Building Applications with Homomorphic Encryption

A Presentation from the Homomorphic Encryption Standardization Consortium

HomomorphicEncryption.org

0.1 – Presenters

- Roger A. Hallman (SPAWAR Systems Center Pacific; Thayer School of Engineering, Dartmouth College, USA)
- Kim Laine (Microsoft Research, USA)
- Wei Dai (Worcester Polytechnic Institute, USA)
- Nicolas Gama (Inpher, Inc., Switzerland)
- Alex J. Malozemoff (Galois, Inc., USA)
- Yuriy Polyakov (NJIT Cybersecurity Research Center, USA)
- Sergiu Carpov (CEA, LIST, France)

0.2 – Agenda – Part 1

1. Introduction to Homomorphic Encryption (Presenter: Roger Hallman)
2. HE Fundamentals (Presenter: Wei Dai)
3. How to Build HE Applications? (Presenter: Yuriy Polyakov)
4. Standardization and Open Problems (Presenter: Kim Laine)
5. Previewing Part 2 of this Tutorial (Presenter: Roger Hallman)

0.3 – Agenda – Break

Assistance will be provided during a 30-minute break for audience members who are downloading and installing HE libraries.

0.4 – Agenda – Part 2

1. A High-level View of Available HE Libraries (Presenter: Roger Hallman)
2. SEAL (Presenter: Kim Laine)
3. PALISADE (Presenter: Yuriy Polyakov)
4. TFHE (Presenter: Nicolas Gama)
5. cuFHE and Hardware Acceleration (Presenter: Wei Dai)
6. Compilers for HE (Presenters: Alex Malozemoff and Sergiu Carpov)
7. Concluding Remarks (Presenter: Roger Hallman)

1.0 – Introduction to Homomorphic Encryption

What is Homomorphic Encryption (HE)?

- ❏ Allows for computation on encrypted data
- ❏ Enables outsourcing of data storage/processing

History of HE:

- ❏ Rivest, Adleman, Dertouzos (1978) -- “On Data Banks and Privacy Homomorphisms”
- ❏ Gentry (2009) -- “A Fully Homomorphic Encryption Scheme”
- ❏ Multiple HE schemes developed after 2009

1.1 – How HE is related to symmetric and public key encryption?

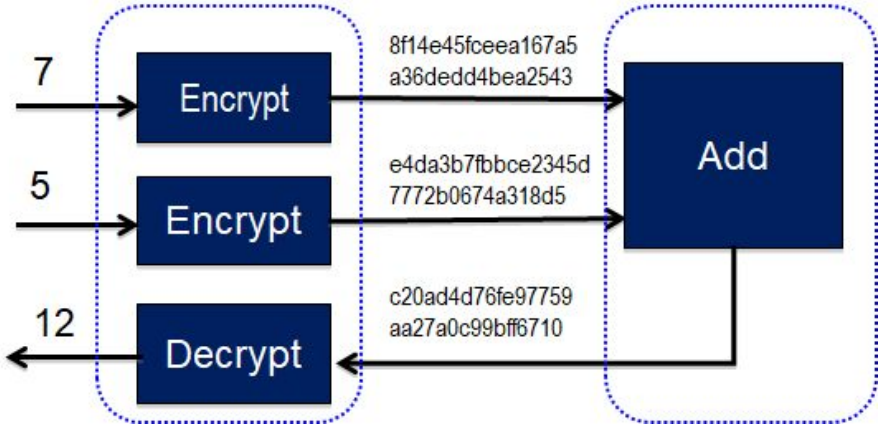
- HE schemes provide efficient instantiations of post-quantum public-key and symmetric-key encryption schemes
- Homomorphic encryption can be viewed as a generalization of public key encryption

1.2 – FAQ

- ❏ Data enter / stay in / leave untrusted networks encrypted.
- ❏ Do operations on ciphertext and plaintext reveal secret?

  No, an operation on ciphertext and plaintext outputs ciphertext.
- ❏ Is decryption performed during computation?

  No, computation is performed without decryption.



1.3 – Applications

Business models and application domains:

Domain	Genomics	Health	National Security	Education	Social Security	Business Analytics	Cloud
Sample Topics	*GWAS*	*billing and reporting*	*smart grid*	*school dropouts*	*credit history*	*prediction*	*storage, sharing*
Data Owner	*medical institutions*	*clinics and hospitals*	*nodes and network*	*schools, welfare*	*government*	*business owners*	*clients*
Why HE?	*HIPAA*	*cyber insurance*	*privacy*	*FERPA*	*cyber crimes*	*data are valuable*	*untrusted server*
Who pays?	*health insurance*	*hospital*	*energy company*	*DoE*	*government*	*business owners*	*clients*

1.3 – Example: Healthcare

Precision medicine requires intensive computation on highly identifiable data.

Challenges:

1. Therapy safety and efficacy must by determined.
2. Patients are concerned about privacy and agency (against breaches).
3. Agency, hospitals must ensure compliance with relevant laws (such as HIPAA).
4. Pharmaceutical companies are concerned about protecting their IP.

Currently, require unappealing trade-offs, sometimes with disastrous outcomes for both organizations and their patients.

HE provides a novel solution to some of these trade-offs at a cost that is minimal compared to such outcomes.

1.4 – Other Secure Computing Approaches

How HE is different from MPC and SGX

	HE	**MPC**	**SGX**
Performance	Compute-bound	Network-bound	
Privacy	Encryption	Encryption / Non-collusion	Trusted Hardware
Non-interactive	✔	✘	✔
Cryptographic security	✔	✔	✘ (known attacks)

- Hybrid approaches possible

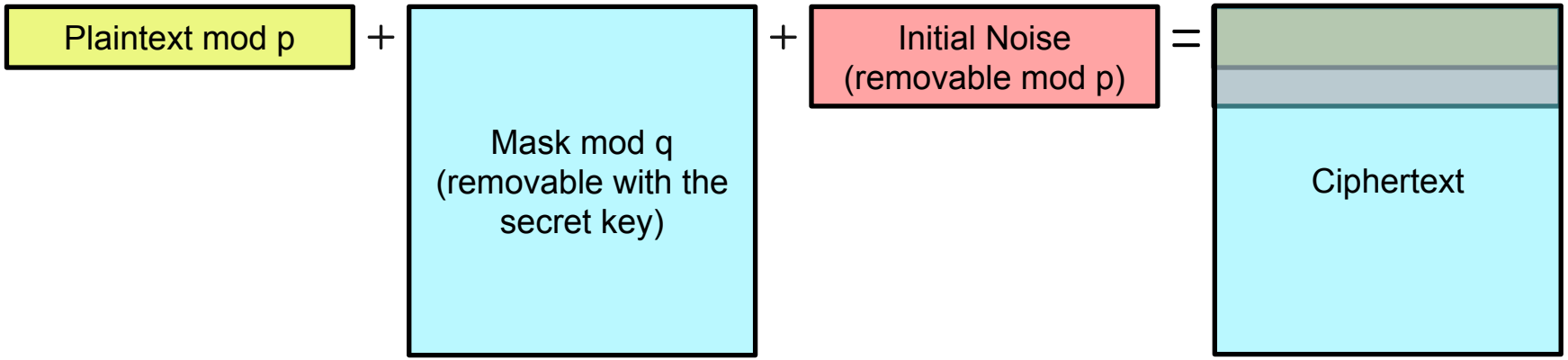
2.0 – Understanding HE

- *“Homomorphic”*: a (secret) mapping from plaintext space to ciphertext space that preserves arithmetic operations.
- *Mathematical Hardness: (Ring) Learning with Errors Assumption*; every image (ciphertext) of this mapping looks uniformly random in range (ciphertext space).
- *“Security level”*: the hardness of inverting this mapping without the secret key.
  - Example: 128 bits → 2^{128} operations to break

2.0 – Understanding HE

- Plaintext: elements and operations of a polynomial ring (mod x^n+1, mod p).
  - Example: $3x^5 + x^4 + 2x^3 + ...$
- Ciphertext: elements and operations of a polynomial ring (mod x^n+1, mod q).
  - Example: $7862x^5 + 5652x^4 + ...$

2.1 – A Fresh Encryption

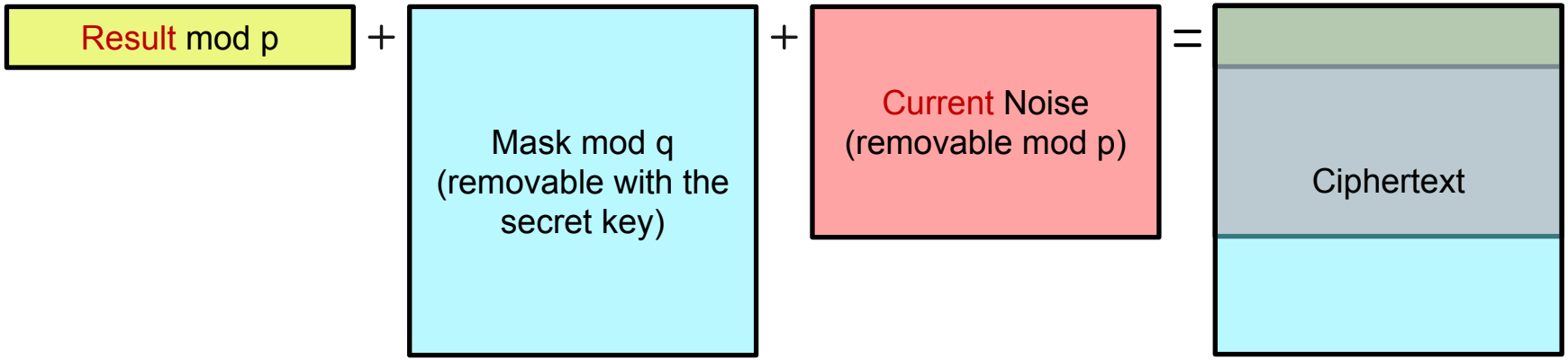


- Horizontal: each coefficient in a polynomial or in a vector.
- Vertical: size of coefficients.

Initial noise is small in terms of coefficients' size.

2.2 – Noise Growth in Computation

After some computation:

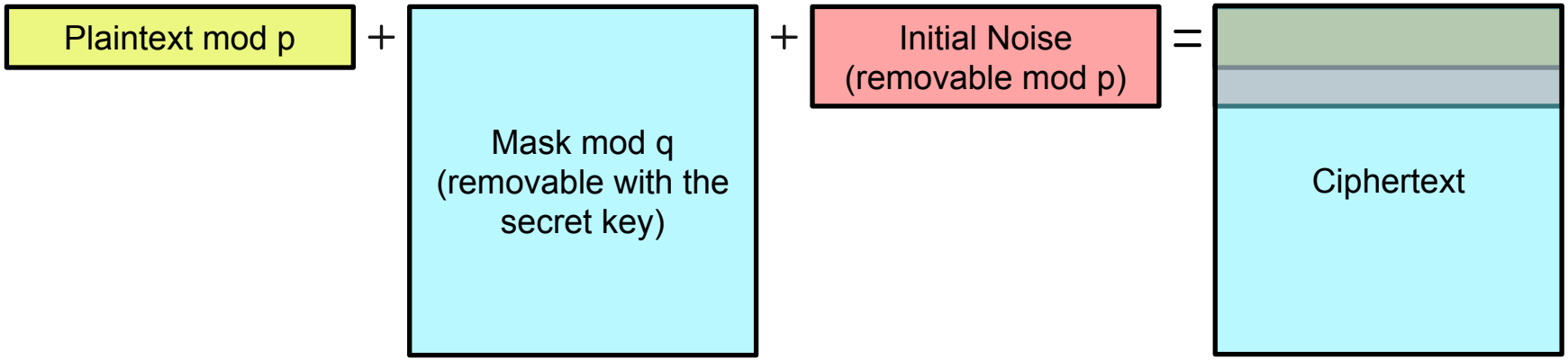


- Horizontal: each coefficient in a polynomial or in a vector.
- Vertical: size of coefficients.

After each level, noise increases.

2.3 – Bootstrapping

Homomorphic decryption with an encrypted secret key.

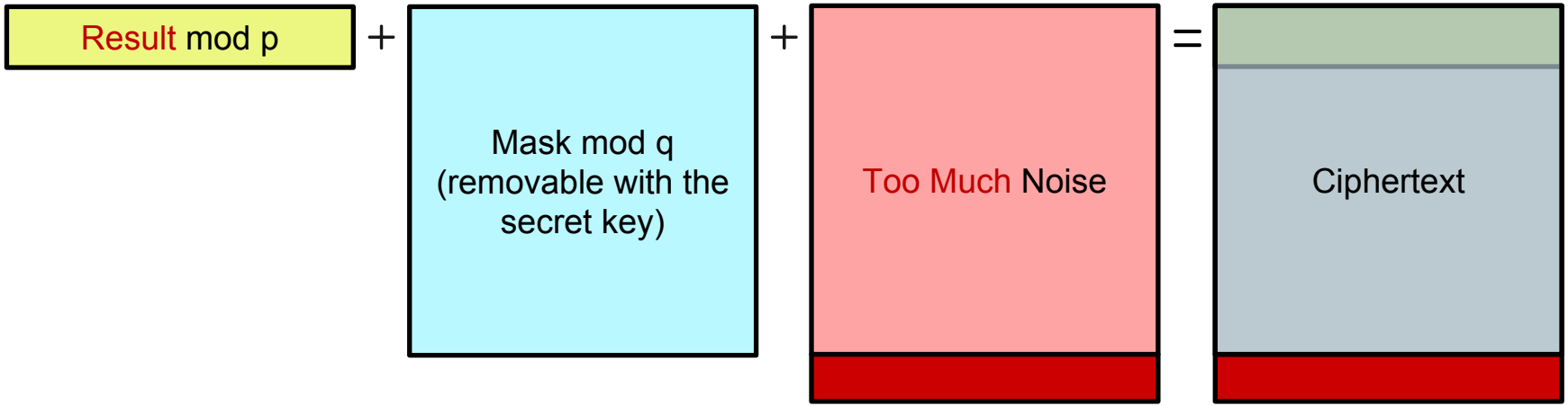


- Horizontal: each coefficient in a polynomial or in a vector.
- Vertical: size of coefficients.

At some level, noise is too much to decrypt.

2.4 – Noise Overflow

Too much computation:



- Horizontal: each coefficient in a polynomial or in a vector.
- Vertical: size of coefficients.

At some level, noise is too much to decrypt.

2.5 – Encoding Techniques

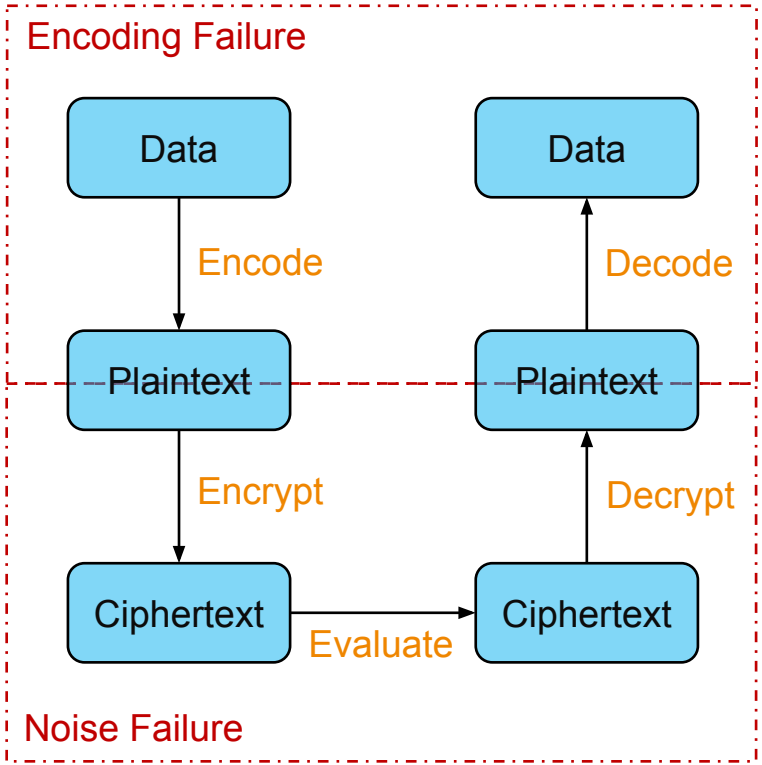
Reduce ciphertext / plaintext size ratio.

1. Multi-precision integers / fractional numbers (mod p^n).
2. Batching a vector of integers / fractional numbers (mod p).

Plaintext encoding should be correct before ciphertext evaluation.

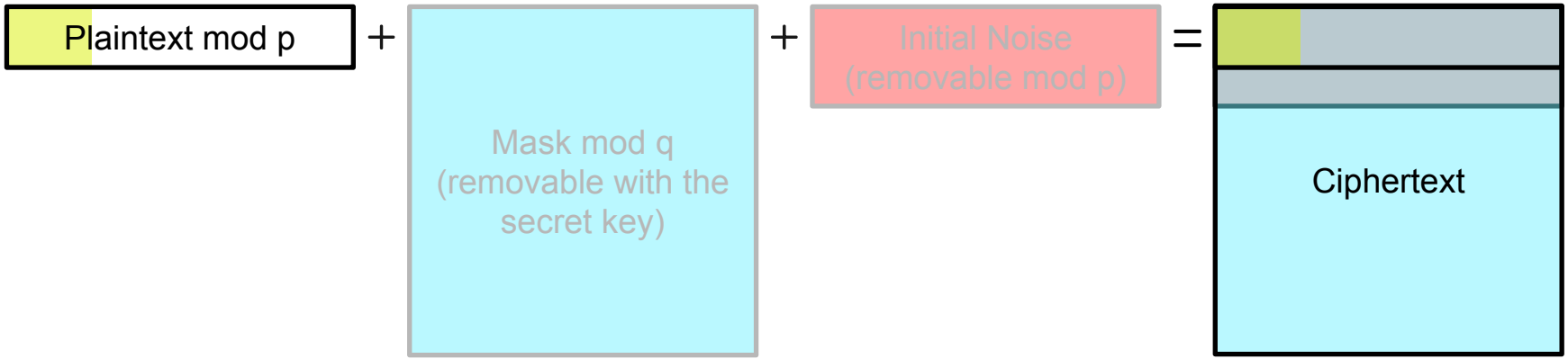
<u>Example:</u>

$5 \times 7 \bmod 17 \neq 35$



2.6 – Encoding Integers / Fractional Numbers

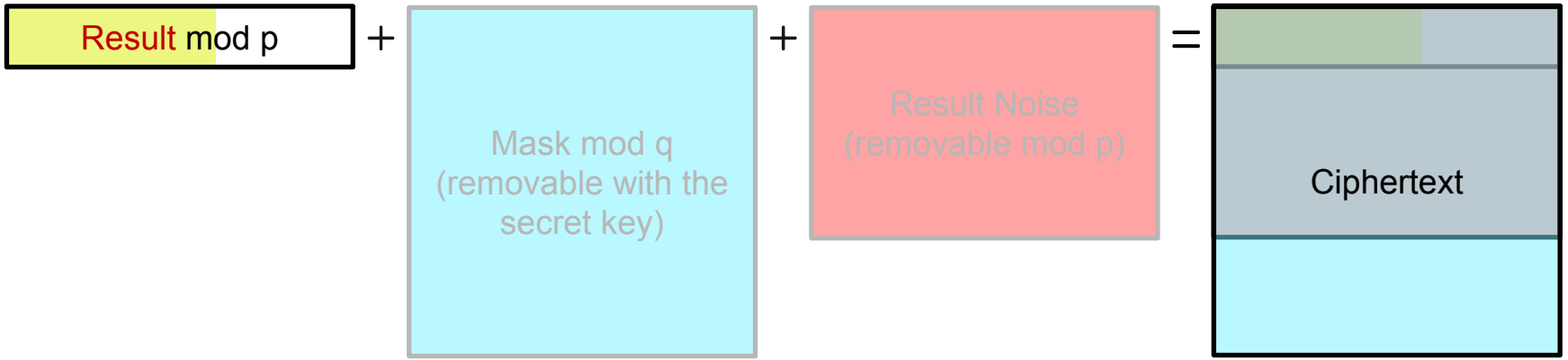
Correctness only depend on plaintext:



- Horizontal: each coefficient in a polynomial or in a vector.
- Vertical: size of coefficients.

Initial noise is small in terms of coefficients' size.
Message are encoded to lower-degree terms of a plaintext.

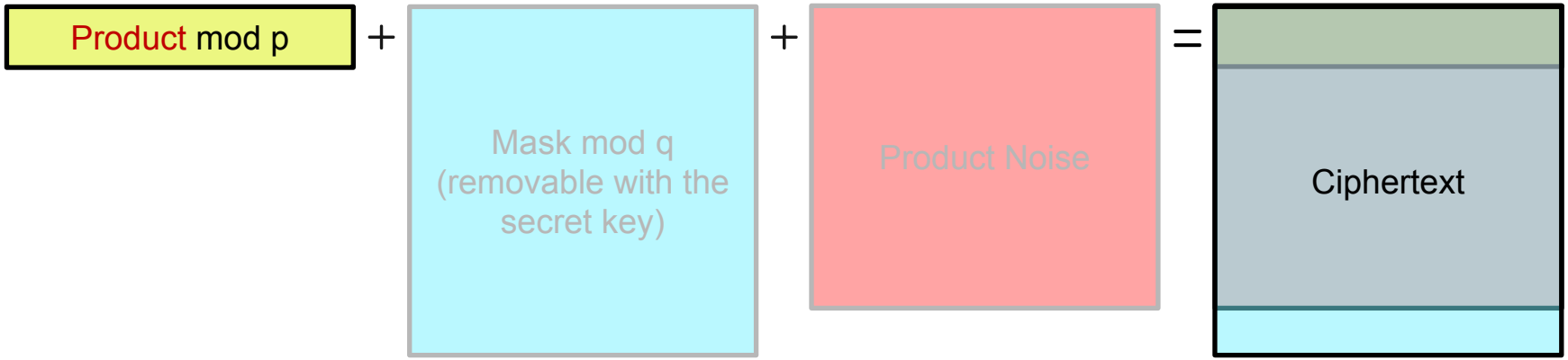
2.7 – Computation on Integer / Fractional Numbers



- Horizontal: each coefficient in a polynomial or in a vector.
- Vertical: size of coefficients.

After each level, noise increases, plaintext spreads to higher-degree terms.

2.8 – Integer / Fractional Encoding Failure



- Horizontal: each coefficient in a polynomial or in a vector.
- Vertical: size of coefficients.

At some level, plaintext reaches the highest-degree term before the noise grows too much.
Message will then be reduced mod p^n.

3.0 – How to Build HE Applications?

- ❏ How to design an HE compute model for your application?
- ❏ How to select the most efficient scheme and its implementation?
- ❏ How to encode the data prior to encryption?
- ❏ How to select the security parameters?
- ❏ How to guarantee the correctness of your implementation?
- ❏ How to optimize your implementation?

3.1 – Models of Homomorphic Computation

It is important to choose the right approach for designing your HE computation:

1. Boolean Circuits
   - Plaintext data represented as **bits**
   - Computations expressed as **Boolean circuits**
2. Modular (Exact) Arithmetic
   - Plaintext data represented as **integers modulo a plaintext modulus "*t*"** (or their vectors)
   - Computations expressed as **integer arithmetic circuits mod *t***
3. Approximate Number Arithmetic
   - Plaintext data represented as **real numbers** (or complex numbers)
   - Compute model similar to **floating-point arithmetic**

3.2 – Boolean Circuits Approach

Features:

- ❏ Fast number comparison
- ❏ Supports arbitrary Boolean circuits
- ❏ Fast bootstrapping (noise refreshing procedure)

Selected schemes:

1. Gentry-Sahai-Waters (GSW) [GSW13] - foundation for other schemes
2. Fastest Homomorphic Encryption in the West (FHEW) [DM15]
3. Fast Fully Homomorphic Encryption over the Torus (TFHE) [CGGI16,CGGI17]

3.3 – Modular (Exact) Arithmetic Approach

Features:

- ❏ Efficient SIMD computations over vectors of integers (using batching)
- ❏ Fast high-precision integer arithmetic
- ❏ Fast scalar multiplication
- ❏ Leveled design (often used without bootstrapping)

Selected schemes:

1. Brakerski-Vaikuntanathan (BV) [BV11] - foundation for other schemes
2. Brakerski-Gentry-Vaikuntanathan (BGV) [BGV12, GHS12]
3. Brakerski/Fan-Vercauteren (BFV) [Brakerski12, FV12, BEHZ16, HPS18]

3.4 – Approximate Number Arithmetic Approach

Features:

- ❏ Fast polynomial approximation
- ❏ Relatively fast multiplicative inverse and Discrete Fourier Transform
- ❏ Deep approximate computations, such as logistic regression learning
- ❏ Efficient SIMD computations over vectors of real numbers (using batching)
- ❏ Leveled design (often used without bootstrapping)

Selected schemes:

1. Cheon-Kim-Kim-Song (CKKS) [CKKS17]

3.5 – Library Matrix

Library/Scheme	FHEW	TFHE	BGV	BFV	CKKS
cuFHE		✔			
FHEW	✔				
FV-NFLlib				✔	
HEAAN					✔
HElib			✔		(✔)
PALISADE			✔	✔	(✔)
SEAL				✔	✔
TFHE(-Chimera)	✔	✔		(✔)	(✔)

3.6 – Application Development Best Practices

Main guidelines:

1. Choose the right compute model
2. Choose the plaintext encoding/batching technique
3. Determine the correctness requirements for the computation
4. Consult the security tables
5. Write the code using standard API
6. Fine-tune the parameters to optimize the performance

3.7 – Application Development: Compute Model

1. Choose the compute model
    - Boolean Circuits
    - Modular (Exact) Arithmetic
    - Approximate Number Arithmetic.
2. Determine how the data should be encoded, and whether multiple pieces of data can be packed in single ciphertexts.
    - One ciphertext per integer (high-precision arithmetic)
    - One ciphertext per vector of integers
    - One ciphertext per vector of real numbers
    - One ciphertext per matrix of real numbers
    - Etc.

3.8 – Application Development: Correctness

The functional parameters, such as “plaintext modulus” and “ciphertext modulus”, should guarantee the correctness of decrypted result.

1. Plaintext computation correctness
    - If the modular (exact) arithmetic approach is selected, verify that the result is correct:
        - 11*7 mod 50 ≠ 77
    - Always build a reference implementation in the clear. This helps a lot in debugging the HE-enabled application code.
2. Encrypted computation correctness
    - Each ciphertext operation increases the noise. Verify that the fresh ciphertext modulus is chosen to be large enough, or bootstrapping is applied before the noise can cause a decryption failure.

3.9 – Application Development: Security

The ring dimension (degree of polynomial) should be chosen according to the security tables published at HomomorphicEncryption.org (some libraries can select it automatically).

distribution	n	security level	logq	uSVP	dec	dual
(-1, 1)	1024	128	27	131.6	160.2	138.7
		192	19	193.0	259.5	207.7
		256	14	265.6	406.4	293.8
	2048	128	54	129.7	144.4	134.2
		192	37	197.5	233.0	207.8
		256	29	259.1	321.7	273.5
	4096	128	109	128.1	134.9	129.9
		192	75	194.7	212.2	198.5
		256	58	260.4	292.6	270.1
	8192	128	218	128.5	131.5	129.2
		192	152	192.2	200.4	194.6
		256	118	256.7	273.0	260.6

3.10 – Application Development: Performance

Fine-tune the parameters affecting the performance:

- ❏ Plaintext encoding settings
- ❏ Choose smallest ring dimension and ciphertext modulus that meet the correctness and security requirements
- ❏ Fine-tune scheme-specific parameters, such as relinearization window
- ❏ Update the order of HE maintenance procedures, such as relinearization, modulus switching/rescaling, and bootstrapping
- ❏ Turn on multi-threading
- ❏ Take advantage of library-specific performance optimization tools, such as memory pools or RNS representation of large integers
- ❏ Use specialized hardware, such as GPU, if supported by the library

4.0 – Standardization

Applications of HE in regulated industries requires standardization

- ❏ Finance
- ❏ Health-care
- ❏ Government
- ❏ Military

Must guarantee HE to be at least as secure as AES, RSA!

4.1 – Standardization Workshops

- In July 2017 at Microsoft
- In March 2018 at MIT
- October 20, 2018 at U Toronto

Outcomes:

- HomomorphicEncryption.org community
- White papers
- Mailing list
- Attended and endorsed by leading experts in crypto and security

4.2 – White papers

Three white papers from the first workshop:

- ❏ Security of Homomorphic Encryption
- ❏ API for Homomorphic Encryption
- ❏ Applications of Homomorphic Encryption

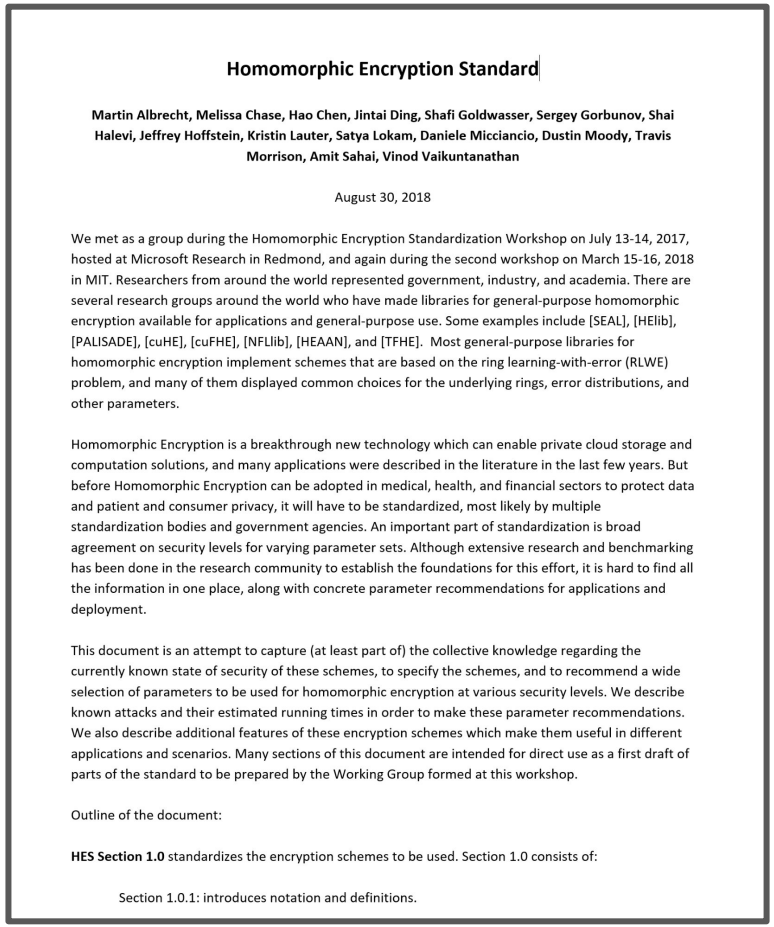
Guiding principles of the standardization effort:

- ❏ Security is priority
- ❏ API standardization needed for making HE developer-friendly
- ❏ Motivated by practical use-cases

4.3 – Security

What is the security standard?

- ❏ Describes encryption schemes
- ❏ Describes best known attacks
- ❏ Describes tables of parameters in terms of standard security levels
- ❏ Written by leading security experts
- ❏ Available at HomomorphicEncryption.org



Homomorphic Encryption Standard

Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, Vinod Vaikuntanathan

August 30, 2018

We met as a group during the Homomorphic Encryption Standardization Workshop on July 13-14, 2017, hosted at Microsoft Research in Redmond, and again during the second workshop on March 15-16, 2018 in MIT. Researchers from around the world represented government, industry, and academia. There are several research groups around the world who have made libraries for general-purpose homomorphic encryption available for applications and general-purpose use. Some examples include [SEAL], [HElib], [PALISADE], [cuHE], [cuFHE], [NFLlib], [HEAAN], and [TFHE]. Most general-purpose libraries for homomorphic encryption implement schemes that are based on the ring learning-with-error (RLWE) problem, and many of them displayed common choices for the underlying rings, error distributions, and other parameters.

Homomorphic Encryption is a breakthrough new technology which can enable private cloud storage and computation solutions, and many applications were described in the literature in the last few years. But before Homomorphic Encryption can be adopted in medical, health, and financial sectors to protect data and patient and consumer privacy, it will have to be standardized, most likely by multiple standardization bodies and government agencies. An important part of standardization is broad agreement on security levels for varying parameter sets. Although extensive research and benchmarking has been done in the research community to establish the foundations for this effort, it is hard to find all the information in one place, along with concrete parameter recommendations for applications and deployment.

This document is an attempt to capture (at least part of) the collective knowledge regarding the currently known state of security of these schemes, to specify the schemes, and to recommend a wide selection of parameters to be used for homomorphic encryption at various security levels. We describe known attacks and their estimated running times in order to make these parameter recommendations. We also describe additional features of these encryption schemes which make them useful in different applications and scenarios. Many sections of this document are intended for direct use as a first draft of parts of the standard to be prepared by the Working Group formed at this workshop.

Outline of the document:

HES Section 1.0 standardizes the encryption schemes to be used. Section 1.0 consists of:

Section 1.0.1: introduces notation and definitions.

4.4 – Third Standardization Workshop

- On Saturday at University of Toronto
- Significant progress towards API standardization
- Automation and developer tools
- Compiler for homomorphic encryption
- If you still want to register, come talk to me

5.0 – Challenges and Open Problems

- HE is hard to use
  - Standardized API
  - Languages and compilers for writing and optimizing HE programs easily
  - Higher-level automation to help developers design efficient HE-based solutions
  - Library interoperability
- HE is not practical for all computations
  - Only small/low depth arithmetic and Boolean circuits are feasible
  - E.g. division, comparison can be costly (scheme-dependent)
  - E.g. data filtering is impossible in the traditional sense
  - Most computational workloads are not designed in an HE-friendly way

6.0 – What to Expect in Part II (After the Break)

30-minute break: we will help you download and install HE libraries

[SEAL] -- http://sealcrypto.org

[PALISADE] -- https://git.njit.edu/palisade/PALISADE

[TFHE] -- https://tfhe.github.io/tfhe

[cuFHE] -- https://github.com/vernamlab/cuFHE (requires an NVIDIA GPU)

An Overview of HE Libraries

- At least 10 open source HE libraries available
  - 4 libraries presented here
- Libraries not included:
  - HeaAn - (https://github.com/kimandrik/HEAAN)
  - HElib - (https://github.com/shaih/HElib)
  - Λ ○ λ (“LOL”) - (https://github.com/cpeikert/Lol)
    - Used by the “ALCHEMY” compiler (Crockett, et al.)
  - NFLlib - (https://github.com/quarkslab/NFLlib)
  - FHEW - (https://github.com/lducas/FHEW)
  - And more...

SEAL

Simple Encrypted Arithmetic Library

Kim Laine / kim.laine@microsoft.com

http://sealcrypto.org

Quick Background

- Homomorphic Encryption library from Microsoft Research
- First version released in 2015; SEAL 3.0 just released
- Developed in standard C++

- Implements BFV and CKKS schemes
  - BFV for exact (e.g. integer) computations
  - CKKS for approximate fixed-point computations

- Header-files extensively commented
- Comes with detailed examples

Downloading SEAL

- SEAL 3.0 source code can be downloaded as .tar.gz (Linux and OS X) or .zip (Windows) packages from http://sealcrypto.org
- SEAL is completely self-contained: no external dependencies
- GitHub release coming soon

Building SEAL and Linking with Applications

- On Visual Studio use accompanying solution and project files
  - Requires Visual Studio 2017
- On Linux/OS X use g++/clang++ and CMake
  - Requires g++ >= 6 or clang++ >= 5
- Uses some features from C++17 but can be compiled as C++14 if necessary
- With CMake easy to configure and link with your application

```
cmake_minimum_required(VERSION 3.10)
project(CCSTutorial)
add_executable(example example.cpp)
find_package(SEAL 3.0.0 REQUIRED)
target_link_libraries(example SEAL::seal)
```

Learning to Use SEAL

- Best way to learn to use SEAL is going over `SEALExamples/main.cpp`
- Doing something with SEAL is not so hard …
- But doing it *well* can be
- Learning to use SEAL efficiently will require a lot of work
- Recommendation: Learn BFV scheme first; CKKS after that
- In the future: Compilers and better developer tools will help
- StackOverflow tag [seal]

Now let’s look at some code ...

SEAL

Simple Encrypted Arithmetic Library

http://sealcrypto.org

NJIT

PALISADE

Yuriy Polyakov (NJIT)

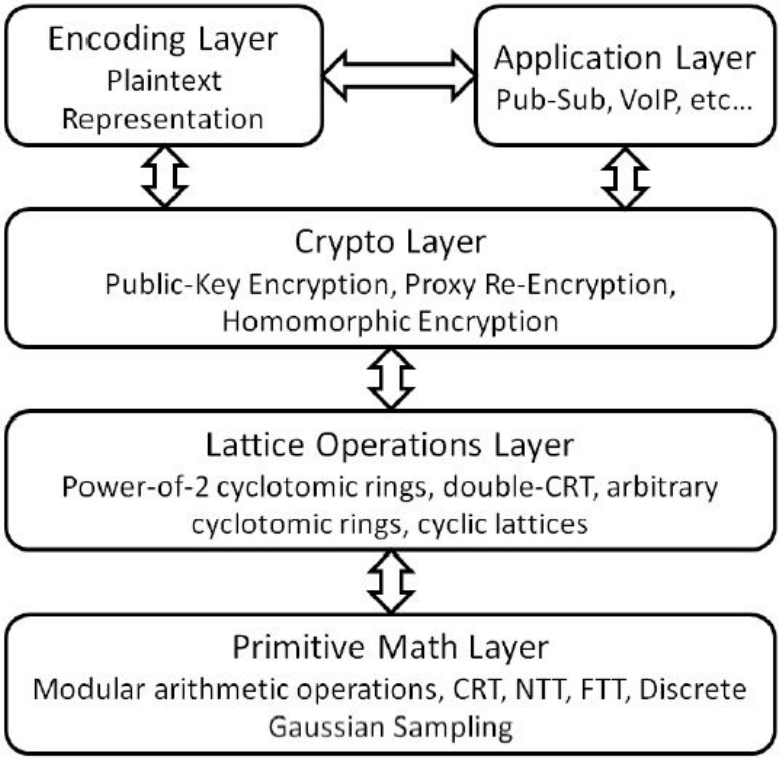
CCS'18 Tutorial: "Building Applications with Homomorphic Encryption"

October 19, 2018

PALISADE Lattice Cryptography Library (NJIT)

- Project-based Development since 2014
  - Next generation of DARPA PROCEED SIPHER project
  - Cryptographic program obfuscation (DARPA Safeware)
  - Homomorphic Encryption for statistical analysis (Sloan, IARPA)
  - Proxy Re-Encryption for Pub/Sub systems (Simons, NSA)
  - HE backend for Secure Programming in Julia (IARPA)
- Implementation Partners and Collaborators
  - Academia: MIT, UCSD, WPI, NUS, Sabanci U
  - Industry: Raytheon (BBN), IBM Research, Lucent, Vencore Labs, Galois, Two Six Labs
- BSD 2-clause license
- Cross-Platform Support

Modular Design



Capabilities

- Public Key Encryption/Homomorphic Encryption
  - 3 variants of BFV scheme
  - BGV
  - LTV, Stehle-Steinfeld
  - Null
  - Proxy Re-Encryption based on all of the above HE schemes
- Capabilities that will be released within next few months (in v1.4 and v2.0)
  - CKKS HE scheme
  - Identity-based encryption, 2 variants of attribute-based encryption
  - GPV digital signature

Key Concepts/Classes

- CryptoContext
  - A wrapper that encapsulates the scheme, crypto parameters, encoding parameters, and keys
  - Provides the same API for all HE schemes
- Ciphertext
  - Stores the ciphertext polynomials
- Plaintext
  - Stores the plaintext data (both raw and encoded)
  - Supports multiple encodings in a polymorphic manner, including PackedEncoding, IntegerEncodering, CoefPackedEncoding, etc.

Sample Program: Step 1 – Set CryptoContext

```
//Set the main parameters
int plaintextModulus = 65537;
double sigma = 3.2;
SecurityLevel securityLevel = HEStd_128_classic;
uint32_t depth = 2;

//Instantiate the crypto context
CryptoContext<DCRTPoly> cryptoContext =
CryptoContextFactory<DCRTPoly>::genCryptoContextBFVrns(
        plaintextModulus, securityLevel, sigma, 0, depth, 0, OPTIMIZED);

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

Sample Program: Step 2 – Key Generation

```
// Initialize Public Key Containers
LPKeyPair<DCRTPoly> keyPair;

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

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

Sample Program: Step 3 – Encryption

```
// First plaintext vector is encoded
std::vector<uint64_t> vectorOfInts1 = {1,2,3,4,5,6,7,8,9,10,11,12};
Plaintext plaintext1 = cryptoContext->MakePackedPlaintext(vectorOfInts1);
// Second plaintext vector is encoded
std::vector<uint64_t> vectorOfInts2 = {3,2,1,4,5,6,7,8,9,10,11,12};
Plaintext plaintext2 = cryptoContext->MakePackedPlaintext(vectorOfInts2);
// Third plaintext vector is encoded
std::vector<uint64_t> vectorOfInts3 = {1,2,5,2,5,6,7,8,9,10,11,12};
Plaintext plaintext3 = cryptoContext->MakePackedPlaintext(vectorOfInts3);

// The encoded vectors are encrypted
auto ciphertext1 = cryptoContext->Encrypt(keyPair.publicKey, plaintext1);
auto ciphertext2 = cryptoContext->Encrypt(keyPair.publicKey, plaintext2);
auto ciphertext3 = cryptoContext->Encrypt(keyPair.publicKey, plaintext3);
```

Sample Program: Step 4 – Evaluation

```
// Homomorphic additions
auto ciphertextAdd12 = cryptoContext->EvalAdd(ciphertext1,ciphertext2);
auto ciphertextAddResult = cryptoContext->EvalAdd(ciphertextAdd12,ciphertext3);

// Homomorphic multiplications
auto ciphertextMul12 = cryptoContext->EvalMult(ciphertext1,ciphertext2);
auto ciphertextMultResult = cryptoContext->EvalMult(ciphertextMul12,ciphertext3);
```

Sample Program: Step 5 – Decryption

```
// Decrypt the result of additions
Plaintext plaintextAddResult;
cryptoContext->Decrypt(keyPair.secretKey, ciphertextAddResult, &plaintextAddResult);

// Decrypt the result of multiplications
Plaintext plaintextMultResult;
cryptoContext->Decrypt(keyPair.secretKey, ciphertextMultResult,
&plaintextMultResult);

// Output results
cout << plaintextAddResult << endl;
cout << plaintextMultResult << endl;
```

Real Application Implemented in PALISADE

- Secure Genome-Wide Association Study (GWAS)
  - 245 individuals
  - 3 phenotypic covariates
  - 15K SNPs (genetic variations)
  - iDASH'18 Track 2
- Goal: Identify which SNPs may be associated to a certain disease/condition
- Result: A highly accurate solution with the following performance
  - End-to-end runtime on a 4-core machine: under 4 minutes
  - RAM utilization: under 10 GB

Design Decisions

Guideline	Decision
Choose the right compute model	Approximate Number Arithmetic; CKKS
Choose the plaintext encoding/batching technique	Packed encoding: two variants
Determine the correctness requirements for the computation	Ran computations in the clear to find the plaintext parameters providing adequate accuracy
Consult the security tables	Chose $N = 2^{15}$ and $\log_2 q = 850$ based on the security standard
Write the code using standard API	Implemented using the PALISADE CryptoContext wrapper
Fine-tune the parameters to optimize the performance	Used a full RNS variant of CKKS; Applied many optimizations: encoding switching, lazy key switching, etc.

More Information

- Download the library
  - https://git.njit.edu/palisade/PALISADE
- Download the manual
  - https://git.njit.edu/palisade/PALISADE/blob/master/doc/palisade_manual.pdf
- Contact by email if you have any questions
  - palisade@njit.edu
- Q&A

TFHE

Fast Fully Homomorphic Encryption over the Torus

https://github.com/tfhe/tfhe

http://lab.algonics.net/slides/index-ccs.html#/

cuFHE

CUDA-enabled Fully Homomorphic Encryption
https://github.com/vernamlab/cuFHE

Wei Dai (WPI) / wdai@wpi.edu

Features

- Implementing the TFHE scheme: binary gates.
  - Single-bit encryption / decryption / evaluation.
- Developed in C++, interfaces wrapped in Python.
- Homomorphic binary gates on CUDA-enabled GPUs.
- Performance

Library	Platform (Price)	Amortized Gate Latency	Throughput	Speedup
TFHE	CPU (unknown)	13 ms	77 gates / sec	1×
cuFHE	Titan Xp ($1,200)	500 μs	2,000 gates / sec	26×
cuFHE	Tesla V100 ($8,000)	137 μs	7,300 gates / sec	95×

How to Use

Every circuit can be expressed with binary gates.

```
import lib.fhepy_gpu as fhe
pubkey, prikey = fhe.KeyGen()
m1 = random.randint(0,1)
m2 = random.randint(0,1)
c1 = fhe.Encrypt(m1, prikey)
c2 = fhe.Encrypt(m2, prikey)
c = c1 & c2
c = c1 | c2
c = c1 ^ c2
fhe.NAND(c, c1, c2)
c = ~(c1 & c2)
result = c.Decrypt(prikey)
```

```
PriKey pri_key;
PubKey pub_key;
KeyGen(pub_key, pri_key);
Ptxt* pt = new Ptxt[2];
pt[0].message_ = rand() % Ptxt::kPtxtSpace;
pt[1].message_ = rand() % Ptxt::kPtxtSpace;
Ctxt* ct = new Ctxt[2];
Encrypt(ct[0], pt[0], pri_key);
Encrypt(ct[1], pt[1], pri_key);
Nand(ct[0], ct[0], ct[1]);
And(ct[0], ct[0], ct[1]);
Xor(ct[0], ct[0], ct[1]);
Decrypt(pt[0], ct[0], pri_key);
```

Hardware Acceleration for HE

Performance Bottlenecks of HE

- Data Efficiency
  - Ciphertexts
  - Relinearization keys, bootstrapping keys
  - Requires high memory bandwidth
- Computational Efficiency
  - Polynomial ring arithmetic
  - Integer modular arithmetic
  - Requires high computational power / cost ratio

Comparison of Platforms

	CPU	GPU	FPGA
Computation / Price	Bad	Good	OK
Computation / Power	Bad	Good	Better
Memory Efficiency	Better	Good	Bad
Portability & Scalability	Good	Good	Bad
Programming Effort	Good	OK	Bad
Performance Growth	Limited	Good	Limited
Popularity	Good	OK	Bad

Previous Works using GPUs

More than 30× speedup over a single-threaded CPU

Comparing homomorphic multiplications of ciphertexts

GPU Works	Scheme	Speedup over CPU
cuHE	general	30×
cuFHE	TFHE	30× (100× on V100)
nuFHE	TFHE	100× on P100
SEAL Dev	full-RNS BFV	50×
ASTAR	full-RNS BFV	30× on P100

By default, the GPU used is Titan Xp, unless specified.

Previous Works using FPGAs

Better performance / power consumption vs. GPUs.

Worse performance / price vs. GPUs.

Comparing homomorphic multiplications of ciphertexts

FPGA Works	Scheme	Speedup over CPU
Öztürk, Doröz, Sunar, Savaş	LTV variant	100×
SEAL Dev	full-RNS BFV	20× (~120×)

Performance varies greatly on different FPGA models.

Application-specific Integrated Circuit (ASIC)

- CPU, GPU and FPGA are general purpose.
- Design driven by
  - CPU: general
  - GPU: AI, Machine Learning, Computer Graphics
  - FPGA: DSP
- There has never been a hardware platform specific for HE.
- Base cost starts at a few millions (USD).
- Ideal in future, when there is a market.

Compilers for HE

Why have compilers?

1. Writing HE code directly can be tedious

Dot product (in SEAL)

```
void
dot(SEALContext &context,
    vector<Ciphertext> &vec1, vector<Ciphertext> &vec2,
    Ciphertext &dotprod)
{
  Evaluator evaluator(context);
  evaluator.multiply(vec1[0], vec2[0], res);
  for (int i = 1; i < vec1.size(); ++i) {
    Ciphertext tmp;
    evaluator.multiply(vec1[i], vec2[i], tmp);
    evaluator.add(dotprod, tmp);
  }
}
```

Parameter selection (in SEAL)

```
void
dot_parms(EncryptionParameters &parms, size_t size)
{
  ChooserEncoder encoder(3);
  ChooserEncryptor encryptor;
  ChooserEvaluator evaluator;
  vector<ChooserPoly> vec1, vec2;
  ChooserPoly dotprod;
  // ...
  res = evaluator.multiply(vec1[0], vec2[0]);
  for (int i = 1; i < size; ++i) {
    ChooserPoly tmp;
    tmp = evaluator.multiply(vec1[i], vec2[i]);
    res = evaluator.add(dotprod, tmp);
  }
  evaluator.select_parameters({ dotprod },0,parms);
}
```

Code very similar: easy to introduce bugs

Why have compilers?

1. Writing HE code directly can be tedious
2. Code not easy to analyze/optimize

```
void
multmany(SEALContext &context,
         vector<Ciphertext> &vec,
         Ciphertext &res)
{
  Evaluator evaluator(context);
  for (int i = 1; i < vec.size(); ++i) {
    evaluator.multiply(vec[i], res, res);
  }
}
```

Goal: Compile as tree of mults to reduce depth

Requires full C++ code analysis

Why have compilers?

1. Writing HE code directly can be tedious
2. Code not easy to analyze/optimize
3. Programs must have finite number of operations

```
int
gcd(int a, int b)
{
    return b == 0 ? a : gcd(b, a % b);
}
```

How do we compute the max depth of this computation?

Why have compilers?

1. Writing HE code directly can be tedious
2. Code not easy to analyze/optimize
3. Programs must have finite number of operations
4. Branching is hard:

- **`if`** / **`else`** / **`switch`**: need to execute all branches
- **`for`** / **`while`**: need computable loop bound

High-level languages + compilers help address all of these concerns!

(Select) compiler approaches

- RAMPARTS
- HE-IR
- Cingulata
- (Several other approaches in the literature)

RAMPARTS

- Compiles **`Julia`** → **`PALISADE`**

julia: Programming language targeting scientific users (similar to MATLAB)

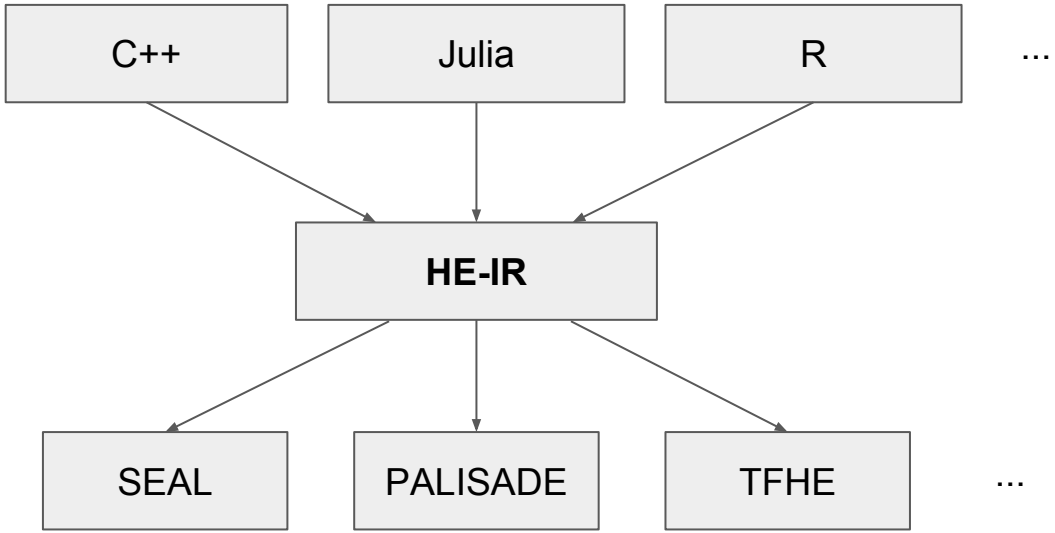
```
function sharpen(image::Array{Int,2})::Array{Int,2}
  weight = [[1 1 1]; [1 -8 1]; [1 1 1]]
  image2 = deepcopy(image)
  dx,dy = size(image)
  for x = 2:dx-1, y = 2:dy-1
    value = 0
    for j = -1:1, i = -1:1
      value += weight[i+2,j+2] * image[x+i,y+j]
    end
    image2[x,y] = image[x,y] - (value >> 1)
  end
  image2
end
```

Uses symbolic simulation to convert **`Julia`** program into arithmetic circuit

HE-IR

- How should we define an ***intermediate representation*** for HE?
  - “Frontend” compilers compile from <*insert favorite language here*> to IR
  - “Backend” compilers compile from IR to <*insert favorite HE library here*>
- IR should be:
  - Easy to analyze (for parameter selection / optimizations)
  - Accurately capture HE capabilities / limitations
- Work in progress:
  - Initially investigated “assembly language” approach
    - Unconstrained branching (e.g., jumps and labels) makes analysis hard
  - Currently targeting SEAL only
  - Feedback/suggestions welcome!

HE-IR: Architecture



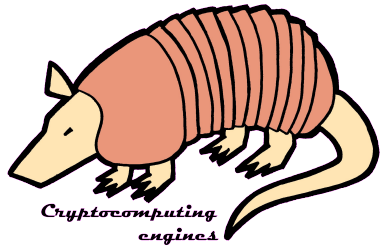
HE-IR: Example

```
input vec1 : ct[<=20]
input vec2 : ct[<=20]
output dotprod : ct

if vec1.length() != vec2.length() {
      fail "both vectors must be the same length"
}

if vec1.length() < 1 {
      fail "vector length must be greater than zero"
}

let! res : ct = vec1[0] * vec2[0]

for i : int in [1 .. vec1.length() - 1] {
  res = res + (vec1[i] * vec2[i])
}

dotprod = res
```

Parameter Selection Code

Executable Program

Cingulata



- Toolchain for compiling and running programs over FHE
  - C++ input, instrumented ints build boolean circuit
  - Efficient multiplicative depth minimization modules
  - Parallel runtime environment
  - Tools for generating keys, encryption, decryption and execution
- Cingulata v2 to come...
  - Python input
  - Generic interface for HE libraries
  - On-the-fly optimized execution for bootstrapped schemes (TFHE)
- Available here:

https://github.com/CEA-LIST/Cingulata

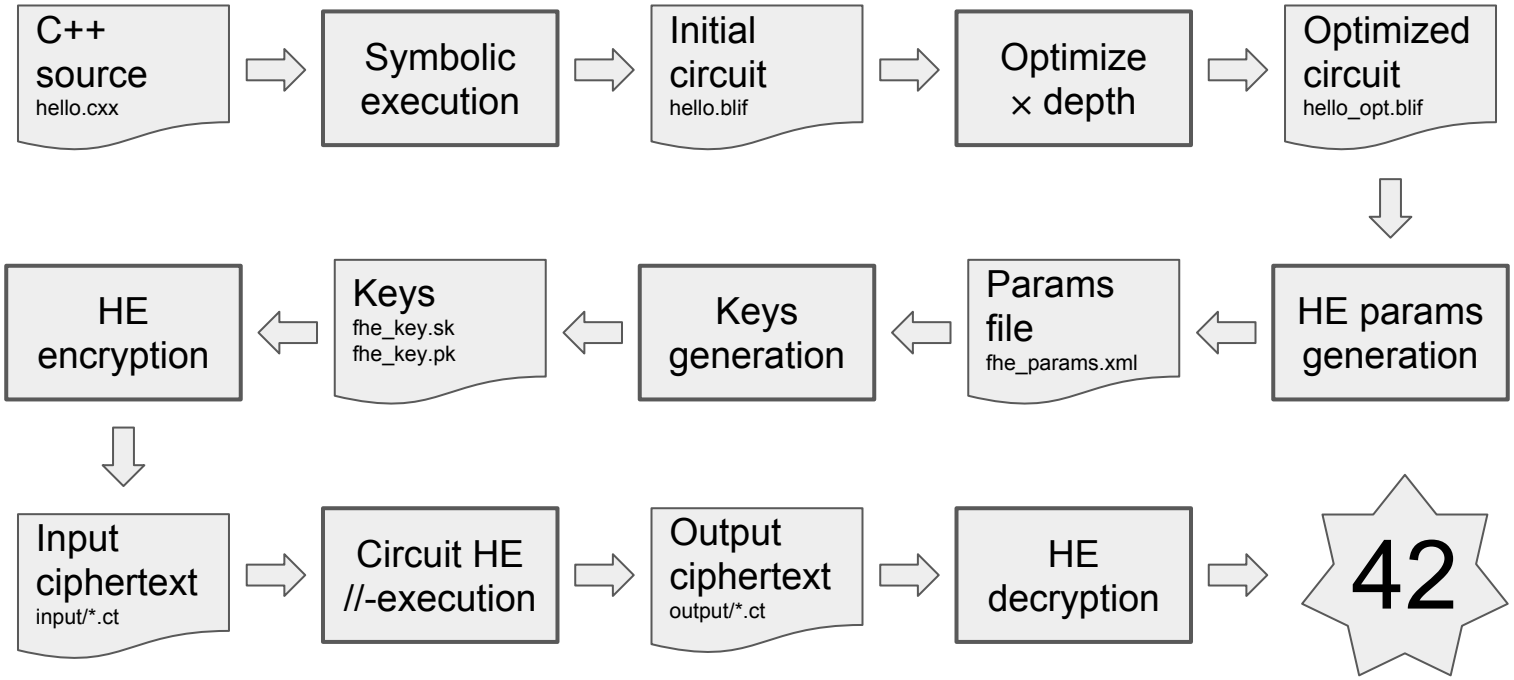
Cingulata - bubble sorting arrays

- Clear and Cingulata versions

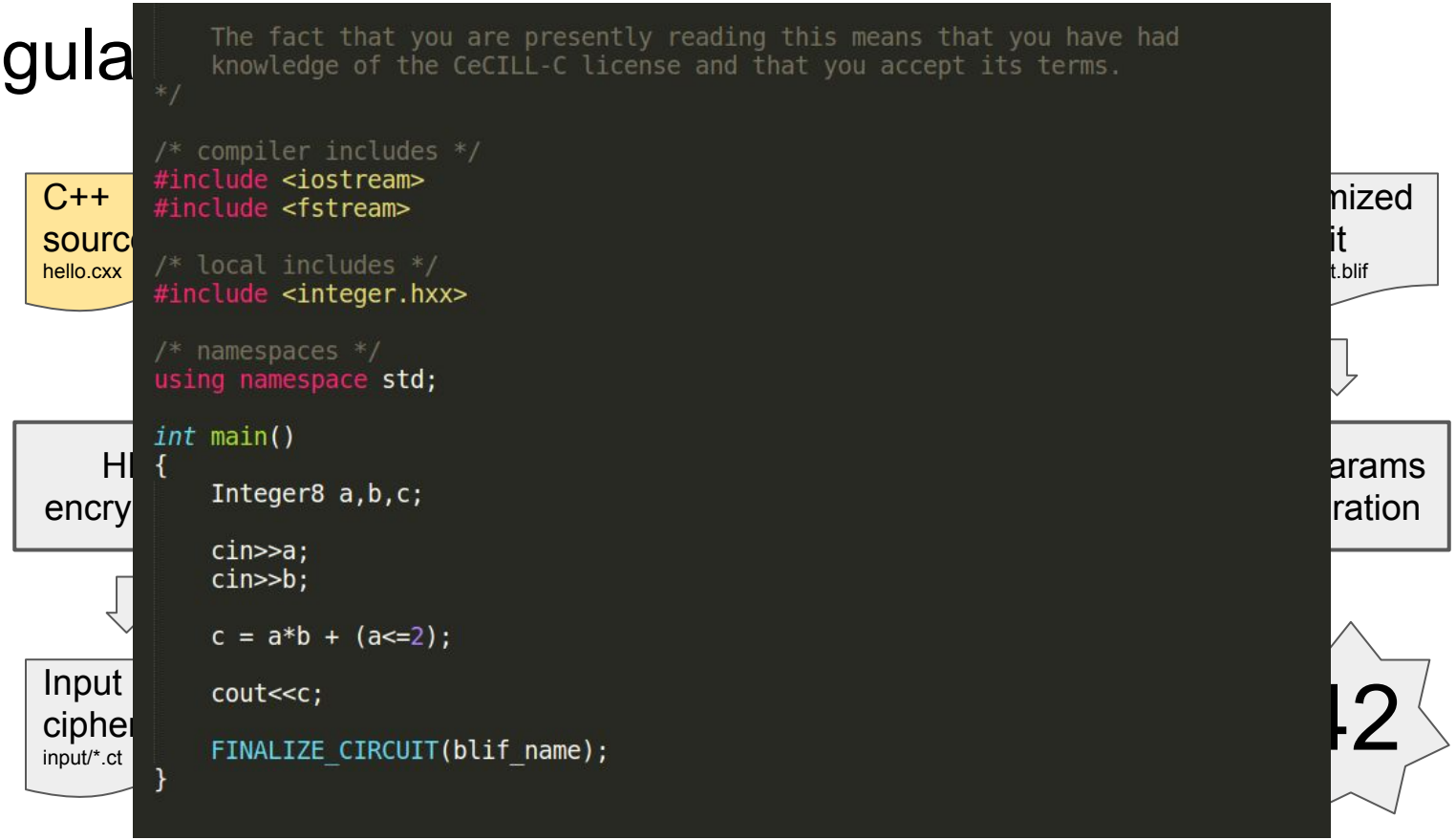
```
template<typename integer>
void bsort(integer* const arr, const int n)
{
  for(int i=0;i<n-1;i++)
  {
    for(int j=1;j<n-i;j++)
    {
      integer swap = arr[j-1]>arr[j];
      if (swap) {
        integer t = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = t;
      }
    }
  }
}
```

```
template<typename integer>
void bsort(integer* const arr, const int n)
{
  for(int i=0;i<n-1;i++)
  {
    for(int j=1;j<n-i;j++)
    {
      integer swap = arr[j-1]>arr[j];
      integer t = select(swap,arr[j-1],arr[j]);
      arr[j-1] = select(swap,arr[j],arr[j-1]);
      arr[j] = t;
    }
  }
}

// here select(c,a,b) ≡ c?a:b
```

Cingulata compilation and execution process

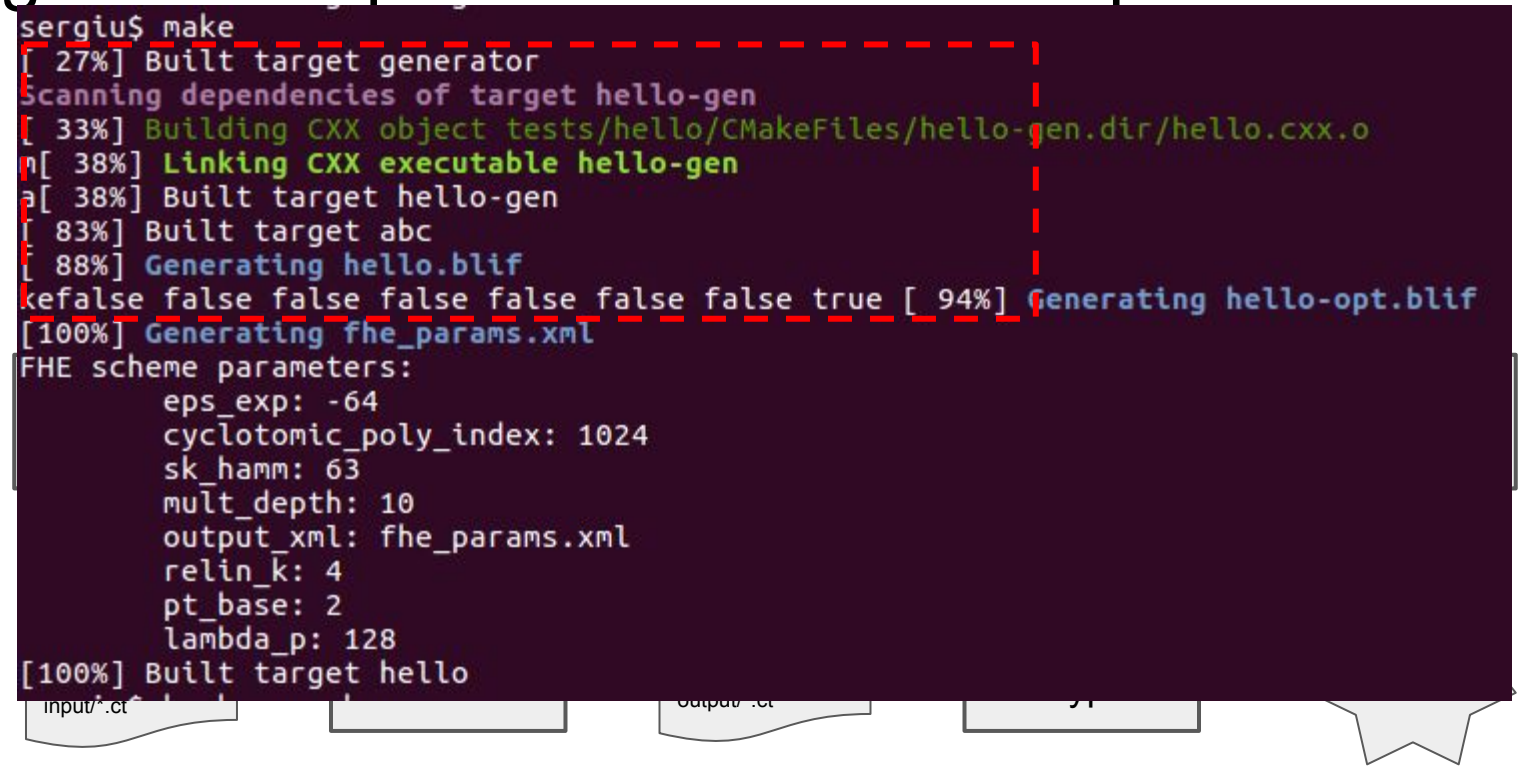


Cingula




```
    The fact that you are presently reading this means that you have had
    knowledge of the CeCILL-C license and that you accept its terms.
*/

/* compiler includes */
#include <iostream>
#include <fstream>

/* local includes */
#include <integer.hxx>

/* namespaces */
using namespace std;

int main()
{
    Integer8 a,b,c;

    cin>>a;
    cin>>b;

    c = a*b + (a<=2);

    cout<<c;

    FINALIZE_CIRCUIT(blif_name);
}
```

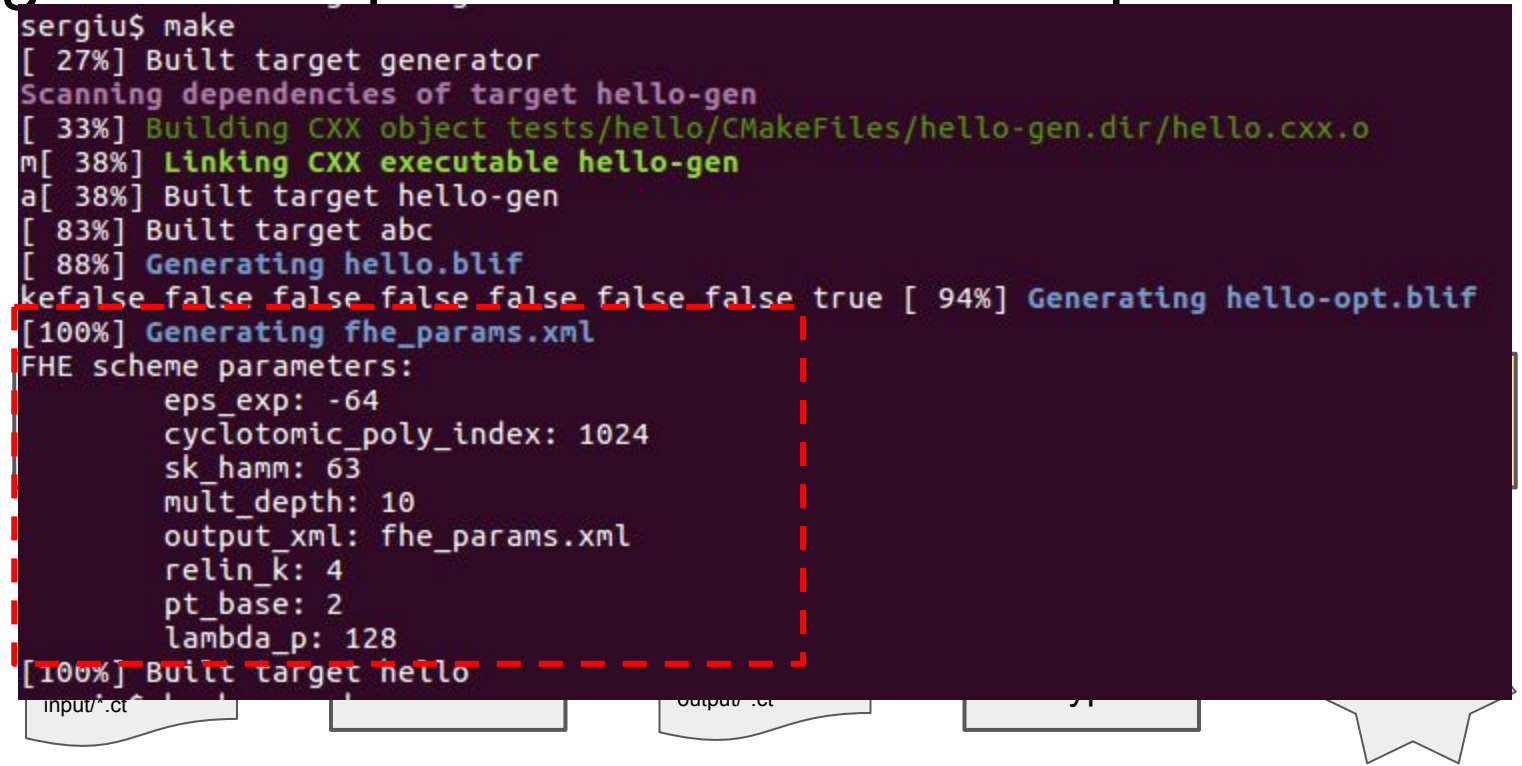
Cingulata compilation and execution process



Cing

```
.model CIRCUIT
.inputs i_2 i_3 i_4 i_5 i_6 i_7 i_8 i_9 i_10 i_11 i_12 i_13 i_14 i_15 i_16 i_17
.outputs m_0 m_1 m_2 m_3 m_4 m_5 m_6 m_7
.names i_0
0
.names i_1
1
.names i_0 n100000
0 1
.names i_9 n100000 n100001
11 1
.names i_8 n100002
0 1
.names n100002 i_1 n100003
01 1
10 1
.names n100003 n100001 n100004
11 1
.names i_1 n100005
0 1
.names i_8 n100005 n100006
11 1
.names n100006 n100004 n100007
01 1
10 1
.names i_7 n100008
0 1
```

Cingulata compilation and execution process



Cingulata

C++ source
hello.cxx

HE encryption

Input ciphertext
input/*.ct

```
sergiu$ bash run.sh
FHE key generation
Input encryption
Command line arguments:
FHE parameters file fhe_params.xml
Public key file fhe_key.pk
Encrypting message [0] into file input/i_2.ct
Encrypting message [0] into file input/i_6.ct
Encrypting message [0] into file input/i_14.ct
Encrypting message [0] into file input/i_10.ct
Encrypting message [1] into file input/i_15.ct
Encrypting message [0] into file input/i_11.ct
Encrypting message [1] into file input/i_7.ct
Encrypting message [0] into file input/i_3.ct
Encrypting message [0] into file input/i_12.ct
Encrypting message [1] into file input/i_16.ct
Encrypting message [1] into file input/i_8.ct
Encrypting message [0] into file input/i_4.ct
Encrypting message [0] into file input/i_13.ct
Encrypting message [0] into file input/i_17.ct
Encrypting message [1] into file input/i_9.ct
Encrypting message [0] into file input/i_5.ct
Homomorphic execution...
Total execution time 3.35204 seconds
READ time 0.0185181 seconds, #execs 16
COPY time 0.0605621 seconds, #execs 213
XOR gates execution time 0.0430649 seconds, #execs 93
NOT gates execution time 0.00558572 seconds, #execs 45
AND gates execution time 10.3223 seconds, #execs 70
OR gates execution time 0.794401 seconds, #execs 5
WRITE time 0.00498272 seconds, #execs 8
Maximal number of simultaneously allocated ciphertexts 45

real    0m3.366s
user    0m11.044s
sys     0m0.112s
Output decryption
42
```

Optimized circuit
hello_opt.blif

HE params generation

42

Cingulata - conclusion

Compiling and running HE applications in Cingulata

- As simple as two lines

```
make hello
bash run.sh
```

https://github.com/CEA-LIST/Cingulata

Demo