Workshop 4: Future of Digital Trust

Prof Bill Buchanan OBE FRSE

https://asecuritysite.com

Zero Knowledge Proofs
Fiat Shamir, DLEQ, zkSnarks,
Ring Signatures, Range Proofs, Accumulators,
Pedersen Commitments
Homomorphic Encryption:
Basics
Concepts and Code
Data Sharing
Cloud Processing
MPC
Machine Learning

BLOCKPASS IDENTITY LAB
World-leading Collaboration between
Blockpass IDN and Edinburgh Napier University

Deep Tech Workshops

- Workshop 1 (Wednesday, 26 March 2025, 10am CET). **Cryptography Fundamentals** here.
- Workshop 2 (Wednesday, 16 April 2025, 10am CET). **Blockchain and Cryptocurrency** here.
- Workshop 3 (Wednesday, 7 May 2025, 10am CET). **Cyber, AI and Post Quantum** here.
- Workshop 4 (Wednesday, 4 June 2025, 10am CET). **Homomorphic Encryption and Zero Knowledge Proofs** here.

- Moodle Page: here and GitHub: here.

Workshop 4

10-11am (CET) Zero Knowledge Proofs and Fully Homomorphic Encryption

The topic covered is Zero Knowledge Proofs and Fully Homomorphic Encryption. The presentation on ZKPs and Fully Homomorphic Encryption is here.

11-12pm (CET) Guest speakers

- Kazım Rıfat Özyılmaz - Co-founder of Arf here (Bill).
- Ibrahim Kushchu CEO, TheNextMinds UK here (Gonca).
- Aditya Shetty - Lead of Superteam India here (Aysegul).

12-1pm (CET) Break.

1-2pm (CET) ZKP/FHE Demos and Quiz

This will provide some demos and a quiz.

2-3pm (CET) Guest speaker

Madjid Golparvaran Tehrani. PQC and Quantum Computing.

3-4pm (CET) World-leaders in Cryptography: Chris Peikert, University of Michigan.

Cryptography Overview

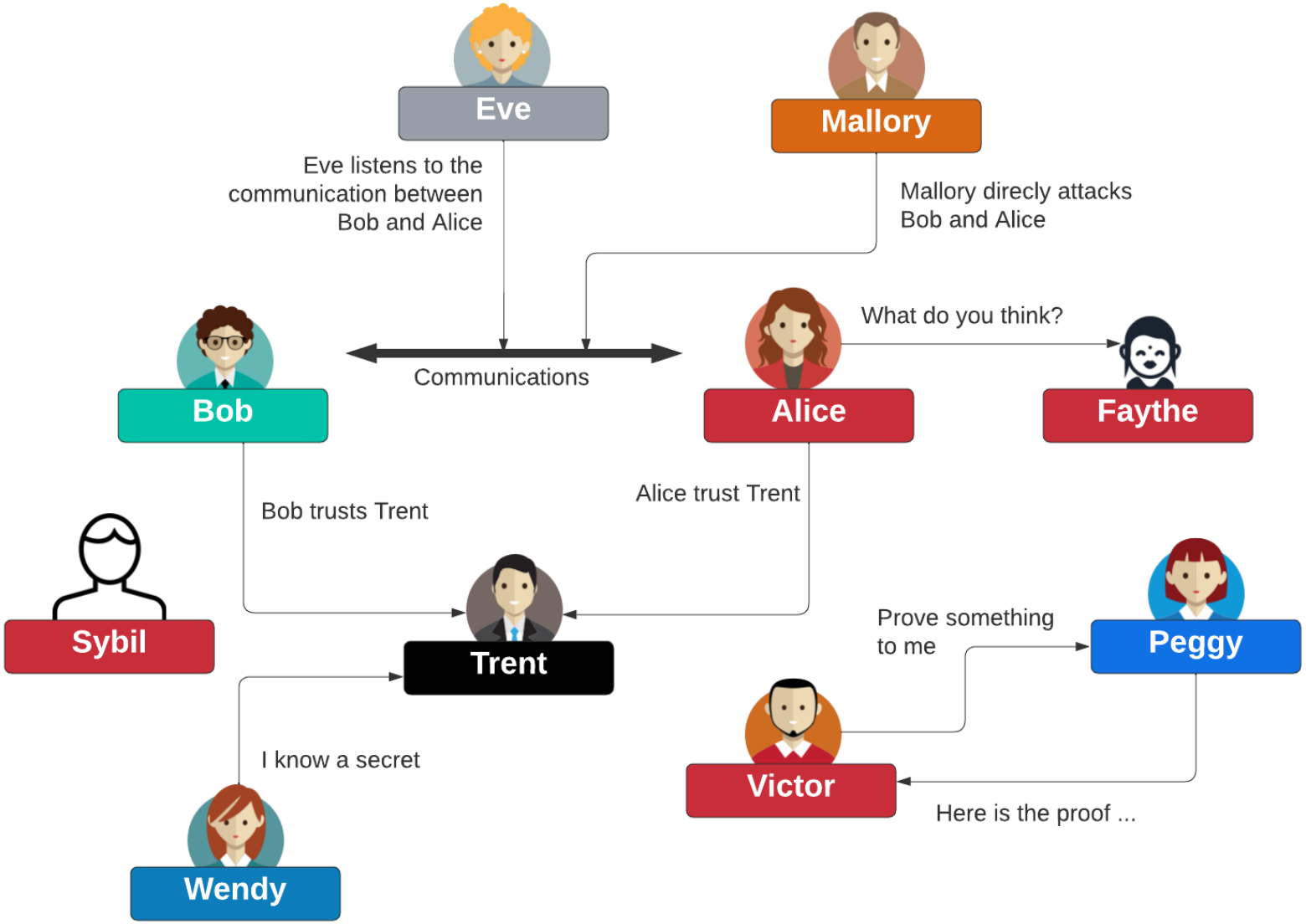
Prof Bill Buchanan OBE FRSE

https://asecuritysite.com

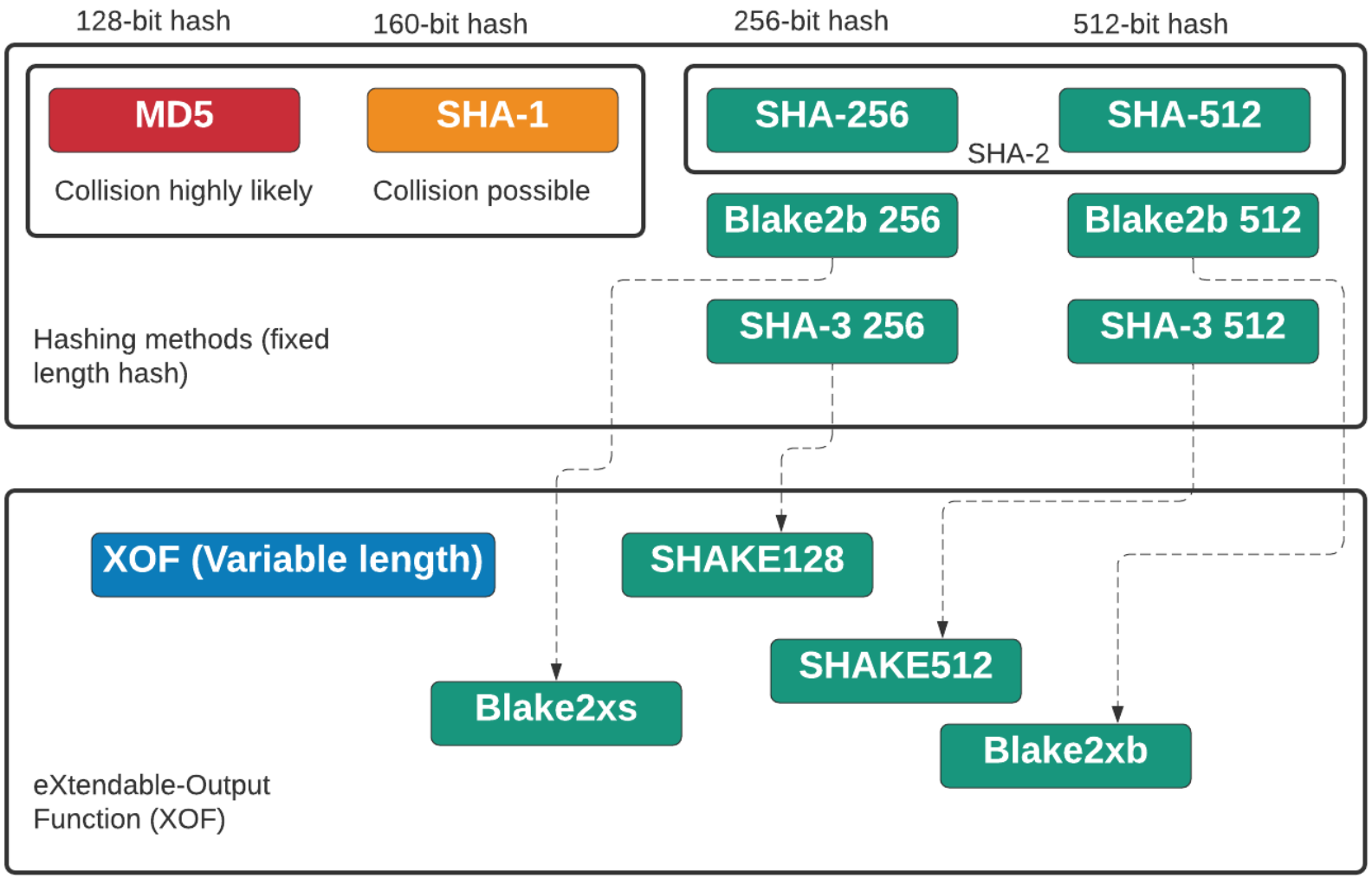
BLOCKPASS IDENTITY LAB

World-leading Collaboration between
Blockpass IDN and Edinburgh Napier University

Some Actors

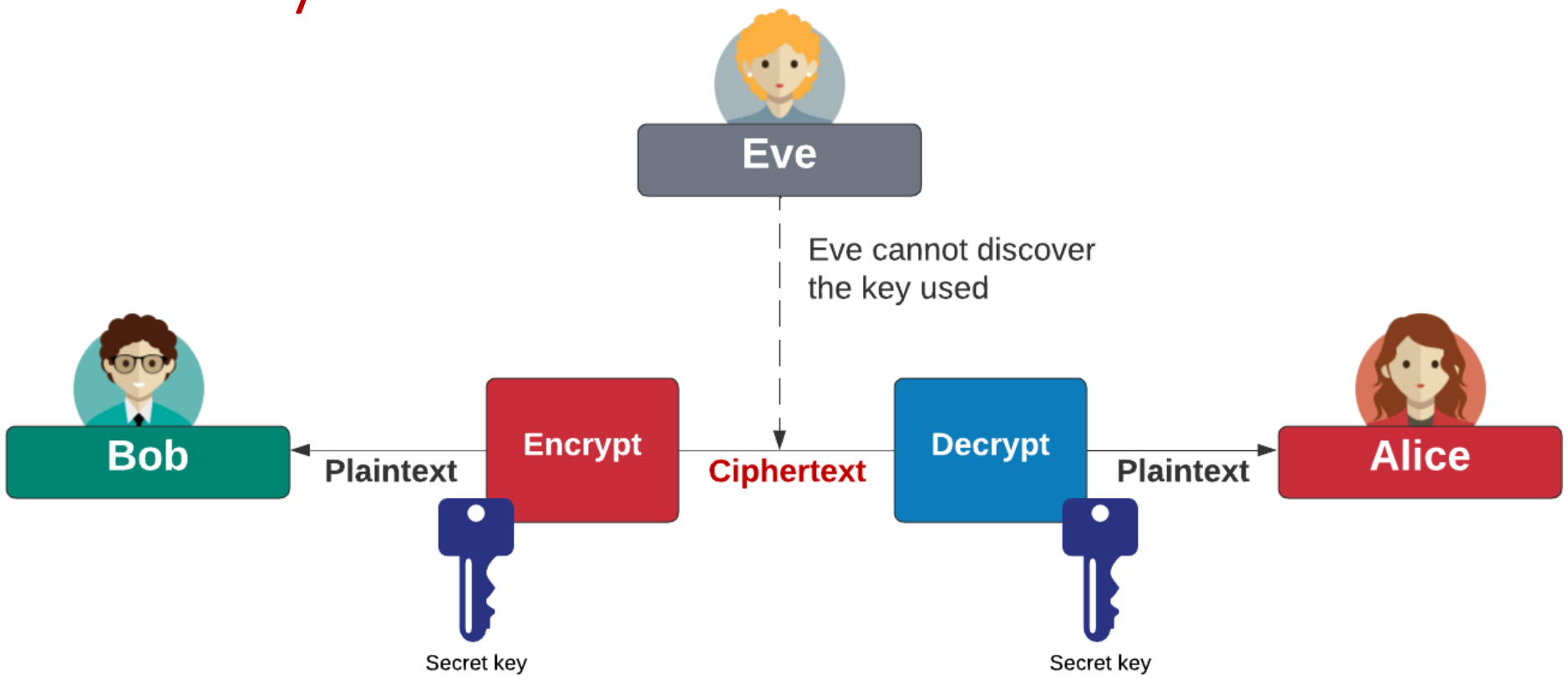

Eve
Mallory
Eve listens to the communication between Bob and Alice
Mallory direcly attacks Bob and Alice
Bob
Communications
Alice
What do you think?
Faythe
Bob trusts Trent
Alice trust Trent
Sybil
Trent
Prove something to me
Peggy
I know a secret
Victor
Here is the proof ...
Wendy

MD5, SHA-1, SHA-2, SHA-3 and XOF



https://asecuritysite.com/hash/cs

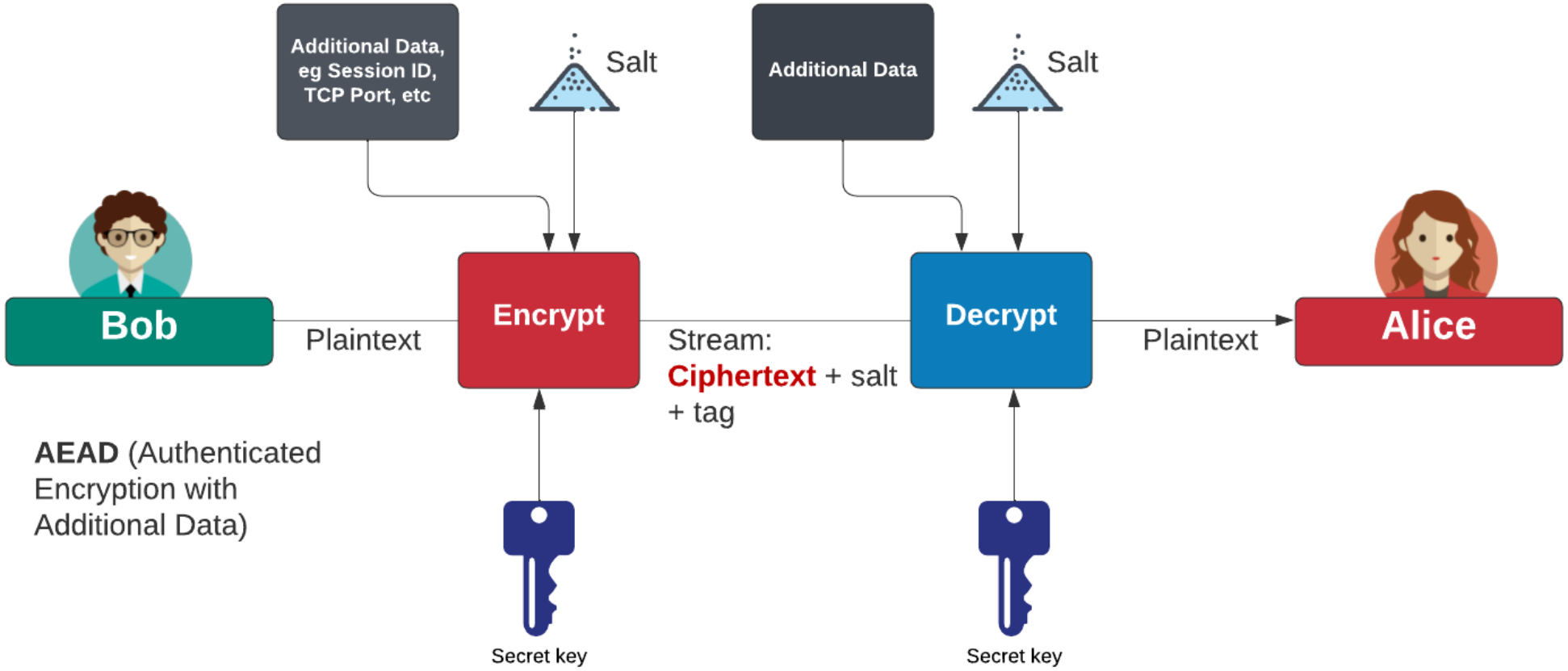
Symmetric Key



Block cipher: **AES (128-bit block)**, DES (64-bit block), 3DES (64-bit block), Blowfish, SM4, Camellia, GOST, IDEA, RC2, Serpent, Skipjack, Twofish, and XTEA
Stream cipher: **ChaCha20**, **AES GCM**, ARIA, Sala20, and RC4.

AES also known as Rijndael (after Joan Daemen and Vincent Rijmen).
ChaCha20 created by Daniel J Bernstein.
Blowfish and Twofish created by Bruce Schneier.

Authenticated Encryption with Additional Data (AEAD)

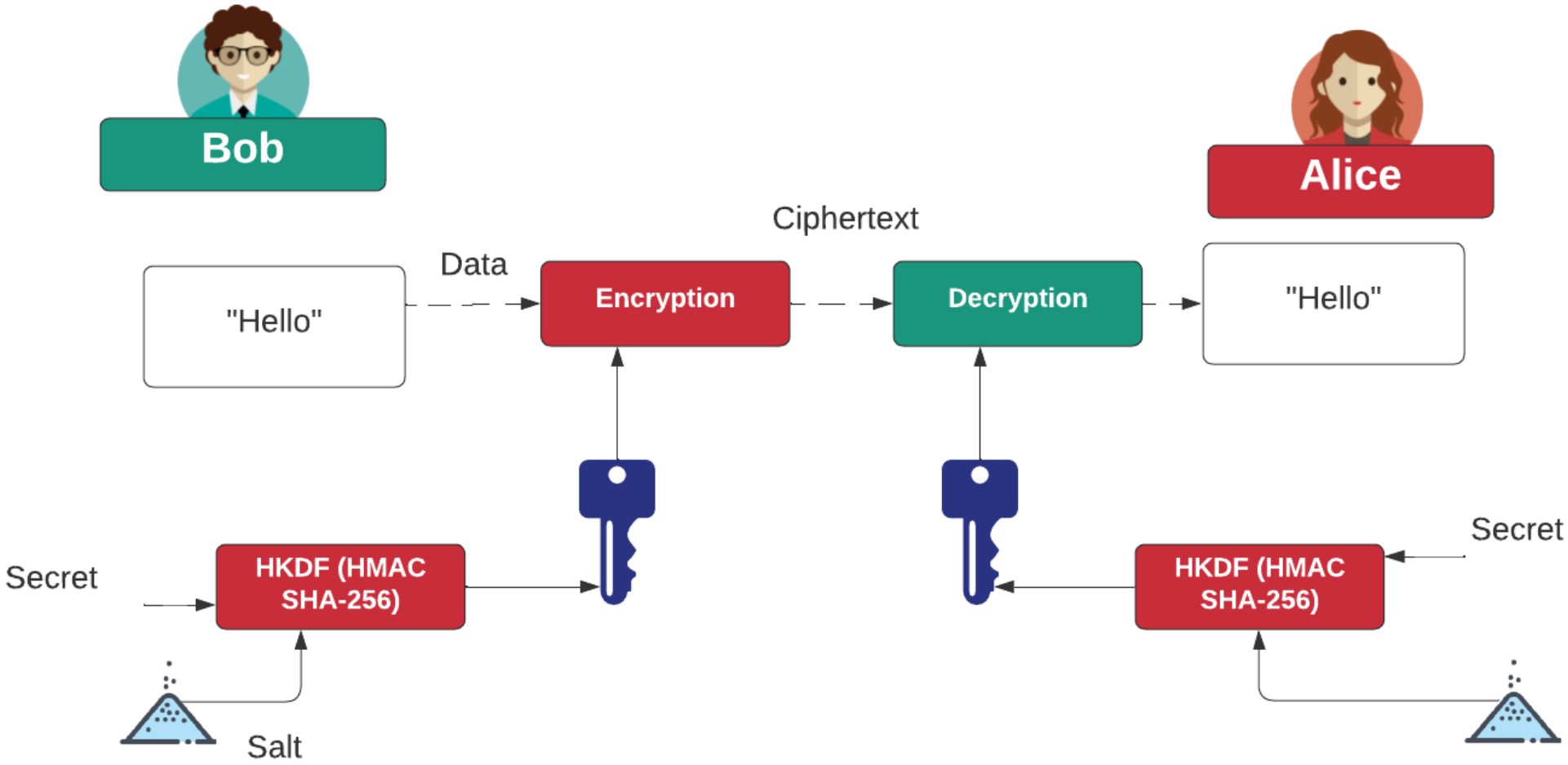


AES Block cipher modes: ECB (No salt), CBC (Cipher Block Chaining).

AES Stream modes: CFB (Cipher Feedback), OFB (Output Feedback), CTR (Counter), GCM (Galois Counter Mode).

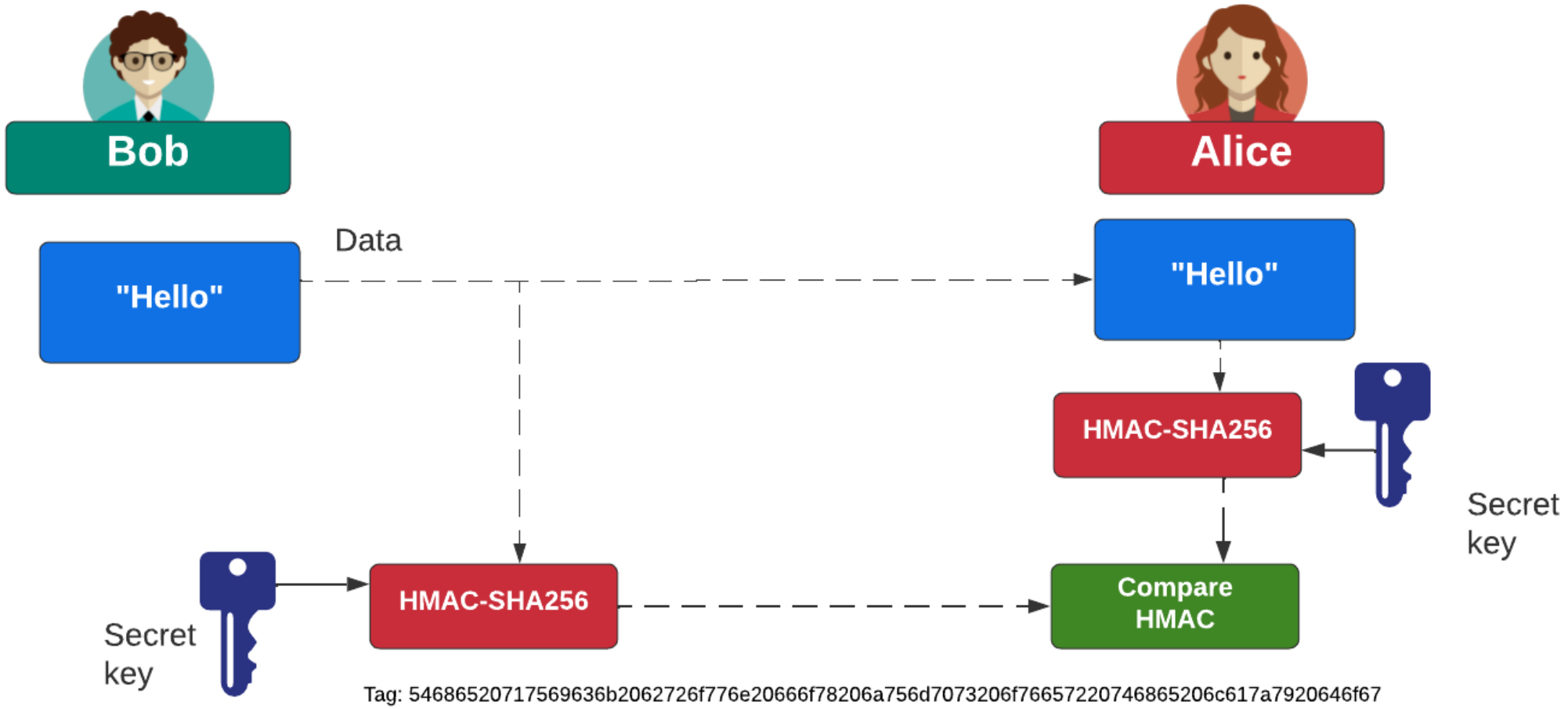
https://asecuritysite.com/golang/go_symmetric

HKDF (HMAC Key Derivation Function)



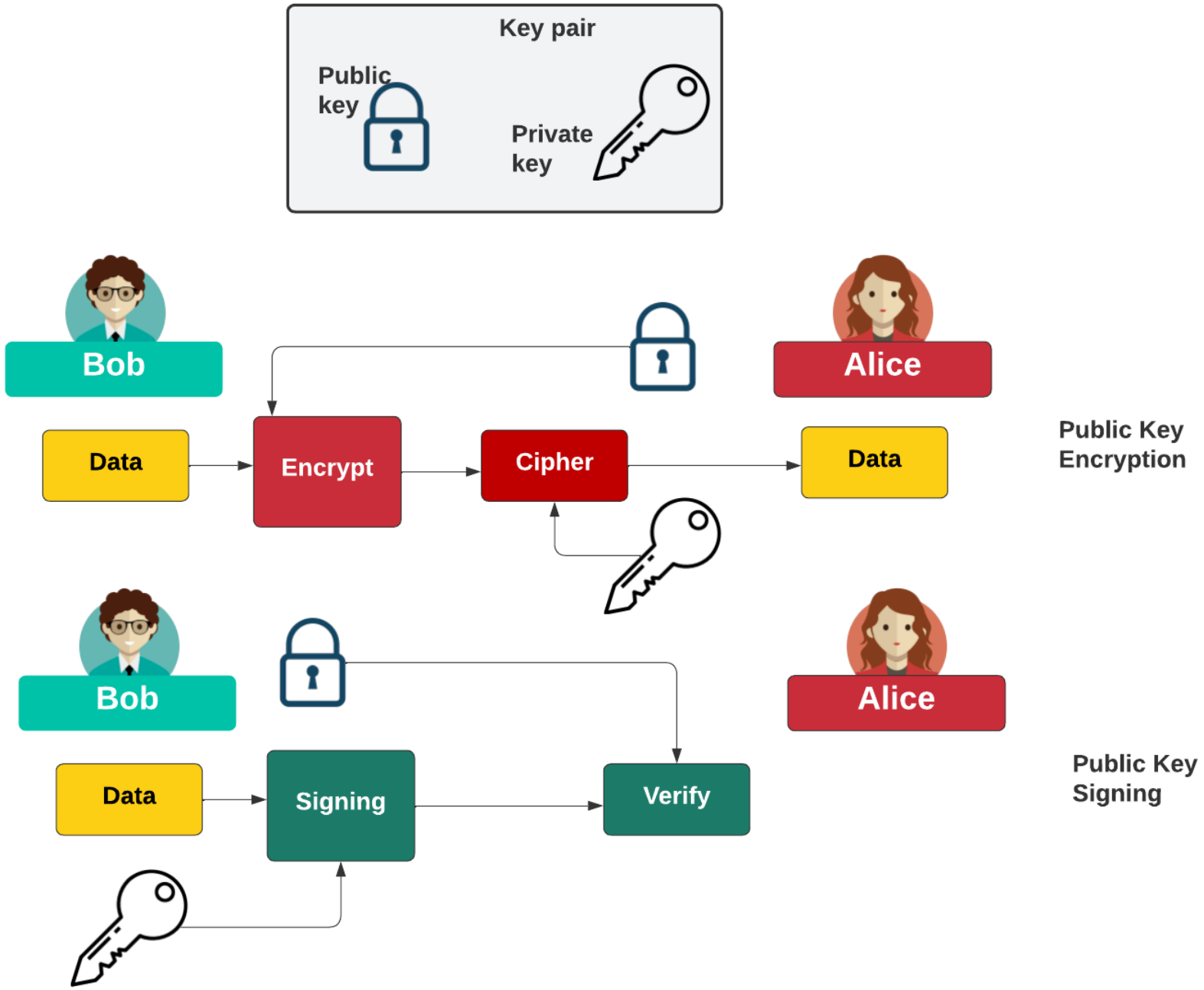
https://asecuritysite.com/pbkdf2/hashnew3

HMAC (Key Hash Message Authentication Code)

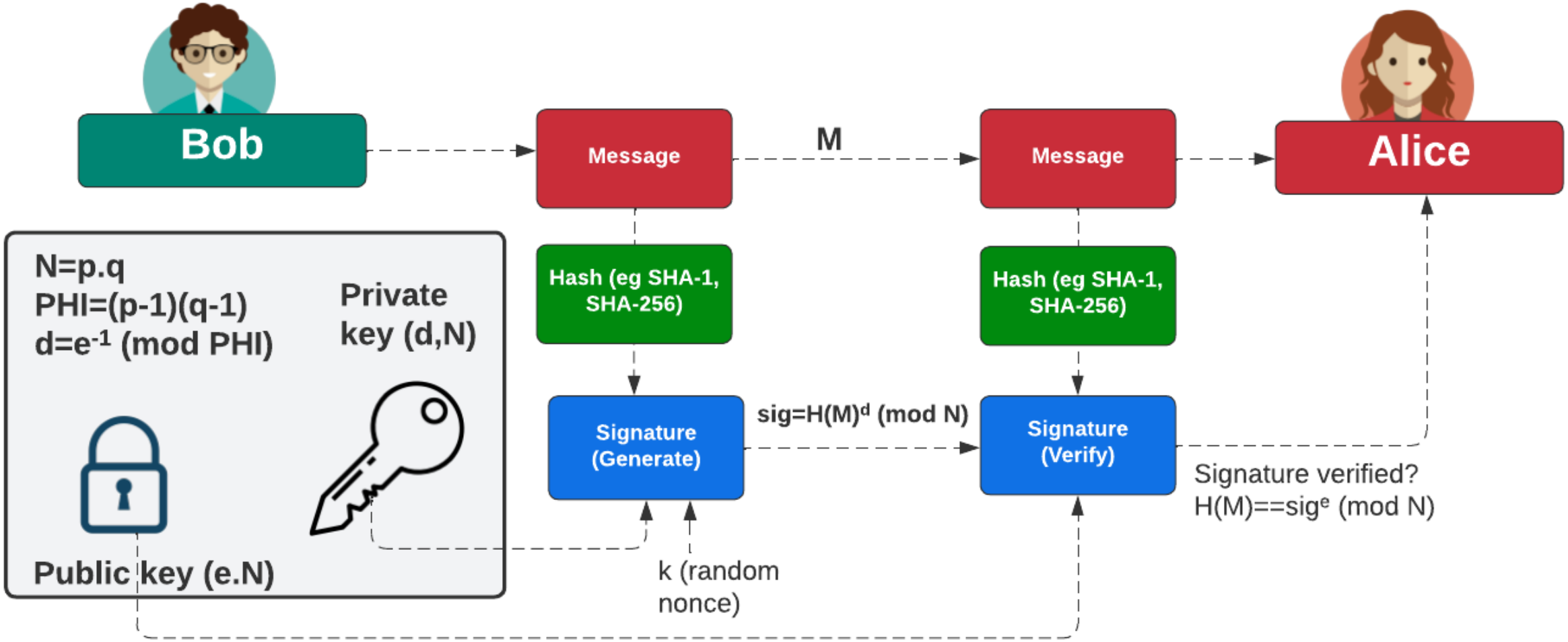


https://asecuritysite.com/mac/hashhew2

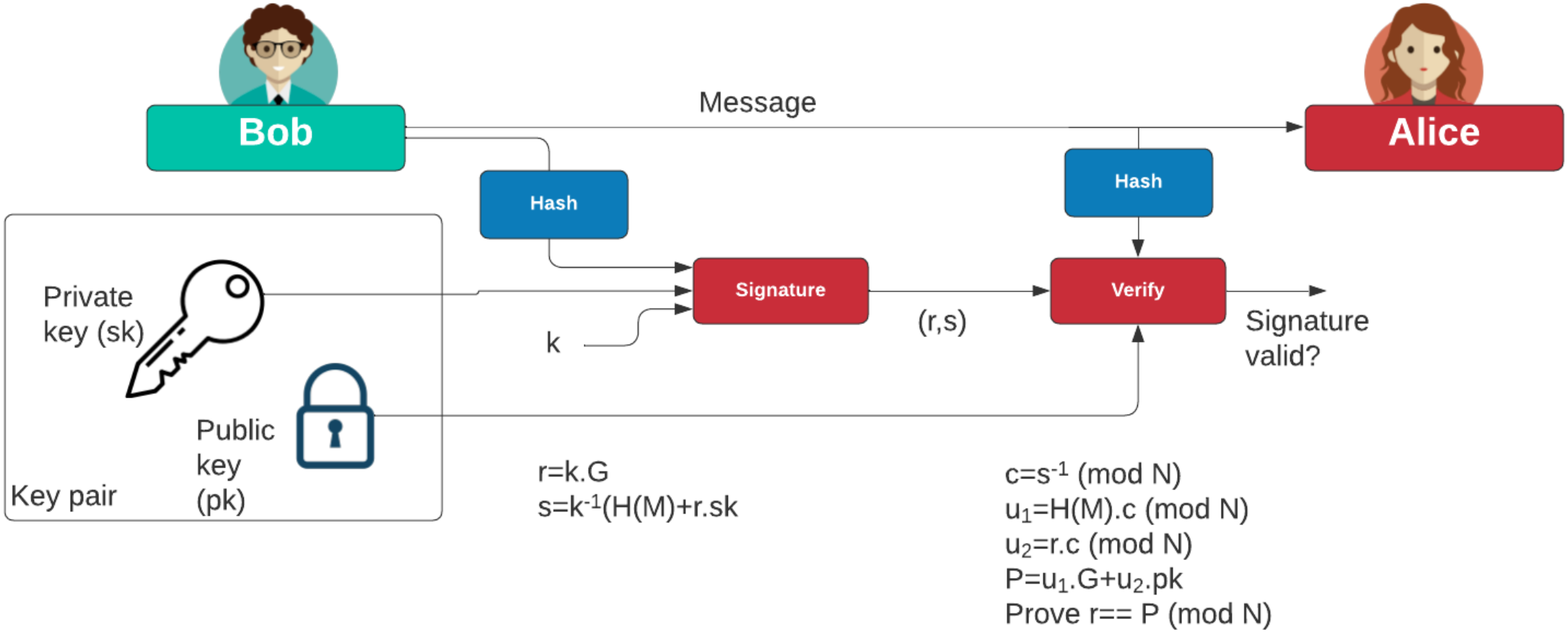
Public Key


Key pair
Public key
Private key
Bob
Alice
Data
Encrypt
Cipher
Data
Public Key Encryption
Bob
Alice
Data
Signing
Verify
Public Key Signing

Digital Signatures



ECDSA



https://asecuritysite.com/ecdsa/ecdsa3

Zero Knowledge Proofs

Prof Bill Buchanan OBE FRSE

https://asecuritysite.com/zero

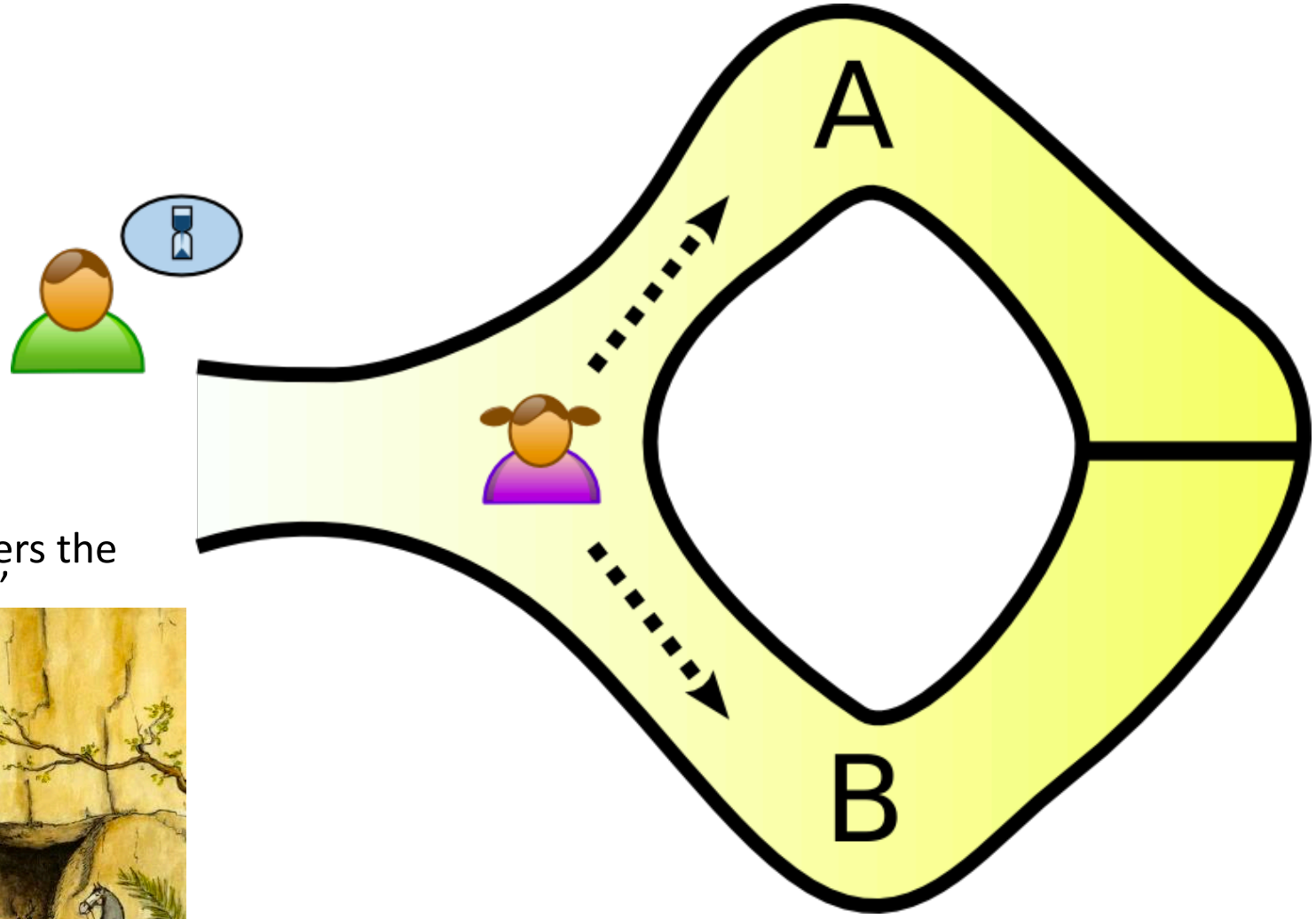
https://asecuritysite.com/dleq

BLOCKPASS IDENTITY LAB

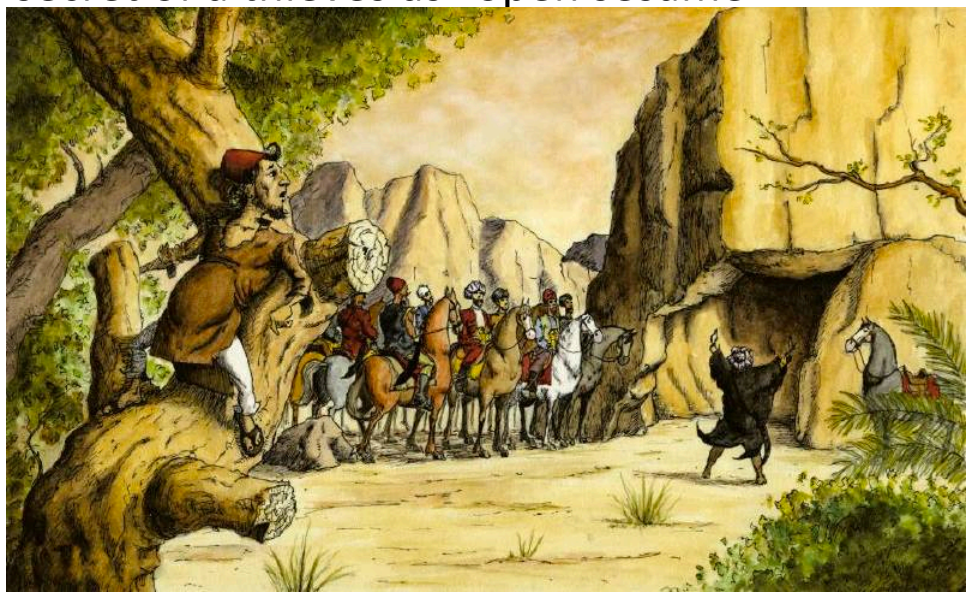
World-leading Collaboration between
Blockpass IDN and Edinburgh Napier University

Zero-knowledge Proof

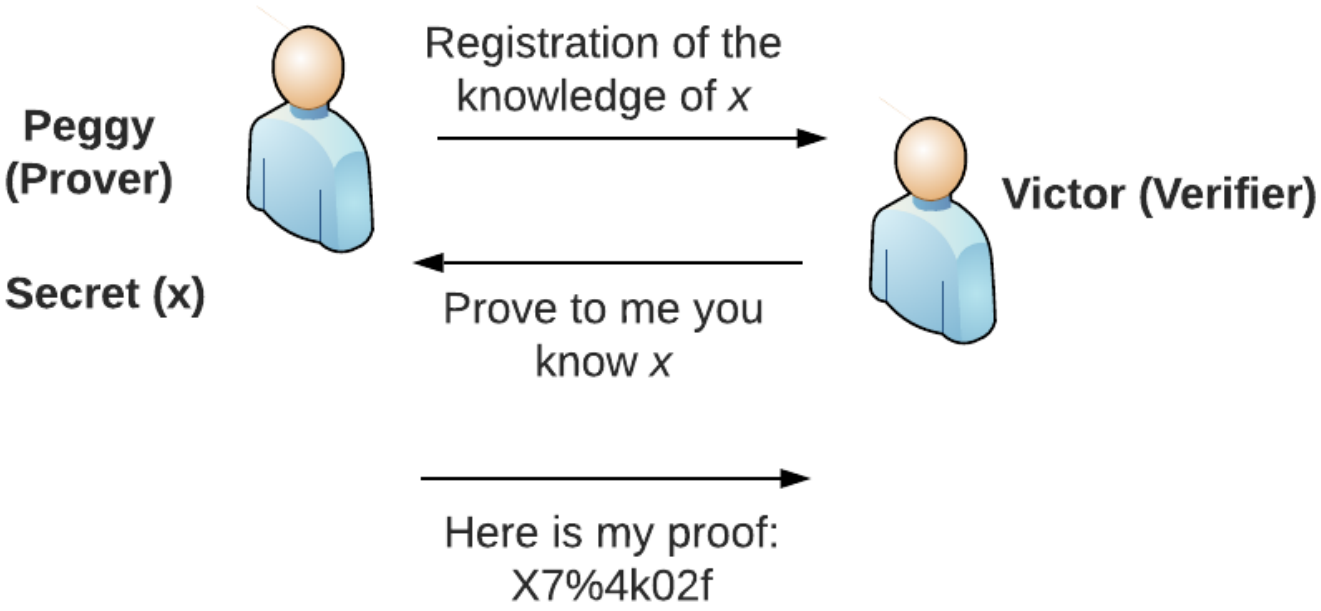
- Peggy is the prover.
- Victor is the verifier.



Ali Baba - poor woodcutter - discovers the secret of a thieves as “open sesame”



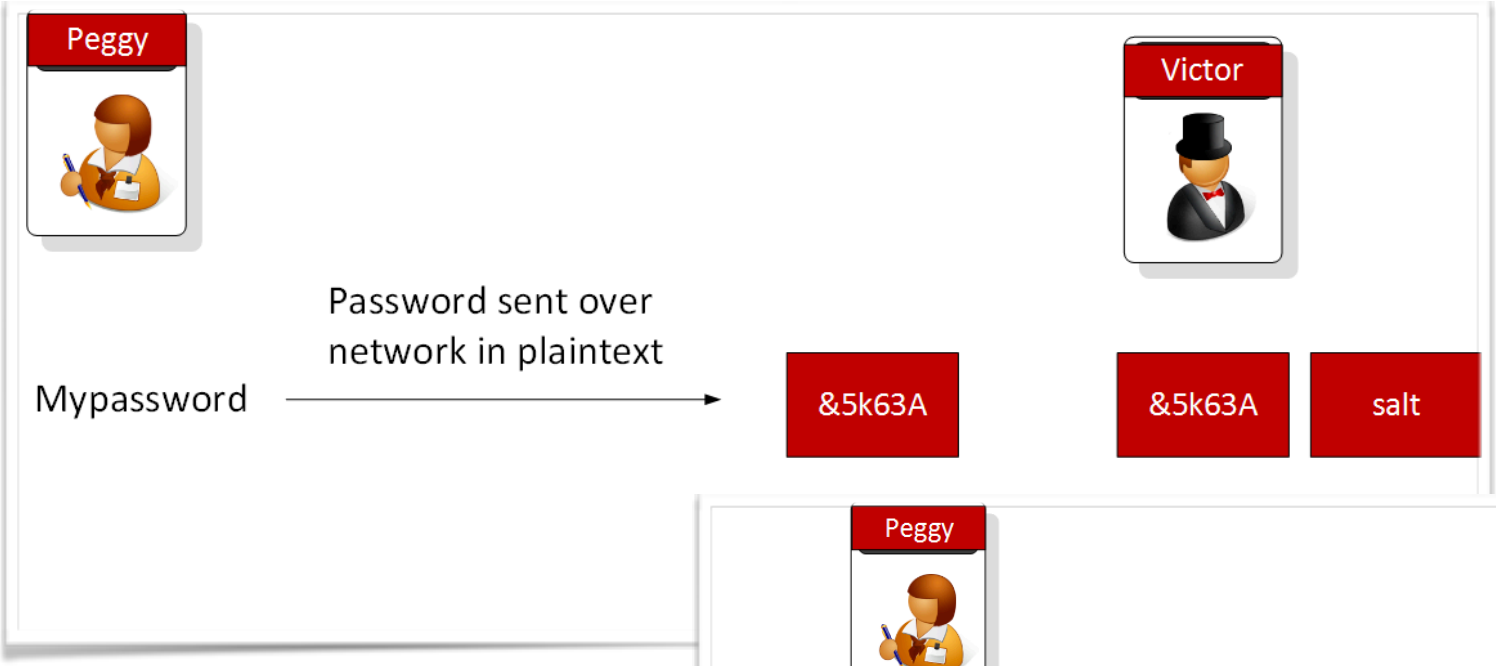
Interactive Zero-Knowledge Proof

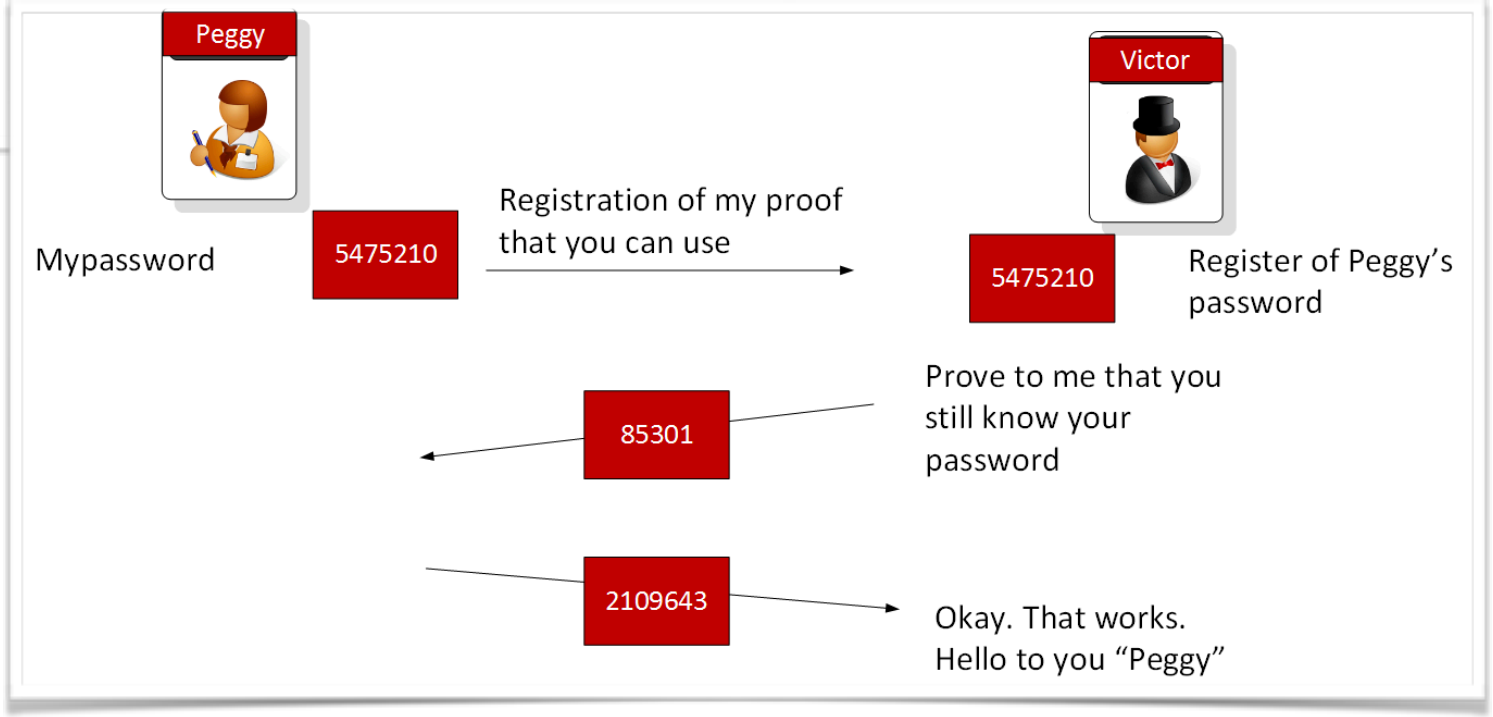


Zero Knowledge Proof

Zero Knowledge Proof

Zero Knowledge Proof





Zero Knowledge Proof

Peggy

Victor

Mypassword –

How To Prove Yourself: Practical Solutions to Identification and Signature Problems

Amos Fiat and Adi Shamir
Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot 76100, Israel

tor

Register of Peggy's password

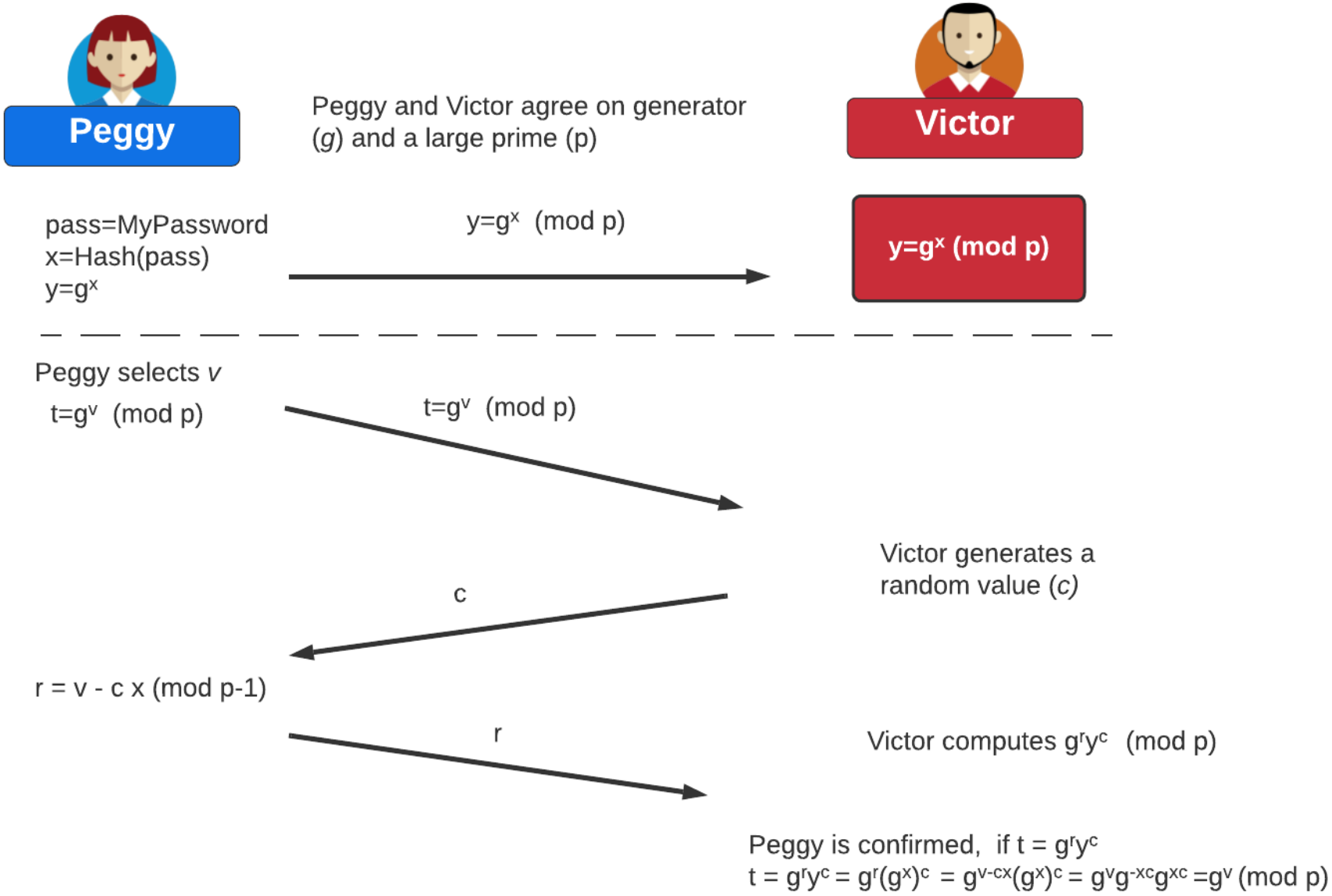
t you

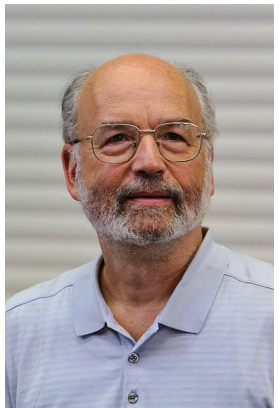
2109643

Okay. That works.
Hello to you "Peggy"

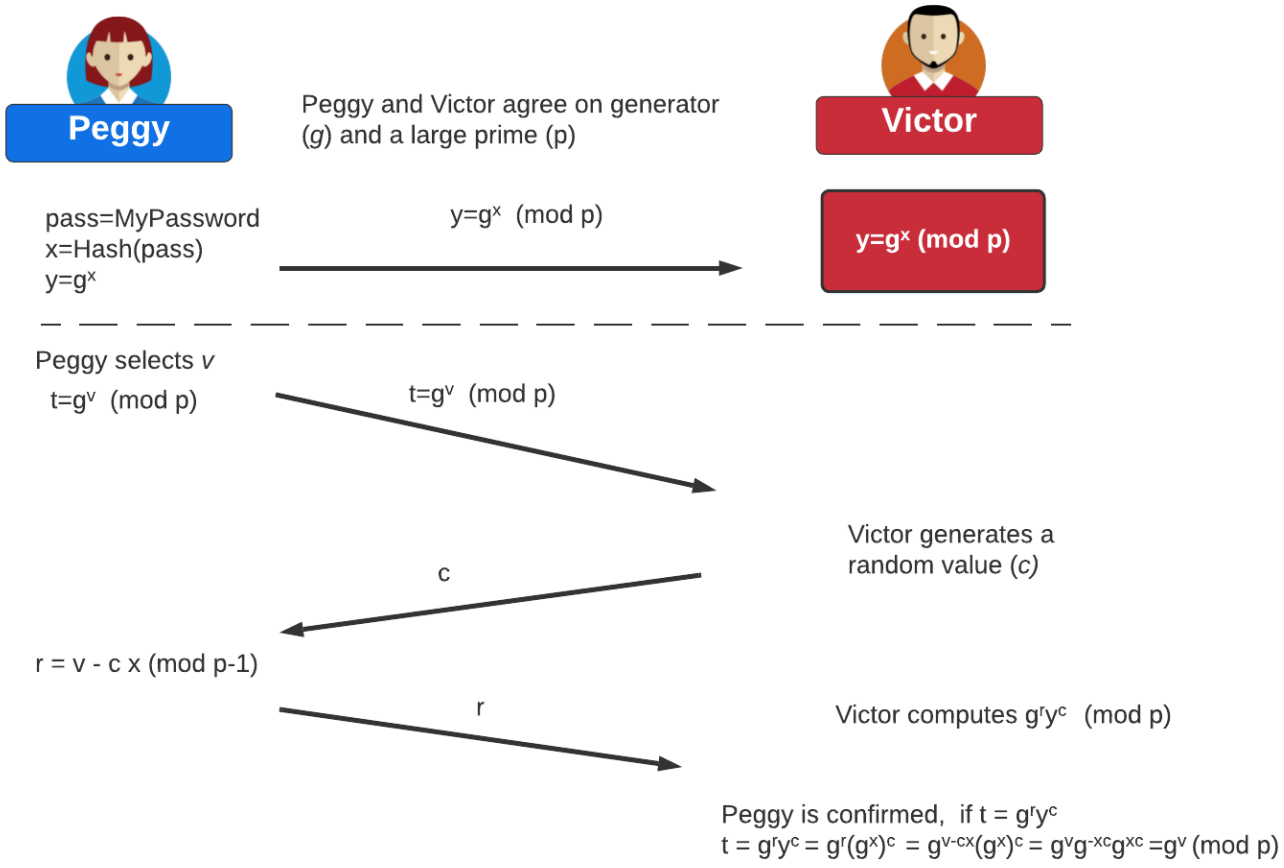
Interactive random oracle access

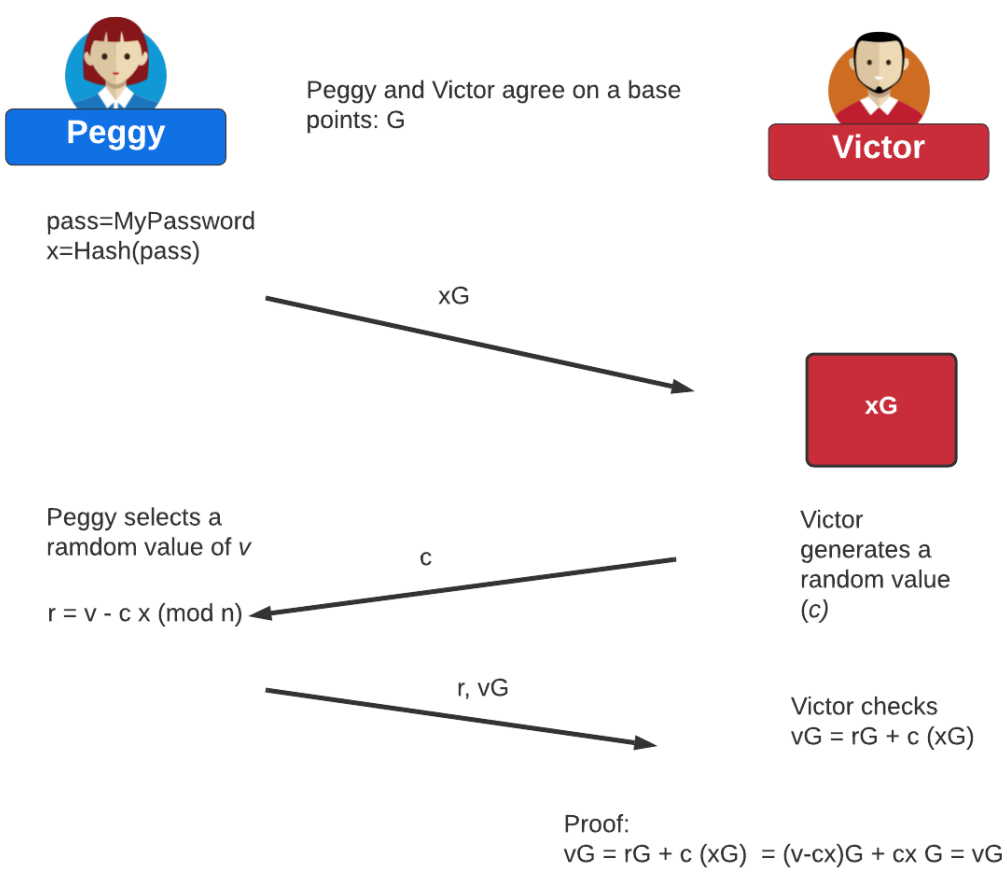
$$g^r y^c = g^{v-cx} (g^x)^c$$
$$= g^{v-cx} g^{xc}$$
$$= g^{v-cx+cx}$$
$$= g^v$$





Zero-knowledge Proof (Interactive – Discrete Logs to ECC)





Non-interactive random oracle access Zero Knowledge Proof (ECC)

Peggy and Victor agree on a base point: G

pass=MyPassword
x=Hash(pass)

xG

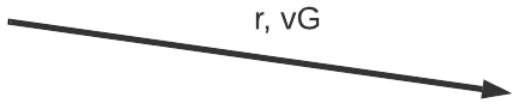
xG

Peggy selects *v*

c=H(G,vG,xG,userID)

r = v - c x (mod n)

r, vG



Victor checks
vG = rG + c (xG)

Proof:
vG = rG + c (xG) = (v-cx)G + cx G = vG

NIZK (Non-interactive Zero Knowledge) proofs of discrete - log equality

I need to prove Y=xG and Z=xM have the same private key (x)

$$log_G(Y) == log_M(Z)$$

Generate k
A=kG
B=kM

c=Hash (G,Y,M,Z,A,B)
s=k-cx

(c,s) →

Check:
A' = sG + cY
B'=sM + cZ
c'=Hash(G,Y,M,Z,A',B
If (c==c') Proof

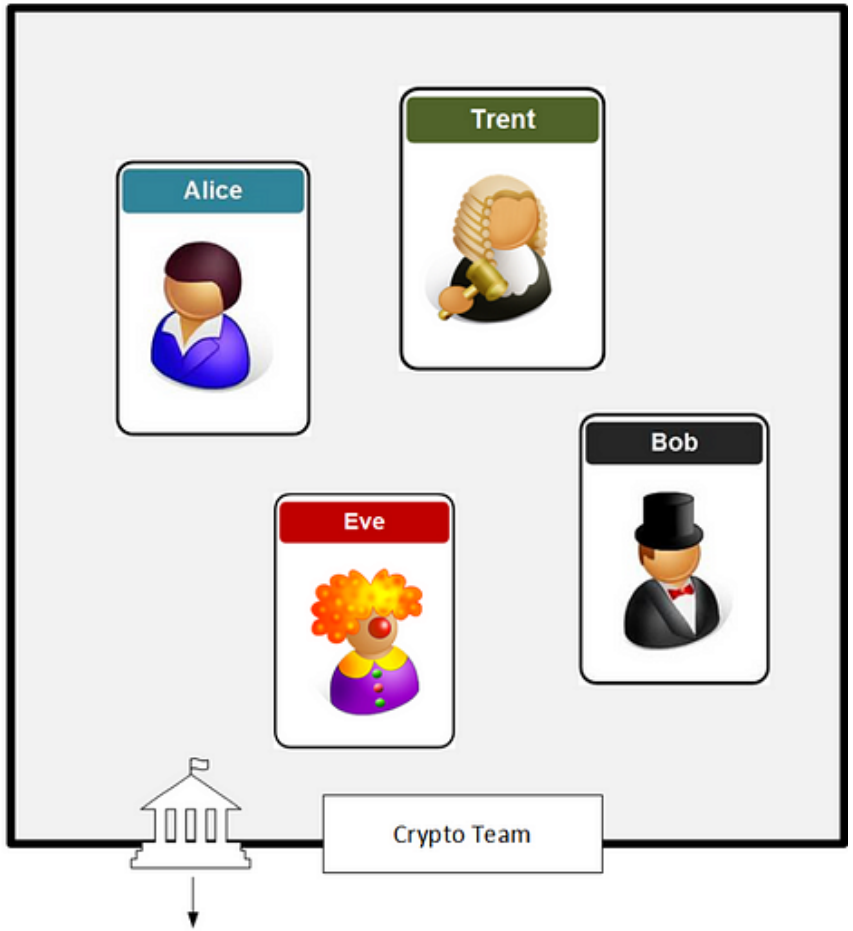
Private key is x and public key is xG

xG, xH →

Proof of identity

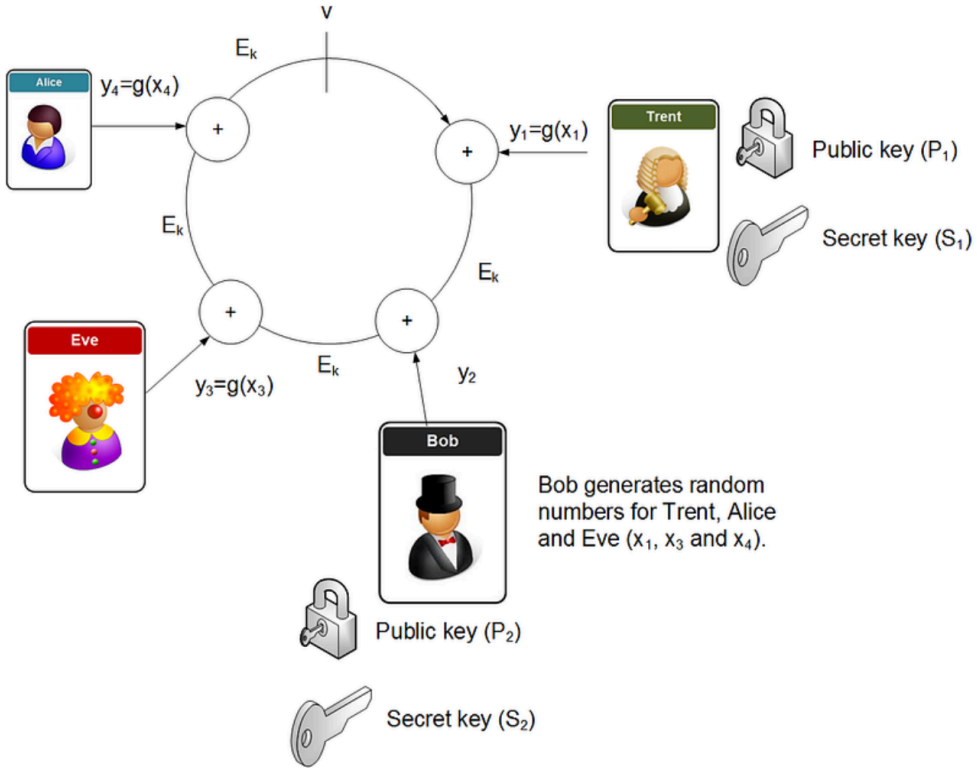
Ring Signatures

I know one of you leaked the information. But which of you was it?



We have a secret.

Signed,
A member of the crypto team



Borromean Ring Signatures*

Gregory Maxwell, Andrew Poelstra†

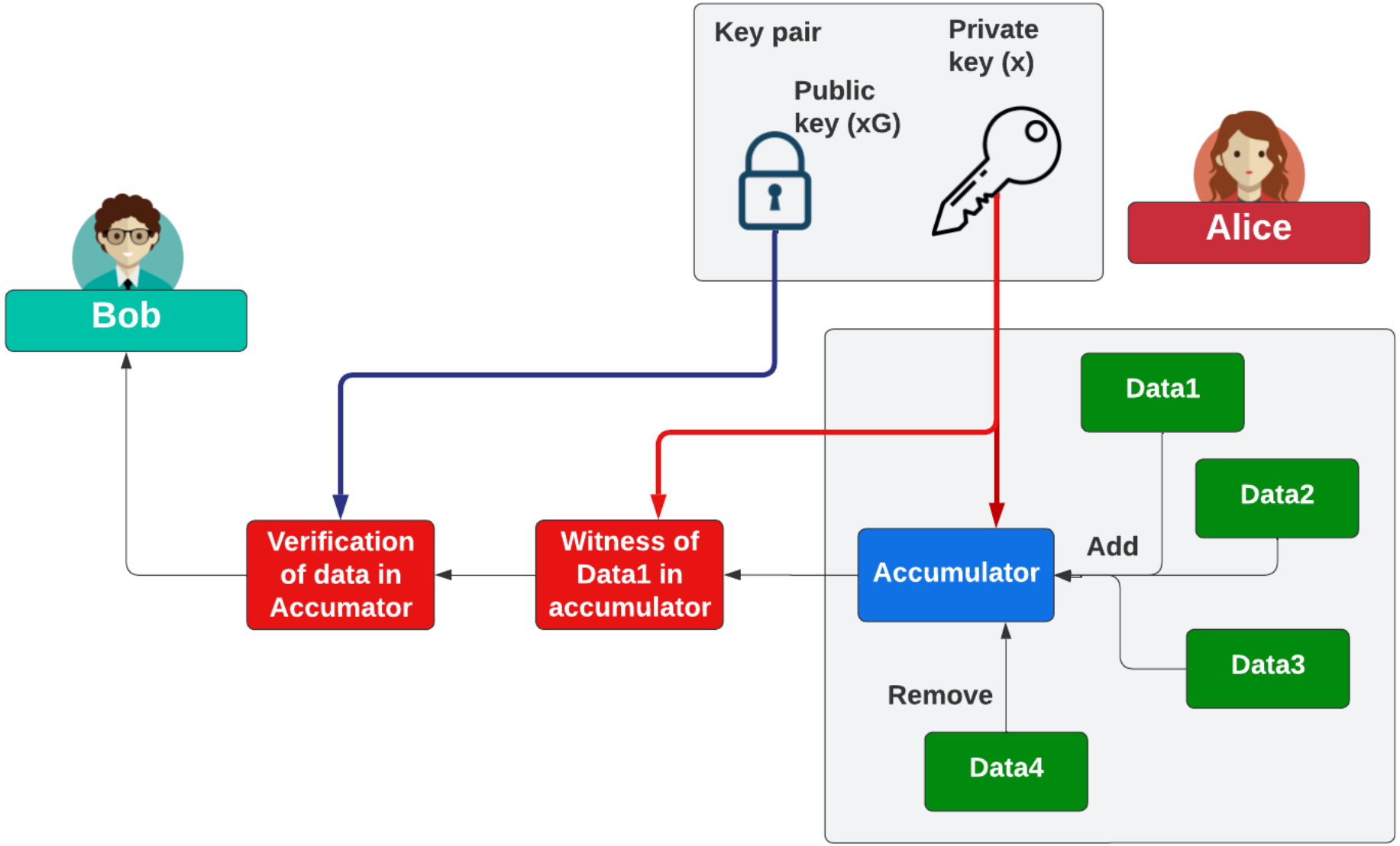
2015-06-02 (commit 34241bb)

Abstract

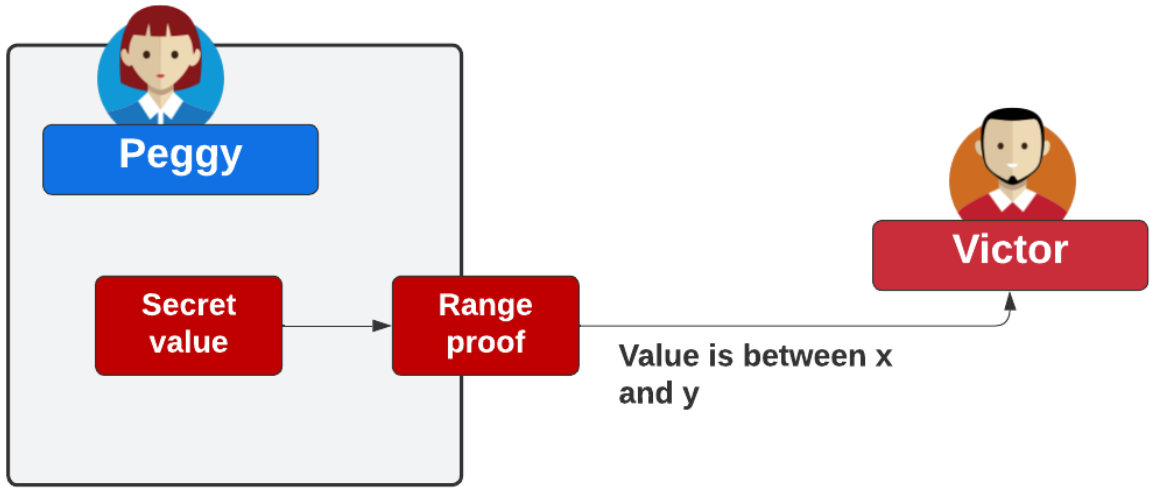
In 2002, Abe, Ohkubo, and Suzuki developed a new type of ring signature based on the discrete logarithm problem, which used a novel commitment structure to gain significant savings in size and verification time for ring signatures[AOS02].

Ring signature, and which provides **anonymity**, **unforgivably** and **collusion resistance.**

Accumulators



Range Proofs – such as Bulletproofs



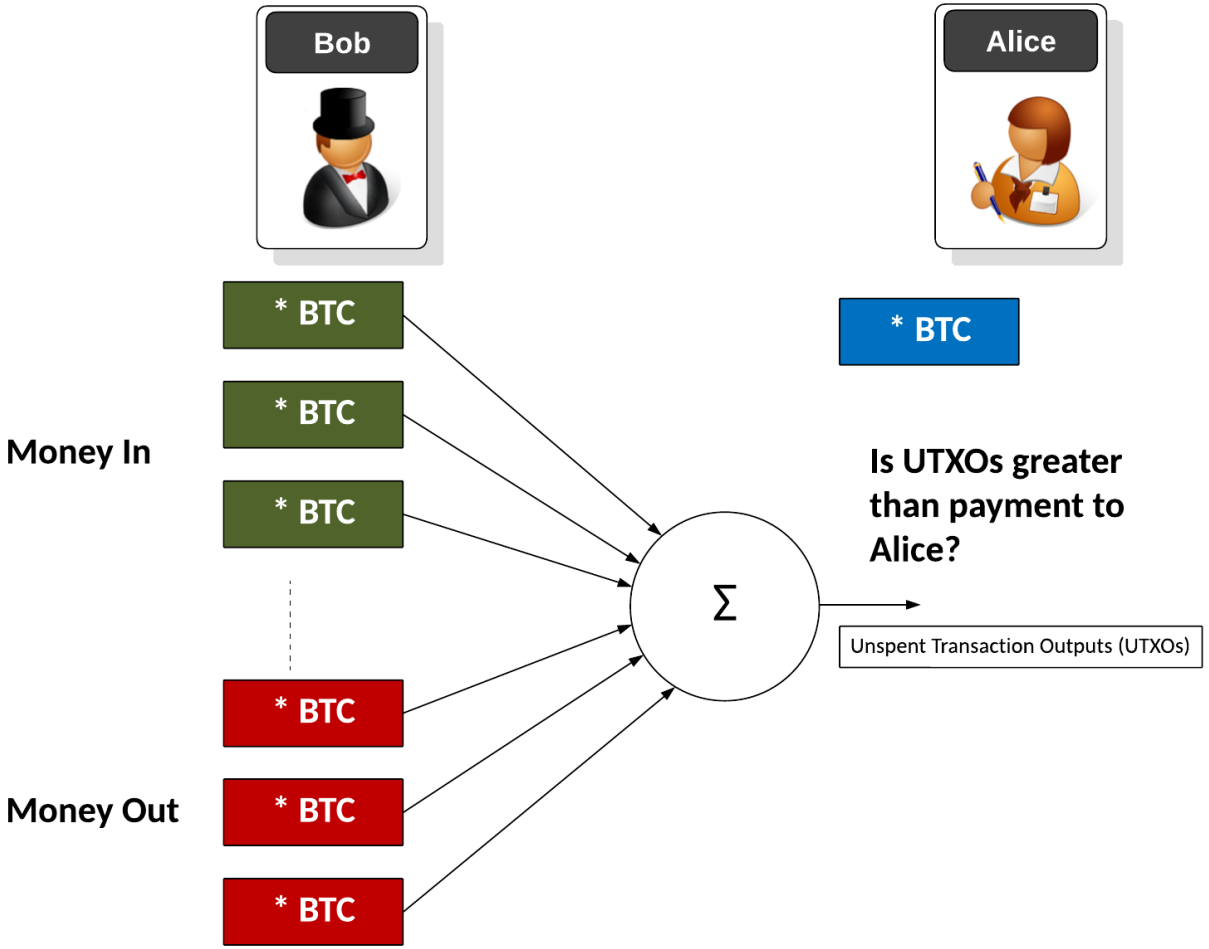
Bulletproofs: Short Proofs for Confidential Transactions and More

Benedikt Bünz*[1], Jonathan Bootle†[2], Dan Boneh‡[1],
Andrew Poelstra§[3], Pieter Wuille¶[3], and Greg Maxwell‖

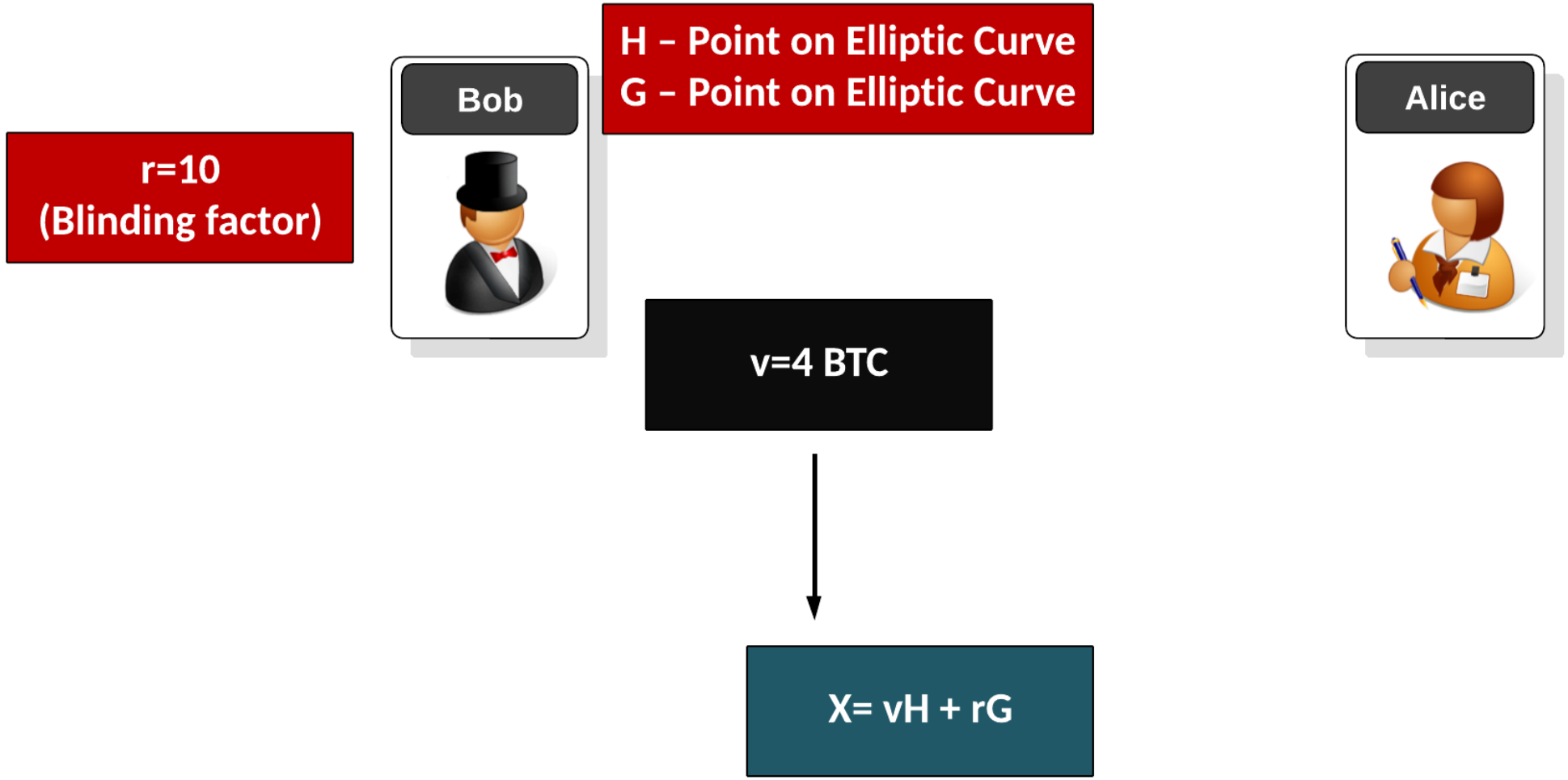
[1]Stanford University
[2]University College London
[3]Blockstream

Abstract

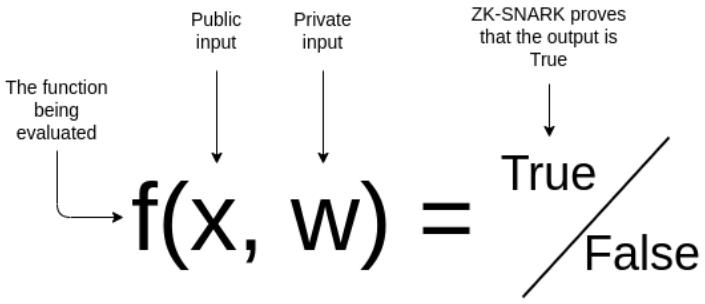
We propose Bulletproofs, a new non-interactive zero-knowledge proof protocol with very short proofs and without a trusted setup; the proof size is only logarithmic in the witness size. Bulletproofs are especially well suited for efficient range proofs on committed values: they enable proving that a committed value is in a range using only $2\log_2(n) + 9$ group and field elements, where n is the bit length of the range. Proof generation and verification times are linear in n.



Pedersen Commitment



zkSnarks (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge)– with R1CS (Rank 1 Constraint System)



$$x^3 + x + 5 == 35$$

```
sym_1 = x * x
y = sym_1 * x
sym_2 = y + x
~out = sym_2 + 5
```

```
x (Private input): 3, y (Public input): 35
Flat: func exp3(private a):
        b = a * a
        c = a * b
        return c

func main(private s0, public s1):
        s3 = exp3(s0)
        s4 = s3 + s0
        s5 = s4 + 5
        equals(s1, s5)
        out = 1 * 1

Witness:  [1 35 3 9 27 30 35 1]

R1CS from flat code
a: [[0 0 1 0 0 0 0 0] [0 0 1 0 0 0 0 0] [0 0 1 0 1 0 0 0] [5 0 0 0 0 1 0 0] [0 0 0 0
0 0 1 0] [0 1 0 0 0 0 0 0] [1 0 0 0 0 0 0 0]]
b: [[0 0 1 0 0 0 0 0] [0 0 0 1 0 0 0 0] [1 0 0 0 0 0 0 0] [1 0 0 0 0 0 0 0] [1 0 0 0
0 0 0 0] [1 0 0 0 0 0 0 0] [1 0 0 0 0 0 0 0]]
c: [[0 0 0 1 0 0 0 0] [0 0 0 0 1 0 0 0] [0 0 0 0 0 1 0 0] [0 0 0 0 0 0 1 0] [0 1 0 0
0 0 0 0] [0 0 0 0 0 0 1 0] [0 0 0 0 0 0 0 1]]

SUCCESS e(piA, Va) == e(piA', g2), valid knowledge commitment for A
SUCCESS e(Vb, piB) == e(piB', g2), valid knowledge commitment for B
SUCCESS e(piC, Vc) == e(piC', g2), valid knowledge commitment for C
SUCCESS e(Vkx+piA, piB) == e(piH, Vkz) * e(piC, g2), QAP disibility checked
SUCCESS e(Vkx+piA+piC, g2KbetaKgamma) * e(g1KbetaKgamma, piB) == e(piK, g2Kgamma)
Valid proofs!!!
```

Fully Homomorphic Encryption (FHE)

Prof Bill Buchanan OBE FRSE

https://asecuritysite.com/openfhe

BLOCKPASS IDENTITY LAB

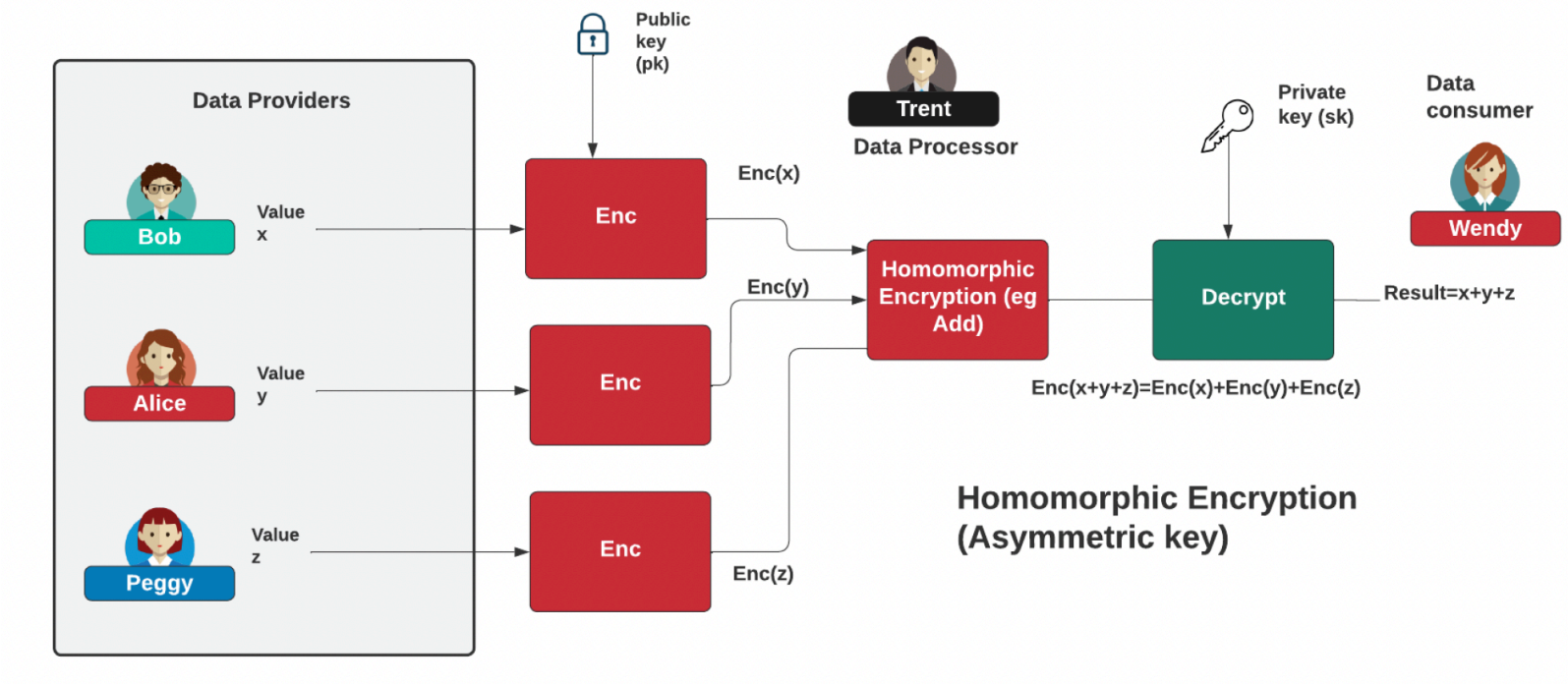
World-leading Collaboration between
Blockpass IDN and Edinburgh Napier University

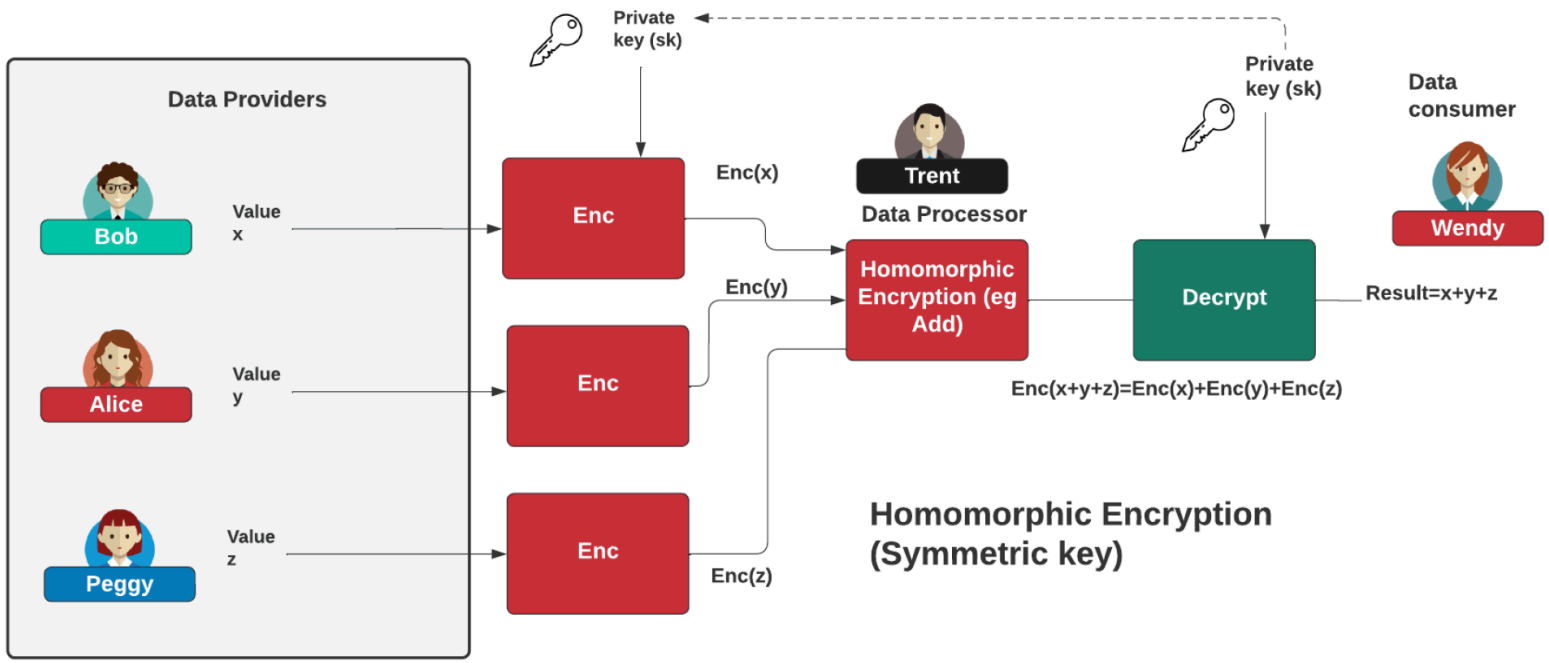
FHE: Background

Prof Bill Buchanan OBE, Blockpass ID Lab

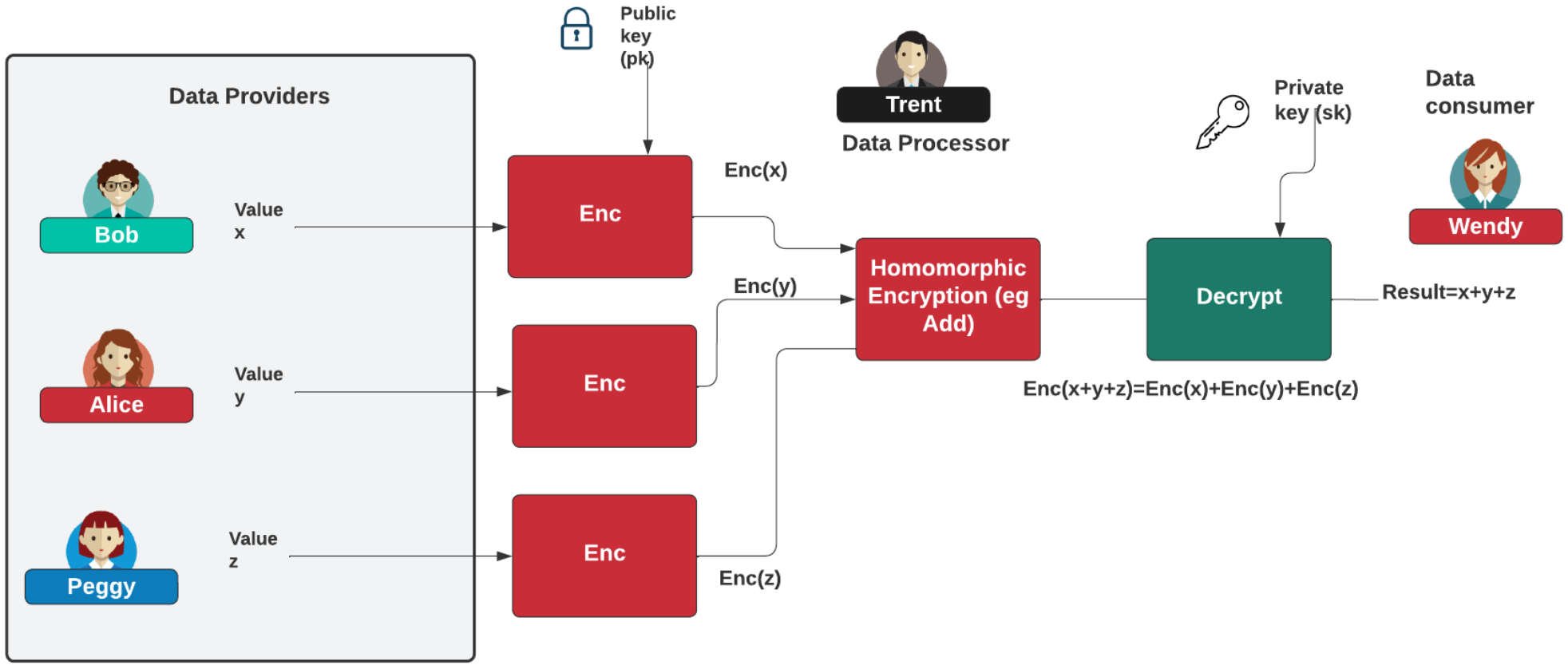
http://asecuritysite.com

FHE


Public key (pk)
Data Providers
Bob
Value x
Alice
Value y
Peggy
Value z
Enc
Enc
Enc
Enc(x)
Enc(y)
Enc(z)
Trent
Data Processor
Homomorphic Encryption (eg Add)
Private key (sk)
Decrypt
Data consumer
Wendy
Result=x+y+z
Enc(x+y+z)=Enc(x)+Enc(y)+Enc(z)
Homomorphic Encryption (Asymmetric key)


Private key (sk)
Data Providers
Bob
Value x
Alice
Value y
Peggy
Value z
Enc
Enc
Enc
Enc(x)
Enc(y)
Enc(z)
Trent
Data Processor
Homomorphic Encryption (eg Add)
Private key (sk)
Decrypt
Data consumer
Wendy
Result=x+y+z
Enc(x+y+z)=Enc(x)+Enc(y)+Enc(z)
Homomorphic Encryption (Symmetric key)

Homomorphic Encryption



https://asecuritysite.com/homomorphic/

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:

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

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

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

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

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

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].

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

$$\mathbf{A} = a_{n-1}x^{n-1} + \ldots + a_1 x + a_1 x^2 + a_0 \tag{2.53}$$

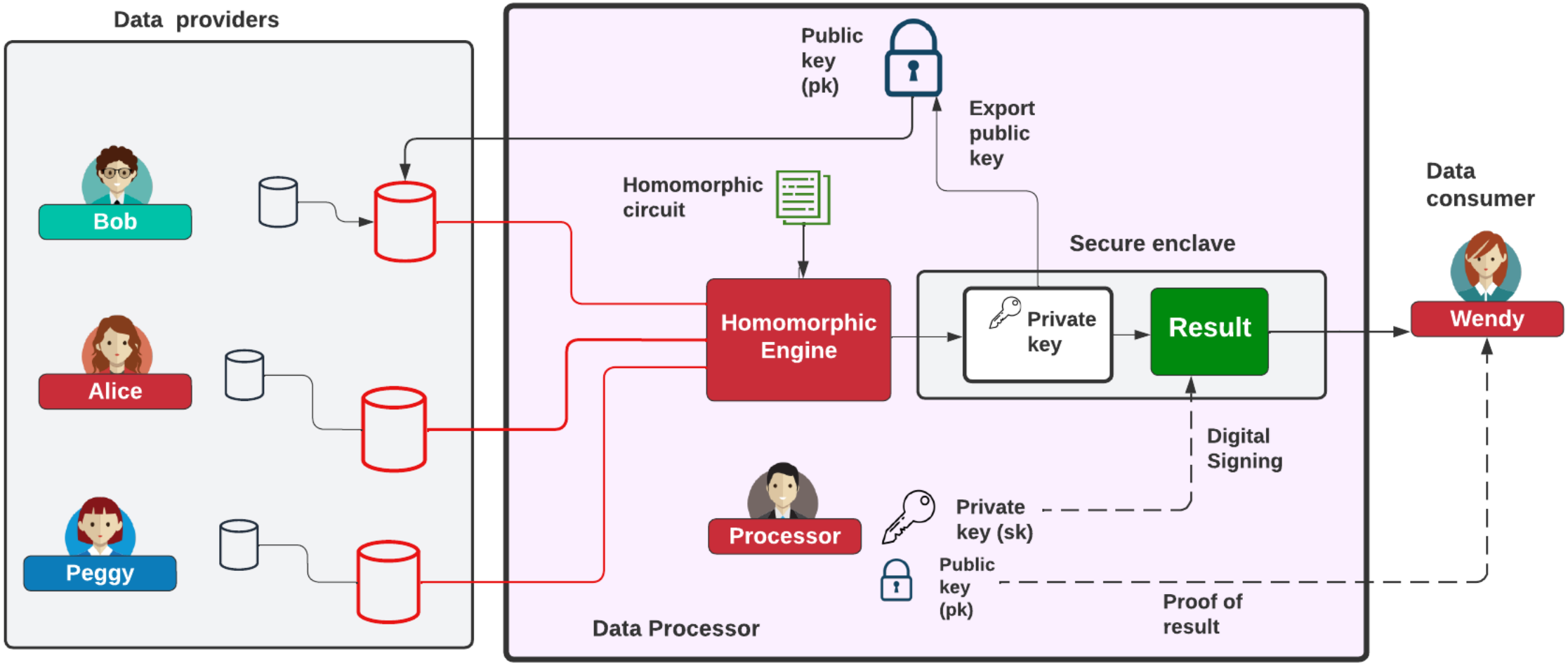
Next Alice will divide by $\Phi(x)$, which is $x^n + 1$:

$$\mathbf{A} = (a_{n-1}x^{n-1} + \ldots + a_1 x + a_1 x^2 + a_0) \div (x^n + 1) \tag{2.54}$$

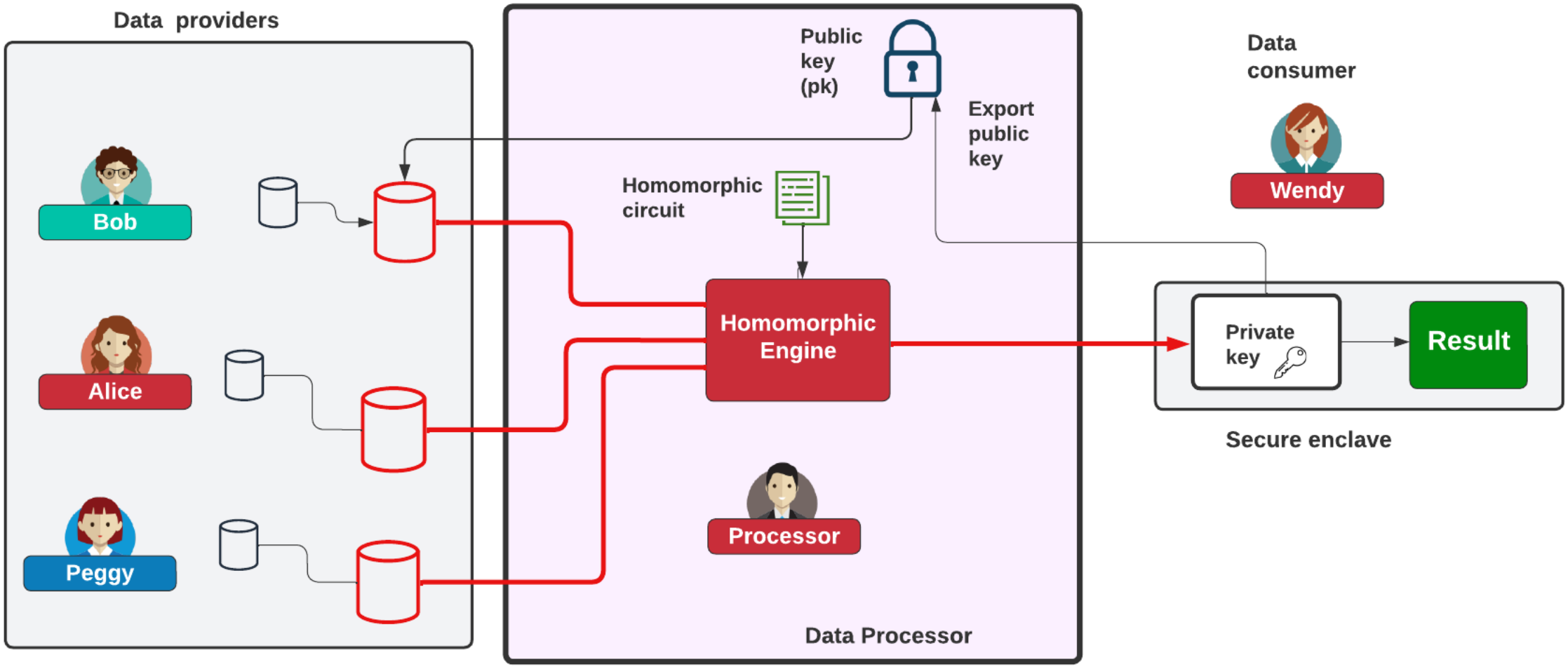
FHE

- 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].

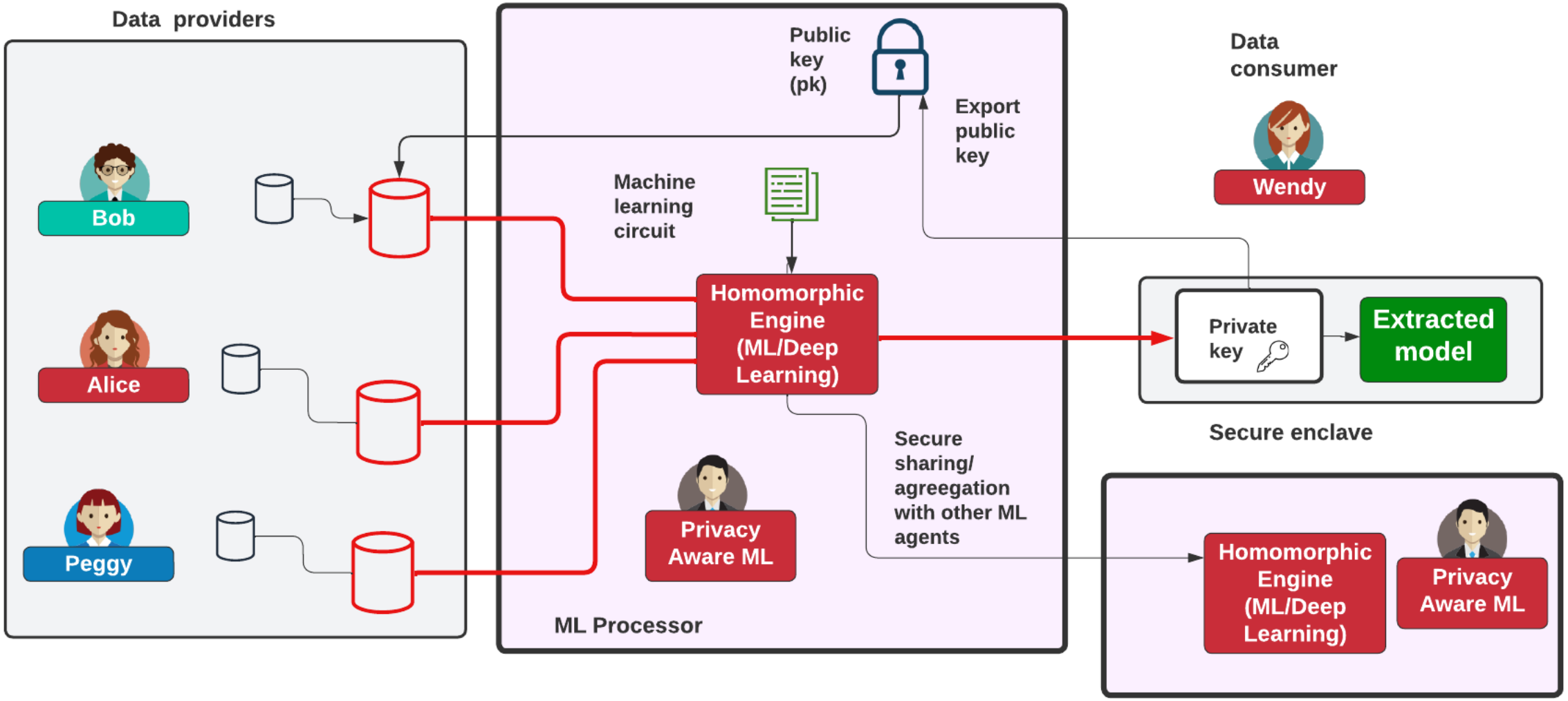
Homomorphic Processing



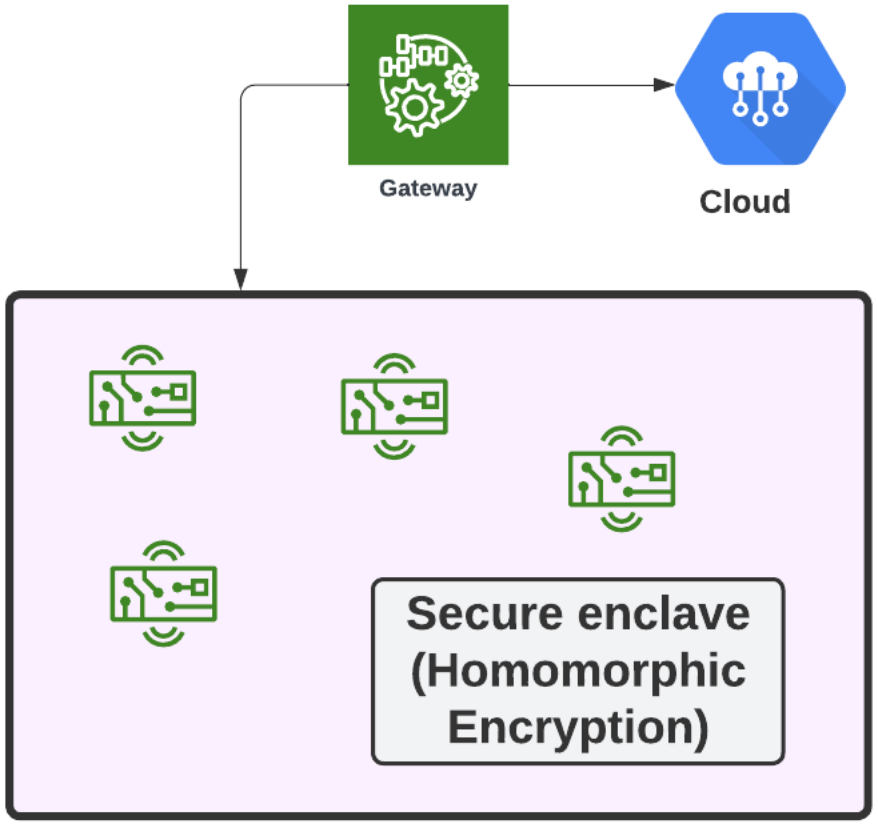
Homomorphic Processing



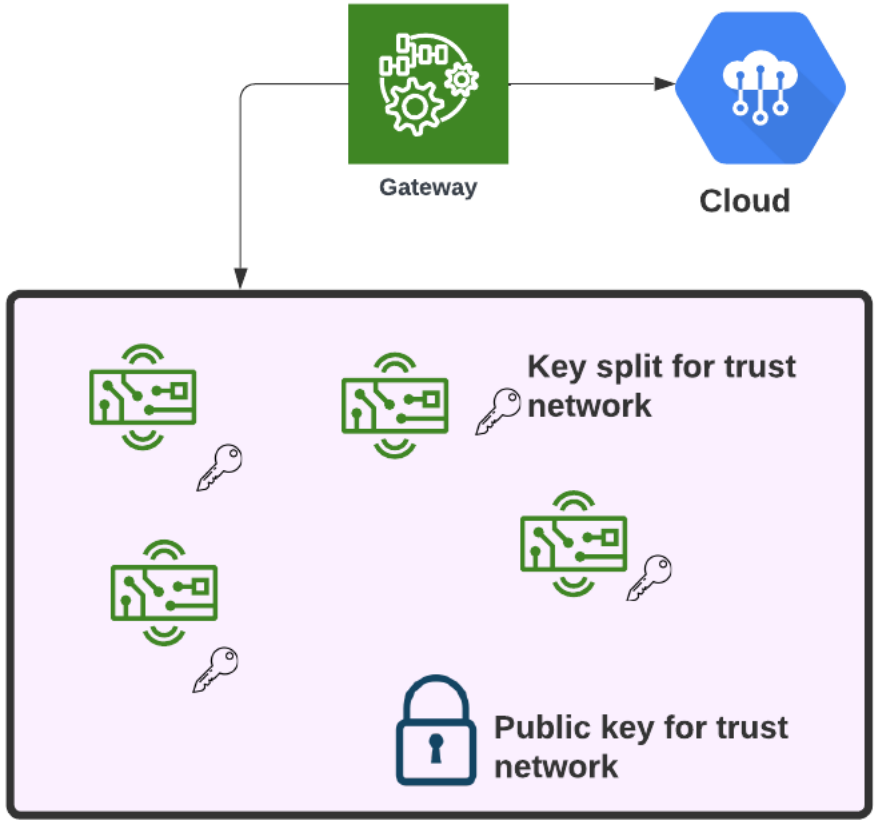
Homomorphic ML



Edge Computing/Trusted Environment



Edge computing



Distributed signing with consensus

FHE: Concepts and Code

Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com

Libraries

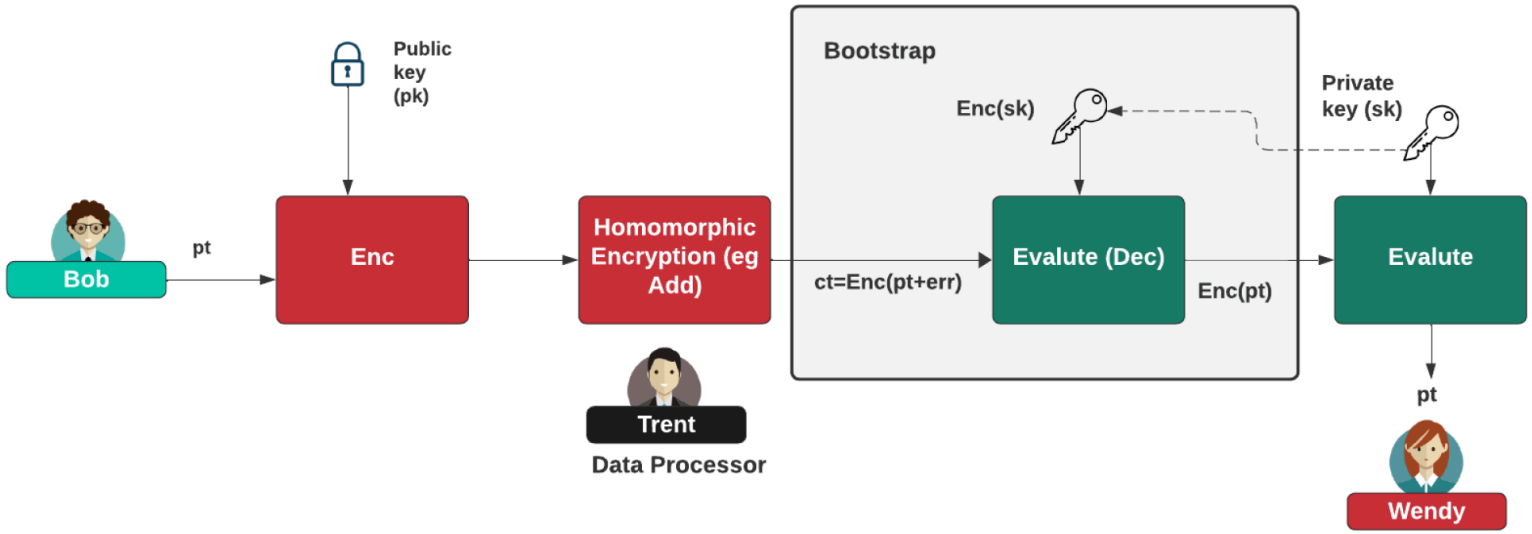
Product	Creator	Language	License	Summary
SEAL [177]	Microsoft	C++	MIT	Widely-used FHE library that implements BFV for modular arithmetic and CKKS for approximate arithmetic.
HElib [116]	IBM	C++	Apache-2.0	Widely-used FHE library that implements BGV for modular arithmetic and CKKS for approximate arithmetic.
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

OpenFHE

- Brakerski/Fan-Vercauteren (**BFV**) scheme for integer arithmetic
- Brakerski-Gentry-Vaikuntanathan (**BGV**) scheme for integer arithmetic
- Cheon-Kim-Kim-Song (**CKKS**) scheme for real-number arithmetic (includes approximate bootstrapping)
- Ducas-Micciancio (**DM**) and Chillotti-Gama-Georgieva-Izabachene (**CGGI**) schemes for Boolean circuit evaluation.

Bootstrapping



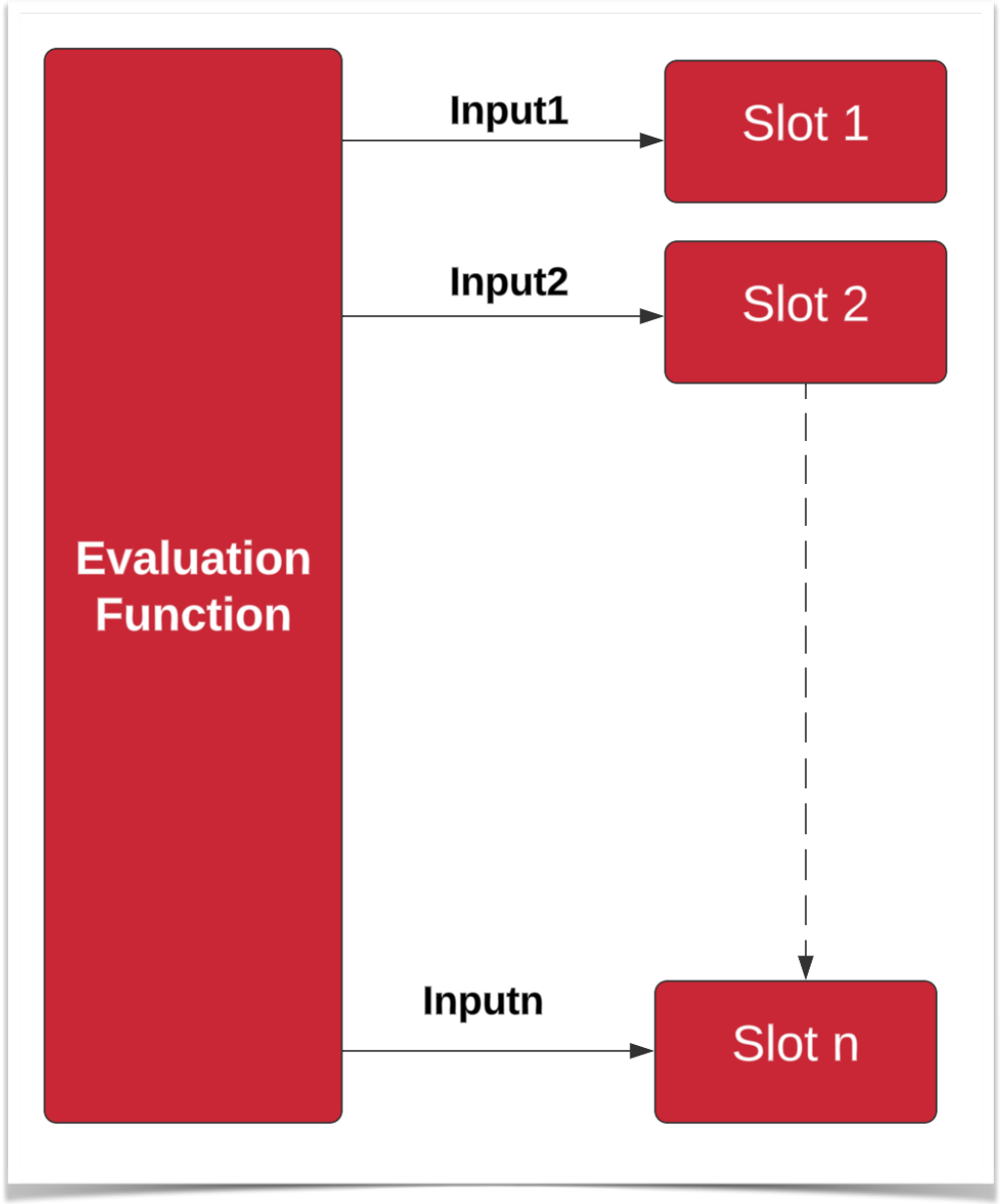
The main bootstrapping methods are CKKS [52], DM [57]/CGGI, and BGV/BFV. Overall, CKKS is generally the fastest bootstrapping method, while DM/CGGI is efficient with the evaluation of arbitrary functions. These functions approximate maths functions as polynomials (such as with Chebyshev approximation).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

Bootstrapping and Slots

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

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

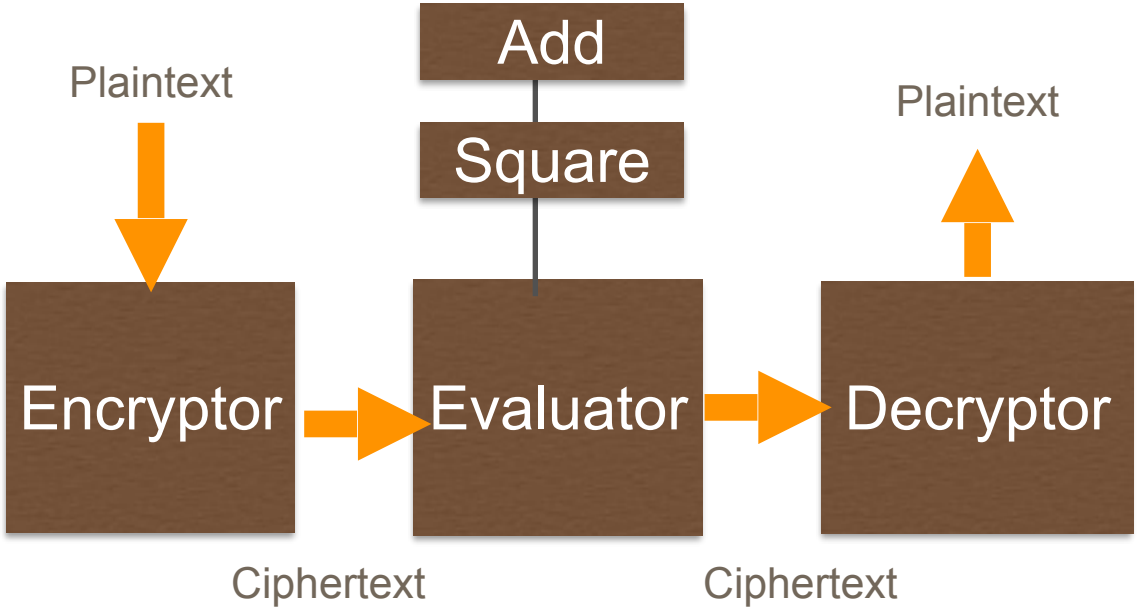
When we reach zero or the noise level is too high, we need to bootstrap - raises the level and reduce noise.



SEAL BFV/BGV

PolyModulusDegree: Defines number of coefficients in plaintext polynomials and the size of ciphertext elements. Must be a power of 2 (such as 1024, 2048, 4096, 8192, 16384, or 32768). Affects performance)

PlainModulus: Largest coefficient that plaintext polynomials can represent. Affects performance.

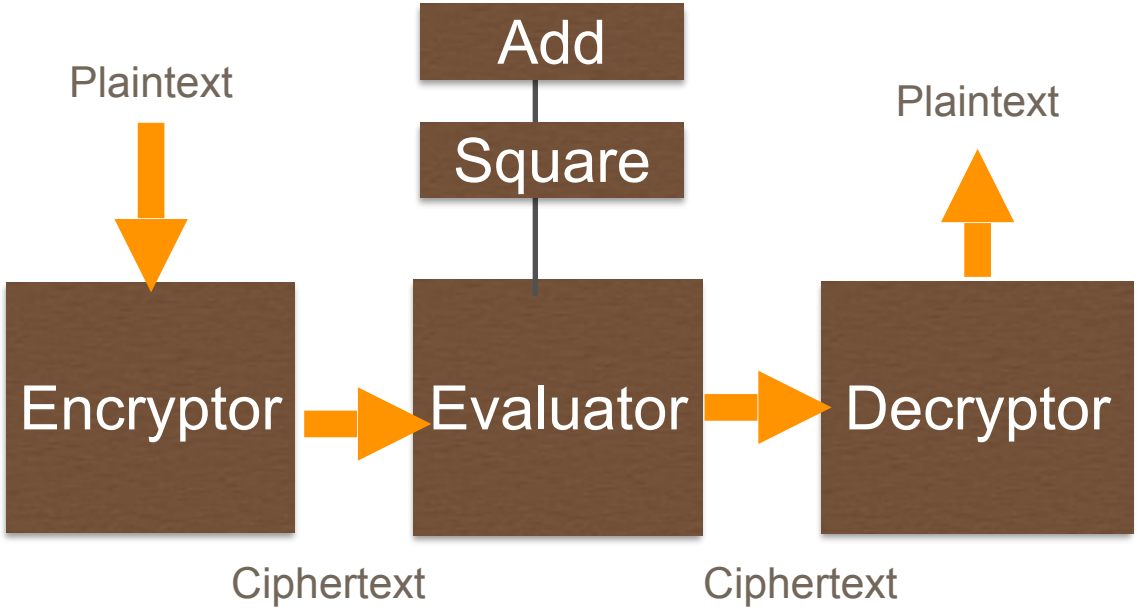



```
EncryptionParameters parms = new EncryptionParameters(SchemeType.BFV);

if (type=="BFV")  parms = new EncryptionParameters(SchemeType.BFV);
else if  (type=="BGV")  parms = new EncryptionParameters(SchemeType.BGV);


Console.WriteLine("Example: {0}\n",type);

ulong polyModulusDegree = mod;
parms.PolyModulusDegree = polyModulusDegree;


parms.CoeffModulus = CoeffModulus.BFVDefault(polyModulusDegree);


parms.PlainModulus = new Modulus(2024);


using SEALContext context = new SEALContext(parms);

using KeyGenerator keygen = new KeyGenerator(context);
using SecretKey secretKey = keygen.SecretKey;
keygen.CreatePublicKey(out PublicKey publicKey);

using Encryptor encryptor = new Encryptor(context, publicKey);
using Evaluator evaluator = new Evaluator(context);
using Decryptor decryptor = new Decryptor(context, secretKey);


using Ciphertext xEncrypted = new Ciphertext();


using Plaintext xPlain = new Plaintext(ULongToString(x));
```

SEAL CKKS

PolyModulusDegree: Defines number of coefficients in plaintext polynomials and the size of ciphertext elements. Must be a power of 2 (such as 1024, 2048, 4096, 8192, 16384, or 32768). Affects performance)

```
using EncryptionParameters parms = new EncryptionParameters(SchemeType.CKKS);


ulong polyModulusDegree = 8192;
parms.PolyModulusDegree = polyModulusDegree;
parms.CoeffModulus = CoeffModulus.Create(
    polyModulusDegree, new int[]{ 60, 40, 40, 60 });


double scale = Math.Pow(2.0, 40);

using SEALContext context = new SEALContext(parms);

Console.WriteLine();

using KeyGenerator keygen = new KeyGenerator(context);
using SecretKey secretKey = keygen.SecretKey;
keygen.CreatePublicKey(out PublicKey publicKey);
keygen.CreateRelinKeys(out RelinKeys relinKeys);
using Encryptor encryptor = new Encryptor(context, publicKey);
using Evaluator evaluator = new Evaluator(context);
using Decryptor decryptor = new Decryptor(context, secretKey);



using CKKSEncoder encoder = new CKKSEncoder(context);
ulong slotCount = encoder.SlotCount;
```



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
```

```
Encrypt
EvalAdd
EvalAddInPlace
EvalAddMany
EvalAddManyInPlace
EvalAddMutable
EvalAddMutableInPlace
EvalAtIndex
EvalAtIndexKeyGen
EvalAutomorphism
EvalAutomorphismKeyGen
EvalBootstrap
```

S 2 OUTPUT TERMINAL DEBUG CONSOLE

```
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
```

```
EvalAdd                inline lbcrypto::Ciphertext<lbcrypto::DCRTPo…
EvalAddInPlace
EvalAddMany
EvalAddManyInPlace
EvalAddMutable
EvalAddMutableInPlace
EvalAtIndex
EvalAtIndexKeyGen
EvalAutomorphism
EvalAutomorphismKeyGen
EvalBootstrap
EvalBootstrapKeyGen
```

```
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;
```

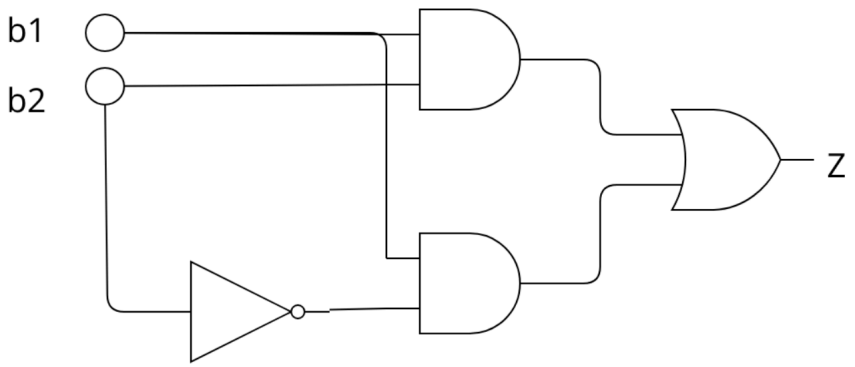
https://asecuritysite.com/openfhe/openfhe_05cpp

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



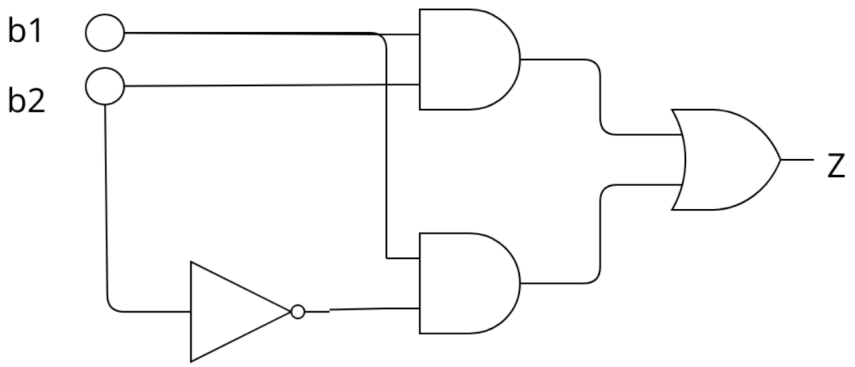
$$(b_1 . b_2) + (b_1 . \bar{b_2})$$

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



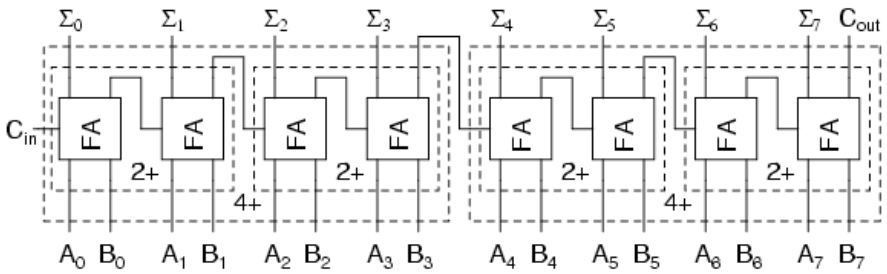
$$(b_1 . b_2) + (b_1 . \bar{b_2})$$

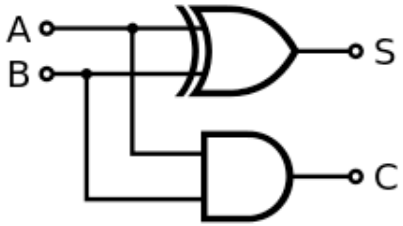
b1	b2	(b1.b2)	(b1.NOT(b2)	Z
0	0	0	0	0
0	1	0	0	0
1	0	0	1	1
1	1	1	0	1

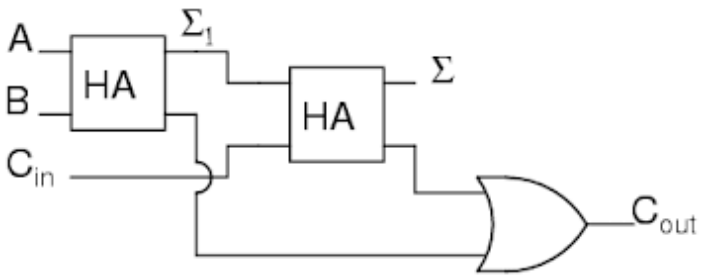
```
auto sk = cc.KeyGen();

std::cout << "Generating the bootstrapping keys... public keys" << std::endl;

// Generate the bootstrapping keys (refresh, switching and public keys)
cc.BTKeyGen(sk, PUB_ENCRYPT);

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

Chebyshev Functions

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

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

```
 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

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 $a_0 \ldots a_n$ 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);
```

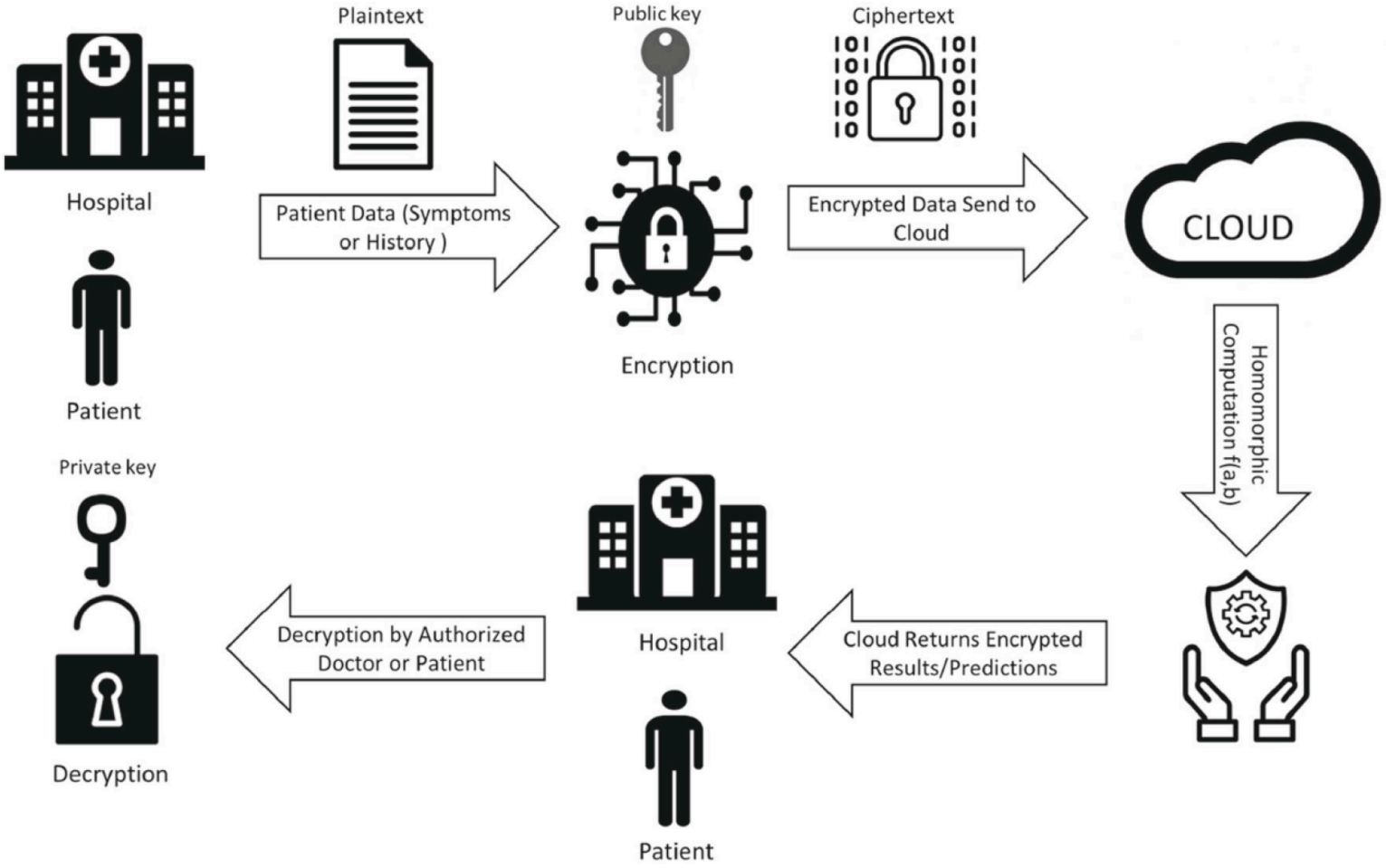
https://asecuritysite.com/openfhe/openfhe_20cpp

FHE: Data Sharing

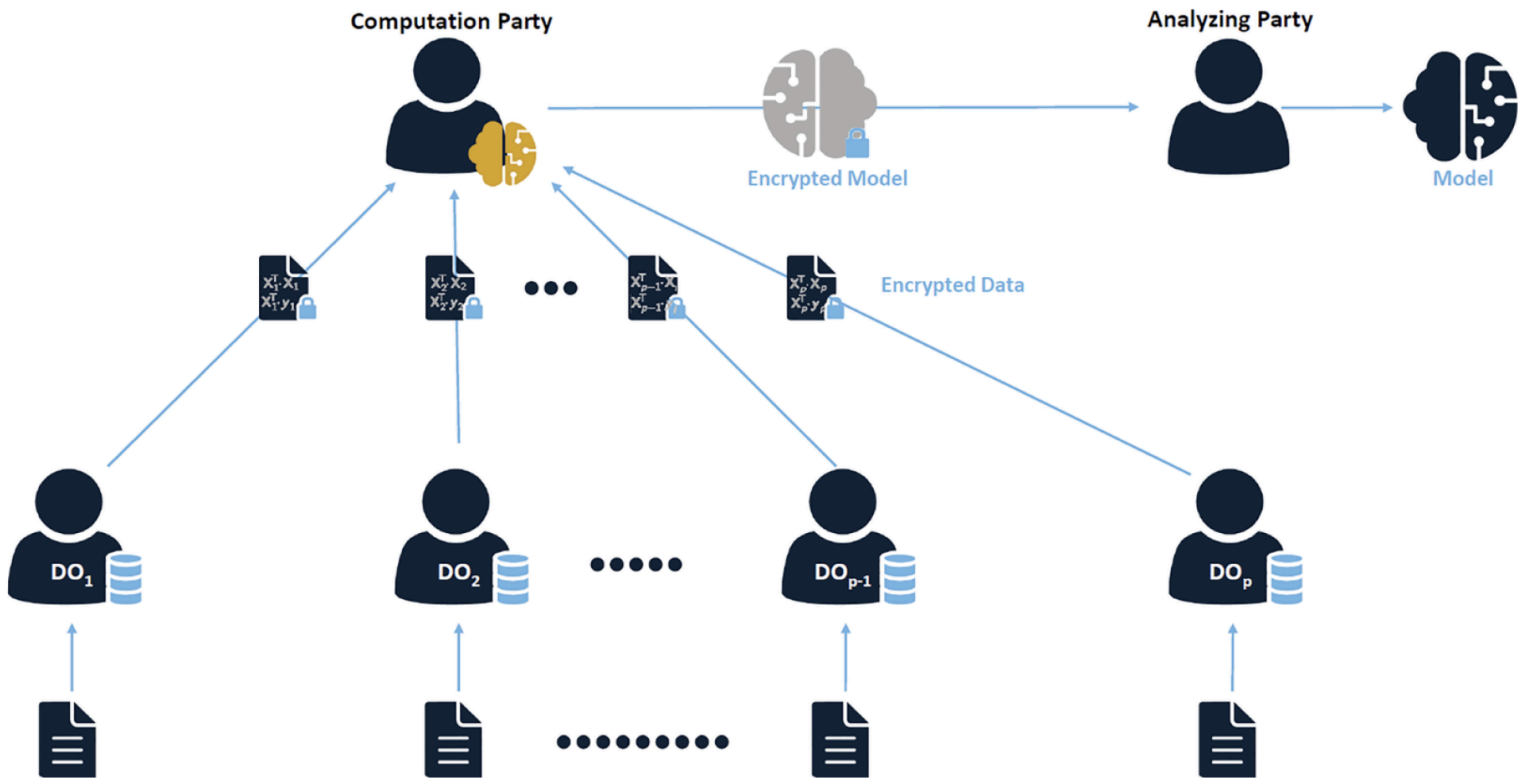
Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com

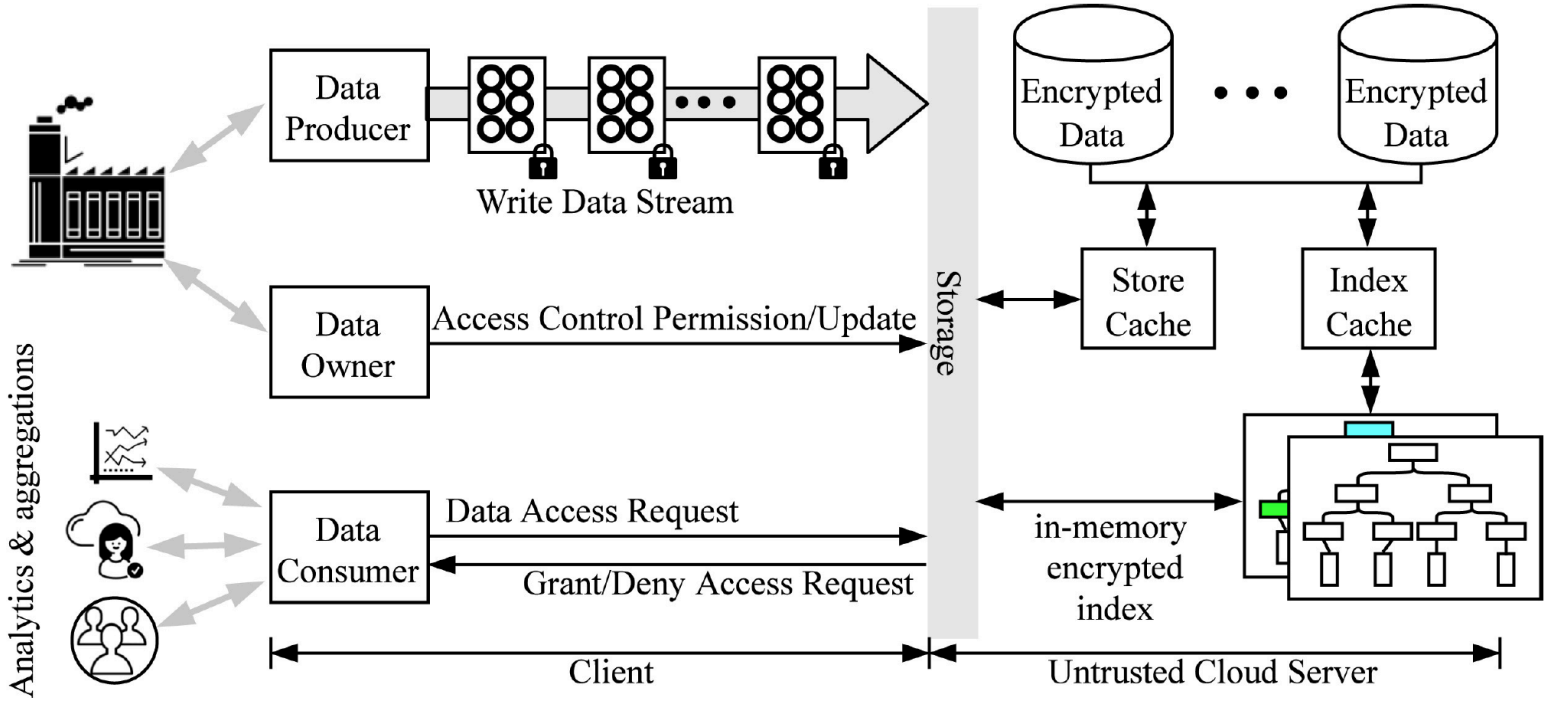
Homomorphic Sharing



Multiparty Sharing



SmartCrypt


Data Producer
Write Data Stream
Data Owner
Access Control Permission/Update
Data Consumer
Data Access Request
Grant/Deny Access Request
Analytics & aggregations
Storage
Encrypted Data
Encrypted Data
Store Cache
Index Cache
in-memory encrypted index
Client
Untrusted Cloud Server

Using Symmetric Key for HE

4.4.1 Order-preserving encryption (OPE)

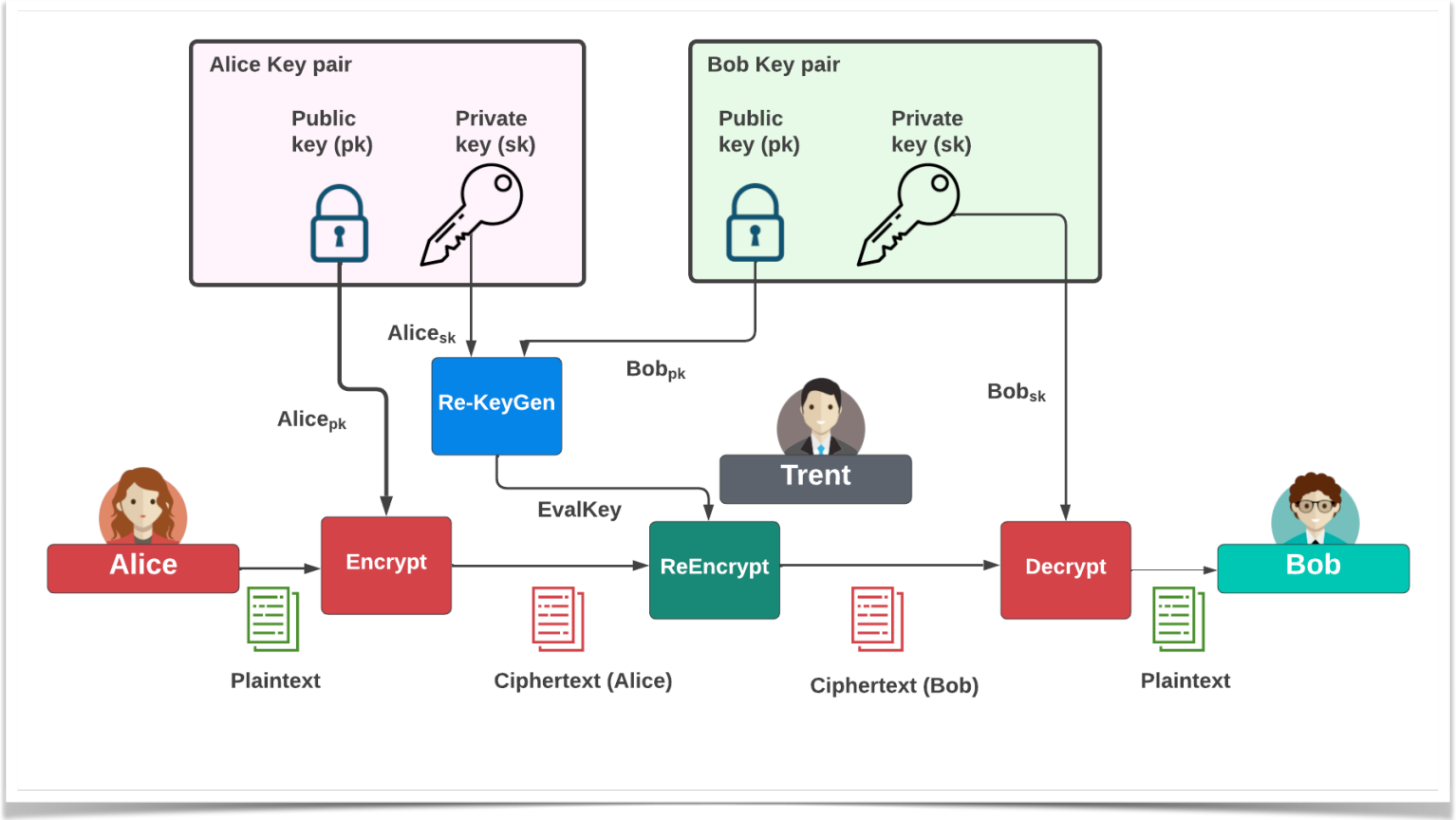
Rahman et al. outline an Order-preserving encryption (OPE) for encrypted searches using symmetric key encryption. [73]. Overall, OPE is a symmetric key encryption method that can be used to create an order on encrypted content. We will lose some information in supporting the ordering of the ciphers, but the security level should still stay acceptable for the application area. If we encrypt a value of A with a key of k to get $E_k(A)$, and then encrypt B to get $E_k(B)$. Then if $A < B$, then:

$$Ek(A) < Ek(B) \tag{4.1}$$

One of the best papers on the subject is [74], and an article is [75].

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);
```



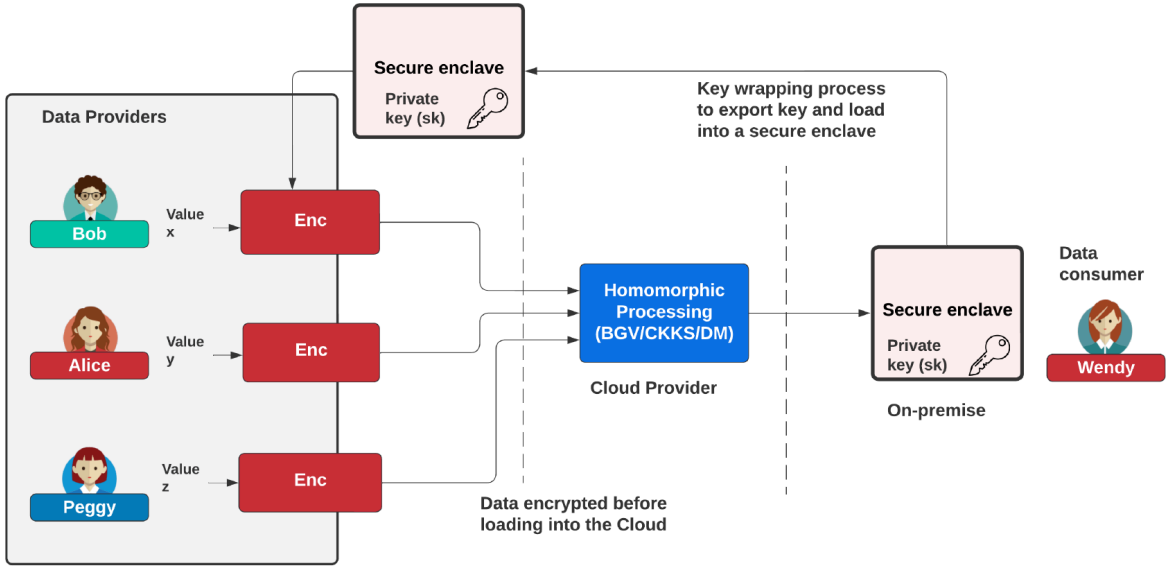
https://asecuritysite.com/openfhe/openfhe_21cpp

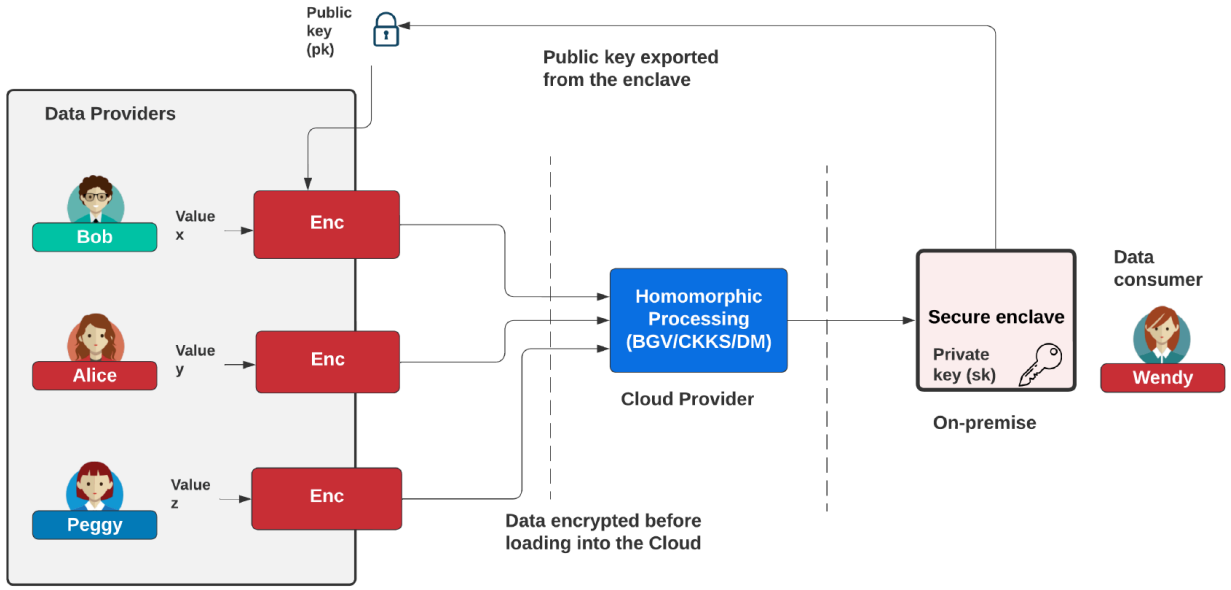
FHE in the Cloud

Prof Bill Buchanan OBE, Blockpass ID Lab

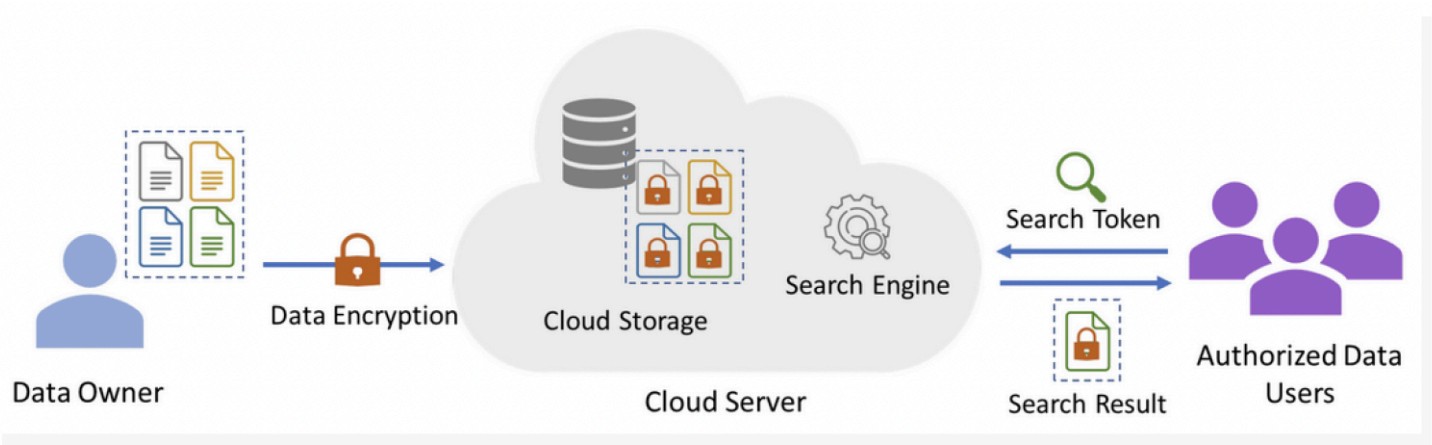
http://asecuritysite.com

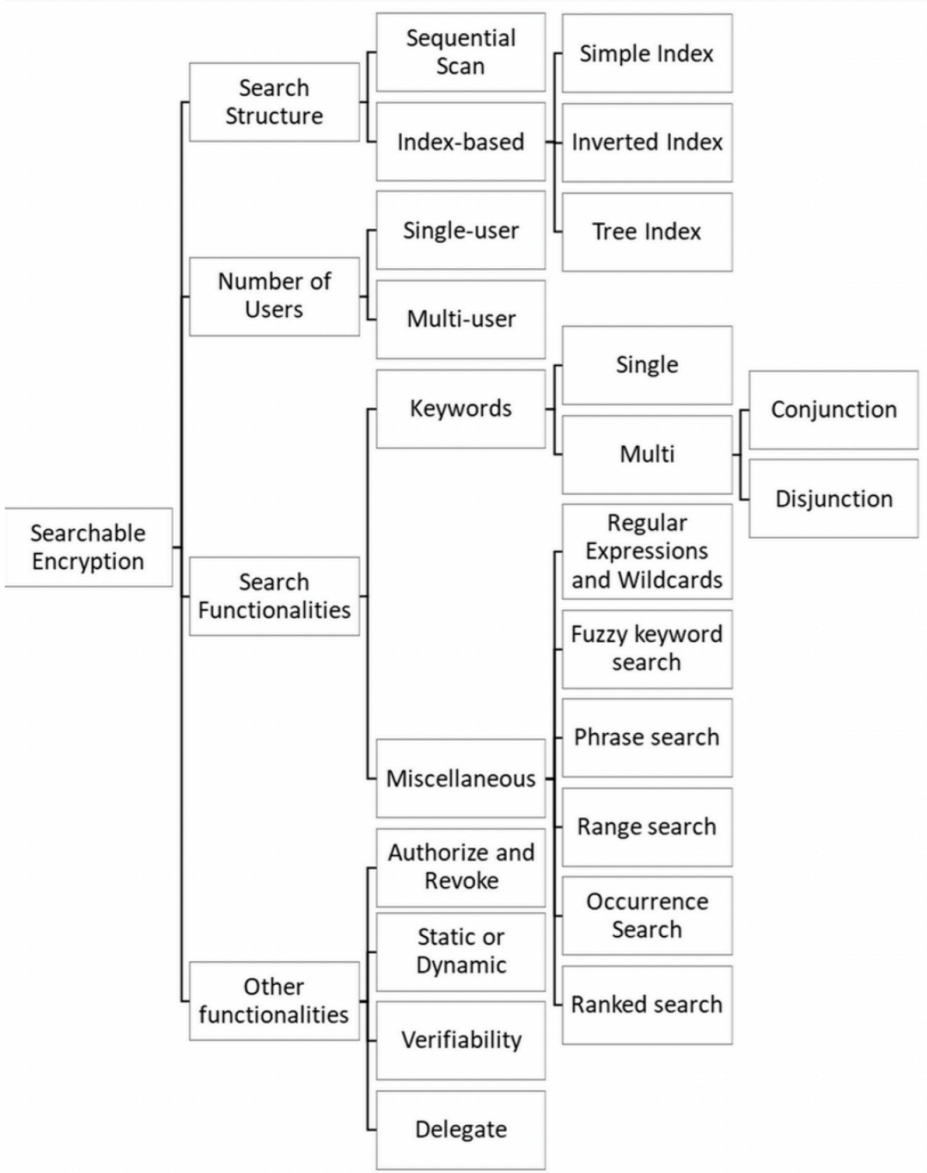
Symmetric/Asymmetric in Cloud





Searchable Encryption

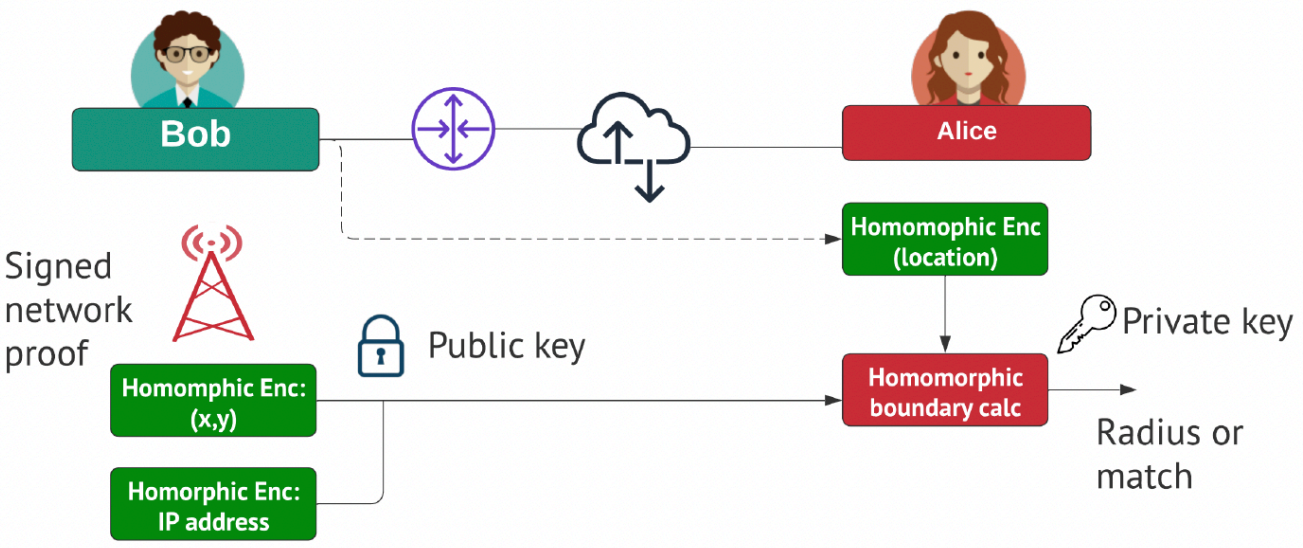




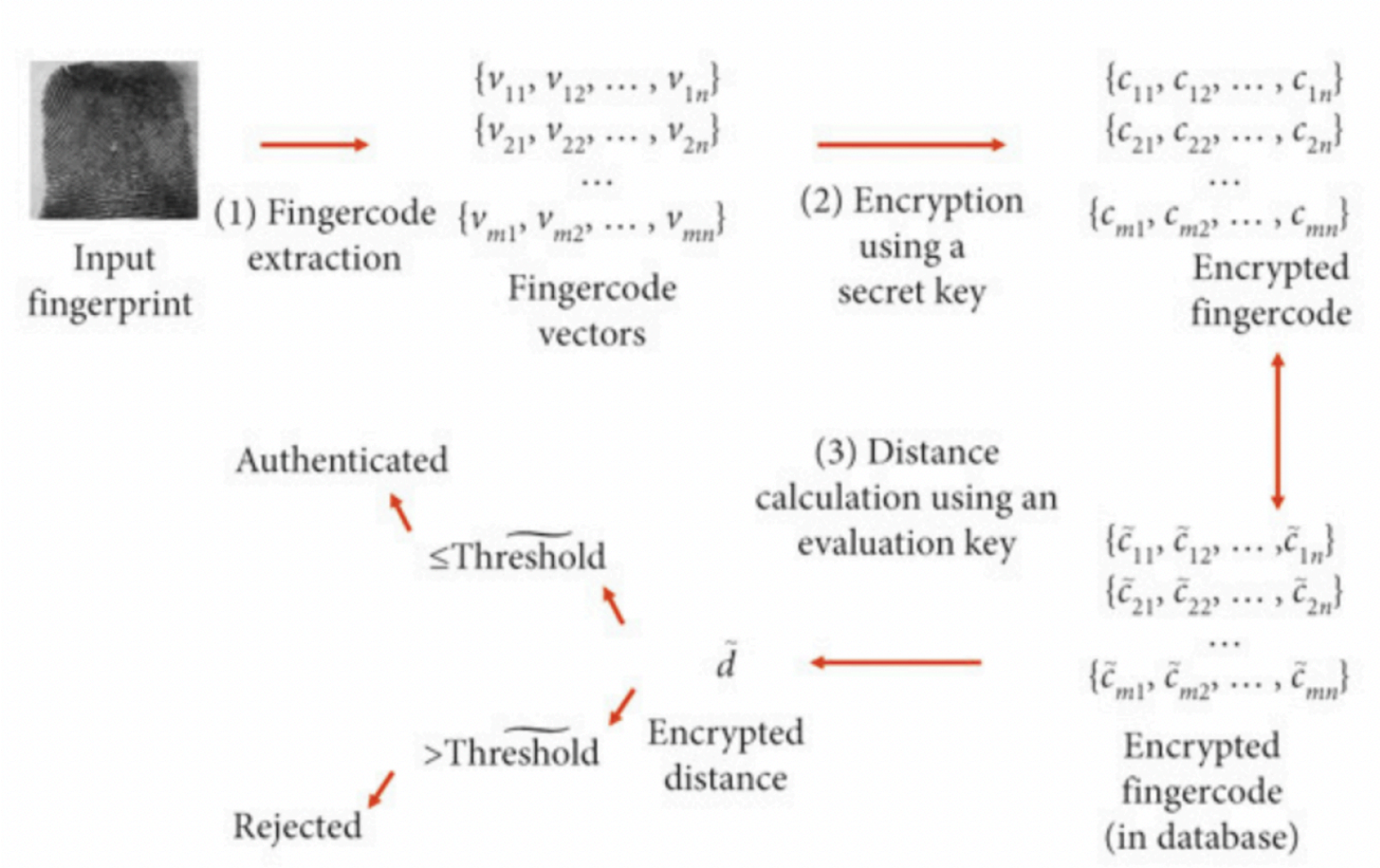
Location Tracking

```
E(TIME1)a E(Location1)a
E(TIME2)a E(Location2)a
E(TIME3)a E(Location2)a
E(TIME1)b E(Location1)b
E(TIME2)b E(Location2)b
E(TIME3)b E(Location2)b
```

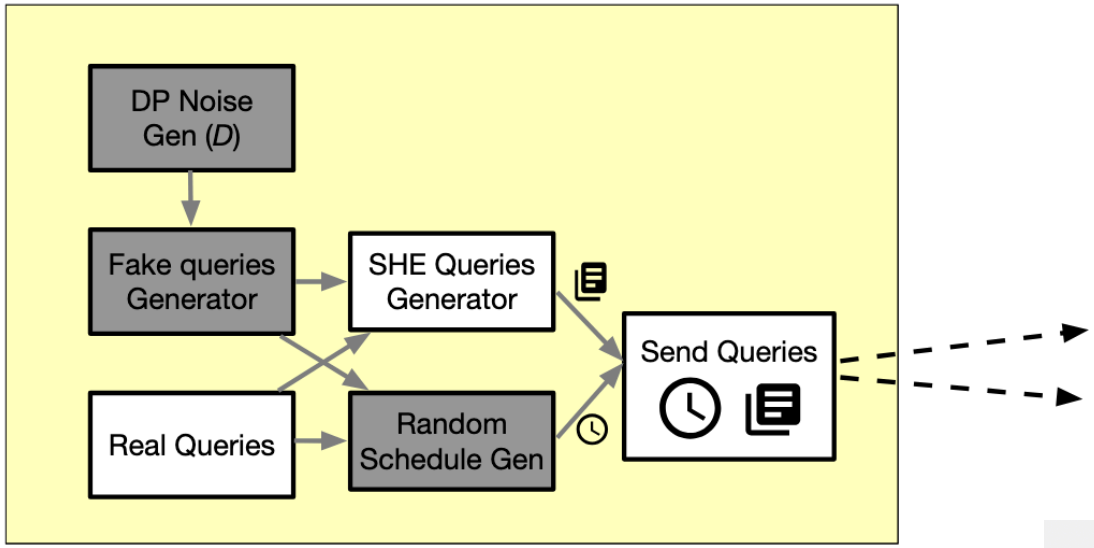
Timestamp	u_1 $rssi$	m	Type	u_2 $rssi$
1509240563.03	-64	D8:84:66:4C:D1:00	WiFi	-85
1509240563.03	-69	D8:84:66:4E:E4:F0	WiFi	-79
1509240563.03	-59	D8:84:66:4E:F0:04	BL	-91

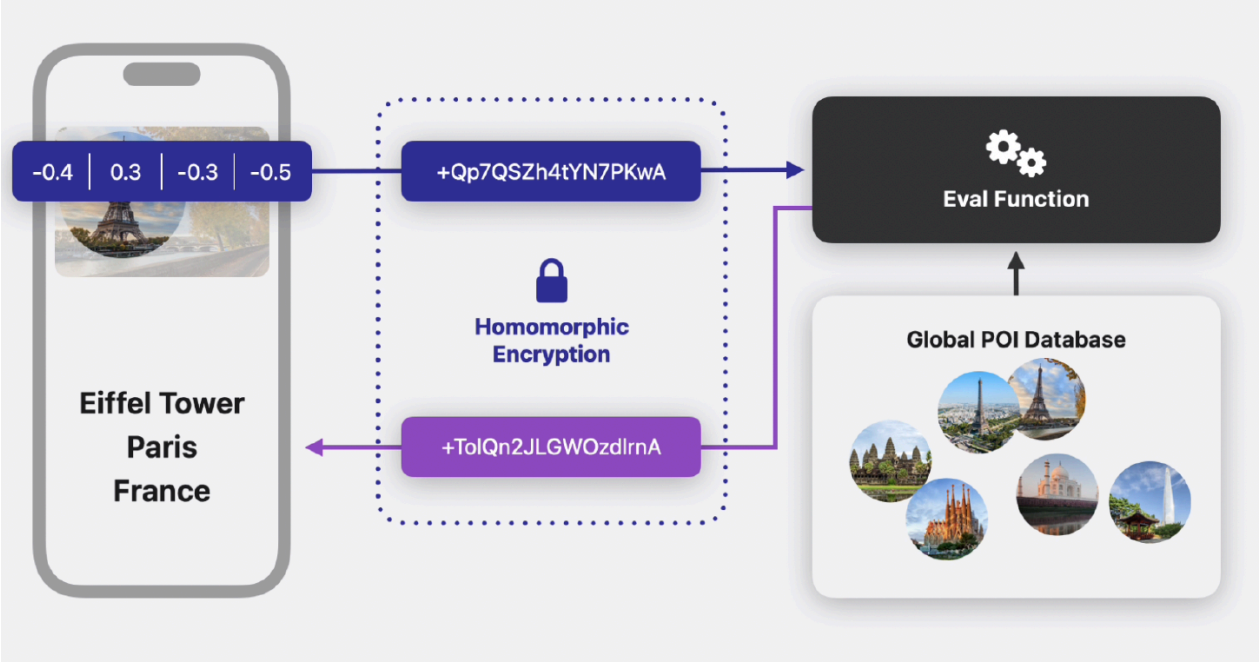


Privacy-aware biometrics/ID



Apple Privacy Aware





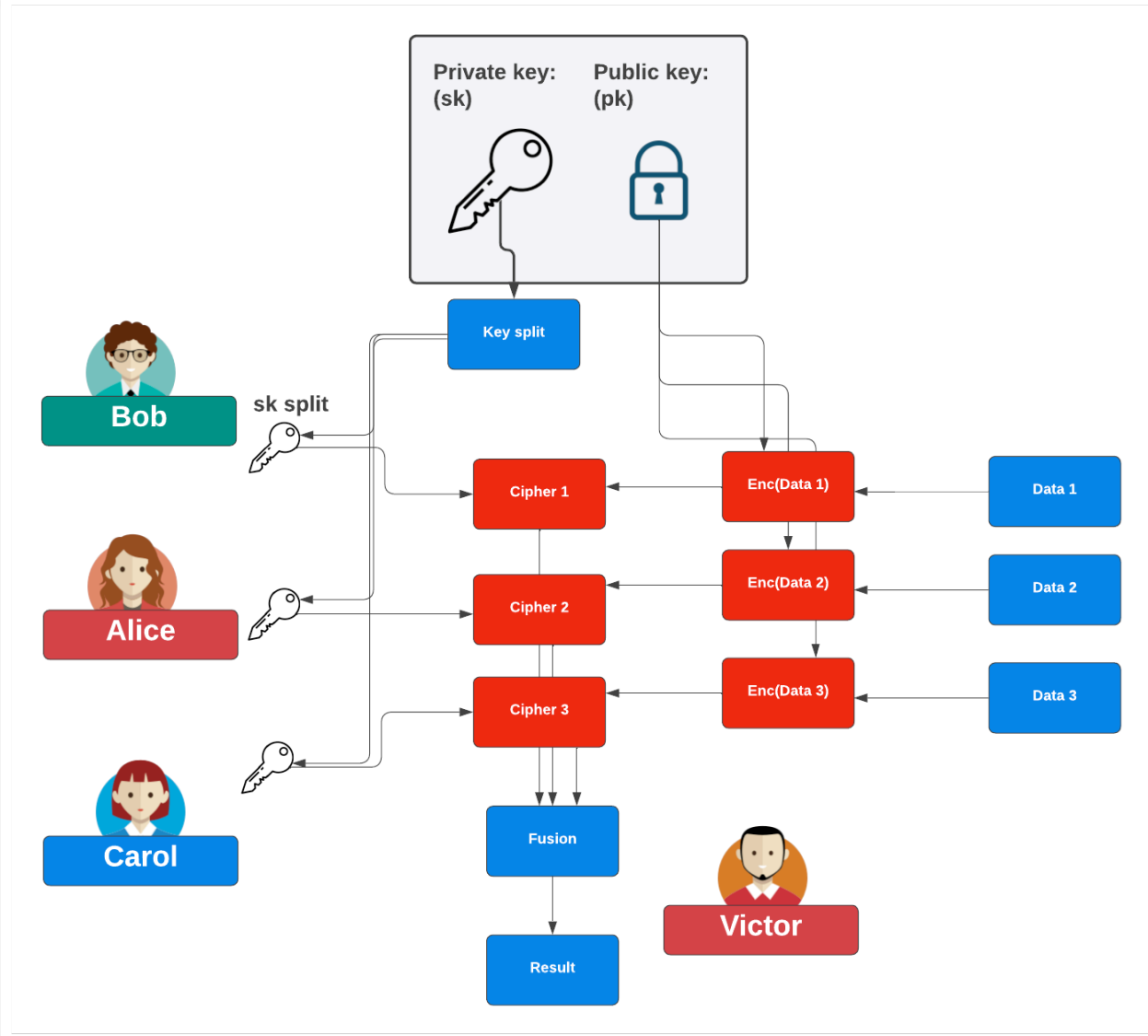
FHE: MPC

Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com

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

FHE: ML

Prof Bill Buchanan OBE, Blockpass ID Lab

http://asecuritysite.com

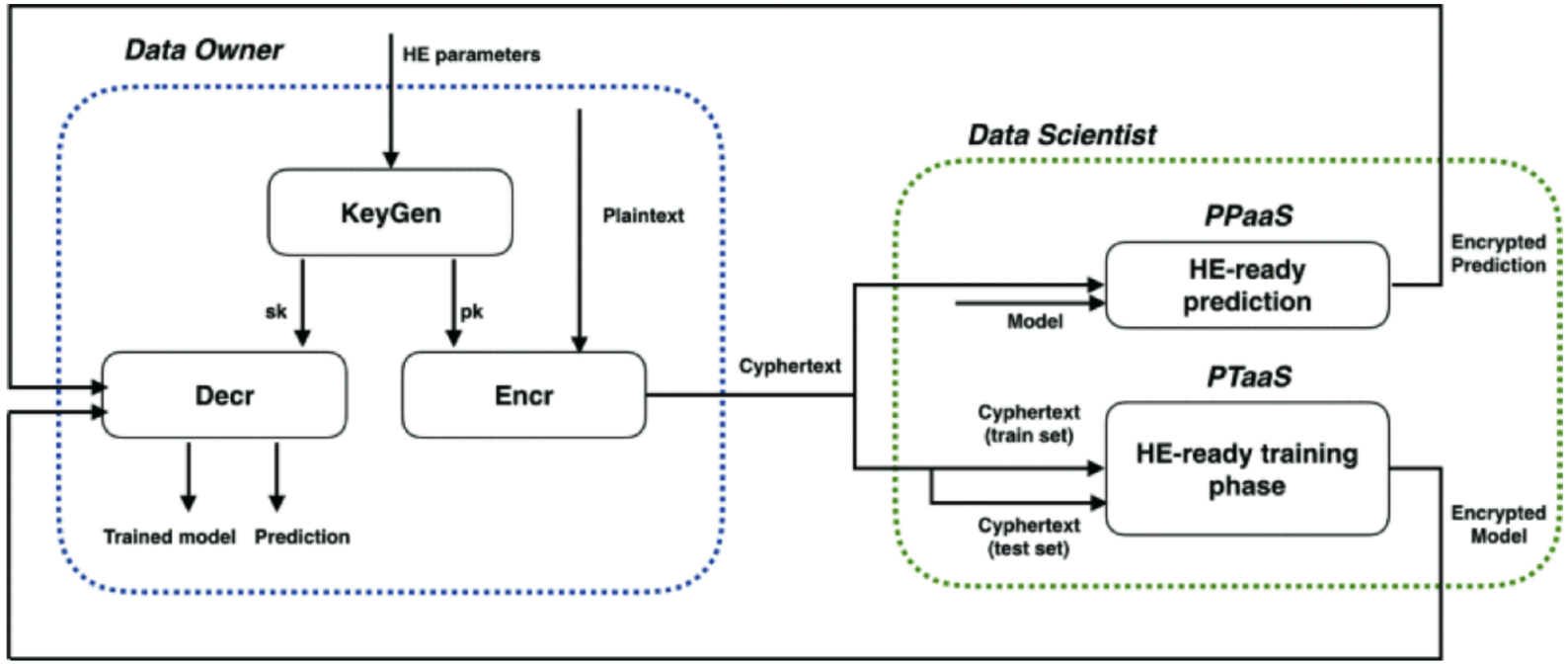
Matrix Operations

Iezzi et al. [87] define two methods of training with homomorphic encryption:

- Private Prediction as a Service (PPaaS). This is where the prediction is outsourced to a service provider who has a pre-trained model and where encrypted data is sent to the service provider. In this case, the data owner does not learn the model used.
- Private Training as a Service (PTaaS). This is where the data owner provides data to a service provider and who will train the model. The service provider can then provide a prediction for encrypted data.

Wood et al [56] adds models of:

- Private outsourced computation. This involves moving computation into the cloud.
- Private prediction. This involves homomorphic data processed into the cloud, and not having access to the training model.
- Private training. This is where a cloud entity trains a model based on the client's data.



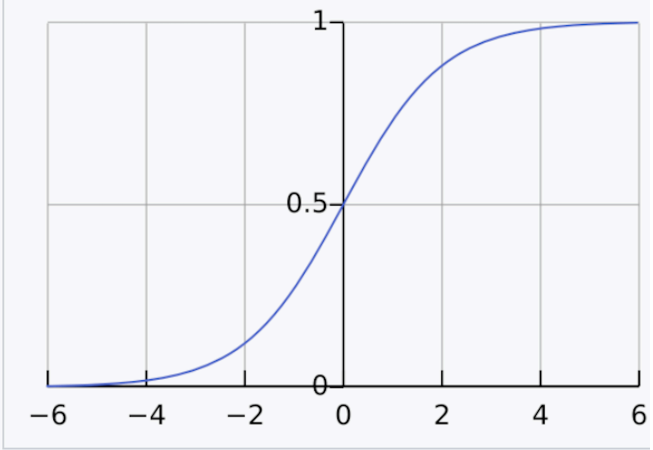
Private Prediction as a Service

Reference	PPaaS Model	HE scheme	Platform	Running Time	Dataset
Graepel et al. [54]	Linear Means Classifier	FV / Bra on SEAL	Intel Core i7 @2.8 GHz with 8GB of RAM	6 sec	Wisconsin Breast Cancer Data
	Fisher's Linear Discriminant classifier			20 sec	
Gilad-Bachrac et al. [32]	Prediction by NN with 5 layers	HE scheme based on [56] on SEAL	Intel Xeon E5-1620 @ 3.5GHz with 16GB RAM	570 sec for a prediction	MNIST
Costantino et al. [45]	Bag-of-word	BGV on HElib	Intel Core i7-6700 @ 3.40GHz with GB RAM	19 min	Tweets
Hesamifard et al. [34]	Prediction by Convolutional Neural Network with 6 layers	LHE on HElib	Intel Xeon E5-2640 @2.4GHz with 16GB RAM	320 sec	MNIST
				11686 sec	CIFAR-10
Masters et al. [31]	Nesterov's Accelerate GD-based logistic regression	CKKS on HElib	Titan V	4500 speed up on mult	Banco Bradesco financial transactions
Brutzkus et al. [33]	Same NN as [32] for prediction	BFV on SEAL	Azure standard VM with 8 vCPUs 32GB of RAM	0.29 sec for a prediction	MNIST
	Linear model trained with features generated by AlexNet			0.16 sec for prediction	CalTech-101
Al Badawi et al. [37]	Fasttext NN trained on plaintext data	CKKS on GPU	NVIDIA DGX-1 multi-GPU with 8 V100 cards	0.66 sec	AGNews
Podschwadt et al. [44]	Embedding layer + RNN layer with 128 units	CKKS on HElib	AMD Ryzen 5 2600 @ 3.5GHz with 32GB RAM.	547.6 sec for a batch of 128 samples	IMDb

[87] M. Iezzi, "Practical privacy-preserving data science with homomorphic encryption: an overview," in 2020 IEEE International Conference on Big Data (Big Data). IEEE, 2020, pp. 3979–3988.

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 $\langle a, b \rangle = a.ycos(\theta)$, 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:

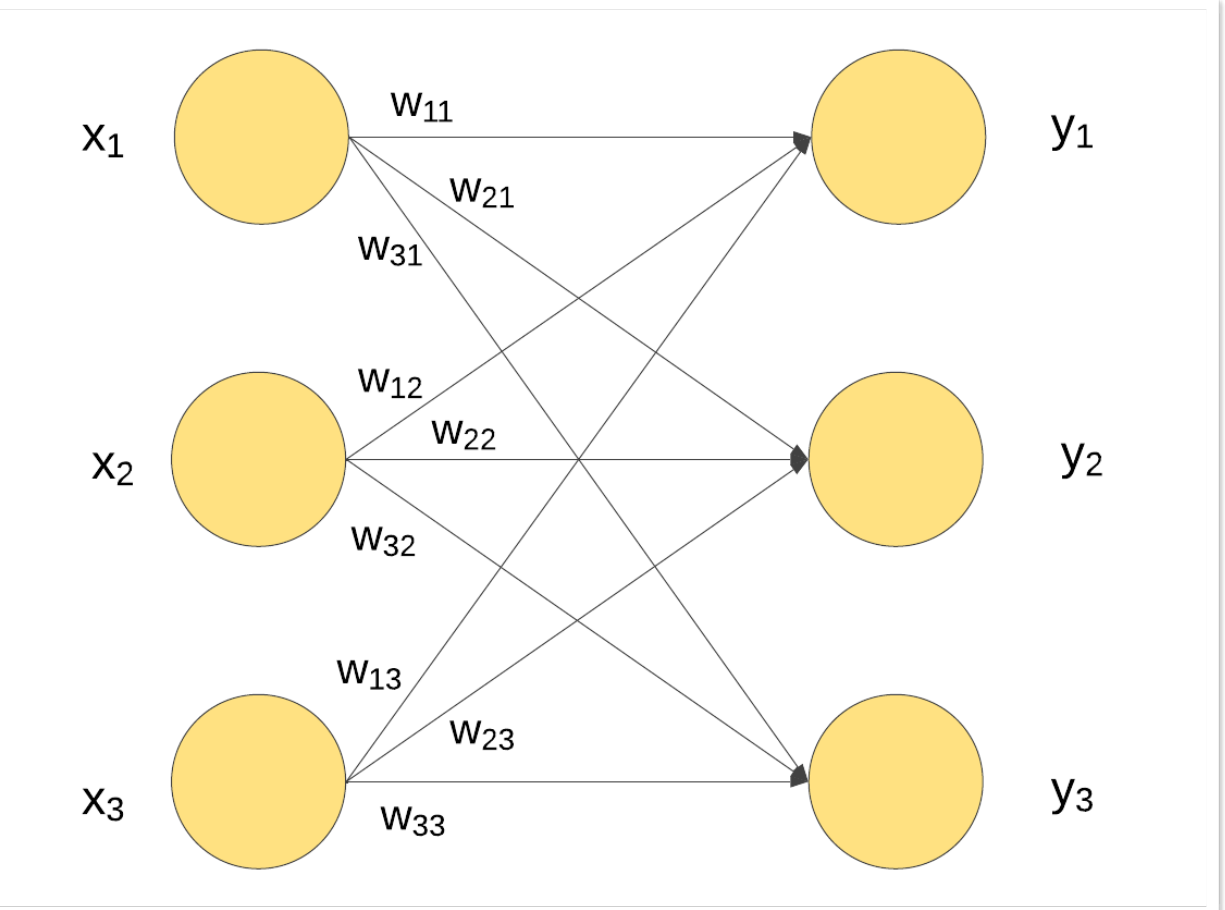
$$\langle a, b \rangle = |a|.\,|b|.\,cos(\theta) = 26.93 \times 26.93.cos(0) = 725$$

```
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:

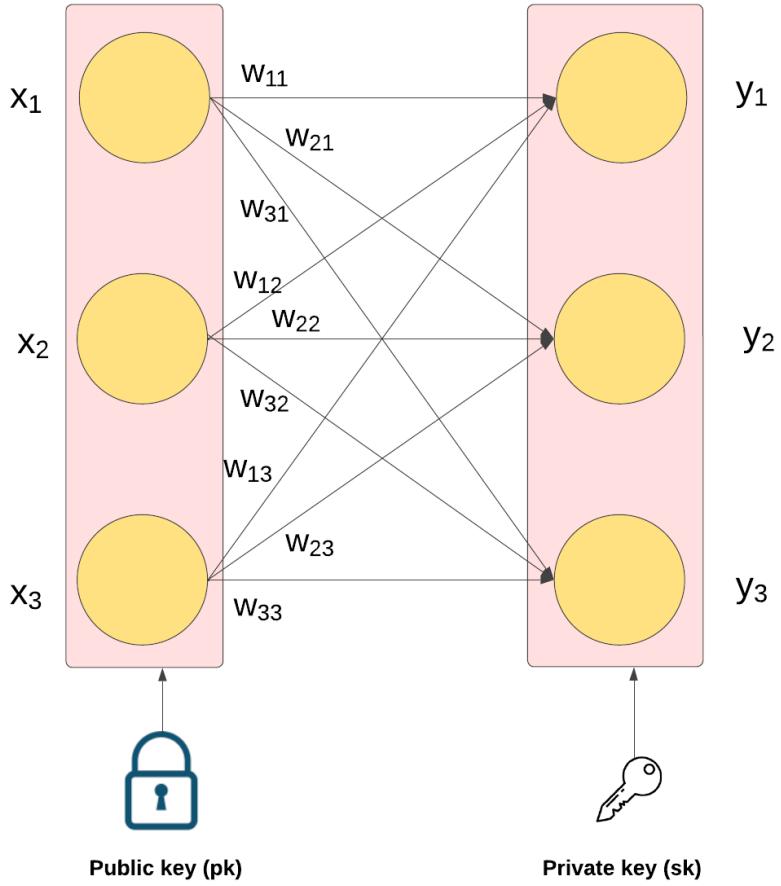
$$y_1 = x_1 . w_{11} + x_2 . w_{21} + x_3 . w_{31}$$

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

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

https://asecuritysite.com/openfhe/openfhe_27cpp

Matrix Operations



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

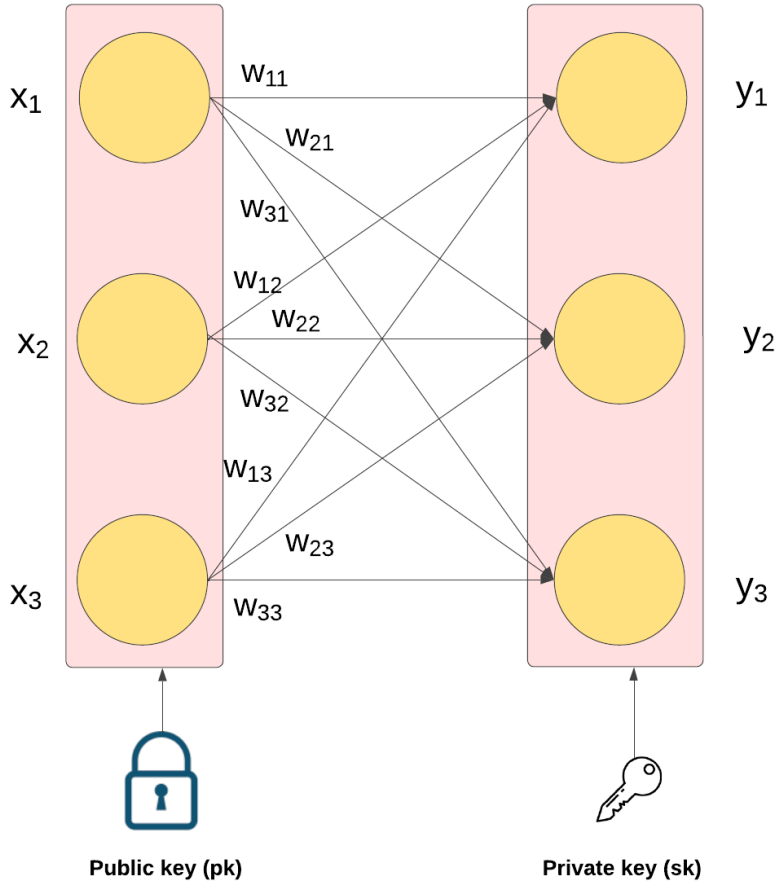
We can then use the private key to determine:

$$\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = Ek_{sk}^{-1}(v_1 . m_1)$$

https://asecuritysite.com/openfhe/openfhe_27cpp

https://asecuritysite.com/openfhe/openfhe_26cpp

Matrix Operations

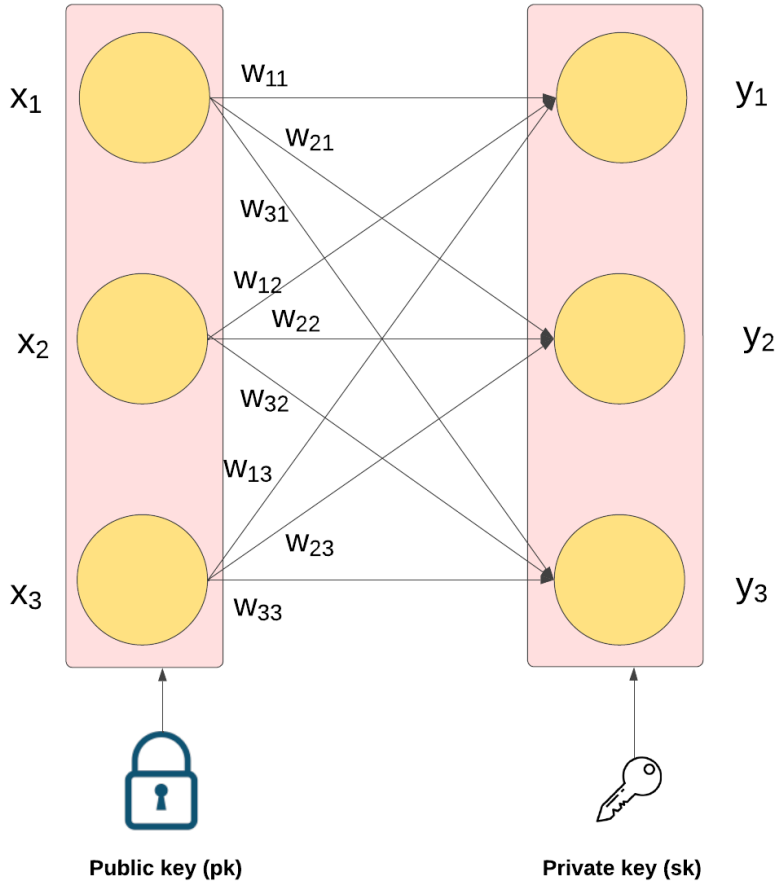



```
std::cout << "Generating rotation keys... ";
std::vector<int32_t> rotationKeys = {};
for (int i = -ROWS*COLS; i <= ROWS*COLS; i++) rotationKeys.push_back(i);
cryptoContext->EvalRotateKeyGen(keyPair.secretKey, rotationKeys);
std::cout << "Done"<< std::endl << std::endl;

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::en
```

https://asecuritysite.com/openfhe/openfhe_27cpp

https://asecuritysite.com/openfhe/openfhe_26cpp

Matrix Operations

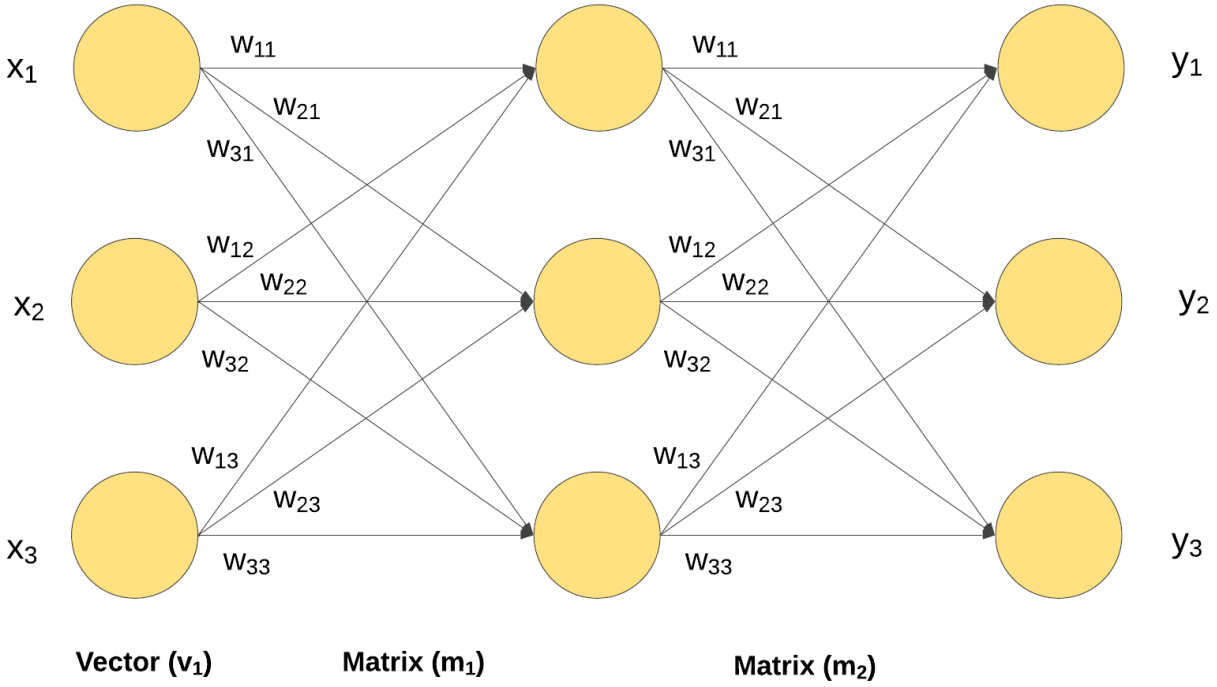



```
Ciphertext<DCRTPoly> vectorC = cryptoContext->Encrypt(keyPair.publicKey, vectorP);

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

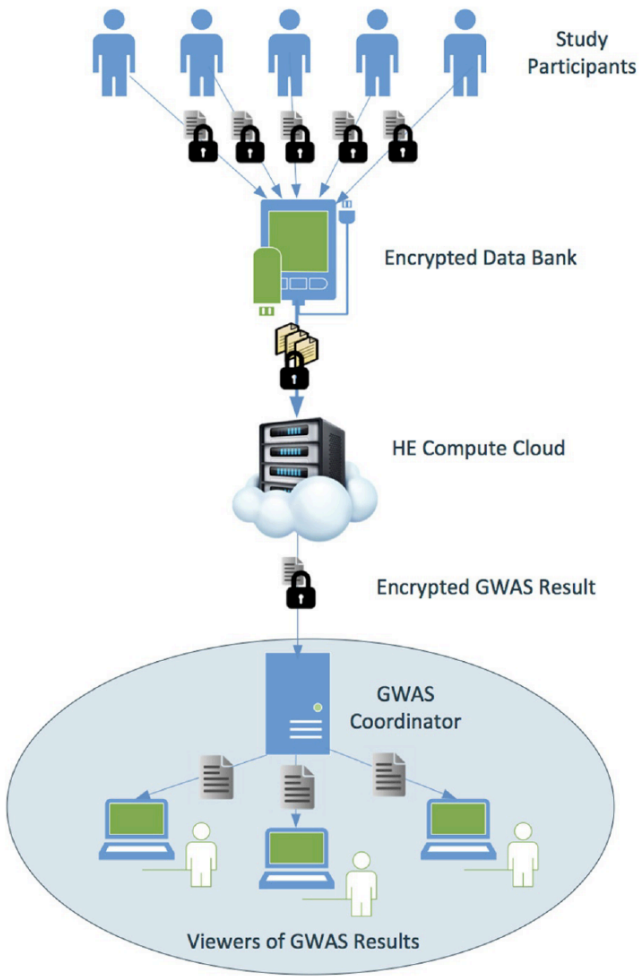



```
Ciphertext<DCRTPoly> vectorC = cryptoContext->Encrypt(keyPair.publicKey, vectorP);

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

GWAS (Gnome-wide Association Studies)



	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]

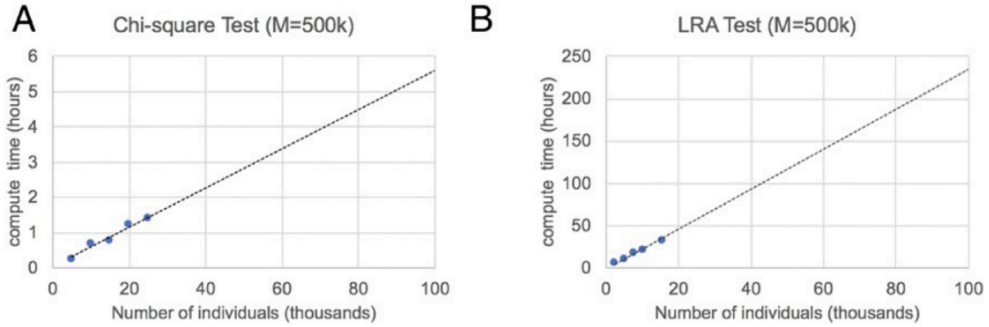
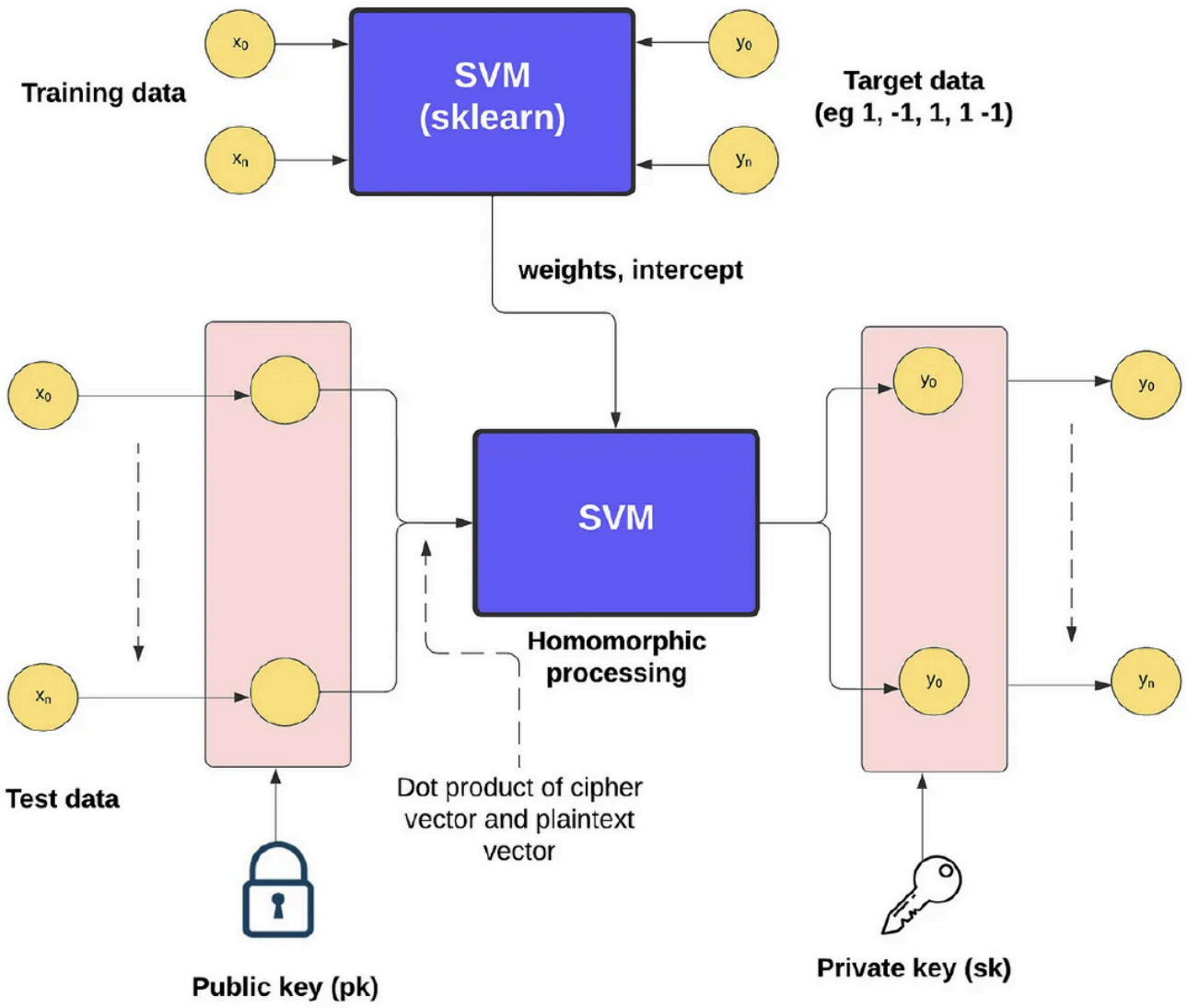


Figure 7.9: Results for GWAS [92]

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
SVM
Homomorphic
processing
y_0
y_0
y_0
y_n
Test data
Dot product of cipher
vector and plaintext
vector
Public key (pk)
Private key (sk)

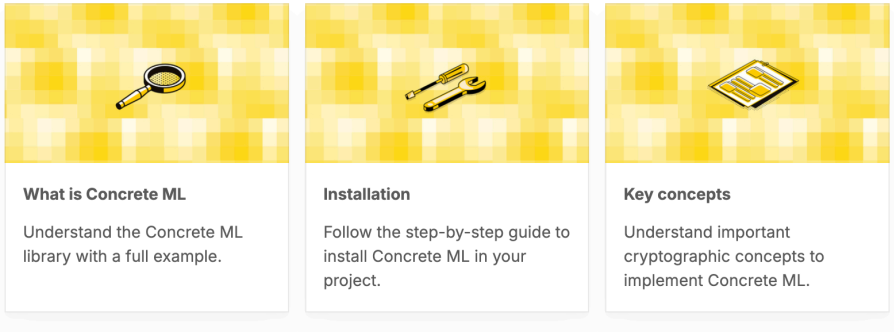
Concrete-ML

Welcome

Concrete ML is an open-source, privacy-preserving, machine learning framework based on Fully Homomorphic Encryption (FHE).

Get started

Learn the basics of Concrete ML, set it up, and make it run with ease.



Concrete-ML also allows for a direct replacement for the following methods:

- Linear models: LinearRegression, LogisticRegression, LinearSVC, LinearSVR PoissonRegressor, TweedieRegressor, GammaRegressor, Lasso, Ridge, ElasticNet and SGDRegressor.

79

- Tree Models: DecisionTreeClassifier, DecisionTreeRegressor, RandomForestClassifier, and RandomForestRegressor.
- NeuralNetClassifier (MLPClassifier) and NeuralNetRegressor (MLPRegressor).
- Nearest Neighbour. KNeighborsClassifier

Future of Digital Trust

Prof Bill Buchanan OBE FRSE

https://asecuritysite.com

Zero Knowledge Proofs
Fiat Shamir, DLEQ, zkSnarks,
Ring Signatures, Range Proofs, Accumulators,
Pedersen Commitments
Homomorphic Encryption:
Basics
Concepts and Code
Data Sharing
Cloud Processing
MPC
Machine Learning

BLOCKPASS IDENTITY LAB

World-leading Collaboration between
Blockpass IDN and Edinburgh Napier University