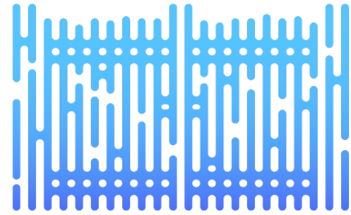


PALISADE

HOMOMORPHIC ENCRYPTION FOR PALISADE USERS: TUTORIAL WITH APPLICATIONS

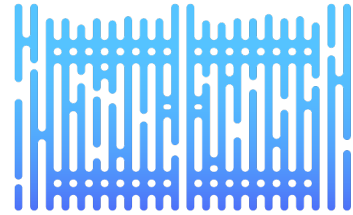
October 30, 2020

Yuriy Polyakov

contact@palisade-crypto.org

HOMOMORPHIC ENCRYPTION FOR PALISADE USERS

- Tutorial with applications consisting of 4 episodes (7 lectures)
- Episode 1
  - Introduction to Homomorphic Encryption
  - Boolean Arithmetic with Applications
- Episode 2
  - Integer Arithmetic
  - Applications of Homomorphic Encryption over Integers
- **Episode 3**
  - Introduction to Multiparty Homomorphic Encryption
- Episode 4
  - Approximate Number Arithmetic
  - Applications of Homomorphic Encryption over Approximate Numbers



PALISADE

HOMOMORPHIC ENCRYPTION FOR PALISADE USERS: TUTORIAL WITH APPLICATIONS

Introduction to Multiparty Homomorphic Encryption

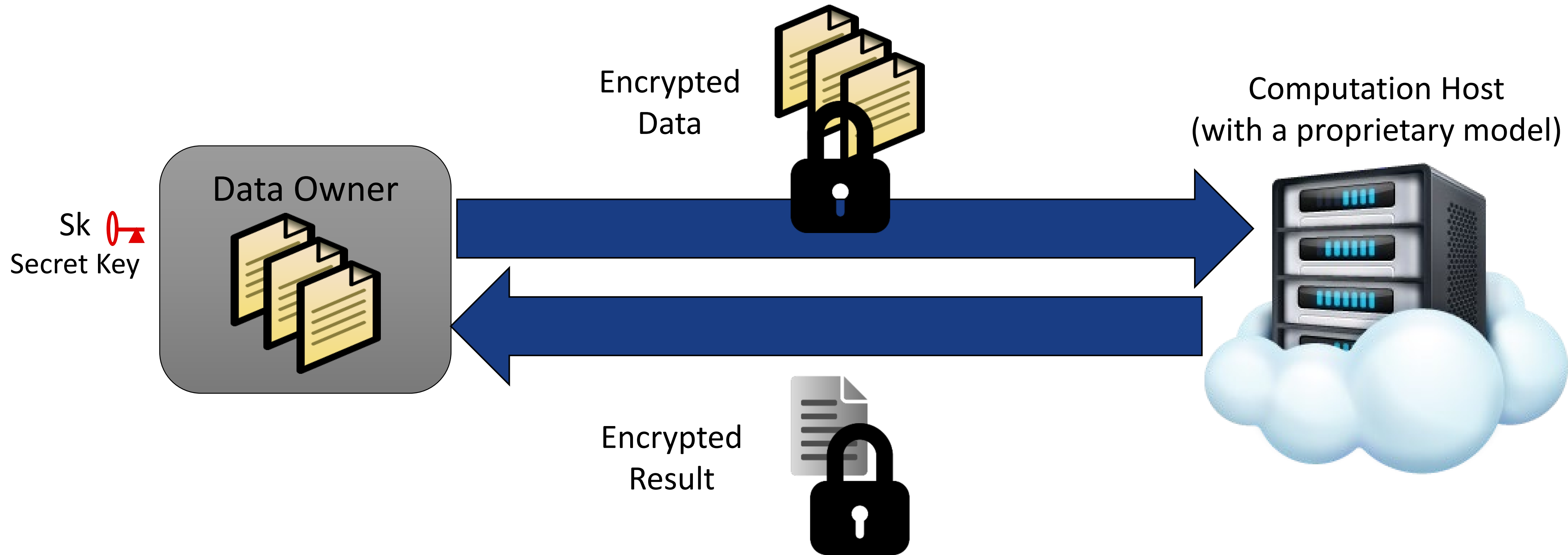
Yuriy Polyakov
ypolyakov@dualitytech.com

AGENDA

- Basics of multiparty HE
  - Why multiparty HE is needed?
  - Two main approaches for multiparty HE
- Threshold HE
  - Key facts
  - Distributed key generation
  - Distributed decryption
- Example of threshold HE in PALISADE

Explains limitations of single-key HE, motivation and main approaches for multiparty HE

SINGLE-KEY HE WORKFLOW

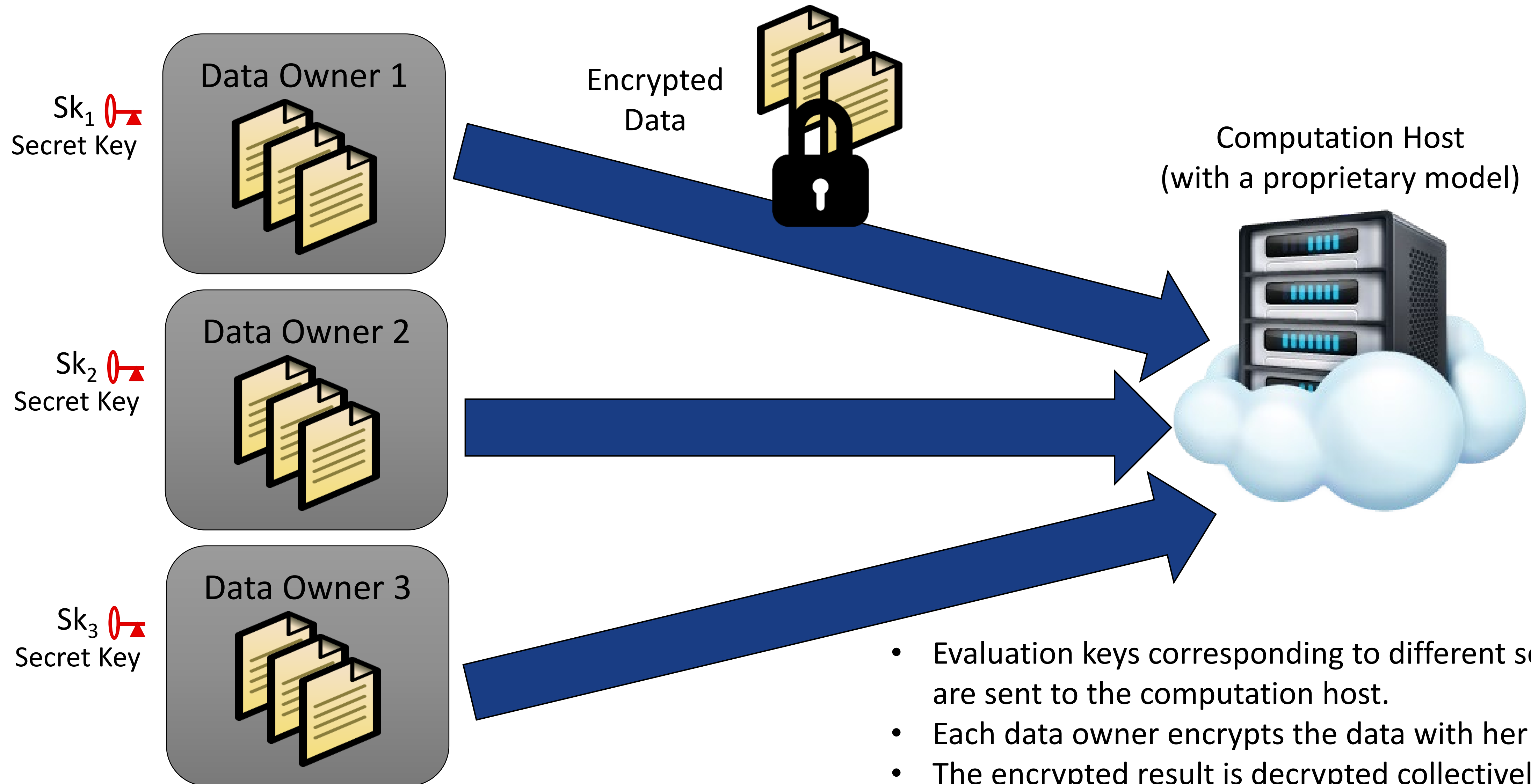


How can this model be extended to multiple data owners that do not want to share a secret key or data?

What if the model needs to be encrypted by model provider and sent to the computation host? What key should the model provider use for encryption?

PALISADE

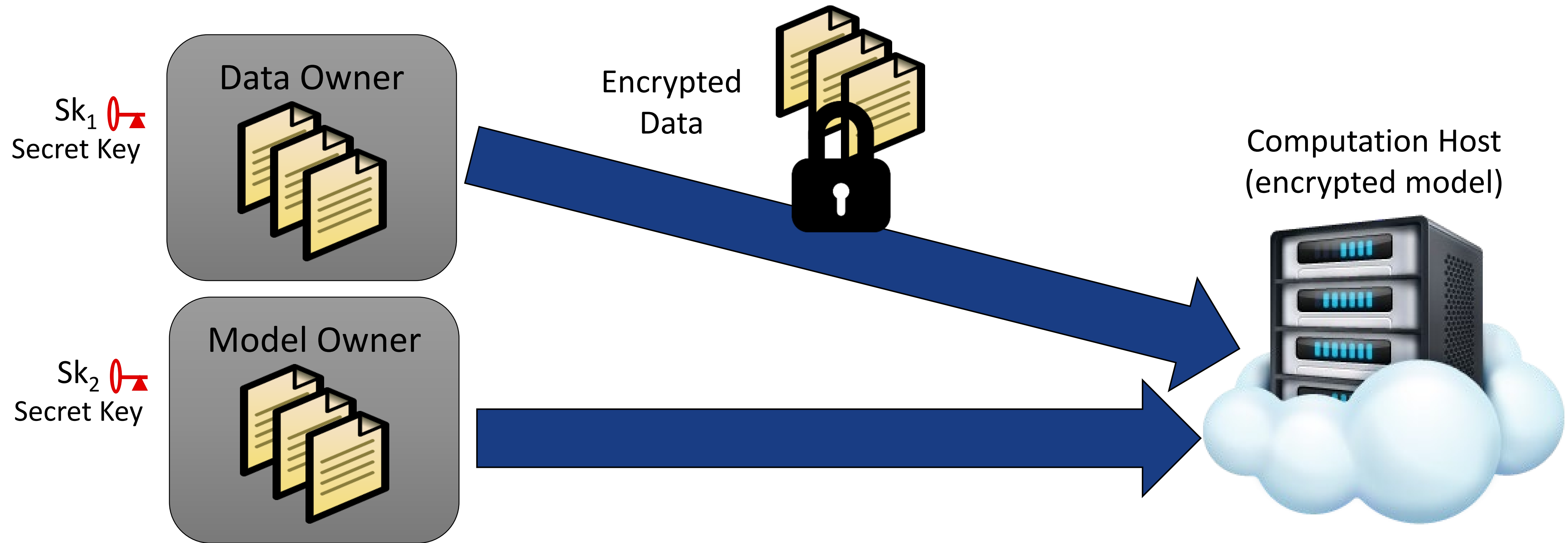
SOLUTION 1: MULTIKEY HE (MULTIPLE DATA OWNERS)



- Evaluation keys corresponding to different secret keys are sent to the computation host.
- Each data owner encrypts the data with her own key.
- The encrypted result is decrypted collectively by all parties (distributed decryption).

PALISADE

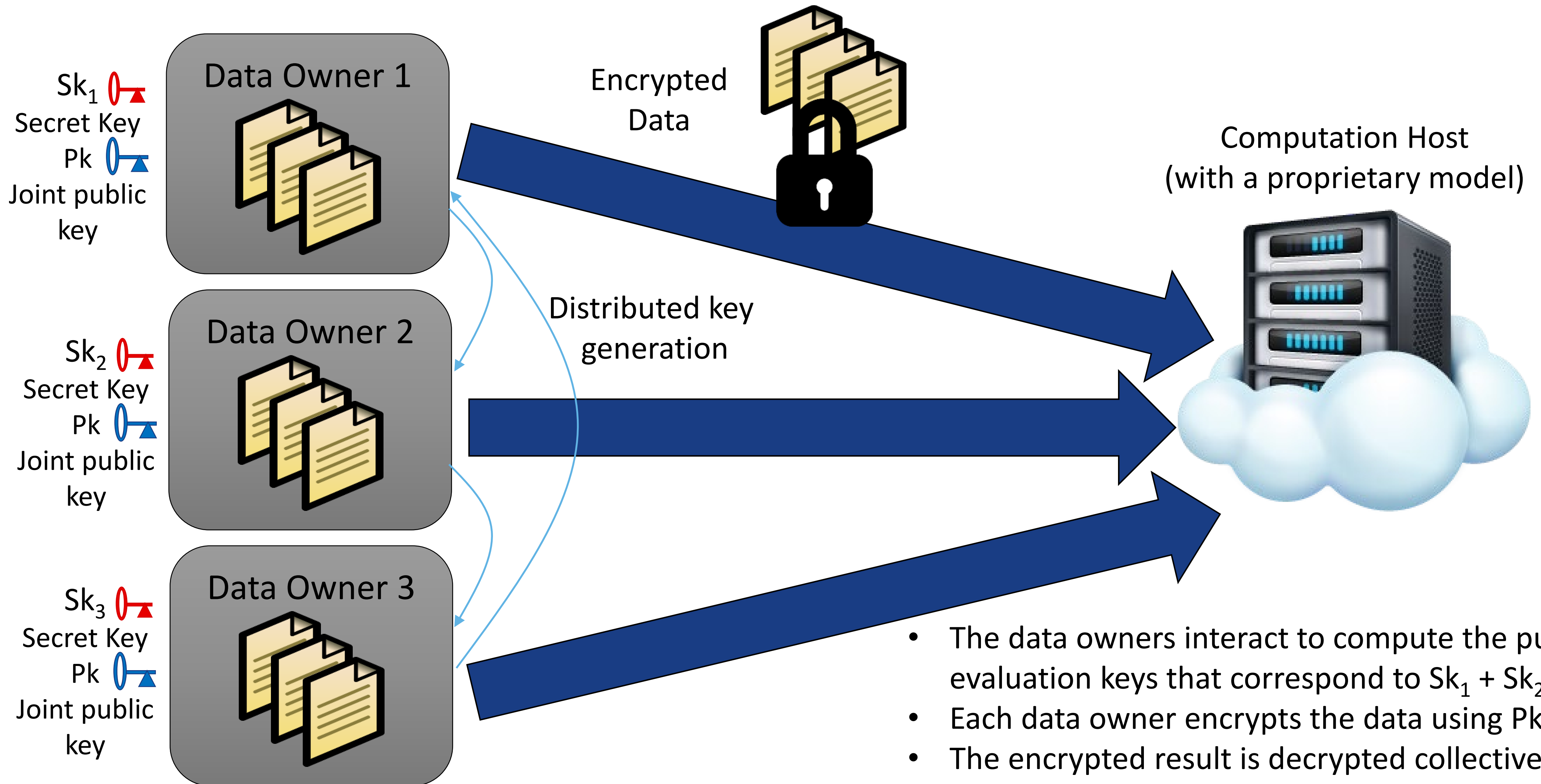
SOLUTION 1: MULTIKEY HE (ENCRYPTED MODEL)



- Evaluation keys corresponding to different secret keys are sent to the computation host.
- Data and model owners encrypt the data with their own keys.
- The encrypted result is decrypted collectively by all parties (distributed decryption).

PALISADE

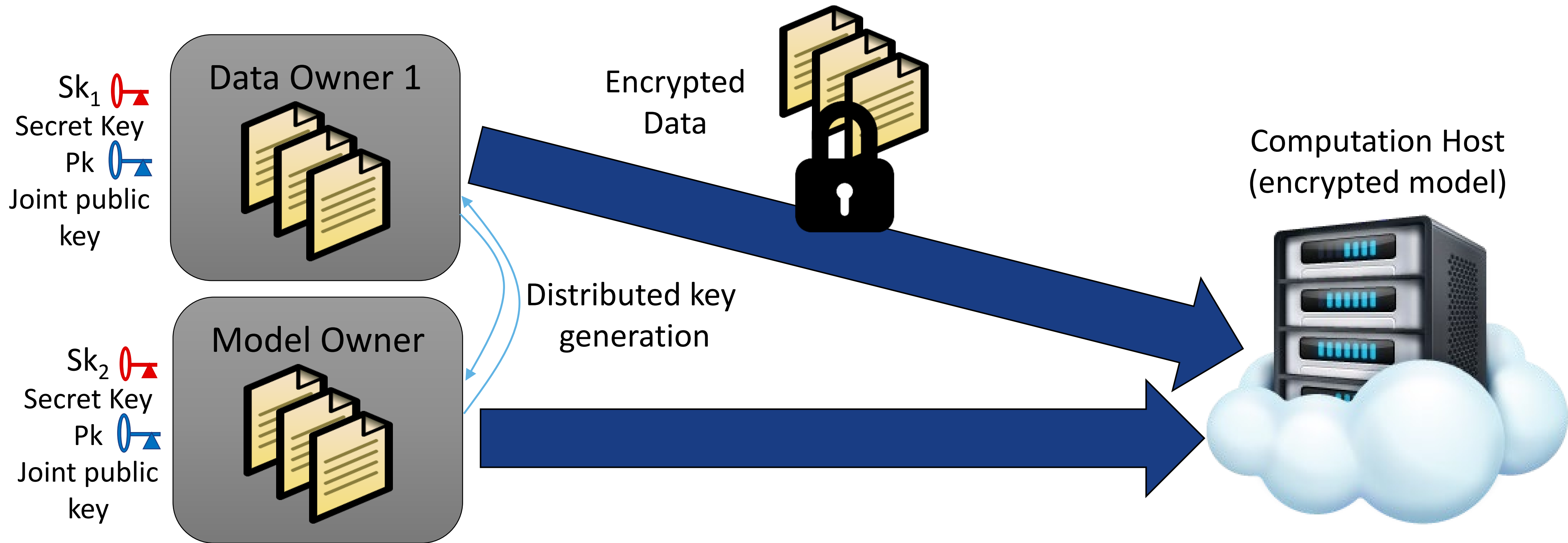
SOLUTION 2: THRESHOLD HE (MULTIPLE DATA OWNERS)



- The data owners interact to compute the public and evaluation keys that correspond to $Sk_1 + Sk_2 + Sk_3$.
- Each data owner encrypts the data using Pk.
- The encrypted result is decrypted collectively by all parties (distributed decryption).

PALISADE

SOLUTION 2: THRESHOLD HE (ENCRYPTED MODEL)



- The data and model owner interact to compute the public and evaluation keys that correspond to Sk_1 + Sk_2.
- Data owner encrypts the data and model owner encrypts the model using Pk.
- The encrypted result is decrypted collectively by both parties (distributed decryption).

PALISADE

COMPARISON OF MULTIKEY AND THRESHOLD HE

Parameter	Multikey HE	Threshold HE
Key generation	Non-interactive (asynchronous)	Interactive (synchronous)
Number of parties	Supports a variable number of parties, bounding only the number of parties involved in a specific computation	The number of parties is fixed
Decryption	Interactive (all parties compute partial decryptions and merge them)	Interactive (all parties compute partial decryptions and merge them)
Computation runtime	Grows quadratically (asymptotically; slightly better in practice) with the number of parties [CDKS19]	Roughly the same as in single-key HE
Evaluation and ciphertext size	Linear in the number of parties [CDKS19]	Roughly the same as in single-key HE

[CDKS19] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient Multi-Key Homomorphic Encryption with Packed Ciphertexts with Application to Oblivious Neural Network Inference. CCS'19.

Explains distributed key generation and decryption

KEY FACTS ABOUT THRESHOLD HE

- All parties interact to generate joint public key and evaluation keys, using a share of the secret key held by each party
- The underlying "full" secret key, which is the sum of secret shares, is never revealed to any party
- The joint public key is then used to perform encrypted computations using FHE (same way as in classical single-key FHE)
- The result is decrypted using an interactive procedure where all parties collaborate to get the result in the clear
- Supported for BGV, BFV, and CKKS schemes in PALISADE

DISTRIBUTED PUBLIC KEY GENERATION (2-PARTY)

- Party 1 generates a secret share Sk_1 and public share Pk_1 from it. A uniform ring element (polynomial) **a** is used in generating Pk_1.
- Party 2 generates a secret share Sk_2 and public share Pk_2 from it. Same uniform ring element **a** is used in generating Pk_2.
- Parties 1 and 2 exchange their public shares. The joint public key Pk = (**a**, Pk_1 + Pk_2) is computed.
  - Pk corresponds to the "full" secret key Sk = Sk_1 + Sk_2. Sk is never available in the clear for any party!
- For *m* parties, *m* - 1 rounds of communication are needed using the sequential topology

DISTRIBUTED ROTATION KEY GENERATION (2-PARTY)

- The goal is to compute an evaluation key that switches from $Rot_i(Sk_1 + Sk_2)$ to $Sk_1 + Sk_2$
- Party 1 generates a rotation key share ($\mathbf{A}$, $\mathbf{B}_1$), where **A** is a matrix of uniform ring elements (polynomials) and $\mathbf{B}_1$ corresponds to the desired rotation by *i* of Sk1.
- Party 2 generates a rotation key share ($\mathbf{A}$, $\mathbf{B}_2$), where **A** is the same matrix of uniform ring elements (polynomials) and $\mathbf{B}_2$ corresponds to the desired rotation by *i* of Sk2.
- Parties A and B exchange their rotation key shares. The joint rotation key Rk = ($\mathbf{A}$, $\mathbf{B}_1 + \mathbf{B}_2$) is computed.
  - Rk corresponds to the “full” secret key $Sk = Sk_1 + Sk_2$. Sk is never available in the clear for any party!
- For *m* parties, *m* - 1 rounds of communication are needed using the sequential topology.

PALISADE

DISTRIBUTED MULTIPLICATION KEY GENERATION (2-PARTY)

- The goal is to compute an evaluation key that switches from $(Sk_1 + Sk_2)^2$ to $Sk_1 + Sk_2$, i.e., an encryption of $(Sk_1 + Sk_2)^2$ under $Sk_1 + Sk_2$
- Round 1
  - Party 1 generates a multiplication key share ($\mathbf{A}$, $\mathbf{B}_1$), where **A** is a matrix of uniform ring elements (polynomials) and $\mathbf{B}_1$ corresponds to the encryption of Sk1.
- Round 2
  - Party 2 generates a multiplication key share ($\mathbf{A}$, $\mathbf{B}_2$), where **A** is the same matrix of uniform ring elements (polynomials) and $\mathbf{B}_2$ corresponds to the encryption of Sk_2.
  - Party 2 combines the multiplication shares to get ($\mathbf{A}$, $\mathbf{B} = \mathbf{B}_1 + \mathbf{B}_2$), which corresponds to $Sk_1 + Sk_2$
  - Using ($\mathbf{A}$, $\mathbf{B}$), Party 2 computes new multiplication share ($\mathbf{C}_2$, $\mathbf{D}_2$), which corresponds to $Sk_2*(Sk_1 + Sk_2)$
  - Party 2 sends **B** and ($\mathbf{C}_2$, $\mathbf{D}_2$) to Party 1
- Round 3
  - Using ($\mathbf{A}$, $\mathbf{B}$), Party 1 computes new multiplication share ($\mathbf{C}_1$, $\mathbf{D}_1$), which corresponds to $Sk_1*(Sk_1 + Sk_2)$
  - Party 1 computes ($\mathbf{C} = \mathbf{C}_1 + \mathbf{C}_2$, $\mathbf{D} = \mathbf{D}_1 + \mathbf{D}_2$), which is the desired multiplication evaluation key corresponding to $(Sk_1 + Sk_2)^2$
  - Party 1 sends ($\mathbf{C}$, $\mathbf{D}$) to Party 2.
- For *m* parties, $2m – 1$ rounds are needed using the sequential topology.

PALISADE

DISTRIBUTED DECRYPTION (2-PARTY)

- Ciphertext ($\mathbf{c}_1$, $\mathbf{c}_2$) is the input
- Party 1 computes a "lead" partial decryption of ($\mathbf{c}_1$, $\mathbf{c}_2$) using Sk_1, $\mathbf{c}_1$, and $\mathbf{c}_2$
- Party 2 computes a partial decryption of ($\mathbf{c}_1$, $\mathbf{c}_2$) using Sk_2 and $\mathbf{c}_2$
- Both partial decryptions are added up (fused) to get the desired decryption result
- For *m* parties, *m* - 1 rounds of communication are needed using the sequential topology

Simple example for 2-party threshold HE

STEP 1 - SET CRYPTOCONTEXT

```
int plaintextModulus = 65537;
double sigma = 3.2;
SecurityLevel securityLevel = HEStd_128_classic;
uint32_t depth = 2;

CryptoContext<DCRTPoly> cc =
    CryptoContextFactory<DCRTPoly>::genCryptoContextBFVrns(
        encodingParams, securityLevel, sigma, 0, 2, 0, OPTIMIZED);

// enable features that you wish to use
cc->Enable(ENCRYPTION);
cc->Enable(SHE);
cc->Enable(MULTIPARTY);
```

STEP 2 – KEY GENERATION – ROUND 1

```
// Initialize Public Key Containers for two parties A and B
LPKeyPair<DCRTPoly> kp1; LPKeyPair<DCRTPoly> kp2;
LPKeyPair<DCRTPoly> kpMultiparty;

// Round 1 (party A) started

// Generate secret key for party A
kp1 = cc->KeyGen();

// Generate evalmult key part for A
auto evalMultKey = cc->KeySwitchGen(kp1.secretKey, kp1.secretKey);

// Generate evalsum key part for A
cc->EvalSumKeyGen(kp1.secretKey);
auto evalSumKeys = std::make_shared<std::map<usint, LPEvalKey<DCRTPoly>>>(
       cc->GetEvalSumKeyMap(kp1.secretKey->GetKeyTag()));

// Round 1 of key generation completed.
```


PALISADE

STEP 3 – KEY GENERATION – ROUND 2

```
// Round 2 (party B) started.
// Joint public key for (s_a + s_b) is generated…
kp2 = cc->MultipartyKeyGen(kp1.publicKey);

// Generate evalmult key part for B
auto evalMultKey2 = cc->MultiKeySwitchGen(kp2.secretKey, kp2.secretKey, evalMultKey);

// Joint evaluation multiplication key for (s_a + s_b) is generated...
auto evalMultAB = cc->MultiAddEvalKeys(evalMultKey, evalMultKey2, kp2.publicKey->GetKeyTag());

// Joint evaluation multiplication key (s_a + s_b) is transformed into s_b*(s_a + s_b)...
auto evalMultBAB = cc->MultiMultEvalKey(evalMultAB, kp2.secretKey, kp2.publicKey->GetKeyTag());

// Generate evalsum key part for B
auto evalSumKeysB = cc->MultiEvalSumKeyGen(kp2.secretKey, evalSumKeys, kp2.publicKey->GetKeyTag());

// Joint evaluation summation key for (s_a + s_b) is generated... evalsum key part for B
auto evalSumKeysJoin = cc->MultiAddEvalSumKeys(evalSumKeys, evalSumKeysB, kp2.publicKey->GetKeyTag());

cc->InsertEvalSumKey(evalSumKeysJoin);
// Round 2 of key generation completed.
```

STEP 4 – KEY GENERATION – ROUND 3

```
// Round 3 (party A) started.

// Joint key (s_a + s_b) is transformed into s_a*(s_a + s_b)...
auto evalMultAAB = cc->MultiMultEvalKey(evalMultAB, kp1.secretKey,
                                        kp2.publicKey->GetKeyTag());

// Computing the final evaluation multiplication key for (s_a + s_b)*(s_a + s_b)...
auto evalMultFinal = cc->MultiAddEvalMultKeys(evalMultAAB, evalMultBAB,
                                              evalMultAB->GetKeyTag());

cc->InsertEvalMultKey({evalMultFinal});

// Round 3 of key generation completed.
```


PALISADE

STEP 5: ENCRYPTION

```
//////////////////////////////////////////////////////////////
 // Encode source data
 //////////////////////////////////////////////////////////////
 std::vector<int64_t> vectorOfInts1 = {1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1, 0};
 std::vector<int64_t> vectorOfInts2 = {1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0};
 std::vector<int64_t> vectorOfInts3 = {2, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 0};

 Plaintext plaintext1 = cc->MakePackedPlaintext(vectorOfInts1);
 Plaintext plaintext2 = cc->MakePackedPlaintext(vectorOfInts2);
 Plaintext plaintext3 = cc->MakePackedPlaintext(vectorOfInts3);

 //////////////////////////////////////////////////////////////
 // Encryption
 //////////////////////////////////////////////////////////////
 auto ciphertext1 = cc->Encrypt(kp2.publicKey, plaintext1);
 auto ciphertext2 = cc->Encrypt(kp2.publicKey, plaintext2);
 auto ciphertext3 = cc->Encrypt(kp2.publicKey, plaintext3);
```

STEP 6: HOMOMORPHIC COMPUTATIONS

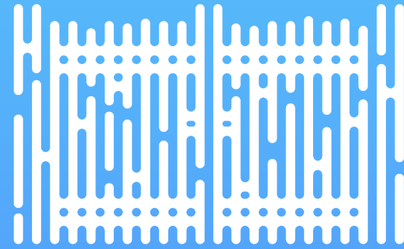
```
////////////////////////////////////////////////////////////
// Homomorphic Operations
////////////////////////////////////////////////////////////

auto ciphertextAdd12 = cc->EvalAdd(ciphertext1, ciphertext2);
auto ciphertextAdd123 = cc->EvalAdd(ciphertextAdd12, ciphertext3);

auto ciphertextMult = cc->EvalMult(ciphertext1, ciphertext3);
auto ciphertextEvalSum = cc->EvalSum(ciphertext3, 1024);
```


PALISADE

STEP 7: DISTRIBUTED DECRYPTION

```
// Distributed decryption

// partial decryption by party A
auto ciphertextPartial1 = cc->MultipartyDecryptLead(kp1.secretKey, {ciphertextAdd123});

// partial decryption by party B
auto ciphertextPartial2 = cc->MultipartyDecryptMain(kp2.secretKey, {ciphertextAdd123});

vector<Ciphertext<DCRTPoly>> partialCiphertextVec;
partialCiphertextVec.push_back(ciphertextPartial1[0]);
partialCiphertextVec.push_back(ciphertextPartial2[0]);

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

Source for this example: https://gitlab.com/palisade/palisade-release/-/blob/master/src/pke/examples/**threshold-fhe.cpp**



THANK YOU

ypolyakov@dualitytech.com