SecureMR: MapReduce Computation Using Homomorphic Encryption and Program Partitioning

Yao Dong¹, Ana Milanova
Rensselaer Polytechnic Institute

and Julian Dolby
IBM TJ Watson Research Center

¹Work and slides by Yao Dong
Motivation – Cloud Computing

- Outsource data and computation to cloud service providers
  - Google Cloud Platform (GCP), Amazon Web Service (AWS)
- Inexpensive, efficient and convenient
- Risk losing confidentiality of their data
  - Encrypt the data
  - Compute over encrypted data?
One Approach - Fully Homomorphic Encryption (FHE)

- Supports arbitrary computation on ciphertexts

- First constructed by Gentry [Gentry, Phd Thesis ‘09]

- Expensive
  - First generation: 30 min / bit operation
  - Second generation: 0.2 ms / add gate
Our Approach - Partially Homomorphic Encryption (PHE)

- More efficient than FHE
  - Paillier (Java): 1.3 μs / add operation
- Support certain operation on ciphertexts
  - AH: addition operation (+)
  - MH: multiplication operation (×)
  - DET: equality checking (==)
  - OPE: comparison operation (<)
  - RND: no operation
- Conversion – different operations on the same variable
Conversion

• Convert between different PHE schemes
  ◦ E.g., if \( d < 0 \) \( d = d + 1 \); // \( d \) is ciphertext

• A conversion process requires private/secret keys which should be kept secure, e.g., in a trusted client

• Conversion is expensive
Our Model

Trusted Client

Untrusted Server

ciphertexts (results)

Conversion

ciphertexts (input)

input data

encrypt

compute over ciphertexts

encryption keys
Our Overarching Goal

original program → analyze and transform → transformed program

Make transformed program efficiently perform computation over encrypted data on the cloud
Contributions

- **SecureMR**
  - Novel program analyses that transform MapReduce programs to work on encrypted data

- Implementation

- Evaluation on Google Cloud Platform
  - Max overhead 220% on programs with conversion (64 nodes)
Outline

- Key aspects of SecureMR
  - Encryption inference
    - Decide encryption scheme(s) for input columns
    - Possibly multiple encryptions per column
  - Optimal conversion placement
  - Program partitioning
    - Extract a segment with a high density of conversions
    - Run the segment on client in plaintext form
- Evaluation
- Related work
A MapReduce Example – Histogram Movies

map (key, value) {
    int sumRatings = 0;
    while (token.hasMoreTokens()) {
        int rating = token.nextToken();
        sumRatings += rating;
    }
    output.collect(outValue, 1);
}

reduce (key, values) {
    int sum = 0;
    while (values.hasNext()) {
        int value = values.next();
        sum += value;
    }
    output.collect(key, sum);
}

Movie1: 3,5,4,5
Movie2: 1,2,1,1,1,2
...
Preliminaries: Reaching Definitions (RD) Analysis

- Defined over a Control Flow Graph (CFG)
  - and are CFG nodes
- Compute *def-use* chains

```
x = ...
```

```
z = x + y
```

- Augment RD to account for the data flow from map to reduce
  - map output *key* → reduce input *key*
  - map output *value* → reduce input *value*
Outline

- Key aspects of SecureMR
  - Encryption inference
  - Optimal conversion placement
  - Program partitioning

- Evaluation

- Related work
Encryption Inference

- User annotates source definitions
- For each def-use chain, nodes give rise to the set of necessary encryptions for:
  - \(+\) AH, MH,
  - \(==\) DET, < OPE.
- One def may have multiple uses and each use may demand different encryptions
- Include all encryptions in the input, each use retrieving the corresponding encryption
Outline

• Key aspects of SecureMR
  ◦ Encryption inference
  ◦ Optimal conversion placement
  ◦ Program partitioning

• Evaluation

• Related work
Optimal Conversion Placement

- Def-use chain \((d,u)\) requires conversion
  - \((\text{e.g., } x=n+m)\) produces an encrypted value incompatible with the use at

- Place conversion
  - Every path from \(d\) to \(u\) is covered
  - **Optimal** placement -- total number of conversion executions is minimal

- Approach: **Min-cut** problem
An Example with Conversions

```c
int gcd (int a, int b) {
    while (a != b) {
        if (a > b)  
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

Optimal placement conversion

ENTRY

F

while

T

if

T

F

a = a - b

b = b - a

return a

: OPE

: AH
Outline

- Key aspects of SecureMR
  - Encryption inference
  - Optimal conversion placement
  - Program partitioning

- Evaluation

- Related work
while ( ... ) {
    sumRatings += rating; // rating is sensitive
    totalReviews++;
}

avgReview = sumRatings / totalReviews;
absReview = Math.floor(avgReview);
fraction = avgReview – absReview;
while (division < limit) {
    if (fraction < division) {
        outValue = absReview + division;
    }
    division++;
}

output.collect(outValue, 1);
Example – Histogram Movies

while ( ... ) {
    sumRatings += rating; // rating is sensitive
    totalReviews++;
}

avgReview = sumRatings / totalReviews;
absReview = Math.floor(avgReview);
fraction = avgReview – absReview;
while (division < limit) {
    if (fraction < division) {
        outValue = absReview + division;
    }
    division++;
}

output.collect(outValue, 1);
Program Partitioning Heuristic

- **map and reduce** represented as sequence:
  - $S_1 \ S_2 \ \ldots \ S_n$ each $S_i$ is either
    - Assignment statement $x = e$
    - While statement **while (e) S**
    - If-then-else statement **if (e) then S1 else S2**

- (Roughly) we identify the minimal sequence $S_i \ldots S_j$ that covers all conversion targets

- Typical structure of **map and reduce**:
  - **while (e) S** $S_1 \ S_2 \ \ldots \ S_n$
Example – Histogram Movies

while ( ... ) {
    sumRatings += rating;
    totalReviews++;
}
sendToClient(sumRatings, totalReviews);

outValue = receiveFromClient();
output.collect(outValue, 1);

[sumRatings, totalReviews] = receiveFromServer();

avgReview = sumRatings / totalReviews;
absReview = Math.floor(avgReview);

fraction = avgReview – absReview;
while (division < limit) {
    if (fraction < division) {
        outValue = absReview + division;
    }
    division++;
}
sendToServer(outValue);
Communication Impact on Performance

- **Case 1:**
  - Communication inside an input-dependent loop in `map`

- **Case 2:**
  - Communication outside an input-dependent loop in `map` (i.e., runs once per `map`)

- **Case 3:**
  - Communication outside the input-dependent loop in `reduce`
Communication Impact on Performance

map (...) {
    while (...) {
        int review = …;
        int r = review * review;
        ... communication
    }
}

20000 rows
Movie1: 3,5,4,5
Movie2: 1,2,1,1,1,2
…
3000 reviews

Map
Reduce
Result

Input Data Files
Rows
Key-value Pairs

• Case 1: worst case
• Conversion is inside an input-dependent loop of map.
• Optimization: row precomputation

6 × 10^7 communications for only 1G data!
Communication Impact on Performance

Figure 3.1: Overview of MapReduce.

The basics of MapReduce are illustrated in Wikipedia [40] and the webpage [41]. The map function takes as input an individual row $r_i$, and produces one or more key-value pairs, where the key and value are computed from columns of $r_i$. MapReduce breaks the original input file into $N$ parts. It runs $N$ processes in parallel, each process running map sequentially on each row from its input file. Each process produces a sequence of key-value pairs corresponding to the portion of the original file. Typically, map is highly parallel.

MapReduce then shuffles outputs based on the output keys, and groups them into per-key "reducer" groups. There are as many "reducer" groups as there are distinct keys. The reduce function aggregates the values in each reducer group.

Fig. 3.2 shows Histogram Movies, a MapReduce program from the classical PUMA benchmark set [42]. As the name suggests, it takes as input a sequence of movie ratings (from 1 to 5), and computes a histogram of the number of movies rated 1.5, 2, 2.5, etc. on average. Each input row has (1) a movie title column, and (2) a review column. For example:

- Movie1: 3, 5, 4, 5
- Movie2: 1, 2, 1, 1, 1, 2

3000 reviews

20000 rows

Case 2:

- Conversion is outside an input-dependent loop of map.

Optimization: piggybacking

20000 communications for only 1G data!
Communication Impact on Performance

reduce (...) {
    while (...) {
        ...
    }
    communication
}

Case 3:
Conversion is outside the input-dependent loop of reduce.

A small number of communication

Fig. 3.1 shows an overview of MapReduce. The basics of MapReduce are illustrated in Wikipedia [40] and the webpage [41]. The map function takes as input an individual row $r_i$, and produces one or more key-value pairs, where the key and value are computed from columns of $r_i$. MapReduce breaks the original input file into $N$ parts. It runs $N$ processes in parallel, each process running map sequentially on each row from its input file. Each process produces a sequence of key-value pairs corresponding to the portion of the original file. Typically, map is highly parallel.

MapReduce then shuffles outputs based on the output keys, and groups them into per-key "reducer" groups. There are as many "reducer" groups as there are distinct keys. The reduce function aggregates the values in each reducer group.

Fig. 3.2 shows Histogram Movies, a MapReduce program from the classical PUMA benchmark set [42]. As the name suggests, it takes as input a sequence of movie ratings (from 1 to 5), and computes a histogram of the number of movies rated 1.5, 2, 2.5, etc. on average. Each input row has (1) a movie title column, and...
Outline

- Key aspects of SecureMR
  - Necessary encryption inference
  - Optimal conversion placement
  - Program partitioning

- Evaluation

- Related work
Experimental Results

• MapReduce benchmarks (36 in total)
  ◦ 3 suites: PIGMIX2 (17), Brown (6) and PUMA (13)

• Analysis results
  ◦ 32 out of 36 benchmarks (89%) require NO conversions
  ◦ 3 require conversion in map (case 2)
  ◦ 1 requires conversion in reduce (case 3)
  ◦ 13 are computation-intensive (requiring AH and/or MH)
  ◦ 23 are search-intensive (requiring DET and/or OPE)
Overhead Evaluation

- Focus on 13 computation-intensive benchmarks
- Experiment environment
  - Untrusted cloud
    - Google Cloud clusters (8, 16, 32 and 64 nodes)
    - Each node has 1 CPU and 3.75 GB of memory
  - Trusted client
    - An ancient machine in Yao’s student office
    - 2 CPUs and 4 GB of memory
Running Time of Benchmarks with Conversions

(a) Histogram Movies

Conversion in map

Case 2

(d) Join Task

Conversion in reduce

Case 3
Running Time of Benchmarks with Conversions

(b) K-Means

Conversion in map

Case 2

(c) Classification

Conversion in map

Case 2
### Running Time of Benchmarks without Conversions

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Running Time (sec)</th>
<th>Original</th>
<th>Transformed</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>607</td>
<td>203</td>
<td>217%</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>269</td>
<td>203</td>
<td>236%</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>433</td>
<td>203</td>
<td>113%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Running Time (sec)</th>
<th>Original</th>
<th>Transformed</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>381</td>
<td>118</td>
<td>167%</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>442</td>
<td>118</td>
<td>205%</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>218</td>
<td>118</td>
<td>85%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Running Time (sec)</th>
<th>Original</th>
<th>Transformed</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>656</td>
<td>322</td>
<td>91%</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>611</td>
<td>322</td>
<td>73%</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>397</td>
<td>322</td>
<td>23%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Running Time (sec)</th>
<th>Original</th>
<th>Transformed</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>359</td>
<td>137</td>
<td>124%</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>363</td>
<td>137</td>
<td>123%</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>211</td>
<td>137</td>
<td>54%</td>
<td></td>
</tr>
</tbody>
</table>

Note: Figure 5.7: Benchmarks without conversion that MrCrypt’s analysis can handle, but does not evaluate experimentally.

Communication with the client. Thus, we observe significantly larger overhead in Figs. 5.4a, 5.4b, and 5.4c than in Fig. 5.4d.

Note that in Fig. 5.5 the overhead does not increase linearly as the cluster size grows, rather it stays stable and even decrease for benchmarks Classification and Join Task. This demonstrates that the communication between server and client associated with the computation over encrypted data, specifically conversions, does not increase the overall running time. Also, we observed super-linear scalability.
Related Work

- Computation over encrypted data using PHE
  - MrCrypt [Tetali et al. OOPSLA’13] (67% vs 89%)
  - CryptDB [Popa et al. SOSP’11]
  - AutoCrypt [Tople et al. CCS’13] (C programs)
  - Monomi [Tu et al. PVLDB’13]

- Program partitioning
  - EnerJ [Sampson et al. PLDI’11], Swift [Chong et al. SOSP’01]
Conclusions

- SecureMR
- Implementation
- Evaluation on Google Cloud Platform
  - Max overhead 220% on programs with conversion (64 nodes)

- Publicly available: https://github.com/proganalysis/type-inference
THANK YOU!