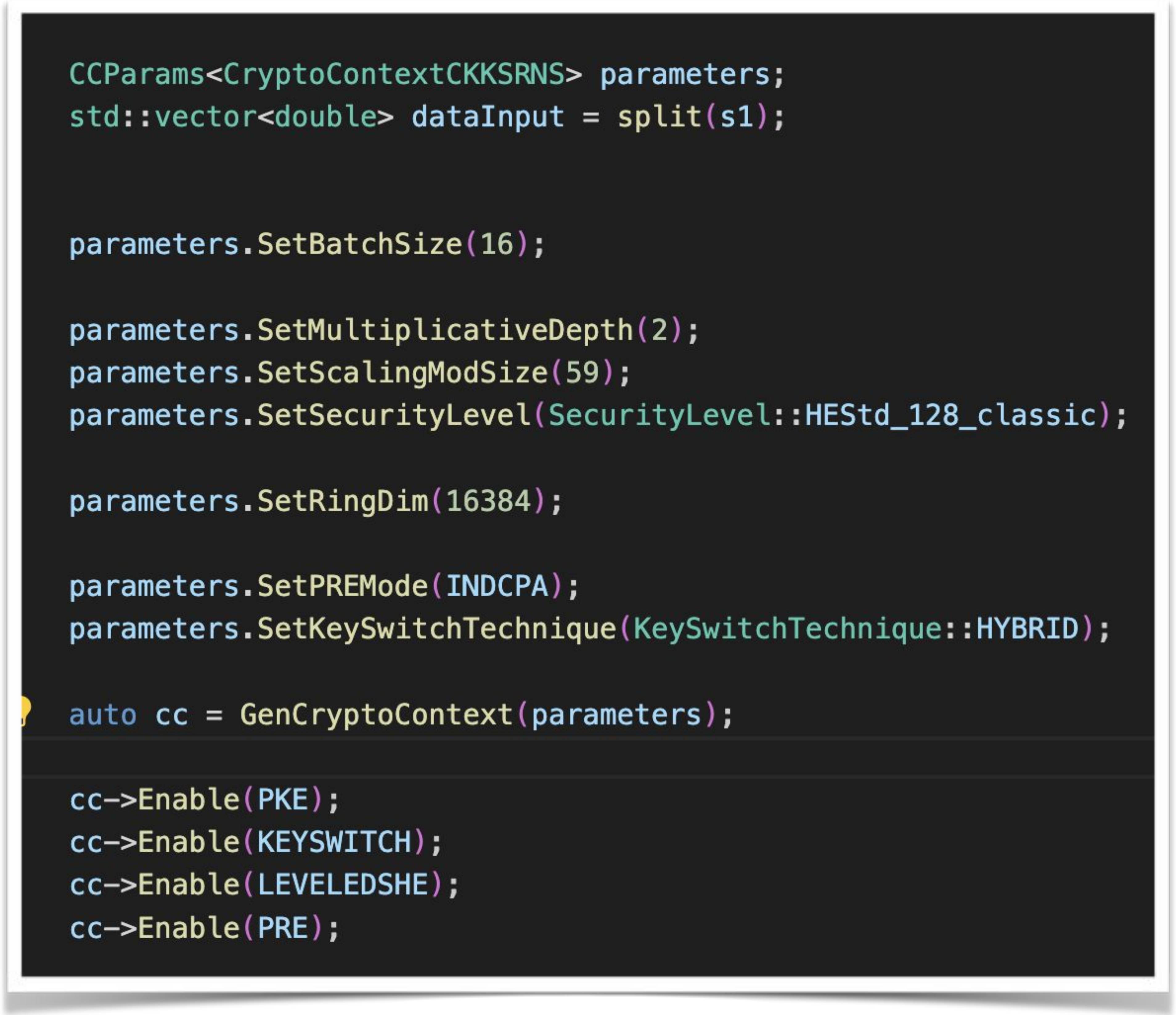
PRE

OpenFHE

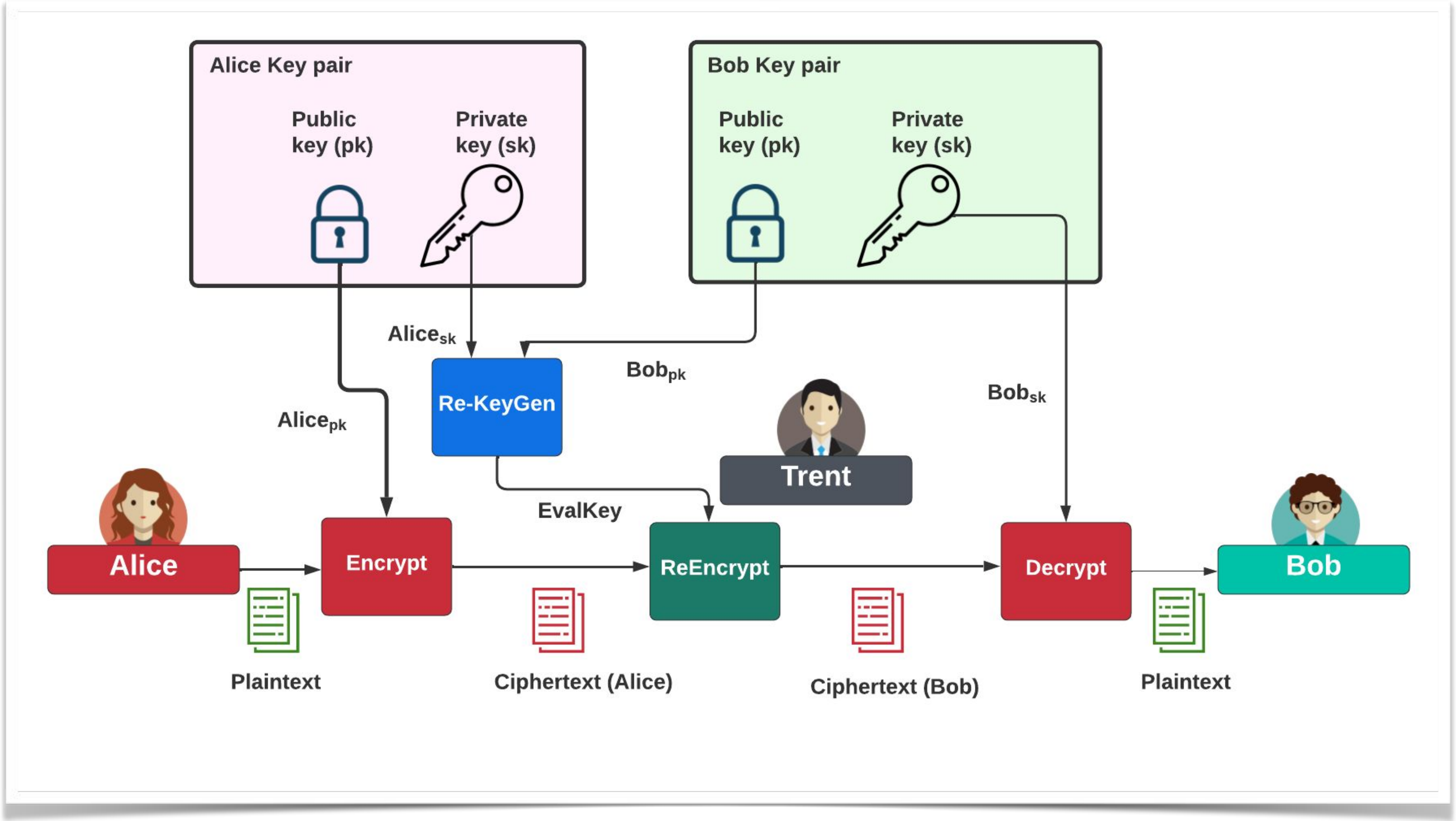
BLOCKPASS
IDENTITY
LAB

World-leading Collaboration between
Blockpass IDN and Edinburgh Napier University

Proxy Re-encryption (PRE)



```
CCParams<CryptoContextCKKSRNS> parameters;
std::vector<double> dataInput = split(s1);


parameters.SetBatchSize(16);

parameters.SetMultiplicativeDepth(2);
parameters.SetScalingModSize(59);
parameters.SetSecurityLevel(SecurityLevel::HEStd_128_classic);

parameters.SetRingDim(16384);

parameters.SetPREMode(INDCPA);
parameters.SetKeySwitchTechnique(KeySwitchTechnique::HYBRID);

auto cc = GenCryptoContext(parameters);

cc->Enable(PKE);
cc->Enable(KEYSWITCH);
cc->Enable(LEVELEDSHE);
cc->Enable(PRE);
```



https://asecuritysite.com/openfhe/openfhe_21cpp

Cryptography

Prof Bill Buchanan OBE, FRSE

http://asecuritysite.com

Twitter: billatnapier

Logistic (sigmoid) Function

OpenFHE

BLOCKPASS
IDENTITY
LAB

World-leading Collaboration between
Blockpass IDN and Edinburgh Napier University

Cryptography

Prof Bill Buchanan OBE, FRSE

http://asecuritysite.com

Twitter: billatnapier

Polynomial Evaluation

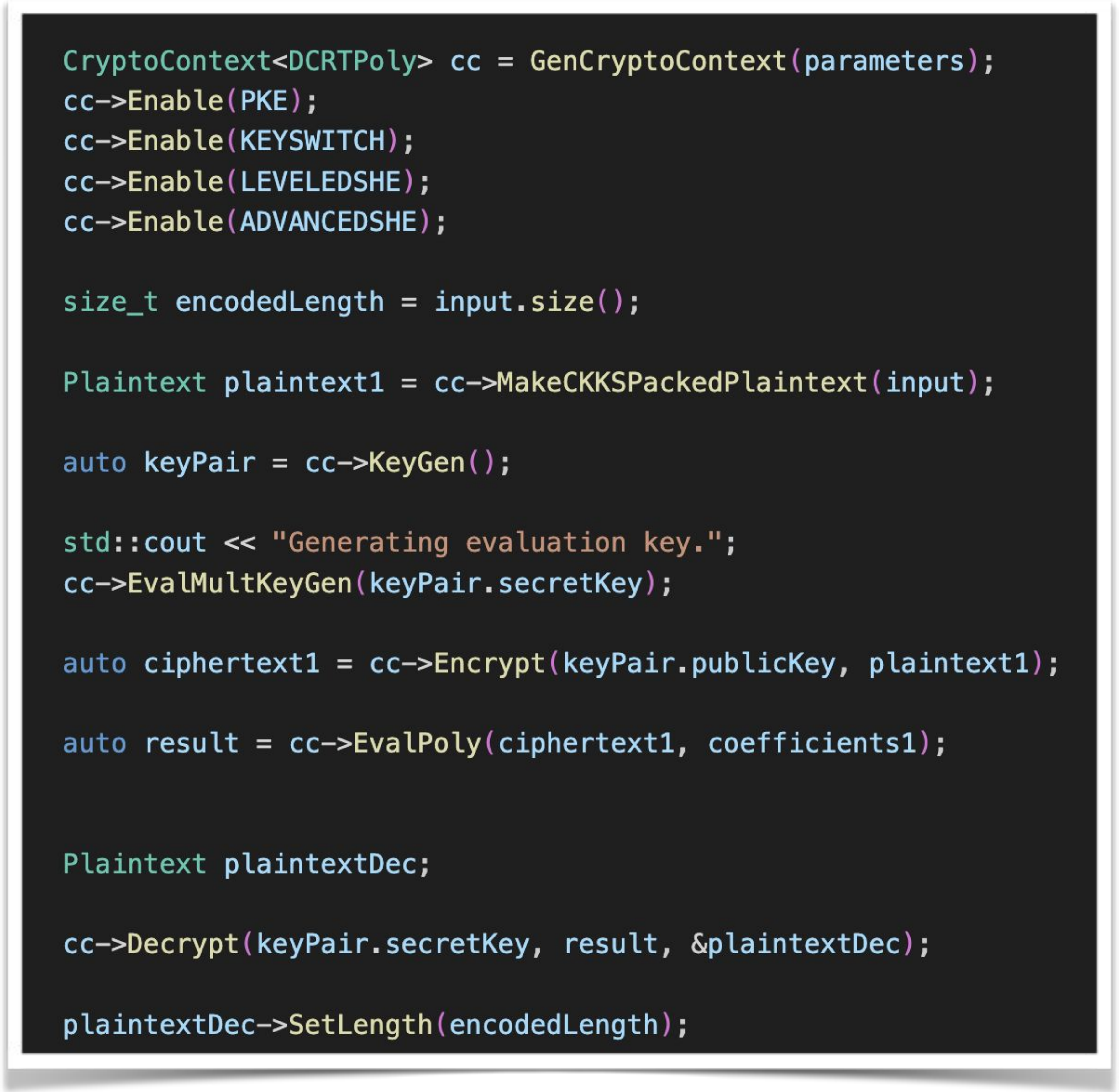
OpenFHE

BLOCKPASS
IDENTITY
LAB

World-leading Collaboration between
Blockpass IDN and Edinburgh Napier University

Polynomial Evaluation

A polynomial takes the form form of $p(x) = a_n . x^n + a_{n-1} . x^{n-1} + a_1 . x + a_0$, and where $a_0 . . . a_n$ are the coeffients of the powers, and n is the maximum power of the polynomial. With CKKS in OpenFHE, we can evaluate the result of a polynomial for a given range of x values. For example, if we have $p(x) = 5.x^2 + 3.x + 7$ will give a result of $p(2) = 33$.



```
CryptoContext<DCRTPoly> cc = GenCryptoContext(parameters);
cc->Enable(PKE);
cc->Enable(KEYSWITCH);
cc->Enable(LEVELEDSHE);
cc->Enable(ADVANCEDSHE);

size_t encodedLength = input.size();

Plaintext plaintext1 = cc->MakeCKKSPackedPlaintext(input);

auto keyPair = cc->KeyGen();

std::cout << "Generating evaluation key.";
cc->EvalMultKeyGen(keyPair.secretKey);

auto ciphertext1 = cc->Encrypt(keyPair.publicKey, plaintext1);

auto result = cc->EvalPoly(ciphertext1, coefficients1);


Plaintext plaintextDec;

cc->Decrypt(keyPair.secretKey, result, &plaintextDec);

plaintextDec->SetLength(encodedLength);
```

https://asecuritysite.com/openfhe/openfhe_20cpp

Privacy-aware ML

Prof Bill Buchanan OBE, FRSE

http://asecuritysite.com

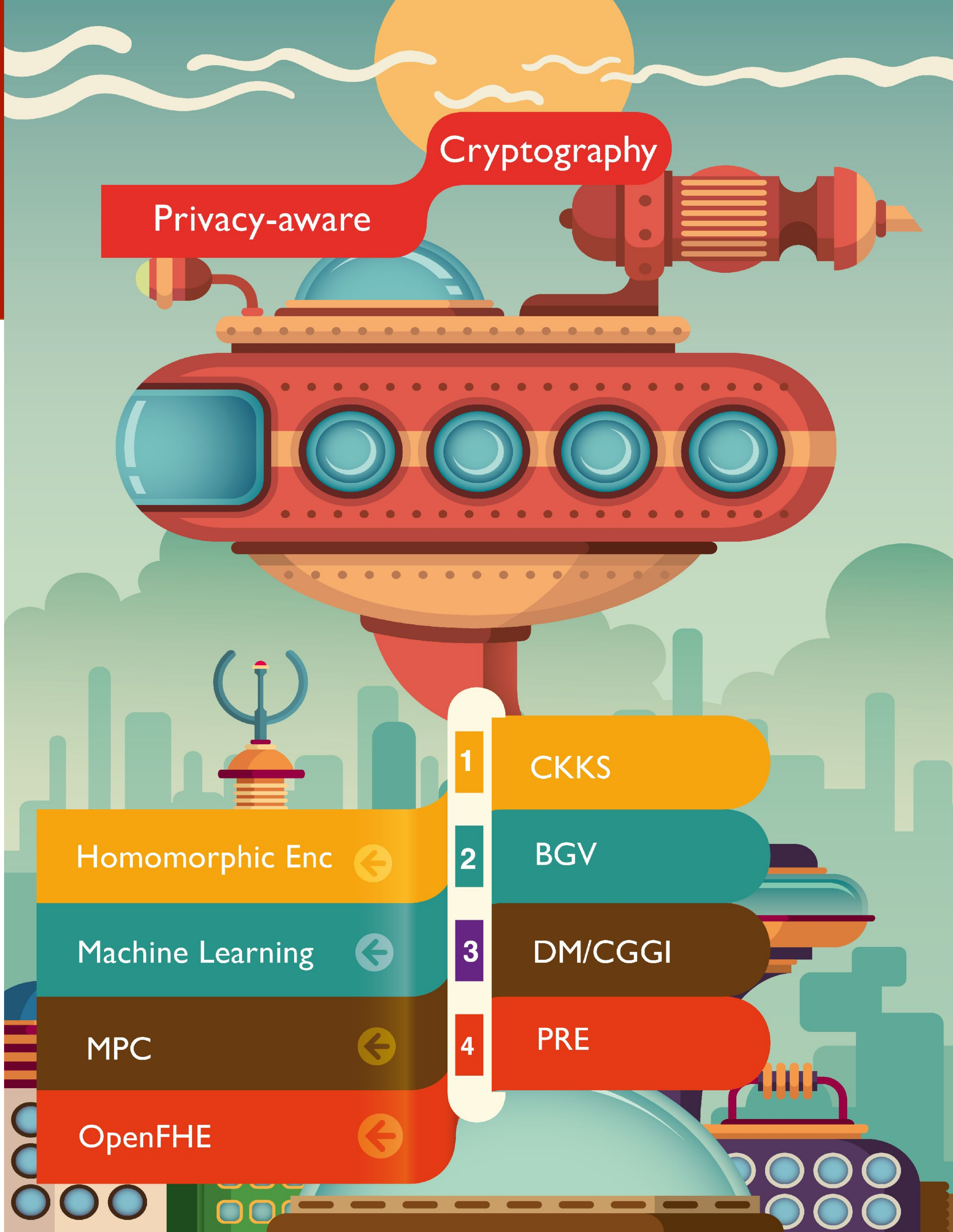
Twitter: billatnapier

Privacy-aware ML

OpenFHE

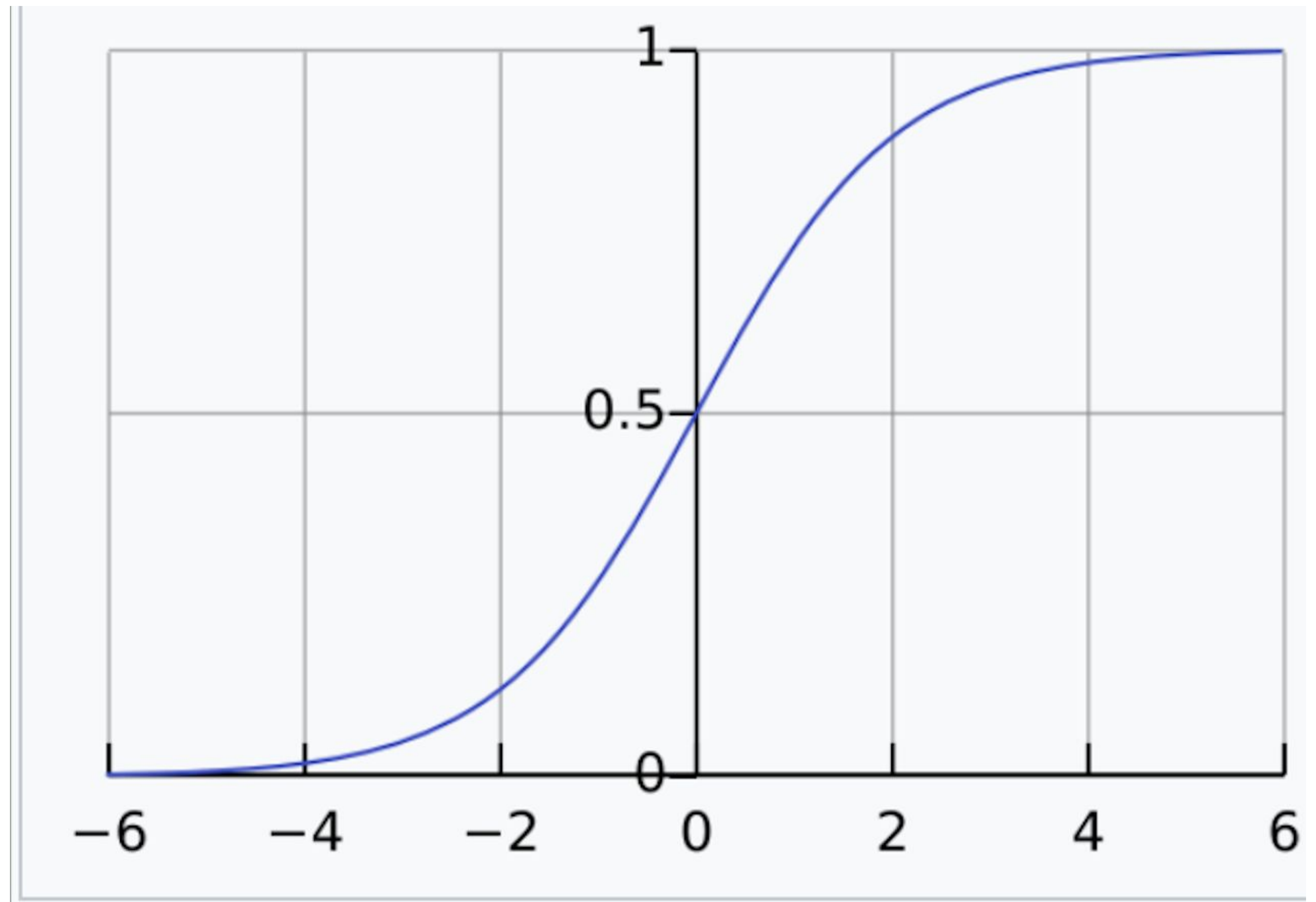
BLOCKPASS IDENTITY LAB

World-leading Collaboration between Blockpass IDN and Edinburgh Napier University



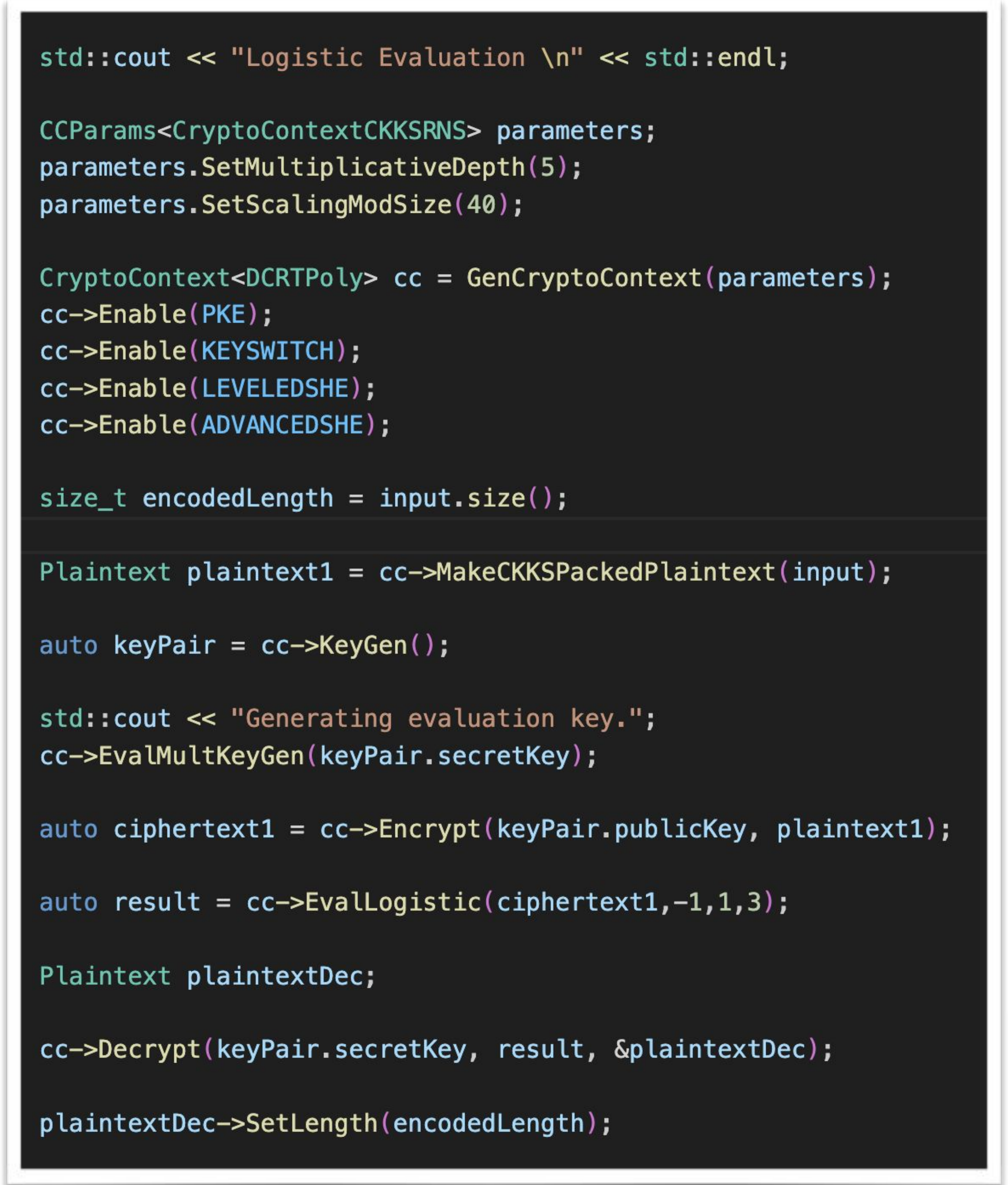
Logistic Function

With homomorphic encryption we can represent a mathematical operation in the form for a homomorphic equation. One of the most widely used methods is to use Chebyshev polynomials, and which allows the mapping of the function to a Chebyshev approximation. In this case, we will use homomorphic encryption to approximate a logistic function (and which is represented by f(x)=1/(1+e^{-x}).



Standard logistic function where $L = 1, k = 1, x_0 = 0.$

$$f(x) = \frac{1}{1+e^{-x}}$$



```
std::cout << "Logistic Evaluation \n" << std::endl;

CCParams<CryptoContextCKKSRNS> parameters;
parameters.SetMultiplicativeDepth(5);
parameters.SetScalingModSize(40);

CryptoContext<DCRTPoly> cc = GenCryptoContext(parameters);
cc->Enable(PKE);
cc->Enable(KEYSWITCH);
cc->Enable(LEVELEDSHE);
cc->Enable(ADVANCEDSHE);

size_t encodedLength = input.size();

Plaintext plaintext1 = cc->MakeCKKSPackedPlaintext(input);

auto keyPair = cc->KeyGen();

std::cout << "Generating evaluation key.";
cc->EvalMultKeyGen(keyPair.secretKey);

auto ciphertext1 = cc->Encrypt(keyPair.publicKey, plaintext1);

auto result = cc->EvalLogistic(ciphertext1,-1,1,3);

Plaintext plaintextDec;

cc->Decrypt(keyPair.secretKey, result, &plaintextDec);

plaintextDec->SetLength(encodedLength);
```

https://asecuritysite.com/openfhe/openfhe_19cpp

Matrix Operations

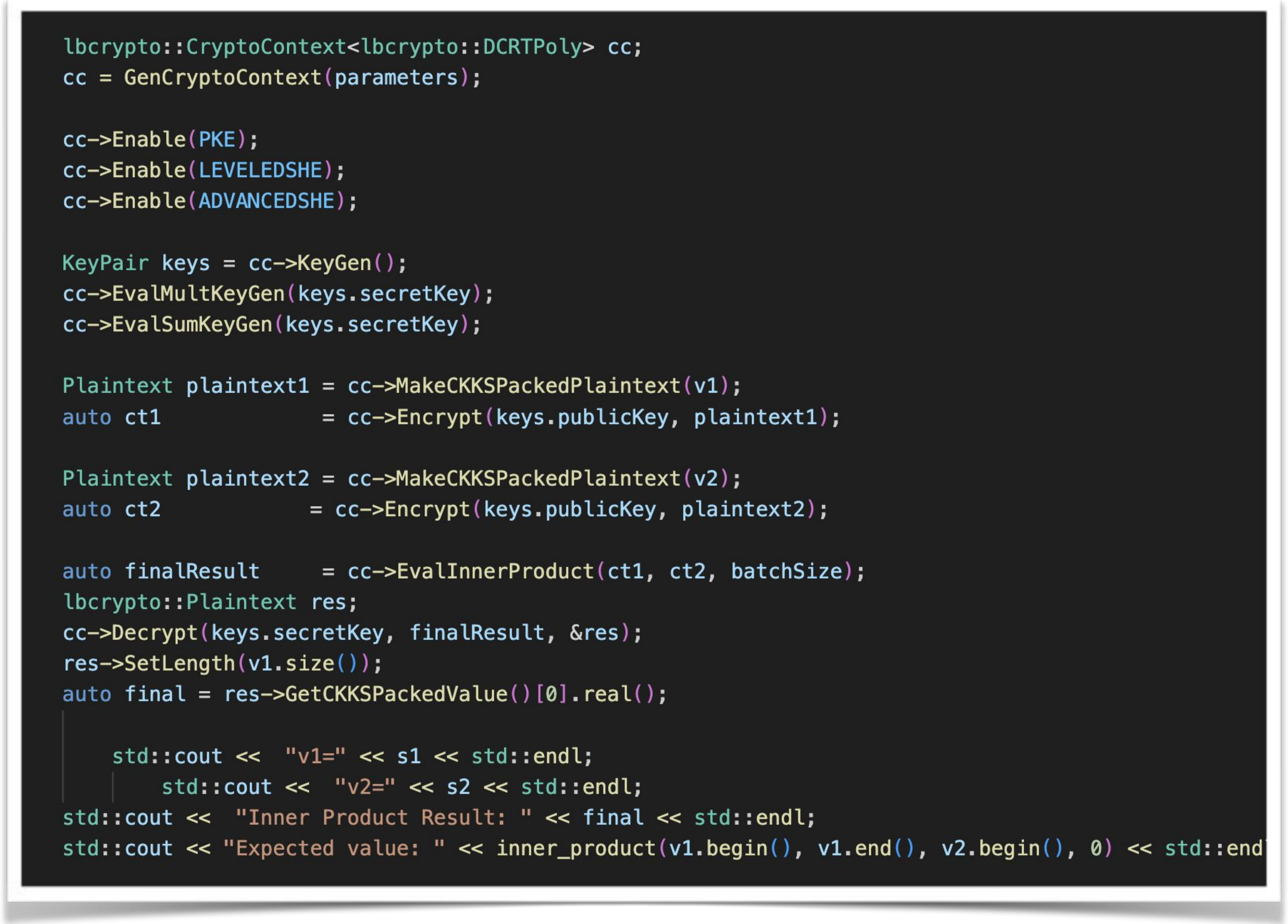
The inner product of two vectors of a and b is represented by $\langle a, b\rangle$. It is the dot product of two vectors, and represented as $\langle a, b\rangle = a.\,y cos(\theta)$, and where θ is the angle between the two vectors.

If we have a vector of x=(10,20,15), then the magniture will be:

$$a = \sqrt{10^2 + 20^2 + 15^2} = 26.93$$

If we have the same vector of b=(10,20,15), we will have the same magnitude. The inner product will then be:

$$\langle a, b\rangle = |a|.\,|b|.\,cos(\theta) = 26.93 \times 26.93.cos(0) = 725$$

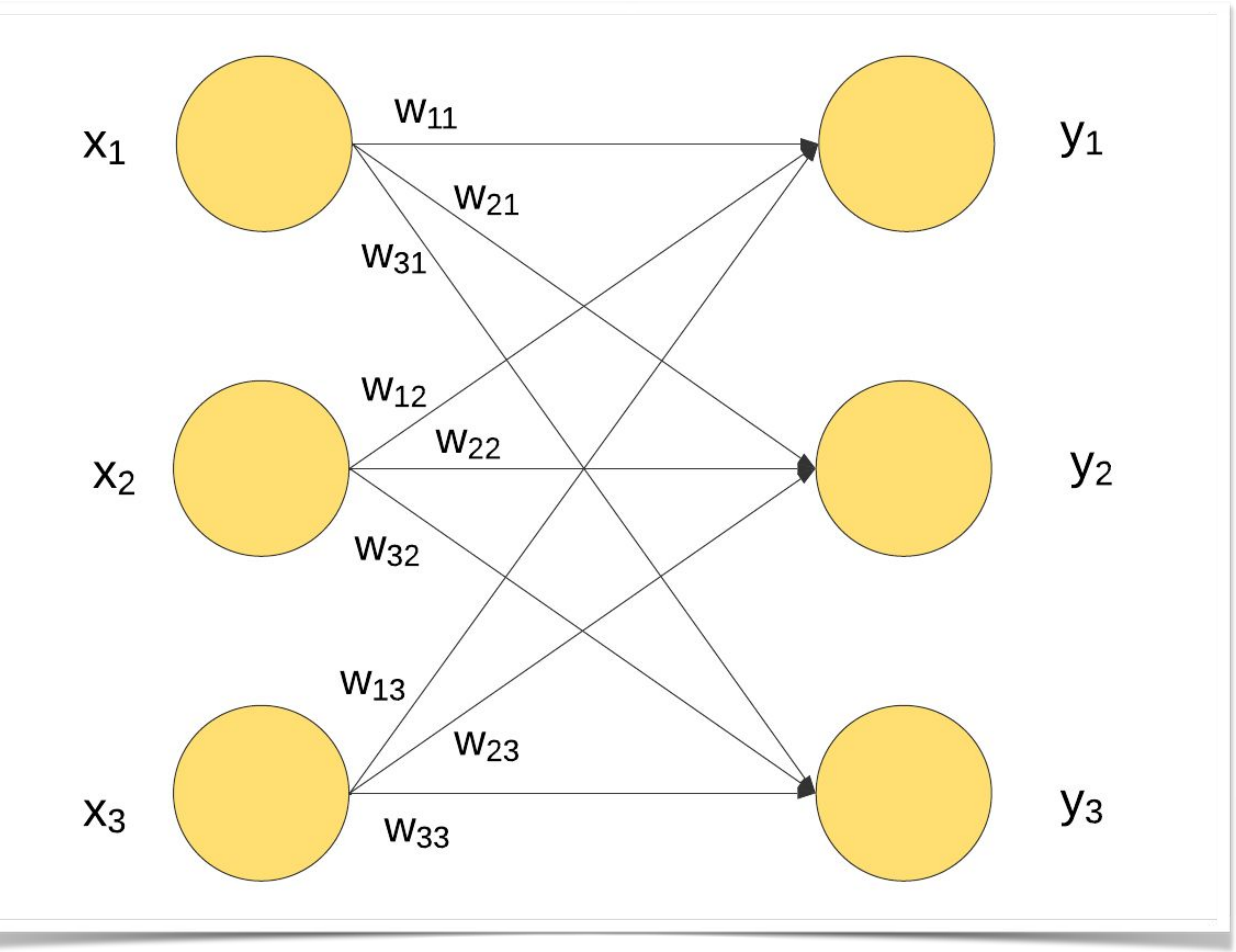


```
lbcrypto::CryptoContext<lbcrypto::DCRTPoly> cc;
cc = GenCryptoContext(parameters);

cc->Enable(PKE);
cc->Enable(LEVELEDSHE);
cc->Enable(ADVANCEDSHE);

KeyPair keys = cc->KeyGen();
cc->EvalMultKeyGen(keys.secretKey);
cc->EvalSumKeyGen(keys.secretKey);

Plaintext plaintext1 = cc->MakeCKKSPackedPlaintext(v1);
auto ct1             = cc->Encrypt(keys.publicKey, plaintext1);

Plaintext plaintext2 = cc->MakeCKKSPackedPlaintext(v2);
auto ct2            = cc->Encrypt(keys.publicKey, plaintext2);

auto finalResult     = cc->EvalInnerProduct(ct1, ct2, batchSize);
lbcrypto::Plaintext res;
cc->Decrypt(keys.secretKey, finalResult, &res);
res->SetLength(v1.size());
auto final = res->GetCKKSPackedValue()[0].real();

    std::cout <<  "v1=" << s1 << std::endl;
        std::cout <<  "v2=" << s2 << std::endl;
std::cout <<  "Inner Product Result: " << final << std::endl;
std::cout << "Expected value: " << inner_product(v1.begin(), v1.end(), v2.begin(), 0) << std::end
```

https://asecuritysite.com/openfhe/openfhe_13cpp

https://asecuritysite.com/openfhe/openfhe_14cpp

Matrix Operations



If we have a vector of the form:

$$v_1 = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}$$

and a matrix of:

$$m_1 = \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \\ w_{13} & w_{23} & w_{33} \end{bmatrix}$$

We now get:

$$v_1 . m_1 = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \\ w_{13} & w_{23} & w_{33} \end{bmatrix}$$

and:

$$v_1 . m_1 = \begin{bmatrix} x_1 . w_{11} + x_2 . w_{21} + x_3 . w_{31} & x_1 . w_{12} + x_2 . w_{22} + x_3 . w_{32} & x_1 . w_{13} + x_2 . w_{23} + x_3 . w_{33} \end{bmatrix}$$

Thus we get:

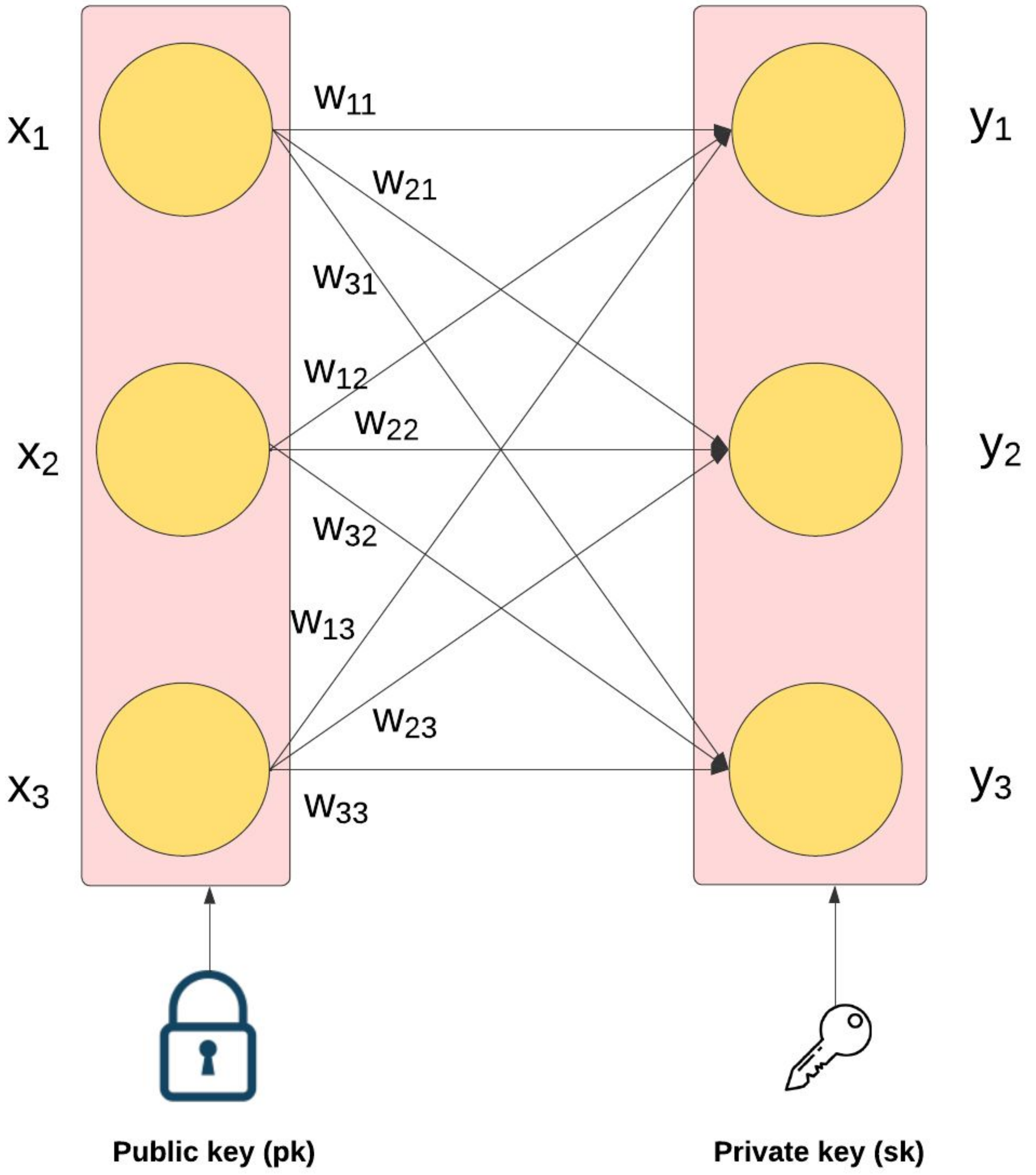
$$y_1 = x_1 . w_{11} + x_2 . w_{21} + x_3 . w_{31}$$

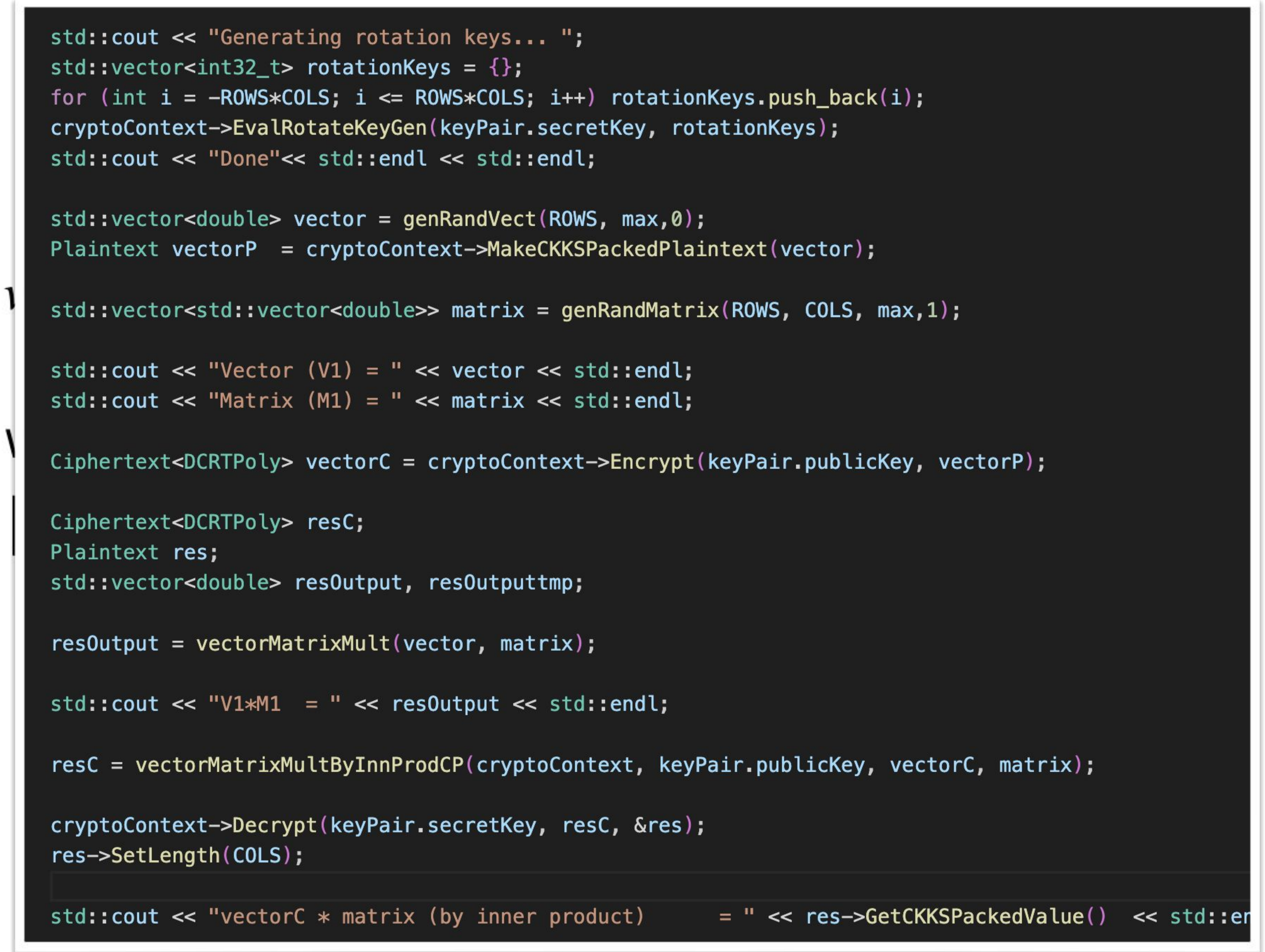
$$y_2 = x_1 . w_{12} + x_2 . w_{22} + x_3 . w_{32}$$

$$y_3 = x_1 . w_{13} + x_2 . w_{23} + x_3 . w_{33}$$

https://asecuritysite.com/openfhe/openfhe_27cpp

Matrix Operations



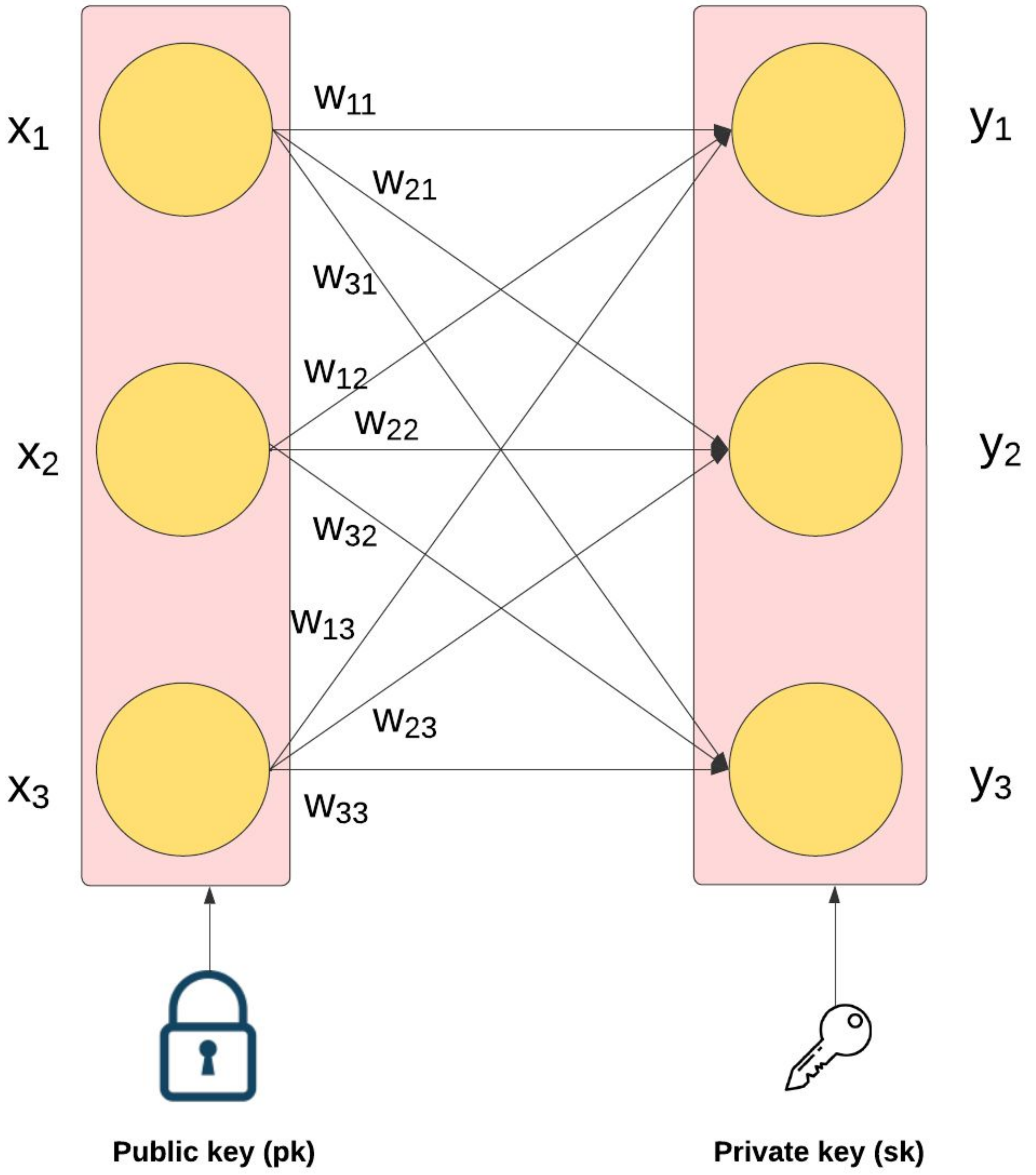


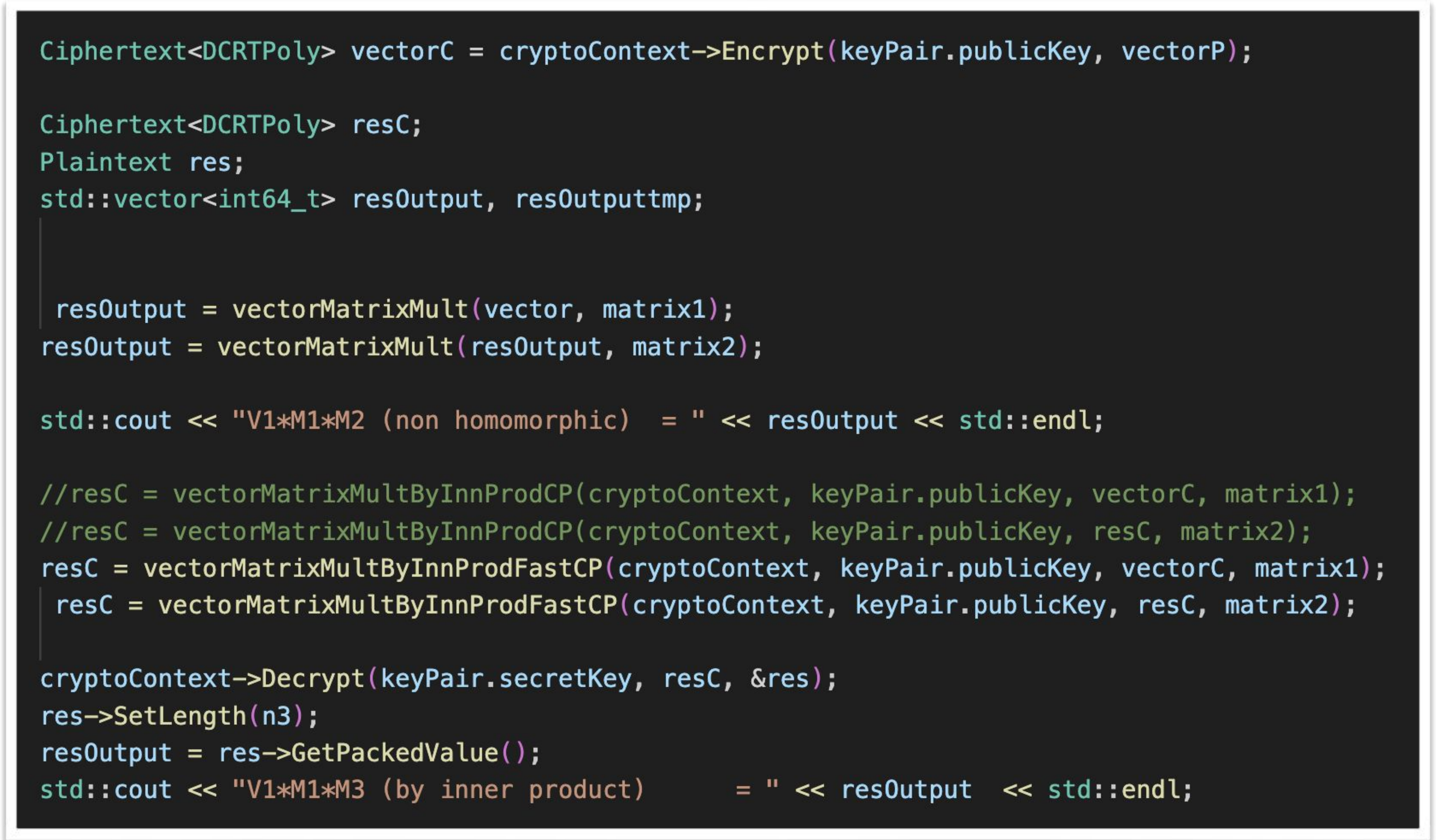
```
std::cout << "Generating rotation keys... ";
std::vector<int32_t> rotationKeys = {};
for (int i = -ROWS*COLS; i <= ROWS*COLS; i++) rotationKeys.push_back(i);
cryptoContext->EvalRotateKeyGen(keyPair.secretKey, rotationKeys);
std::cout << "Done"<< std::endl << std::endl;

std::vector<double> vector = genRandVect(ROWS, max,0);
Plaintext vectorP  = cryptoContext->MakeCKKSPackedPlaintext(vector);

std::vector<std::vector<double>> matrix = genRandMatrix(ROWS, COLS, max,1);

std::cout << "Vector (V1) = " << vector << std::endl;
std::cout << "Matrix (M1) = " << matrix << std::endl;

Ciphertext<DCRTPoly> vectorC = cryptoContext->Encrypt(keyPair.publicKey, vectorP);

Ciphertext<DCRTPoly> resC;
Plaintext res;
std::vector<double> resOutput, resOutputtmp;

resOutput = vectorMatrixMult(vector, matrix);

std::cout << "V1*M1  = " << resOutput << std::endl;

resC = vectorMatrixMultByInnProdCP(cryptoContext, keyPair.publicKey, vectorC, matrix);

cryptoContext->Decrypt(keyPair.secretKey, resC, &res);
res->SetLength(COLS);

std::cout << "vectorC * matrix (by inner product)      = " << res->GetCKKSPackedValue()  << std::er
```

https://asecuritysite.com/openfhe/openfhe_27cpp

https://asecuritysite.com/openfhe/openfhe_26cpp

Matrix Operations



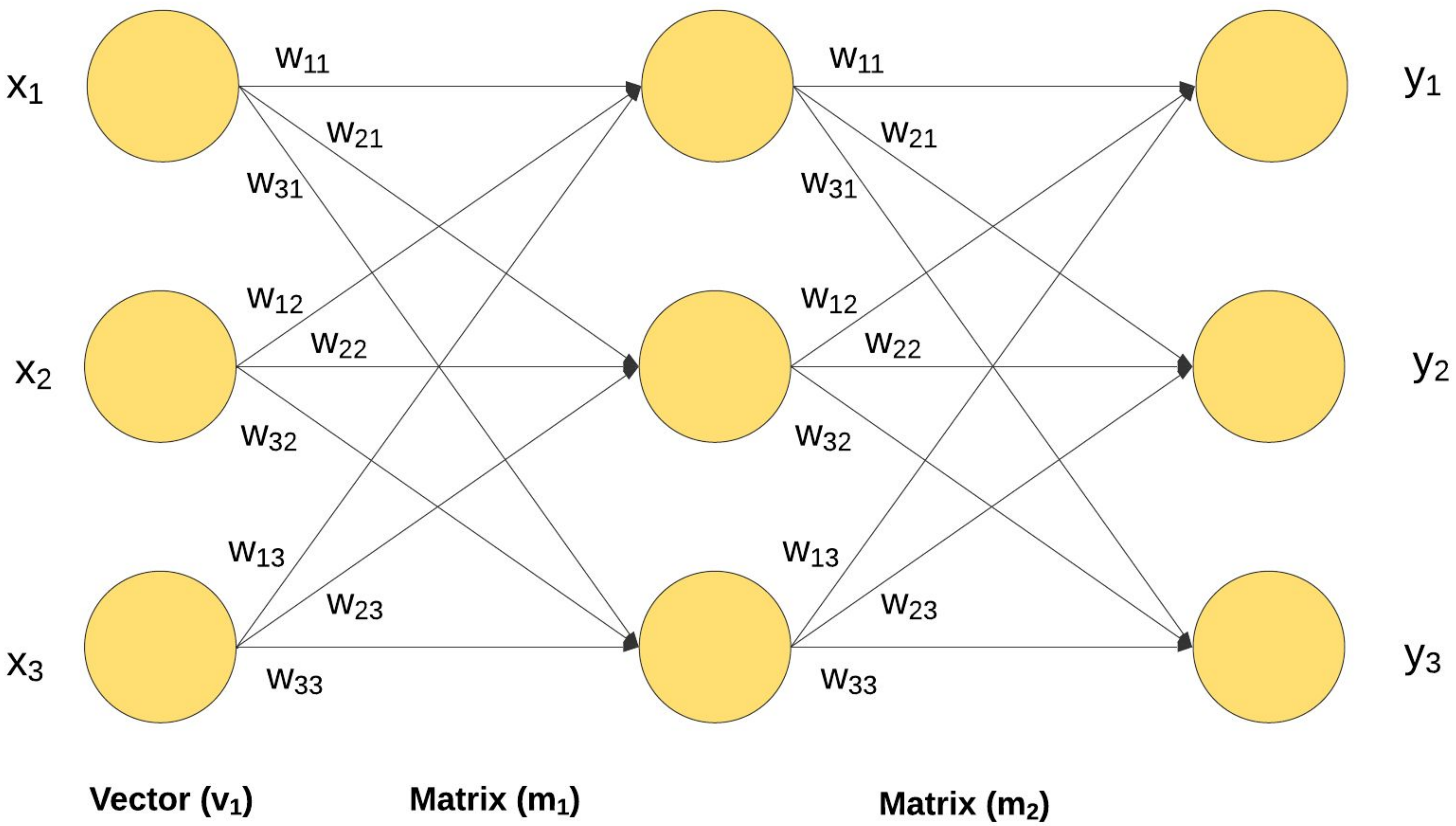


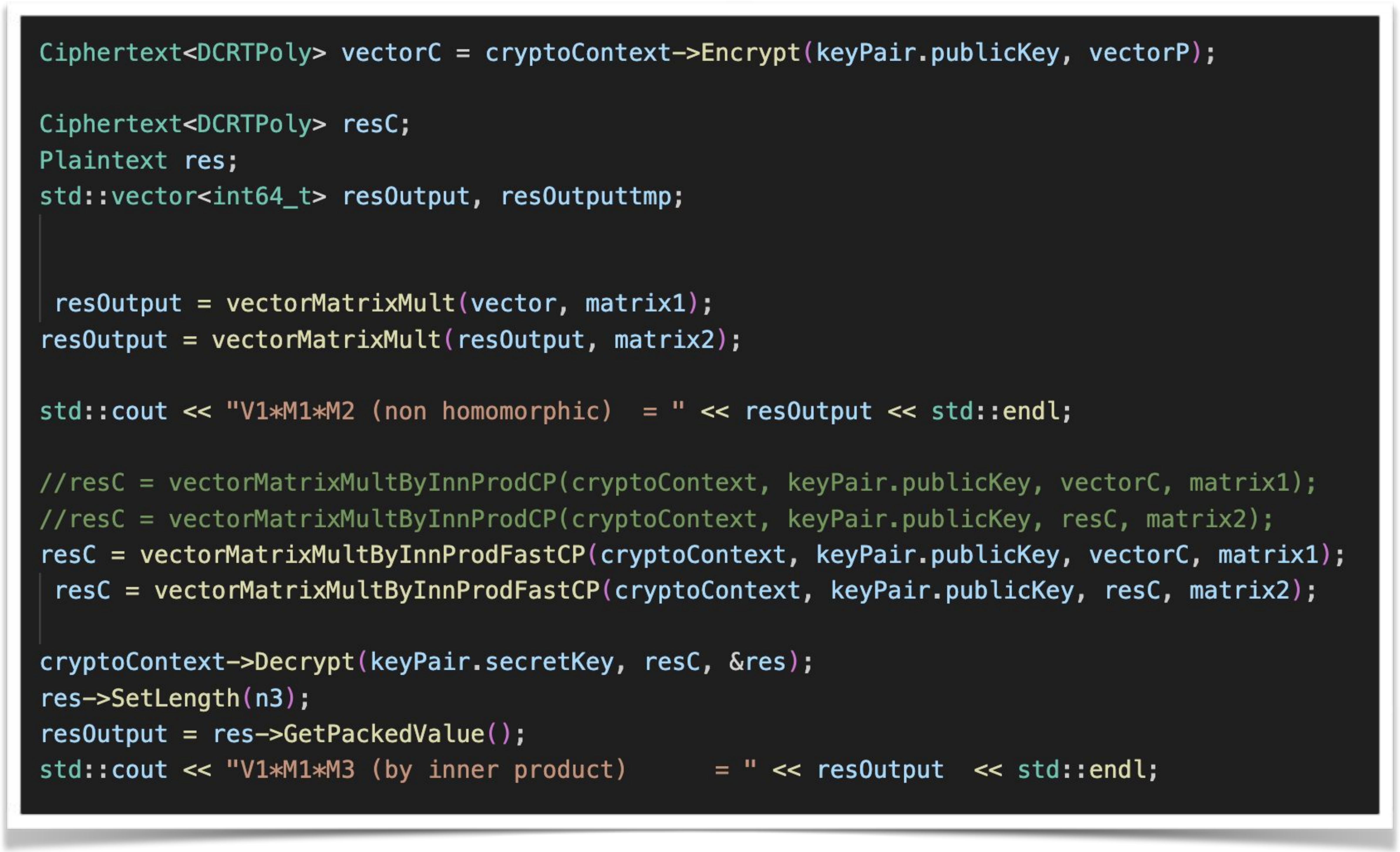
```
Ciphertext<DCRTPoly> vectorC = cryptoContext->Encrypt(keyPair.publicKey, vectorP);

Ciphertext<DCRTPoly> resC;
Plaintext res;
std::vector<int64_t> resOutput, resOutputtmp;


 resOutput = vectorMatrixMult(vector, matrix1);
resOutput = vectorMatrixMult(resOutput, matrix2);

std::cout << "V1*M1*M2 (non homomorphic)  = " << resOutput << std::endl;

//resC = vectorMatrixMultByInnProdCP(cryptoContext, keyPair.publicKey, vectorC, matrix1);
//resC = vectorMatrixMultByInnProdCP(cryptoContext, keyPair.publicKey, resC, matrix2);
resC = vectorMatrixMultByInnProdFastCP(cryptoContext, keyPair.publicKey, vectorC, matrix1);
 resC = vectorMatrixMultByInnProdFastCP(cryptoContext, keyPair.publicKey, resC, matrix2);

cryptoContext->Decrypt(keyPair.secretKey, resC, &res);
res->SetLength(n3);
resOutput = res->GetPackedValue();
std::cout << "V1*M1*M3 (by inner product)      = " << resOutput  << std::endl;
```

https://asecuritysite.com/openfhe/openfhe_27cpp

https://asecuritysite.com/openfhe/openfhe_26cpp

Matrix Operations





```
Ciphertext<DCRTPoly> vectorC = cryptoContext->Encrypt(keyPair.publicKey, vectorP);

Ciphertext<DCRTPoly> resC;
Plaintext res;
std::vector<int64_t> resOutput, resOutputtmp;


 resOutput = vectorMatrixMult(vector, matrix1);
resOutput = vectorMatrixMult(resOutput, matrix2);

std::cout << "V1*M1*M2 (non homomorphic)  = " << resOutput << std::endl;

//resC = vectorMatrixMultByInnProdCP(cryptoContext, keyPair.publicKey, vectorC, matrix1);
//resC = vectorMatrixMultByInnProdCP(cryptoContext, keyPair.publicKey, resC, matrix2);
resC = vectorMatrixMultByInnProdFastCP(cryptoContext, keyPair.publicKey, vectorC, matrix1);
 resC = vectorMatrixMultByInnProdFastCP(cryptoContext, keyPair.publicKey, resC, matrix2);

cryptoContext->Decrypt(keyPair.secretKey, resC, &res);
res->SetLength(n3);
resOutput = res->GetPackedValue();
std::cout << "V1*M1*M3 (by inner product)      = " << resOutput  << std::endl;
```

https://asecuritysite.com/openfhe/openfhe_28cpp

https://asecuritysite.com/openfhe/openfhe_29cpp

FHE and Neural Networks

Neuron
Back propagation
Compare
Expected results
x_1
x_2
x_3
w_{11}
w_{21}
w_{31}
w_{12}
w_{22}
w_{32}
w_{13}
w_{23}
w_{33}
y_1
y_2
y_3
Feed-forward
Input vector
Hidden layer
Hidden layer
Output vector

Privacy-aware ML

Prof Bill Buchanan OBE, FRSE

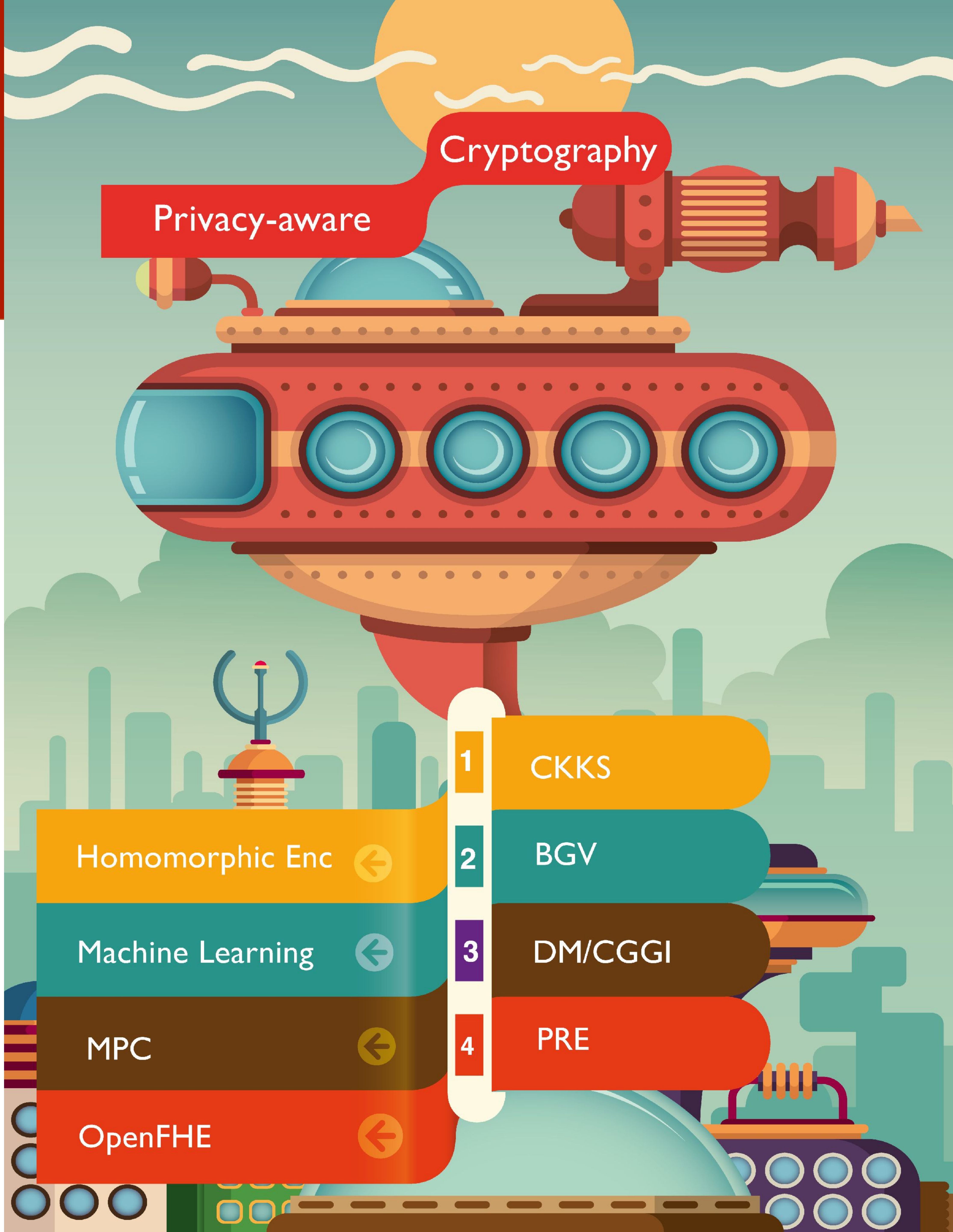
http://asecuritysite.com

Twitter: billatnapier

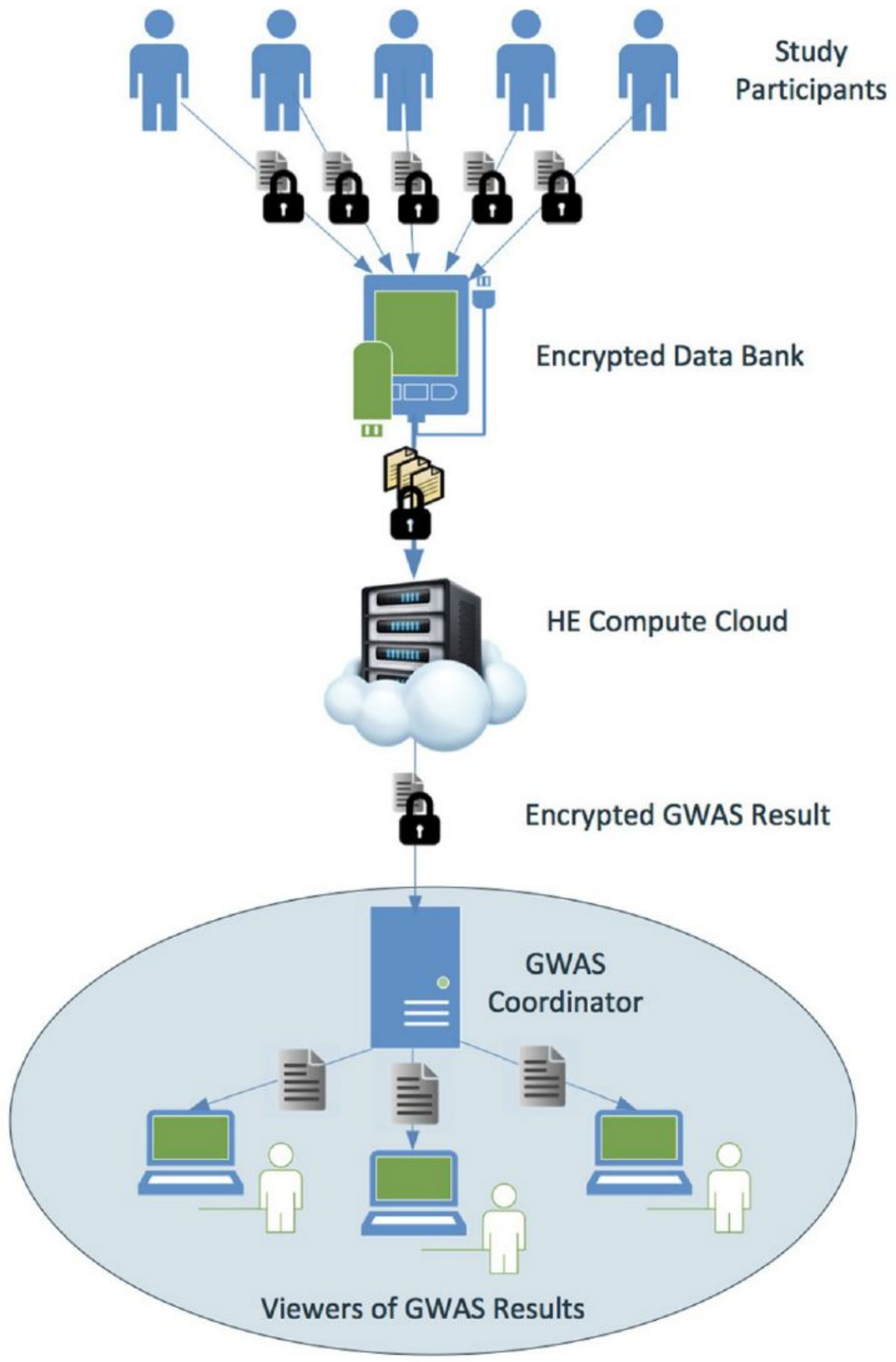
Support Vector Machine (SVM)

BLOCKPASS IDENTITY LAB

World-leading Collaboration between Blockpass IDN and Edinburgh Napier University



GWAS (Gnome-wide Association Studies)



Study Participants

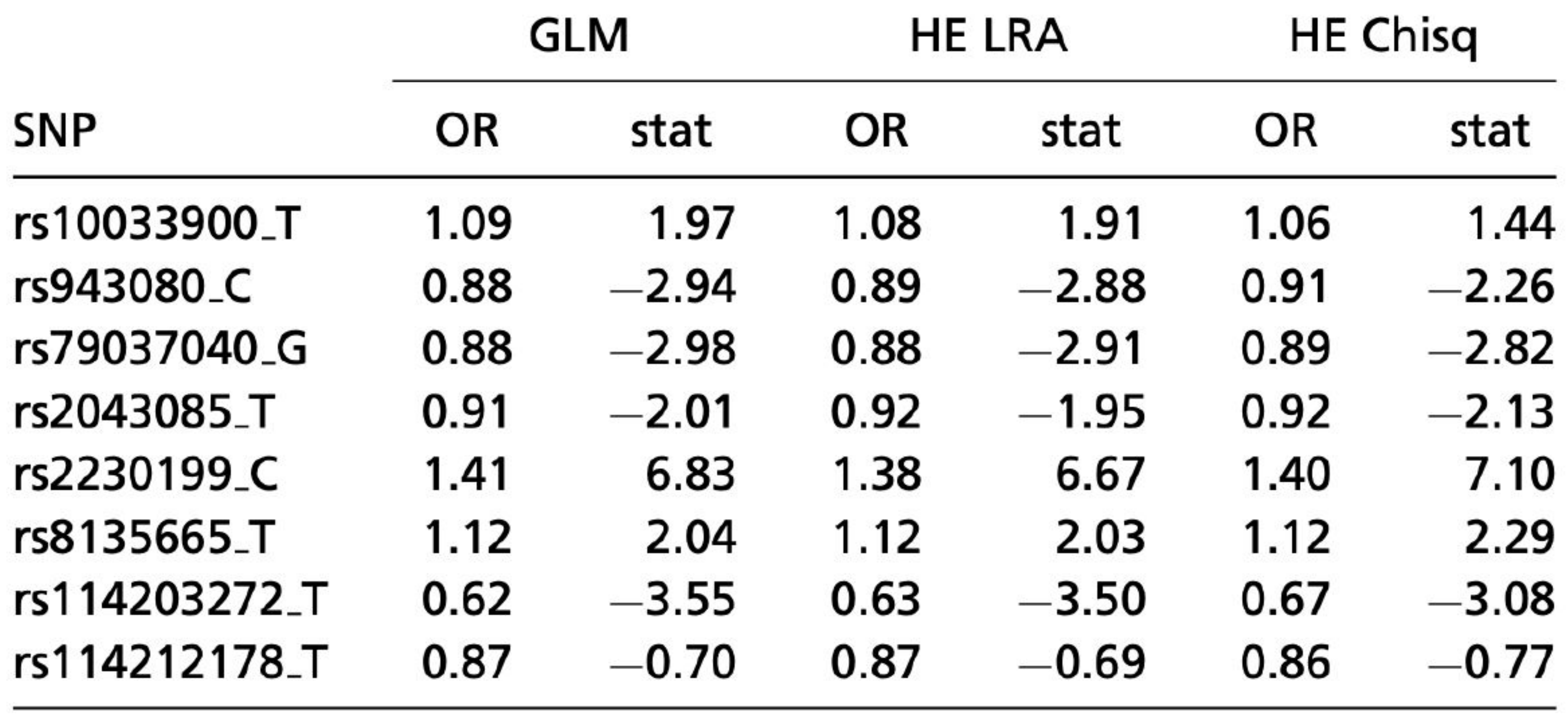
Encrypted Data Bank

HE Compute Cloud

Encrypted GWAS Result

GWAS Coordinator

Viewers of GWAS Results



	GLM		HE LRA		HE Chisq	
SNP	OR	stat	OR	stat	OR	stat
rs10033900_T	1.09	1.97	1.08	1.91	1.06	1.44
rs943080_C	0.88	−2.94	0.89	−2.88	0.91	−2.26
rs79037040_G	0.88	−2.98	0.88	−2.91	0.89	−2.82
rs2043085_T	0.91	−2.01	0.92	−1.95	0.92	−2.13
rs2230199_C	1.41	6.83	1.38	6.67	1.40	7.10
rs8135665_T	1.12	2.04	1.12	2.03	1.12	2.29
rs114203272_T	0.62	−3.55	0.63	−3.50	0.67	−3.08
rs114212178_T	0.87	−0.70	0.87	−0.69	0.86	−0.77

Figure 7.8: Results for GWAS [92]

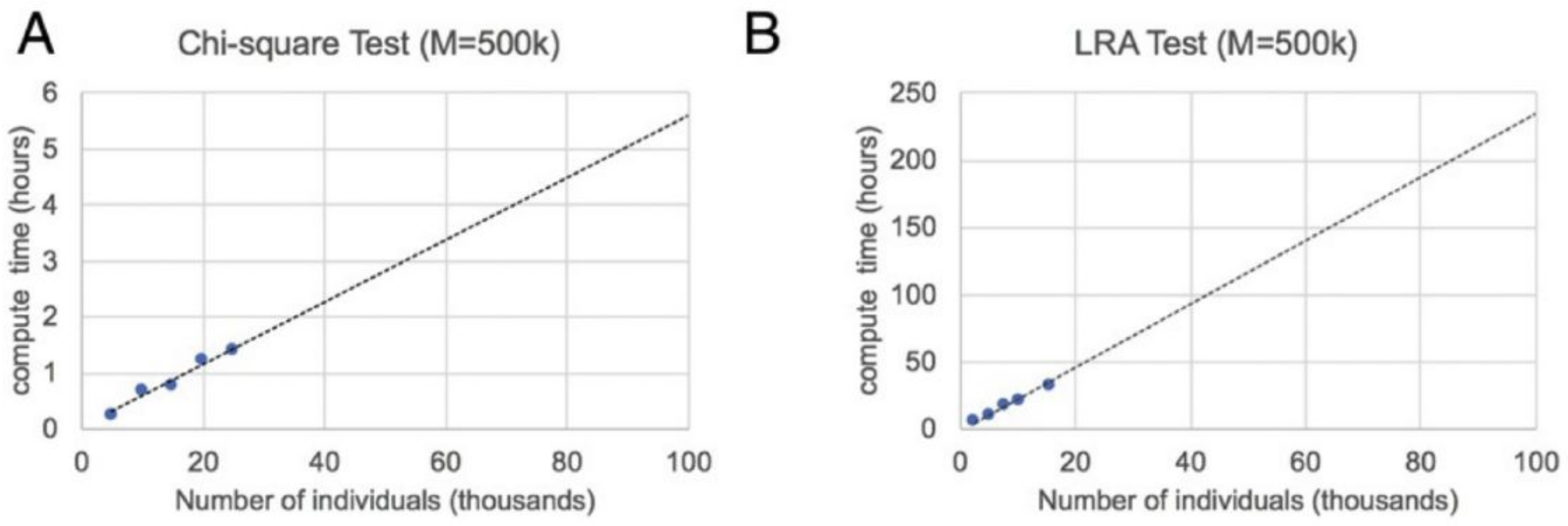
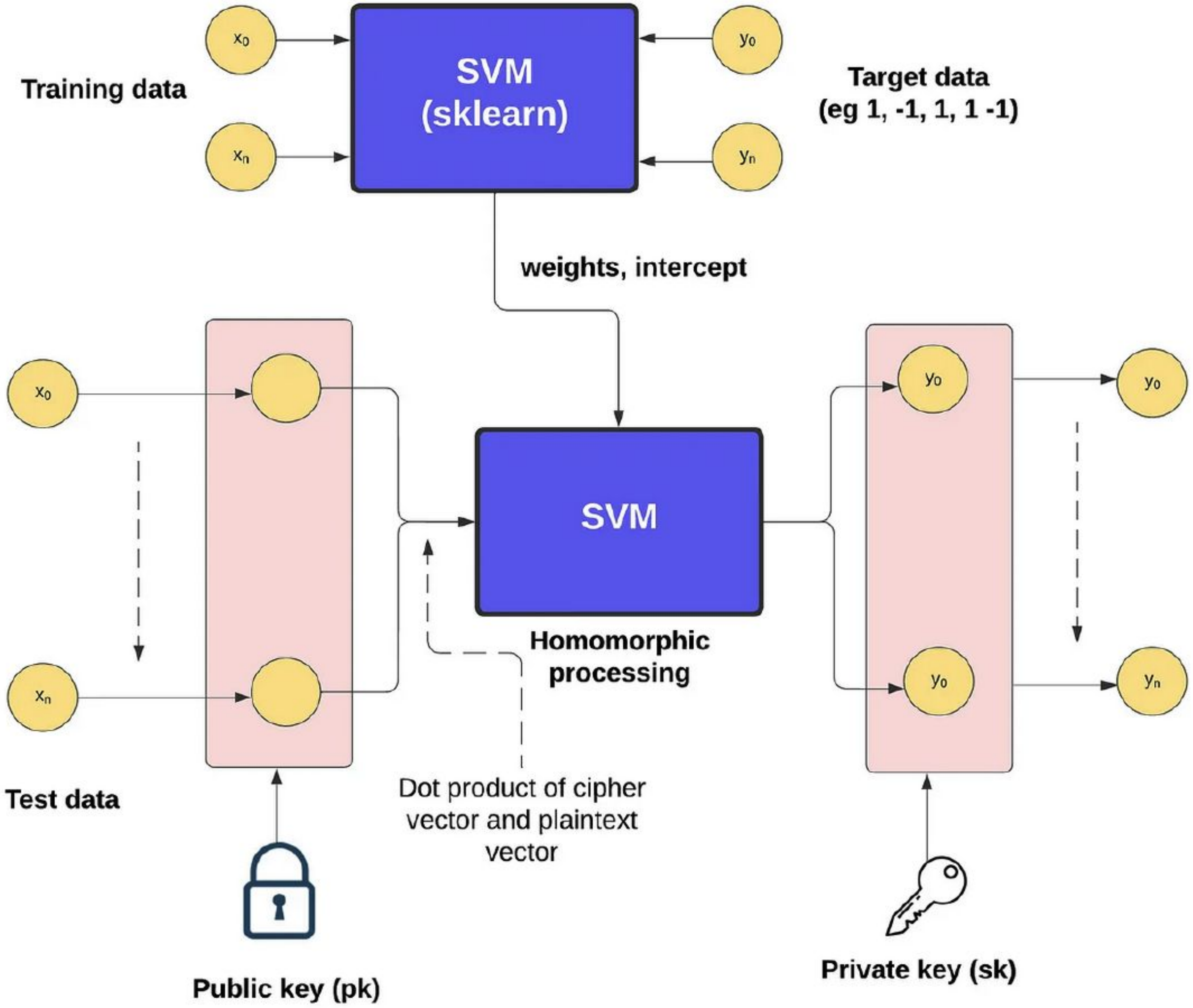
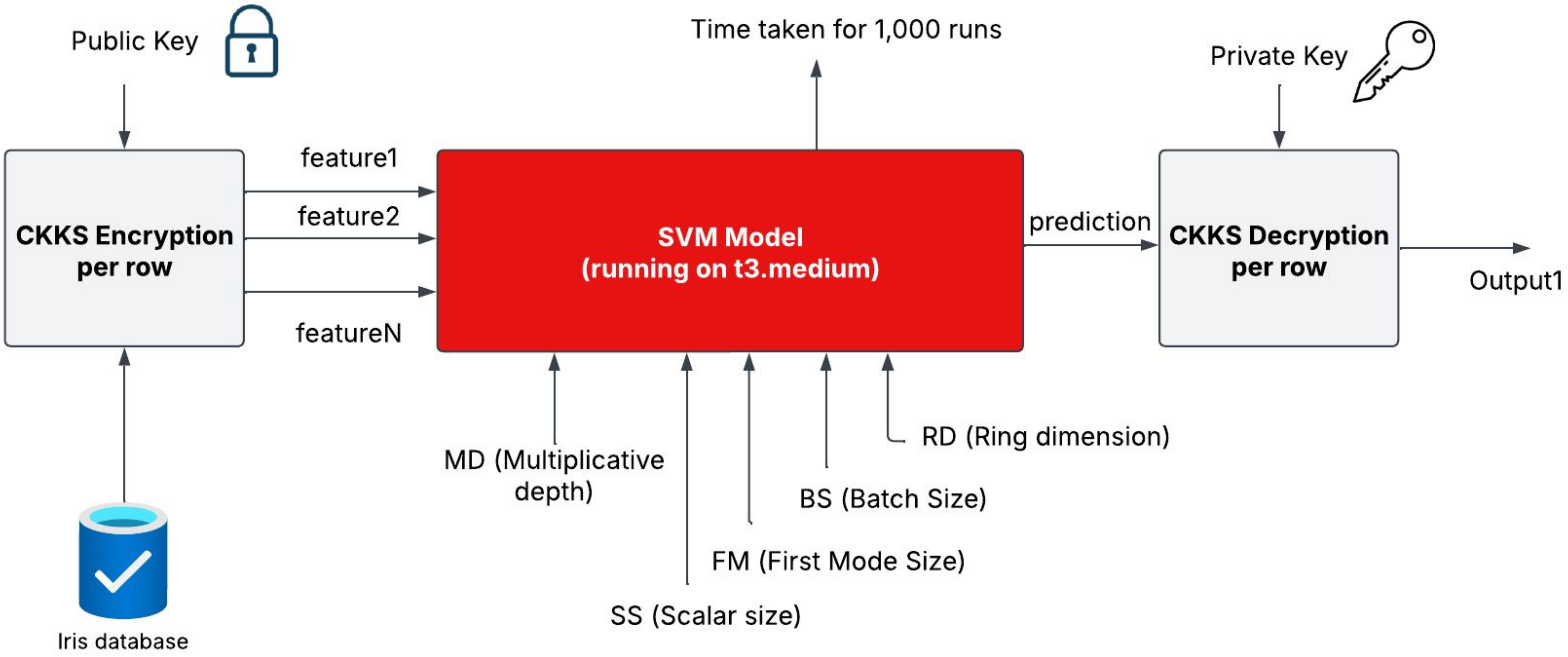


Figure 7.9: Results for GWAS [92]

SVM


Training data
x_0
x_n
SVM (sklearn)
y_0
y_n
Target data (eg 1, -1, 1, 1 -1)
weights, intercept
x_0
x_n
Test data
SVM
Homomorphic processing
Dot product of cipher vector and plaintext vector
Public key (pk)
y_0
y_n
y_0
y_n
Private key (sk)

Results


Public Key

CKKS Encryption per row

feature1

feature2

featureN

Iris database

SVM Model (running on t3.medium)

Time taken for 1,000 runs

MD (Multiplicative depth)

SS (Scalar size)

FM (First Mode Size)

BS (Batch Size)

RD (Ring dimension)

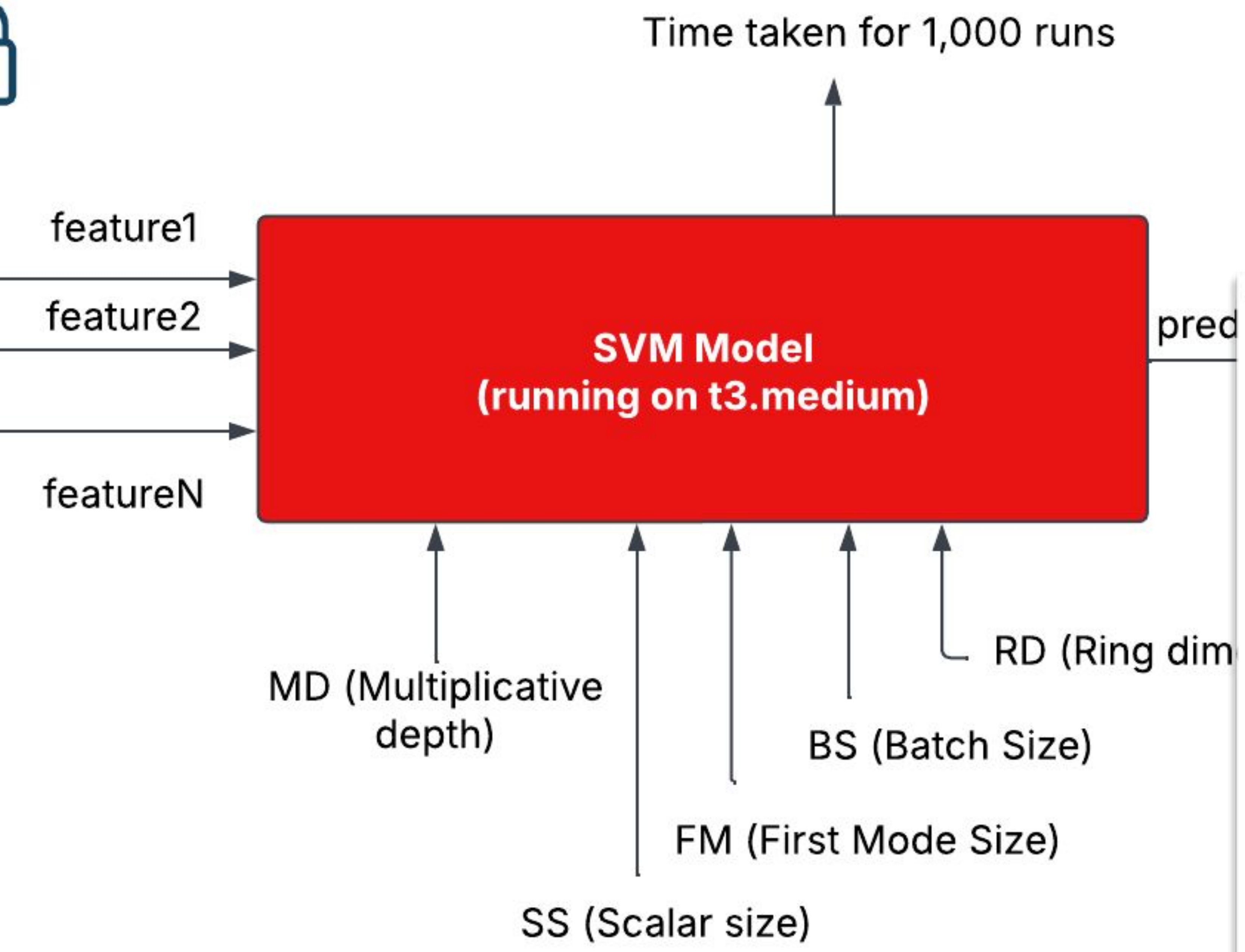
prediction

Private Key

CKKS Decryption per row

Output1

Results



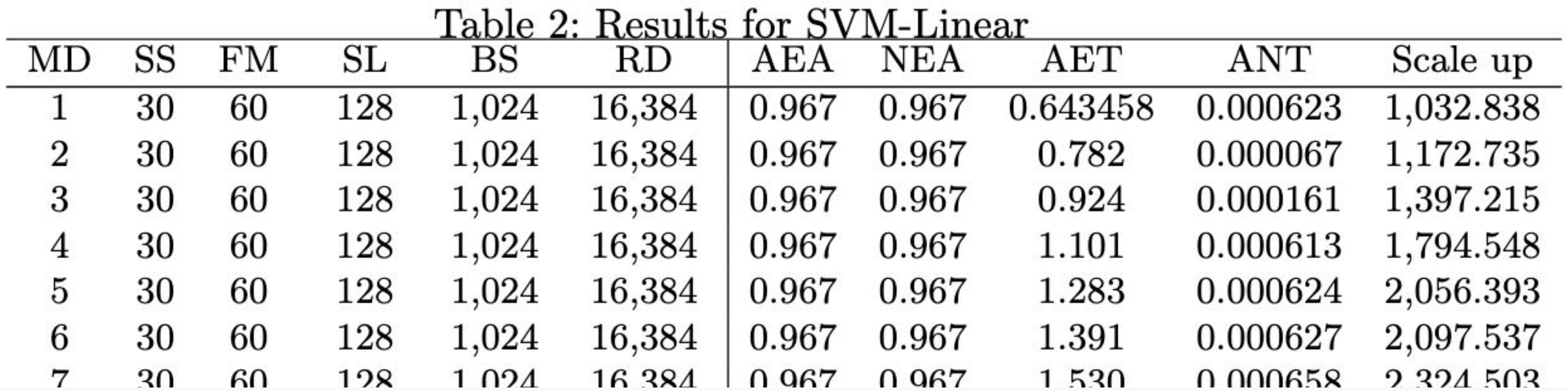


Table 2: Results for SVM-Linear

MD	SS	FM	SL	BS	RD	AEA	NEA	AET	ANT	Scale up
1	30	60	128	1,024	16,384	0.967	0.967	0.643458	0.000623	1,032.838
2	30	60	128	1,024	16,384	0.967	0.967	0.782	0.000067	1,172.735
3	30	60	128	1,024	16,384	0.967	0.967	0.924	0.000161	1,397.215
4	30	60	128	1,024	16,384	0.967	0.967	1.101	0.000613	1,794.548
5	30	60	128	1,024	16,384	0.967	0.967	1.283	0.000624	2,056.393
6	30	60	128	1,024	16,384	0.967	0.967	1.391	0.000627	2,097.537
7	30	60	128	1,024	16,384	0.967	0.967	1.530	0.000658	2,324.503

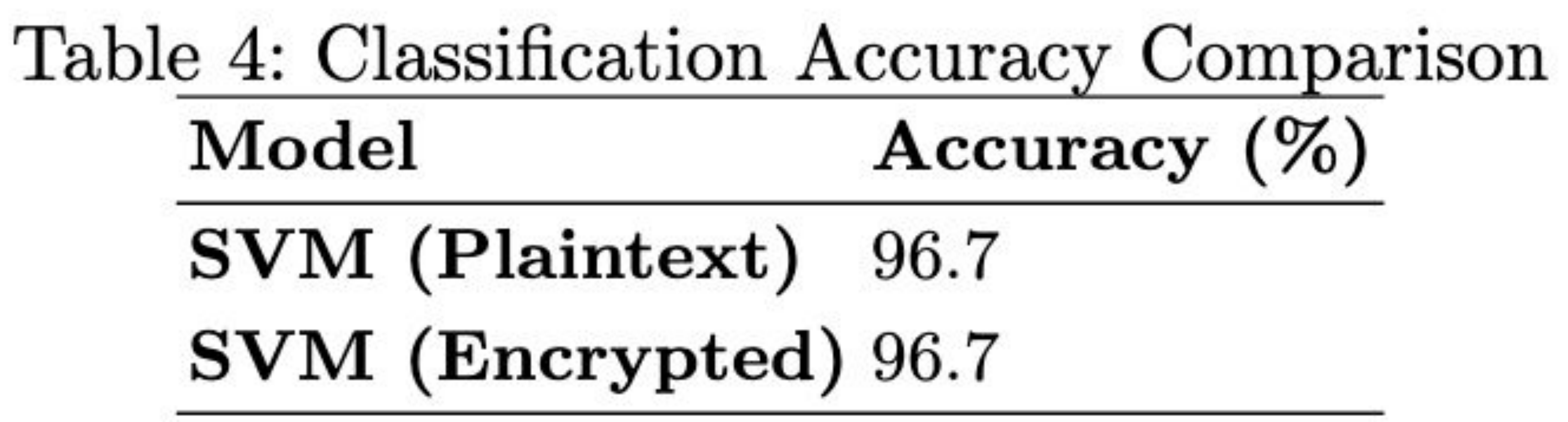
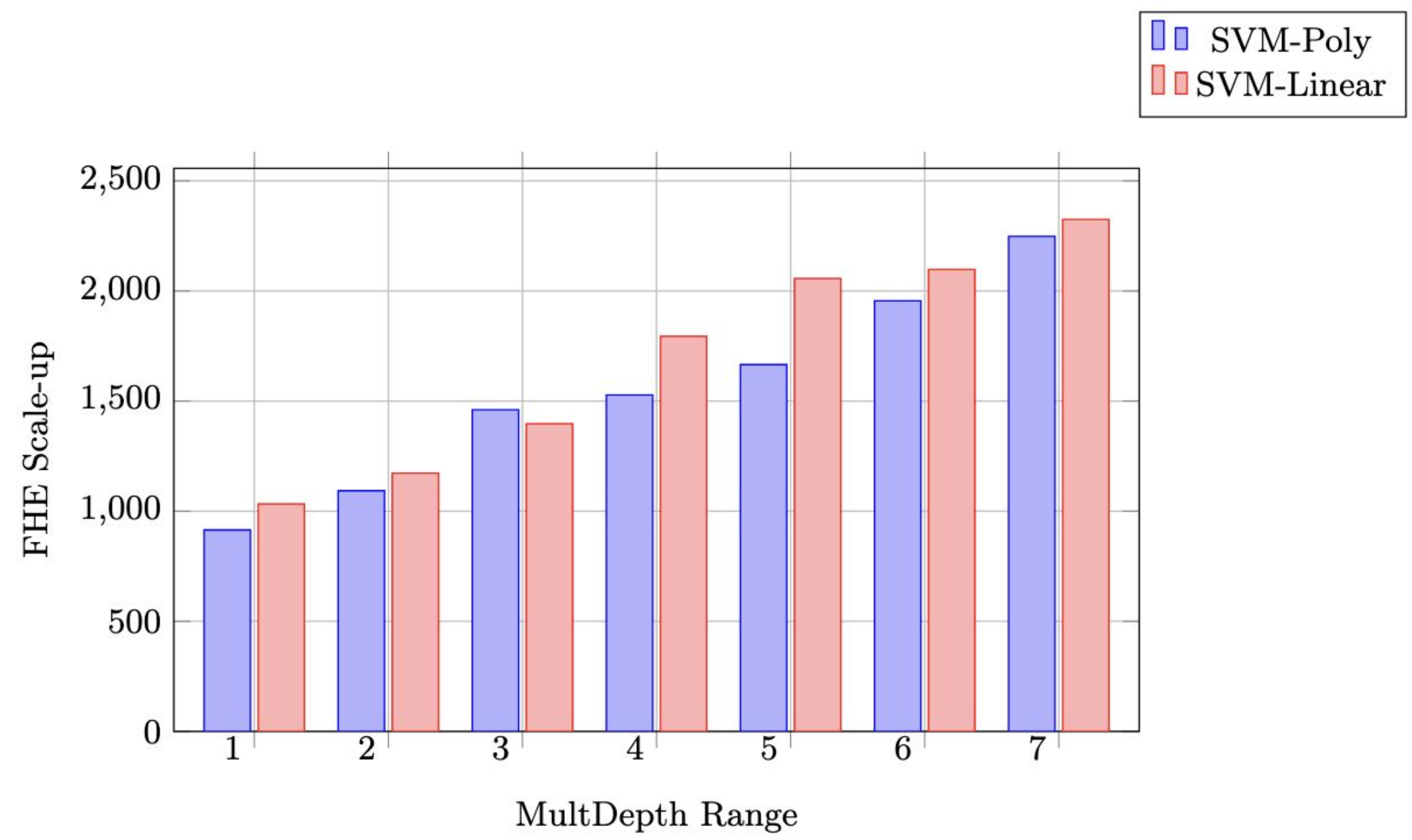


Table 4: Classification Accuracy Comparison

Model	Accuracy (%)
SVM (Plaintext)	96.7
SVM (Encrypted)	96.7



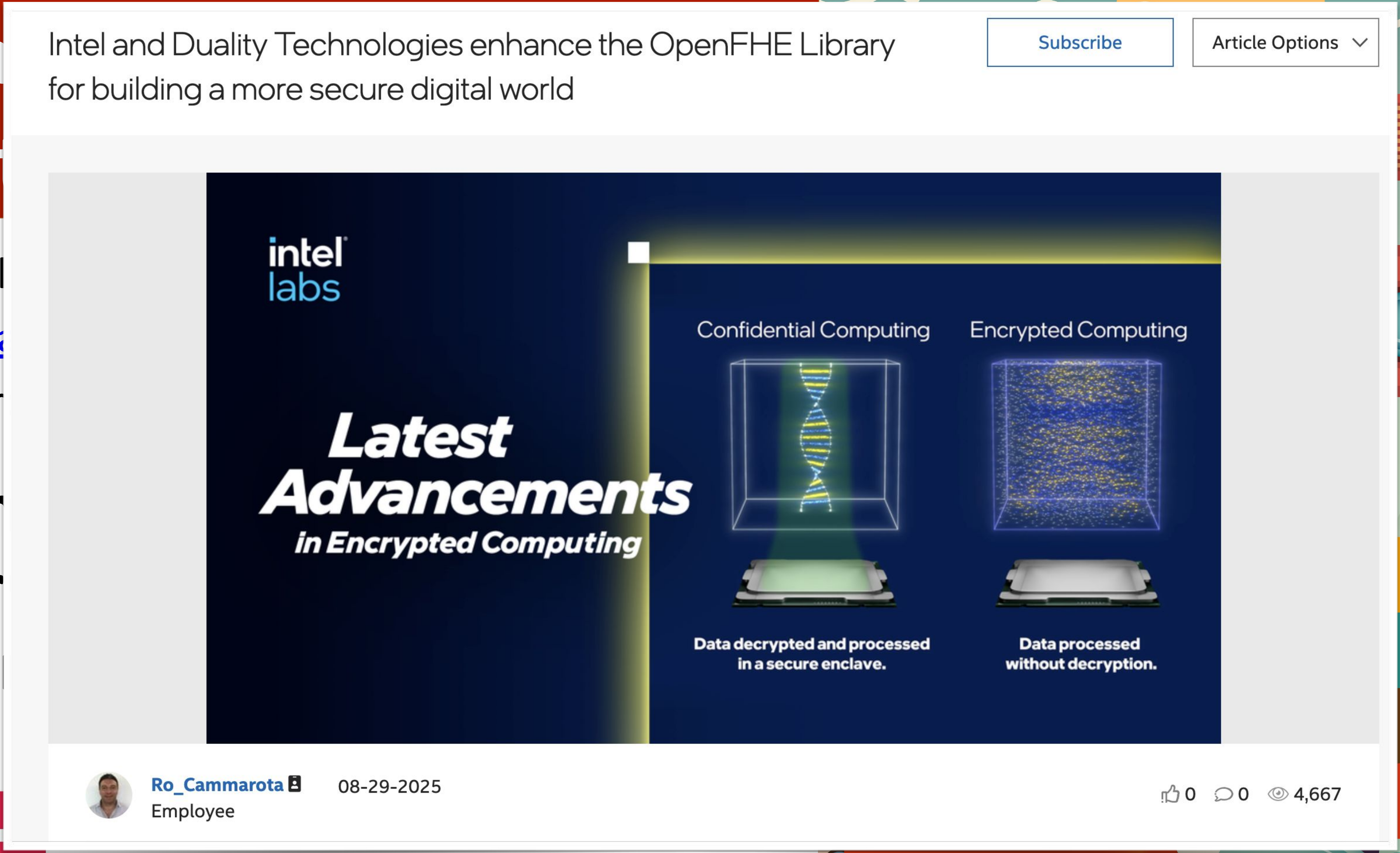
Priva
Cryp

Prof Bil

http://a

Twitter

Hom
Encr
Ope



Intel and Duality Technologies enhance the OpenFHE Library for building a more secure digital world

Subscribe
Article Options
intel labs
Latest Advancements in Encrypted Computing
Confidential Computing
Encrypted Computing
Data decrypted and processed in a secure enclave.
Data processed without decryption.

Ro_Cammarota
Employee

08-29-2025

LAB

World-leading Collaboration between
Blockpass IDN and Edinburgh Napier University