FHE & Cheddar

A Tutorial on Fully Homomorphic Encryption &
Cheddar, an Optimized GPU Library

ISCA 2026 Tutorial

Seoul National University
Scalable Computer Architecture Laboratory

Organizers

Jung Ho Ahn
gajh@snu.ac.kr

Jongmin Kim
jongmin.kim@snu.ac.kr

Wonseok Choi
wonseok.choi@snu.ac.kr

Tentative schedule

Time	Topic	Presenter
8:00–8:20	Welcome and introduction to FHE	Jung Ho Ahn
8:20–9:00	Computational aspects of CKKS	Jongmin Kim
9:00–9:15	Break	-
9:15–10:00	Cryptographic construction of CKKS	Jongmin Kim
10:00–10:30	Coffee Break	-
10:30–10:50	Introduction to Cheddar	Wonseok Choi
10:50–11:40	Live coding: Implementing FHE applications with Cheddar	Wonseok Choi
11:40–11:50	Summary	Jung Ho Ahn

Quantum

Post-Quantum Cryptography

PQC standards from NIST

- FIPS (Federal Information Processing Standards) @ *Aug 2024*
  - FIPS 203: Module-lattice-based key-encapsulation mechanism standard (ML-KEM)
  - FIPS 204: Module-lattice-based digital signature standard (ML-DSA)
  - FIPS 205: Stateless hash-based digital signature standard (SLH-DSA)

Post-Quantum Cryptography

NIST = National Institute of Standards and Technology

- FIPS
  - FIPS 203: Module-lattice-based key-encapsulation mechanism standard (ML-KEM)
  - FIPS 204: Module-lattice-based digital signature standard (ML-DSA)
  - FIPS 205: Stateless hash-based digital signature standard (SLH-DSA)

Homomorphic encryption (**HE**) is a lattice-based scheme
that enables computation (e.g., add & mult) on encrypted data.

- Cryptography confs
  - Crypto
  - Eurocrypt
  - Asiascrypt

- Security confs
  - CCS
  - IEEE S&P (“Oakland”)
  - USENIX Security
  - NDSS

HE

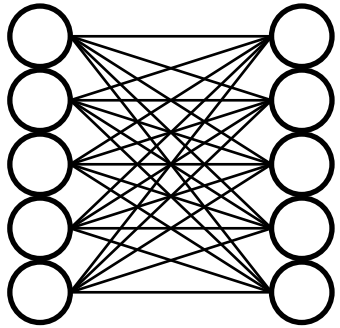
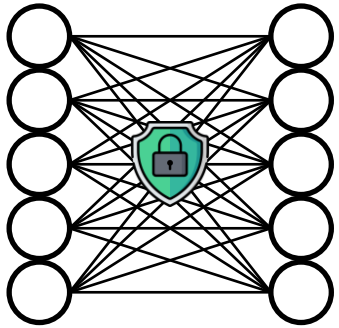
- HE is math-based
  - Algebra
  - Number theory
  - Ring
  - Polynomial
  - …

- Systems (and architecture) confs
  - ISCA
  - MICRO
  - HPCA
  - ASPLOS
  - …

Homomorphic encryption (**HE**) is a lattice-based scheme
that enables computation (e.g., add & mult) on encrypted data.

Scheme	Data type in a message	Arithmetic operation
BGV, B/FV	Finite integer $\mathbb{Z}_q$	Integer (mod q)
FHEW, TFHE	Binary $\{0,1\}$	Boolean (XOR, AND)
CKKS	Real / complex $\mathbb{R}$ / $\mathbb{C}$	Approximate (fixed-point)

FHE is Expensive

	Unencrypted computation	FHE-encrypted computation
		
1. Encryption results in severe data size expansion	1MB message	24MB ciphertext
2. HE operations are much more complex	Simple element-wise operation	Complex sequence of NTT / BConv / …
3. Bootstrapping		+ additional cost of bootstrapping
e.g.) ResNet-20 Inference	< 0.1 second	**38 minutes [1]**

[1] Lee, E., Lee, J.W., Lee, J., Kim, Y.S., Kim, Y., No, J.S. and Choi, W. Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. ICML 22'.

(M’07–SM'14) received the B.S. degree (’00) in electrical engineering from Seoul National University and the M.S. (‘02) and Ph.D. (‘07) degrees in electrical engineering from Stanford University, Stanford, CA, USA.

Since 2009, he has been an assistant/associate/full professor at the Graduate School of Convergence Science and Technology, Seoul National University. He

https://en.wikipedia.org/wiki/Homomorphic_encryption

First generation

Craig Gentry, using lattice-based cryptography, described the first plausible construction for a fully homomorphic encryption (FHE) scheme in 2009.

Fourth-generation

In 2016, Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song (CKKS) proposed an approximate homomorphic encryption scheme that supports a special kind of fixed-point arithmetic that is commonly referred to as block floating point arithmetic.

CKKS stands for Cheon, Kim, Kim, Song.

교수 및 직원

Department of Mathematical Information Science (since 2013)

교수진

안정호 교수

Computer Architecture and Systems 전공

- 서울대학교 융합과학기술대학원 지능정보융합학과
- Stanford University
- 이메일: gajh@snu.ac.kr

천정희 교수

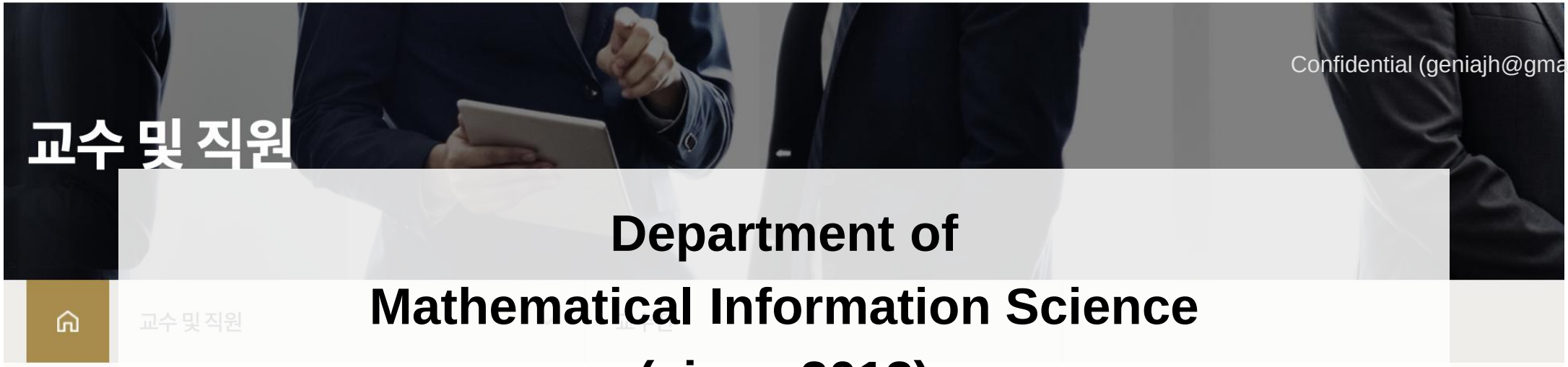
정수론, 암호론 전공

- 서울대학교 자연과학대학 수리과학부
- 수학기반산업데이터해석연구센터 센터장
- 융합과학기술대학원 수리정보과학과 학과장
- KAIST
- 이메일: jhcheon@snu.ac.kr

수리정보과학과
Department of Mathematical Information Science

로그인 회원가입 사이트맵

학과소개 입학 학사 교수 및 직원 연구 정보광장



교수 및 직원

Department of Mathematical Information Science (since 2013)

교수진

안정호 교수

Computer Architecture and Systems 전공

- 서울대학교 융합과학기술대학원 지능정보융합학과
- Stanford University
- 이메일: gajh@snu.ac.kr

Me

천정희 교수

정수론, 암호론 전공

- 서울대학교 자연과학대학 수리과학부
- 수학기반산업
- 융합과학기술대학원 수리정보과학과 학과장
- KAIST
- 이메일: jhch

C in the CKKS scheme (HEAAN)

What professor Cheon said (around 2018/2019)

- FHE is slow as it is computationally heavy.
- They have been trying to use FPGA.

1000x faster CKKS will be
a game changer.

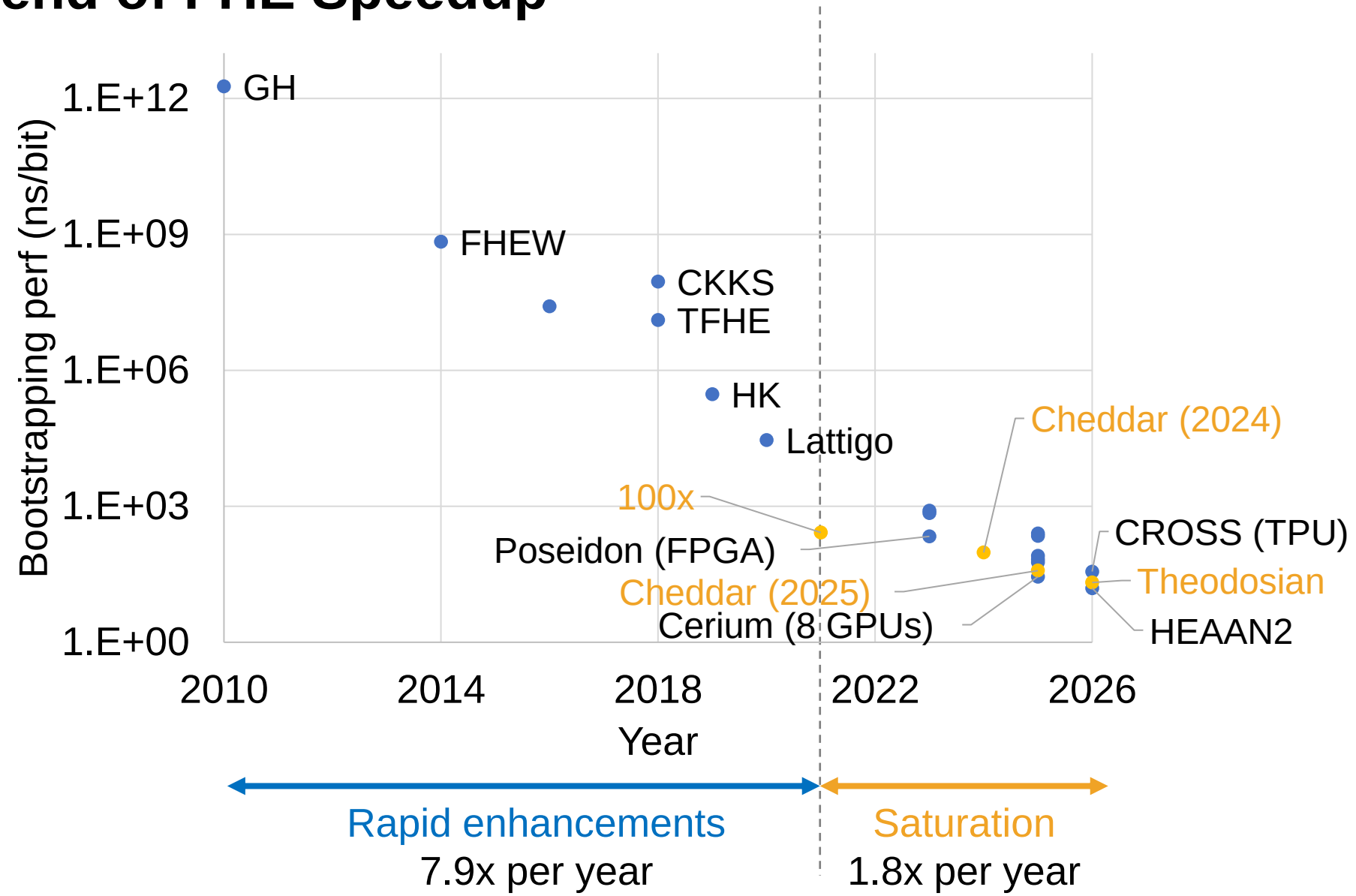
(professor Cheon)

We can do it for you.

(me)

The last is history.

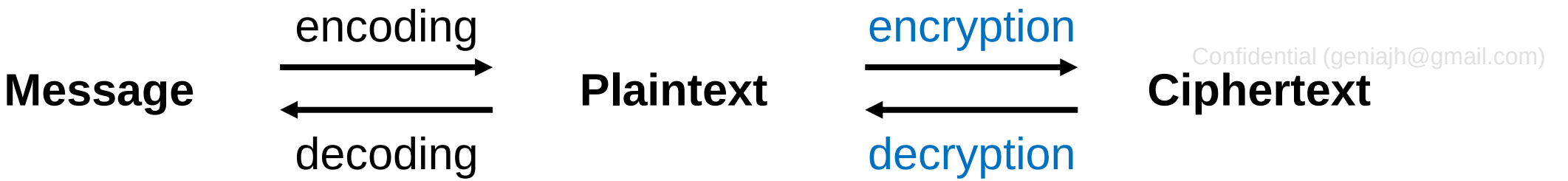
Historic Trend of FHE Speedup



Scheme	Data type in a message	Arithmetic operation
BGV, B/FV	Finite integer $\mathbb{Z}_q$	Integer (mod q)
FHEW, TFHE	Binary $\{0,1\}$	Boolean (XOR, AND)
CKKS	Real / complex $\mathbb{R}$ / $\mathbb{C}$	Approximate (fixed-point)

[CHES 2021]
"Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs," W. Jung, S. Kim, **J. Ahn**, J. H. Cheon, Y. Lee

[CCS 2024]
"NeuJeans: Private Neural Network Inference with Joint Optimization of Convolution and FHE Bootstrapping," J. H. Ju, J. Park, J. Kim, M. Kang, D. Kim, J. H. Cheon, **J. Ahn**



- A **vector**
  - Multiple complex, fixed-point numbers **pack**ed in a plaintext approximately
  - $\mathbf{m} = (\mathrm{m}_0, \mathrm{m}_1, \ldots, \mathrm{m}_{n-1})$
  - $n = N/2$

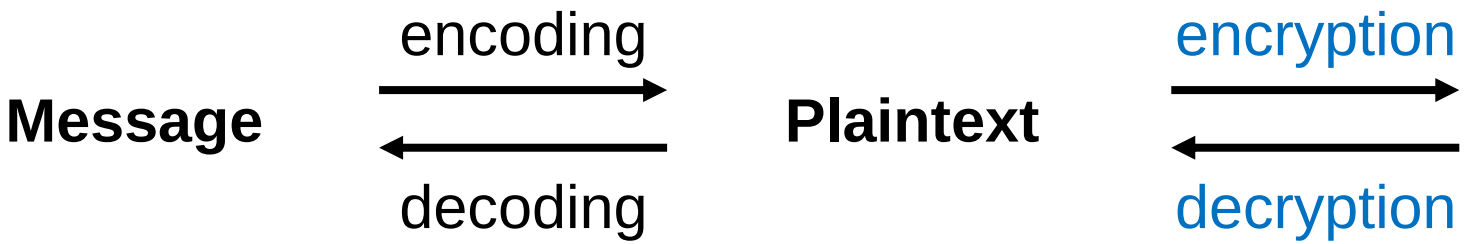
- $C^{N/2} \rightarrow \mathcal{R}_q = Z_q[x]/(x^N + 1)$
  - N: degree of a **polynomial** $\mathcal{R}_q$
  - Mult between $\mathcal{R}_q$ s in a plaintext is equivalent to element-wise mult between vectors in a message
  - $\mathrm{P}_{\mathbf{m}} = a_0 + a_1 x + \cdots + a_{N-1} x^{N-1}$
- Mult complexity
  - $O(N^2)$: without NTT
  - $O(NlogN)$: with NTT

- RLWE-based **HE** scheme
  - Errors in coefficients of $\mathcal{R}_q$
  - Messages and secret keys hidden in a pair of $\mathcal{R}_q$
  - $[\![\mathbf{m}]\!] = (\mathrm{b}_{\mathbf{m}}, \mathrm{a}_{\mathbf{m}})$
    - = $\mathrm{b}_{\mathbf{m}} = \mathrm{a}_{\mathbf{m}} * s + \Delta \cdot \mathrm{P}_{\mathbf{m}} + \mathrm{e}$
    - = $\mathrm{a}_{\mathbf{m}}$: random polynomial
    - = s : secret key
    - = e : error added for security
    - = Δ : scale
  - 100s to 1000s bits per coefficient
  - https://gmplib.org/

- A **vector**
  - Multiple complex, fixed-point numbers **pack**ed in a plaintext approximately
  - $\mathbf{m} = (\mathrm{m}_0, \mathrm{m}_1, \ldots, \mathrm{m}_{n-1})$
  - $n = N/2$

- $C^{N/2} \rightarrow \mathcal{R}_q = Z_q[x]/(x^N + 1)$
  - N: degree of a **polynomial** $\mathcal{R}_q$
  - Mult between $\mathcal{R}_q$ s in a plaintext is equivalent to element-wise mult between vectors in a message
  - $\mathrm{P}_\mathbf{m} = a_0 + a_1 x + \cdots + a_{N-1} x^{N-1}$
- Mult complexity
  - $\mathrm{O}(N^2)$: without NTT
  - $\mathrm{O}(NlogN)$: with NTT

- RLWE-based **HE** scheme
  - Errors in coefficients of $\mathcal{R}_q$
  - Messages and secret keys hidden in a pair of $\mathcal{R}_q$
  - $[\![\mathbf{m}]\!] = (\mathrm{b}_\mathbf{m}, \mathrm{a}_\mathbf{m})$
    - $= \mathrm{b}_\mathbf{m} = \mathrm{a}_\mathbf{m} * s + \Delta \cdot \mathrm{P}_\mathbf{m} + \mathrm{e}$
    - $= \mathrm{a}_\mathbf{m}$: random polynomial
    - $= s$: secret key
    - $= \mathrm{e}$: error added for security
    - $= \Delta$: scale
  - 100s to 1000s bits per coefficient
  - https://gmplib.org/



- A **vector**
  - Multiple complex, fixed-point numbers **pack**ed in a plaintext approximately
  - $\mathbf{m} = (\mathrm{m}_0, \mathrm{m}_1, \dots, \mathrm{m}_{n-1})$
  - $n = N/2$

- $C^{N/2} \rightarrow \mathcal{R}_q = Z_q[x]/(x^N + 1)$
  - N: degree of a **polynomial** $\mathcal{R}_q$
  - Mult between $\mathcal{R}_q$ s in a plaintext is equivalent to element-wise mult between vectors in a message
  - $\mathrm{P}_\mathbf{m} = a_0 + a_1 x + \cdots + a_{N-1} x^{N-1}$
- Mult complexity
  - $O(N^2)$: without NTT
  - $O(NlogN)$: with NTT

- RLWE-based **HE** scheme
  - Errors in coefficients of $\mathcal{R}_q$
  - Messages and secret keys hidden in a pair of $\mathcal{R}_q$
  - $[\![\mathbf{m}]\!] = (\mathrm{b}_\mathbf{m}, \mathrm{a}_\mathbf{m})$
    - = $\mathrm{b}_\mathbf{m} = \mathrm{a}_\mathbf{m} * \mathrm{s} + \Delta \cdot \mathrm{P}_\mathbf{m} + \mathrm{e}$
    - = $\mathrm{a}_\mathbf{m}$: random polynomial
    - = s : secret key
    - = e : error added for security
    - = Δ : scale
  - 100s to 1000s bits per coefficient
  - https://gmplib.org/

$$a \in Z, \Pi = \prod_{i=0}^{\ell-1} q_i \text{ where } q_i\text{s are co-prime}$$

$$a + b \bmod \Pi \mapsto (a_0 + b_0 \bmod q_0, a_1 + b_1 \bmod q_1, \cdots, a_{\ell-1} + b_{\ell-1} \bmod q_{\ell-1})$$

$$a \cdot b \bmod \Pi \mapsto (a_0 \cdot b_0 \bmod q_0, a_1 \cdot b_1 \bmod q_1, \cdots, a_{\ell-1} \cdot b_{\ell-1} \bmod q_{\ell-1})$$

	(mod 7)	(mod 11)	(mod 13)
15	1	4	2
8	1	8	8
15+8 = 23	2	1	10
15*8 = 120	1	10	3

- 100s to 1000s bits per coefficient

$$\text{mod} \quad (\quad \text{mod} \quad \text{mod} \quad \text{mod} \quad)$$
$$\text{mod} \quad (\quad \text{mod} \quad \text{mod} \quad \text{mod} \quad)$$

mod (%) reduction method	Computation Requirement	# of const	Output range
Barret	*mulhi64* + *mullo64* + *add64*	1	$[0, 2q_i)$
Montgomery	*mulwide32* + *mullo32* + *add64*	1	$[0, 2q_i)$
Shoup	*2 * mullo32* + *add32*	Many	$[0, 2q_i)$
Signed Montgomery	*mulhi32* + *mullo32* + *add32*	1	$[-q_i, q_i)$

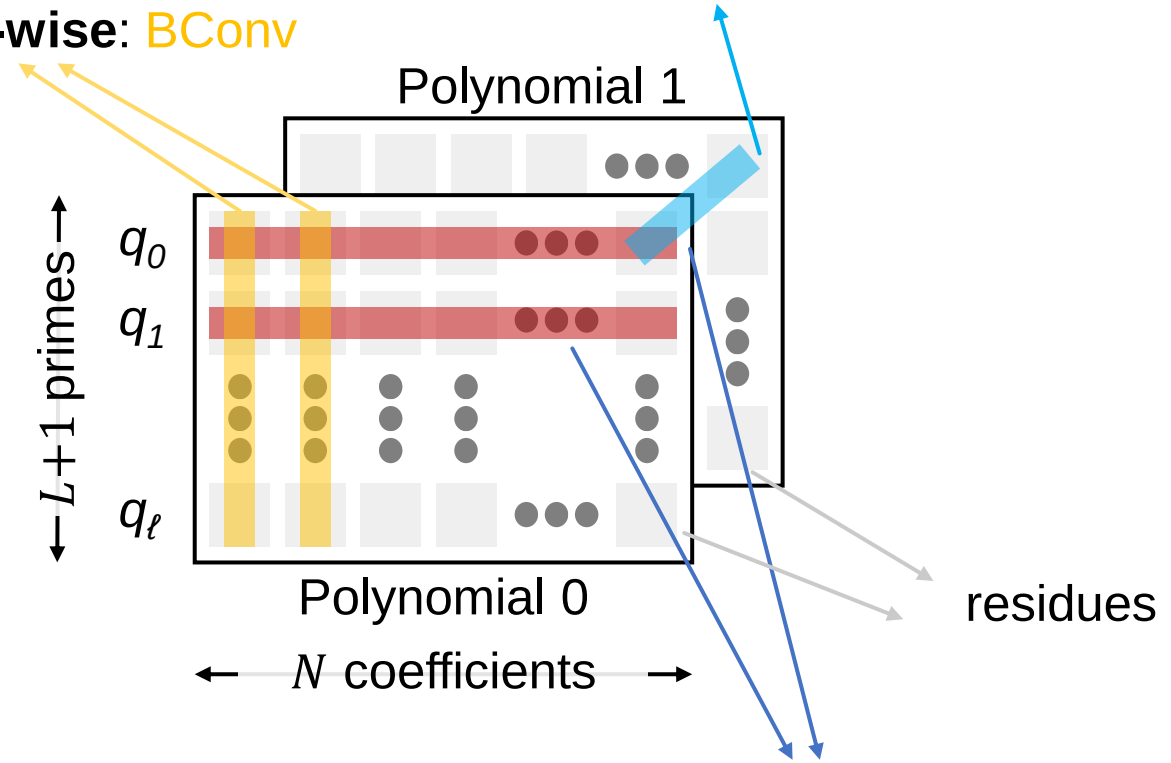
[ASPLOS 2026]
"Cheddar: A Swift Fully Homomorphic Encryption Library Designed for GPU Architectures,"
W. Choi, J. Kim, **J. Ahn**

element-wise: element-wise mult / add

coefficient-wise: BConv

an $(L+1) \times N$ **matrix** of small integers

$$\begin{pmatrix} a_0 \ \% \ q_0 & \cdots & a_{N-1} \ \% \ q_0 \\ \vdots & \ddots & \vdots \\ a_0 \ \% \ q_L & \cdots & a_{N-1} \ \% \ q_L \end{pmatrix} \begin{matrix} \rightarrow q_0\text{-limb, } [\mathrm{P_m}]_{q_0} \\ \vdots \\ \rightarrow q_L\text{-limb , } [\mathrm{P_m}]_{q_L} \end{matrix}$$

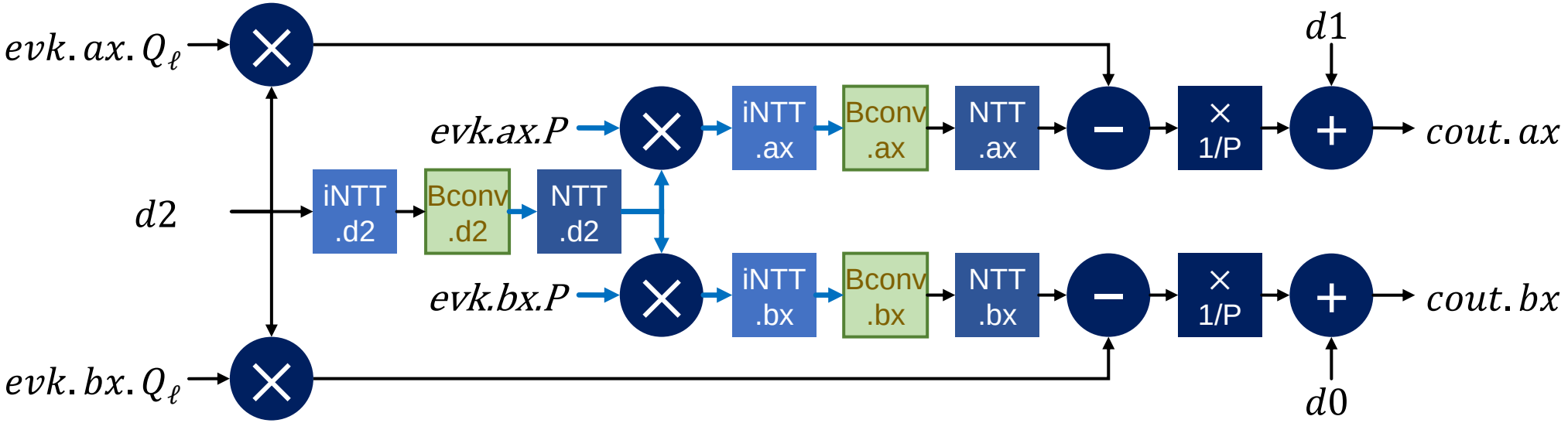


limb-wise: (I)NTT, automorphism

- Evaluation representation
- Coefficient representation

$$d0 = (cin0.bx \otimes cin1.bx)$$
$$d1 = (cin0.ax \otimes cin1.bx) \oplus (cin0.bx \otimes cin1.ax)$$
$$d2 = (cin0.ax \otimes cin1.ax)$$

HMult (homomorphic multiplication)

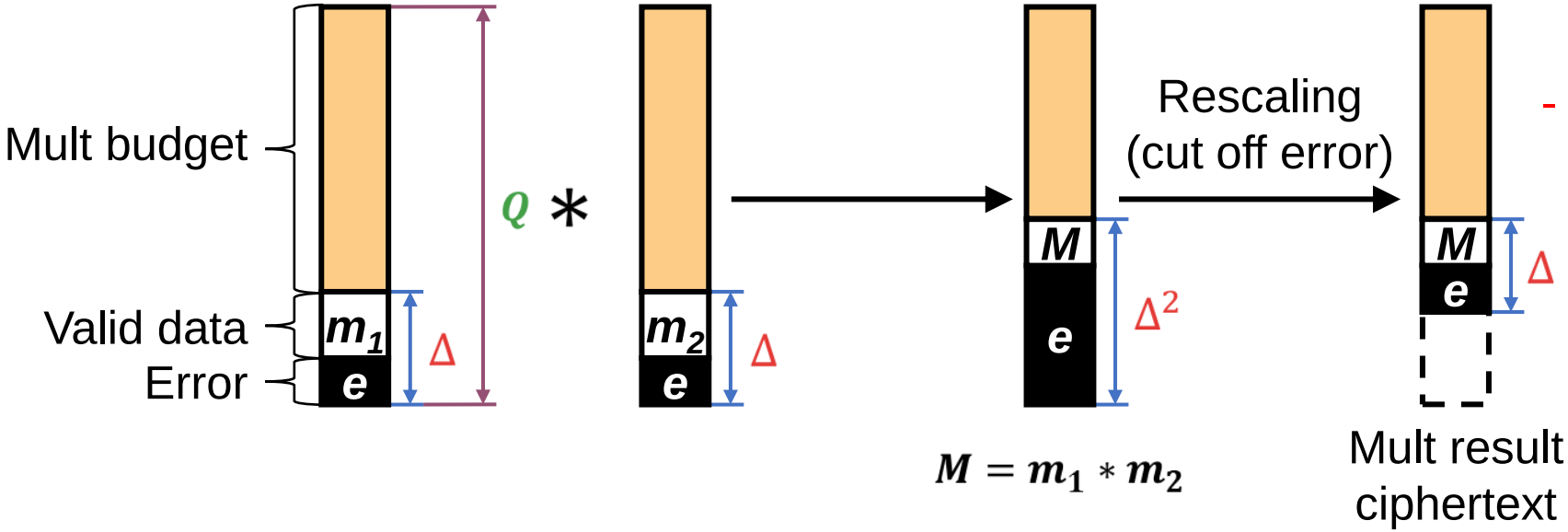


→ A matrix sized $(N, \ell + 1)$

→ A matrix sized (N, k)

Fewer limbs
(consuming **a level**)

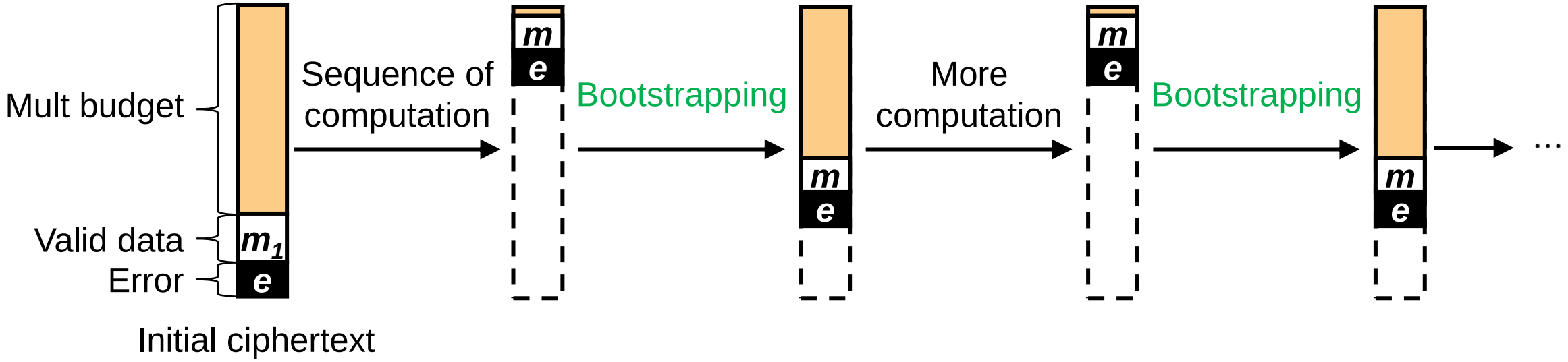
Homomorphic
Mult (**HMult**)



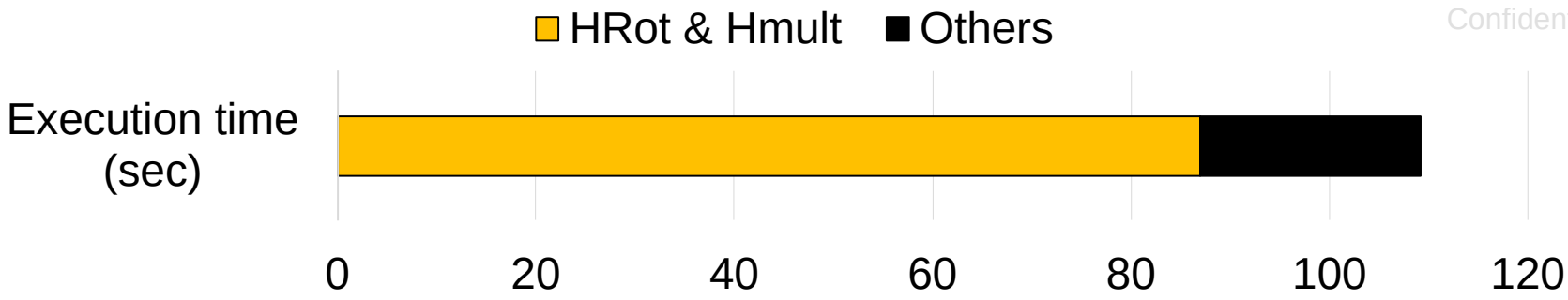
- $[\![\mathbf{m}]\!] = (\mathrm{b_m}, \mathrm{a_m})$
  - = $\mathrm{b_m} = \mathrm{a_m} * \mathrm{s} + \Delta \cdot \mathrm{P_m} + \mathrm{e}$
  - = $\mathrm{a_m}$: random polynomial
  - = s : secret key
  - = e : error added for security
  - = Δ : scale

a level can't be negative

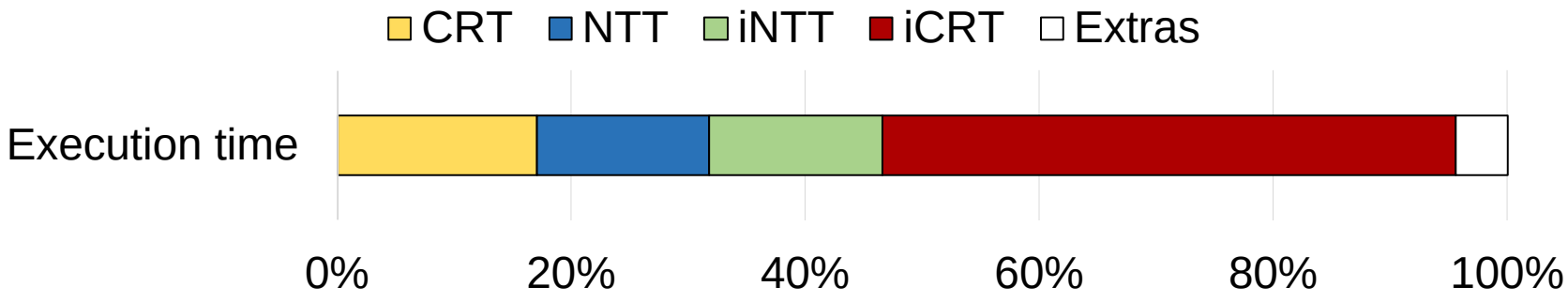
- Leveled **HE** (LHE): when an application requires a limited number of levels
- Fully **HE** (**FHE**): restore the level of a ciphertext through an expensive process of bootstrapping



A brief (non-exhaustive) recap on our research regarding FHE acceleration



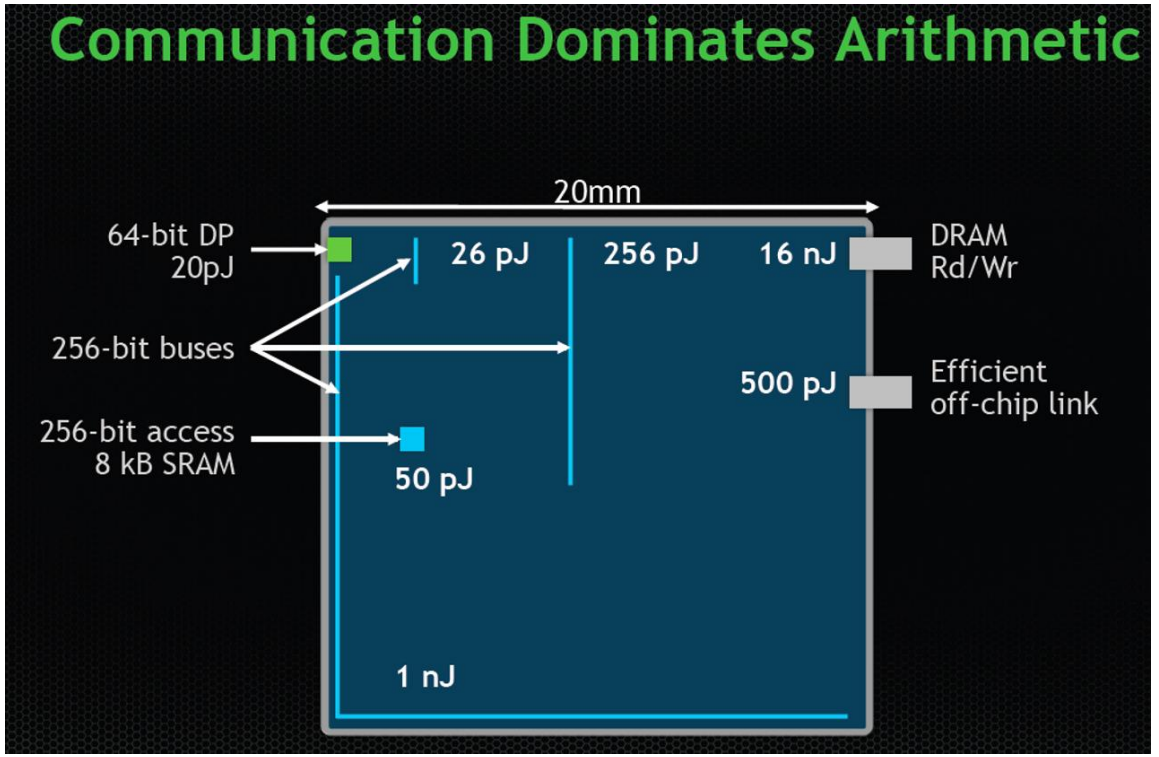
Time-breakdown of one iteration in training a binary logistic regression model using binary CKKS



HMult time breakdown (total 3,907 ms)

NTT & CRT are dominant for time consuming FHE operations.

- An ample chance of exploiting DLP
- Large datasets, various chances of data reuse at different scales in time & space
- Computation cheap, communication expensive.



"A keynote speech from Bill Dally," **HiPEAC** 2015

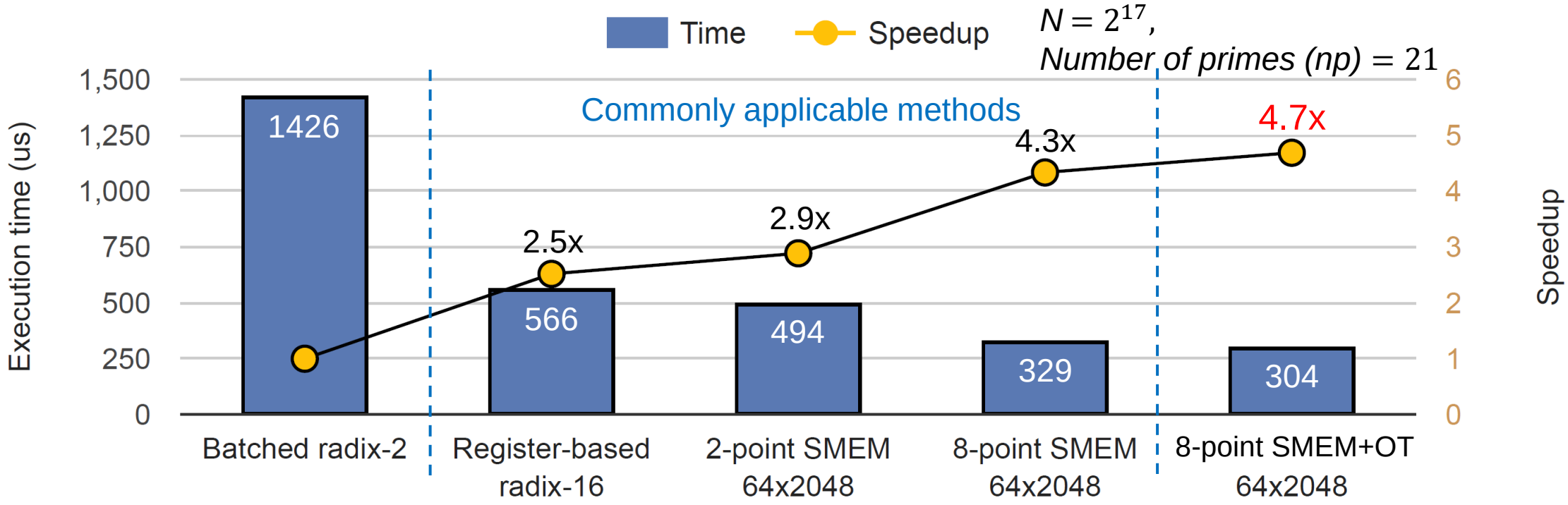
GPU can be a nice acceleration platform for CKKS (HE in general).

First, NTT (Number Theoretic Transform)

- A variant of DFT of which the data type is (big) integer, not float/double.
- FFT is a classic problem.
  - Hard to find a new break-through (e.g., FFT in HPC & FFTW)

[IISWC 2020]
"Accelerating Number Theoretic Transformations for Bootstrappable Homomorphic Encryption on GPUs," S. Kim, W. Jung, J. Park, J. Ahn

- Textbook optimizations
  - High-radix
  - Exploit precomputation (only) when beneficial (on-the-fly twiddling)
  - Reuse registers and shared memory as much as possible
  - Pipeline matrix transposition and local butterfly operations

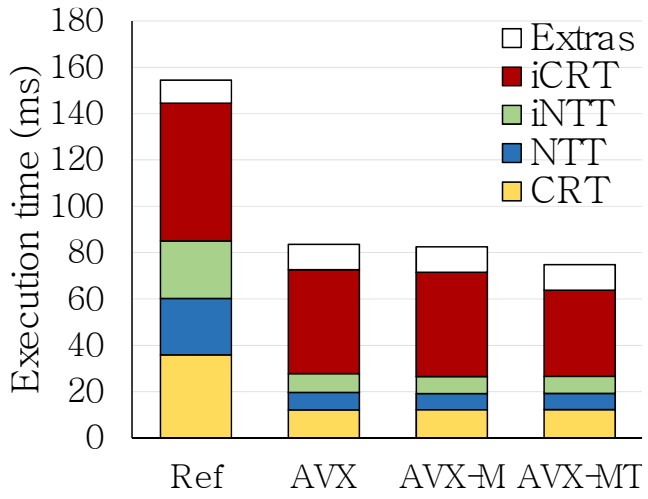


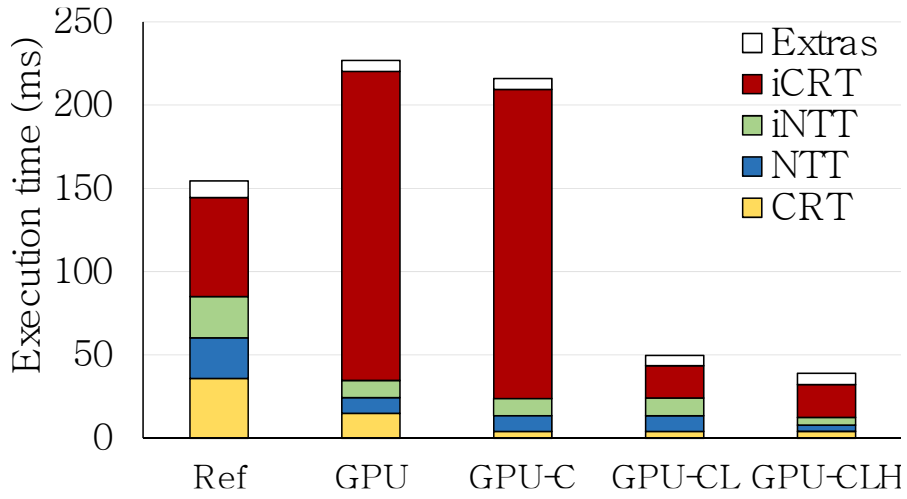
Then, analyzing and optimizing HMult

- Targeting binary (non-RNS) CKKS
  - NTT and CRT (Chinese Remainder Theorem) are dominant.
  - This time, optimizing for CPU, GPU, or both.
  - Tried usual suspects
  - 2.06x speedup for CPU, 4.05x speedup for GPU over the 24-thread baseline CPU.

AVX: vectorization with AVX-512
AVX-M: AVX + efficient emulation
AVX-MT: AVX-M + implicit transpose
GPU: Turing Titan RTX
GPU-C: GPU + ADC for CRT
GPU-CL: GPU-C + loop reordering for iCRT
GPU-CLH: GPU-CL + high-radix NTT/iNTT

Parameters: $Q = 2^{1200}, N = 216, p = 2^{40}$





[IEEE Access 2021]
"Accelerating Fully Homomorphic Encryption Through Architecture-Centric Analysis and Optimization,"
W. Jung, E. Lee, S. Kim, J. Kim, N. Kim, K. Lee, J. Min, J. Cheon, J. Ahn

- A bit of frustration
  - Went through **7** rounds of the review process
  - ∵ Novelty game
  - Just single HE op (Hmult), no application results

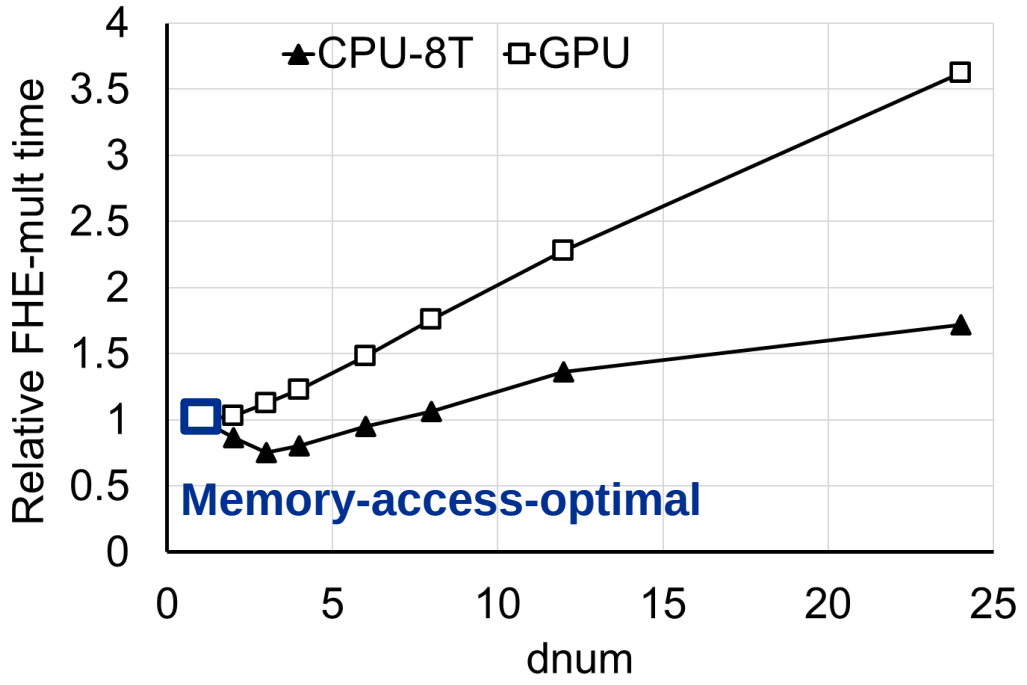
Sometimes, you should declare success and move on.

Move on to the next level

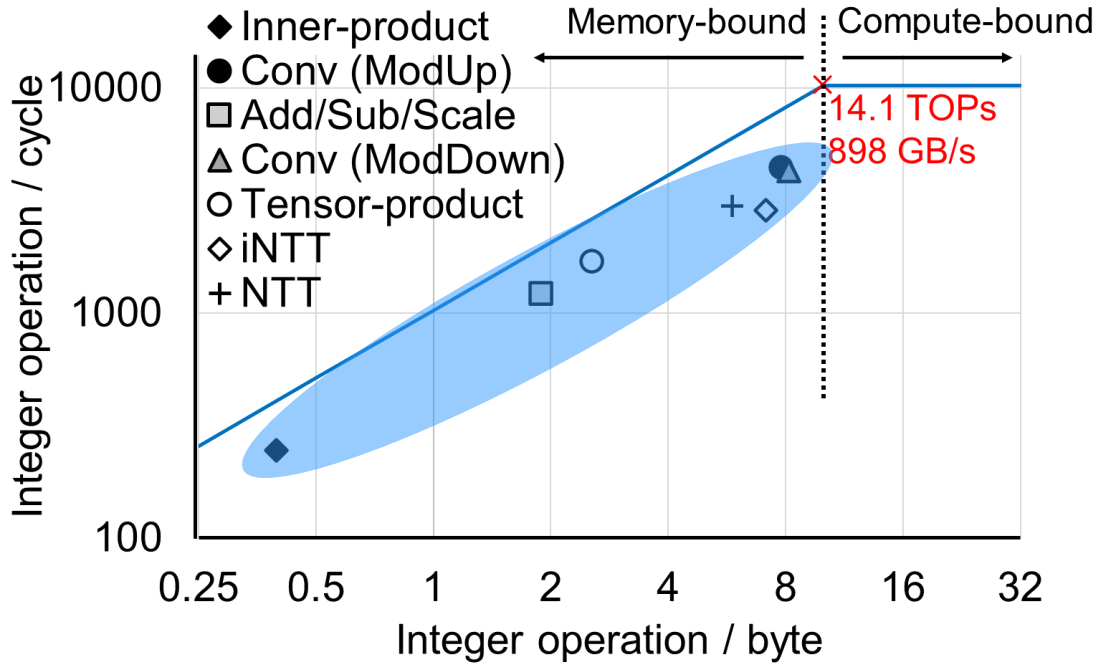
- Focus on GPUs
- Implement the bootstrapping operation on GPUs
- Logistic regression: an ML application
- Not binary CKKS, but full-RNS CKKS

Key observations and insight

- Roofline model (memory-bound)
- Reducing memory access is critical.



dnum vs. multiplication time for a fixed Q

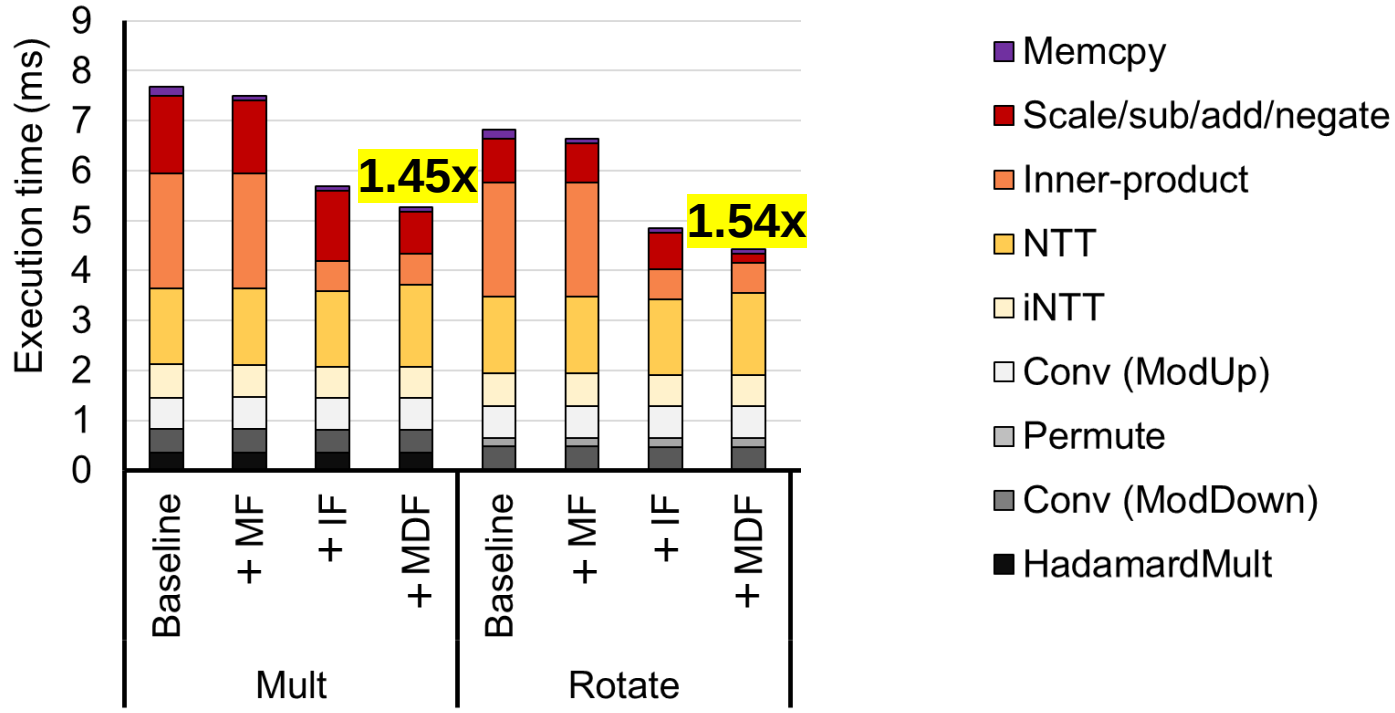


Roofline plot of an HMult

Fusing kernels is effective on GPUs.

- Latency of HMult and HRot after applying various kernel fusion techniques
- **7.02x** faster HMult over a prior GPU implementation [BHM+20] which uses **dnum** = L

Latency with **dnum** = L (maximum)



Latency with **dnum** = 3 [HK20]

$N = 2^{16}$, $\log Q = 2305$, $L = 44$. Used an NVIDIA V100.

[CHES 2021]

"Over **100x** Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs," W. Jung, S. Kim, J. Ahn, J. Cheon, Y. Lee

- Bootstrapping latency

	Latency (ms)				
Parameter set (N, L, dnum, λ)	Baseline	Intra-HE fusion	Mult-and-add Batching	Hoisting	**Speedup (vs. single-thread CPU)**
$(2^{16}, 34, 5, 106)$	428.94	377.78	351.09	328.25	**242x**
$(2^{17}, 29, 3, 173)$	719.87	623.92	568.2	526.96	**257x**

- End-to-end evaluation: training a binary classification (logistic regression) model
  - Compared to 8-threaded CPU case, GPU implementation exhibits a speedup of **40x** in total.

- We suggest a new metric for performance considering bootstrapping:

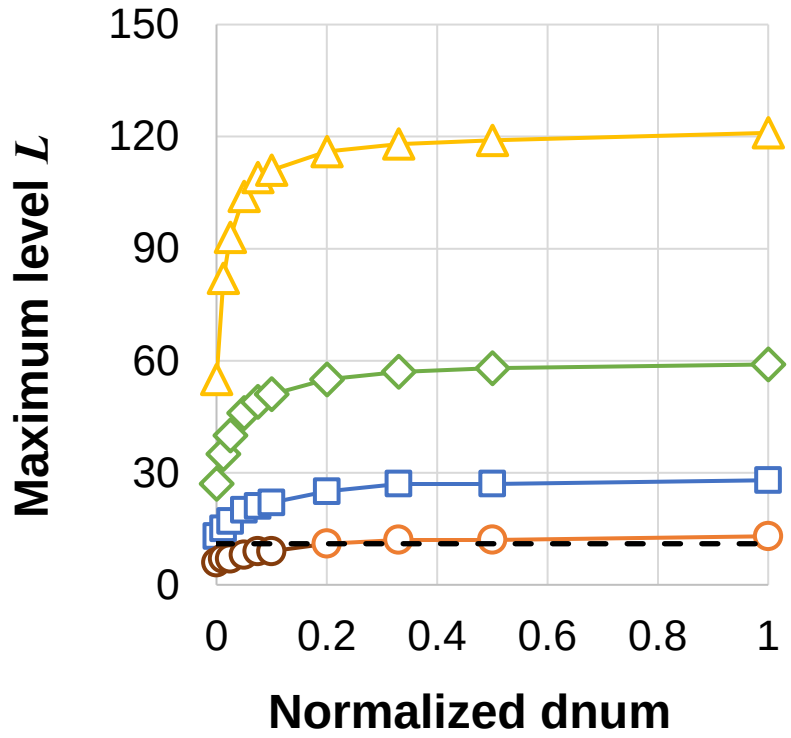
$$Amotized\ mult\ time = \frac{Bootstrapping\ time + multiplication\ times}{\#\ of\ multiplications\ after\ bootstrapping}$$

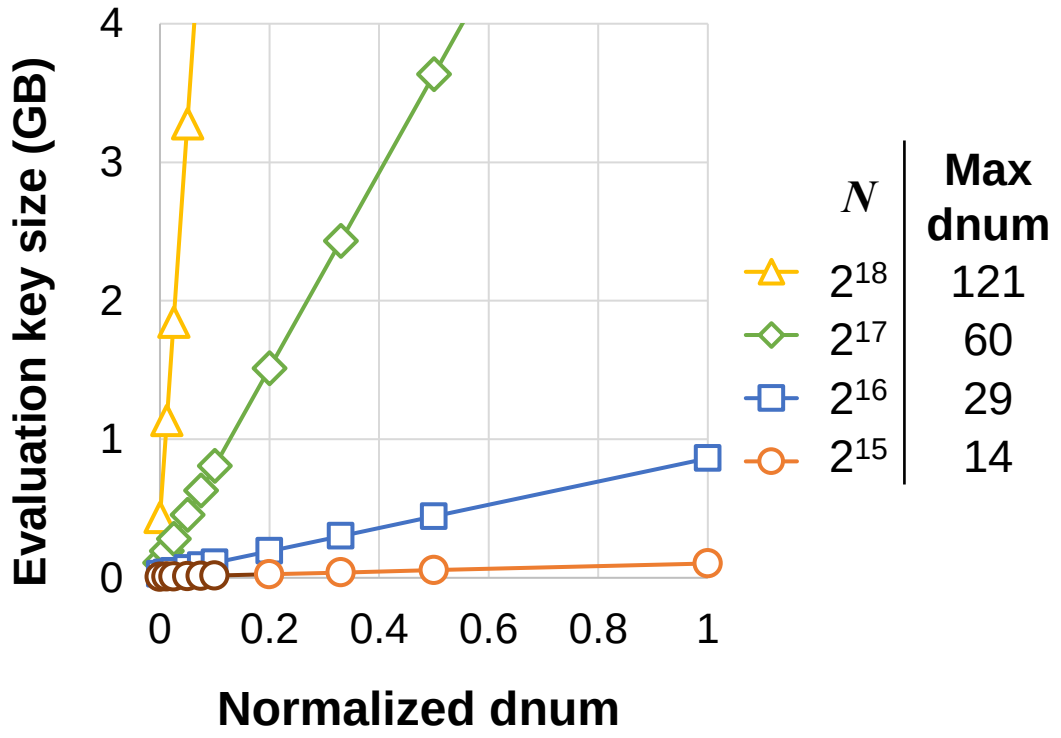
New horizons – 1)

- Can you make it better?
  - Yes! With domain specific architecture (DSA, a.k.a., ASIC)
- Where are the new opportunities? Key limitations of GPUs
  - Massive floating-point & low-precision units
  - (Relatively) small on-chip storage
  - Limited shuffling bandwidth between SMs (streaming multiprocessors)
  - …

The ones that cannot be seen before you climb a hill.

BTS: Bootstrapping-oriented FHE accelerator design



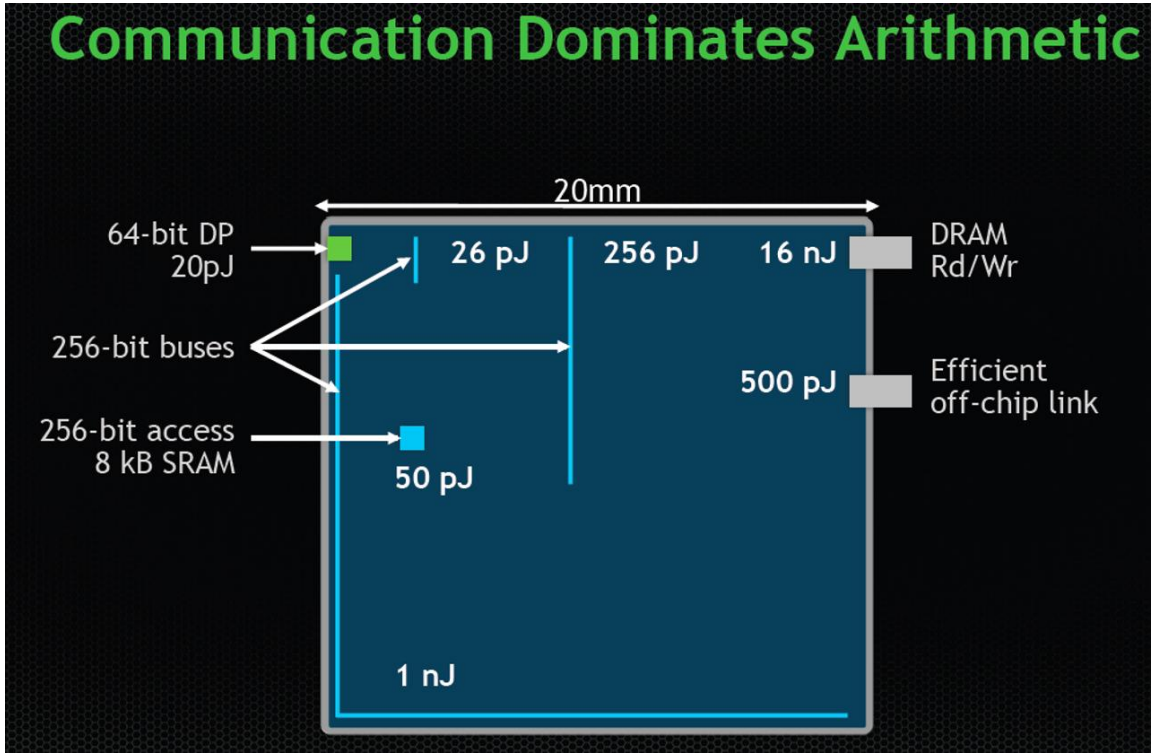


Evaluation key (evk): the last man standing

BTS: Technology-driven accelerator design

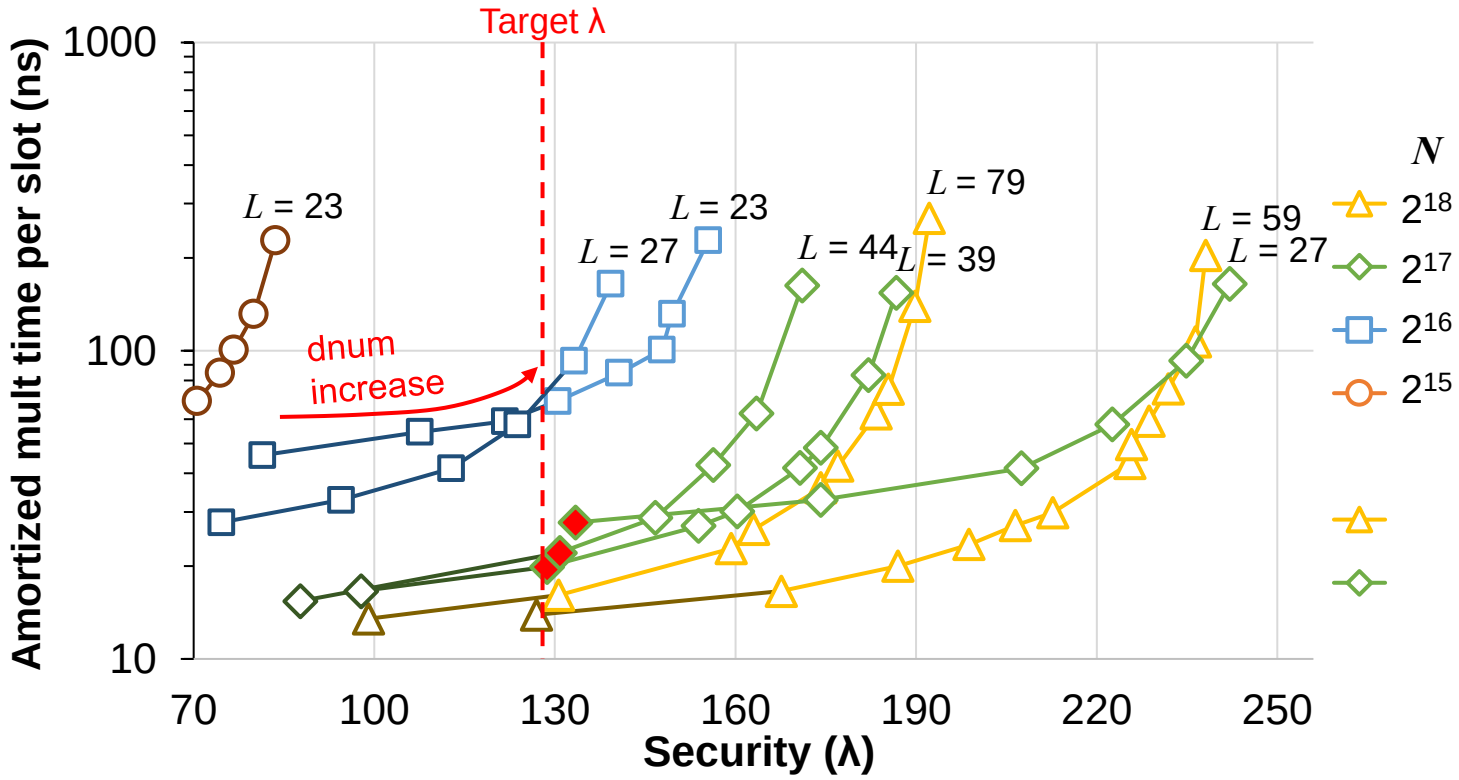
- Bootstrapping requires GBs of evk data, which is not easy to reuse.

- A bold assumption:
  - Let's assume that evk is in memory and design an accelerator that is just good enough to conduct an HE operation while an evk is being loaded, through overlapping computation and communication.
  - Yes, a recurring theme



"A keynote speech from Bill Dally," **HiPEAC** 2015

BTS: Security level aware design

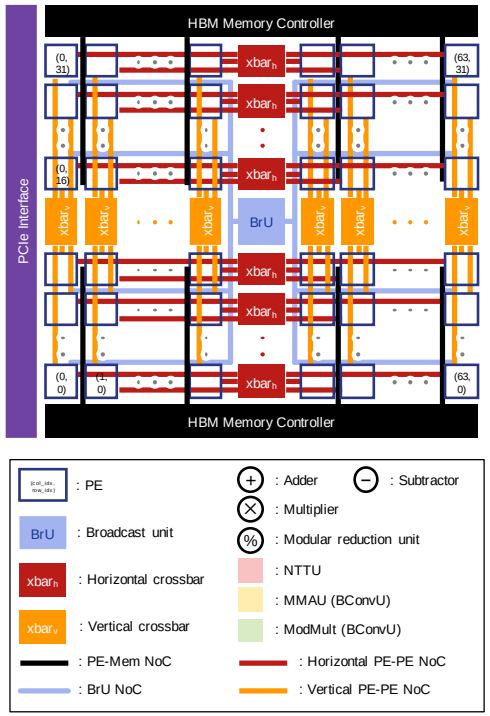


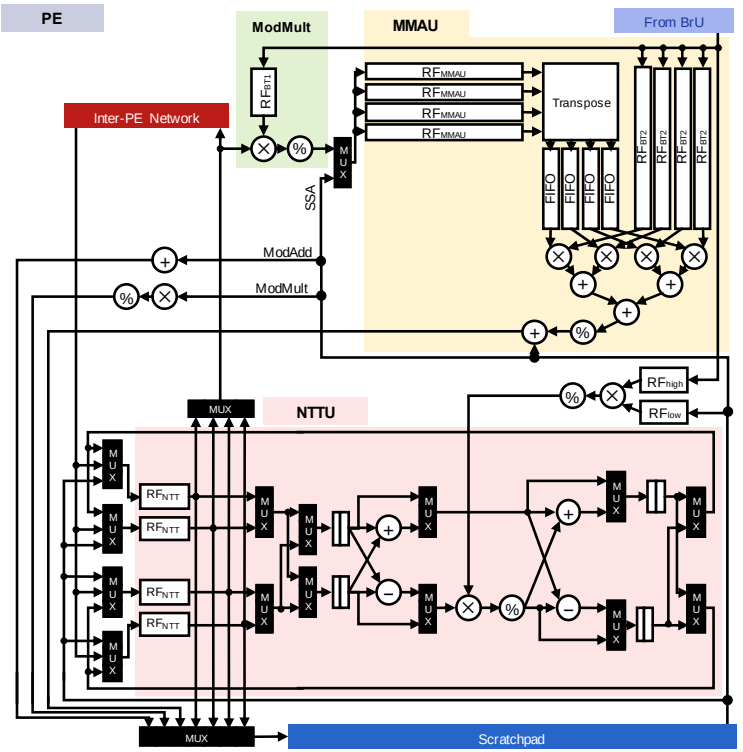
	N	L	$dnum$	$logPQ$	λ
Parameter set 1 (BTS-1)	2^{17}	27	1	3090	133.4
Parameter set 2 (BTS-2)	2^{17}	39	2	3210	128.7
Parameter set 3 (BTS-3)	2^{17}	44	3	3160	130.8

[ISCA 2022]

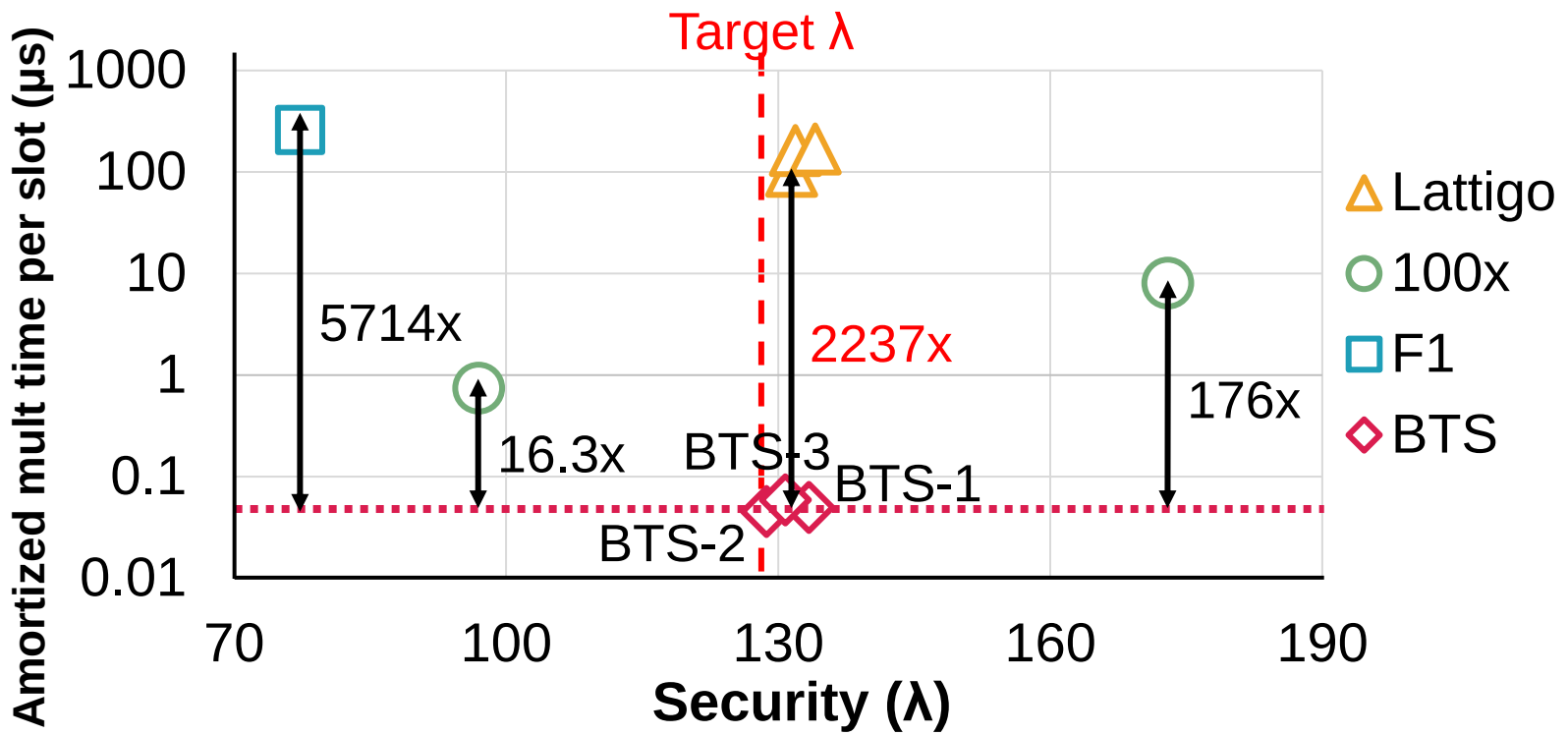
"BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption,"
S. Kim, J. Kim, M. J. Kim, W. Jung, M. Rhu, J. Kim, J. Ahn

- The philosophy of the accelerator design
  - Exploit the massive parallelism
  - Thousand of processing elements
  - One (prime) at a time
  - Minimize global communication, NoC: a few simple hardware, software scheduling





[ISCA 2022]
"BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption,"
S. Kim, J. Kim, M. J. Kim, W. Jung, M. Rhu, J. Kim, J. Ahn



Not yet done. Another room for improvement exists.

FHE accelerator trilogy

[ISCA 2022]
"BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption,"
S. Kim, J. Kim, M. J. Kim, W. Jung, M. Rhu, J. Kim, J. Ahn

[MICRO 2022]
"ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse," J. Kim, G. Lee, S. Kim, G. Sohn, M. Rhu, J. Kim, J. Ahn

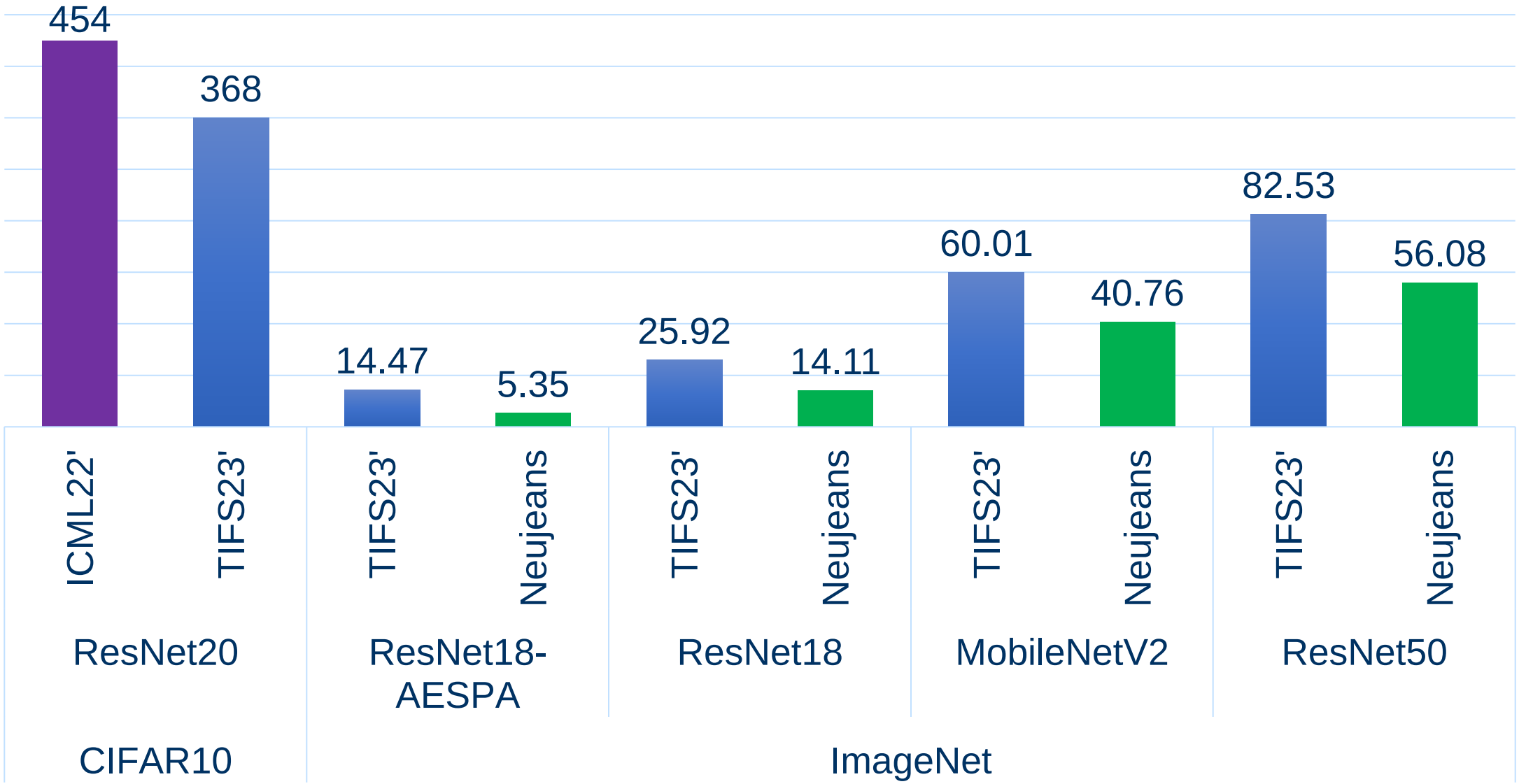
[ISCA 2023]
"SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption," J. Kim, S. Kim, J. Choi, J. Park, D. Kim, J. Ahn

New horizons – 2)

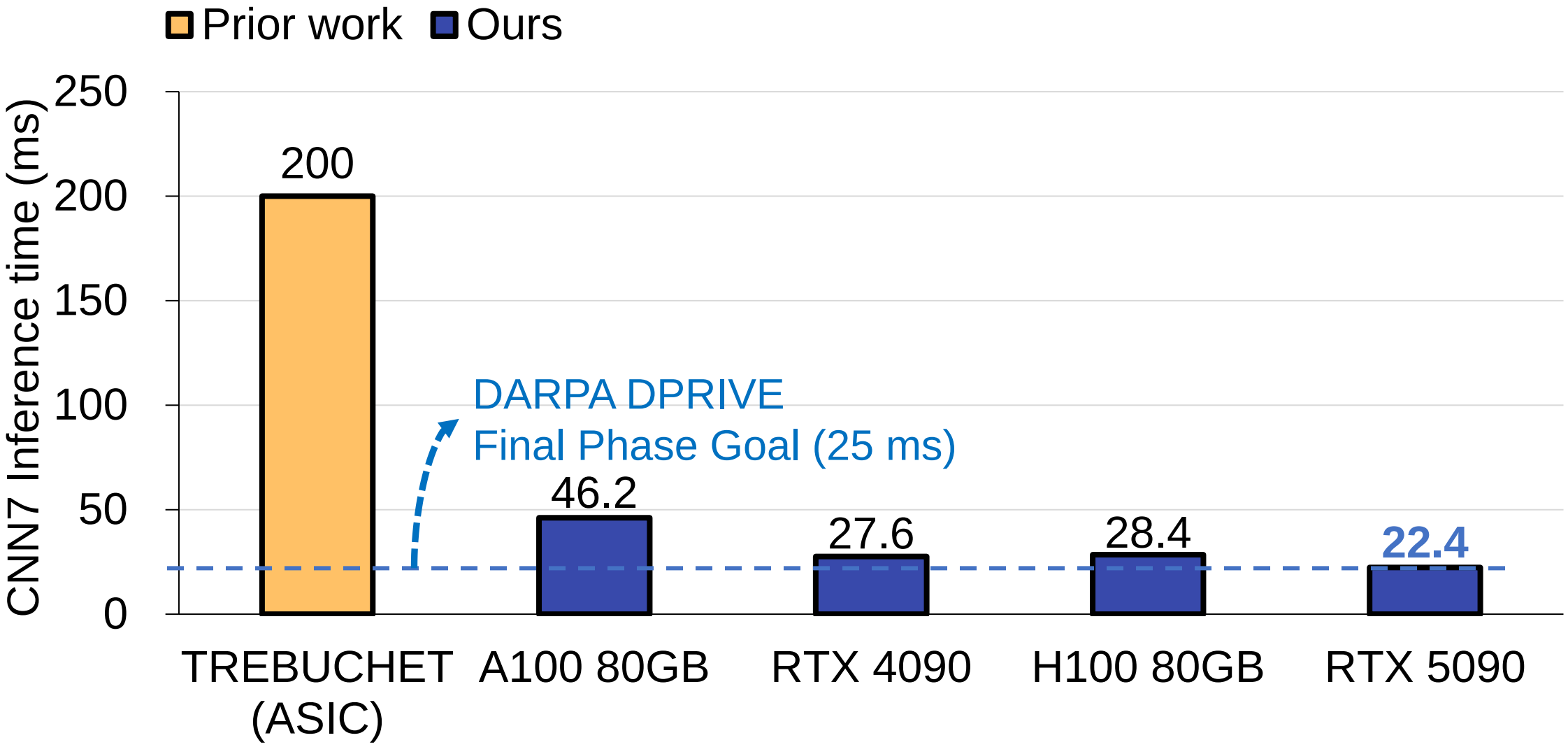
- Can you make it better?
  - Yes! By modifying CKKS applications through deep understanding of architecture-algorithm interaction

The ones that cannot be seen before you climb a hill.

End-to-End Inference of CNN models (unit: second)



CNN7 inference time in milliseconds



D. B. Cousins et al., “TREBUCHET Fully Homomorphic Encryption Accelerator: Phase Two Performance Estimation Results,” in GOMACTech, 2024.

New horizons – 3)

- Can you make it better?
  - Yes! Optimized GPU libraries (Cheddar and Theodosian).
  - The topic of the last of today's tutorial.

Enjoy!

1) Wonseok Choi, Hyunah Yu, Jongmin Kim, Hyesung Ji, Jaiyoung Park, Jung Ho Ahn: "Theodosian: A Deep Dive into Memory-Hierarchy-Centric FHE Acceleration," ISPASS 2026

2) Wonseok Choi, Jongmin Kim, Jung Ho Ahn: "*Cheddar: A Swift Fully Homomorphic Encryption Library Designed for GPU Architectures*," ASPLOS 2026

3) Jaiyoung Park, Sangpyo Kim, Jongmin Kim, Jung Ho Ahn: "*Unlocking Private Computation at Scale: The Acceleration of Homomorphic Encryption*," IEEE Computer Magazine 2025

4) Sangpyo Kim, Hyesung Ji, Jongmin Kim, Wonseok Choi, Jaiyoung Park, Jung Ho Ahn: "*IVE: An Accelerator for Single-Server Private Information Retrieval Using Versatile Processing Elements*," HPCA 2026

5) Jongmin Kim, Sungmin Yun, Hyesung Ji, Wonseok Choi, Sangpyo Kim, Jung Ho Ahn: "*Anaheim: Architecture and Algorithms for Processing Fully Homomorphic Encryption in Memory*," HPCA 2025

6) Jae Hyung Ju, Jaiyoung Park, Jongmin Kim, Minsik Kang, Donghwan Kim, Jung Hee Cheon, Jung Ho Ahn: "*NeuJeans: Private Neural Network Inference with Joint Optimization of Convolution and FHE Bootstrapping*," ACM CCS 2024

7) Sangpyo Kim, Jongmin Kim, Jaeyoung Choi, Jung Ho Ahn: "*CiFHER: A Chiplet-Based FHE Accelerator with a Resizable Structure*," IEEE SEED 2024

8) Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, Jung Ho Ahn: "*SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption*," ISCA 2023

9) Rashmi Agrawal, Jung Ho Ahn, Flavio Bergamaschi, Ro Cammarota, Jung Hee Cheon, Fillipe D. M. de Souza, Huijing Gong, Minsik Kang, Duhyeong Kim, Jongmin Kim, Hubert de Lassus, Jai Hyun Park, Michael Steiner, Wen Wang: "*High-precision RNS-CKKS on fixed but smaller word-size architectures: theory and application*," WAHC 2023

10) Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, Jung Ho Ahn: "*ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse*," MICRO 2022

11) Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, Jung Ho Ahn: "*BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption*," ISCA 2022

12) Sangpyo Kim, Wonkyung Jung, Jaiyoung Park, Jung Ho Ahn: "*Accelerating Number Theoretic Transformations for Bootstrappable Homomorphic Encryption on GPUs*," IISWC 2020

13) Donghwan Kim, Jaiyoung Park, Jongmin Kim, Sangpyo Kim, Jung Ho Ahn: "*HyPHEN: A Hybrid Packing Method and Its Optimizations for Homomorphic Encryption-Based Neural Networks*," IEEE Access 2024

14) Wonkyung Jung, Eojin Lee, Sangpyo Kim, Jongmin Kim, Namhoon Kim, Keewoo Lee, Chohong Min, Jung Hee Cheon, Jung Ho Ahn: "*Accelerating Fully Homomorphic Encryption Through Architecture-Centric Analysis and Optimization*," IEEE Access 2021

15) Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, Younho Lee: "*Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs*," IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES) 2021

Questions?

Implementation (Hardware, Year)	Boot (ms)
100x (V100, ‘21)	328
TensorFHE (A100 40GB, ‘23)	250
HEaaN GPU (A100 80GB, ‘23)	171
WarpDrive (A100 80GB, ‘25)	121
FIDESlib (RTX 4090, ‘25)	146
Cerium (DGX A100, ‘25)	34.2
Cerium (DGX H100, ‘25)	20.0
Cerium (DGX B200, ‘25)	14.5
Cheddar (A100 80GB)	40.0
Cheddar (H100 80GB)	31.2
Cheddar (RTX 5090)	22.1
Theodosian (A100 80GB)	31.8
Theodosian (H100 80GB)	23.6
Theodosian (RTX 5090)	15.2

Implementation (Hardware, Year)	Boot (ms)
FAB (FPGA, ‘23)	477
Poseidon (FPGA, ’23)	128
EFFACT (FPGA, ‘25)	148
Cross (TPUv4-8, ‘26)	129.8
Cross (TPUv5e-4, ‘26)	59.2
Cross (TPUv5p-8, ’26)	68.3
Cross (TPUv6e-8, ‘26)	21.5

RTX4090

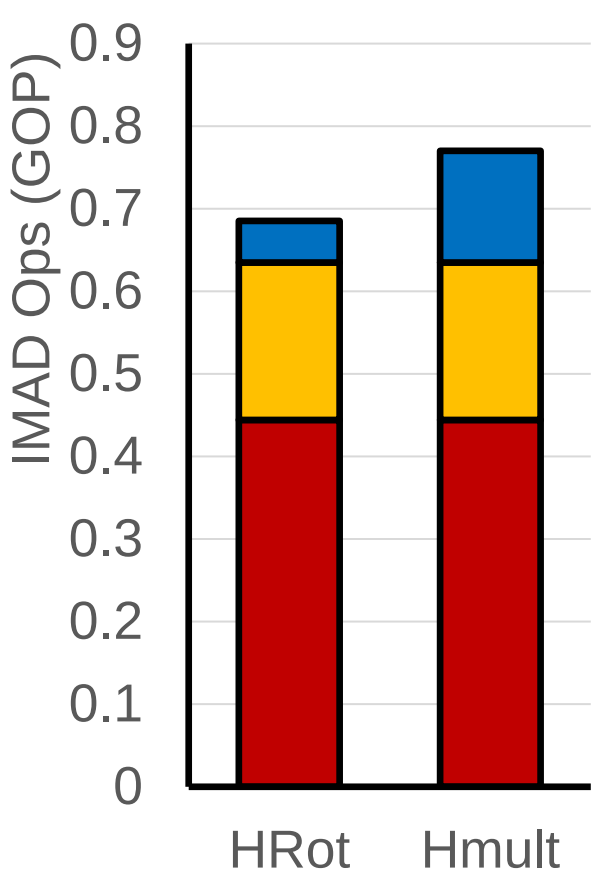
Library	limbs	α	HMult	HRot	HAdd	Rescale
Liberate.FHE	24	6	5989 μs	4554 μs	96 μs	150 μs
HEonGPU	24	6	2427 μs	2279 μs	79 μs	255 μs
Phantom	24	6	958 μs	850 μs	83 μs	114 μs
Cheddar	48	12	533 μs	476 μs	48 μs	68 μs
Liberate.FHE	12	6	2848 μs	2052 μs	22 μs	120 μs
HEonGPU	12	6	825 μs	740 μs	10 μs	113 μs
Phantom	12	6	380 μs	336 μs	18 μs	68 μs
Cheddar	24	12	222 μs	191 μs	11 μs	47 μs

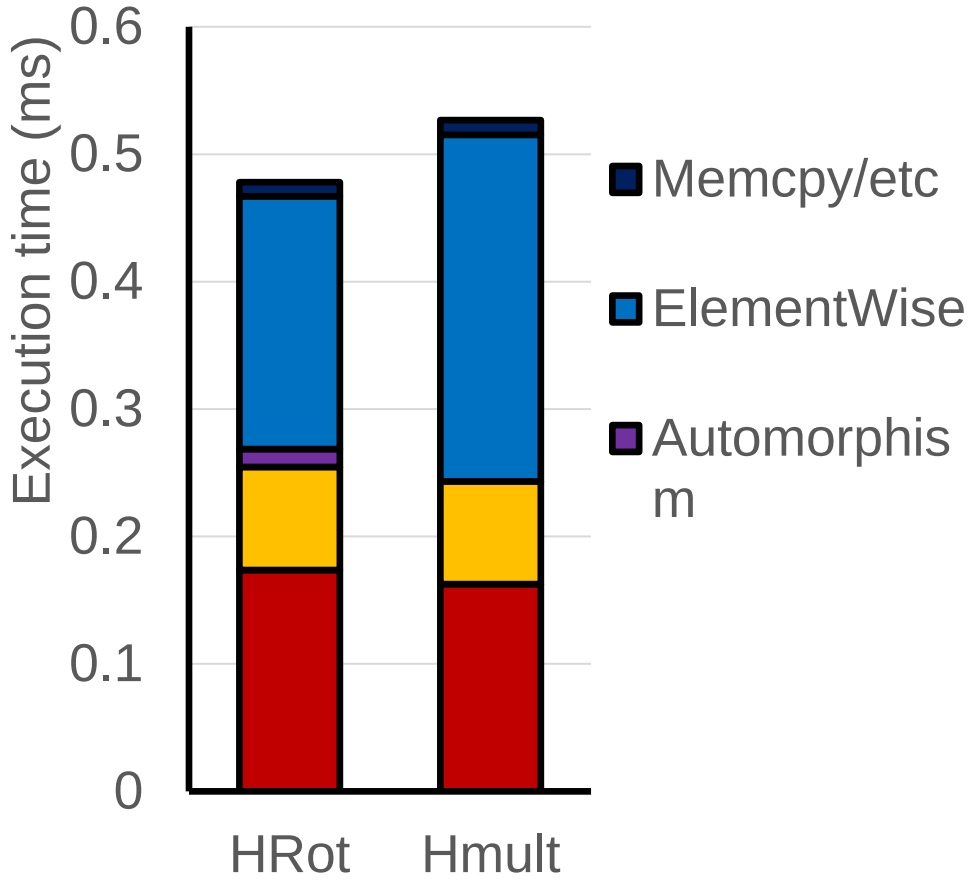
[arxiv 2024, ASPLOS 2026]

“Cheddar: A Swift Fully Homomorphic Encryption Library Designed for GPU Architectures,”

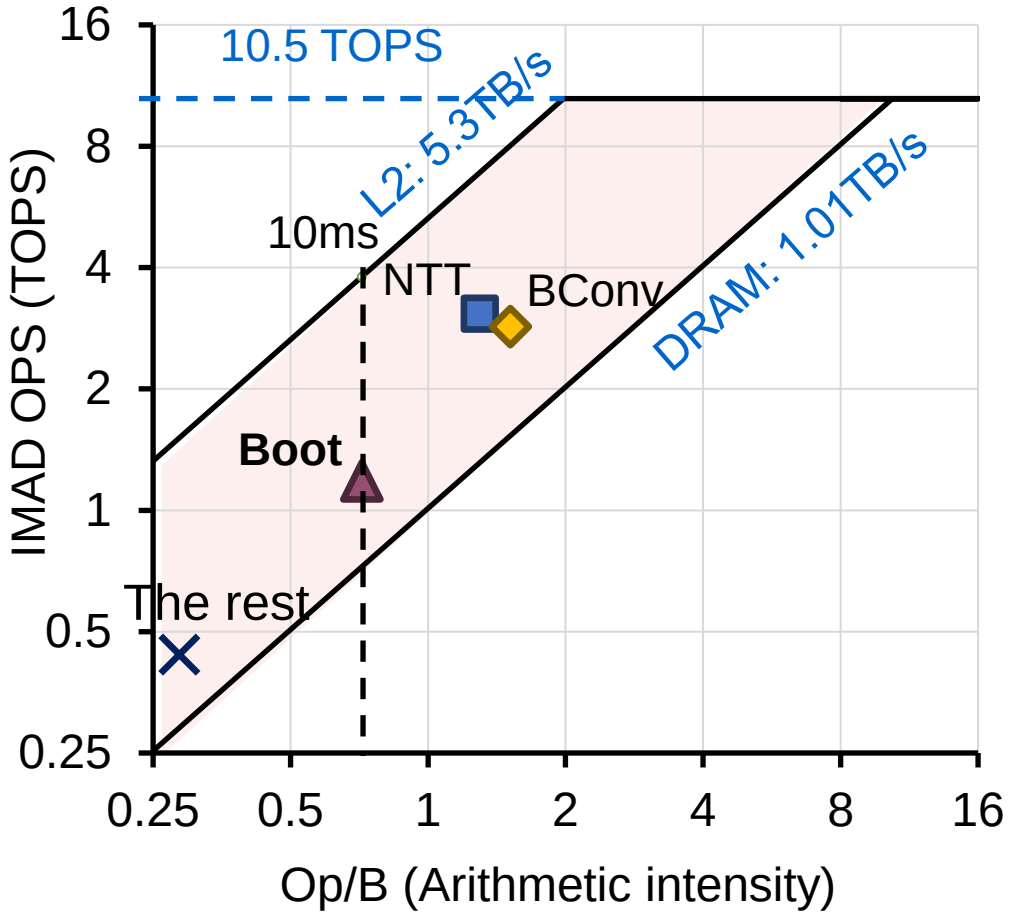
W. Choi, J. Kim, **J. Ahn**

RTX4090





RTX4090

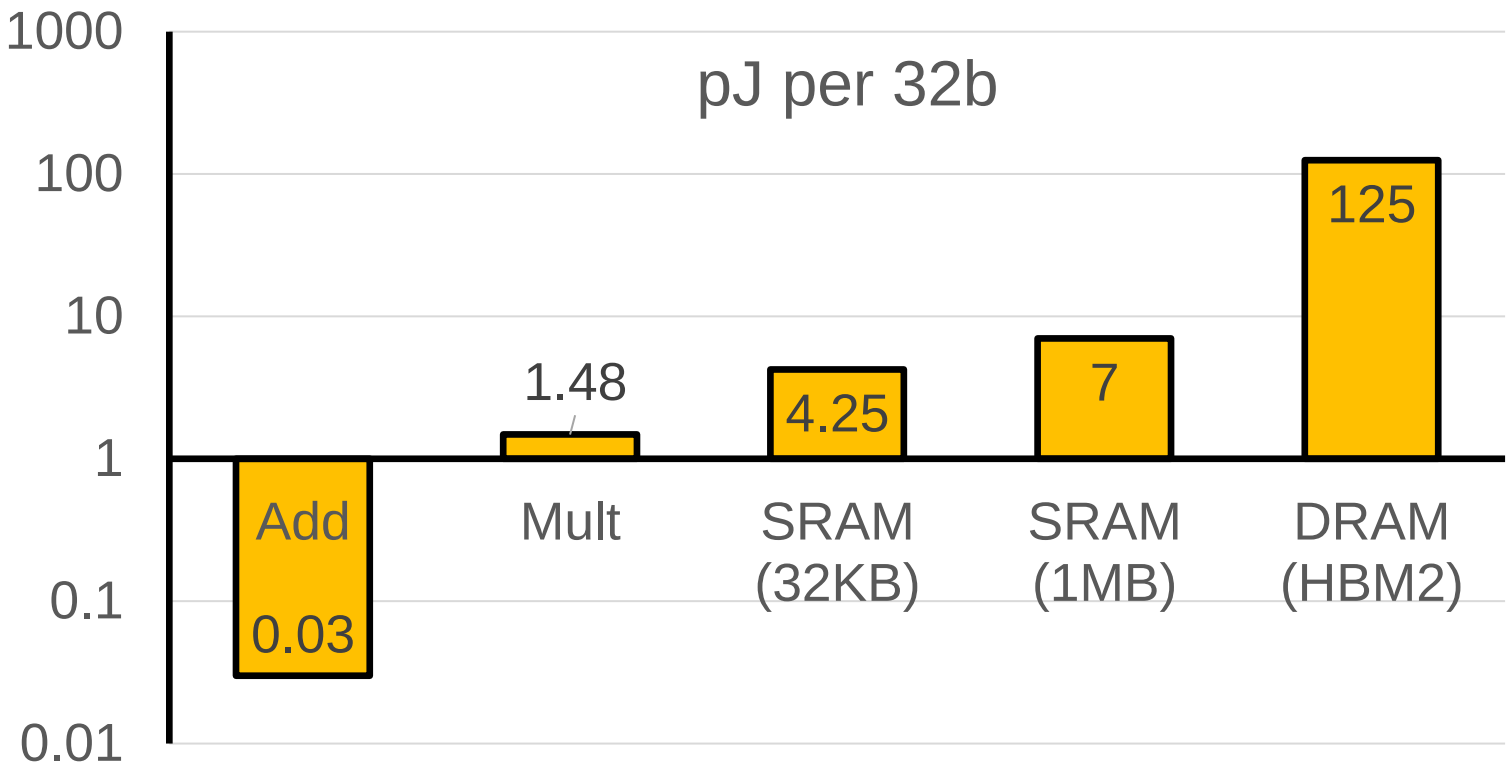


Operation	Time (ms)	Memory (GB)	IMAD (GOP)	Others (GOP)
NTT	6.7	16	21	34
BConv	3	5.6	8	6
The rest	20	31	9	30
Memory	0.6	0.6	-	-
Boot	**32**	**53**	**38**	**70**

[arxiv 2026]

"Theodosian: A Deep Dive into Memory-Hierarchy-Centric FHE Acceleration,"

W. Choi, H. Yu, J. Kim, H. Ji, J. Park, **J. Ahn**



7nm, [ISCA 2021] “Ten Lessons From Three Generations Shaped Google’s TPUv4i”

GPU key numbers over generations

*w/o sparsity

	V100 SXM2	A100 SXM4	H100 SXM5	B200 SXM6
FP32 (TFLOPS)	15.7		67	80
TF32 (TFLOPS)		156	495	1100
*FP16 (TFLOPS)	125	312	990	2250
*FP8 (TFLOPS)			1979	4500
*FP4 (TOPS)				9000
*INT8 (TOPS)		624	1979	4500
INT32 (TOPS)	14.1	19.5	25.6	
LLC capacity (MB)	6	40	50	126
HBM capacity (GB)	32	80	80	192
HBM BW (GB/s)	900	2039	3350	8000
Interconnect BW (GB/s)	300	600	900	1800
TDP (W)	300	400	700	1000
FP16 Op/B (arithmetic intensity)	139	153	295	281

DPRIVE:
Data protection in virtual environments

- Started 2021
  - "Four research teams take on development of novel hardware accelerator to enable new levels of data and privacy protection"
  - "DARPA Awards Duality Technologies Multi-Million Dollar Contract to Accelerate Machine Learning on Encrypted Data"

DPRIVE:
Data protection in virtual environments

- Its key milestone includes 7-layer CNN inference with CIFAR-10 data set per image.

Challenge		Phase 1: Building Blocks in emulation	Phase 2: Full Design in emulation	Phase 3: Prototype and Software Port
1	Ability to build and emulate all blocks (add, sub, mul, mod, shifts, transforms)	Binary yes/no		
1	Verification coverage of logic circuits	≥ 90%	100%	
1	Time to perform logic circuit verification	≤ 1 day	≤ 1 day	
2	Chip dimensions	≤ 150 mm^2 (estimate)	≤ 150 mm^2 (RTL)	≤ 150 mm^2 (real chip)
3	FHE parameter range: Plaintext Modulus	2 – 1024	2 – 1024	2 – 1024
3	FHE parameter range: Ciphertext Modulus	$2^{15} – 2^{500}$	$2^{15} – 2^{500}$	$2^{15} – 2^{500}$
3	FHE parameter range: RingSize	512 – 16384	512 – 16384	512 – 16384
Overall	Execution of a 1024-point logistic regression model	≤ 10 ms (100x)	≤ 1 ms (10x)	≤ 0.1 ms (1x)
	Execution of 7-layer CNN **inference** w/ CIFAR-10 data set per image		≤ 250 ms (100x)	≤ 25 ms (10x)
	Execution of 7-layer CNN **training** w/ CIFAR-10 data set over 10 epochs			≤ 10 hours (10x)

GPU-Driven FHE: Real-Time Private AI Inference Outperforming Custom ASIC [P73516]

Jongmin Kim, Graduate Student, Seoul National University

Hyesung Ji, Graduate Student, Seoul National University

Wonseok Choi, Graduate Student, Seoul National University

Sangpyo Kim, Graduate Student, Seoul National University

Jaiyoung Park, Graduate Student, Seoul National University

Jung Ho Ahn, Professor, Seoul National University

Discover how our GPU-accelerated fully homomorphic encryption (FHE) framework achieves real-time private AI inference, significantly outperforming custom application-specific integrated circuits (ASICs). Using the Cheddar CKKS library, novel packing techniques, and polynomial activation functions, we demonstrate the feasibility of secure AI on GPUs without compromising speed or accuracy.

Key Takeaways:

- Achieved sub-100ms inference time using a single GPU for FHE-based AI tasks
- Outperformed custom ASICs in the DARPA DPRIVE Phase 2 target for FHE CNN acceleration
- Developed innovative packing and polynomial activation techniques to enhance GPU efficiency
- Leveraged the Cheddar library for significant speedups in encrypted AI computations

Topic: Cybersecurity - Security for AI

FHE for Computer Architecture Researchers

- Numerous FHE-related papers are being published at architecture conferences
  - Over a dozen in 2026, already

HPCA 2026 UniFHE: Faster Accelerator for FHE with Diverse Algebraic Structure and Balanced Memory System	**HPCA 2026** Peregrine: Accelerating TFHE Bootstrapping on GPUs via Multi-Level External Product Co-Design	**HPCA 2026** Leveraging ASIC AI Chips for Homomorphic Encryption	**HPCA 2026** CROPHE: Cross-Operator Dataflow Optimization for Fully Homomorphic Encryption Accelerators	**HPCA 2026** An Efficient and Scalable Hardware Architecture for Number Theoretic Transform on FPGA with Design Automation
HPCA 2026 IVE: An Accelerator for Single-Server Private Information Retrieval Using a Versatile Processing Element	**HPCA 2026** Conflux: A High-Performance Keyword Private Retrieval System for Dynamic Datasets	**ASPLOS 2026** ReliaFHE: Resilient Design for Fully Homomorphic Encryption Accelerators	**ASPLOS 2026** A Framework for Developing and Optimizing Fully Homomorphic Encryption Programs on GPUs	**ASPLOS 2026** HEPIC: Private Inference over Homomorphic Encryption with Client Intervention
ASPLOS 2026 Falcon: Algorithm-Hardware Co-Design for Efficient Fully Homomorphic Encryption Accelerator	**ASPLOS 2026** Maverick: Rethinking TFHE Bootstrapping on GPUs via Algorithm-Hardware Co-Design	**ASPLOS 2026** Cheddar: A Swift Fully Homomorphic Encryption Library Designed for GPU Architectures	**ASPLOS 2026** CHEHAB RL: Learning to Optimize Fully Homomorphic Encryption Computations	• • •

- Yet, many computer architecture researchers find FHE very difficult

FHE is Difficult, Indeed

- For **fully understanding** FHE, substantial knowledge in mathematics and modern cryptography is required

- However, only a **basic understanding** of FHE might be enough for…
  - Using FHE to develop privacy-preserving applications
  - Understanding architectural papers about FHE
  - Or even advancing the state-of-the-art in FHE

Lemma 8 (Multiplicative Inverse). *Let* $(\boldsymbol{c}, \ell, p/2, B_0 = \beta_0 \cdot p/2)$ *be an encryption of* $\hat{m} \in \mathcal{S}$ *and let* $m = p - \hat{m}$. *Then Algorithm 2 outputs a valid encryption* $(\boldsymbol{v}_r, \ell - r, 2p, \beta \cdot 2p)$ *of* $m' = p \cdot \prod_{i=0}^{r-1}(1 + (\hat{m}/p)^{2^i})$ *for some* $\beta \leq \beta_0 + r\beta_*$.

Proof. From Lemma 4, $(\boldsymbol{v}_1, \ell - 1, 3p/2, B_0)$ is a valid encryption of $p + \hat{m}$ and its relative error is $\beta'_1 = \beta_0/3$. It also follows from Lemma 5 that $(\boldsymbol{c}_j, \ell - j, 2^{-2^j} \cdot p, \beta_j \cdot 2^{-2^j} \cdot p)$ is a valid encryption of $\hat{m}^{2^j}/p^{2^j-1}$ for some real number $\beta_j \leq 2^j \cdot (\beta_0 + \beta_*)$, and so $(\boldsymbol{p} + \boldsymbol{c}_j, \ell - j, (1 + 2^{-2^j})p, \beta_j \cdot 2^{-2^j} \cdot p)$ is a valid encryption of $p + \hat{m}^{2^j}/p^{2^j-1} = (p^{2^j} + \hat{m}^{2^j})/p^{2^j-1}$ with a relative error $\beta'_j \leq \beta_j/(2^{2^j} + 1) \leq 2^j \cdot (\beta_0 + \beta_*)/(2^{2^j} + 1)$, respectively.

Using the induction on j, we can show that

$$\left(\boldsymbol{v}_j, \ell - j, p \cdot \prod_{i=0}^{j-1}(1 + 2^{-2^i}), \beta''_j \cdot p \cdot \prod_{i=0}^{j-1}(1 + 2^{-2^i})\right)$$

is a valid encryption of $\prod_{i=0}^{j-1}(p^{2^i} + \hat{m}^{2^i})/p^{2^j-2} = p \cdot \prod_{i=0}^{j-1}(1 + (\hat{m}/p)^{2^i})$ with a relative error $\beta''_j \leq \sum_{i=0}^{j-1} \beta'_i + (j-1) \cdot \beta_*$. Note that the message is bounded by $p \cdot \prod_{i=0}^{j-1}(1 + 2^{-2^i}) = (2p) \cdot (1 - 2^{-2^j}) < 2p$ and the relative error satisfies

$$\beta''_j \leq \left(\sum_{i=0}^{j-1} \frac{2^i}{2^{2^i} + 1}\right) \cdot (\beta_0 + \beta_*) + (j-1) \cdot \beta_* \leq \beta_0 + j \cdot \beta_*$$

from the fact that $\sum_{i=0}^{\infty} \frac{2^i}{2^{2^i}+1} = 1$. Therefore, the output $\boldsymbol{v}_r$ of Algorithm 2 represents a valid encryption $(\boldsymbol{v}_r, \ell - r, 2p, \beta \cdot 2p)$ of $m' = p \cdot \prod_{i=0}^{r-1}(1 + (\hat{m}/p)^{2^i})$ for some $\beta \leq \beta_0 + r \cdot \beta_*$. □

Source: The "CKKS" paper
J. H. Cheon, et al. "Homomorphic Encryption for Arithmetic of Approximate Numbers." Asiacrypt, 2017.

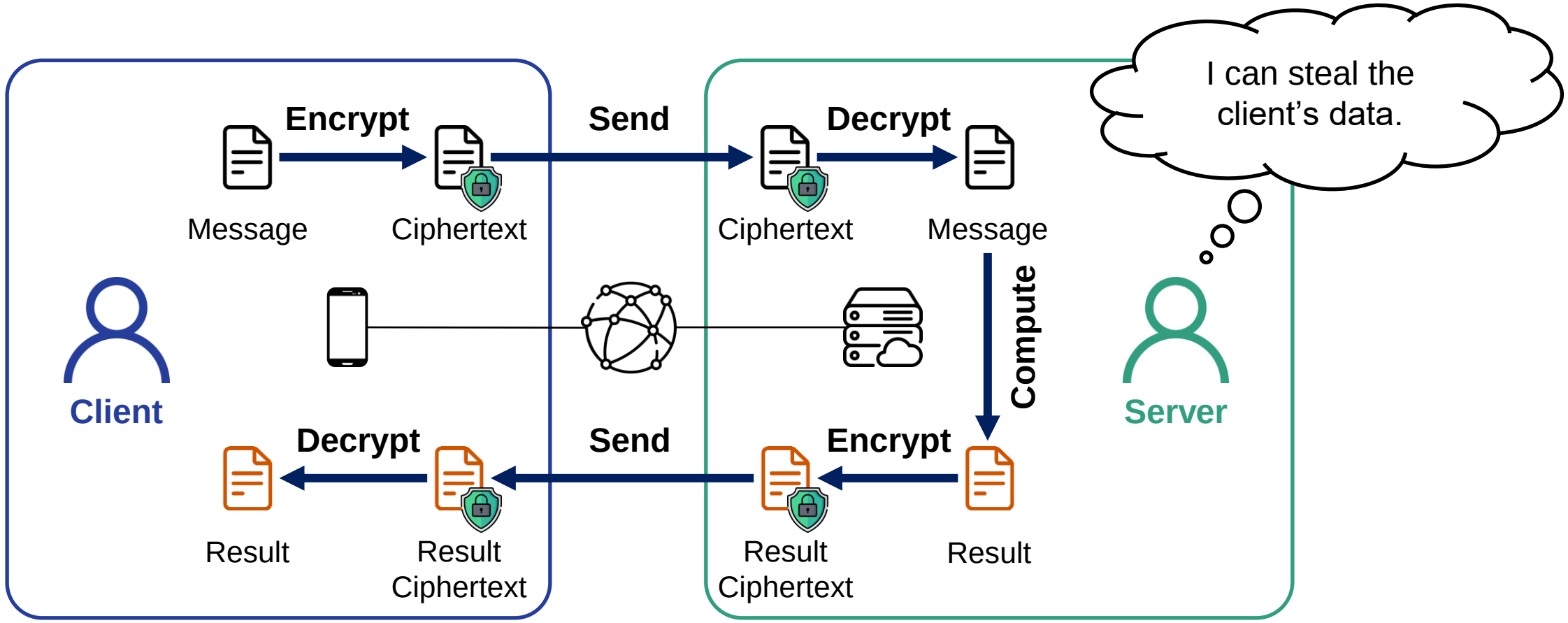
Through this Tutorial, the Participants Will…

- Get to know fully homomorphic encryption (FHE)
- Find out why FHE is crucial for privacy-preserving computation
- Understand FHE basics with a focus on CKKS, a prominent FHE scheme
  - Computational aspects of CKKS
  - Cryptographic aspects of CKKS
- Learn how to use Cheddar, a high-performance CKKS library for NVIDIA GPUs
- Develop privacy-preserving applications with Cheddar
  - Ranging from **basic addition** to **RNN inference**

Fully Homomorphic Encryption (FHE)

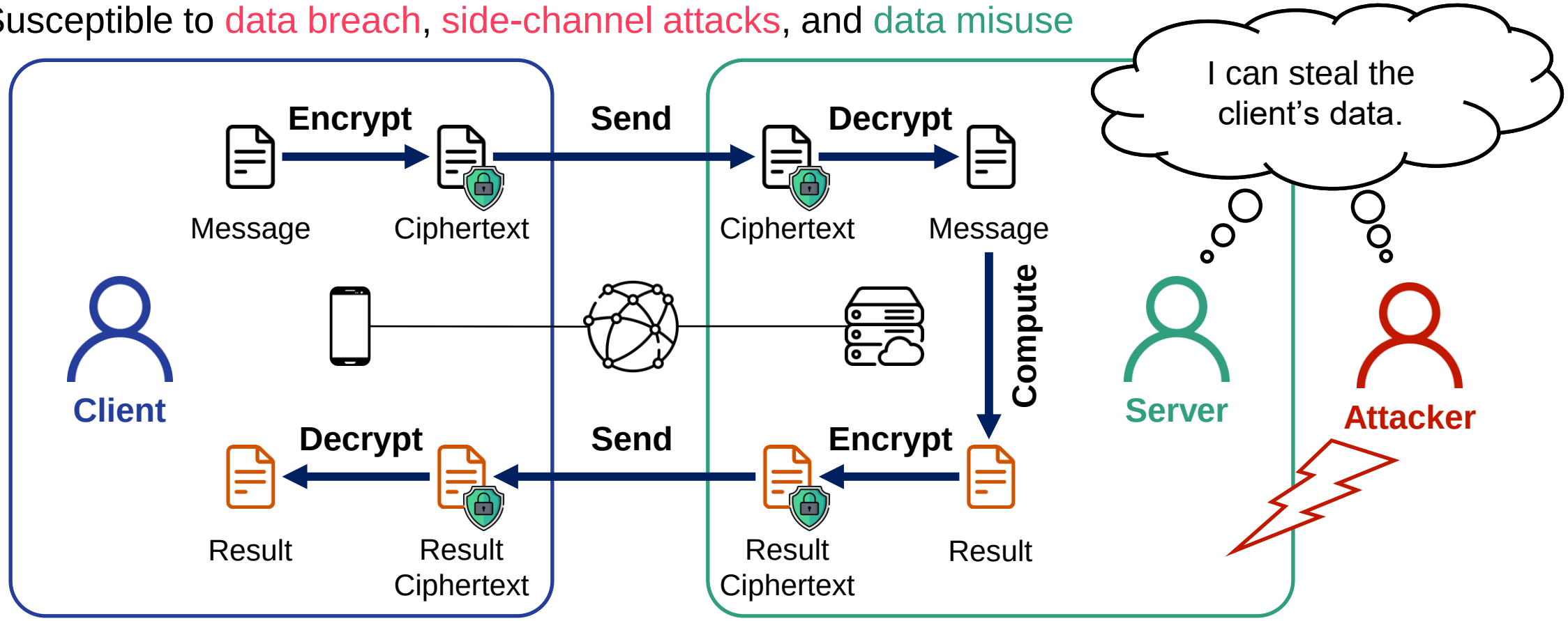
Conventional Client–Server Scenarios

- The client's message is known to the server
  - The server can use the client's data for other purposes w/o the client's consent



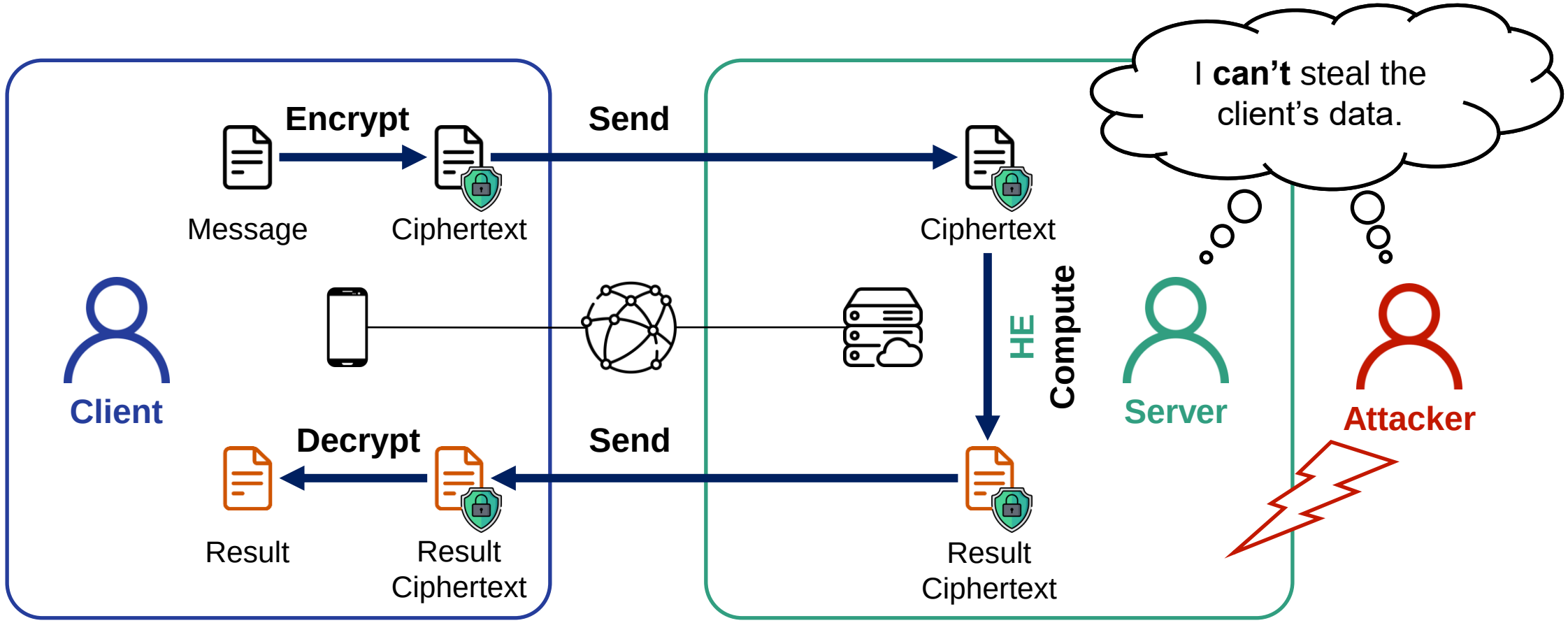
Conventional Client–Server Scenarios

- The client's message is known to the server
  - The server can use the client's data for other purposes w/o the client's consent
- Susceptible to data breach, side-channel attacks, and data misuse



Homomorphic Encryption (HE)

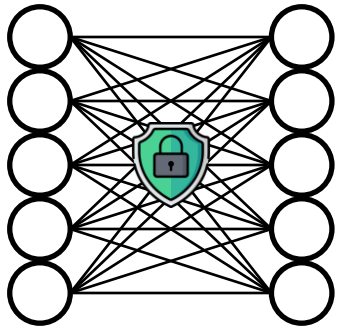
- HE enables direct computation on encrypted data without decryption
  - HE allows offloading computation to cloud servers without any leakage of private data



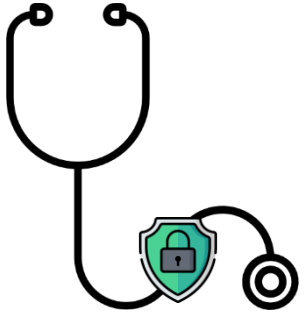
Homomorphic Encryption (HE)

- HE enables direct computation on encrypted data without decryption
  - HE allows offloading computation to cloud servers without any leakage of private data

- HE is emerging as a game-changer in **privacy-preserving computation**



Private
Machine Learning



Private
Healthcare

Private
Data Analytics

• • •

Homomorphic Encryption (HE)

- HE enables direct computation on encrypted data without decryption
  - HE allows offloading computation to cloud servers without any leakage of private data

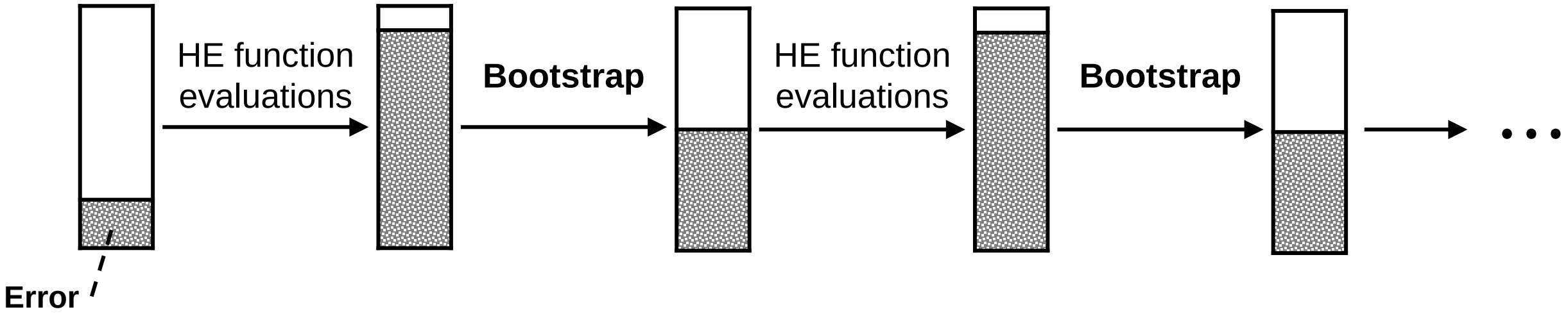
- HE is emerging as a game-changer in **privacy-preserving computation**

- Among many HE schemes, we target emerging **CKKS**
  - CKKS supports real arithmetic, thus is suitable for machine learning tasks

Schemes	Data Type in a Message	Operation
BGV, BFV	Finite integer $\mathbb{Z}_Q$	Integer (mod Q) arithmetic
FHEW, TFHE	Single bit $\{0,1\}$	Boolean (XOR, AND) arithmetic
CKKS	Real / complex numbers $\mathbb{R}$ / $\mathbb{C}$	Approximate (fixed-point) arithmetic

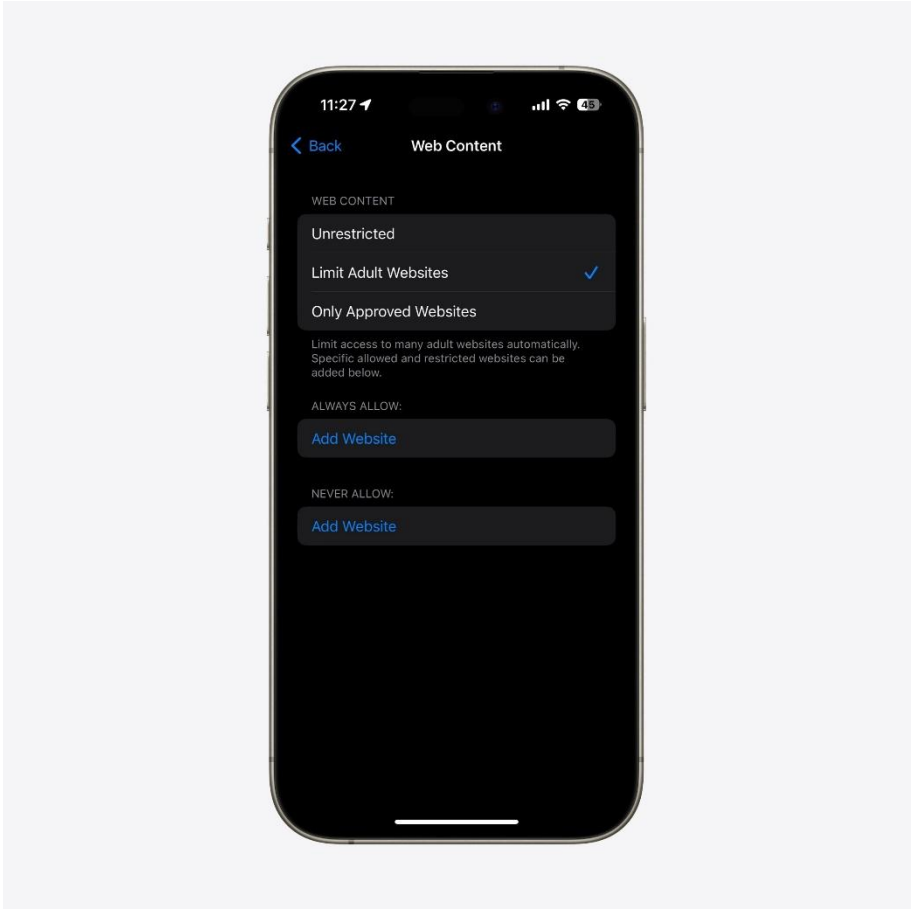
Fully Homomorphic Encryption (FHE)

- Error accumulates on a ciphertext over HE function evaluations
- Bootstrapping (partially) refreshes the **error**
- **Fully homomorphic encryption (FHE):** HE + bootstrapping → unlimited function evaluations



FHE Use Case 1: Apple

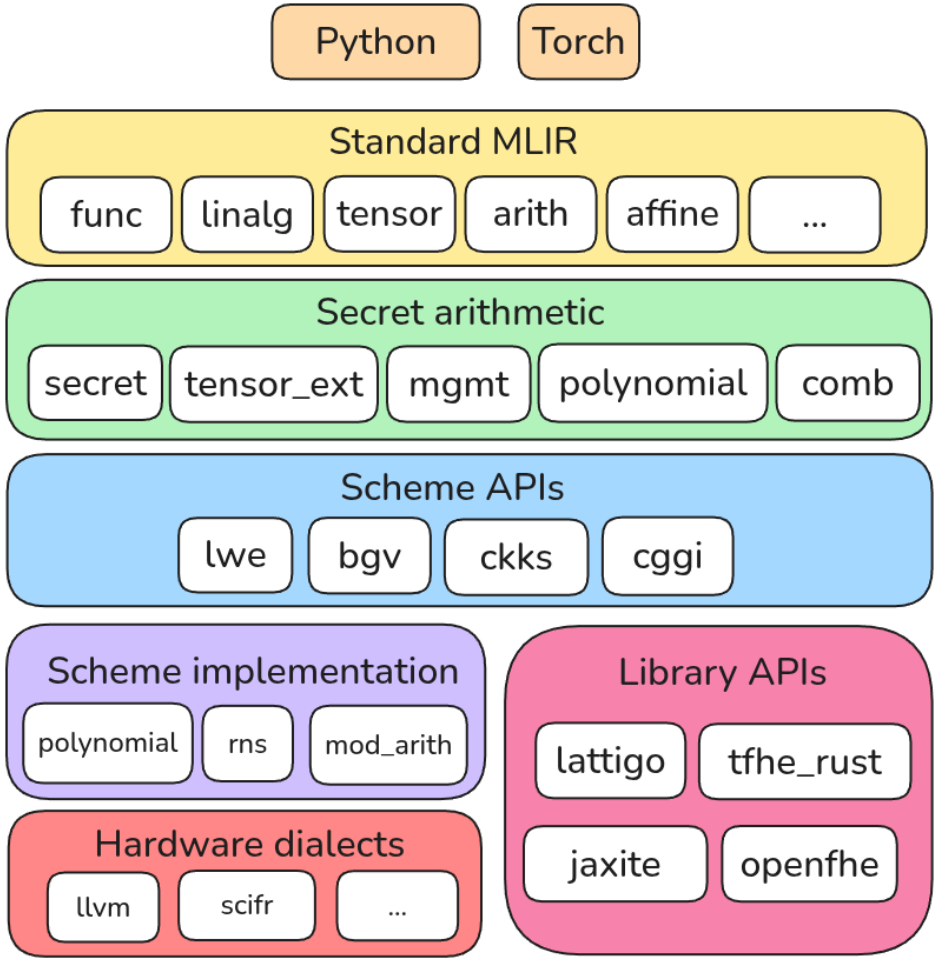
Private Caller ID Lookup



Private Web Content Filtering

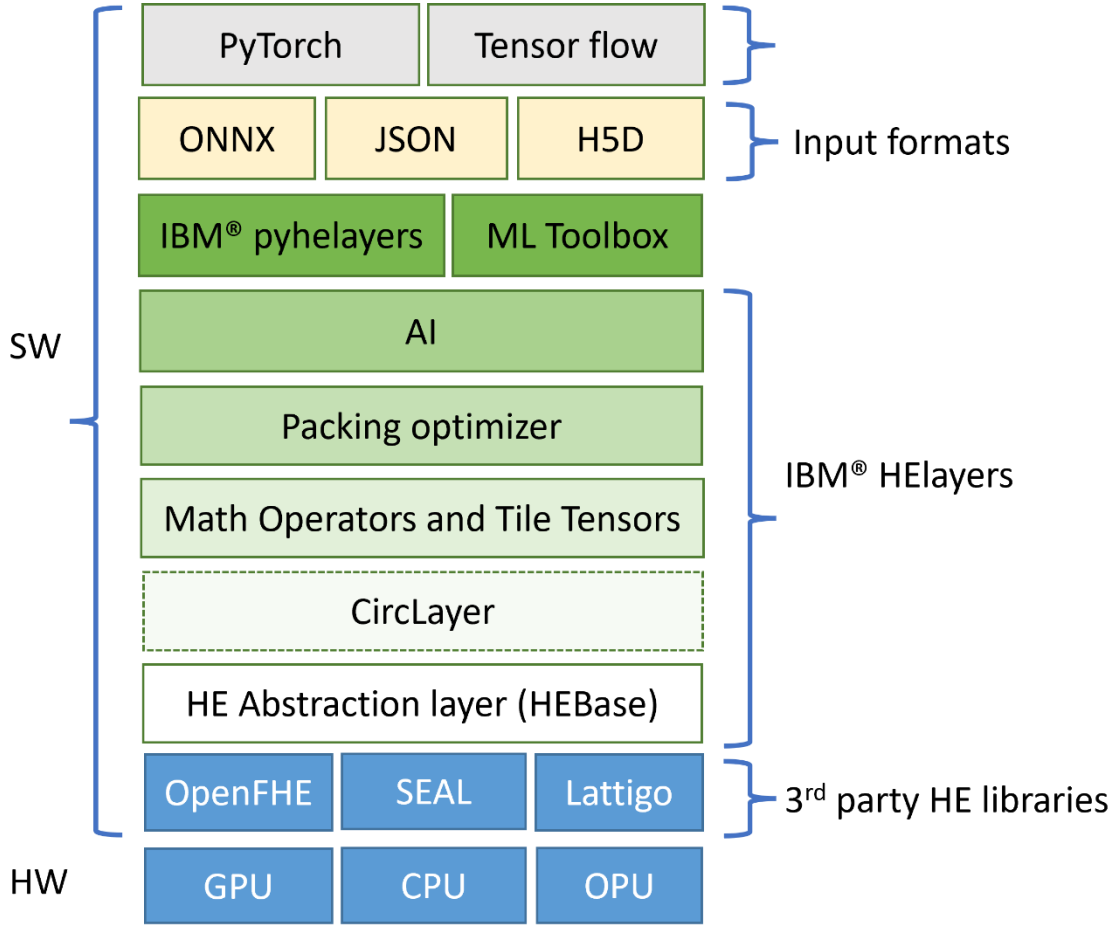
Source: Apple. “Combining Machine Learning and Homomorphic Encryption in the Apple Ecosystem.” https://machinelearning.apple.com/research/homomorphic-encryption

FHE Use Case 2: Encrypted AI



Google HEIR

Source: https://heir.dev/docs/design/



IBM HELayers

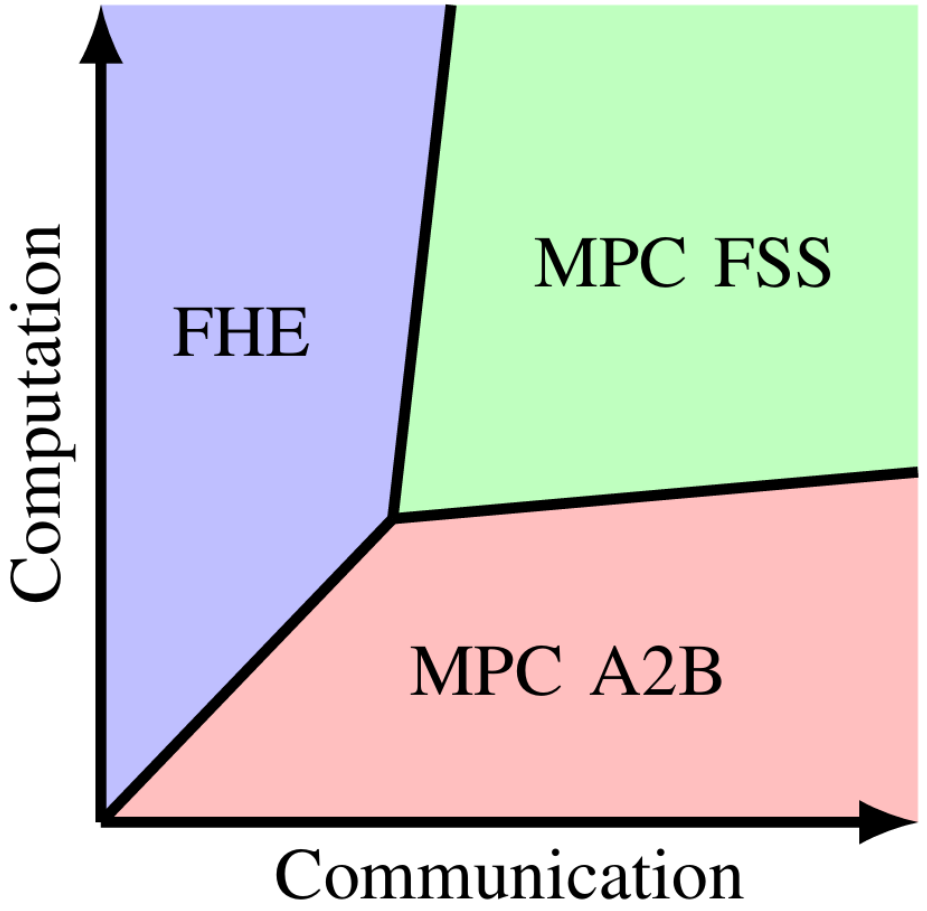
Source: https://ibm.github.io/helayers/user/overview.html

FHE vs. MPC

- FHE is often compared with **secure multi-party computation (MPC)**
- Various MPC protocols based on…
  - Function secret sharing (FSS)
  - Arithmetic / binary secret sharing (A2B)

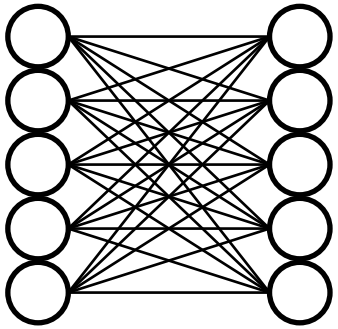
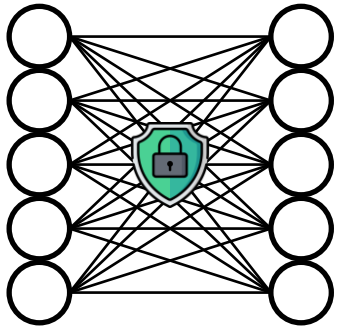
- Compared to MPC, FHE has
  - Lower communication demands
  - Greater computation costs

Increasing communication bandwidth for MPC
vs.
Designing systems for FHE with high computational throughput



Source: P. Huang, et al. "Beyond Latency: A System-Level Characterization of MPC and FHE for PPML.“ ISPASS, 2026.

Computational Challenges of FHE

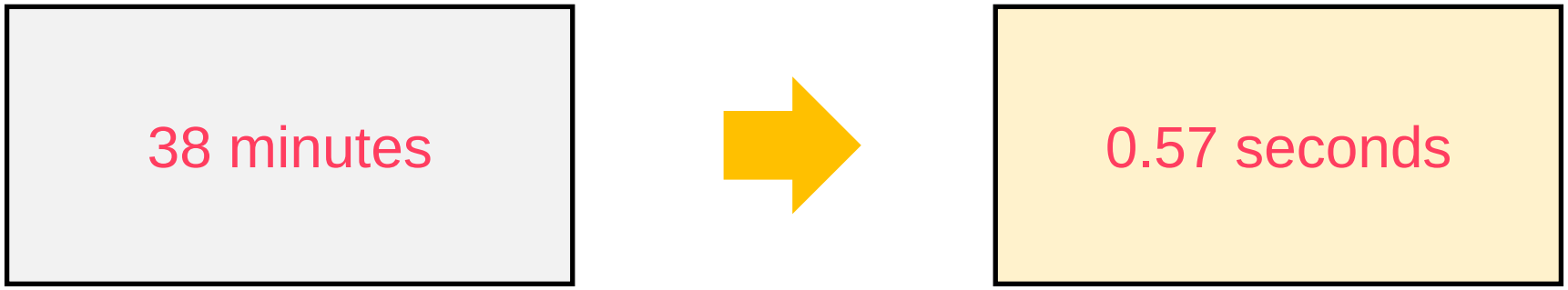
	Unencrypted computation	FHE-encrypted computation
		
1. Encryption results in severe data size expansion	1MB message	24MB ciphertext
2. HE operations are much more complex	Simple element-wise operation	Complex sequence of NTT / BConv / …
3. Bootstrapping		+ additional cost of bootstrapping
[e.g.] ResNet-20 inference (single-thread CPU)	< 0.1 second	38 minutes [1]

[1] E. Lee, et al. “Low-complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Parallel Convolutions.” ICML, 2022.

Hardware Acceleration is Crucial for FHE

- Hardware acceleration is crucial for FHE
  - GPU / FPGA / ASIC

[e.g.] ResNet-20 inference

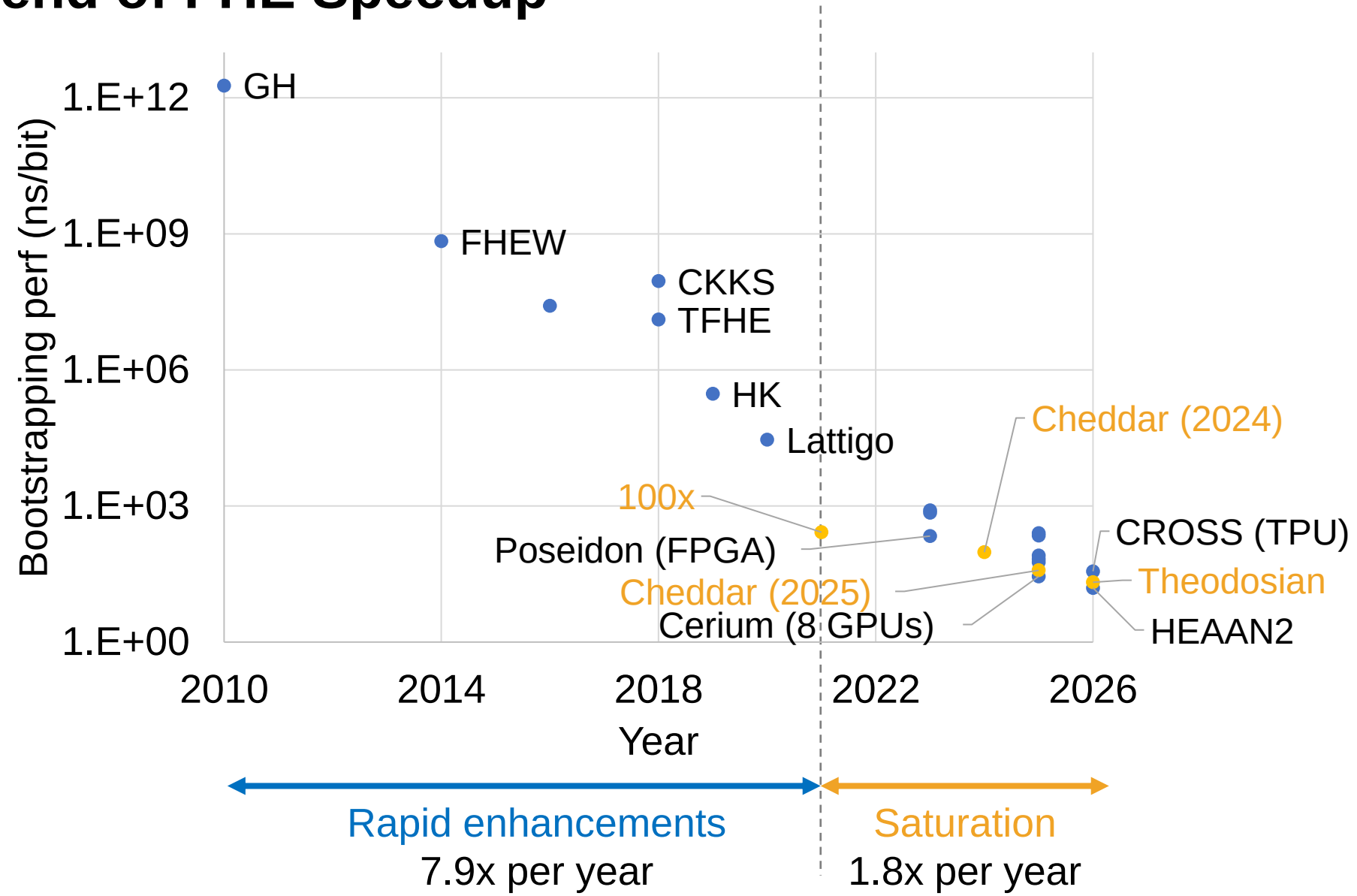


Single-thread CPU [1]

GPU (RTX 5090)
Cheddar 🧀

[1] E. Lee, et al. “Low-complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Parallel Convolutions.” ICML, 2022.

Historic Trend of FHE Speedup



Computational Aspects of CKKS

Polynomials

- Most FHE schemes, including CKKS, handle polynomials
  - **Notation:** $a(X)$ or simply a

- A polynomial is represented by an array of its coefficients

$$a(X) = 1 + 2X + 3X^2 + 4X^3$$

1	2	3	4

- We denote constant numbers (e.g., coefficient, parameter, …) in uppercase
  - e.g., A, N

Polynomial Ring

- These polynomials are defined in a ring
  - [Wikipedia] **ring** is an algebraic structure consisting of a set with two binary operations typically called **addition** and **multiplication**

- Therefore, the main operations are
  - Polynomial addition

$a(X) = 1 + 2X + 3X^2 + 4X^3$

1	2	3	4

$b(X) = 2 + 2X + 4X^2 + 4X^3$

2	2	4	4

$a(X) + b(X) = 3 + 4X + 7X^2 + 8X^3$

1+2 =3	2+2 =4	3+4 =7	4+4 =8

  - Polynomial multiplication

$$a(X) \cdot b(X) = 2 + 6X + 14X^2 + 26X^3 + 28X^4 + 28X^5 + 16X^6$$

Polynomial Ring

Complication 1

This ring has an integer **modulus** Q. Every coefficient must be reduced by Q.

(we will use an exemplar $Q = 7$ for now)

- Therefore, the main operations are
  - Polynomial addition

$a(X) = 1 + 2X + 3X^2 + 4X^3$

1	2	3	4

$b(X) = 2 + 2X + 4X^2 + 4X^3$

2	2	4	4

$a(X) + b(X) = 3 + 4X + \cancel{7}^{0}X^2 + \cancel{8}^{1}X^3$

1+2 =3	2+2 =4	3+4 =~~7~~ 0	4+4 =~~8~~ 1

  - Polynomial multiplication

$$a(X) \cdot b(X) = 2 + 6X + \cancel{14}^{0}X^2 + \cancel{26}^{5}X^3 + \cancel{28}^{0}X^4 + \cancel{28}^{0}X^5 + \cancel{16}^{2}X^6$$

Polynomial Ring

Complication 1

This ring has an integer **modulus** $\boldsymbol{Q}$. Every coefficient must be reduced by $\boldsymbol{Q}$.

(we will use an exemplar $\boldsymbol{Q} = 7$ for now)

- Therefore, the main operations are
  - Polynomial addition

$$a(X) = 1 + 2X + 3X^2 + 4X^3$$

1	2	3	4

$$b(X) = 2 + 2X + 4X^2 + 4X^3$$

2	2	4	4

$$a(X) + b(X) = 3 + 4X + 1X^3$$

1+2 =3	2+2 =4	3+4 =0	4+4 =1

  - Polynomial multiplication

$$a(X) \cdot b(X) = 2 + 6X + 5X^3 + 2X^6$$

Polynomial Ring

Complication 1

This ring has an integer **modulus** $\boldsymbol{Q}$. Every coefficient must be reduced by $\boldsymbol{Q}$.

(we will use an exemplar $\boldsymbol{Q = 7}$ for now)

Complication 2

This ring has a **degree** $\boldsymbol{N}$ (power-of-two). Polynomial is reduced by $(X^N + 1)$

(we will use an exemplar $\boldsymbol{N = 4}$ for now)

– Polynomial addition

$a(X) = 1 + 2X + 3X^2 + 4X^3$

1	2	3	4

$b(X) = 2 + 2X + 4X^2 + 4X^3$

2	2	4	4

$a(X) + b(X) = 3 + 4X + 1X^3$

1+2 =3	2+2 =4	3+4 =0	4+4 =1

– Polynomial multiplication

Reduction by $X^4 + 1$ is equivalent to replacing X^4 with -1

$a(X) \cdot b(X) = 2 + 6X + 5X^3 + 2X^6$ → $2 + 6X + 5X^2 + 5X^3$

2	6	5	5

Polynomial Ring

Complication 1

This ring has an integer **modulus Q**. Every coefficient must be reduced by Q.

(we will use an exemplar $Q = 7$ for now)

Complication 2

This ring has a **degree N** (power-of-two). Polynomial is reduced by $(X^N + 1)$

(we will use an exemplar $N = 4$ for now)

- Formal notation
  - Polynomials (e.g., a, b) belong to the ring $\mathbb{Z}_Q[X]/(X^N + 1)$, or $\mathcal{R}_Q$ in short:

$$a \in \mathcal{R}_Q, \qquad b \in \mathcal{R}_Q$$

  - Coefficients of a polynomial belong to $\mathbb{Z}_Q = \mathbb{Z}/Q\mathbb{Z}$ → simply think it as $\{0, 1, 2, \cdots, Q-1\}$
  - Addition and multiplication are defined over $\mathcal{R}_Q$:

$$a + b \in \mathcal{R}_Q, \qquad a \cdot b \in \mathcal{R}_Q$$

Actual Polynomials in CKKS

- CKKS data objects are composed of **huge polynomials**, such as

$$a = \underbrace{314 \cdots 1}_{} + 592 \cdots 6\, \boldsymbol{X} + 535 \cdots 8\, \boldsymbol{X^2} + \cdots + 979 \cdots 3\, \boldsymbol{X^{N-1}}$$

Extremely big integer coefficients:
modulus Q can be as large as 2^{1000}

Polynomial degree $< N$,
where $N = 2^{16}$ (typical)

Optimization 1
Residue number system (RNS)

Optimization 2
Number-theoretic transform (NTT)

Residue Number System (RNS)

- Suppose the modulus $Q = 3 \cdot 5 \cdot 7 = 105$
  - We can represent numbers in $\mathbb{Z}_Q$ (i.e., $[0, 104]$) with the residues divided by 3, 5, and 7
  - This conversion is unique for each element in $\mathbb{Z}_Q$

	22	**31**
	→ 1 mod 3	→ 1 mod 3
	→ 2 mod 5	→ 1 mod 5
	→ 1 mod 7	→ 3 mod 7

- Then, we can replace mod-Q computation with a set of mod-3/5/7 computations

Instead of this,	**22 + 31 = 53 mod *Q***	**22 * 31 = 52 mod *Q***
compute these	→ 1 + 1 = 2 mod 3	→ 1 * 1 = 1 mod 3
	→ 2 + 1 = 3 mod 5	→ 2 * 1 = 2 mod 5
	→ 1 + 3 = 4 mod 7	→ 1 * 3 = 3 mod 7

RNS for Polynomials: Addition

- Suppose the modulus $Q = 3 \cdot 5 \cdot 7 = 105$ and $N = 4$
  - For polynomials, we reduce each coefficient by 3/5/7
  - We get three small-coefficient polynomials

$\mathbf{22 + 23X + 24X^2 + 25X^3}$

→ $1 + 2X + 1X^3$ mod 3

→ $2 + 3X + 4X^2$ mod 5

→ $1 + 2X + 3X^2 + 4X^3$ mod 7

$\mathbf{31 + 32X + 33X^2 + 34X^3}$

→ $1 + 2X + 1X^3$ mod 3

→ $1 + 2X + 3X^2 + 4X^3$ mod 5

→ $3 + 4X + 5X^2 + 6X^3$ mod 7

- Then, we can replace mod-Q computation with a set of mod-3/5/7 computations

Instead of this, $\mathbf{22 + 23X + 24X^2 + 25X^3}$ + $\mathbf{31 + 32X + 33X^2 + 34X^3}$ = $\mathbf{53 + 55X + 57X^2 + 59X^3}$ **mod** $\boldsymbol{Q}$

compute these → $1 + 2X + 1X^3 + 1 + 2X + 1X^3 = 2 + 1X + 2X^3$ mod 3

→ $2 + 3X + 4X^2 + 1 + 2X + 3X^2 + 4X^3 = 3 + 2X^2 + 4X^3$ mod 5

→ $1 + 2X + 3X^2 + 4X^3 + 3 + 4X + 5X^2 + 6X^3 = 4 + 6X + X^2 + 3X^3$ mod 7

RNS for Polynomials: Addition

With RNS, a polynomial is expressed as an $L \times N$ **matrix**
when the modulus Q is decomposed into L **sub-moduli** $\{Q_0, Q_1, \cdots, Q_{L-1}\}$ **satisfying** $Q = \prod_i Q_i$
(we are using exemplar $Q = 3 \cdot 5 \cdot 7$ and $N = 4$ for now)
$L = 3$: $Q_0 = 3$, $Q_1 = 5$, $Q_2 = 7$

$22 + 23X + 24X^2 + 25X^3$

→	1	2	0	1	mod 3
→	2	3	4	0	mod 5
→	1	2	3	4	mod 7

$31 + 32X + 33X^2 + 34X^3$

→	1	2	0	1	mod 3
→	1	2	3	4	mod 5
→	3	4	5	6	mod 7

Row = Limb

- Then, we can replace mod-Q computation with a set of mod-3/5/7 computations

Instead of this, $22 + 23X + 24X^2 + 25X^3$ + $31 + 32X + 33X^2 + 34X^3$ = $53 + 55X + 57X^2 + 59X^3$ **mod** Q

compute these

→	1 + 1 = 2	2 + 2 = 1	0 + 0 = 0	1 + 1 = 2	mod 3
→	2 + 1 = 3	3 + 2 = 0	4 + 3 = 2	0 + 4 = 4	mod 5
→	1 + 3 = 4	2 + 4 = 6	3 + 5 = 1	4 + 6 = 3	mod 7

RNS for Actual Polynomials in CKKS

- RNS decomposes each coefficient modulo small primes $\{Q_0, Q_1, \cdots, Q_{L-1}\}$ satisfying $Q = \prod_i Q_i$

$$a = 314 \cdots 1 + 592 \cdots 6\, \boldsymbol{X} + 535 \cdots 8\, \boldsymbol{X^2} + \cdots + 979 \cdots 3\, \boldsymbol{X^{N-1}}$$

RNS

N

L	$314 \cdots 1 \ \% \ Q_0$	$592 \cdots 6 \ \% \ Q_0$	$535 \cdots 8 \ \% \ Q_0$	$\cdots$	$979 \cdots 3 \ \% \ Q_0$	*Limb 0*
	$314 \cdots 1 \ \% \ Q_1$	$592 \cdots 6 \ \% \ Q_1$	$535 \cdots 8 \ \% \ Q_1$	$\cdots$	$979 \cdots 3 \ \% \ Q_1$	*Limb 1*
	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
	$314 \cdots 1 \ \% \ Q_{L-1}$	$592 \cdots 6 \ \% \ Q_{L-1}$	$535 \cdots 8 \ \% \ Q_{L-1}$	$\cdots$	$979 \cdots 3 \ \% \ Q_{L-1}$	*Limb $L-1$*

$a[i][j] = (j\text{-th coefficient}) \ \% \ Q_i$

- **Result: $L \times N$ matrix of words**
  - Each **sub-modulus Q_i** is selected to fit in a **word** (e.g., INT32, INT64)

RNS for Polynomials: Multiplication

- Suppose the modulus $Q = 3 \cdot 5 \cdot 7 = 105$ and $N = 4$
  - For polynomials, we reduce each coefficient by 3/5/7
  - We get three small-coefficient polynomials

$\mathbf{22 + 23X + 24X^2 + 25X^3}$

→ $1 + 2X + 1X^3$ mod 3

→ $2 + 3X + 4X^2$ mod 5

→ $1 + 2X + 3X^2 + 4X^3$ mod 7

$\mathbf{31 + 32X + 33X^2 + 34X^3}$

→ $1 + 2X + 1X^3$ mod 3

→ $1 + 2X + 3X^2 + 4X^3$ mod 5

→ $3 + 4X + 5X^2 + 6X^3$ mod 7

- Then, we can replace mod-Q computation with a set of mod-3/5/7 computations

Instead of this, $\mathbf{(22 + 23X + 24X^2 + 25X^3) * (31 + 32X + 33X^2 + 34X^3) = 93 + 91X + 96X^2 + 5X^3}$ **mod** $\boldsymbol{Q}$

→ $(1 + 2X + 1X^3) * (1 + 2X + 1X^3) = X + 2X^3$ mod 3

→ $(2 + 3X + 4X^2) * (1 + 2X + 3X^2 + 4X^3) = 3 + X + X^2$ mod 5

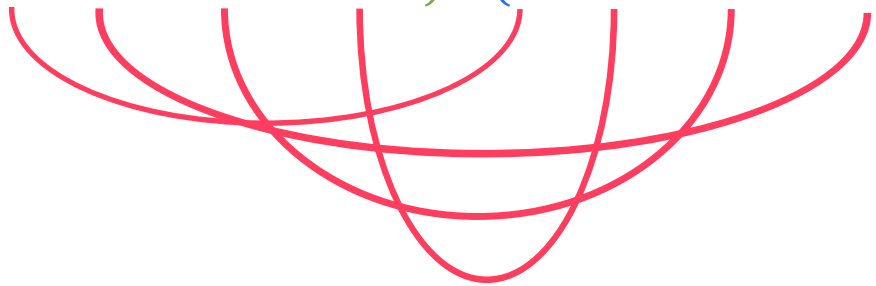
→ $(1 + 2X + 3X^2 + 4X^3) * (3 + 4X + 5X^2 + 6X^3) = 2 + 5X^2 + 5X^3$ mod 7

This is still very complex, especially for large $N = 2^{16}$

RNS for Polynomials: Multiplication

1	2	0	1	1	2	0	1	mod 3
2	3	4	0	1	2	3	4	mod 5
1	2	3	4	3	4	5	6	mod 7

→ $(1 + 2X + 3X^2 + 4X^3) * (3 + 4X + 5X^2 + 6X^3) = 2 + 5X^2 + 5X^3$ mod 7



Reduction by $X^4 + 1$
is equivalent to
plugging in $X^4 = -1$

Const term	$1 * 3 - 2 * 6 - 3 * 5 - 4 * 4$
X term	$1 * 4 + 2 * 3 - 3 * 6 - 4 * 5$
X^2 term	$1 * 5 + 2 * 4 + 3 * 3 - 4 * 6$
X^3 term	$1 * 6 + 2 * 5 + 3 * 4 + 4 * 3$

Negacyclic convolution

1	2	3	4	*	3	-6	-5	-4
					4	3	-6	-5
					5	4	3	-6
					6	5	4	3

RNS for Polynomials: Multiplication

Polynomial multiplication = L independent inter-limb multiplications

Inter-limb multiplication = (Negacyclic) convolution

Convolution —**Fourier transform**→ Element-wise multiplication

Convolution theorem: $\mathcal{F}(a * b) = \mathcal{F}(a) \odot \mathcal{F}(b)$

Convolution in the time domain

Element-wise multiplication in the frequency domain

Number-Theoretic Transform (NTT)

- NTT is a special variant of Fourier transform
  - Integer sequence
  - Sub-modulus Q_i for each limb
  - Converts **negacyclic convolution** to **element-wise multiplication**

- NTT computation = **fast Fourier transform (FFT)**
  - Computational complexity of $\mathcal{O}(N \log N)$

Number-Theoretic Transform (NTT)

1	2	0	1	mod 3
2	3	4	0	mod 5
1	2	3	4	mod 7

NTT for each limb

A1	B1	C1	D1	mod 3
E1	F1	G1	H1	mod 5
I1	J1	K1	L1	mod 7

1	2	0	1	mod 3
1	2	3	4	mod 5
3	4	5	6	mod 7

NTT for each limb

A2	B2	C2	D2	mod 3
E2	F2	G2	H2	mod 5
I2	J2	K2	L2	mod 7

A1 * A2 = Ar	B1 * B2 = Br	C1 * C2 = Cr	D1 * D2 = Dr	mod 3
E1 * E2 = Er	F1 * F2 = Fr	G1 * G2 = Gr	H1 * H2 = Hr	mod 5
I1 * I2 = Ir	J1 * J2 = Jr	K1 * K2 = Kr	L1 * L2 = Lr	mod 7

We often delay INTT until absolutely necessary

Inverse NTT (INTT) for each limb

0	1	0	2	mod 3
3	1	1	0	mod 5
2	0	5	5	mod 7

Discussed So Far

- In CKKS, we deal with polynomials in a ring $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N + 1)$
  - Polynomial addition and multiplication is defined over $\mathcal{R}_Q$
  - Parameters: modulus Q, degree N

- RNS converts a polynomial into an $L \times N$ matrix
  - Sub-moduli $\{\mathcal{Q}_0, \mathcal{Q}_1, \cdots, \mathcal{Q}_{L-1}\}$ is used for RNS, where $Q = \prod_i \mathcal{Q}_i$
  - Row = limb, and i^{th} limb corresponds to $\mathcal{Q}_i$

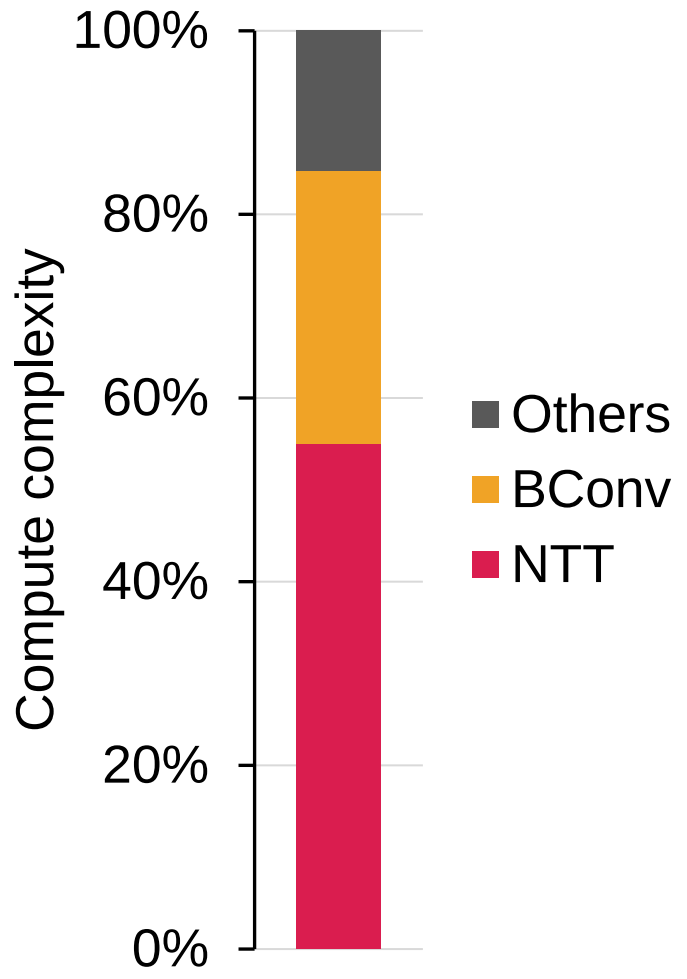
- Polynomial addition = element-wise addition between $L \times N$ matrices
- Polynomial multiplication = element-wise multiplication between $L \times N$ matrices (after NTT)
  - NTT is a Fourier transform applied to each limb independently

- Polynomials are kept in the NTT domain by default

Computational Breakdown

- NTT is the most dominant operation in CKKS
- Others = mostly element-wise operations

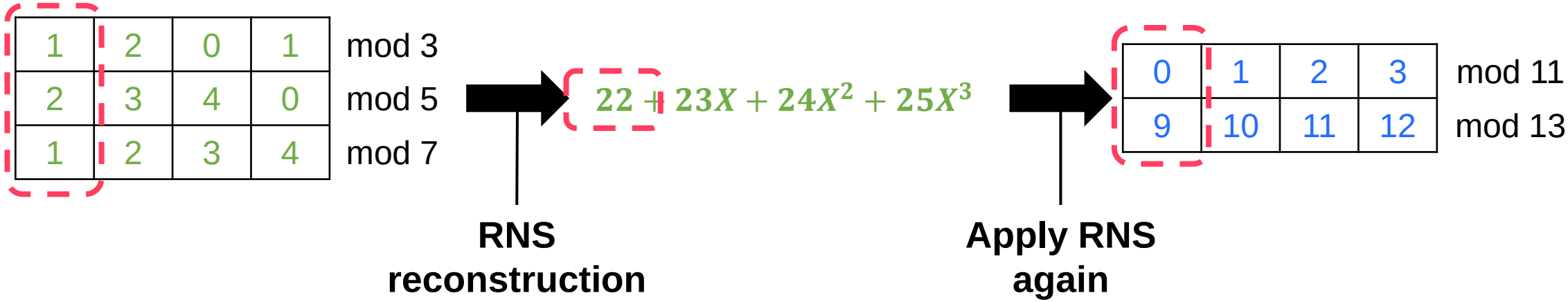
- BConv?



Changing the Modulus

- Changing the modulus from $Q = Q_0 Q_1 \cdots Q_{L-1}$ to $P = \mathcal{P}_0 \mathcal{P}_1 \cdots \mathcal{P}_{K-1}$ $(Q < P)$
  - We will look at an example of $Q = 3 \cdot 5 \cdot 7 = 105$ $(L = 3)$ & $P = 11 \cdot 13 = 143$ $(K = 2)$

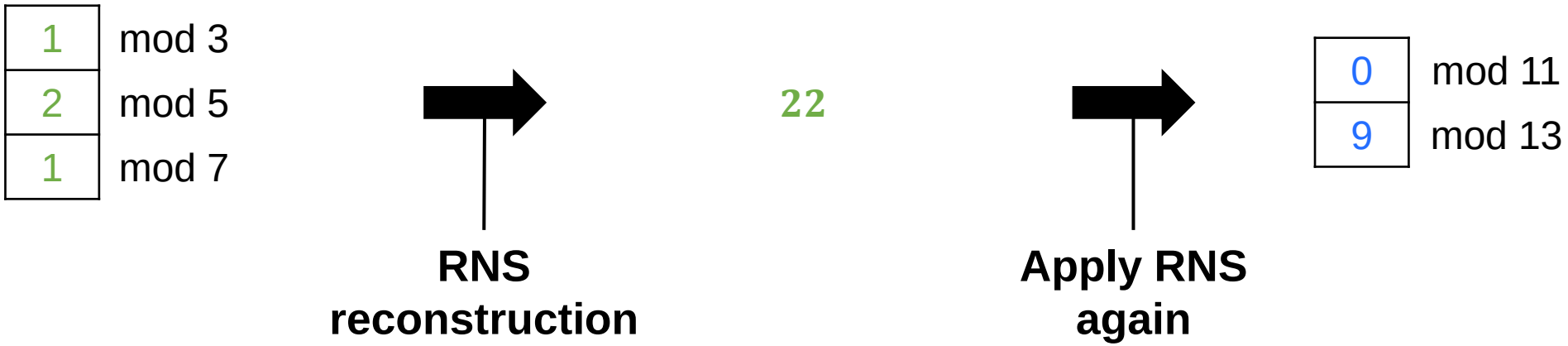
- Basic method



Changing the Modulus

- Changing the modulus from $Q = q_0 q_1 \cdots q_{L-1}$ to $P = p_0 p_1 \cdots p_{K-1}$ $(Q < P)$
  - We will look at an example of $Q = 3 \cdot 5 \cdot 7 = 105$ $(L = 3)$ & $P = 11 \cdot 13 = 143$ $(K = 2)$

- Basic method



Chinese Remainder Theorem (CRT) for RNS Reconstruction

- **Problem:** Find the smallest natural number A that satisfies
  - A = x mod 3
  - A = y mod 5
  - A = z mod 7

1	mod 3
2	mod 5
1	mod 7

- **Answer**

$$\left\{ \begin{array}{l} (x \cdot (5 \cdot 7)^{-1} \ \%\ 3) \cdot (5 \cdot 7) + \\ (y \cdot (3 \cdot 7)^{-1} \ \%\ 5) \cdot (3 \cdot 7) + \\ (z \cdot (3 \cdot 5)^{-1} \ \%\ 7) \cdot (3 \cdot 5) \end{array} \right\} \ \%\ 105$$

$$\begin{array}{l} (1 \cdot 2 \ \%\ 3) \cdot (5 \cdot 7) + \\ (2 \cdot 1 \ \%\ 5) \cdot (3 \cdot 7) + \\ (1 \cdot 1 \ \%\ 7) \cdot (3 \cdot 5) \end{array}$$

$$= 70 + 42 + 15 = 127 = \mathbf{22} \ \text{mod}\ 105$$

22

Generalization of RNS Reconstruction

- **Problem:** Find $A \in \mathbb{Z}_Q$ that satisfies

$$\begin{aligned} A &= A_0 \mod Q_0 \\ A &= A_1 \mod Q_1 \\ &\vdots \\ A &= A_{L-1} \mod Q_{L-1} \end{aligned}$$

- **Answer**

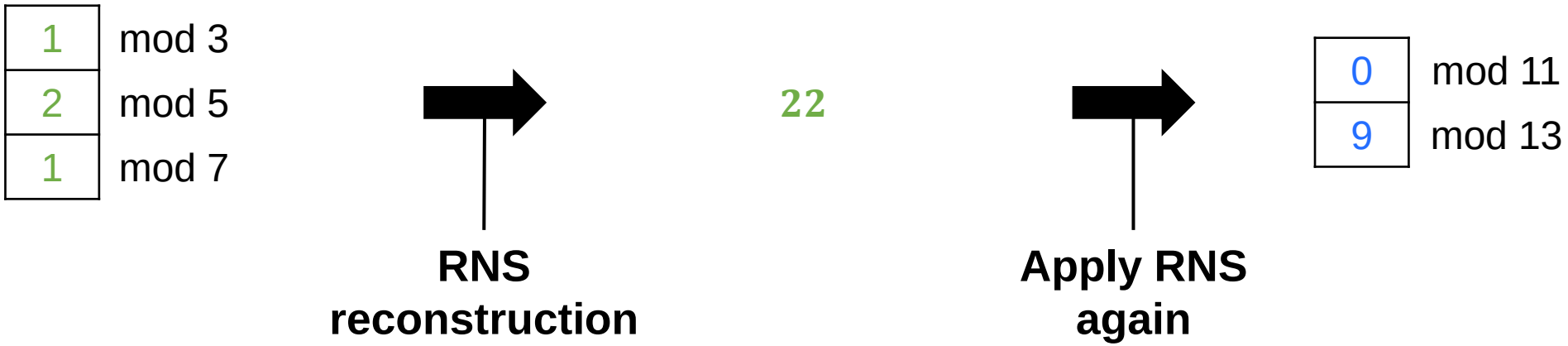
$$A = \left\{ \sum_{i=0}^{L-1} \left(A_i \cdot \left(\frac{Q}{Q_i} \right)^{-1} \ \% \ Q_i \right) \cdot \frac{Q}{Q_i} \right\} \ \% \ Q$$

where,

$$Q = \prod_{i=0}^{L-1} Q_i$$

Changing the Modulus

- **RNS reconstruction:** $A = \left\{\sum_{i=0}^{L-1}\left(A_i \cdot \left(\frac{Q}{\mathcal{Q}_i}\right)^{-1} \ \% \ \mathcal{Q}_i\right) \cdot \frac{Q}{\mathcal{Q}_i}\right\} \ \% \ Q$
- **Applying RNS:** $A \mod 11$ & $A \mod 13$



- **Merged:** $\left[\left\{\sum_{i=0}^{L-1}\left(A_i \cdot \left(\frac{Q}{\mathcal{Q}_i}\right)^{-1} \ \% \ \mathcal{Q}_i\right) \cdot \frac{Q}{\mathcal{Q}_i}\right\} \ \% \ Q\right] \ \% \ \mathcal{P}_j$ (for each $\mathcal{P}_j$)
  - In our example, $\mathcal{P}_0 = 11$ & $\mathcal{P}_1 = 13$

Fast but Approximate Modulus Changing (BConv)

- **Exact:** $\left[\left\{\sum_{i=0}^{L-1}\left(A_i \cdot \left(\frac{Q}{Q_i}\right)^{-1} \ \% \ Q_i\right) \cdot \frac{Q}{Q_i}\right\} \ \% \ Q\right] \ \% \ \mathcal{P}_j$
  - In our example, $\mathcal{P}_0 = 11$ & $\mathcal{P}_1 = 13$

- **BConv:** $\underline{\left\{\sum_{i=0}^{L-1}\left(A_i \cdot \left(\frac{Q}{Q_i}\right)^{-1} \ \% \ Q_i\right) \cdot \frac{Q}{Q_i}\right\}} \ \% \ \mathcal{P}_j$

  This actually recovers $A + Q \cdot I$
  and not exact A,
  where $I \in [0, L-1]$

$$
\begin{aligned}
&(1 \cdot 2 \mod 3) \cdot (5 \cdot 7) + \\
&(2 \cdot 1 \mod 5) \cdot (3 \cdot 7) + \\
&(1 \cdot 1 \mod 7) \cdot (3 \cdot 5) \\
= \underline{70 + 42 + 15 = 127} &= \mathbf{22} \mod 105
\end{aligned}
$$

$22 + 105 \cdot 1$ recovered

Fast but Approximate Modulus Changing (BConv)

- **Exact:** $\left[\left\{\sum_{i=0}^{L-1}\left(A_i \cdot \left(\frac{Q}{Q_i}\right)^{-1} \ \%\ Q_i\right)\cdot \frac{Q}{Q_i}\right\} \ \%\ Q\right] \ \%\ \mathcal{P}_j$

  Multiplication & reduction by a large constant

  - In our example, $\mathcal{P}_0 = 11$ & $\mathcal{P}_1 = 13$

- **BConv:** $\left\{\sum_{i=0}^{L-1}\left(A_i \cdot \left(\frac{Q}{Q_i}\right)^{-1} \ \%\ Q_i\right)\cdot \frac{Q}{Q_i}\right\} \ \%\ \mathcal{P}_j$

  Multiplication & reduction by a small constant

This actually recovers $A + Q \cdot I$
and not exact A,
where $I \in [0, L-1]$

$$
\begin{aligned}
&(1 \cdot 2 \mod 3)\cdot(5\cdot 7) + \\
&(2 \cdot 1 \mod 5)\cdot(3\cdot 7) + \\
&(1 \cdot 1 \mod 7)\cdot(3\cdot 5) \\
= 70 + 42 + 15 = 127 &= 22 \mod 105
\end{aligned}
$$

$22 + 105 \cdot 1$ recovered

Computation of BConv (Constant)

- **Bconv:** For each $\mathcal{P}_j$, compute $\left\{\sum_{i=0}^{L-1}\left(A_i \cdot \left(\frac{Q}{Q_i}\right)^{-1} \ \% \ Q_i\right) \cdot \frac{Q}{Q_i}\right\} \ \% \ \mathcal{P}_j$
- Step 1: Multiply **element-wise** by $\left(\frac{Q}{Q_i}\right)^{-1} \ \% \ Q_i$

$(5 \cdot 7)^{-1} \ \% \ 3$	$\odot$	1	mod 3	$=$	**2**	mod 3
$(3 \cdot 7)^{-1} \ \% \ 5$		2	mod 5		**2**	mod 5
$(3 \cdot 5)^{-1} \ \% \ 7$		1	mod 7		**1**	mod 7

Computation of BConv (Constant)

- **Bconv:** For each $\mathcal{P}_j$, compute $\left\{\sum_{i=0}^{L-1}\left(A_i \cdot \left(\frac{Q}{Q_i}\right)^{-1} \% Q_i\right) \cdot \frac{Q}{Q_i}\right\} \% \mathcal{P}_j$
- Step 1: Multiply **element-wise** by $\left(\frac{Q}{Q_i}\right)^{-1} \% Q_i$

$(5 \cdot 7)^{-1} \% 3$	$\odot$	1	mod 3	=	2 mod 3
$(3 \cdot 7)^{-1} \% 5$		2	mod 5		2 mod 5
$(3 \cdot 5)^{-1} \% 7$		1	mod 7		1 mod 7

- Step 2: Multiply by $\frac{Q}{Q_i} \% \mathcal{P}_j$ and accumulate → **GEMM**

$(5 \cdot 7) \% 11$	$(3 \cdot 7) \% 11$	$(3 \cdot 5) \% 11$	*	2
$(5 \cdot 7) \% 13$	$(3 \cdot 7) \% 13$	$(3 \cdot 5) \% 13$		2
				1

Computation of BConv (Polynomial)

- **Bconv:** For each $\mathcal{P}_j$, compute $\left\{\sum_{i=0}^{L-1}\left(a_i \cdot \left(\frac{Q}{Q_i}\right)^{-1} \% \ Q_i\right) \cdot \frac{Q}{Q_i}\right\} \% \ \mathcal{P}_j$
- Step 1: Multiply **limb-wise** by $\left(\frac{Q}{Q_i}\right)^{-1} \% \ Q_i$

$(5 \cdot 7)^{-1} \% \ 3$
$(3 \cdot 7)^{-1} \% \ 5$
$(3 \cdot 5)^{-1} \% \ 7$

$\odot$

1	2	0	1	mod 3
2	3	4	0	mod 5
1	2	3	4	mod 7

$=$

2	**1**	**0**	**2**	mod 3
2	**3**	**4**	**0**	mod 5
1	**2**	**3**	**4**	mod 7

- Step 2: Multiply by $\frac{Q}{Q_i} \% \ \mathcal{P}_j$ and accumulate → **GEMM**

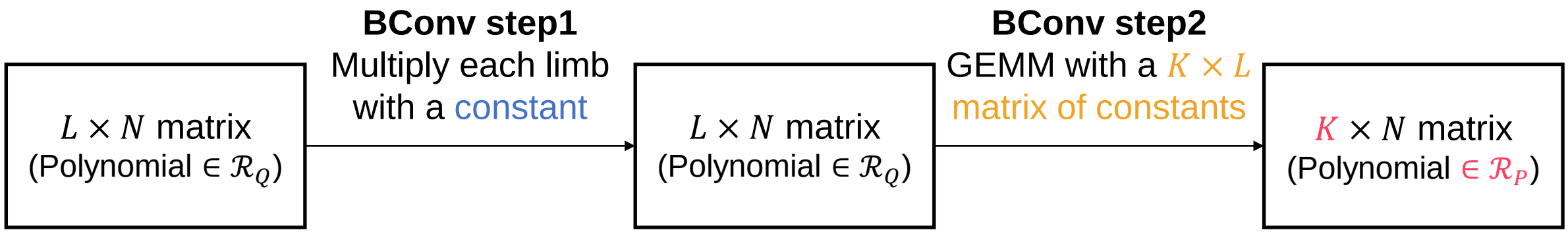
$(5 \cdot 7) \% \ 11$	$(3 \cdot 7) \% \ 11$	$(3 \cdot 5) \% \ 11$
$(5 \cdot 7) \% \ 13$	$(3 \cdot 7) \% \ 13$	$(3 \cdot 5) \% \ 13$

$*$

2	**1**	**0**	**2**
2	**3**	**4**	**0**
1	**2**	**3**	**4**

BConv Computation in a Nutshell

- Converting the modulus from $Q = \prod_{i=0}^{L-1} Q_i$ to $P = \prod_{i=0}^{K-1} \mathcal{P}_i$



BConv Routine

- BConv cannot be performed on polynomials in the NTT domain
- Polynomials are by default in the NTT domain

- Therefore, BConv **almost always** involves a computational routine of **INTT → BConv → NTT**

Four Types of Polynomial Operations

(I)NTT

- Number-theoretic transform (NTT) & inverse NTT (INTT)
- Fourier transform on **each limb**

$A_0 \,\%\, Q_0$	$\cdots$	$A_{N-1} \,\%\, Q_0$	$\mathcal{F}_{Q_0}$
$\vdots$	$\ddots$	$\vdots$	
$A_0 \,\%\, Q_{L-1}$	$\cdots$	$A_{N-1} \,\%\, Q_{L-1}$	$\mathcal{F}_{Q_{L-1}}$

BConv

- Base conversion
- **Matrix-matrix multiplication**, mostly

$T_{0,0}$	$\cdots$	$T_{0,L-1}$
$\vdots$	$\ddots$	$\vdots$
$T_{K-1,0}$	$\cdots$	$T_{K-1,L-1}$

BConv table ($K \times L$ matrix)

$\cdot$

$A_0 \,\%\, Q_0$	$\cdots$	$A_{N-1} \,\%\, Q_0$
$\vdots$	$\ddots$	$\vdots$
$A_0 \,\%\, Q_{L-1}$	$\cdots$	$A_{N-1} \,\%\, Q_{L-1}$

Polynomial ($L \times N$ matrix)

Automorphism

- Shuffling the order of columns according to an index mapping

$A_{123} \,\%\, Q_0$	$\cdots$	$A_{456} \,\%\, Q_0$
$\vdots$	$\ddots$	$\vdots$
$A_{123} \,\%\, Q_{L-1}$	$\cdots$	$A_{456} \,\%\, Q_{L-1}$

Element-wise operations (the rest)

- Matrix-matrix or matrix-constant addition, subtraction, Hadamard product, …

$$a * b = \begin{array}{|c|c|c|} \hline (A_0 * B_0) \,\%\, Q_0 & \cdots & (A_{N-1} * B_{N-1}) \,\%\, Q_0 \\ \hline \vdots & \ddots & \vdots \\ \hline (A_0 * B_0) \,\%\, Q_{L-1} & \cdots & (A_{N-1} * B_{N-1}) \,\%\, Q_{L-1} \\ \hline \end{array}$$

BConv Routine

- Main sequence in CKKS

(I)NTT

- Number-theoretic transform (NTT) & inverse NTT (INTT)
- Fourier transform on **each limb**

$A_0 \,\%\, Q_0$	$\cdots$	$A_{N-1} \,\%\, Q_0$	$\mathcal{F}_{Q_0}$
$\vdots$	$\ddots$	$\vdots$	
$A_0 \,\%\, Q_{L-1}$	$\cdots$	$A_{N-1} \,\%\, Q_{L-1}$	$\mathcal{F}_{Q_{L-1}}$

BConv

- Base conversion
- **Matrix-matrix multiplication**, mostly

$T_{0,0}$	$\cdots$	$T_{0,L-1}$
$\vdots$	$\ddots$	$\vdots$
$T_{K-1,0}$	$\cdots$	$T_{K-1,L-1}$

BConv table ($K \times L$ matrix)

$\cdot$

$A_0 \,\%\, Q_0$	$\cdots$	$A_{N-1} \,\%\, Q_0$
$\vdots$	$\ddots$	$\vdots$
$A_0 \,\%\, Q_{L-1}$	$\cdots$	$A_{N-1} \,\%\, Q_{L-1}$

Polynomial ($L \times N$ matrix)

Automorphism

- Shuffling the order of columns according to an index mapping

$A_{123} \,\%\, Q_0$	$\cdots$	$A_{456} \,\%\, Q_0$
$\vdots$	$\ddots$	$\vdots$
$A_{123} \,\%\, Q_{L-1}$	$\cdots$	$A_{456} \,\%\, Q_{L-1}$

Element-wise operations (the rest)

- Matrix-matrix or matrix-constant addition, subtraction, Hadamard product, …

$$a * b = \begin{array}{|c|c|c|} \hline (A_0 * B_0) \,\%\, Q_0 & \cdots & (A_{N-1} * B_{N-1}) \,\%\, Q_0 \\ \hline \vdots & \ddots & \vdots \\ \hline (A_0 * B_0) \,\%\, Q_{L-1} & \cdots & (A_{N-1} * B_{N-1}) \,\%\, Q_{L-1} \\ \hline \end{array}$$

Cryptographic Construction of CKKS

RLWE “Game”

Step 1
Alice prepares three polynomials

Polynomial	Name	Coefficient sampled from	Coeff size
a	-	Uniform random over $\mathbb{Z}_Q$	Huge
s	Secret	Uniform random over $\{-1,0,1\}$	Small
e	Error / Noise	Gaussian distribution $\mathcal{N}(0,\sigma)$	Small

Then, Alice computes

$(a, b = a \cdot s + e)$

Alice

Step 2
Alice sends either of these to Bob:
① (a, b)
② Completely random pair of polynomials

Bob

Step 3
Bob tries to guess whether Alice sent ① or ②

RLWE assumption
It is very hard to distinguish ① & ②

RLWE Encryption "Game"

Step 1
Alice prepares three polynomials

Polynomial	Name	Coefficient sampled from	Coeff size
a	-	Uniform random over $\mathbb{Z}_Q$	Huge
s	Secret	Uniform random over $\{-1,0,1\}$	Small
e	Error / Noise	Gaussian distribution $\mathcal{N}(0,\sigma)$	Small

Then, Alice encrypts a polynomial v by computing

$$(a, b = a \cdot s + e + v)$$

Alice

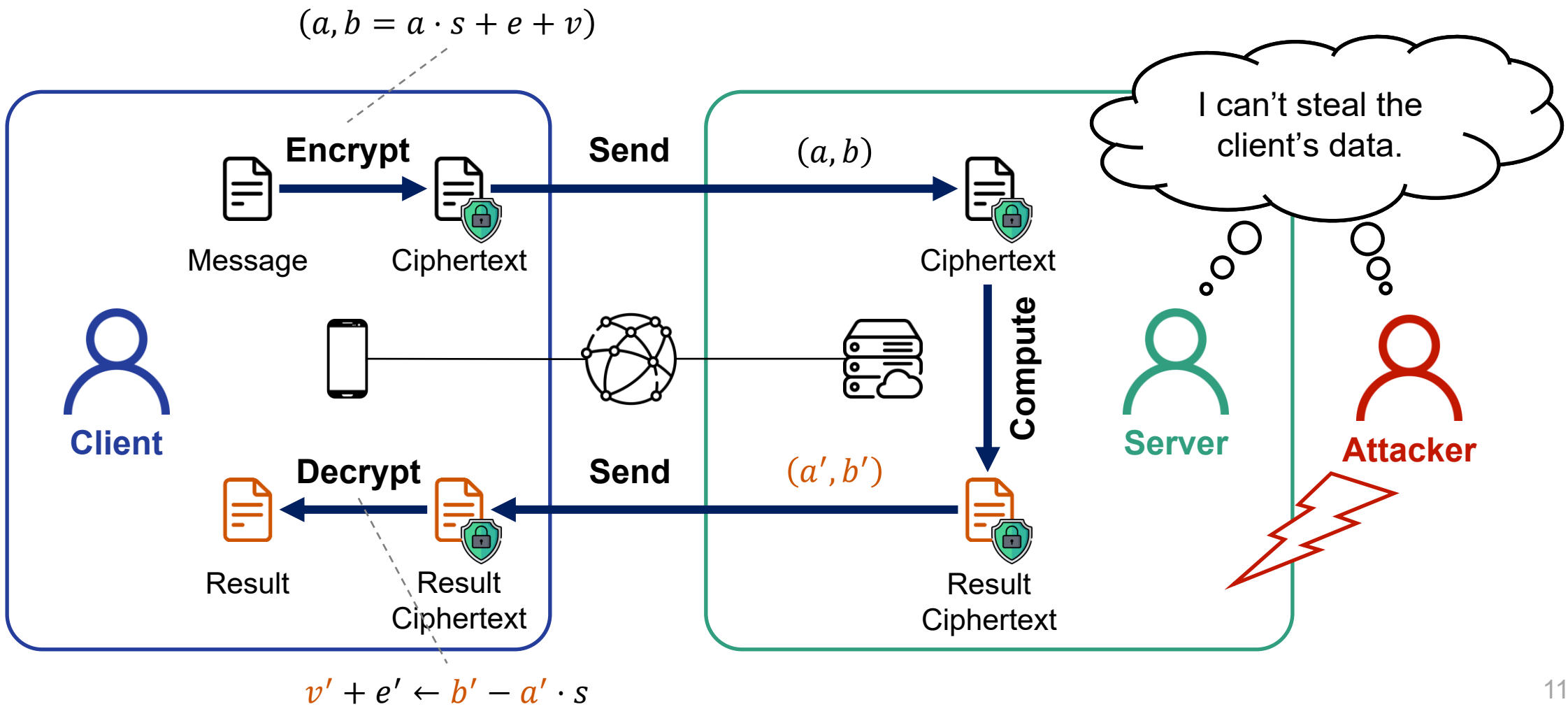
Step 2
Alice sends either of these to Bob:
① (a, b)
② Completely random pair of polynomials

Bob

Step 3
Bob tries to guess whether Alice sent ① or ②

RLWE assumption
It is still very hard to distinguish ① & ②

Overall HE Process



Function Evaluation with HE

- We evaluate functions on ciphertexts (e.g., F)

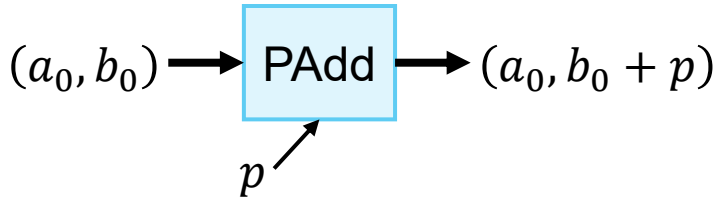
$$f(v_0, v_1, v_2, \cdots) \leftarrow \text{Decrypt}\left(\mathbf{Eval}_f(\text{Encrypt}(v_0), \text{Encrypt}(v_1), \text{Encrypt}(v_2), \cdots)\right)$$

- Basic function evaluations
  - Plaintext–ciphertext addition: $v_0 + p \leftarrow \text{Decrypt}\big(\mathbf{PAdd}(\text{Encrypt}(v_0), p)\big)$
  - Ciphertext–ciphertext addition: $v_0 + v_1 \leftarrow \text{Decrypt}\left(\mathbf{HAdd}\big(\text{Encrypt}(v_0), \text{Encrypt}(v_1)\big)\right)$
  - Plaintext–ciphertext multiplication: $v_0 \cdot p \leftarrow \text{Decrypt}\big(\mathbf{PMult}(\text{Encrypt}(v_0), p)\big)$
  - Ciphertext–ciphertext multiplication: $v_0 \cdot v_1 \leftarrow \text{Decrypt}\left(\mathbf{HMult}\big(\text{Encrypt}(v_0), \text{Encrypt}(v_1)\big)\right)$
  - …

PAdd: Plaintext–Ciphertext Addition

- $v_0 + p \leftarrow \text{Decrypt}\big(\mathbf{PAdd}(\text{Encrypt}(v_0), p)\big)$

- $\text{Encrypt}(v_0) = (a_0, b_0)$
  - $b_0 - a_0 \cdot s = v_0 + e_0$ for client's secret s and small e_0



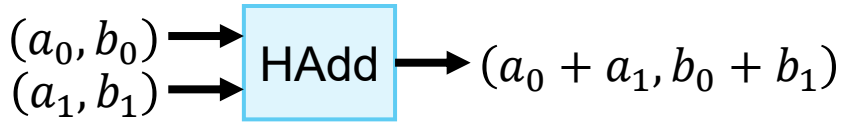
- $\text{Decrypt}\big((a_0, b_0 + p)\big) = b_0 + p - a_0 \cdot s = (b_0 - a_0 \cdot s) + p = v_0 + p + e_0$

Same error

HAdd: Ciphertext–Ciphertext Addition

- $v_0 + v_1 \leftarrow \text{Decrypt}\left(\mathbf{HAdd}\left(\text{Encrypt}(v_0), \text{Encrypt}(v_1)\right)\right)$

- $\text{Encrypt}(v_0) = (a_0, b_0)$ & $\text{Encrypt}(v_1) = (a_1, b_1)$
  - $b_i - a_i \cdot s = v_i + e_i$ for client's secret s and small e_0



- $\text{Decrypt}\left((a_0 + a_1, b_0 + b_1)\right) = b_0 + b_1 - (a_0 + a_1) \cdot s = (b_0 - a_0 \cdot s) + (b_1 - a_1 \cdot s)$
$= (v_0 + e_0) + (v_1 + e_1) = v_0 + v_1 + e_0 + e_1$

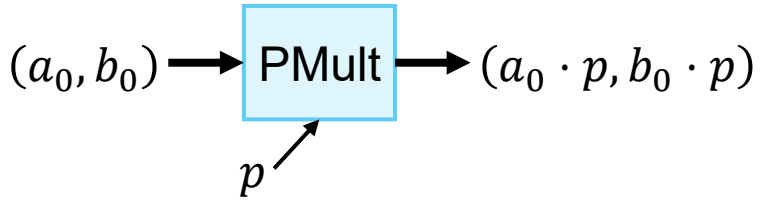
Additive error increase

PMult: Plaintext–Ciphertext Multiplication

- $v_0 \cdot p \leftarrow \text{Decrypt}\big(\textbf{PMult}(\text{Encrypt}(v_0), p)\big)$

  Encrypted evaluation of polynomial multiplication is NOT very useful

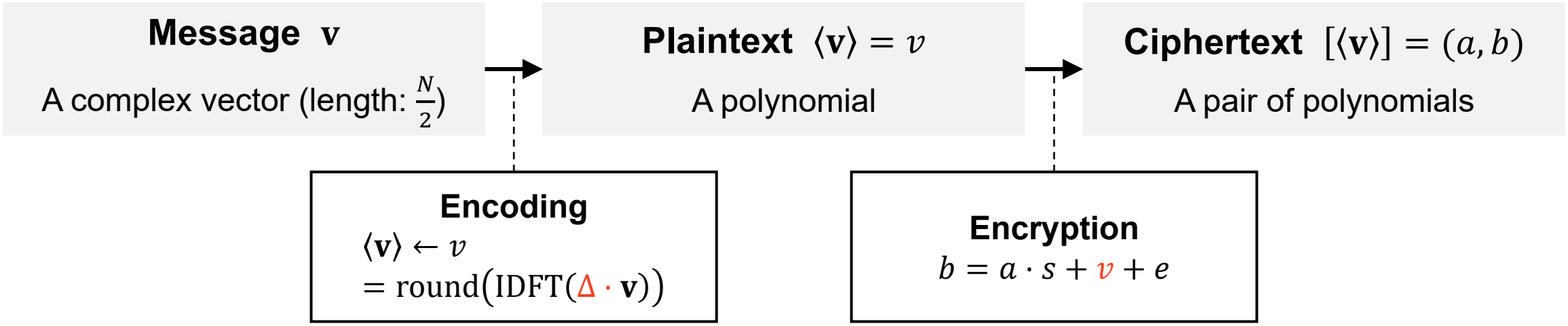
- $\text{Encrypt}(v_0) = (a_0, b_0)$
  - $b_0 - a_0 \cdot s = v_0 + e_0$ for client's secret s and small e_0



- $\text{Decrypt}\big((a_0 \cdot p, b_0 \cdot p)\big) = b_0 \cdot p - a_0 \cdot p \cdot s = (b_0 - a_0 \cdot s) \cdot p = v_0 \cdot p + e_0 \cdot p$

  Multiplicative error increase

CKKS Encoding

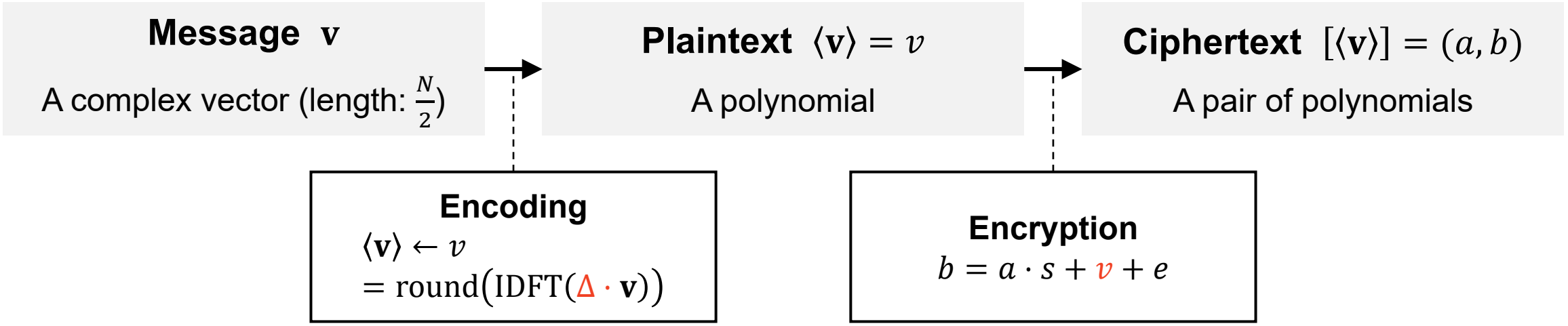


- Polynomial mult = convolution ⇄ element-wise mult between vectors

Fourier transform

Inverse Fourier transform

CKKS Encoding

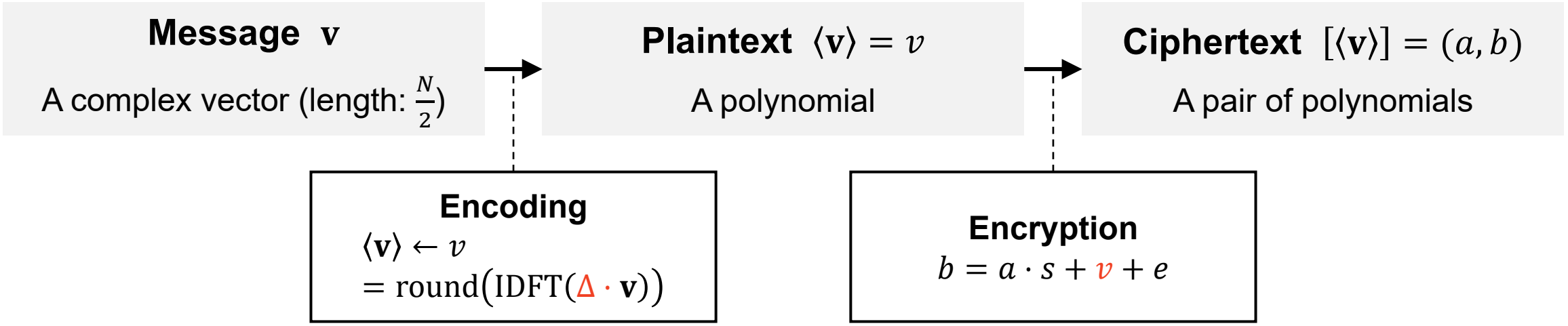


Element-wise mult between complex vector messages
$\mathbf{u} \odot \mathbf{v}$

Polynomial mult between plaintexts
$u \cdot v$

Encrypted computation of polynomial mult

CKKS Encoding



Scale Δ is multiplied to the vector during encoding
(e.g., $\Delta = 2^{40}$)

Element-wise mult between complex vector messages
$\Delta \cdot (\mathbf{u} \odot \mathbf{v})$

⟷

Polynomial mult between plaintexts
$u \cdot v$

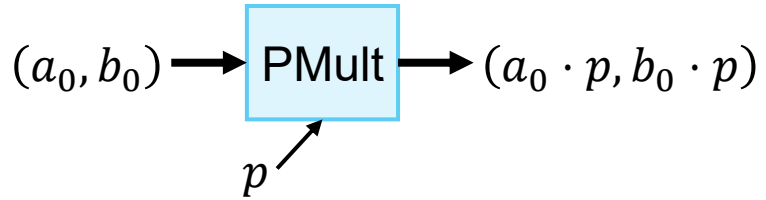
⟷

Encrypted computation of polynomial mult

Multiplication will introduce additional Δ term to the message
(i.e., scale = Δ^2)

PMult: Plaintext–Ciphertext Multiplication

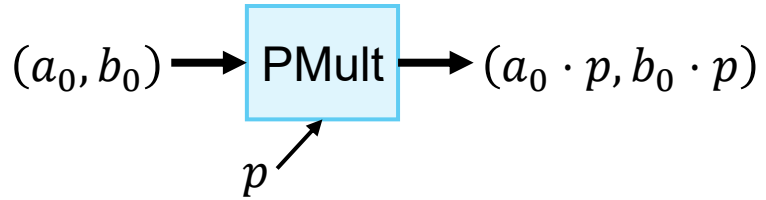
- $v_0 \cdot p \leftarrow \mathrm{Decrypt}\big(\mathbf{PMult}([\langle \mathbf{v_0} \rangle], \langle \mathbf{p} \rangle)\big)$
  - $\langle \mathbf{p} \rangle = p$: Plaintext encoding a vector $\mathbf{p}$
  - $[\langle \mathbf{v_0} \rangle]$: Ciphertext encrypting $\langle \mathbf{v_0} \rangle = v_0$, which is a plaintext encoding a vector $\mathbf{v_0}$
- $[\langle \mathbf{v_0} \rangle] = (a_0, b_0)$
  - $b_0 - a_0 \cdot s = v_0 + e_0$ for client's secret s and small e_0



- $\mathrm{Decrypt}\big((a_0 \cdot p, b_0 \cdot p)\big) = b_0 \cdot p - a_0 \cdot p \cdot s = (b_0 - a_0 \cdot s) \cdot p = v_0 \cdot p + e_0 \cdot p$
  $= \langle \mathbf{v_0} \rangle \cdot \langle \mathbf{p} \rangle + e_0 \cdot p$
  $\simeq \mathrm{IDFT}(\Delta \cdot \mathbf{v_0}) \cdot \mathrm{IDFT}(\Delta \cdot \mathbf{p}) + e_0 \cdot p$
  $= \mathrm{IDFT}\big(\Delta^2 \cdot (\mathbf{v_0} \odot \mathbf{p})\big) + e_0 \cdot p$
  $\simeq \Delta \cdot \langle \mathbf{v_0} \odot \mathbf{p} \rangle + e_0 \cdot p$

PMult: Plaintext–Ciphertext Multiplication

- $v_0 \cdot p \leftarrow \mathrm{Decrypt}\big(\mathbf{PMult}([\langle \mathbf{v_0} \rangle], \langle \mathbf{p} \rangle)\big)$
  - $\langle \mathbf{p} \rangle = p$: Plaintext encoding a vector $\mathbf{p}$
  - $[\langle \mathbf{v_0} \rangle]$: Ciphertext encrypting $\langle \mathbf{v_0} \rangle = v_0$, which is a plaintext encoding a vector $\mathbf{v_0}$
- $[\langle \mathbf{v_0} \rangle] = (a_0, b_0)$
  - $b_0 - a_0 \cdot s = v_0 + e_0$ for client's secret s and small e_0



- $\mathrm{Decrypt}\big((a_0 \cdot p, b_0 \cdot p)\big) = b_0 \cdot p - a_0 \cdot p \cdot s = (b_0 - a_0 \cdot s) \cdot p = v_0 \cdot p + e_0 \cdot p$
  $= \langle \mathbf{v_0} \rangle \cdot \langle \mathbf{p} \rangle + e_0 \cdot p$
  $\simeq \mathrm{IDFT}(\Delta \cdot \mathbf{v_0}) \cdot \mathrm{IDFT}(\Delta \cdot \mathbf{p}) + e_0 \cdot p$
  $= \mathrm{IDFT}\big(\Delta^2 \cdot (\mathbf{v_0} \odot \mathbf{p})\big) + e_0 \cdot p$
  $\simeq \Delta \cdot \langle \mathbf{v_0} \odot \mathbf{p} \rangle + e_0 \cdot p$

Keep error sizes small enough

$\rightarrow \langle \mathbf{v_0} \odot \mathbf{p} \rangle + \frac{1}{\Delta} e_0 \cdot p$

If we can somehow divide the results by Δ

Ciphertext Rescaling

- Getting rid of the additional Δ term with rescaling

$$(a, b) \xrightarrow{\textbf{Rescaling}} \left(\frac{1}{\Delta}a, \frac{1}{\Delta}b\right)$$

- However, we cannot divide the polynomial easily in the RNS form
  - Unless a and b are multiples of Δ
    → in this case, we can multiply modular multiplicative inverse of Δ for division
  - Example

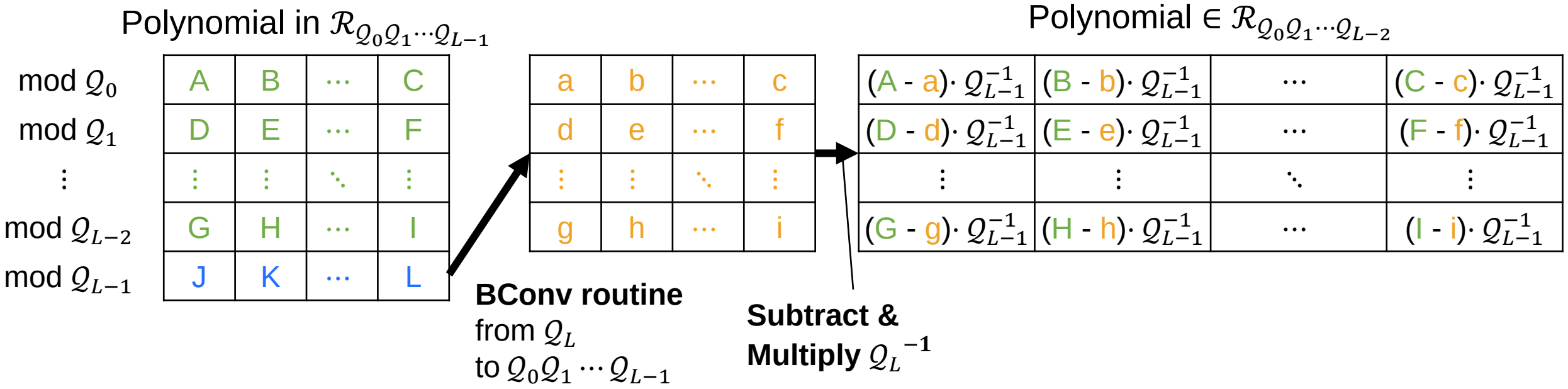
If $\Delta = 4$, $\boldsymbol{\Delta^{-1} = 8 \bmod 31}$ because $4 \cdot 8 = 1 \bmod 31$

In $\mathbb{Z}_{31}$, 16 can be easily divided by Δ → $16 \cdot \Delta^{-1} = 16 \cdot 8 = 128 = 4 \bmod 31$

But, 15 cannot be easily divided by Δ → $15 \cdot \Delta^{-1} = 15 \cdot 8 = 120 = 27 \bmod 31$

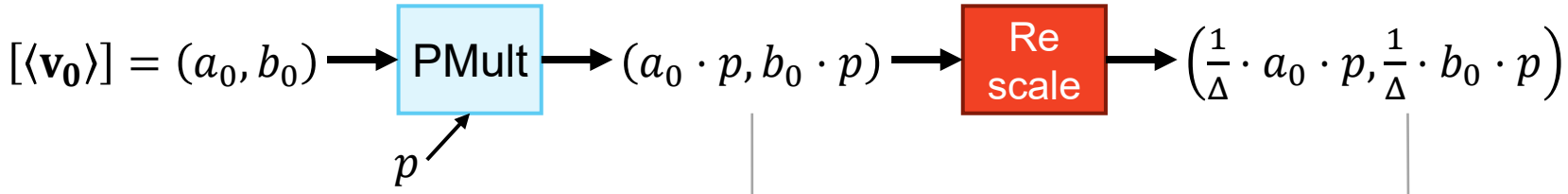
Rescaling of a Polynomial

- Rescaling = **approximate division by the last sub-modulus**
  - Get rid of the mod-$\mathcal{Q}_{L-1}$ residue by subtraction → Make the polynomial become a multiple of $\mathcal{Q}_{L-1}$
  - Multiply $\mathcal{Q}_{L-1}^{-1}$ to each limb = Division by $\mathcal{Q}_{L-1}$
  - **Last limb is removed during rescaling**



By making every $\mathcal{Q}_i \simeq \Delta$,
we can perform **rescaling by Δ multiple times**

PMult & Rescaling



Decrypts to

$$\Delta \cdot \langle \mathbf{v_0} \odot \mathbf{p} \rangle + e_0 \cdot p$$

Decrypts to

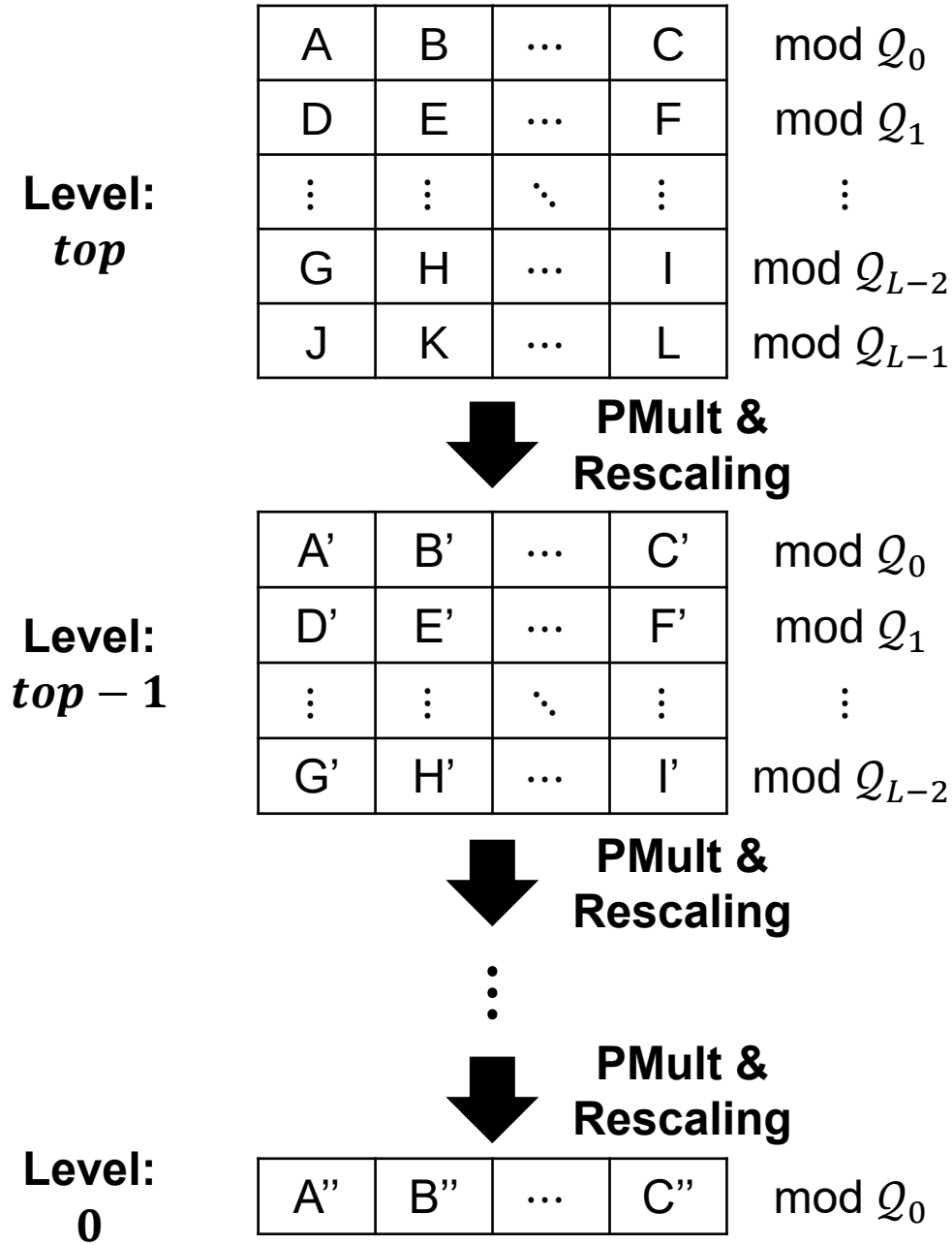
$$\langle \mathbf{v_0} \odot \mathbf{p} \rangle + \frac{1}{\Delta} \cdot e_0 \cdot p$$

This new error term should be sufficiently small

Rescaling and Level

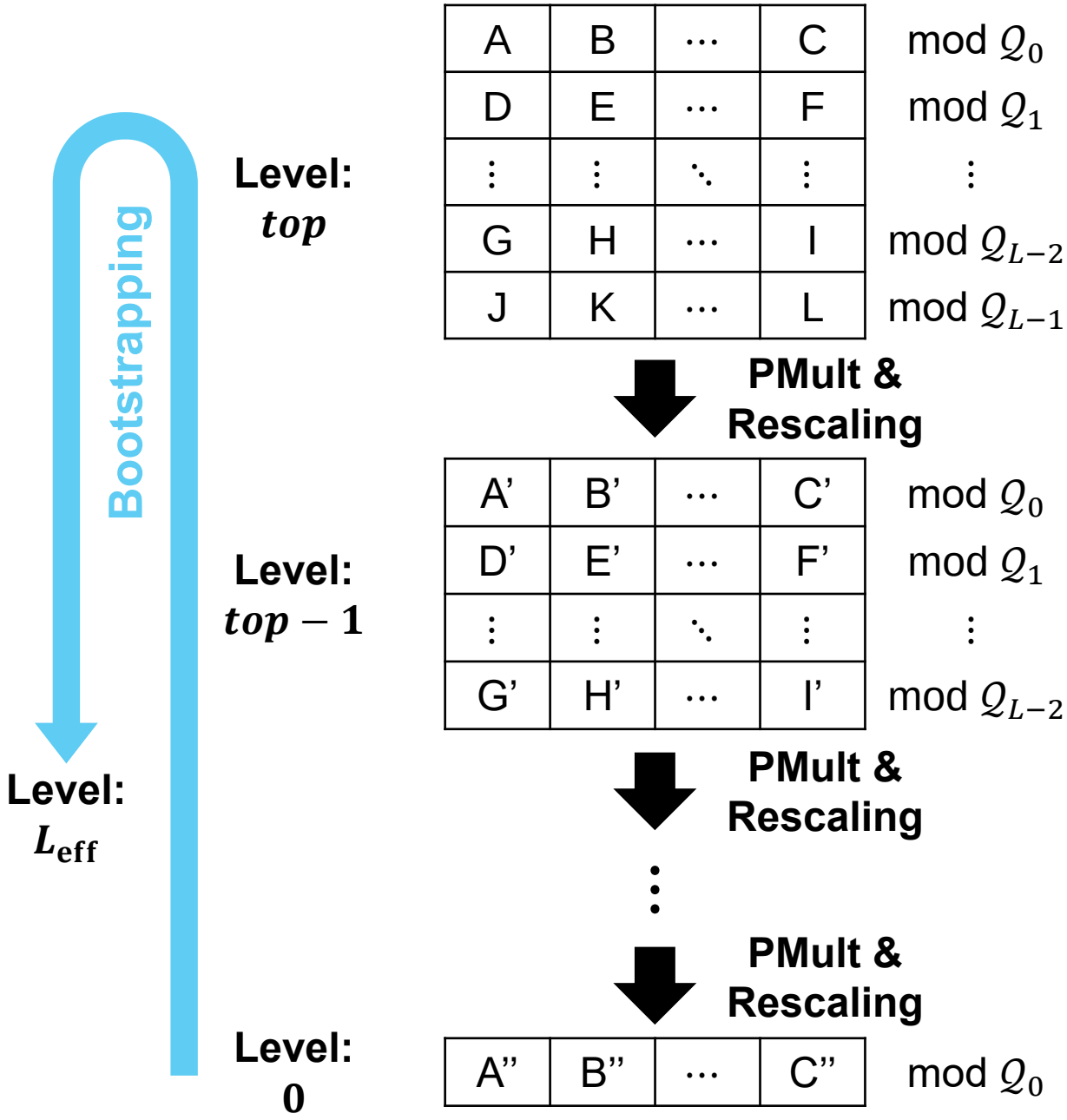
- Repeated rescaling gradually reduces the number of remaining limbs in each polynomial

- We associate each with a level from top to 0



Bootstrapping

- Bootstrapping restores the level from $\mathbf{0}$
- However, bootstrapping in itself consumes some levels
- Final level after bootstrapping = effective level (L_{eff})



HMult: Ciphertext–Ciphertext Multiplication

$$[\langle \mathbf{v_0} \rangle] = (a_0, b_0) \qquad [\langle \mathbf{v_1} \rangle] = (a_1, b_1)$$
$$b_0 - a_0 \cdot s = v_0 + e_0 \qquad b_1 - a_1 \cdot s = v_1 + e_1$$

$$(b_0 - a_0 \cdot s) \cdot (b_1 - a_1 \cdot s) = (v_0 + e_0) \cdot (v_1 + e_1)$$

$$b_0 \cdot b_1 - (b_0 \cdot a_1 + b_1 \cdot a_0) \cdot s + (a_0 \cdot a_1) \cdot s^2 \qquad v_0 \cdot v_1 + e_0 \cdot v_1 + e_1 \cdot v_0 + e_0 \cdot e_1$$

The server can compute each of these

These error terms can be adequately controlled by rescaling

HMult: Ciphertext–Ciphertext Multiplication

$$[\langle \mathbf{v_0} \rangle] = (a_0, b_0) \qquad [\langle \mathbf{v_1} \rangle] = (a_1, b_1)$$
$$b_0 - a_0 \cdot s = v_0 + e_0 \qquad b_1 - a_1 \cdot s = v_1 + e_1$$

Want: something that can be decrypted like this:

$$b_0 \cdot b_1 - (b_0 \cdot a_1 + b_1 \cdot a_0) \cdot s + (a_0 \cdot a_1) \cdot s^2$$

A new ciphertext
$(b_0 \cdot a_1 + b_1 \cdot a_0, b_0 \cdot b_1)$

+

Multiply $(a_0 \cdot a_1)$ with **a special ciphertext**:
$\mathrm{Encrypt}(s^2)$

$(a_0 \cdot a_1)$ **is much larger than plaintexts multiplied at PMult**

$\because Q \gg \Delta$

Key-Switching

- The special $\mathrm{Encrypt}(s^2)$ is called an **evaluation key (**evk**)**
  - An evk is composed of polynomials with a larger modulus PQ
  - With RNS, each polynomial has $(K + L)$ limbs with $P = \prod_{i=0}^{K-1} \mathcal{P}_i$ & $Q = \prod_{i=0}^{L-1} \mathcal{Q}_i$
  - There are also other types of evaluation keys

- **Key-switching** multiplies an evk with a large polynomial $\in \mathcal{R}_Q$

Basic Key-Switching

- evk is composed of two polynomials $\mathrm{evk} = (a, b)$, which satisfies
$$b - a \cdot s = P \cdot s^2 + e$$
- We want to multiply a polynomial d
  - For HMult, $d = a_0 \cdot a_1$

$d \in \mathcal{R}_Q$

A	B	⋯	C	mod $\mathcal{Q}_0$
D	E	⋯	F	mod $\mathcal{Q}_1$
⋮	⋮	⋱	⋮	⋮
G	H	⋯	I	mod $\mathcal{Q}_{L-1}$

$a \in \mathcal{R}_{PQ}$

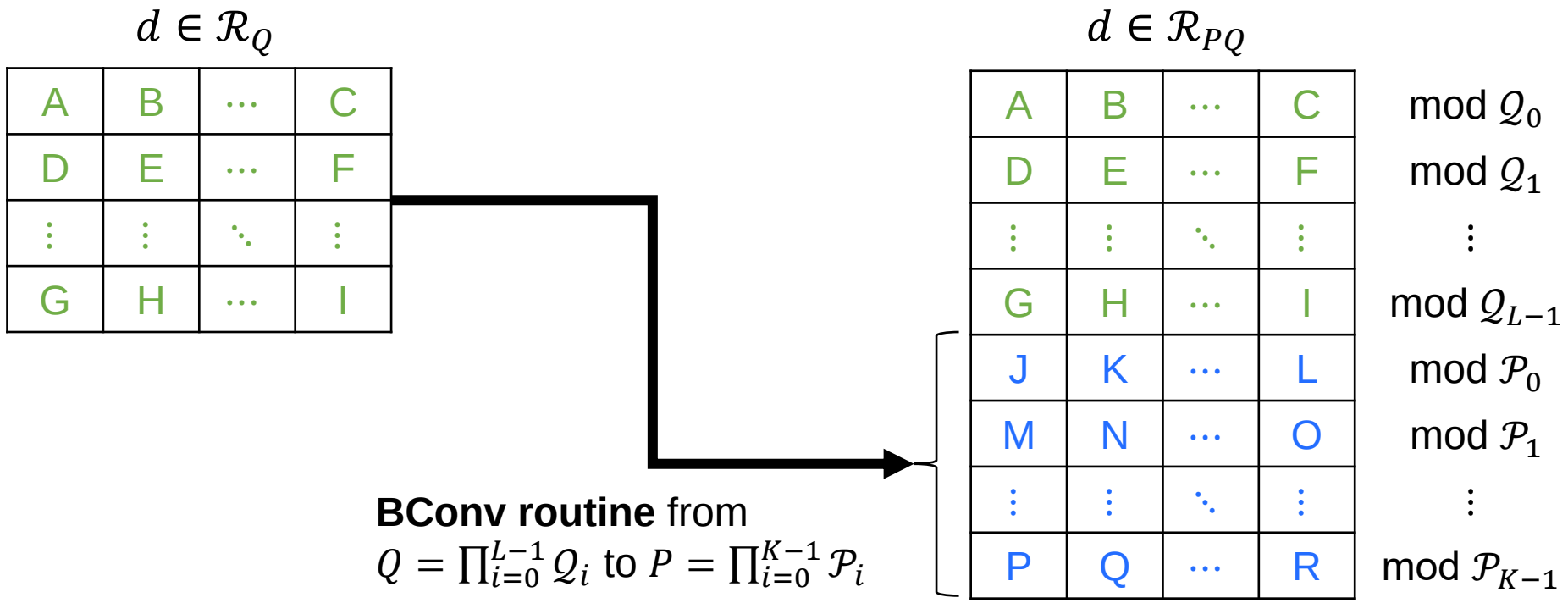
A'	B'	⋯	C'	mod $\mathcal{Q}_0$
D'	E'	⋯	F'	mod $\mathcal{Q}_1$
⋮	⋮	⋱	⋮	⋮
G'	H'	⋯	I'	mod $\mathcal{Q}_{L-1}$
J'	K'	⋯	L'	mod $\mathcal{P}_0$
M'	N'	⋯	O'	mod $\mathcal{P}_1$
⋮	⋮	⋱	⋮	⋮
P'	Q'	⋯	R'	mod $\mathcal{P}_{K-1}$

$b \in \mathcal{R}_{PQ}$

A''	B''	⋯	C''	mod $\mathcal{Q}_0$
D''	E''	⋯	F''	mod $\mathcal{Q}_1$
⋮	⋮	⋱	⋮	⋮
G''	H''	⋯	I''	mod $\mathcal{Q}_{L-1}$
J''	K''	⋯	L''	mod $\mathcal{P}_0$
M''	N''	⋯	O''	mod $\mathcal{P}_1$
⋮	⋮	⋱	⋮	⋮
P''	Q''	⋯	R''	mod $\mathcal{P}_{K-1}$

ModUp: Increasing the Modulus of d

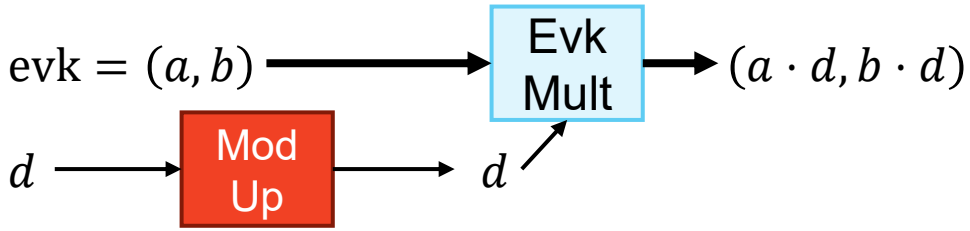
- ModUp is used to change the modulus from Q to PQ
  - Q part is kept as is
  - P part is generated by **BConv routine** and concatenated to the Q part



Basic Key-Switching

- evk is composed of two polynomials $\mathrm{evk} = (a, b)$, which satisfies
$$b - a \cdot s = P \cdot s^2 + e$$
- We want to multiply a polynomial d
  - For HMult, $d = a_0 \cdot a_1$

- After performing ModUp to d, multiply it with evk (just as in PMult)

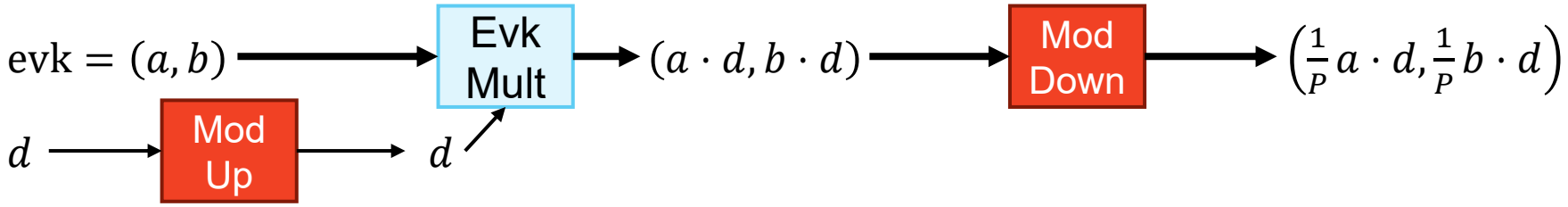


- $\mathrm{Decrypt}\big((a \cdot d, b \cdot d)\big) = b \cdot d - (a \cdot d) \cdot s = (b - a \cdot s) \cdot d = P \cdot d \cdot s^2 + e \cdot d$

Basic Key-Switching

- evk is composed of two polynomials $\mathrm{evk} = (a, b)$, which satisfies
$$b - a \cdot s = P \cdot s^2 + e$$
- We want to multiply a polynomial d
  - For HMult, $d = a_0 \cdot a_1$

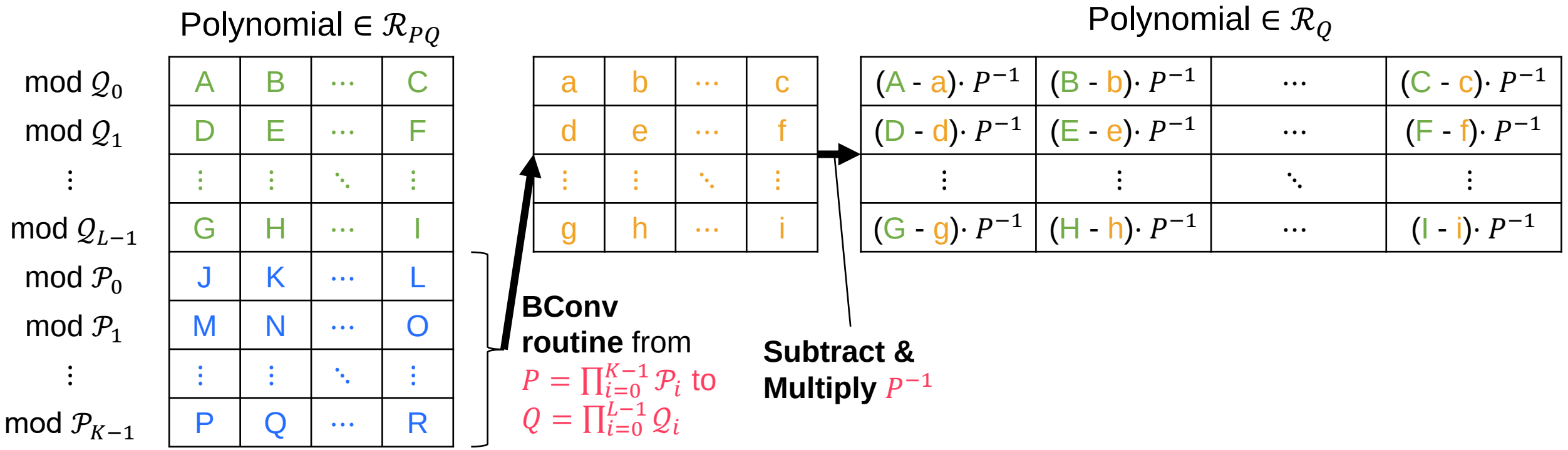
- After performing ModUp to d, multiply it with evk (just as in PMult)



- $\mathrm{Decrypt}\big((a \cdot d, b \cdot d)\big) = b \cdot d - (a \cdot d) \cdot s = (b - a \cdot s) \cdot d = P \cdot d \cdot s^2 + e \cdot d$
- **Division by P results in** $d \cdot s^2 + \frac{1}{P} e \cdot d$ **This new error term should be sufficiently small if $P > Q$**

ModDown

- ModDown = **approximate division by** P
  - Get rid of the mod-P residue by subtraction → Make the polynomial become a multiple of P
  - Multiply P^{-1} to each limb = Division by P



HMult with Key-Switching

$$[\langle \mathbf{v_0} \rangle] = (a_0, b_0) \qquad [\langle \mathbf{v_1} \rangle] = (a_1, b_1)$$
$$b_0 - a_0 \cdot s = v_0 + e_0 \qquad b_1 - a_1 \cdot s = v_1 + e_1$$

Want: something that can be decrypted like this:

$$\underbrace{b_0 \cdot b_1 - (b_0 \cdot a_1 + b_1 \cdot a_0) \cdot s}_{\text{A new ciphertext}} + \underbrace{(a_0 \cdot a_1) \cdot s^2}_{\text{Key-switch } (a_0 \cdot a_1) \text{ with}}$$

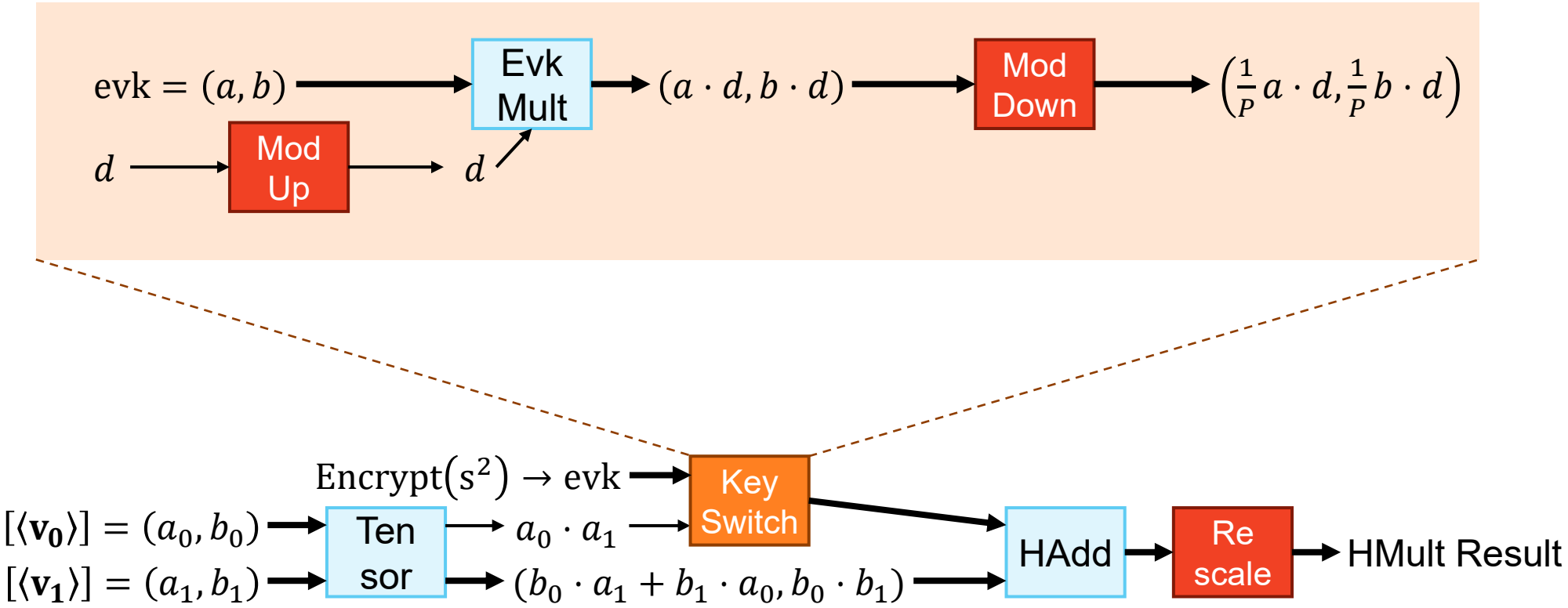
$$(b_0 \cdot a_1 + b_1 \cdot a_0, b_0 \cdot b_1) \qquad \text{evk} \leftarrow \text{Encrypt}(s^2)$$

End result: ciphertext that decrypt to

$$\Delta \cdot \langle \mathbf{v_0} \odot \mathbf{v_1} \rangle + (\boldsymbol{error\ terms} \ldots)$$

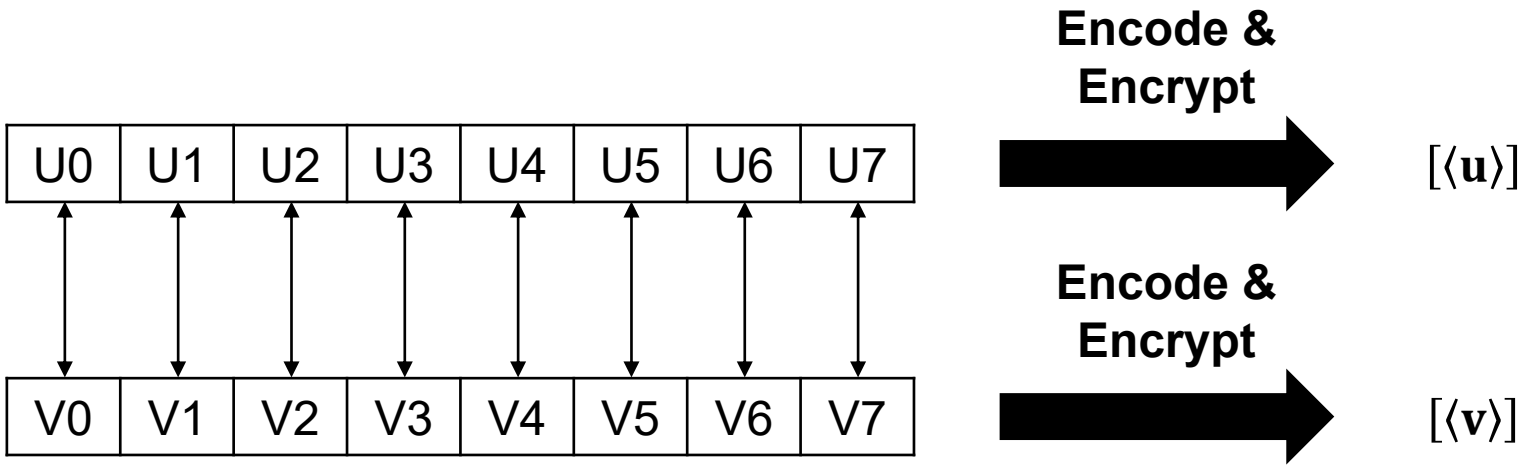
→ Needs rescaling

HMult with Key-Switching



Computation Between Different Slots

- **Slot:** Each vector that is encoded has $N/2$ **slots** to put (complex) message elements
- PAdd, PMult, CAdd, and CMult all perform same-slot computations



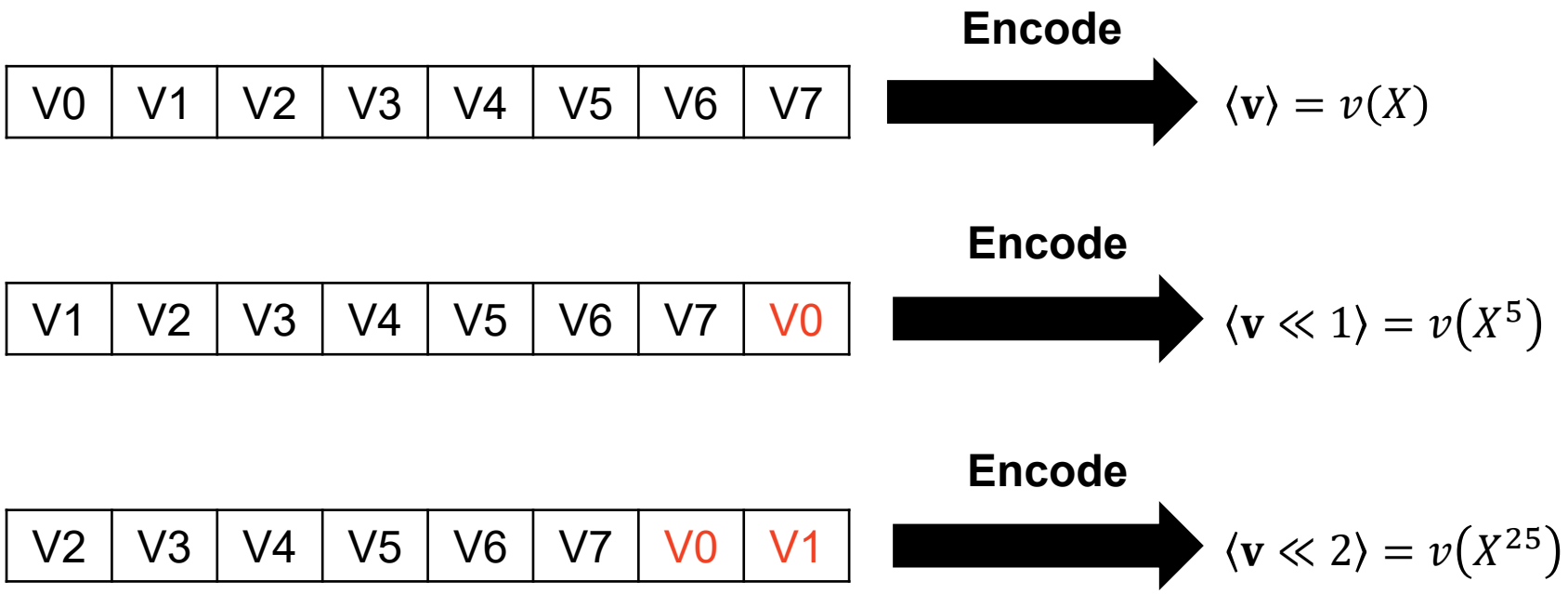
- How can we compute results between different slots?

HRot: Ciphertext Rotation

- Due to the special encoding structure, we can compute **automorphism** to rotate messages encoded in plaintexts

$$\langle \mathbf{v} \ll R \rangle = v\left(X^{5^R}\right)$$

Cyclic rotation by R **Automorphism**



HRot: Ciphertext Rotation

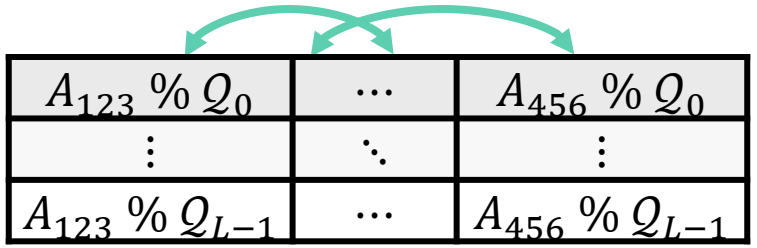
- Due to the special encoding structure, we can compute **automorphism** to rotate messages encoded in plaintexts

$$\langle \mathbf{v} \ll R \rangle = v\left(X^{5^R}\right)$$

Cyclic rotation by R — **Automorphism**

Automorphism

- Shuffling the order of columns according to an index mapping



HRot: Ciphertext Rotation

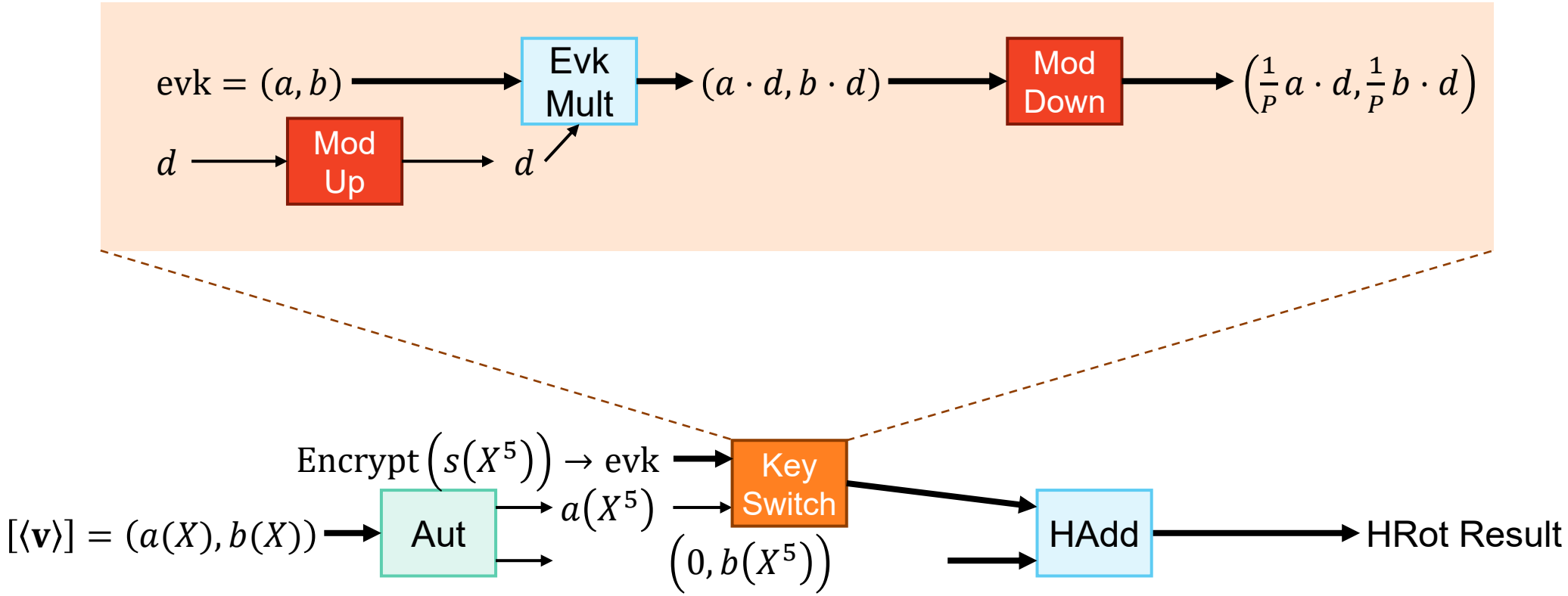
$$[\langle \mathbf{v} \rangle] = (a(X), b(X))$$
$$b(X) - a(X) \cdot s(X) = v(X) + e(X)$$

HMult had s^2 term to remove,
HRot has $s(X^5)$

$$b(X^5) - a(X^5) \cdot s(X^5) = v(X^5) + e(X^5)$$

The server computes automorphism on both a & b

HMult with Key-Switching



Very similar to HMult, except that
1. Automorphism instead of tensor
2. Rescaling NOT required

Inter-Slot Computation Example: Accumulation

V0	V1	V2	V3	V4	V5	V6	V7
V1	V2	V3	V4	V5	V6	V7	**V0**
V0+ V1	V1+ V2	V2+ V3	V3+ V4	V4+ V5	V5+ V6	V6+ V7	V7+ V0
V2+ V3	V3+ V4	V4+ V5	V5+ V6	V6+ V7	V7+ V0	**V0+ V1**	**V1+ V2**
V0+ V1+ V2+ V3	V1+ V2+ V3+ V4	V2+ V3+ V4+ V5	V3+ V4+ V5+ V6	V4+ V5+ V6+ V7	V7+ V0+ V1+ V2	V6+ V7+ V0+ V1	V7+ V0+ V1+ V2
V4+ V5+ V6+ V7	V7+ V0+ V1+ V2	V6+ V7+ V0+ V1	V7+ V0+ V1+ V2	**V0+ V1+ V2+ V3**	**V1+ V2+ V3+ V4**	**V2+ V3+ V4+ V5**	**V3+ V4+ V5+ V6**
Sum	Sum	Sum	Sum	Sum	Sum	Sum	Sum

$[\langle \mathbf{v} \rangle]$

Want: Sum = V0 + V1 + … + V7

$[\langle \mathbf{v} \ll 1 \rangle]$

$[\langle \mathbf{t_0} \rangle] = [\langle \mathbf{v} \rangle] + \left[[\langle \mathbf{v} \ll 1 \rangle]\right]$

$[\langle \mathbf{t_0} \ll 2 \rangle]$

$[\langle \mathbf{t_1} \rangle] = [\langle \mathbf{t_0} \rangle] + [\langle \mathbf{t_0} \ll 2 \rangle]$

$[\langle \mathbf{t_1} \ll 4 \rangle]$

$[\langle \mathbf{t_1} \rangle] + [\langle \mathbf{t_1} \ll 4 \rangle]$

Developing a CKKS Application

- Tools we have
  - Vector addition
  - Element-wise vector multiplication
  - Cyclic vector rotation

- Tools we do not have
  - Comparison
  - Branches
  - …

Finding efficient methods to evaluate various functions with only **additions, multiplications,** and **rotations** is crucial for CKKS application development