

Fully Homomorphic Encryption Development Ecosystems

Tools, Compilers & Challenges

Alexander Viand*, Patrick Jattke, Anwar Hithnawi

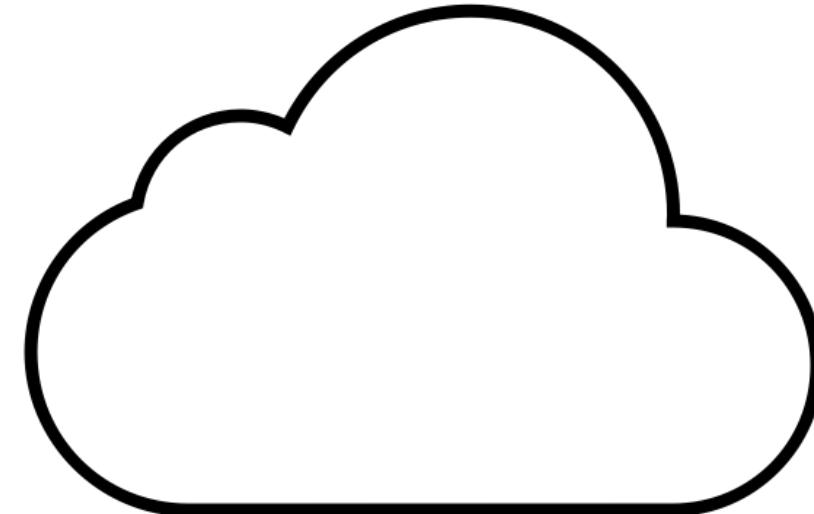
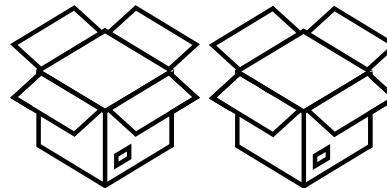


*alexander.viand@inf.ethz.ch

What is FHE?

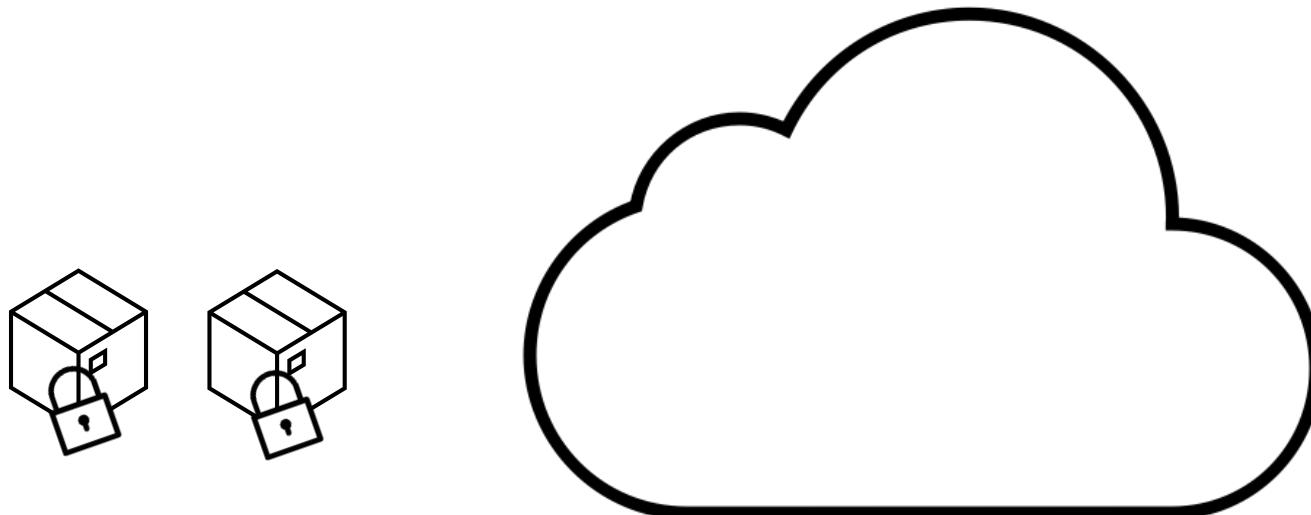
- Allows data to be processed while remaining encrypted

3 | 6



What is FHE?

- Allows data to be processed while remaining encrypted

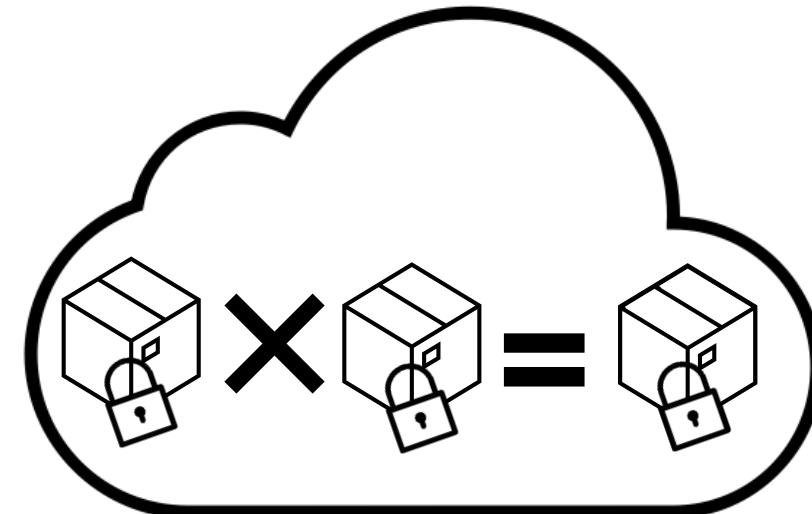


What is FHE?

- Allows data to be processed while remaining encrypted

$$18 = \boxed{3} \times \boxed{6}$$

A diagram illustrating a multiplication operation. On the left, the equation $18 = \boxed{3} \times \boxed{6}$ is shown above two stacked boxes. A key icon is positioned below the boxes. To the right, a cloud-shaped container holds two locked boxes (one labeled with a multiplication sign) and a third box labeled with an equals sign, representing the result of the multiplication.



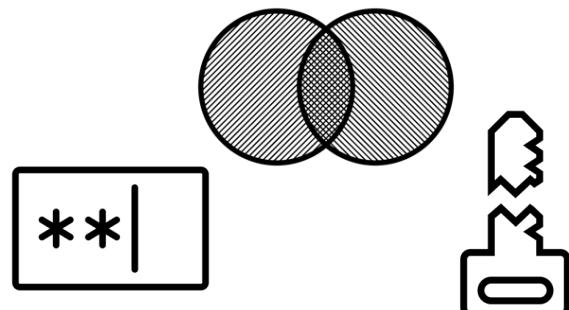
- Secure Outsourcing



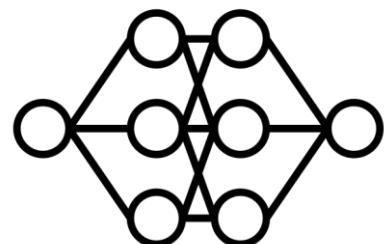
Is a gene mutation
associated to a disease?



- Private Set Intersection (PSI)



- Private Machine Learning as a Service

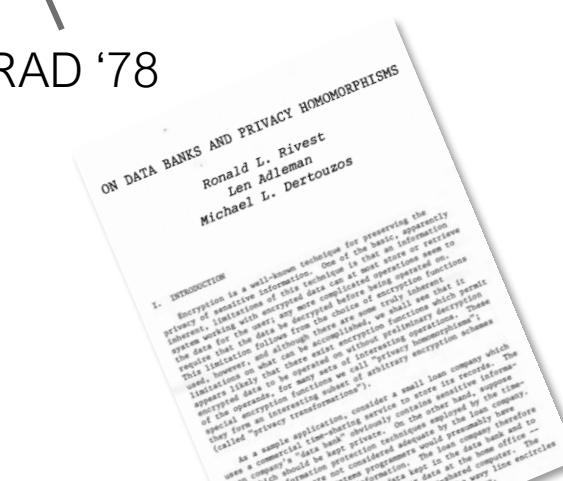
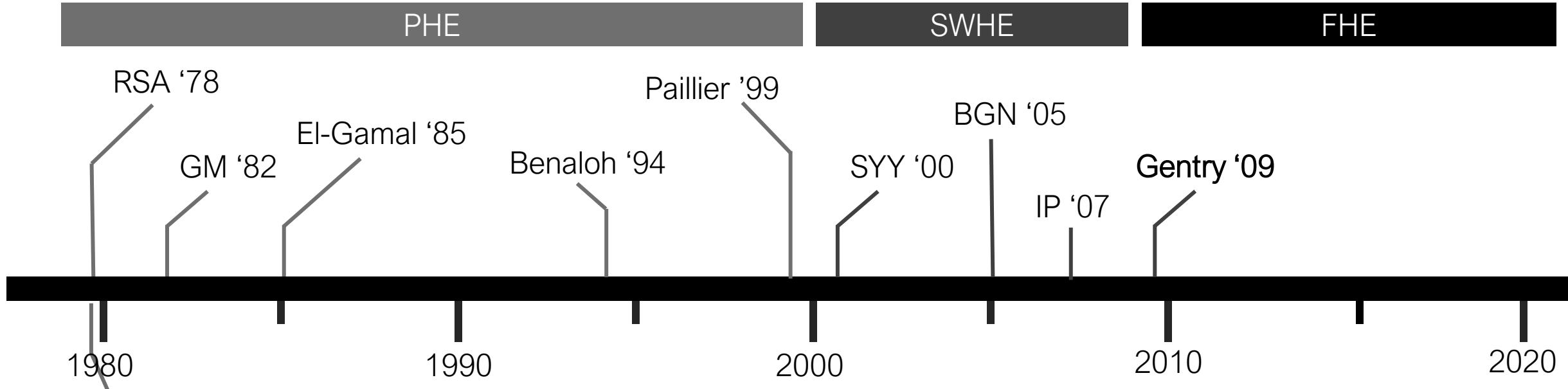


n
NuCypher

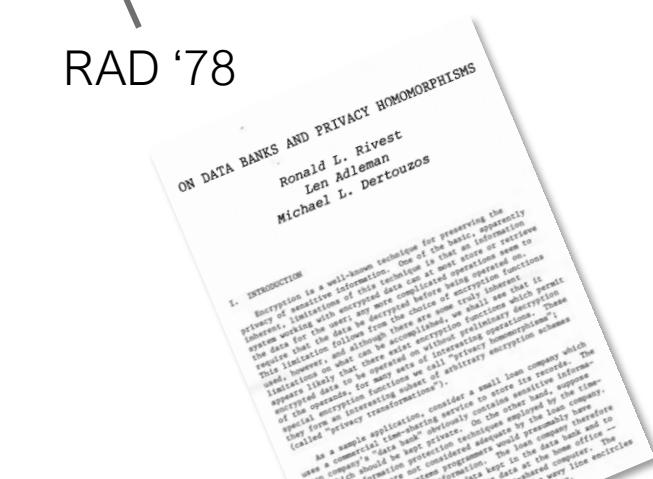
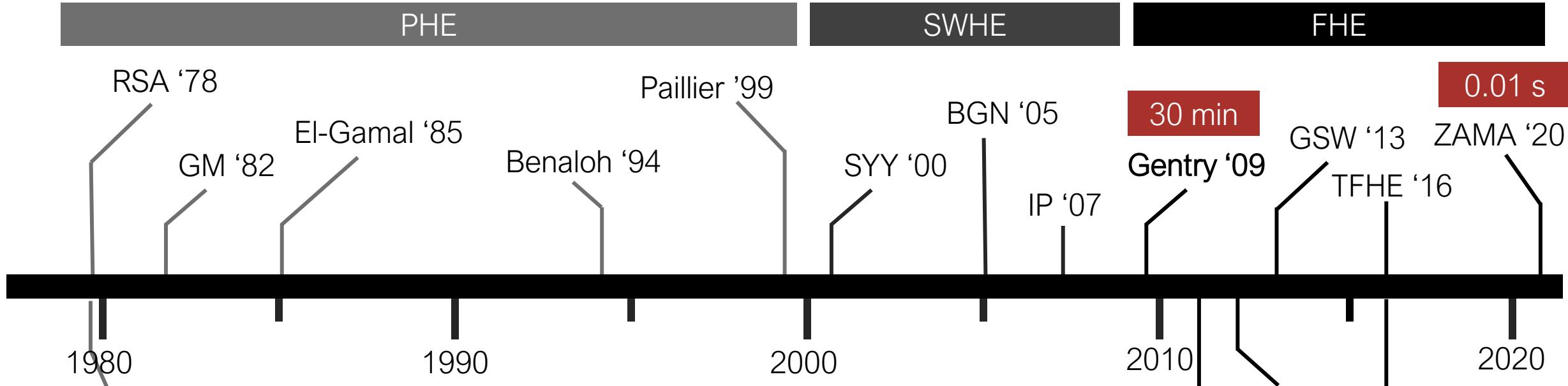
ZAMA

Duality

40 Years of FHE History

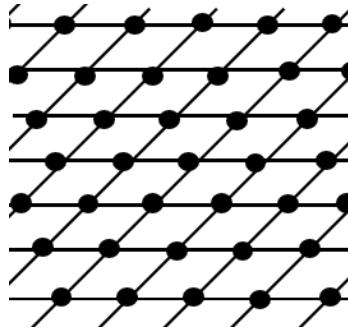


40 Years of FHE History



The path to adoption

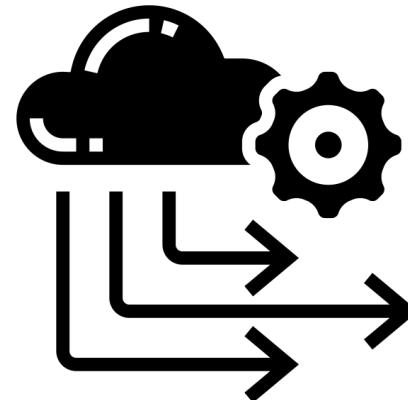
Underlying Math



Development



Deployment



← Cryptographic Challenges →

← Engineering Challenges →

Stack Overflow

How to use CRT batch technique in Microsoft SEAL 3.1?

Can you please tell me whether SEAL 3.1 supports PolyCRTBuilder class for encrypting? I am trying the following program but failed because the class is not declared in this scope.

```
/** Suppose I have two arrays x = [1,2,3,4,5] and xMean = [3,3,3,3,3] generated as follows. I want to subtract the two array using PolyCRTBuilder (xCiphertext and xMeanCiphertext) . If I subtract them directly, the result is xCiphertext MINUS xMeanCiphertext , I should get xResult = [-2, -1, 0, 1, 2] but after the homomorphic multiplication I am getting xResultDecrypted = [40959, 40960, 0, 1, 2] . I can relate the overflow from the plain text modulus set but is there a work around for this problem. Here is the code: */
```

#include <iostream>

Stack Overflow

Truncate in Homomorphic Encryption

How do you implement truncation in homomorphic encryption libraries like HElib or SEAL when no division operation is allowed?

I have two floating point numbers $a=2.3, b=1.5$ which scale to integers with 2-digit precision. Hence my encoder looks something like this $\text{encode}(x)=x \times 10^2$. Assuming $\text{enc}(x)$ is the encryption function, then $\text{enc}(\text{encode}(a))=\text{enc}(230)$ and $\text{enc}(\text{encode}(b))=\text{enc}(150)$.

Upon multiplication we obtain the huge value of $a \times b = \text{enc}(23 \times 15) = \text{enc}(3450)$ because the scaling factors multiply too. This means that my inputs grow exponentially unless I can truncate the result, so that $\text{truncate}(\text{enc}(3450))=\text{truncate}(\text{enc}(345))$.

Answer

What makes developing FHE applications hard

Compare

Design decisions in existing FHE compilers and how they address some of FHE's complexities

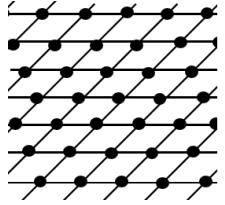
Highlight

Where barriers to entry have been lowered and where they still remain

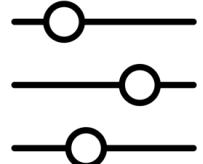
Outline

Future directions for FHE tool development

Challenges of Developing FHE Applications



Underlying Math

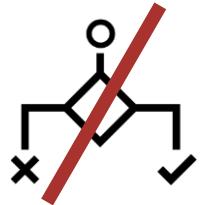
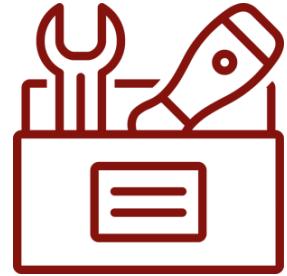


Parameter Selection

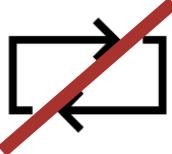


Security

Libraries



No If/Else



No Loops

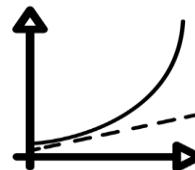


No Jumps

Compilers



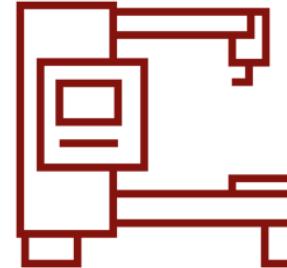
Optimizations



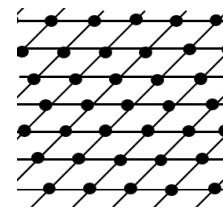
Approximations



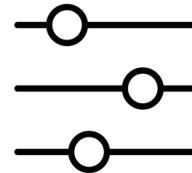
SIMD Batching



Cryptographic Challenges



Underlying Math

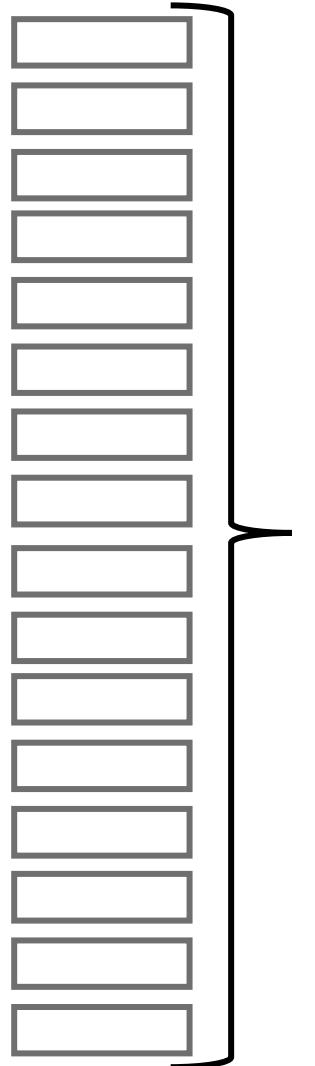


Parameter Selection



Security

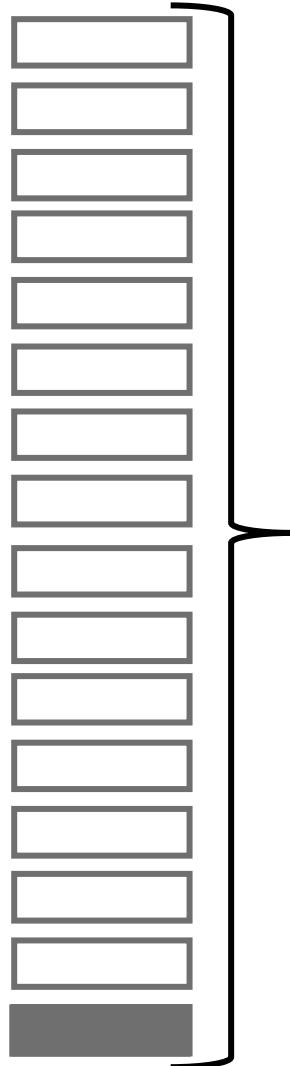
Cryptographic Challenges



(R)ing-)Learning W~~i~~th E~~r~~rors: It is hard to find s given c and a

$$c = a \cdot s + e \quad \text{where } a \xleftarrow{\$} R_q, e \xleftarrow{\chi} R_q, s \in R_q$$
$$R = \mathbb{Z}[X]/(X^n + 1)$$

Cryptographic Challenges



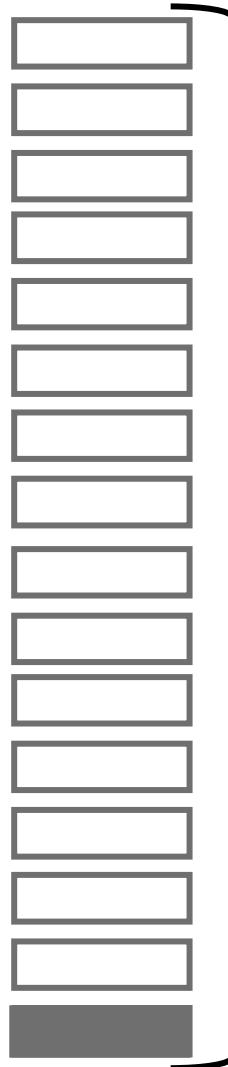
(R)ing-)Learning WWith EErrors: It is hard to find s given c and a

$$c = a \cdot s + e$$

where $a \xleftarrow{\$} R_q, e \xleftarrow{\chi} R_q, s \in R_q$
 $R = \mathbb{Z}[X]/(X^n + 1)$



Cryptographic Challenges



(R)ing-)Learning W~~i~~th E~~r~~rors: It is hard to find s given c and a

$$c = a \cdot s + e$$

where $a \xleftarrow{\$} R_q, e \xleftarrow{\chi} R_q, s \in R_q$
 $R = \mathbb{Z}[X]/(X^n + 1)$

Encryption

$$c = a \cdot s + \mu + e$$

where $\mu = m * \left\lfloor \frac{q}{t} \right\rfloor$ for $m \in R_t, t \ll q$



$\log t$

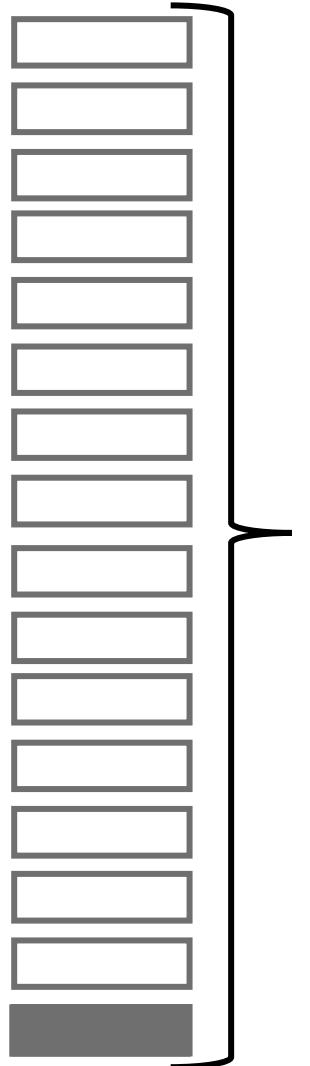
$\log q$

m

e

$[2^{\log q - \log t}]$

Cryptographic Challenges



(R)ing-)Learning W~~i~~th E~~r~~rors: It is hard to find s given c and a

$$c = a \cdot s + e$$

where $a \xleftarrow{\$} R_q, e \xleftarrow{\chi} R_q, s \in R_q$
 $R = \mathbb{Z}[X]/(X^n + 1)$

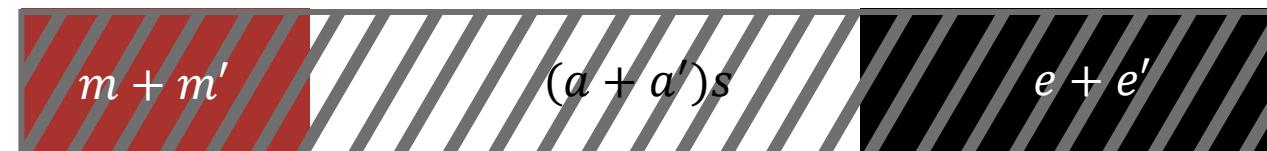
Encryption

$$c = a \cdot s + \mu + e$$

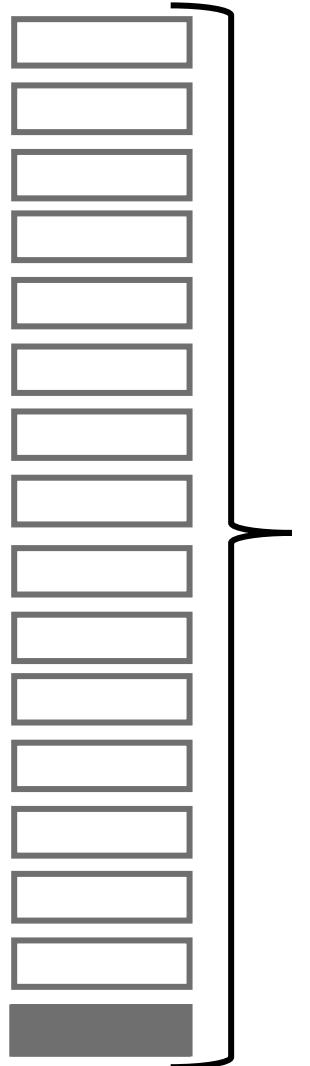
where $\mu = m * \left\lfloor \frac{q}{t} \right\rfloor$ for $m \in R_t, t \ll q$

Homomorphic Addition

- Noise increases linearly



Cryptographic Challenges



(R)ing-)Learning W~~i~~th E~~r~~rors: It is hard to find s given c and a

$$c = a \cdot s + e$$

where $a \xleftarrow{\$} R_q, e \xleftarrow{\chi} R_q, s \in R_q$
 $R = \mathbb{Z}[X]/(X^n + 1)$

Encryption

$$c = a \cdot s + \mu + e$$

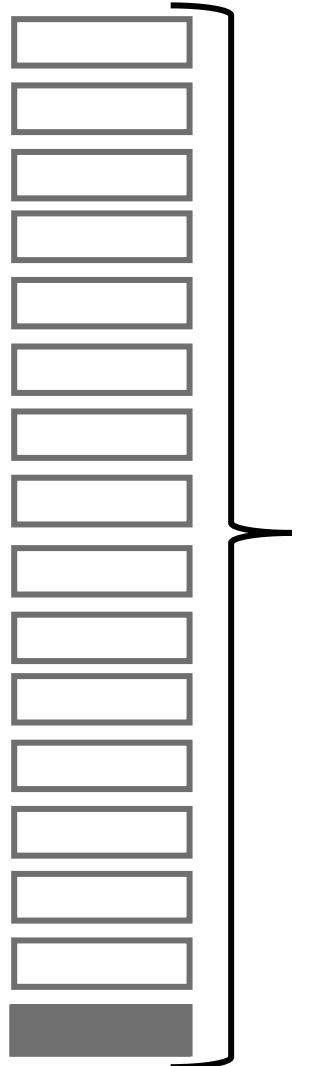
where $\mu = m * \left\lfloor \frac{q}{t} \right\rfloor$ for $m \in R_t, t \ll q$

Homomorphic Addition / Multiplication

- Noise increases linearly / quadratically



Cryptographic Challenges



(R)ing-)Learning W~~i~~th E~~r~~rors: It is hard to find s given c and a

$$c = a \cdot s + e \quad \text{where } a \xleftarrow{\$} R_q, e \xleftarrow{\chi} R_q, s \in R_q$$
$$R = \mathbb{Z}[X]/(X^n + 1)$$

Encryption

$$c = a \cdot s + \mu + e \quad \text{where } \mu = m * \left\lfloor \frac{q}{t} \right\rfloor \text{ for } m \in R_t, t \ll q$$

Homomorphic Addition / Multiplication

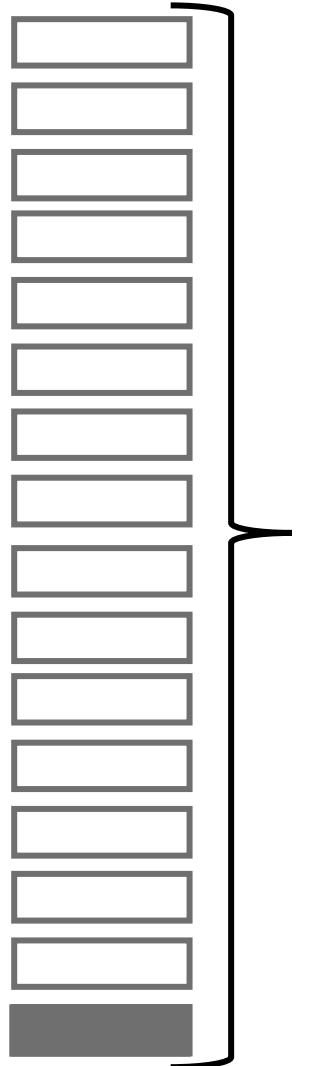
- Noise increases linearly / quadratically

If noise grows too large, decryption fails

- Bootstrapping reduces noise homomorphically



Cryptographic Challenges



(R)ing-)Learning W~~i~~th E~~r~~rors: It is hard to find s given c and a

$$c = a \cdot s + e \quad \text{where } a \xleftarrow{\$} R_q, e \xleftarrow{\chi} R_q, s \in R_q$$
$$R = \mathbb{Z}[X]/(X^n + 1)$$

Encryption

$$c = a \cdot s + \mu + e \quad \text{where } \mu = m * \left\lfloor \frac{q}{t} \right\rfloor \text{ for } m \in R_t, t \ll q$$

Homomorphic Addition / Multiplication

- Noise increases linearly / quadratically

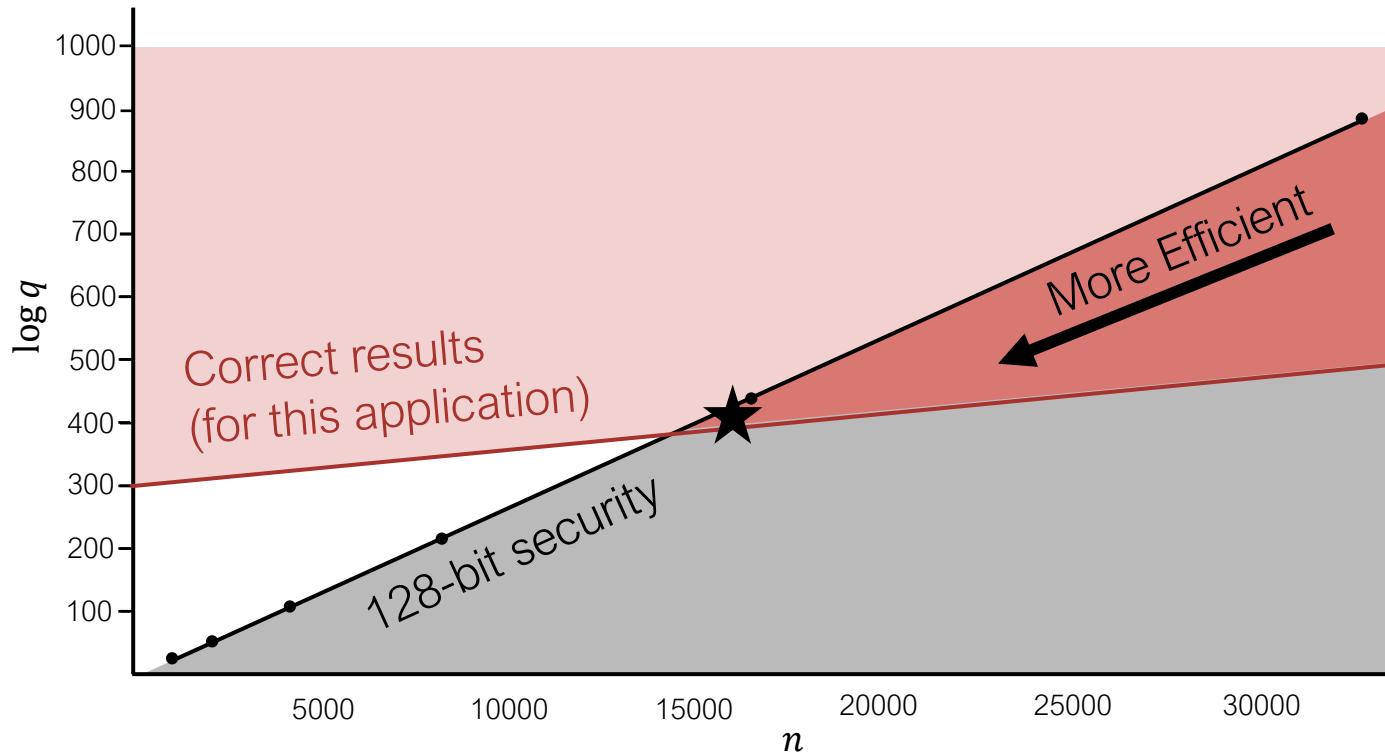
If noise grows too large, decryption fails

- Bootstrapping reduces noise homomorphically



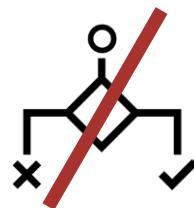
Cryptographic Challenges

- Parameter Selection
 - Based on current knowledge about Learning With Error (LWE) hardness
 - Careful trade-off between efficiency, security and correctness

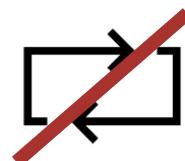


$$R = \mathbb{Z}[X]/(X^n + 1) \quad R_q = R/q$$

Engineering Challenges



No If/Else



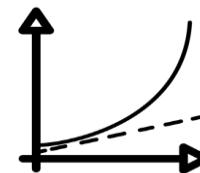
No Loops



No Jumps



Optimizations

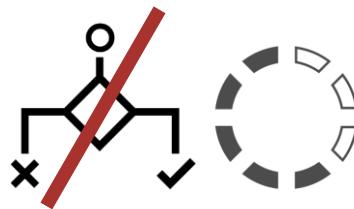


Approximations



SIMD Batching

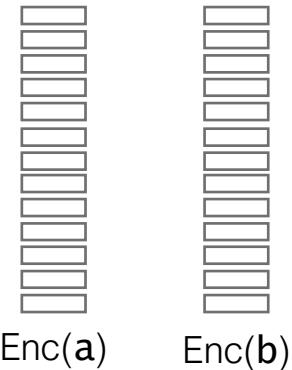
FHE Programming Paradigm



No If/Else



SIMD Batching

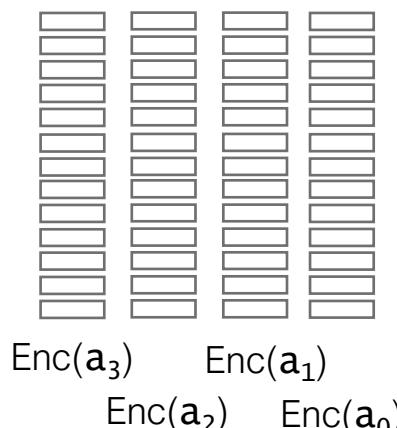


Standard C++

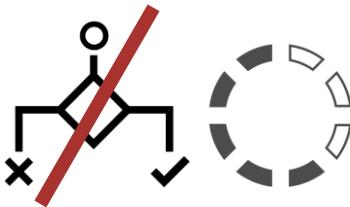
```
int foo(int a, int b) {  
    if(a < b) {  
        return a * b;  
    } else {  
        return a + b;  
    }  
}
```

FHE

```
int foo(int[] a,int[] b){  
  
    int[] c = less(a, b);  
    int[] i = mult(a, b);  
    int[] e = add(a, b);  
    int[] r;  
    for (k=0;k<len(a);++k)  
        r[k] = c[i[k]] +  
                (1-c)*e[k];  
    return r;  
}
```



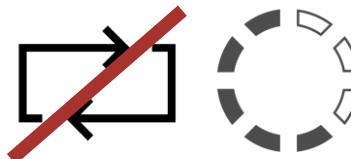
FHE Programming Paradigm



No If/Else

Standard C++

```
int foo(int a, int b) {  
    if(a < b) {  
        return a * b;  
    } else {  
        return a + b;  
    }  
}
```



No Loops

Standard C++

```
int foo(int a, int b){  
    int s = 1;  
    for(i = 0; i < a; ++i){  
        s = s + i*b;  
    }  
    return s;  
}
```



SIMD Batching

FHE

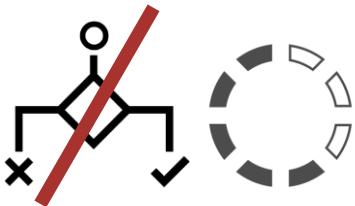
```
int foo(int[] a,int[] b){  
  
    int[] c = less(a, b);  
    int[] i = mult(a, b);  
    int[] e = add(a, b);  
    int[] r;  
    for (k=0;k<len(a);++k)  
        r[k] = c[i[k]] +  
               (1-c)*e[k];  
    return r;  
}
```

FHE

```
int foo(int a, int b){  
    int s = 1;  
    s = less(a,b) *  
        (s + 0*b) +  
        (1-c) * s;  
    s = less(a,b) *  
        (s + 1*b) +  
        (1-c) * s;  
    s = less(a,b) *  
        (s + 2*b) +  
        (1-c) * s;
```



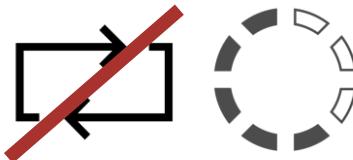
FHE Programming Paradigm



No If/Else

Standard C++

```
int foo(int a, int b) {  
    if(a < b) {  
        return a * b;  
    } else {  
        return a + b;  
    }  
}
```



No Loops

Standard C++

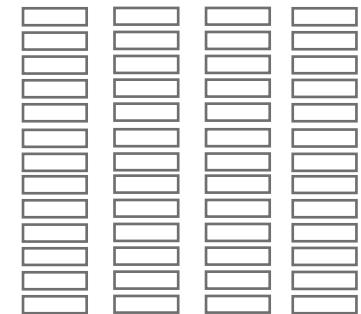
```
int foo(int a, int b){  
  
    int s = 1;  
    for(i = 0; i < a; ++i){  
        s = s + i*b;  
    }  
    return s;  
}
```



SIMD Batching

Standard C++

```
int foo(int[] x,int[] y){  
  
    int[] r;  
    for(i = 0; i < 6; ++i){  
        r[i] = x[i] * y[i];  
    }  
    return r;  
}
```



Enc(x[0])
Enc(x[1])
Enc(x[2])...

FHE

```
int foo(int[] a,int[] b){  
  
    int[] c = less(a, b);  
    int[] i = mult(a, b);  
    int[] e = add(a, b);  
    int[] r;  
    for (k=0;k<len(a);++k)  
        r[k] = c*i[k] +  
               (1-c)*e[k]  
    return r;  
}
```

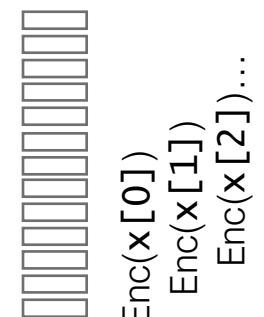
FHE

```
int foo(int a, int b){  
    int s = 1;  
    s = less(a,b) *  
        (s + 0*b) +  
        (1-c) * s;  
    s = less(a,b) *  
        (s + 1*b) +  
        (1-c) * s;  
    s = less(a,b) *  
        (s + 2*b) +  
        (1-c) * s;  
}
```



FHE

```
int foo(int[] x,int[] y){  
  
    int[] r;  
    r[0] = x[0] * y[0];  
    r[1] = x[1] * y[1];  
    r[2] = x[2] * y[2];  
    r[3] = x[3] * y[3];  
    r[4] = x[4] * y[4];  
    r[5] = x[5] * y[5];  
    return r;  
}
```

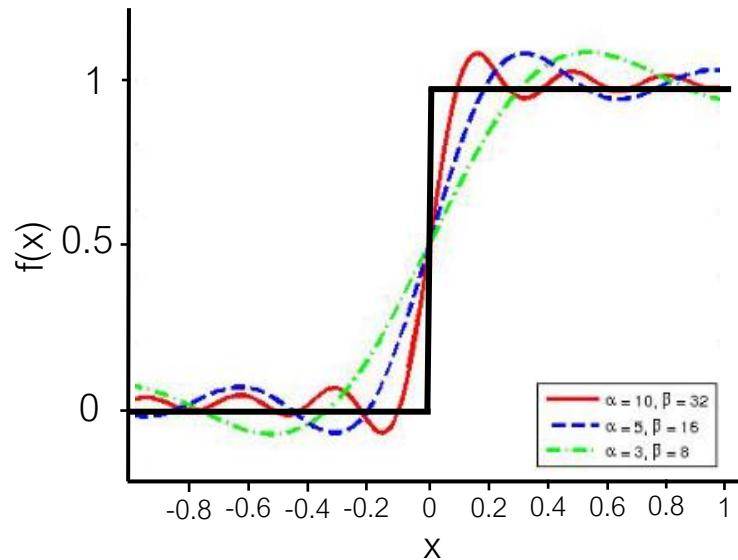


Enc(x[0])
Enc(x[1])
Enc(x[2])...

Approximation

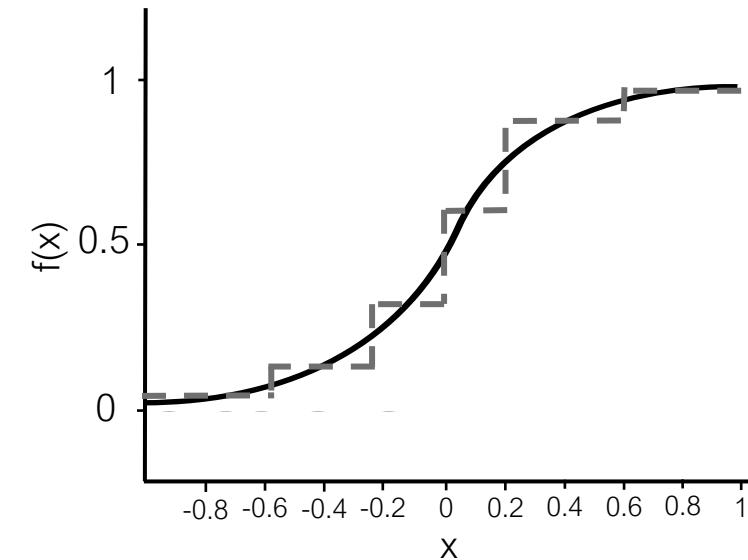
- FHE is most efficient when used over integers
 - However, here + and * only allow polynomials
 - Boolean emulation is powerful, but expensive

- Polynomial Approximation



| x | $f(x)$ |
|------|--------|
| -1 | 0 |
| -0.6 | 0 |
| -0.2 | 0 |
| 0 | 1 |
| 0.2 | 1 |
| 0.6 | 1 |
| 1 | 1 |

- Alternative: Programmable Bootstrapping
 - Technique used in ZAMA's Concrete library
 - Evaluate a Look-Up Table (LUT) during bootstrapping

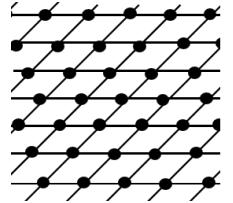


| x | $\tilde{f}(x)$ |
|------|----------------|
| -1 | 0.05 |
| -0.6 | 0.15 |
| -0.2 | 0.33 |
| 0 | 0.66 |
| 0.2 | 0.85 |
| 0.6 | 0.95 |
| 1 | 1 |

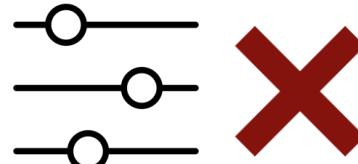
- Frequently requires too many chained multiplications

- Limited to univariate functions (single ctxt as input)

Challenges of Developing FHE Applications



Underlying Math



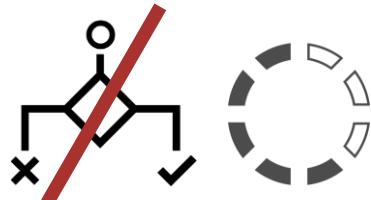
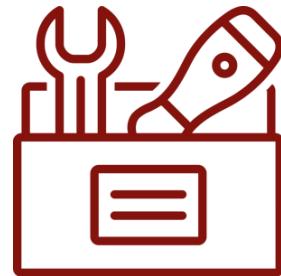
Parameter Selection



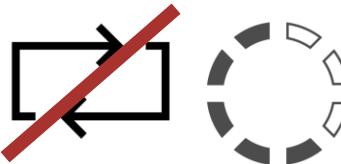
Security



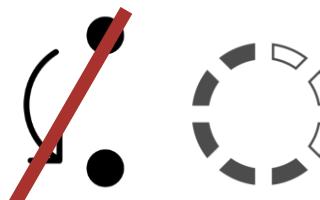
Libraries



No If/Else



No Loops

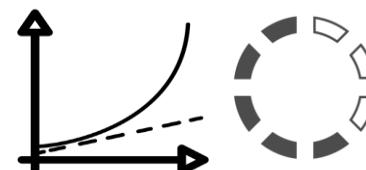


No Jumps

Compilers



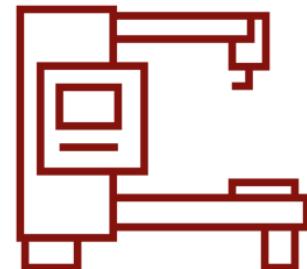
Optimizations



Approximations



SIMD Batching

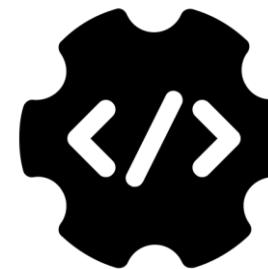


- FHE “Compilers” are frequently not actually technically compilers
 - Term used more generally for tools doing high-level to low-level translation

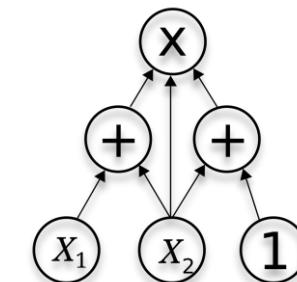
Computer Program



FHE Compiler



Arithmetic Circuit



Input Language

SEAL (library)



```
// First Dense Layer
DenseLayer d1(32, input_size);
seal::Ciphertext result;
mvp(*galoisKeys, *evaluator, *encoder, d1.units(), d1.input_size(),
     d1.weights_as_diags(), image_ctxt, result);
seal::Plaintext b1;
encoder->encode(d1.bias(), result.parms_id(), result.scale(), b1);
evaluator->add_plain_inplace(result, b1);
evaluator->rescale_to_next_inplace(result);

// Activation, x -> x^2
evaluator->square_inplace(result);
evaluator->relinearize_inplace(result, *relinKeys);
evaluator->rescale_to_next_inplace(result);

// Duplication of certain values necessary for the SIMD layout
seal::Plaintext mask; encoder->encode(vec(1, d1.units()),
                                         result.parms_id(),
                                         result.scale(), mask);
evaluator->multiply_plain_inplace(result, mask);
seal::Ciphertext tmp;
evaluator->rotate_vector(result, d1.units(), *galoisKeys, tmp);
evaluator->add_inplace(tmp, result);
evaluator->rescale_to_next_inplace(tmp);

// Second Dense Layer
DenseLayer d2(16, d1.units());
mvp(*galoisKeys, *evaluator, *encoder, d2.units(), d2.input_size(),
     d2.weights_as_diags(), tmp, result);
seal::Plaintext b2; encoder->encode(d2.bias(), result.parms_id(),
                                         result.scale(), b2);
evaluator->add_plain_inplace(result, b2);
evaluator->rescale_to_next_inplace(result);

// Activation, x -> x^2
evaluator->square_inplace(result);
```

EVA



```
def diag(matrix, d):
    m, n = matrix.shape
    r = [0] * n
    for k in range(n):
        r[k] = matrix[k % m][(k + d) % n]
    return r

def mvp(ptxt_matrix, enc_vector):
    m, n = ptxt_matrix.shape
    log2_n_div_m = math.ceil(math.log(n // m, 2)) t = 0
    for i in range(m):
        t += (enc_vector << i) * diag(ptxt_matrix, i)

    for i in range(log2_n_div_m):
        offset = n // (2 << i) t += t << offset
    return t

def compile():
    mlp = EvaProgram('NN (MLP)', vec_size=32 * 32)
    with mlp:
        image = Input('input_0')
        d1 = mvp(weights_1, image)
        d1 = d1 + bias_1.tolist()

        act1 = d1 * d1
        d2 = mvp(weights_2, act1)
        d2 = d2 + bias_2.tolist()
        act2 = d2 * d2

        Output('output', act2)
        Output('output', d1)
```

nGraph-HE



```
import numpy as np
import tensorflow as tf
from tf.keras.layers import Input, Dense, Activation

def mnist_mlp_model(input):
    def square_activation(x):
        return x * x
    known_shape = input.get_shape()[1:]

    size = np.prod(known_shape) print('size', size)
    y = tf.reshape(input, [-1, size])
    y = Dense(input_shape=[1, 784], units=30, use_bias=True)(y)
    y = Activation(square_activation)(y)
    y = Dense(units=10, use_bias=True, name="output")(y)
    known_shape = y.get_shape()[1:] size =
    np.prod(known_shape)
    return y
```

SEALion



```
from sealion.heras.models import Sequential
from sealion.heras.layers import Dense, Activation

model = Sequential()
model.add(Dense(units=30, input_dim=784,
                input_saturation=4, saturation=2 ** 4))
model.add(Activation())
model.add(Dense(units=10, saturation=2 ** 4))

model.compile(loss='categorical_crossentropy',
               optimizer='adam', metrics=['accuracy'])
```

Input Language

SEAL (library)



```
// First Dense Layer
DenseLayer d1(32, input_size);
seal::Ciphertext result;
mvp(*galoisKeys, *evaluator, *encoder, d1.units(), d1.input_size(),
    d1.weights_as_diags(), image_ctxt, result);
seal::Plaintext b1;
encoder->encode(d1.bias(), result.parms_id(), result.scale(), b1);
evaluator->add_plain_inplace(result, b1);
evaluator->rescale_to_next_inplace(result);

// Activation, x -> x^2
evaluator->square_inplace(result);
evaluator->relinearize_inplace(result, *relinKeys);
evaluator->rescale_to_next_inplace(result);

// Duplication of certain values necessary for the SIMD layout
seal::Plaintext mask; encoder->encode(vec(1, d1.units()),
    result.parms_id(),
    result.scale(), mask);
evaluator->multiply_plain_inplace(result, mask);
seal::Ciphertext tmp;
evaluator->rotate_vector(result, d1.units(), *galoisKeys, tmp);
evaluator->add_inplace(tmp, result);
evaluator->rescale_to_next_inplace(tmp);

// Second Dense Layer
DenseLayer d2(16, d1.units());
mvp(*galoisKeys, *evaluator, *encoder, d2.units(), d2.input_size(),
    d2.weights_as_diags(), tmp, result);
seal::Plaintext b2; encoder->encode(d2.bias(), result.parms_id(),
    result.scale(), b2);
evaluator->add_plain_inplace(result, b2);
evaluator->rescale_to_next_inplace(result);

// Activation, x -> x^2
evaluator->square_inplace(result);
```

EVA



```
def diag(matrix, d):
    m, n = matrix.shape
    r = [0] * n
    for k in range(n):
        r[k] = matrix[k % m][(k + d) % n]
    return r

def mvp(ptxt_matrix, enc_vector):
    m, n = ptxt_matrix.shape
    log2_n_div_m = math.ceil(math.log(n // m, 2)) t = 0
    for i in range(m):
        t += (enc_vector << i) * diag(ptxt_matrix, i)

    for i in range(log2_n_div_m):
        offset = n // (2 << i) t += t << offset
    return t

def compile():
    mlp = EvaProgram('NN (MLP)', vec_size=32 * 32)
    with mlp:
        image = Input('input_0')
        d1 = mvp(weights_1, image)
        d1 = d1 + bias_1.tolist()

        act1 = d1 * d1
        d2 = mvp(weights_2, act1)
        d2 = d2 + bias_2.tolist()
        act2 = d2 * d2

        Output('output', act2)
        Output('output', d1)
```

nGraph-HE



```
import numpy as np
import tensorflow as tf
from tf.keras.layers import Input, Dense, Activation

def mnist_mlp_model(input):
    def square_activation(x):
        return x * x
    known_shape = input.get_shape()[1:]

    size = np.prod(known_shape) print('size', size)
    y = tf.reshape(input, [-1, size])
    y = Dense(input_shape=[1, 784], units=30, use_bias=True)(y)
    y = Activation(square_activation)(y)
    y = Dense(units=10, use_bias=True, name="output")(y)
    known_shape = y.get_shape()[1:] size =
    np.prod(known_shape)
    return y
```

SEALion



```
from sealion.heras.models import Sequential
from sealion.heras.layers import Dense, Activation

model = Sequential()
model.add(Dense(units=30, input_dim=784,
                input_saturation=4, saturation=2 ** 4))
model.add(Activation())
model.add(Dense(units=10, saturation=2 ** 4))

model.compile(loss='categorical_crossentropy',
              optimizer='adam', metrics=['accuracy'])
```

Input Language

SEAL (library)



```
// First Dense Layer
DenseLayer d1(32, input_size);
seal::Ciphertext result;
mvp(*galoisKeys, *evaluator, *encoder, d1.units(), d1.input_size(),
    d1.weights_as_diags(), image_ctxt, result);
seal::Plaintext b1;
encoder->encode(d1.bias(), result.parms_id(), result.scale(), b1);
evaluator->add_plain_inplace(result, b1);
evaluator->rescale_to_next_inplace(result);

// Activation, x -> x^2
evaluator->square_inplace(result);
evaluator->relinearize_inplace(result, *relinKeys);
evaluator->rescale_to_next_inplace(result);

// Duplication of certain values necessary for the SIMD layout
seal::Plaintext mask; encoder->encode(vec(1, d1.units()),
    result.parms_id(),
    result.scale(), mask);
evaluator->multiply_plain_inplace(result, mask);
seal::Ciphertext tmp;
evaluator->rotate_vector(result, d1.units(), *galoisKeys, tmp);
evaluator->add_inplace(tmp, result);
evaluator->rescale_to_next_inplace(tmp);

// Second Dense Layer
DenseLayer d2(16, d1.units());
mvp(*galoisKeys, *evaluator, *encoder, d2.units(), d2.input_size(),
    d2.weights_as_diags(), tmp, result);
seal::Plaintext b2; encoder->encode(d2.bias(), result.parms_id(),
    result.scale(), b2);
evaluator->add_plain_inplace(result, b2);
evaluator->rescale_to_next_inplace(result);

// Activation, x -> x^2
evaluator->square_inplace(result);
```

```
def diag(matrix m, n = mat.size(), r = [0]*n):
    for k in range(n):
        r[k] = m[k][k]
    return r

def mvp(ptxt, matrix m, n = ptxt.size(), log2_n_div_m = 0):
    for i in range(n):
        t += (evaluator.multiply_plain(m[i], ptxt[i]) *
               seal::Ciphertext ctxt_t)
        for j in range(n):
            offset = i * n + j
            if (offset < n):
                t += (evaluator.multiply_plain(m[i], ptxt[j]) *
                       seal::Ciphertext ctxt_t))
            else:
                offset -= n
                t += (evaluator.multiply_plain(m[i], ptxt[offset]) *
                       seal::Ciphertext ctxt_t))
    return t

def compile():
    mlp = EvalFunc()
    with mlp:
        image = seal::Plaintext()
        d1 = mlp.encrypt(image)
        d1 = d1.decrypt()
        act1 = cipher(d1)
        d2 = mlp.encrypt(act1)
        d2 = d2.decrypt()
        act2 = cipher(d2)
        Output()
        Output()
```

```
void mvp(const seal::GaloisKeys &galois_keys, seal::Evaluator &evaluator, seal::CKKSEncoder &encoder,
         size_t m, size_t n, std::vector<vec> diagonals, const seal::Ciphertext &ctv, seal::Ciphertext &enc_result) {
    if (m==0 || m!=diagonals.size())
        throw invalid_argument( "Matrix must not be empty, and diagonals vector must have size m!");
    if (n!=diagonals[0].size() || n==0)
        throw invalid_argument( "Diagonals must have non-zero dimension that matches n");
    size_t n_div_m = n/m;
    size_t log2_n_div_m = ceil(log2(n_div_m));
    if (m*n_div_m!=n || (2ULL << (log2_n_div_m - 1))!=n_div_m && n_div_m!=1)
        throw invalid_argument( "Matrix dimension m must divide n and the result must be power of two");

    Ciphertext ctxt_t;
    for (size_t i = 0; i < m; ++i) {
        // rotated_v = rot(v,i)
        Ciphertext ctxt_rotated_v = ctv;
        if (i != 0)
            evaluator.rotate_vector_inplace(ctxt_rotated_v, i, galois_keys);

        Plaintext ptxt_current_diagonal;
        encoder.encode(diagonals[i], ctxt_rotated_v.parms_id(), ctxt_rotated_v.scale(), ptxt_current_diagonal);
        Ciphertext ctxt_tmp;
        evaluator.multiply_plain(ctxt_rotated_v, ptxt_current_diagonal, ctxt_tmp);

        if (i==0)
            ctxt_t = ctxt_tmp;
        else
            evaluator.add_inplace(ctxt_t, ctxt_tmp);

        Ciphertext ctxt_r = std::move(ctxt_t);

        for (int i = 0; i < log2_n_div_m; ++i) {
            Ciphertext ctxt_rotated_r = ctxt_r;
            size_t offset = n/(2ULL << i);
            evaluator.rotate_vector_inplace(ctxt_rotated_r, offset, galois_keys);
            evaluator.add_inplace(ctxt_r, ctxt_rotated_r);
        }
        enc_result = std::move(ctxt_r);
    }
}
```

SEAL (library)



```
// First Dense Layer
DenseLayer d1(32, input_size);
seal::Ciphertext result;
mvp(*galoisKeys, *evaluator, *encoder,
    d1.weights_as_diags(), image);
seal::Plaintext b1;
encoder->encode(d1.bias(), result);
evaluator->add_plain_inplace(result, b1);
evaluator->rescale_to_next_inplace(result);

// Activation, x -> x^2
evaluator->square_inplace(result);
evaluator->relinearize_inplace(result);
evaluator->rescale_to_next_inplace(result);

// Duplication of certain values needed
seal::Plaintext mask; encoder->encode(
    result.parms_id(),
    result.scale(), mask);
evaluator->multiply_plain_inplace(result, mask);
seal::Ciphertext tmp;
evaluator->rotate_vector(result, d1);
evaluator->add_inplace(tmp, result);
evaluator->rescale_to_next_inplace(result);

// Second Dense Layer
DenseLayer d2(16, d1.units());
mvp(*galoisKeys, *evaluator, *encoder, d2.weights(), d2.inputs());
d2.weights_as_diags(), tmp, result);
seal::Plaintext b2; encoder->encode(d2.bias(), result.parms_id(),
    result.scale(), b2);
evaluator->add_plain_inplace(result, b2);
evaluator->rescale_to_next_inplace(result);

// Activation, x -> x^2
evaluator->square_inplace(result);
```

EVA



```
def diag(matrix, d):
```

```
def mvp(ptxt_matrix, enc_vector):
    m, n = ptxt_matrix.shape
    log2_n_div_m = math.ceil(math.log(n // m, 2)) t = 0
    for i in range(m):
```

nGraph-HE



```
import numpy as np
```

```
known_shape = input.get_shape()[1:]
size = np.prod(known_shape) print('size', size)
y = tf.reshape(input, [-1, size])
use_bias=True)(y)
="output")(y)
```

Disconnect between low- and high-level languages

Libraries and compilers target very different audiences

Consider UX beyond programming

SEALion



```
def compile():
    mlp = EvaProgram('NN (MLP)', vec_size=32 * 32)
    with mlp:
        image = Input('input_0')
```

```
act2 = d2 - d2
Output('output', act2)
Output('output', d1)
```

```
model.add(Activation())
model.add(Dense(units=10, saturation=2 ** 4))

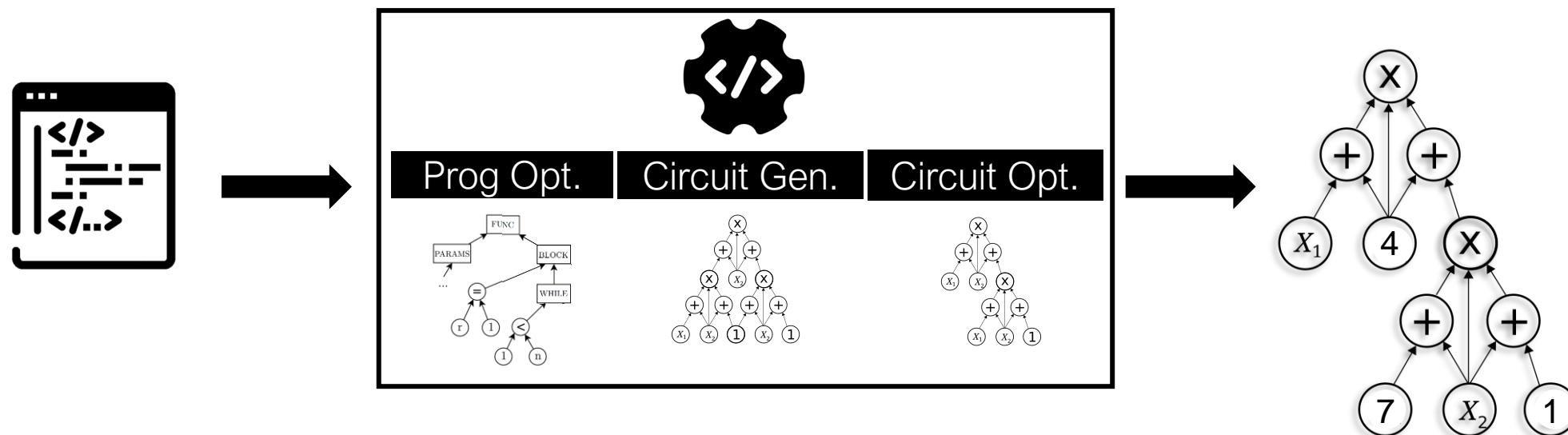
model.compile(loss='categorical_crossentropy',
              optimizer='adam', metrics=['accuracy'])
```

- FHE “Compilers” are frequently not actually technically compilers
 - Term used more generally for tools doing high-level to low-level translation

Computer Program

FHE Compiler

Arithmetic Circuit

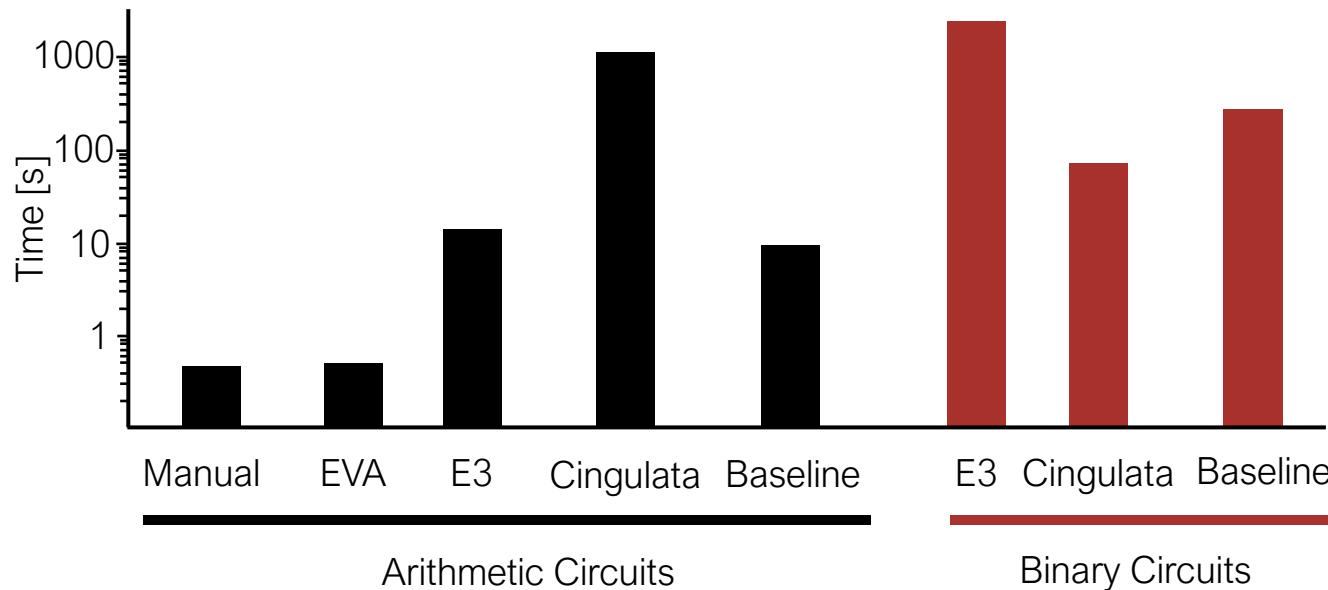
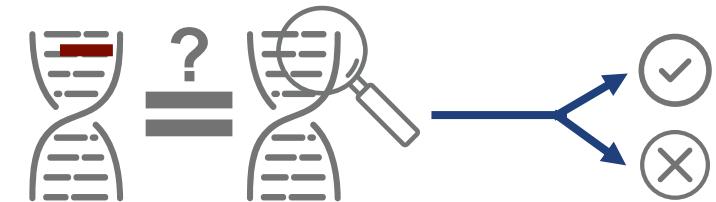


- Where do compilers apply optimizations?

| | Input | Prog Opt | Circuit Gen | Circuit Opt |
|----------------|---|----------|-------------|-------------|
| ALCHEMY |  | ✓ | | |
| Cingulata |  | | | ✓ |
| E ³ |  | | ... | |
| EVA |  | | | ✓ |
| Marble |  | | ... | |
| Ramparts |  | ✓ | | |
| CHET |  | | ✓ | |
| nGraph-HE |  | | ✓ | |
| SEALion |  | | ✓ | |

Evaluation: χ^2 Test

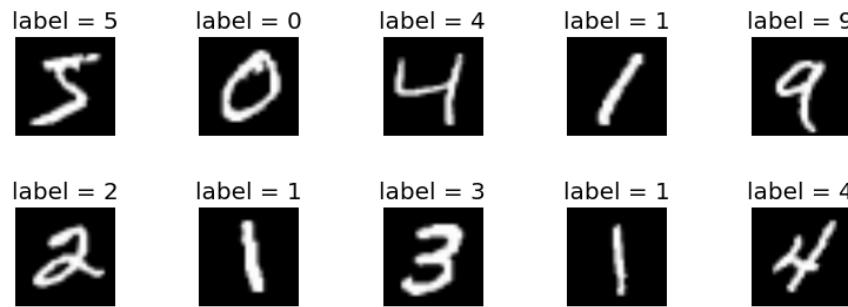
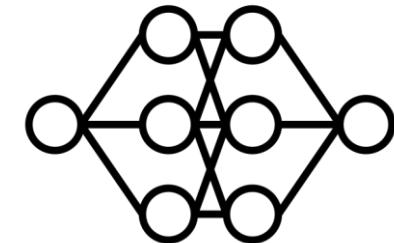
- Chi-squared tests are common statistical tests
 - Here: Pearson's Goodness-of-Fit test
 - Used in Genome-Wide Association Studies (GWAS) [LLN14]
 - Can be rearranged to require only + and x



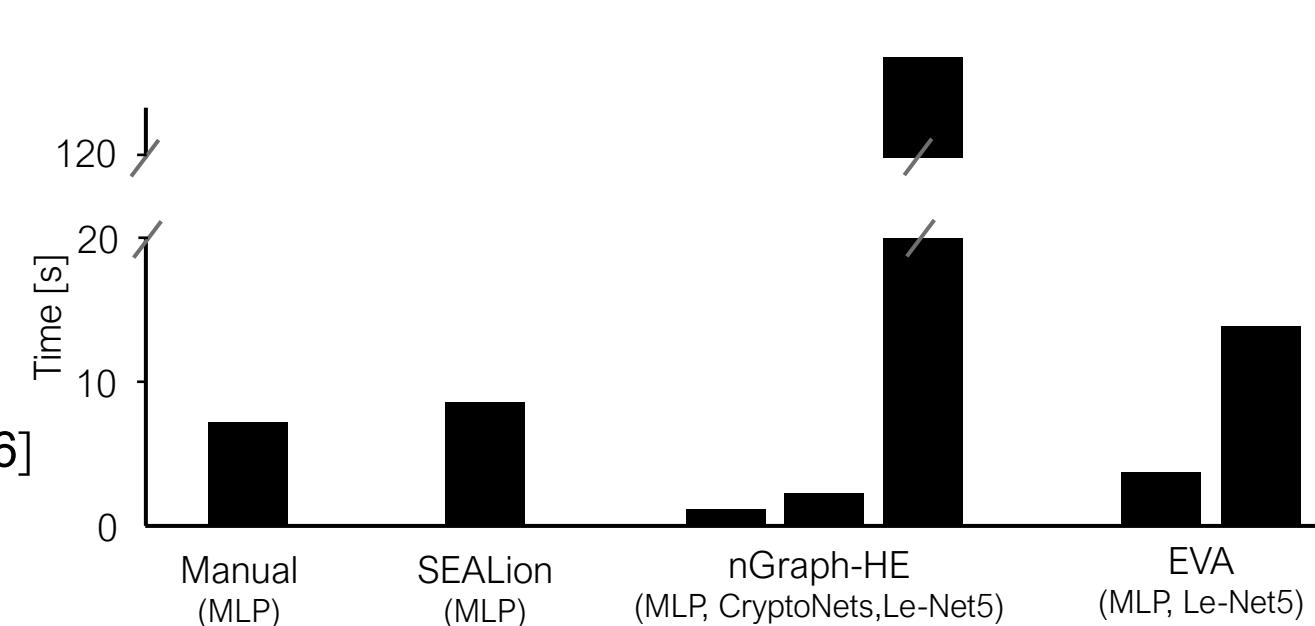
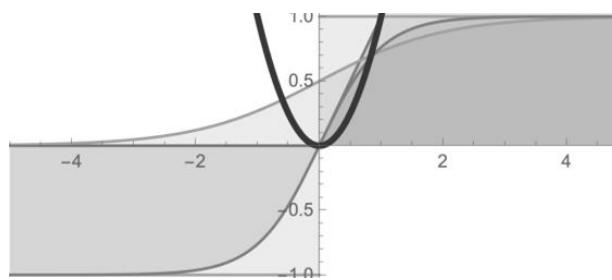
$$\alpha = (4N_0 N_2 - N_1^2)^2$$
$$\beta_1 = 2(2N_0 + N_1)^2$$
$$\beta_2 = (2N_0 + N_1)(2N_2 + N_1)$$
$$\beta_3 = 2(2N_2 + N_1)^2$$

Evaluation: Machine Learning Inference

- MNIST – handwritten digit recognition
 - Simple task used as a reference benchmark in ML
 - Given an image of a digit, recognize a number in 0-9



- Approximate Activation Function with x^2 [GDK+16]



Performance

Wide gap between naïve implementations & expert solutions

Current Tools

Solve individual aspects, but fall short of the wider vision

Open Issues

Hardware Accelerators, Intermediate Representation

Future Directions for FHE Tool Development

E2E

Toolchains must address all aspects of FHE development

Automate

Translation between imperative paradigm and FHE

Optimize

Identify & exploit opportunities for SIMD parallelism

SoK: Fully Homomorphic Encryption Compilers

Alexander Vianu
ETH Zurich
alexander.vianu@inf.ethz.ch

Patrick Jätkä
ETH Zurich
pjatk@ethz.ch

Abstract—Fully Homomorphic Encryption (FHE) allows a third party to perform arbitrary computations on encrypted data, learning nothing about the inputs or the computation results. Hence, it provides a solution to the “computation privacy” problem. However, it remained first described by Rivest et al. in the 1970s. However, it remained until Craig Gentry presented the first feasible FHE scheme in 2009. This scheme has led to several breakthroughs in a variety of data services, making it highly relevant in the development of cloud computing. This document, in turn, has led to a large uptake in recent FHE tool evaluations to expand the current state of the art and justify series for future development and companies. We perform experiments on a variety of applications to evaluate these tools. We provide recommendations for developers interested in FHE-based applications and a discussion on future directions for FHE tool development.

1. INTRODUCTION

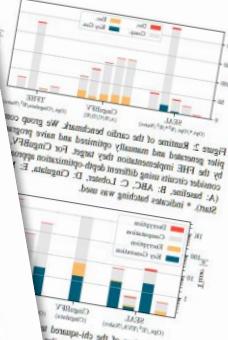
Recent years have seen unprecedented growth in the adoption of cloud computing services. More and more highly regulated businesses and organizations (e.g., banks, governments, insurance, health) have moved their data and services to the cloud. This trend has led to a surge in demand for secure and confidentially within transit, especially in the light of the numerous reports of data breaches [1], [2]. A key technological enabler for secure computation and privacy is Fully Homomorphic Encryption (FHE). It is a key technology for practical real-world use [3]–[9]. FHE allows arbitrary computations to be performed over encrypted data, eliminating the need to decrypt the data and exposing it to potential risks while in use. While first proposed in the 1970s [10], FHE was long considered impractical or uninteresting. However, thanks to advances in the underlying theory, general hardware improvements, and more efficient implementations, it has become increasingly practical. In 2009, a breakthrough work from Craig Gentry proposed the first feasible FHE scheme [11]. In the last decade, performance has gone from a theoretical concept to reality. For example, improving by up to five orders of magnitude. For example, times for a multiplication between ciphertexts dropped from 30 minutes to less than 20 milliseconds. While this is still around

seven orders of magnitude slower than an SIMD instruction on a modern CPU, it is sufficient to make many applications practical. Additionally, modern schemes introduce SIMD-style parallelism, encoding thousands of plaintext values into applications, which FHE tools can use to further improve throughput [12].

These advances have enabled a wide range of applications to provide a new type of domain. These include mobile devices, where FHE has been used to encrypt the back-end of a privacy-preserving system [13], where continuing analysis [14] has been used to enable privacy-preserving genome analysis [15] and to run various well-known problems, FHE has been used to run large datasets. More generally, FHE has been used for tasks ranging from linear machine learning [16] to run time [17] to superimposing previous logistic regression [18] to Encrypted Neural Network inference [19]. As a consequence, there has been increasing interest [19] that “by 2025, at least 20% of companies will have a budget for projects that include fully homomorphic encryption.”

Despite these recent breakthroughs, building secure and efficient FHE-based applications remains a challenging task. This largely stems from the differences between traditional programming paradigms and FHE computation model, which poses unique challenges. For example, virtually all standard programming paradigms rely on data-dependent branching, e.g., these statements are typically data-independent. FHE computation are, by definition, data-independent. Working with FHE also violates the privacy guarantees. Working with FHE different schemes offer varying performance tradeoffs. To address some of the engineering challenges in this space, we have seen a range of tools and frameworks aiming to improve accessibility and reduce barriers to entry in this field.

Without tool support, realizing FHE-based computations by implementing the required mathematical operations directly or using an arbitrary precision arithmetic library is complex, requiring considerable expertise in both cryptography and high-performance numerical computation. Therefore, FHE libraries like the Simple Fully Encrypted Arithmetic Library (SEAL) [20] or the Fast Fully Homomorphic Encryption Library over the



github.com/MarbleHE/SoK



To appear at IEEE S&P 2021

preprint: arxiv.org/abs/2101.07078