SecureMR: Secure MapReduce Computation Using Homomorphic Encryption and Program Partitioning

Yao Dong
Rensselaer Polytechnic Institute
dongy6@rpi.edu

Ana Milanova
Rensselaer Polytechnic Institute
milanova@cs.rpi.edu

Julian Dolby
IBM Thomas J. Watson Research Center
dolby@us.ibm.com

ABSTRACT

In cloud computing customers upload data and computation to cloud providers. As they upload their data to the cloud provider, they typically give up data confidentiality. We develop SecureMR, a system that analyzes and transforms MapReduce programs to operate over encrypted data. SecureMR makes use of partially homomorphic encryption and a trusted client. We evaluate SecureMR on a set of complex computation-intensive MapReduce benchmarks.

CCS CONCEPTS

• **Security and privacy → Domain-specific security and privacy architectures**; *Management and querying of encrypted data*;

KEYWORDS

cloud computing, homomorphic encryption, MapReduce

1 INTRODUCTION

Cloud service providers such as Google Cloud Platform (GCP) and Amazon Web Service (AWS) offer a wide range of data storage and computation products. Customers increasingly outsource data and computation to such third-party cloud providers. Unfortunately, when customers upload their data to the cloud provider, they typically give up data confidentiality. The problem is, can one take advantage of inexpensive, efficient, and convenient cloud services, while preserving (to an extent) the confidentiality of their data?

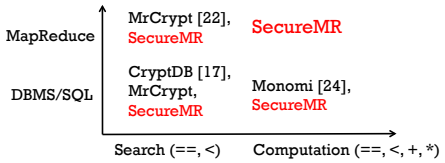
One approach towards this problem, is to use homomorphic encryption. Customers encrypt their data using homomorphic encryption, which allows computation on ciphertext. They upload encrypted data, and programs that operate on ciphertext, thus preserving data confidentiality.

There are two categories of homomorphic encryption. Fully Homomorphic Encryption (FHE) [8] supports arbitrary computation on ciphertext. This is an important theoretical achievement [20], however, the existing implementations of FHE are still prohibitively expensive [4, 9, 10]. Partially Homomorphic Encryption (PHE) is relatively efficient, however, it is limited in the sense that each PHE cryptosystem supports a single operation on ciphertext. For example, the Pailler [16] cryptosystem, which we refer to as AH, supports addition, but does not support comparison. To make use of PHE, one must combine different schemes. The principal problem is data *conversion* — when data is computed using one encryption scheme (e.g., AH) but requires an operation that is not supported (e.g., comparison), data must be converted into a new scheme that supports the operation.

Partially homomorphic encryption has been recognized as a promising direction [6, 19, 24–26]. One way to address the problem of conversion is to maintain a *trusted client* [6, 23, 26]. The client stores the cryptographic keys. It can perform conversion, i.e., receive data in one encryption scheme (e.g., AH), decrypt it, encrypt it into the new scheme, then send it back to the untrusted cloud server that runs the program. It can also run segments of the program to alleviate conversion cost, or compute operations not supported by current encryption schemes. Clearly, this approach entails communication between the server and the trusted client.

This paper presents SecureMR, a system that enables secure MapReduce computation on untrusted cloud servers using partially homomorphic encryption and a trusted client. It makes the following contributions:

- We develop novel static analysis that infers necessary encryption schemes for input data.
- We formalize the problem of conversion placement as an instance of the classical Min-cut problem. Min-cut ensures *optimal* conversion placement.
- We propose a cost model that captures two conflicting goals: (1) maximize the amount of computation running on the cloud server, while in the same time (2) minimize communication between the server and the client. This cost model guides a heuristic refactoring that partitions the MapReduce program into server and client partitions.
- We evaluate the system on Google Cloud clusters of up to 64 nodes, using standard MapReduce benchmark suites [1, 17, 18]. We are the first to report on complex computation-intensive MapReduce applications such as K-Means clustering. The slowdown of transformed programs relative to original programs ranges from 1.4x to 3.9x on applications that require communication between server and client.



The graph above positions our work in the context of existing work on PHE for cloud computing. There are two dimensions:

HoTSoS '18, April 10–11, 2018, Raleigh, NC, USA

ACM ISBN 978-1-4503-6455-3/18/04…$15.00
https://doi.org/10.1145/3190619.3190638

DBMS/SQL vs. MapReduce, and search-intensive vs. computation-intensive applications. Search-intensive applications make use of equality and comparison operations, which entail inexpensive encryption schemes. CryptDB [19] and MrCrypt [24] evaluate search-intensive applications, respectively in the database and MapReduce domain. Computation-intensive applications perform arithmetic operations (e.g., +), which entail substantially more expensive encryption schemes, and often require conversion from one scheme to another. CryptDB and MrCrypt cannot handle applications that require conversion. Monomi [26] evaluates computation-intensive SQL queries. Similarly to SecureMR, it makes use of a trusted client that can perform conversion and computation. To the best of our knowledge, SecureMR is the first system that handles and evaluates computation-intensive MapReduce applications. It covers the largest set of MapReduce programs, assesses conversion and communication cost on a real cloud platform, and sets directions for future work in this important problem domain. (The graph includes MrCrypt and SecureMR in the database domain, because the SQL queries can be easily translated into MapReduce.)

The rest of the paper is organized as follows. Sect. 2 presents an overview of MapReduce. Sects. 3 and 4 give, respectively, an informal and formal account of SecureMR. Sect. 5 details our experiments and results. Sect. 6 discusses related work, and Sect. 7 concludes.

2 OVERVIEW OF MAPREDUCE

The MapReduce paradigm supports large-scale parallel data analysis. At a high-level, a MapReduce program takes as input large files of *rows*, where, just as in a database table, each row is composed of *columns*.

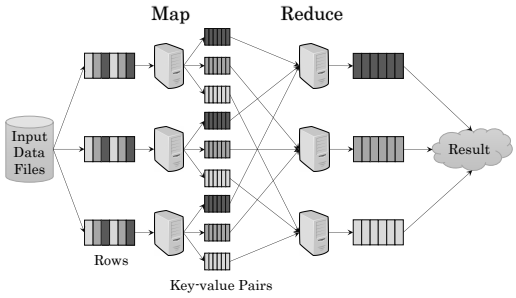


Figure 1: Overview of MapReduce.

Fig. 1 shows an overview of MapReduce. 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. 2 shows Histogram Movies, a MapReduce program from the classical PUMA benchmark set [1]. (We reference Fig. 2 extensively throughout the paper. We copy the figure in the last page, Appendix B. Reviewers may rip the last page and reference the figure.) As the name suggests, Histogram Movies 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 ratings column, for example:

$$Movie_1 : 3, 5, 4, 5$$
$$Movie_2 : 1, 2, 1, 1, 1, 2$$
$$\ldots$$

Function map extracts the *ratings* column (line 6) of each row. It computes the average rating per movie (row), and subsequently rounds it down to the nearest 0.5. The result is a key-value pair $\langle rounded\_avg, 1\rangle$ per row:

$$\langle 4.5, 1\rangle$$
$$\langle 1.5, 1\rangle$$
$$\ldots$$

The framework shuffles the above key-value pairs into 8 reducer groups. Each reducer receives an iterator of 1 values, which it sums up, thus completing the ratings histogram.

We observe that MapReduce applications are particularly amenable to static analysis. Firstly, memory references tend to be easily resolved. Aliasing is essentially non-existent. Memory references are local variables (see map in Fig. 2), and even expressions that denote heap locations can be resolved uniquely. Secondly, map and reduce are short (about 100 LOC), and exhibit predictable control flow. For example, map may have `if-then-else` statements that distinguish between distinct database tables and/or select certain rows by column values; reduce has a `while` statement that iterates over the values in the reducer group. There are few calls to Java libraries other than `Math`. Method calls to local libraries are rare as well, and typically they can be inlined.

Therefore, MapReduce programs are particularly amenable to precise static analysis. We apply Reaching definitions, a classical static analysis [14] adapting it to MapReduce.

3 OVERVIEW OF SECUREMR

We assume an architecture that includes two basic components, an untrusted cluster, called *server*, and a trusted *client*. The server is a cluster on the cloud (e.g., Google cloud, AWS, etc.), where users can upload and run MapReduce applications. We assume that the server provides inexpensive but untrusted computation. We assume that server administrators are passive adversaries — that is, they can observe the data and code uploaded on the server, but would not modify that data or code. The client is a trusted machine where users store original input data and private keys.

We make use of five cryptosystems to encrypt input data, and perform computation on encrypted data. They are randomized encryption (RND), additively homomorphic encryption (AH), multiplicatively homomorphic encryption (MH), deterministic encryption (DET) and order-preserving encryption (OPE). RND supports no operations on ciphertext. The rest support a single operation: addition/subtraction (AH), multiplication/division (MH), equality checking (DET) and comparison (OPE). RND is useful for MapReduce programs because often entire columns of input data are not involved in any operation. RND provides the strongest security, while OPE, which reveals order, provides the weakest security.

```
1  map(LongWritable key, Text value) {
2    int totalReviews = 0, sumRatings = 0;
3    float avgReview, absReview, fraction, outValue = 0.0f;
4    String line = value.toString();
5
6    String reviews = line.substring(line.indexOf(":") + 1);
7    StringTokenizer token = new StringTokenizer(reviews, ",");
8    while (token.hasMoreTokens()) {
9      int rating = Integer.parseInt(token.nextToken()); // rating is encrypted source
10     sumRatings = sumRatings + rating;
11     totalReviews = totalReviews + 1;
12   }
13
14   avgReview = sumRatings/totalReviews;
15   absReview = Math.floor(avgReview);
16   fraction = avgReview − absReview;
17   int limit = 1.0f/division; // division = 0.5f.
18   int i = 1;
19   while (i <= limit) {
20     float x = i*division;
21     if (x − division <= fraction && fraction < x) {
22       outValue = absReview + x;
23     }
24     i = i + 1;
25   }
26
27   output.collect(outValue, 1);
28 }
```

```
1  reduce(FloatWritable key, Iterator<...> values) {
2    int sum = 0;
3    while (values.hasNext()) {
4      int value = values.next();
5      sum += value;
6    }
7    output.collect(key, sum);
8  }
```

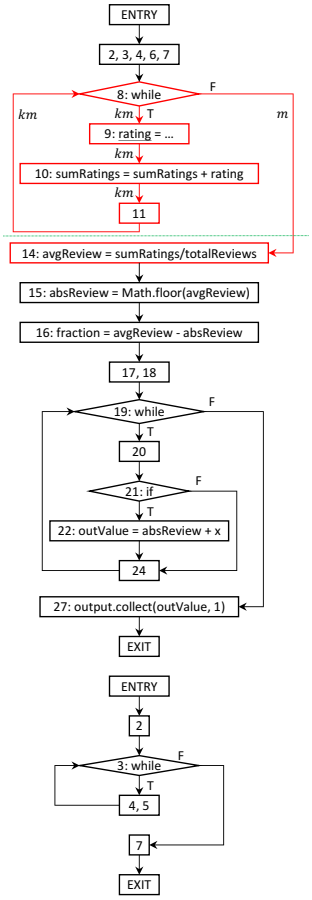


Figure 2: Histogram Movies from the PUMA benchmark set, and its Control-flow Graph (CFG). Nodes 8-14 (shown in red) comprise the flow network for def-use chain (10,14); edge (8,14), cut with a green dotted line, is the Min-cut.

Necessary encryption inference. First, we infer the encryption schemes for each column of input data, based on the operations run on column data. We make use of Reaching definitions analysis. We start by annotating variable definitions that are read directly from input data. We assume that all input columns are sensitive, and must be sent to the server in encrypted form. Reaching definitions analysis links the *definition* of a variable to the *uses* of that variable. There must be an encryption for the operation at each use. For example in Fig. 2, we annotate the definition of rating in line 9, as rating is read directly from the input file. There is a single use of rating, in the addition in line 10, which entails that it is encrypted with AH. The other column of data, "movie title", is not involved in any operation, so it is encrypted with RND. Note that there are multiple encryptions per column if different uses demand different encryptions.

Optimal conversion placement. Unfortunately, encrypting input columns, even with multiple encryptions, is not enough. When the same data is involved in different operations, the program needs *conversion* from one encryption scheme to another. For instance in Fig. 2, the ciphertext in sumRatings at line 10 is encrypted with AH because it is the result of an addition. Reaching definition analysis links the definition in line 10 to the use in line 14 and determines that there is need for conversion from AH to MH in order to carry out the division in line 14. Reaching definitions analysis computes the set of statements that depend on sensitive input data and therefore must be carried out in encrypted form. In information-flow terms, it tracks *explicit flow* from input data to the rest of the MapReduce program.

Conversion entails *communication* with the trusted client during program execution. Specifically, a conversion of a variable consists of the following steps: (1) the server sends the ciphertext (e.g., AH) to the client, (2) the client decrypts it, and encrypts it into the new scheme (e.g., MH), (3) the client sends the new ciphertext back to the server. Clearly, communication is costly, and we must minimize its impact. One key insight of our work is that conversion placement matters. When there is a definition-and-use pair that requires conversion, the conversion can be placed at many different edges. Consider 10-and-14. The conversion of sumRatings from AH to MH can be placed at edges (10,11), (11,8), or at (8,14), that is, immediately before line 14, outside the loop. Clearly, the first two choices are worse because they redundantly convert all intermediate values of sumRatings; the conversion may be executed many

```
s   ::= s; s
      | x = y
      | x = y aop z
      | while (x bop y) { s }
      | if (x bop y) { s } else { s }   statement
aop ::= + | − | ∗ | /                   arithmetic operator
bop ::= == | ! = | < | ≤                comparison operator
```

Figure 3: Syntax. s represents a sequence of statements. x, y, and z denote locations, including constants, local variables, parameters, and expressions referring to heap locations.

times, depending on the loop bound. Converting right before line 14 presents the best choice — conversion runs only once.

A key contribution of our work is the formalization of conversion placement. We frame the problem in terms of the classical Min-cut/Max-flow problem. We first assign weights on the edges of the program control-flow graph, which statically estimate the number of times each edge executes. Then, given a definition-and-use pair (d, u) that requires conversion, we place conversions at the Min-cut edges on the graph from d to u. This achieves two purposes: (1) it covers all control-flow paths from d to u, and (2) it guarantees minimal number of executions of the required conversion.

Cost model and heuristic refactoring. The previous phase places conversions at the appropriate control-flow-graph edges. The program can run on the server initiating communication with the client at each conversion edge. However, this approach may turn prohibitively expensive. It would be more efficient to extract a short segment with high conversion density, and run the entire segment on the client in plaintext form. For example, the code segment 14-25 in Fig. 2 demands a conversion at each statement (roughly). It is substantially more efficient to extract this segment into a method that runs on the client. The server sends the AH-encrypted sumRatings (and plaintext numRatings) to the client, the client decrypts it, computes outValue, encrypts it into DET, and sends the ciphertext back to the server.

Another contribution of our work is a cost model using Integer programming. The cost model guides our heuristic solution, which is simple, but effective for MapReduce programs: intuitively, we extract the minimal segment that contains all conversions to run on the client.

4 FORMAL ACCOUNT OF SECUREMR

Sect. 4.1 formalizes the syntax of MapReduce programs, and the notion of the control flow graph (CFG) with weights on nodes and edges. It also introduces the classical Reaching definitions analysis, which is the foundation for our transformation. Sect. 4.2 explains the inference of encryption schemes using def-use chains. Sect. 4.3 describes the optimal placement of conversions. Sect. 4.4 discusses the cost model and the heuristic for program partitioning.

4.1 Preliminaries

4.1.1 Syntax and Control-flow Graph. Fig. 3 abstracts the syntax of the *map* and *reduce* methods. Map and reduce are sequences of statements, where each statement is either (1) a copy propagation assignment, (2) a three-address assignment, (3) a while statement, or (4) an if − then − else statement. The grammar gives rise to the Control-flow Graph (CFG) of the program.[1] (We elide the actual construction of the CFG as it is standard [2, 22].) We assume that each CFG represents a single procedure, either map or reduce, and perform standard intraprocedural analysis. The CFGs for map and reduce of Histogram Movies are shown adjacent to the code in Fig. 2.

4.1.2 Weights on CFG Nodes and Edges. We assign weights to CFG nodes and edges to approximate the number of times a node or edge executes. These weights serve two purposes: (1) we use weights on nodes and edges to determine the optimal placement of conversions; we discuss this in more detail in Sect. 4.3, and (2) we use weights in the cost model, e.g., a conversion in a while-loop with input-dependent bounds in map, is substantially more costly than a conversion at the end of a reduce, when the number of reduce groups is 1; we discuss this in Sect. 4.4.

$$
\begin{array}{lcl}
s ::= s_1; s_2 & \Rightarrow & s_1.w = s_2.w = s.w \\
s ::= \text{while (x bop y) } \{ s_1 \} & \Rightarrow & s_1.w = k \cdot s.w \\
s ::= \text{if (x bop y) } \{ s_1 \} \text{ else } \{ s_2 \} & \Rightarrow & s_1.w = s_2.w = \frac{s.w}{2}
\end{array}
$$

Figure 4: A top-down attribute grammar that assigns weights to CFG nodes. Attribute w approximates the number of times a construct executes: (1) in a sequence, s_1 and s_2 inherit the weight of their parent in the parse tree, (2) loop body s_1 executes k times s's weight, where k is the number of loop iterations, and (3) assuming the two branches of an if-statement are of equal-probability, each branch executes half the times s does.

Fig. 4 defines an attribute grammar over the syntax in Fig. 3. This is a standard top-down grammar [2, Chapter 5], [22, Chapter 4]. Weight $s.w$ of start symbol s in map is initialized to a large constant m, to reflect a large number of executions of map, and to account for parallelism. Constant m propagates as multiplier towards individual statements in map. Similarly, there is weight $s.w$ of start symbol s in reduce. We set $s.w$ of reduce to the number of reduce keys whenever this number is known statically; we set it to $\frac{m}{2}$ otherwise. (Many benchmarks have either 1 or a small constant number of reduce input groups. For example, Histogram Movies has 8.) Loop bounds that are known statically carry the corresponding multiplier. For example, the while loop in lines 19-25 in Fig. 2 iterates only twice, and therefore carries multiplier 2. Loops that are unknown statically, i.e., are input-dependent, carry a multiplier k. E.g., loops 8-12 in map, and 3-6 in reduce are input-dependent. Both m and k are parameters that may be set at the discretion of analysis designer.

Let us return to Fig. 2. The weight of statement 10: sumRatings = sumRatings + rating is $m \cdot k$, the weights propagating down from map, and then while-loop 8-12. The weight accounts for an input-dependent (i.e., unknown, and potentially large) number of movie rows in the input file, and subsequently, an input-dependent number of ratings per movie row. The weight of statement 22: outValue = absReview + x is m, which is derived from $m \cdot 2 \cdot \frac{1}{2}$, i.e., m propagating down from map, 2 propagating from while loop 19-25, which has a static bound of 2, and $\frac{1}{2}$ from the if statement.

[1] The grammar is ambiguous due to the first production, however the ambiguity has no influence on our analysis.

In addition, we must assign weights to CFG edges. This is necessary because we place conversions based on Min-cut, which takes a directed graph with edge weights. Assignment is straightforward given the weights on grammar constructs assigned by Fig. 4. For example, the edge that originates at an assignment node, has the weight of that assignment (e.g., edge (14,15) in Fig. 2 has the weight of node 14). The True branch edge of a while statement executes k times the compound while statement s (e.g., edge (8,9) has weight km). Finally, the weights of edges originating at the if condition, are 1/2 the weights of the compound if statement (e.g., edge (21,22) is m).

4.1.3 Reaching Definitions. Reaching definitions (RD) is a classical data-flow analysis [2, 14]. It computes *def-use chains* (d, u) where d is a definition of a variable x: e.g., x = y + z, and u is a use of x: e.g., z = x * y, or x > y. Reaching definitions is defined over a CFG where d and u are nodes in the CFG. A def-use chain (d, u) entails that there is a definition-free path from d to u, or in other words, the definition of x at d *may* reach the use of x at u. RD is standard, and we do not elaborate on it. As mentioned earlier, the nature of MapReduce programs lends well to this classical precise analysis.

Examples of def-use chains in Fig. 2 are (9,10) (a definition of rating at 9, and use of rating at 10), (16,21), and (24,19). We augment standard RD to account for the connection and data flow from map to reduce. In our example, we introduce a def-use chain from outValue at 27 in map, to key in reduce, and from 1 at 27 in map, to value at line 4 in reduce.

Let DU denote the set of def-use chains computed by Reaching definitions. Without loss of generality we assume that for each $(d, u) \in DU$, u is either a computation x = y aop z, or a comparison x bop y, i.e., it is not a propagation x = y. One can easily augment standard RD analysis to accommodate this requirement.

4.2 Necessary Encryption Inference and Row Precomputation

Given def-use chains, determining the necessary encryptions is straightforward. First, our analysis requires that the user annotates as *sources* definitions that are read directly from input columns in map. In Fig. 2, rating in line 9 is a source. Let c denote a source, 9 in our example. For each def-use chain $(c, u) \in DU$, nodes u give rise to the set of necessary encryptions for c: if u contains an addition/subtraction, then c is encrypted with AH, if u contains a multiplication/division, then c is MH, if u contains an inequality comparison (e.g., <), then c is OPE, and finally, if u contains an equality comparison, then c is DET. Note that there may be multiple def-use chains (c, u) and each one may demand different encryption. We include all encryptions in the input, transforming each use to refer to the corresponding encryption. In the running example, there is a single def-use chain for rating, (9,10). Since 10 is an addition, rating must be encrypted with AH. Fig. 5 shows an example that requires multiple encryptions for the same input column. The example comes from MapReduce program L16 from the PIGMIX2 benchmark suite [18].

Reaching definitions enables *row precomputation*, an optimization technique similar to the one in [26], that reduces the need for conversion. We consider all uses u in map, such that u is an arithmetic operation, where one operand is directly read from the input file (i.e., it has exactly one def-use chain $(c, u) \in DU$), and the other operand is either a constant, or is also read from the input file. In such cases, one can precompute the expression at u and encrypt it in the input file according to subsequent uses of u.

```
1  reduce(Text key, Iterator<...> iter) {
2    Set<Double> hash = new HashSet<Double>();
3    while (iter.hasNext()) {
4      Double value = iter.next(); // value is encrypted source
5      hash.add(value);
6    }
7    Double rev = new Double(0);
8    for (Double d : hash)
9      rev += d;
10   output.collect(key, rev);
11 }
```

Figure 5: Reduce of L16 from PIGMIX2. Map selects two input columns and passes them to reduce. There is no computation in map, therefore, value **in line 4 stores the encrypted input source. There are two def-use pairs for** value**: (4,5) and (4,9). The use at 5 requires that** value **is encrypted with DET due to the inherent equality test in** HashSet**. The use at 9 requires that** value **is encrypted with AH. (We elide containers and copy propagation, as mentioned earlier, and link 4 directly to 9.) Thus,** value **must be encrypted with both DET and AH. We include the two distinct ciphertexts in the input.**

Consider the following code snippet from K-Means:

```
1  map(LongWritable Key, Text value) {
2    String line = value.toString();
3    String reviews = line.substring(line.indexOf(":") + 1);
4    StringTokenizer token = new StringTokenizer(reviews, ",");
5    while (token.hasMoreTokens()) {
6      int review = token.nextToken(); // review is encrypted
7      int r = review * review;
8      int sq_a = sq_a + r;
9      ...
10   }
11   ...
12 }
```

The multiplication in line 7 meets the above condition, as it takes review, which is directly read from the input file. The result of the multiplication is used in the addition in line 8. Therefore, we precompute the value review * review, encrypt it using AH encryption, and add it to the input file. The while loop becomes

```
1  while (token.hasMoreTokens()) {
2    int review = token.nextToken(); // review is encrypted
3    int r = token.nextToken(); // AH−encrypted review^2
4    int sq_a = sq_a + r;
5    ...
6  }
```

Without row precomputation the def-use chain from (7,8) requires conversion from MH to AH. Since this conversion happens in an input-dependent while loop, i.e., there are potentially many executions, it would be costly. Row precomputation eliminates the need for conversion in input-dependent loops in map. In contrast to this case, the use of rating at line 10 in Fig. 2 does not meet the

```
1  int gcd(int a, int b) {
2    while (a != b) {
3      if (a > b)
4        a = a − b;
5      else
6        b = b − a;
7    }
8    return a;
9  }
```

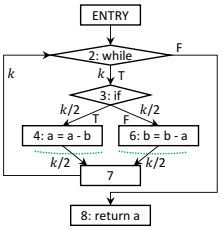


Figure 6: GCD program and the CFG. The green dotted lines show the Min-cuts for the def-use chains (4,3) and (6,3).

condition for row precomputation because operand sum_rating does not come from the input file.

4.3 Optimal Conversion Placement using Min-cut

Let c be a source as defined in Sect. 4.2. *Closure*(c) is the set of all def-use chains reachable from c:

1. Every $(c, u) \in DU$ is in *Closure*(c).
2. For every $(u, v) \in$ *Closure*(c) $\wedge$ $(v, w) \in DU$, (v, w) is in *Closure*(c).

Roughly, *Closure*(c) is the forward slice of c, i.e., it includes all CFG nodes that are data-dependent on the value of c. Therefore, the computation at these nodes must be carried out on encrypted values.

A def-use chain $(d, u) \in$ *Closure*(c) *requires conversion* if the definition d produces an encrypted value *incompatible* with the use in u. For example, def-use chain (16,21) in Fig. 2 requires conversion, because the subtraction in line 16 produces fraction in AH, however, the comparison in 21 requires OPE. Therefore, we must add conversion(s) on the paths from d to u. We must add conversion(s) so that:

(1) Every path from d to u in the CFG is "covered", i.e., regardless of which path the execution takes, the value is available at u properly encrypted, and
(2) Conversion placement is optimal, i.e., the total number of conversion executions is *minimal*.

We observe that the conversion placement problem can be cast as a Min-cut problem [5]. The Min-cut problem takes a directed graph with weights on the edges, a source node s and a sink node t. The goal is to partition the nodes into S and T such that $s \in S, t \in T$, and the sum of the weights of edges from S to T is *minimal*.

In our setting, d is s and u is t. Let $G_{(d,u)} \subseteq$ CFG be the flow network with source d and target u. As it is customary for Min-cut, $G_{(d,u)}$ contains all CFG nodes n, and edges between them, such that (1) n lies on some path from d to u and (2) d reaches n. The Min-cut on $G_{(d,u)}$ gives an optimal conversion placement that covers def-use chain (d, u). Consider Fig. 2 again. Def-use chain $(10, 14) \in$ *Closure*(rating) requires conversion. $G_{(10,14)}$ consists of nodes 8,9,10,11 and 14, and the edges between them. The Min-cut is loop exit edge (8,14) with weight m, which entails that conversion is placed right above statement 14.

To appreciate Min-cut placement, consider the naive (and most obvious) approach, which places conversion right at node u for each (d, u) chain that requires conversion. In the above example this turns out to be the optimal placement, however, one can easily see this is not the case in general. Fig. 6 shows the Euclidian algorithm for computing the greatest common divisor of two integers. Suppose a and b are encrypted *sources*. Consider def-use chain (4,3). It requires conversion from AH to OPE since 4 computes a in AH but 3 performs comparison on a. The graph on which we compute Min-cut consists of nodes 2, 3, 4, 6 and 7, and the edges between them. Clearly, edge (4,7) yields the least costly cut. Under our equal probability assumption, edge (4,7) executes half as many times as edges (7,2) and (2,3). Therefore, the optimal placement is after the assignment at 4.

In contrast, the naive approach places the conversion at the use at 3. Conversions of variable a run at the beginning of each loop iteration, even though only one of the assignments, either to a or b takes place in the previous loop iteration.

We extend this reasoning to multiple definitions and/or multiple uses of a variable as follows. Let $S_v^{e_1 \to e_2}$ be the set of all def-use chains on variable v that require conversion from encryption scheme e_1 to encryption scheme e_2. Intuitively, these def-use chains can "share" conversions. Let $G_v^{e_1 \to e_2}$ be the graph that corresponds to those def-use chains. It is formally defined as follows:

$$G_v^{e_1 \to e_2} = \bigcup_{(d,u) \in S_v^{e_1 \to e_2}} G_{(d,u)}$$

To compute the Min-cut set of edges, we extend $G_v^{e_1 \to e_2}$ with supersources and supersinks in the standard way [5]. There is a supersource s, and edges (s, d) with infinite weight for each definition d. Similarly, there is supersink t, and edges (u, t) with infinite weight for each use u.

4.4 Cost Model and Heuristic Refactoring

We develop a cost model using integer programming. It takes into account execution cost (on the client and on the server), and conversion and communication cost. Due to space constrains we have transferred the details to Appendix A.

The cost model guides our heuristic refactoring. Under certain assumptions about the model parameters, detailed in Appendix A, the objective function achieves minimum when all targets of conversion edges are on the client. To balance server placement (which we aim to maximize) and communication/messaging cost (which we aim to minimize), we use the following heuristic. We extract the minimal construct s (recall the grammar in Fig. 3) in each of map and/or reduce, that contains *all targets of conversion edges*. Intuitively, extracting a contiguous chunk of statements, rather than multiple disconnected chunks, works towards the goal of minimizing communication cost — there are message sends at the start when the server sends the arguments, and at the end, when the client sends the result to the server. On the other hand, the minimality of s works towards the goal of maximizing server placement.

Importantly, the analysis suggests a contiguous region (start line to end line). Programmers can extract the method by using the Extract Method refactoring in Eclipse (for example) fitting the extracted method into a template that includes socket programming. As mentioned earlier the analysis works top-down over the grammar in Fig. 3. It selects the minimal non-terminal s, such that s contains all conversion targets. At the top level, let there be a

sequence $s_1; s_2; \ldots; s_k$ (where each s_i is either an assignment, an IF-THEN-ELSE, or a while loop). The suggested region starts at the first s_i containing conversion target and ends with the last. If just a single s_i contains conversion, the analysis considers the three cases, as expected:

- if s_i is an assignment, the segment starts and ends with s_i.
- if s_i is IF-THEN-ELSE, and all targets are either in the THEN part or in the ELSE part, the algorithm breaks that part into a sequence and proceeds recursively. Otherwise, i.e., if some targets are in the THEN part, and others are in the ELSE part, the algorithm selects the entire s_i.
- if s_i is a WHILE statement, the algorithm allows for two choices, 1) the programmer selects the entire s_i, or 2) descends recursively into the body of the WHILE loop.

The heuristic captures well the nature of MapReduce programs. In all cases we have seen, the suggested regions is the trailing sequence of assignments at the top level. It is easy to extend the heuristic to suggest multiple segments.

In the example of Fig. 2, the conversion edges include (10,14) (for definition sumRatings at line 10), (14,15), (15,16), (16,21) and (22,27) (for definition outValue at line 22). The algorithm suggests the segment from 14 to 25. This places all targets of conversion edges on the client. [2]:

```
1  int m_client(sumRatings, totalReviews) {
2    float avgReview = sumRatings/totalReviews;
3    float absReview = Math.floor(avgReview);
4    float fraction = avgReview − absReview;
5    int limit = 1.0f/division; // division = 0.5f.
6    int i = 1;
7    float outValue = 0.0f;
8    while (i <= limit) {
9      float x = i*division;
10     if (x − division <= fraction && fraction < x) {
11       outValue = absReview + x;
12     }
13     i = i + 1;
14   }
15   return outValue;
16 }
```

The extracted method m_client runs on the client where the computation is done over plaintext. The server sends the ciphertext of sumRatings and totalReviews through the socket. The client decrypts them and passes the plaintext to m_client. After the method execution, the client encrypts the result outValue and sends it back to the server. The mapper on the server takes the encrypted value to continue the rest of the program. This partitioning entails only two message sends between server and client.

Communication has significant impact on performance. In our current setting, there is a single client machine, hence communication between multiple mappers/reducers and the single client can have significant negative impact. The impact depends on where the extracted method s takes place. There are three cases: (1) s is enclosed into an input-dependent while loop, or it contains an input dependent while loop, (2) s is a short segment of code in map, and (3) s is a short segment in reduce.

Case (1) is the worst case. It would lead to either running the entire map on the client, which would defeat the parallelism of map, or to frequent communication, which would create bottleneck at the client. Recall the original K-Means:

```
1  map(LongWritable Key, Text value) {
2    String line = value.toString();
3    String reviews = line.substring(line.indexOf(":") + 1);
4    StringTokenizer token = new StringTokenizer(reviews, ",");
5    while (token.hasMoreTokens()) {
6      int review = token.nextToken(); // review is encrypted
7      int sq_a += review * review;
8      ...
9    }
10   ...
11 }
```

The while loop 5-9 is clearly input-dependent and may execute many times per input row. There is conversion from MH to AH at line 7. If the code outside the loop (this code is elided at line 10), contains conversion, then our heuristic refactoring would extract the entire method to execute on the client, which would clearly defeat parallelism. If there is no conversion outside the loop, then the huristic would extract line 7. That would be problematic as well, because it would entail frequent communication. Fortunately, row precomputation, which we detailed at the end of Sect. 4.2 eliminates conversions in input-dependent loops.

Case (2) is when s is a short segment (a short straight line of statements) in map, and s falls outside of input-dependent loops. Our running example, Histogram Movies, illustrates this case. Each invocation of map processes one row of input. Thus, each s entails one round-trip network communication. MapReduce typically processes very large datasets and communication at each row could incur significant slowdown. In order to decrease this overhead, we leverage a technique we call "piggybacking communication". The key idea is that instead of initiating communication at the end of each input row, we store results for N rows, and initiate communication when N rows have been processed, transferring N values at once. Piggybacking reduces the number of sockets, but increases the size of each message.

Case (3) is when s is a short segment of code in reduce that falls outside of input-dependent loops. Typically, the number of reduce input groups is small. For example, there are 8 reduce input groups in Fig. 2 and 16 in K-Means. In many MapReduce benchmarks, we observed a single reduce input group. Therefore, if a conversion happens in reduce, there is a relatively small overhead of the communication cost. The impact of case (2) is substantially more pronounced because MapReduce benchmarks tend to exhibit substantially higher parallelism in the map phase than in the reduce phase.

5 EXPERIMENTS

We have implemented the Reaching definitions and program transformation using Soot [15, 27]. Reaching definitions determines how to encrypt the input columns (encryption is automated with scripts). Next, we compute the optimal placement of conversions. Once we have the placement, we apply the heuristic refactoring, which gives the client-server partitioning. Transformation of benchmarks that

[2] Strictly, the heuristic determines that 27 be placed on the client. However, it references a special heap object, outValue, which resides on the server, and therefore, 27 must remain on the server. In general, the heuristic fails if the extracted method includes heap references that cannot be moved. It does work successfully in our benchmarks with minor modification as the above.

Benchmark (suite)	Input size (GB)		Conversions	Category
	Plaintext	Ciphertext		
Word Count (PUMA)	20	53	no	search-intensive
Inverted Index (PUMA)	20	53	no	
K-Means (PUMA)	20	198.7	in map	computation-intensive
Classification (PUMA)	20	198.7	in map	
Histogram Movies (PUMA)	20	112.7	in map	
Aggregate Task (Brown)	18.7	46.8	no	
Aggregate Task Variant (Brown)	18.7	46.8	no	
Join Task (Brown)	19.6	43.5	in reduce	
L3 (PIGMIX2)	20.5	44.8	no	
L6 (PIGMIX2)	20.5	44.9	no	
L8 (PIGMIX2)	20.5	45.1	no	
L12 (PIGMIX2)	20.5	44.8	no	
L15 (PIGMIX2)	20.5	45.5	no	
L16 (PIGMIX2)	20.5	45	no	
L17 (PIGMIX2)	21.5	47.8	no	

Figure 7: Benchmarks.

do not require conversion is fully automatic. For benchmarks that do require conversion, the algorithm suggests the segment, but the actual Extract Method refactoring including fitting into boilerplate socket communication is done manually. It is standard that Extract Method is not fully automated (e.g., Eclipse requires confirmation from the user before performing Extract Method).

5.1 Execution Environment

We ran all experiments on the Google Cloud Platform [11]. The big data processing platform provides an Apache Hadoop service to create managed clusters. We created 4 separate clusters, with 8, 16, 32 and 64 nodes, and ran the MapReduce programs on them separately. Each cluster contains one master node and the remaining nodes serve as worker nodes. All 4 clusters have access to 4096 GB of disk in total. Each node is a Linux machine with 1 virtual CPU (Intel® Xeon® CPU @ 2.60GHz) and 3.75 GB of memory. The trusted client is a local Ubuntu machine (Intel® Core™2 Duo CPU @ 2.33GHz) with 4 GB of memory. The local machine is used to encrypt the input files and transform the programs into server and client partitions. It also stores the public/private keys (necessary for decryption/encryption) and the client partition (extracted from the original program to run over plaintext). We upload to the cloud the encrypted input, the server partition and the necessary public keys for homomorphic computations.

5.2 Benchmarks

We use three MapReduce benchmark suites, PUMA [1] (13 programs), Brown [17] (6 programs) and PIGMIX2 [18] (17 programs), a total of 36 programs. These suites are established in the literature, and have been used in the previous work on PHE for MapReduce [6, 24].

Based on the results of our RD analysis (Sect. 4.1.3), 4 benchmarks require conversion. Thus, 88.9% can run without communication with the client. This is a larger percentage than the one reported by MrCrypt [24] because our RD analysis allows for duplicate ciphertexts of input columns (recall example in Fig. 5). In the 4 benchmarks, 3 conversions happen in map and 1 happens in reduce, representing cases (2) and (3) from Sect. 4.4, respectively.

We classify the benchmarks into *computation-intensive* and *search-intensive*. Computation-intensive benchmarks are those that require AH and/or MH encryption according to RD. In contrast, search-intensive benchmarks require DET and/or OPE only. The overhead of AH and MH is significantly higher than the overhead of DET and OPE. Tetali et al. [24] cite specific slowdowns: 2× for DET, 4× for OP, 500× for AH, and 75× for MH. (We have confirmed these in experiments of our own.)

We focus our experiments on computation-intensive benchmarks. This is because overhead in search-intensive applications has been reported on — Tetali et. al. [24] report median slowdown of 1.57x on, essentially, the search-intensive subset of PUMA. In contrast, overhead in computation-intensive program has not been studied, to the best of our knowledge. Moreover, computation-intensive benchmarks *without* conversion (i.e., without communication) provide a baseline for benchmarks *with* conversion — roughly speaking, the true overhead of communication is the increase over a computation-intensive benchmark without conversion.

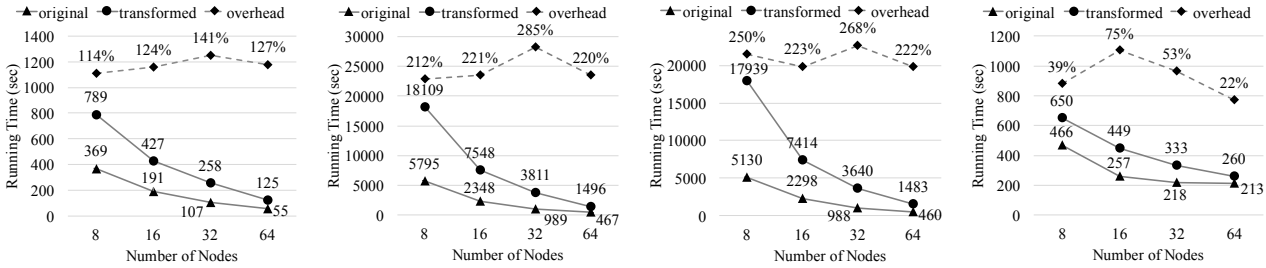
There are 13 computation-intensive benchmarks according to our RD analysis, out of 36. 4 of those require conversion as mentioned earlier. We run all 13 programs. In addition, we run 2 search-intensive programs in order to compare our overhead with that reported by MrCrypt [24]. Fig. 7 summarizes the benchmarks.

5.3 Results

Figs. 8, 9, 10 and 11 present the results. Each benchmark result is shown in a subfigure, reporting (1) the running time of the original program running on plaintext, (2) the running time of the transformed program running on ciphertext, and (3) the overhead percentage, on the 8, 16, and 32-node clusters. To better understand the impact of client communication, Fig. 8 includes results on the 64-node cluster. We use separate figures to better position our work in the context of the most related work, MrCrypt [24].

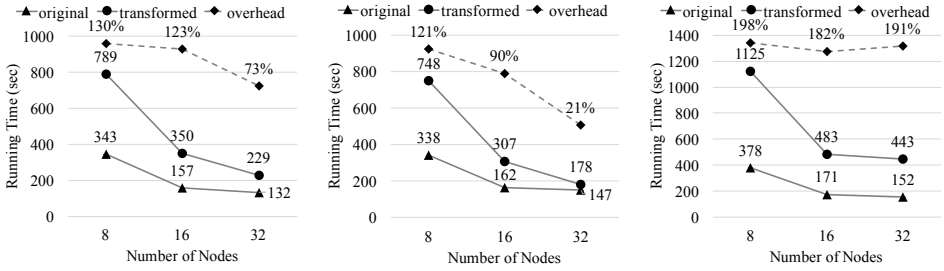
In Fig. 8, the maximal overhead is 285%, or slowdown of 3.85x, and the minimal overhead is 39%. In Figs. 8a, 8b, and 8c, conversion happens in map, while in 8d, it happens in reduce. Map in the first three benchmarks is called 540 times the reduce function in the fourth benchmark, and each function call is bound to one round trip communication with the client. Thus, we observe significantly larger overhead in Figs. 8a, 8b, and 8c than in Fig. 8d. In Fig. 8, the overhead remains relatively stable as the number of cluster nodes increases from 8 to 64. In theory, the centralized model (single client node communicates with multiple server nodes) introduces a sequential bottleneck as the number of server nodes increases. However, the overhead dose not rise on 64 nodes in the figures. The reason is that the extracted code segment running on the client is short and simple, i.e., the execution time is small. The client machine is capable of feeding the cluster with large enough number of communication requests (i.e., socket connections and plaintext computation). In other words, the time intervals between the requests are large enough for the client machine to digest the communication. This indicates the potential of scalability of our approach, that is, the overhead is low even for large number of server nodes, and the communication requests from larger number of server nodes can be handled with a very small number of client nodes (e.g., 64:1).

In Figs. 9 and 10, which show computation-intensive benchmarks without conversion, the overhead is generally smaller compared to



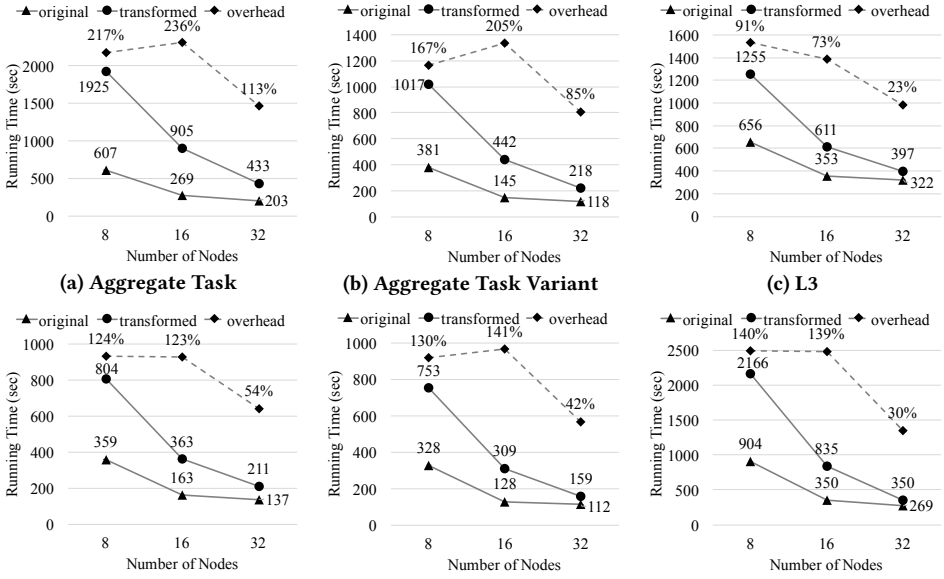
(a) Histogram Movies (b) K-Means (c) Classification (d) Join Task

Figure 8: Running time of benchmarks with conversion.



(a) L15 (b) L16 (c) L17

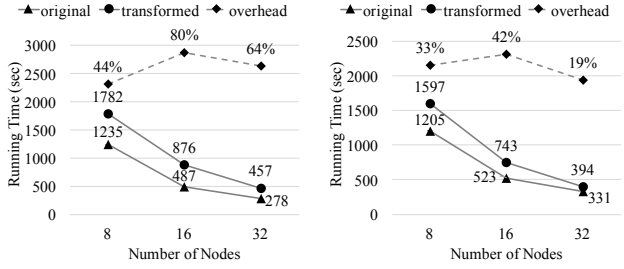
Figure 9: Benchmarks without conversion that MrCrypt's analysis cannot handle.



(a) Aggregate Task (b) Aggregate Task Variant (c) L3

(d) L6 (e) L8 (f) L12

Figure 10: Benchmarks without conversion that MrCrypt's analysis can handle, but does not evaluate experimentally.



(a) Word Count (b) Inverted Index

Figure 11: Search-intensive benchmarks that MrCrypt evaluates experimentally.

Fig. 8. They show benefits from increased parallelism, while Fig. 8 shows substantially less. The overhead in Figs. 9 and 10 decreases as the number of nodes increases. This conforms to Amdahl's law[3]. Let E_{32} denotes the running time of the encrypted program on 32 nodes, P_{32} denotes the running time of the plaintext program on 32 nodes, etc. $E_{32}/P_{32} < E_{16}/P_{16}$ because $E_{16}/E_{32} > P_{16}/P_{32}$, i.e., because encrypted programs scale better than plaintext ones. This is because the proportion of execution time that benefits from parallelization in encrypted programs is larger than the proportion in plaintext programs, where fixed costs tend to dominate much less expensive computation.

As expected the overhead on search-intensive benchmarks (Fig. 11) is comparable to the one reported by MrCrypt.

6 RELATED WORK

Partially Homomorphic Encryption (PHE) for cloud computing has received significant attention in recent years [6, 19, 23–26]. Sect. 1 and Sect. 5 already position our work with respect to the most closely related work, MrCrypt [24], and other related work, specifically CryptDB [19] and Monomi [26]. In addition, AutoCrypt [25] enables homomorphic computations on C applications. None of CryptDB, MrCrypt, AutoCrypt, or JCrypt [6] handle conversion. In contrast, SecureMR handles conversion; it covers the largest set of MapReduce programs, and assesses conversion and communication cost on a real cloud platform.

MrCrypt and JCrypt build type systems to infer encryption schemes for input columns. In contrast, SecureMR leverages a classical Reaching definitions analysis. Reaching definitions enables row precomputation and optimal conversion placement, which no work has addressed.

Program partitioning is a popular technique in various computing domains. EnerJ [21] uses program partitioning to save energy. Swift [3] partitions a web application into a server part and a client part in order to secure sensitive data. In this work, we partition MapReduce programs towards secure and efficient computation on the cloud. We adapt Swift's integer program to our problem setting.

Trying to minimize communication is well studied in the parallelization community [12, 13]. One key difference is that previous work is not concerned with security and encryption, while our work focuses on minimizing communication while preserving privacy, which introduces constraints not present in the prior work.

7 CONCLUSIONS

We presented SecureMR, a system that provides data confidentiality for MapReduce applications running on untrusted clouds. SecureMR statically analyzes MapReduce programs and infers efficient encryption schemes for input data. Furthermore, it provides optimal conversion placement, and a cost model that guides program partitioning. We evaluated SecureMR on 15 MapReduce benchmarks (13 computation-intensive and 2 search-intensive) running on the Google Cloud.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback on our work. This work was supported by NSF Award CCF-1319384. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

[3]https://en.wikipedia.org/wiki/Amdahl%27s_law

REFERENCES

[1] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and T.N. Vijaykumar. 2012. *PUMA: Purdue MapReduce Benchmarks Suite.* Technical Report. Purdue University.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[3] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure Web Applications via Automatic Partitioning. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07).* ACM, New York, NY, USA, 31–44.

[4] Michael Cooney. 2009. IBM touts encryption innovation: New technology performs calculations on encrypted data without decrypting it. *Network World* (June 2009).

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[6] Yao Dong, Ana Milanova, and Julian Dolby. 2016. JCrypt: Towards Computation over Encrypted Data. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16).* ACM, New York, NY, USA, Article 8, 12 pages.

[7] Dong, Yao and Milanova, Ana and Dolby, Julian. 2017. *SecureMR: Secure MapReduce Computation Using Homomorphic Encryption and Program Partitioning.* Technical Report.

[8] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing (STOC '09).* ACM, New York, NY, USA, 169–178.

[9] Craig Gentry. 2010. Computing Arbitrary Functions of Encrypted Data. *Commun. ACM* 53, 3 (March 2010), 97–105.

[10] Craig Gentry and Shai Halevi. 2011. Implementing Gentry's Fully-homomorphic Encryption Scheme. In *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology (EUROCRYPT'11).* Springer-Verlag, Berlin, Heidelberg, 129–148.

[11] Google Cloud Platform 2017. Cloud Dataproc. (2017). https://cloud.google.com/dataproc

[12] Manish Gupta, Edith Schonberg, and Harini Srinivasan. 1995. *A unified data-flow framework for optimizing communication.* Springer Berlin Heidelberg, Berlin, Heidelberg, 266–282.

[13] M. Gupta, E. Schonberg, and H. Srinivasan. 1996. A unified framework for optimizing communication in data-parallel programs. *IEEE Transactions on Parallel and Distributed Systems* 7, 7 (Jul 1996), 689–704.

[14] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis.* Springer Publishing Company, Incorporated.

[15] Rohan Padhye and Uday P. Khedker. 2013. Interprocedural Data Flow Analysis in Soot Using Value Contexts. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis (SOAP '13).* ACM, New York, NY, USA, 31–36.

[16] Pascal Paillier. 1999. Public-key Cryptosystems Based on Composite Degree Residuosity Classes. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT'99).* Springer-Verlag, Berlin, Heidelberg, 223–238.

[17] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. 2009. A Comparison of Approaches to Large-scale Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09).* ACM, New York, NY, USA, 165–178.

[18] Pig Mix 2013. PIGMIX2 Benchmarks. (2013). https://cwiki.apache.org/confluence/display/PIG/PigMix

[19] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11).* ACM, New York, NY, USA, 85–100.

[20] Stefan Rass and Daniel Slamanig. 2013. *Cryptography for Security and Privacy in Cloud Computing.* Artech House, Inc., Norwood, MA, USA.

[21] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-power Computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11).* ACM, New York, NY, USA, 164–174.

[22] Michael L. Scott. 2015. *Programming Language Pragmatics (2Nd Edition).* Morgan Kaufmann.

[23] Meelap Shah, Emily Stark, Raluca Ada Popa, and Nickolai Zeldovich. 2012. Language support for efficient computation over encrypted data. In *Off the*

Beaten Track Workshop: Underrepresented Problems for Programming Language Researchers. Philadelphia, PA.

[24] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd Millstein. 2013. MrCrypt: Static Analysis for Secure Cloud Computations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 271–286.

[25] Shruti Tople, Shweta Shinde, Zhaofeng Chen, and Prateek Saxena. 2013. AUTOCRYPT: Enabling Homomorphic Computation on Servers to Protect Sensitive Web Content. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 1297–1310.

[26] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. In *Proceedings of the 39th international conference on Very Large Data Bases (PVLDB'13)*. VLDB Endowment, 289–300.

[27] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–23.

A COST MODEL AND HEURISTIC REFACTORING

In this section, we define the cost model as an integer programming problem. We adapt the integer program from Chong et. al. [3] to the settings of our problem. Sect. A.1 defines the parameters of the integer program. The parameters approximate execution costs, i.e., costs to run on the server and on the client, and the communication costs, i.e., costs to send a message from the server to the client and vice versa. Sect. A.2 defines the variables and constraints, and Sect. A.3 defines the objective function.

A.1 Parameters

There are the following parameters:

(1) c_{uv} is the cost to run conversion at edge $e = (u, v)$. If optimal conversion placement converts a value at edge e (e.g., AH to OPE), then the cost is included into c_{uv}. If there are multiple conversions at e, all costs are included in c_{uv}. If there is no conversion at e, c_{uv} is 0. Recall that conversions entail a message from the server to the client, decryption, encryption, and a return message from the client to the server.

(2) p_u is the cost to run CFG node u on the *server*. This cost varies per node. For example, if u performs addition over AH-encrypted values, then p_u will be set to a large constant to account for the cost. On the other hand, if u computes over plaintext, p_u will be set to a small constant.

(3) q_u is the cost to run node u on the *client*. This is generally a large constant. The client runs u over plaintext, however, we assume high monetary cost due to the increased security requirements. Another reason for setting large cost per operation on the client is to discourage from outsourcing large segments to the client (a single machine), and thus defeating the parallelism inherent in MapReduce.

(4) m_{uv} is the cost to send a message (from the client to the server, or from the server to the client) at edge (u, v). The message cost happens when there is transition of control from the server to the client, and vice versa. Strictly, this cost depends on the size of the message and it is possible to have messages of varying length. For simplicity, we assume constant message cost.

These costs do not include node and edge weights w_u and w_{uv}. They denote the cost to run the operation at node u, or conversion/-communication at edge e, as if they were executed exactly once. Constants w_u and w_{uv} are accounted for in the objective function that we minimize.

A.2 Variables and Constraints

We follow [3] to define variables and constraints. Let variables s_u and c_u be integers in the interval $\{0, 1\}$. They denote whether node u executes on the server or on the client. $s_u = 1$ if u runs on the server, and $s_u = 0$ if u runs on the client. Analogously, $c_u = 1$ if u runs on the client, and $c_u = 0$ if u runs on the server. To enforce that each node runs either on the server or on the client, we have constraint

$$s_u + c_u \geq 1$$

Let $x_{uv} \in \{0, 1\}$ denote whether edge $e = (u, v)$ is a transition *from the client to the server*, i.e., node u is on the client but v is on the

server. $x_{uv} = 1$ if there is transition from the client to the server, and it equals 0 otherwise (including when u is on the server, v is on the client, or both are either on the client or on the server). Analogously, $y_{uv} \in \{0, 1\}$ denotes whether there is a transition *from the server to the client.* $y_{uv} = 1$ if there is such a transition, and 0 otherwise. The following constraints account for this:

$$x_{uv} \geq c_u - c_v \qquad y_{uv} \geq s_u - s_v$$

Note that if $c_u - c_v$ (or $s_u - s_v$) is -1, then x_{uv} (or y_{uv}) would be 0 because of the interval restriction: $x_{uv}, y_{uv} \in \{0, 1\}$.

A.3 Objective Function

The integer programming problem must find an assignment for variables s_u, c_u, x_{uv} and y_{uv} that satisfies the above constraints, and minimizes the cost of running the MapReduce program. The total cost is the sum of execution cost and messaging cost:

$$\sum_u (s_u \cdot w_u \cdot (p_u + \sum_{v \in pred(u)} c_{vu}) + c_u \cdot w_u \cdot q_u) \\ + \\ \sum_{e=(u,v)} (x_{uv} \cdot w_{uv} \cdot (m_{uv} - c_{uv}) + y_{uv} \cdot w_{uv} \cdot m_{uv})$$

The first summation term models execution cost. E.g., if u runs on the server, then its cost would include p_u as well as all conversions on incoming edges c_{vu}. The cost of a single run of u is multiplied by w_u, the static estimate of the number of times u executes. The second term models communication (i.e., messaging) cost. Note that we subtract c_{uv} from m_{uv} when u runs on the client but v runs on the server (i.e., when x_{uv} is 1). This is necessary because conversion cost term c_{uv} is added by the first summation term, however, it is unnecessary if the predecessor node runs on the server. In that case, the edge adds message cost alone, transferring values (usually encrypted) from the server to the client.

A.4 Heuristic Refactoring

Integer programming is NP-hard, and it is unclear whether the linear relaxation of our integer program produces an optimal integer solution. Furthermore, it is nontrivial to choose appropriate values for the parameters. Therefore, in this work, we aim to devise suitable heuristic solutions. In future work, we will study the integer program and experiment with parameter values.

Users can vary cost model parameters to fit specific hardware configurations or other requirements such as performance. A heuristic solution heavily depends on the parameters. In one scenario, performance is of highest importance. Thus, users would set $q_u \approx p_u$, while setting c_{uv} and m_{uv}, which negatively impact performance in a significant way, to large constants. Under this scenario we are likely to have larger segments of code on the client in order to minimize communication between client and server. In another scenario, keeping maximal amount of computation on the server is of highest importance. For example, it may cost a lot to run and maintain a secure client. Users would set $q_u >> p_u$ and $c_{uv} \approx p_u$. Under this scenario, we are likely to have minimal amount of computation on the client, but more frequent communication.

We assume the former scenario. Setting $q_u = p_u$ in

$$\sum_u (s_u \cdot w_u \cdot (p_u + \sum_{v \in pred(u)} c_{vu}) + c_u \cdot w_u \cdot q_u) \\ + \\ \sum_{e=(u,v)} (x_{uv} \cdot w_{uv} \cdot (m_{uv} - c_{uv}) + y_{uv} \cdot w_{uv} \cdot m_{uv})$$

yields

$$\sum_u (w_u \cdot p_u + s_u \cdot w_u \cdot \sum_{v \in pred(u)} c_{vu}) \\ + \\ \sum_{e=(u,v)} (x_{uv} \cdot w_{uv} \cdot (m_{uv} - c_{uv}) + y_{uv} \cdot w_{uv} \cdot m_{uv})$$

Assuming that we must have a server partition (i.e., it is not allowed to place everything on the client), the minimum of the above expression is achieved when the c_{uv} terms contribute 0, or equivalently, when all targets of conversion edges are on the client forcing s_u to 0. To balance server placement (which we aim to maximize) and communication/messaging cost (which we aim to minimize), we use the following heuristic. We extract the minimal construct s (recall the grammar in Fig. 3) in each of map and/or reduce, that contains *all targets of conversion edges.* We use the Extract Method refactoring to extract s into a method that executes on the client. Intuitively, extracting a contiguous chunk of statements, rather than multiple disconnected chunks, works towards the goal of minimizing communication cost — there are message sends at the start when the server sends the arguments, and at the end, when the client sends the result to the server. On the other hand, the minimality of s works towards the goal of maximizing server placement.

In the example of Fig. 2, the conversion edges include (10,14) (for definition sumRatings at line 10), (14,15), (15,16), (16,21) and (22,27) (for definition outValue at line 22). To place all targets of conversion edges on the client, we extract the segment from 14 to 25 into a client method [4]:

```
1  int m_client(sumRatings, totalReviews) {
2    float avgReview = sumRatings/totalReviews;
3    float absReview = Math.floor(avgReview);
4    float fraction = avgReview − absReview;
5    int limit = 1.0f/division; // division = 0.5f.
6    int i = 1;
7    float outValue = 0.0f;
8    while (i <= limit) {
9      float x = i*division;
10     if (x − division <= fraction && fraction < x) {
11       outValue = absReview + x;
12     }
13     i = i + 1;
14   }
15   return outValue;
16 }
```

The extracted method m_client runs on the client where the computation is done over plaintext. The server sends the ciphertext of sumRatings and totalReviews through the socket. The client decrypts them and passes the plaintext to m_client. After the method execution, the client encrypts the result outValue and sends it back to the server. The mapper on the server takes the encrypted value to continue the rest of the program. This partitioning entails only two message sends between server and client.

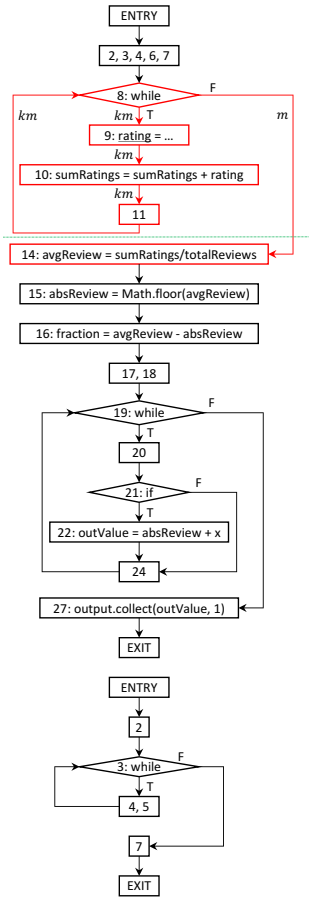
[4] Strictly, the heuristic determines that 27 be placed on the client. However, it references a special heap object, out, which resides on the server, and therefore, 27 must remain on the server. In general, the heuristic fails if the extracted method includes heap references that cannot be moved. It does work successfully in our benchmarks with minor modification as the above.

B FIG. 2. HISTOGRAM MOVIES AND ITS CONTROL-FLOW GRAPH

Reviewers may rip this page to reference the figure while going over the paper.

```
1  map(LongWritable key, Text value) {
2    int totalReviews = 0, sumRatings = 0;
3    float avgReview, absReview, fraction, outValue = 0.0f;
4    String line = value.toString();
5
6    String reviews = line.substring(line.indexOf(":") + 1);
7    StringTokenizer token = new StringTokenizer(reviews, ",");
8    while (token.hasMoreTokens()) {
9      int rating = Integer.parseInt(token.nextToken()); // rating is encrypted source
10     sumRatings = sumRatings + rating;
11     totalReviews = totalReviews + 1;
12   }
13
14   avgReview = sumRatings/totalReviews;
15   absReview = Math.floor(avgReview);
16   fraction = avgReview − absReview;
17   int limit = 1.0f/division; // division = 0.5f.
18   int i = 1;
19   while (i <= limit) {
20     float x = i*division;
21     if (x − division <= fraction && fraction < x) {
22       outValue = absReview + x;
23     }
24     i = i + 1;
25   }
26
27   output.collect(outValue, 1);
28 }
```

```
1  reduce(FloatWritable key, Iterator<...> values) {
2    int sum = 0;
3    while (values.hasNext()) {
4      int value = values.next();
5      sum += value;
6    }
7    output.collect(key, sum);
8  }
```



Histogram Movies from the PUMA benchmark set, and its Control-flow Graph (CFG). Nodes 8-14 (shown in red) comprise the flow network for def-use chain (10,14); edge (8,14), cut with a green dotted line, is the Min-cut.