Arithmetic operations on encrypted integers

陈经纬

September 16, 2018 @ Guangzhou

Main references on which this talk is based

- J. Crawford, C. Gentry, S. Halevi, D. Platt, and V. Shoup.
  Doing real work with FHE: The case of logistic regression.
  Cryptology ePrint Archive, Report 2018/202.
- C. Xu, J. Chen, W. Wu, and Y. Feng.
  Homomorphically encrypted arithmetic operations over the integer ring. In: Proc. ISPEC'16.
- Y. Chen, and G. Gong
  Integer arithmetic over ciphertext and homomorphic data aggregation. In: Proc. CNS'15.
- S. Halevi and V. Shoup.
  HElib – An implementation of homomorphic encryption.
  Available at github.com/shaih/HElib/

Roadmap

1. Background
2. Arithmetic algorithms
3. Performance

Roadmap

1 Background

2 Arithmetic algorithms

3 Performance

Fully homomorphic encryption (FHE)

FHE allows "arbitrary" computation to be done on encrypted data.

Figure: The Damsel of the Sanct Grael by Dante Gabriel Rossetti (wiki)

Fully homomorphic encryption (FHE)

A public key encryption scheme consists of

- $\texttt{KeyGen}$: $(\mathrm{sk}, \mathrm{pk}) \leftarrow \texttt{KeyGen}(1^k)$,
- $\texttt{Enc}$: $c \leftarrow \texttt{Enc}(\mathrm{pk}, x)$ for $x \in \mathscr{P} = \{0,1\}^*$,
- $\texttt{Dec}$: $x \leftarrow \texttt{Dec}(\mathrm{sk}, c)$ for $c \in \mathscr{C}$.

Fully homomorphic encryption (FHE)

Definition [Brakerski'18, ECCC report no. 125]

Let $\mathscr{F}$ be a set of function in $\{0,1\}^* \to \{0,1\}$. A public key scheme is *$\mathscr{F}$-homomorphic* if there exists an evaluation algorithm $\mathtt{Eval}$ s.t.

$$\forall f \in \mathscr{F},\ \forall x \in \{0,1\}^*,\ \mathtt{Dec}(\mathtt{sk}, \mathtt{Eval}(f, \mathtt{Enc}(\mathtt{pk}, x))) = f(x).$$

Fully homomorphic encryption (FHE)

Definition [Brakerski'18, ECCC report no. 125]

Let $\mathscr{F}$ be a set of function in $\{0,1\}^* \to \{0,1\}$. A public key scheme is *$\mathscr{F}$-homomorphic* if there exists an evaluation algorithm $\mathtt{Eval}$ s.t.

$$\forall f \in \mathscr{F},\ \forall x \in \{0,1\}^*,\ \mathtt{Dec}(\mathtt{sk}, \mathtt{Eval}(f, \mathtt{Enc}(\mathtt{pk}, x))) = f(x).$$

A *FHE* is a homomorphic encryption where $\mathscr{F}$ is the set of all functions.

Fully homomorphic encryption (FHE)

Definition [Brakerski'18, ECCC report no. 125]

Let $\mathscr{F}$ be a set of function in $\{0,1\}^* \to \{0,1\}$. A public key scheme is *$\mathscr{F}$-homomorphic* if there exists an evaluation algorithm $\texttt{Eval}$ s.t.

$$\forall f \in \mathscr{F},\ \forall x \in \{0,1\}^*,\ \texttt{Dec}(\texttt{sk}, \texttt{Eval}(f, \texttt{Enc}(\texttt{pk}, x))) = f(x).$$

A *FHE* is a homomorphic encryption where $\mathscr{F}$ is the set of all functions.

- It is most common to use the boolean circuit model to represent f.

Fully homomorphic encryption (FHE)

Definition [Brakerski'18, ECCC report no. 125]

Let $\mathscr{F}$ be a set of function in $\{0,1\}^* \to \{0,1\}$. A public key scheme is *$\mathscr{F}$-homomorphic* if there exists an evaluation algorithm $\texttt{Eval}$ s.t.

$$\forall f \in \mathscr{F},\ \forall x \in \{0,1\}^*,\ \texttt{Dec}(\texttt{sk}, \texttt{Eval}(f, \texttt{Enc}(\texttt{pk}, x))) = f(x).$$

A *FHE* is a homomorphic encryption where $\mathscr{F}$ is the set of all functions.

- It is most common to use the boolean circuit model to represent f.

Existing FHE schemes

- 1st generation: [Gentry'09], ...
- 2nd generation: [Brakerski, Gentry, Vaikuntanathan'11], ...
- 3rd generation: [Gentry, Sahai, Waters'13], ...

Applications of FHE – HomomorphicEncryption.org

Domain	**Genomics**	**Health**	**National Security**	**Education**
Topic	Match Maker	Billing & Reporting	Municipal Service	School Dropouts
Data Owner	Medical Institutions	Clinic	Nodes	School, Hospital, Welfare
Latency	Hours	Hours	Quasi-Real Time	Week
Data Volume (size×no.)	DB: $\mathcal{O}(1000 \times 1\,\mathrm{M})$; Query: $\mathcal{O}(1\mathrm{K})$	$\mathcal{O}(10\,\mathrm{M} \times 1\,\mathrm{M})$	$\mathcal{O}(1\,\mathrm{M} \times 1\,\mathrm{M})$	$\mathcal{O}(10\,\mathrm{K} \times 1\,\mathrm{M})$
Data Persistency	Add only	Add only	Add only	Add only
Technical issues	Comparison Sorting Auditing Privacy	Tabulation Linear Algebra	Comparison	Comparison Matrix Analysis
When?	1 year	2–3 years	Now	2–3 years
Why HE?	HIPAA	Cyber Insurance	Privacy	FERPA
Who pays?	Health Insurance	Hospital	Energy Company	DoE

Applications of FHE – HomomorphicEncryption.org

Domain	**Genomics**	**Health**	**National Security**	**Education**
Topic	Match Maker	Billing & Reporting	Municipal Service	School Dropouts
Data Owner	Medical Institutions	Clinic	Nodes	School, Hospital, Welfare
Latency	Hours	Hours	Quasi-Real Time	Week
Data Volume (size×no.)	DB: $\mathcal{O}(1000 \times 1\,\mathrm{M})$; Query: $\mathcal{O}(1\mathrm{K})$	$\mathcal{O}(10\text{ M} \times 1\text{ M})$	$\mathcal{O}(1\text{ M} \times 1\text{ M})$	$\mathcal{O}(10\text{ K} \times 1\text{ M})$
Data Persistency	Add only	Add only	Add only	Add only
Technical issues	Comparison Sorting Auditing Privacy	Tabulation Linear Algebra	Comparison	Comparison Matrix Analysis
When?	1 year	2–3 years	Now	2–3 years
Why HE?	HIPAA	Cyber Insurance	Privacy	FERPA
Who pays?	Health Insurance	Hospital	Energy Company	DoE

Homomorphic arithmetic on integers

- RLWE-based somewhat HE:
  - [Naehrig, Lauter, Vaikuntanathan '11], [Wu & Haven '12] ($p > 2^{128}$), ...
- DGHV with optimizations over $\mathbb{Z}_p$:
  - [Dijk, Gentry, Halevi, Vaikuntanathan '11], [Cheon, Coron, Kim, Lee, Lepoint, Tibouchi, Yun '13], ...
- HElib-based
  - Symmetric ternary coding: [Fu, Cai, Xiang, Sang '18], ...
  - One ciphtertext one integer, SIMD, $p = 2$: [Cheon, Kim, Kim '15], ...

Homomorphic arithmetic on integers

- RLWE-based somewhat HE:
  - [Naehrig, Lauter, Vaikuntanathan '11], [Wu & Haven '12] ($p > 2^{128}$), ...
- DGHV with optimizations over $\mathbb{Z}_p$:
  - [Dijk, Gentry, Halevi, Vaikuntanathan '11], [Cheon, Coron, Kim, Lee, Lepoint, Tibouchi, Yun '13], ...
- HElib-based
  - Symmetric ternary coding: [Fu, Cai, Xiang, Sang '18], ...
  - One ciphtertext one integer, SIMD, $p = 2$: [Cheon, Kim, Kim '15], ...
  - One ciphtertext one bit, $p = 2$: [Chen & Gong '15], ...

Homomorphic arithmetic on integers

- RLWE-based somewhat HE:
  - [Naehrig, Lauter, Vaikuntanathan '11], [Wu & Haven '12] ($p > 2^{128}$), ...
- DGHV with optimizations over $\mathbb{Z}_p$:
  - [Dijk, Gentry, Halevi, Vaikuntanathan '11], [Cheon, Coron, Kim, Lee, Lepoint, Tibouchi, Yun '13], ...
- HElib-based
  - Symmetric ternary coding: [Fu, Cai, Xiang, Sang '18], ...
  - One ciphtertext one integer, SIMD, $p = 2$: [Cheon, Kim, Kim '15], ...
  - One ciphtertext one bit, $p = 2$: [Chen & Gong '15], ...

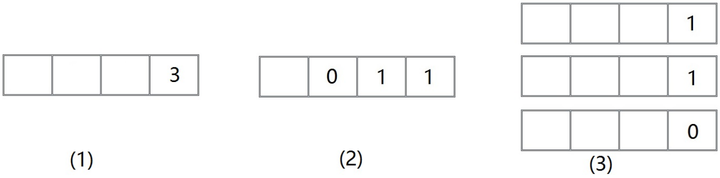

Figure: Encrypted binary integer representation

Homomorphic arithmetic on integers

- RLWE-based somewhat HE:
  - [Naehrig, Lauter, Vaikuntanathan '11], [Wu & Haven '12] ($p > 2^{128}$), ...
- DGHV with optimizations over $\mathbb{Z}_p$:
  - [Dijk, Gentry, Halevi, Vaikuntanathan '11], [Cheon, Coron, Kim, Lee, Lepoint, Tibouchi, Yun '13], ...
- HElib-based
  - Symmetric ternary coding: [Fu, Cai, Xiang, Sang '18], ...
  - One ciphtertext one integer, SIMD, $p = 2$: [Cheon, Kim, Kim '15], ...
  - One ciphtertext one bit, $p = 2$: [Chen & Gong '15], ...

Advantages of $p = 2$

- XOR ($\oplus$) $\longleftrightarrow$ mod 2 addition; AND ($\cdot$) $\longleftrightarrow$ mod 2 multiplication.
- More suitable for bootstrapping.

Advantages of one ciphtertext one bit

- Support element-wise vector arithmetic.

BGV scheme and HElib

BGV scheme

- One of the most efficient FHE schemes, RLWE based;
- Designed for circuits;
- Noise management: modulus switch, key switch;
- Support SIMD operations.

BGV scheme and HElib

BGV scheme

- One of the most efficient FHE schemes, RLWE based;
- Designed for circuits;
- Noise management: modulus switch, key switch;
- Support SIMD operations.

HElib: BGV implementation based on NTL

- "Assembly language for HE";
- Double CRT representation;
- Ciphertext packing techniques (SIMD);
- Support bootstrapping;
- Thread safe.

BGV scheme and HElib

- Plaintext space: $\mathbb{Z}[X]/(\Phi_m(X), p)$.
- Ciphertext space: $\mathbb{Z}[X]/(\Phi_m(X), q)$, $q = p_1 p_2 \cdots p_\ell$, p_i prime.
  - Every ciphertext contains the same number of slots.
    - Each slot has the same size.
  - Each ciphertext is represented as an $\ell \times \phi(m)$ matrix.
- $L = \mathcal{O}(\ell)$ is the circuit level we want to support.
- Given security parameter k, we can decide m from [a]

$$\phi(m) \geq \frac{(L(\log \phi(m) + 23) - 8.5)(k + 110)}{7.2}.$$

- It evaluates L-level circuits with $\mathcal{O}(k \cdot L^3)$ per-gate computation.
  - We should optimize the circuit by reducing L.

[a] https://eprint.iacr.org/2012/099

Roadmap

1. Background
2. Arithmetic algorithms
3. Performance

Addition: Ripple carry adder (RCA) – [ChenGong'15]

- RCA: Add two n-bit numbers in a natural way.

- RCA: Add two n-bit numbers in a natural way.

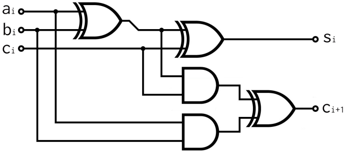

Figure: A 1-bit full adder

$$c_{i+1} = a_i \cdot b_i \oplus c_i \cdot (a_i \oplus b_i),$$
$$s_i = a_i \oplus b_i \oplus c_i.$$

- RCA: Add two n-bit numbers in a natural way.

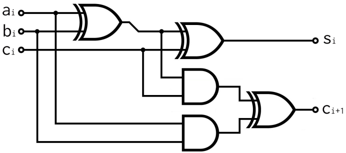

Figure: A 1-bit full adder

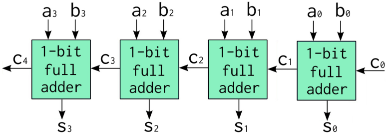

Figure: A 4-bit RCA (wiki)

$$\begin{aligned} c_{i+1} &= a_i \cdot b_i \oplus c_i \cdot (a_i \oplus b_i), \\ s_i &= a_i \oplus b_i \oplus c_i. \end{aligned}$$

- RCA: Add two n-bit numbers in a natural way.

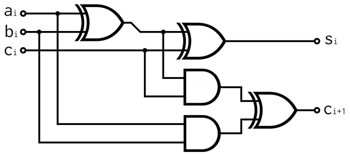

Figure: A 1-bit full adder

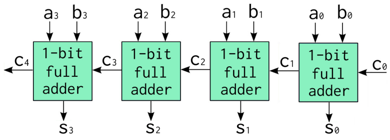

Figure: A 4-bit RCA (wiki)

$$c_{i+1} = a_i \cdot b_i \oplus c_i \cdot (a_i \oplus b_i),$$
$$s_i = a_i \oplus b_i \oplus c_i.$$

- Multiplicative depth: $L = n - 1$.

Addition: Ripple carry adder (RCA) – [ChenGong'15]

- RCA: Add two n-bit numbers in a natural way.

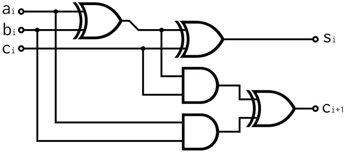

Figure: A 1-bit full adder

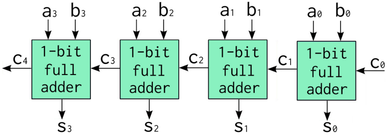

Figure: A 4-bit RCA (wiki)

$$\begin{aligned} c_{i+1} &= a_i \cdot b_i \oplus c_i \cdot (a_i \oplus b_i), \\ s_i &= a_i \oplus b_i \oplus c_i. \end{aligned}$$

- Multiplicative depth: $L = n - 1$.
- Optimize the number of AND gates: $c_{i+1} = (a_i \oplus c_i) \cdot (b_i \oplus c_i) \oplus c_i$.

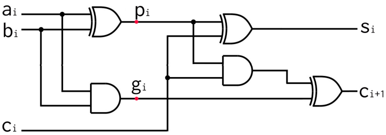

Figure: A 1-bit CLA

- Generate: $g_i = a_i \cdot b_i$
- Propagate: $p_i = a_i \oplus b_i$
- Carry: $c_{i+1} = g_i \oplus p_i \cdot c_i$
- Sum: $s_i = p_i \oplus c_i$

Addition: Carry-lookahead adder (CLA) – [Xu *et al.* '16]

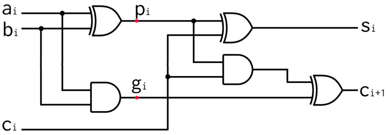

Figure: A 1-bit CLA

- Generate: $g_i = a_i \cdot b_i$
- Propagate: $p_i = a_i \oplus b_i$
- Carry: $c_{i+1} = g_i \oplus p_i \cdot c_i$
- Sum: $s_i = p_i \oplus c_i$

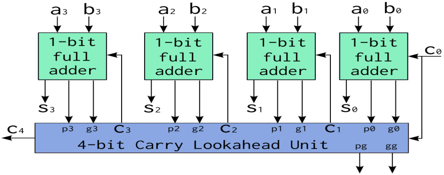

Figure: A 4-bit CLA unit with $c_4 = gg \oplus pg \cdot c_0$, where

- $pg = p_3 \cdot p_2 \cdot p_1 \cdot p_0$,
- $gg = g_3 \oplus p_3 \cdot g_2 \oplus p_3 \cdot p_2 \cdot g_1 \oplus p_3 \cdot p_2 \cdot p_1 \cdot g_0$.

Addition: Carry-lookahead adder (CLA) – [Xu *et al.* '16]

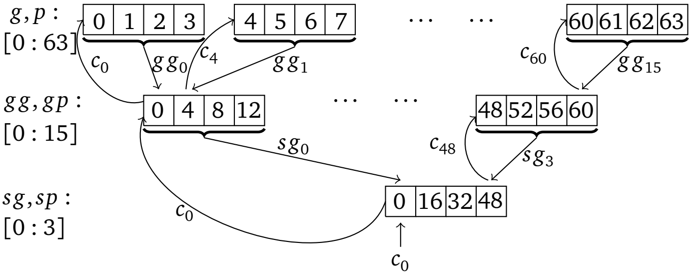

Figure: A 64-bit CLA with 4-bit CLA unit

Addition: Carry-lookahead adder (CLA) – [Xu *et al.* '16]

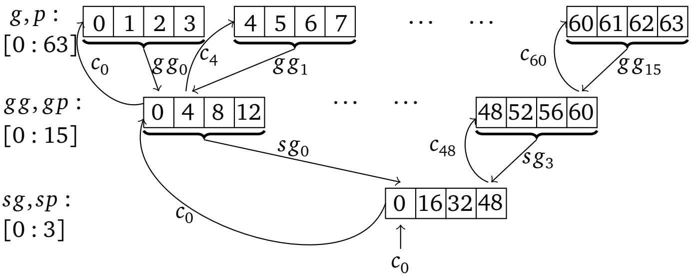

Figure: A 64-bit CLA with 4-bit CLA unit

- Multiplicative depth of an n-bit ($n = k^\ell$) CLA with k-bit CLA unit is

$$L \leq (2\ell - 1)\lceil \log k \rceil + 1 \lesssim 2 \log n.$$

Addition: Carry-lookahead adder (CLA) – [Xu *et al.* '16]

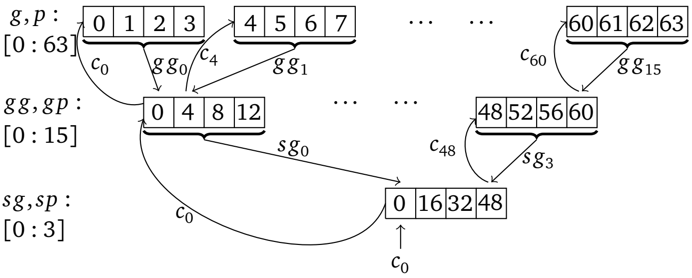

Figure: A 64-bit CLA with 4-bit CLA unit

Table: Multiplicative depth comparison

	RCA	CLA with 4-bit unit
16-bit	15	7
64-bit	63	11

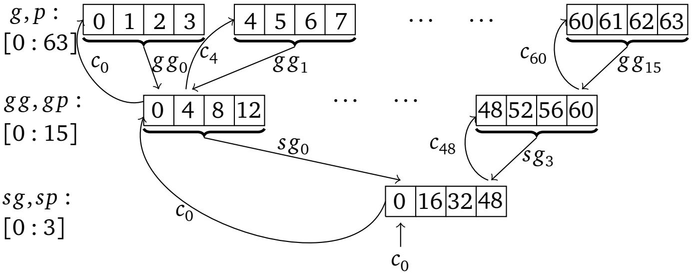

Figure: A 64-bit CLA with 4-bit CLA unit

Can we do better?

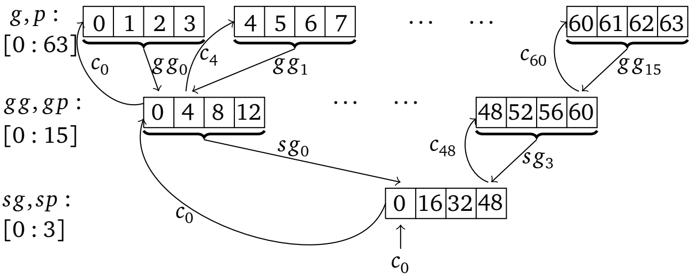

Figure: A 64-bit CLA with 4-bit CLA unit

Can we do better? Yes!

Addition: Further opitimizition in HElib – [Crawford *et al.* '18]

Recall CLA

- Generate: $g_i = a_i \cdot b_i$; propagate: $p_i = a_i \oplus b_i$; carry: $c_{i+1} = g_i \oplus p_i \cdot c_i$.

Addition: Further opitimizition in HElib – [Crawford *et al.* '18]

Recall CLA

- Generate: $g_i = a_i \cdot b_i$; propagate: $p_i = a_i \oplus b_i$; carry: $c_{i+1} = g_i \oplus p_i \cdot c_i$.
- For example,

$$c_4 = \sum_{i=0}^{3} \left(g_i \cdot \prod_{k=i+1}^{3} p_k \right) \oplus c_0 \cdot \prod_{k=0}^{3} p_k.$$

Recall CLA

- Generate: $g_i = a_i \cdot b_i$; propagate: $p_i = a_i \oplus b_i$; carry: $c_{i+1} = g_i \oplus p_i \cdot c_i$.
- For example,

$$c_4 = \sum_{i=0}^{3} \left(g_i \cdot \prod_{k=i+1}^{3} p_k \right) \oplus c_0 \cdot \prod_{k=0}^{3} p_k.$$

- The idea: extend the “generate” and “propagate” bits to intervals.
  - $p_{[i,j]} = \prod_{k=i}^{j} p_k, \quad g_{[i,j]} = g_i \cdot p_{[i+1,j]}, \quad \forall i \leq j.$

Recall CLA

- Generate: $g_i = a_i \cdot b_i$; propagate: $p_i = a_i \oplus b_i$; carry: $c_{i+1} = g_i \oplus p_i \cdot c_i$.
- For example,

$$c_4 = \sum_{i=0}^{3} \left(g_i \cdot \prod_{k=i+1}^{3} p_k \right) \oplus c_0 \cdot \prod_{k=0}^{3} p_k.$$

- The idea: extend the "generate" and "propagate" bits to intervals.
  - $p_{[i,j]} = \prod_{k=i}^{j} p_k, \quad g_{[i,j]} = g_i \cdot p_{[i+1,j]}, \quad \forall i \le j.$
  - Carry: $c_j = \sum_{i=0}^{j} g_{[i,j]} \oplus c_0 \cdot p_{[0,j-1]}$.

Addition: Further opitimizition in HElib – [Crawford *et al.* '18]

Recall CLA

- Generate: $g_i = a_i \cdot b_i$; propagate: $p_i = a_i \oplus b_i$; carry: $c_{i+1} = g_i \oplus p_i \cdot c_i$.
- For example,

$$c_4 = \sum_{i=0}^{3} \left(g_i \cdot \prod_{k=i+1}^{3} p_k \right) \oplus c_0 \cdot \prod_{k=0}^{3} p_k.$$

- The idea: extend the "generate" and "propagate" bits to intervals.
  - $p_{[i,j]} = \prod_{k=i}^{j} p_k, \quad g_{[i,j]} = g_i \cdot p_{[i+1,j]}, \quad \forall i \leq j.$
  - Carry: $c_j = \sum_{i=0}^{j} g_{[i,j]} \oplus c_0 \cdot p_{[0,j-1]}$.
  - Multiplicative depth: $L \leq \lceil \log(n+2) \rceil$ (for computing $g_{[0,n]}$)

The three-for-two procedure

- Input $u = (u_{n-1}, \ldots, u_0)$, $v = (v_{n-1}, \ldots, v_0)$, $w = (w_{n-1}, \ldots, w_0)$.

Addition: For many integers – [Crawford *et al.* '18]

The three-for-two procedure

- Input $u = (u_{n-1}, \ldots, u_0)$, $v = (v_{n-1}, \ldots, v_0)$, $w = (w_{n-1}, \ldots, w_0)$.
- Compute $u_i + v_i + w_i = x_i + 2 \cdot y_i$, where

Addition: For many integers – [Crawford *et al.* ’18]

The three-for-two procedure

- Input $u = (u_{n-1}, \ldots, u_0)$, $v = (v_{n-1}, \ldots, v_0)$, $w = (w_{n-1}, \ldots, w_0)$.
- Compute $u_i + v_i + w_i = x_i + 2 \cdot y_i$, where
  - $x_i = u_i + v_i + w_i \mod 2, \quad y_i = u_i v_i + v_i w_i + w_i u_i \mod 2.$

The three-for-two procedure

- Input $u = (u_{n-1}, \ldots, u_0)$, $v = (v_{n-1}, \ldots, v_0)$, $w = (w_{n-1}, \ldots, w_0)$.
- Compute $u_i + v_i + w_i = x_i + 2 \cdot y_i$, where
  - $x_i = u_i + v_i + w_i \mod 2, \quad y_i = u_i v_i + v_i w_i + w_i u_i \mod 2.$
- Then $u + v + w = x + y$, where

Addition: For many integers – [Crawford *et al.* '18]

The three-for-two procedure

- Input $u = (u_{n-1}, \ldots, u_0)$, $v = (v_{n-1}, \ldots, v_0)$, $w = (w_{n-1}, \ldots, w_0)$.
- Compute $u_i + v_i + w_i = x_i + 2 \cdot y_i$, where
  - $x_i = u_i + v_i + w_i \mod 2, \quad y_i = u_i v_i + v_i w_i + w_i u_i \mod 2.$
- Then $u + v + w = x + y$, where
  - $x = (x_{n-1}, \cdots, x_0), \quad y = (y_{n-1}, \cdots y_0, 0).$

Addition: For many integers – [Crawford *et al.* '18]

The three-for-two procedure

- Input $u = (u_{n-1}, \ldots, u_0)$, $v = (v_{n-1}, \ldots, v_0)$, $w = (w_{n-1}, \ldots, w_0)$.
- Compute $u_i + v_i + w_i = x_i + 2 \cdot y_i$, where
  - $x_i = u_i + v_i + w_i \mod 2, \quad y_i = u_i v_i + v_i w_i + w_i u_i \mod 2.$
- Then $u + v + w = x + y$, where
  - $x = (x_{n-1}, \cdots, x_0), \quad y = (y_{n-1}, \cdots y_0, 0).$
- Multiplicative depth: $L \leq \lceil \log(n + 2 + 1) \rceil + 1$.

Addition: For many integers – [Crawford *et al.* '18]

The three-for-two procedure

- Input $u = (u_{n-1}, \ldots, u_0)$, $v = (v_{n-1}, \ldots, v_0)$, $w = (w_{n-1}, \ldots, w_0)$.
- Compute $u_i + v_i + w_i = x_i + 2 \cdot y_i$, where
  - $x_i = u_i + v_i + w_i \mod 2, \quad y_i = u_i v_i + v_i w_i + w_i u_i \mod 2.$
- Then $u + v + w = x + y$, where
  - $x = (x_{n-1}, \cdots, x_0), \quad y = (y_{n-1}, \cdots y_0, 0).$
- Multiplicative depth: $L \leq \lceil \log(n + 2 + 1) \rceil + 1$.

Adding t integers

- Apply the three-for-two procedure until only two integers are left.
  - Multiplicative depth of this reduction: $d \approx \log_{3/2}(t)$.
  - Bitsize of input integers increases at most d.
- Then apply the addition circuit.

Subtraction

Two ways to subtract numbers:

- Design circuits for subtraction (RCS)

$$\begin{aligned} c_{i+1} &= (a_i \oplus c_i) \cdot (b_i \oplus c_i) \oplus b_i, \\ d_i &= a_i \oplus b_i \oplus c_i. \end{aligned}$$

  - Multiplicative depth: $L = n - 1$.
- Use adders to carry out subtraction:
  - Represent numbers in 2's complement form

$$\begin{aligned} &a - b = a + \tilde{b} + 1 \text{ with } c_0 = 1, \\ &\widetilde{b_i} = b_i \oplus 1. \end{aligned}$$

  - Multiplicative depth: same as adders.

Multiplication

Constructed by additions, in a pencil and paper way, plus

- [Xu *et al.*'16]: truncation and rearrange the order of additions;
- [Crawford *et al.* '18]: the add-many-numbers procedure.

			0	0	1	0
		×	0	0	1	1
			0	0	1	0
truncated bits		0	0	1	0	
	0	0	0	0		
0	0	0	0			

Figure: Multiplying two integers 2 x 3 in a 4-bit binary circuit

Multiplication

Constructed by additions, in a pencil and paper way, plus

- [Xu *et al.*'16]: truncation and rearrange the order of additions;
- [Crawford *et al.* '18]: the add-many-numbers procedure.

			0	0	1	0
		×	0	0	1	1
			0	0	1	0
truncated bits		0	0	1	0	
	0	0	0	0		
0	0	0	0			

Figure: Multiplying two integers 2 x 3 in a 4-bit binary circuit

- Multiplicative depth: $L \leq 1 + d + \lceil \log(n + d + 2) \rceil$, $d = \lceil \log_{3/2} n \rceil$.

Division with remainder: Non-restoring algorithm – [ChenGong'15]

- Start from the most significant bit of the dividend.
- Try to subtract the divisor from each digit.
- Compute the quotient and reminder accordingly.
- Multiplicative depth: $L \approx n^2$.

Division with remainder: Non-restoring algorithm – [ChenGong'15]

- Start from the most significant bit of the dividend.
- Try to subtract the divisor from each digit.
- Compute the quotient and reminder accordingly.
- Multiplicative depth: $L \approx n^2$.
  - [Çetin, Doröz, Sunar, Martin, eprint 2015/1195]: To divide a $2n$-bit number by a n-bit divisor, we can build a binary division circuit with depth of $n(2 + \log n)$.

Roadmap

1. Background
2. Arithmetic algorithms
3. Performance

Parameter settings

Most of parameters in HElib are used to compute the integer m, there is a heuristic routine called FindM:

```
long FindM( long  k,          // Security parameter
            long  L_c,        // levels, L_c ≈ 2⌈L/2⌉ + 1
            long  c,
            long  p,          // p = 2
            long  d,
            long  s,
            long  chosen_m,
            bool  verbose)
```

Performance

Table: Performance of [Xu *et al.*'16]: Run on an i7-4790 CPU at 3.60 GHz with 8 GB RAM; S-time is for single thread timing

Arithmetic	Circuit	#bits	m	#slots	L_c	S-time
Addition	RCA	16	14351	504	17	2.16
	CLA	16	7781	150	7	2.53
	CLA	64	13981	600	13	37.69
Subtraction	RCS	16	14351	504	17	2.17
	CLA	16	7781	150	7	2.52
	CLA	64	13981	600	13	37.16
Multiplication	RCA	8	8191	630	9	4.62
	RCA	16	14351	504	17	46.32
Division	RCA	4	18631	720	21	14.63
	[Chen & Gong '15]*	4	18631	720	21	67.94

*[ChenGong'15] use a machine with 8 Xeon 2.13 GHz processors and 512 GB RAM.

Performance

Table: Performance of [Xu *et al.*'16]: Run on an i7-4790 CPU at 3.60 GHz with 8 GB RAM; S-time is for single thread timing and M-time for 8 threads, $k = 80$.

Arithmetic	Circuit	#bits	m	#slots	L_c	S-time	M-time
Addition	RCA	16	14351	504	17	2.16	1.16
	CLA	16	7781	150	7	2.53	2.05
	CLA	64	13981	600	13	37.69	24.36
Subtraction	RCS	16	14351	504	17	2.17	1.20
	CLA	16	7781	150	7	2.52	2.02
	CLA	64	13981	600	13	37.16	24.73
Multiplication	RCA	8	8191	630	9	4.62	2.63
	RCA	16	14351	504	17	46.32	29.34
Division	RCA	4	18631	720	21	14.63	7.74
	[Chen & Gong '15]*	4	18631	720	21	67.94	–

*[ChenGong'15] use a machine with 8 Xeon 2.13 GHz processors and 512 GB RAM.

Performance

Table: Performance of current HElib's built-in: $m = 15709$ ($k = 210$)

		[Xu *et al.*'16]	HElib's built-in
16-bit addtion	L_c	7	5
	single thread	4.90	5.96
	8-threads	3.43	2.33
64-bit addition	L_c	13	7
	single thread	35.68	31.23
	8-threads	20.89	11.58
16-bit multiplication	L_c	17	15
	single thread	40.94	21.11
	8-threads	23.89	8.36

[Crawford *et al.*'18] applies FHE (HElib) to a logistic-regression model.

- binary arithmetic, comparisons, sorting, reciprocals, logarithms
- It takes a few hours to handle thousands of genomic records and hundreds of fields.

Towards practical applications

[Crawford *et al.*'18] applies FHE (HElib) to a logistic-regression model.

- binary arithmetic, comparisons, sorting, reciprocals, logarithms
- It takes a few hours to handle thousands of genomic records and hundreds of fields.

Further consideration

- Of course, the performance needs to be optimized further.
- However, the main obstacle is not the performance.

Towards practical applications

[Crawford *et al.*'18] applies FHE (HElib) to a logistic-regression model.

- binary arithmetic, comparisons, sorting, reciprocals, logarithms
- It takes a few hours to handle thousands of genomic records and hundreds of fields.

Further consideration

- Of course, the performance needs to be optimized further.
- However, the main obstacle is not the performance.
  - We lack good development and support tools.
  - We need many more libraries and FHE toolboxes.

Towards practical applications

[Crawford *et al.*'18] applies FHE (HElib) to a logistic-regression model.

- binary arithmetic, comparisons, sorting, reciprocals, logarithms
- It takes a few hours to handle thousands of genomic records and hundreds of fields.

Further consideration

- Of course, the performance needs to be optimized further.
- However, the main obstacle is not the performance.
  - We lack good development and support tools.
  - We need many more libraries and FHE toolboxes.

THANKS