SecureMCMR: Computation Outsourcing for MapReduce Applications

Lindsey Kennard and Ana Milanova
kennal@rpi.edu,milanova@cs.rpi.edu
Rensselaer Polytechnic Institute

ABSTRACT

In the last decade, cloud infrastructures such as Google Cloud and Amazon AWS have grown vastly in scale and utilization. Therefore, research into the security and confidentiality of sensitive data passed through these infrastructures is of great importance. We present SecureMCMR, a system that utilizes two public clouds for privacy preserving computation outsourcing for MapReduce applications. We also present analysis of 87 MapReduce applications and the operations they use. Our results on three MapReduce applications show overhead of 160%, 254%, and 380% over plaintext execution.

ACM Reference Format:
Lindsey Kennard and Ana Milanova. 2020. SecureMCMR: Computation Outsourcing for MapReduce Applications. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Encryption is a promising approach to privacy preserving computation outsourcing. First, users encrypt their sensitive data. Then, they transform their program to work using the corresponding encryption scheme. Finally, they upload the transformed program and encrypted data to the cloud server. The final result is that they can run their program on a public cloud without leaking sensitive data. Researchers have proposed different encryption schemes to support a variety of operations, allowing for larger and larger subsets of programs to be run this way. Fully Homomorphic Encryption (FHE) [29, 62] can perform arbitrary operations on encrypted data, but in its current state FHE is still prohibitively expensive [30]. An alternative to FHE is Partially Homomorphic Encryption (PHE), which scales substantially better, but cannot perform arbitrary operations. PHE is limited in the sense that a given encryption scheme supports only certain operations on ciphertexts. For example, Linearly Homomorphic Encryption (LHE) (e.g., the Paillier cryptosystem [48]) supports addition over ciphertexts but not multiplication or comparison. PHE and LHE have been recognized as a promising direction towards privacy preserving computation outsourcing [25, 31, 51, 58, 59, 61].

Unpublished working draft. Not for distribution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn


```
1 ...
2   while (token.hasMoreTokens()) {
3     int value = token.nextToken(); // encrypted
4     total = total + value;
5   }
6   if (total > limit) { ... }
7 ...
```

Figure 1: Data conversion. Variables value **and** total **are encrypted with LHE, allowing the addition in line 4. However, we cannot perform the comparison on line 6 over LHE-encrypted** total.

The principal problem with PHE is *data incompatibility*—when a program uses one encryption scheme (e.g., Paillier) but later requires an operation that is not supported (e.g., comparison). Fig. 1 illustrates.

One way to address the problem of data incompatibility is to maintain a *trusted machine* [25, 57, 61] that stores cryptographic keys and in some cases plaintext input data. The approach resolves data incompatibility by having the trusted machine perform conversion from one encryption scheme to another, or carry out the unsupported operations. For example, the trusted machine may receive data in one encryption scheme (e.g., Paillier), decrypt it, carry out the computation (e.g., multiplication, exponentiation, etc.), re-encrypt, and send the result back to the untrusted cloud server. Clearly, this approach entails communication between the server and the trusted machine. This communication restricts PHE in highly-parallel MapReduce applications, where multiple server nodes send work to the trusted machine and create a bottleneck.

In this paper, we propose SecureMCMR (Secure [M]ulti-[C]loud computation for [M]ap [R]educe). SecureMCMR falls into a line of work that makes use of PHE for computation outsourcing [25, 51, 57–59, 61] improving the trusted-machine approach in two ways:

(1) SecureMCMR replaces the trusted machine with an *untrusted cloud server*, thus eliminating the need for clients to maintain secure and computationally powerful trusted machines.
(2) The untrusted cloud server is *highly parallel* thus alleviating bottlenecks that arise when multiple server nodes access a single trusted machine.

The overarching problem we address is the following. Can users take advantage of inexpensive, efficient, and convenient cloud services, while (1) preserving data privacy, and (2) retaining efficiency of computation. Our specific focus is on MapReduce applications.

SecureMCMR uses two non-colluding public clouds, cloud A (e.g., Google Cloud) and cloud B (e.g., Amazon AWS). Cloud A stores public keys and runs the MapReduce task using LHE. It runs LHE-unsupported operations (e.g., multiplication) collaboratively with

cloud B, which stores private keys and can decrypt LHE-encrypted sensitive values. The key guarantee of SecureMCMR is that cloud A sees only *LHE-encrypted ciphertexts* of sensitive values, and cloud B sees only *blinded* sensitive values, which ensures that neither public cloud can retrieve sensitive input values.

We present a framework for reasoning about the security of MapReduce applications, particularly applications that use comparisons. We use the framework to classify a corpus of 87 programs. We run 3 MapReduce applications under SecureMCMR in a real cloud environment, present findings and outline future directions.

This paper makes the following contributions:

- SecureMCMR, a novel system for privacy preserving computation outsourcing for MapReduce applications.
- A new protocol inspired by Randomized Encoding that allows servers A and B to collaboratively compute certain LHE-unsupported operations securely.
- A study of the security of a large corpus of MapReduce applications. We study 87 MapReduce applications and the kinds of operations they use. Understanding of ways different operations are used in practice may guide development of new protocols.
- Results running MapReduce applications under SecureMCMR on Google Cloud (cloud A) and Amazon AWS (cloud B). The overhead of SecureMCMR decreases as the number of Amazon nodes increases, and it remains acceptable in our experiments at 380%.

The rest of the paper is organized as follows. Sect. 2 presents an overview of SecureMCMR and positions our work among related works. Sect. 3 presents the cryptographic primitives we make use of, including a description of our protocol for collaborative computation of unsupported operations. Sect. 4 presents the programming language primitives we make use of. Sect. 5 presents our security analysis, including our framework for reasoning about programs that use comparisons on top of LHE-encrypted execution. Sect. 6 details our experiments and evaluation. Sect. 7 concludes.

2 OVERVIEW AND RELATED WORK

SecureMCMR runs transformed MapReduce applications with Hadoop, thus we begin with a brief description of MapReduce. We proceed with an overview of SecureMCMR and a discussion of related work.

2.1 MapReduce

MapReduce [21] is a highly parallel programming model created to compute "big data" problems on a large cluster of nodes. Each cluster contains one or more *master* nodes along with many *worker* nodes. A MapReduce job generally consists of 3 steps: **map**, **shuffle**, and **reduce**. The framework splits the input file into chunks where each chunk is assigned to a worker node. Each worker node calls the special map function on each line of input. The **map** phase outputs a list of key-value pairs. The **shuffle** phase sorts the values associated with each key k. Finally, the **reduce** phase runs a reduce function per each key k, collapsing the list of values associated to k into a final result.

MapReduce has been implemented in different frameworks [6, 7, 24]; it is actively used in a wide variety of applications.

2.2 Overview of SecureMCMR

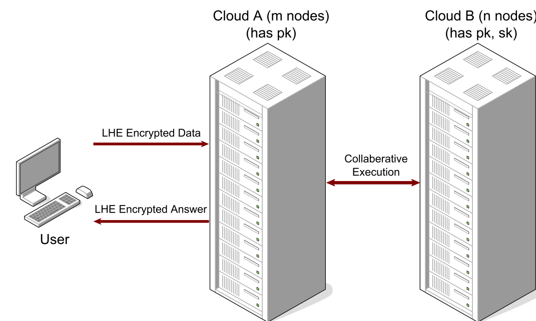

Figure 2: Multi-cloud infrastructure: Cloud A (Google) acts as the server and computes over LHE-encrypted data. Cloud B (AWS) acts in liue of the trusted machine. It receives blinded data, decrypts it, computes over, re-encrypts and sends it back to A. A and B collaboratively compute LHE-unsupported operations.

Unpublished working draft. Not for distribution.

SecureMCMR takes a MapReduce application and transforms it to work on two non-colluding, *untrusted* servers, cloud A and cloud B. Cloud A takes the input files in LHE-encrypted form, and runs LHE-supported operations locally. Whenever it encounters LHE-unsupported operations, it performs computation collaboratively with cloud B. Cloud A has access to the public key associated with the LHE scheme and cloud B has access to the public and private keys, i.e., B can decrypt the values that A sends.

Clouds A and B collaboratively compute arbitrary operations that are unsupported by LHE, however, there is varying degree of security (i.e., leakage) depending on the operation. SecureMCMR computes certain operations, e.g., multiplication, *securely*. We achieve security through a protocol inspired by Randomized Encoding, which encodes multiplications in a way that server B sees only blinded sensitive values and server A sees only LHE ciphertexts. The protocol achieves at least statistical security as we show in Sect. 5. Other operations, particularly comparisons, inherently leak *order sequences* of sensitive values, thus creating adversarial advantage [13, 14] for the servers to "guess" plaintext *values* in the sequence or *distances* between two plaintexts; a longer order sequence implies lower degree of security [13, 14]. SecureMCMR computes such operations *OPE-securely*, thus realizing the guarantees of OPE security. Yet SecureMCMR computes other operations with leakage of aggregate values, e.g., to compute e^{-x} where x is an inner product of two input vectors, it leaks the value of x to B. Computing scientific operations *securely*, *precisely*, and *efficiently* is an ongoing area of research in the secure computation community. For example, [42] considers a novel approximation of the sigmoid and [4] presents implementations and benchmarking of scientific operations based on numerical approximation in the SCALE-MAMBA system [3] for secure multi-party computation.

Across a large corpus of MapReduce programs, we find the majority can be computed securely or OPE-securely. In addition, the

nature of programs is such that OPE security comes with low adversarial advantage against guessing the exact value of an OPE ciphertext or the exact distance between two OPE ciphertexts.

2.3 Related Work

The closest related work is work on computation outsourcing for MapReduce programs. MrCrypt [58] reasons about the uses of variables in the program and assigns an encryption scheme to each variable. For example, if a variable is used only in addition operations, it assigns LHE, if it is used in only a comparison, it assigns OPE, etc. MrCrypt simply gives up on programs that exhibit data incompatibility; e.g., it cannot handle the program in Fig. 1. SecureMR [25] handles certain programs that exhibit data incompatibility using a trusted machine. Our work falls in line with MrCrypt and SecureMR and their "counterparts" in the database world, CryptDB [51], which makes use of different encryption schemes for different columns, but cannot handle incompatibility and Monomi [61], which handles data incompatibility by sending unsupported operations to a trusted machine. Our work improves upon MrCrypt and SecureMR in at least two major ways: (1) it eliminates the trusted machine, and (2) it significantly expands the security analysis. Another related work is GraphSC framework [45]. It allows non-cryptography experts to write secure versions of parallel algorithms in the areas of machine learning, data mining, and graph theory. Our work differs from GraphSC in several ways. We seek to automatically transform *existing* MapReduce programs and deploy them with Hadoop, while GraphSC is a library that allows programmers to write parallel programs in a secure way. Secondly, we use LHE as the key cryptographic primitive, while GraphSC uses garbled circuits. Finally, we deploy on Google and AWS, while GraphSC deploys on AWS using two geographic regions. Additionally, the work of Dinh et. [23] is also related to secure Hadoop computation. However it focuses on hardware solutions while our work focuses on software.

Secure Multi-party Computation (MPC) is another approach to secure computation [11, 17, 23, 32, 47, 63]. In fact, our approach was inspired in part by MPC and adapts techniques from two-party computation (2-PC). In MPC two or more mistrusting parties attempt to compute a task collaboratively, without revealing their individual inputs. In our model, clouds A and B can be viewed as mistrusting parties that collaboratively compute an operation, although the premise of collaborative computation is different. There has been significant progress in MPC and adaptations of MPC that target both classical machine learning algorithms [28, 31, 42, 46, 47, 55] and deep learning ones [35, 39].

Building programming frameworks for MPC is an active, rapidly evolving area [33]. Early research such as Fairplay [40] (and its extension to MPC FairplayMP [10]), Sharemind [12], VIFF [19], and CBMC-GC [27] were later built upon in works like CheapSMC [49], Frigate [43], and ABY [22]. Those were then extended by newer compilers including HyCC [15], EzPC [16], SCALE-MAMBA (formerly SPDZ) [36], and ABY3 [41]. Unfortunately, none of these systems directly target Hadoop-like frameworks and cannot be used for the analysis, transformation and deployment of Hadoop MapReduce applications. In addition, MPC compilers (including ABY, HyCC, and EzPC) restrict loop bounds to constant values to enhance security. Thus, they cannot be applied on our benchmark programs with input dependent loop bounds.

Our system can be viewed as a variant of two-party computation (2-PC). The major difference is that SecureMCMR allows for the automatic transformation of existing MapReduce programs — we can replace LHE-supported operations with the equivalent operations over LHE-ciphertexts (e.g. plaintext $a = b + c$ becomes $LHE(a) = LHE(b) \oplus LHE(c)$), and LHE-unsupported operations with the corresponding collaborative protocol (if there is one). The program runs in Hadoop on cloud A with minimal synchronous communication with B. In contrast, existing 2-PC compilers (e.g., ABY, HyCC) do not support Hadoop. Extending such compilers to run in Hadoop is a significant research and engineering task; furthermore, automatic transformation of MapReduce to the inputs that would be accepted by those systems is unresolved as well. Despite the current state of MPC implementation, we believe that MPC is the most promising approach to secure computation, including in the parallel domain, due to its strong security guarantees. Our results stress the importance of parallelism in the two cloud systems, and study the kinds of operations used in real world MapReduce programs, which may guide development of MPC compilers and protocols for the MapReduce domain.

3 CRYPTOGRAPHIC PRIMITIVES

This section describes the cryptographic tools we make use of, Linearly Homomorphic Encryption (Sect. 3.1), Randomized Encoding (Sect. 3.2), and Order Preserving Encryption (Sect. 3.3). Sect. 3.2 details our adaptation of RE for the purposes of SecureMCMR.

3.1 Linearly Homomorphic Encryption (LHE)

Given a message space M, an LHE scheme is defined, per [31], by:

(1) Algorithm *Gen* generates a pair of keys, a *private key sk* and a *public key pk* given a security parameter κ: $(sk, pk) \longleftarrow Gen(\kappa)$
(2) Encryption algorithm *Enc* takes $m \in M$ and the public key, and generates ciphertext c: $c \longleftarrow Enc(m, pk)$
(3) Decryption algorithm *Dec* is a deterministic algorithm that takes c and the private key and decrypts the plaintext message corresponding to c: $m = Dec(c, sk)$
(4) There is a homomorphic operation that operates on ciphertexts: $Dec(Enc(m_1, pk) \oplus Enc(m_2, pk), sk) = m_1 + m_2$

As it is standard, the LHE plaintexts are in $\mathbb{Z}_N$ where N is an RSA modulus. Negative numbers are represented by the upper half of this range:

$$\left[\lceil \frac{n}{2} \rceil, n-1\right] \equiv \left[-\lfloor \frac{n}{2} \rfloor, -1\right]$$

We assume that N is sufficiently large and computation does not overflow the modulus.

There are several known instantiations of LHE, e.g., Paillier [48], Damgård and Jurik [20], and Damgård, Geisler, and Krøigaard (DGK) [18] Our framework uses a Java implementation of Paillier [5]. LHEs follows the standard semantic security property, which states, informally, that a computationally bounded algorithm cannot gain additional information about m given only the public key pk and the ciphertext c of m.

Unpublished working draft. Not for distribution.

The key takeaway for our purposes is that LHE performs arbitrary affine transformations on ciphertexts. We can compute (the ciphertext of) $m_1 \cdot c_1 + m_2 \cdot c_2 + m_3$, where m_1, m_2 and m_3 are plaintexts and c_1 and c_2 are ciphertexts. Therefore, all affine transformations on program inputs can be computed locally on Server A. We found that a significant amount of computation in the MapReduce applications in our corpus was affine operations.

3.2 Randomized Encoding (RE)

3.2.1 Standard RE. Randomized Encoding, described in [9], works as follows. Let $f(x)$ be a function. RE introduces an *encoding function* $\hat{f}(x;r)$, where r is uniformly random input, and a *decryption algorithm*, *Decode*. REs enforce two key properties: (1) correctness: $Decode(\hat{f}(x;r)) = f(x)$, and (2) privacy: the distribution of $\hat{f}(x;r)$ depends only on $f(x)$, i.e., it reveals only $f(x)$, and does not reveal any additional information on x. An additional property, (3) efficiency, states that computing $\hat{f}(x;r)$ is more efficient than computing $f(x)$.

An example from [9] is $f(x) = x^2 \ mod \ N$, where x is private input and N is a public integer. Take $\hat{f}(x;r) = x^2 + r \cdot N$, and $Decode(\hat{f}(x;r)) = \hat{f}(x;r) \ mod \ N$. Clearly, the randomized encoding is correct because $Decode = (x^2 + r \cdot N) \ mod \ N = x^2 \ mod \ N$. The encoding is also private, as argued by Appelbaum; this is because the distributions $\hat{f}(x;r)$ and $\hat{f}(x';r)$ where $f(x) = f(x') = y$ (i.e., $x^2 = q \cdot N + y$, and $(x')^2 = q' \cdot N + y$), are statistically the same (intuitively, choosing r or r', s.t., $q \cdot N + y + (r - q) \cdot N = q' \cdot N + y + (r' - q') \cdot N$, is equally probable).

In traditional RE one party, say party A, may send the result of $\hat{f}(x;r)$, in our example this is the value $x^2 + r \cdot N$, to another party, say party B. Party B then retrieves the value of $f(x)$, by applying algorithm *Decode*. Party B will not learn anything additional on x (besides $x^2 \ mod \ N$, of course), which is guaranteed by privacy. And party A computes $\hat{f}$ more efficiently than f because modular division is expensive. Crucially, in traditional RE, both parties A and B can perform arbitrary operations and compute arbitrary functions. Therefore, neither the encryption (the encoding algorithm) nor the decryption (decoding algorithm) are constrained. In contrast, in our system *encoding of input* and *decoding of output* are constrained to only affine transformations. We build on RE, but impose different constraints.

3.2.2 SecureMCMR RE. Our variation of RE, which we call SecureMCMR RE, defines privacy as follows. For the rest of this paper, when we write RE, we refer to SecureMCMR RE. We require $\hat{f}(e(x;r))$, where $e(x;r)$ is an encoding of the input x such that it does not reveal anything on x. The inputs x are of bit length l, and the random numbers r are drawn from the integers of bit length $\sigma + l$, where σ is a security parameter. It is an onus on the protocol designer to show that $e(x;r)$ is secure as we demonstrate for the two protocols we fit in this framework (recall that $e(x;r)$ is a ciphertext on the side of A, but a plaintext on the side of B). SecureMCMR RE defines correctness as follows: $Decode(\hat{f}(e(x;r)), x, r) = f(x)$; the decoding operation *Decode* takes $\hat{f}(e(x;r))$, x, and r and produces the result of $f(x)$. The difference with traditional RE is that the decoder depends on x and the random inputs r used to encode x. The motivation is our setup where Server A performs both the encoding and decoding, x is an LHE-encrypted ciphertext and r is a random plaintext. Server B computes $\hat{f}(e(x;r))$ over blinded inputs, thus $\hat{f}$ can perform arbitrary operations.

Encoding of multiplication. We consider multiplication $f(x, y) = x \cdot y$. The encoding $\hat{f}(e(x;r_1), e(y;r_2)) = (x + r_1) \cdot (y + r_2) = v$. $Decode(v, x, y, r1, r2) = v - x \cdot r_2 - y \cdot r_1 - r_1 \cdot r_2$. Privacy holds, as $x + r_1$ and $y + r_2$ blind the values of x and y, and $(x + r_1) \cdot (y + r_2)$ does not reveal any additional information about x or y. Correctness holds as well, since $(x + r_1) \cdot (y + r_2) - x \cdot r_2 - y \cdot r_1 - r_1 \cdot r_2 = x \cdot y$.

To establish privacy, we show that $e(x;r)$ is at least statistically secure. We argue that this protocol is statistically secure following the proofs of security in [22, 53]. We assume that the Paillier modulus N is large enough and computations do not overflow N; this is convenient for correctness as $(x + r_1) \cdot (y + r_2)$ does not overflow the modulus and can be viewed as computation over the integers; however it limits the security of blinding, as now $x + r$ may reveal the length of x when B decrypts $x + r$. Consider the view of cloud B. The value $x + r$ may leak information about the length of x if its length is shorter than l, where l is the length of the input. The standard approach [22, 53] is to use a statistical parameter σ, i.e., add σ random bits by randomly selecting padding r of bit length $\sigma + l$. The only way $x + r$ reveals information about the length of x is if the σ most significant bits of r are 0, which happens with probability $2^{-\sigma}$; this is negligible in the security parameter σ. Therefore, $e(x;r)$ as applied to x and y is statistically secure. From the point of view of server A, A receives the Paillier encrypted value $(x + r_1) \cdot (y + r_2)$ which is indistinguishable from any other encrypted value by the guarantees of the Paillier cryptosystem.

We now plug in SecureMCMR RE into our system. SecureMCMR RE defines a protocol to compute LHE-unsupported function $f(x)$. It is parameterized by $\hat{f}$, $e(x, r)$, and *Decode*, meeting the above requirements.

1. Server A generates random r and computes LHE ciphertext $c = e(x;r)$. Recall that the computation uses only affine transformations on x as A holds the LHE ciphertext of x.
2. Server A sends c to B.
3. Server B decrypts c and computes $m = \hat{f}(Dec(c))$.
4. Server B encrypts $c = Enc(m)$ and sends the LHE ciphertext c to A.
5. Server A decodes $\hat{f}(c)$, computing the ciphertext of $f(x)$.

Server B receives *only* input encodings $e(x;r)$, in ciphertext. It is a requirement that $e(x;r)$ is secure, and the protocol designer must establish security of $e(x;r)$ as we did for the multiplication protocol. Server A receives $\hat{f}$ as LHE ciphertext which is secure based on the security guarantees of LHE.

As it is well known, we can approximate any function $f(x)$ by the first n terms of its Taylor series expansion; A and B can collaboratively compute n terms, $f''(0) \cdot x^2/2!$, $f'''(0) \cdot x^3/3!$, etc., running steps 1-5 n times or more. At the end, server A sums up the resulting ciphertexts to compute $f(x)$. A downside of this is the inefficiency that n communication rounds entail. Therefore, we define our requirement for *efficiency* as a single round of communication, where A and B each send at most a small constant number of values. Operations are computable in our framework when we

Unpublished working draft. Not for distribution.

can design a protocol that fits the security and efficiency requirements; operations are *Unknown* otherwise. Sect. 5 has additional discussion.

One can use the framework to encode different operations. We have constructed encodings for multiplication (described above) and comparison $x \leq y$ (described below), which are necessary to run our benchmarks; we envision encoding of other operations in future work.

Encoding of comparison. We adapt a protocol based on multiplication hiding described by Kerschbaum et al. [38]. We expand the protocol to suit our needs and argue both correctness and security (there are no proofs in ref. [38]). Recall that server A wishes to compute $x \leq y$, however, A holds both x and y as LHE-encrypted values. First, Server A computes the ciphertext $d = y - x$. (Note that computing $Enc(x)$ in either Paillier or Damgard and Jurik amounts to computing the multiplicative inverse of $Enc(x)$ in $\mathbb{Z}_{N^2}$.) Then A generates a large uniformly random integers r and r' of length $\sigma + l$ such that $r' < r$ and computes $r \cdot d + r'$. We assume the existence of such random values r and r', as in [38], that the distribution of $r \cdot d + r'$ is statistically close to uniform over the set of values of d.

Server A sends the ciphertext $r \cdot d + r'$ to B.[1] B decrypts to a plaintext value v and sends True if $v > 0$ and False otherwise. In terms of SecureMCMR RE

$$\hat{f}_{x \leq y}(e(y - x; r, r')) = \begin{cases} True & \text{if } r \cdot (y - x) + r' > 0 \\ False & \text{otherwise} \end{cases}$$

$$Decode_{x \leq y}(v, x, y, r, r') = v$$

The protocol is correct: $x \leq y$ if and only if $r \cdot (y - x) + r' > 0$. Correctness follows from the assumption that the Paillier modulus N is large and computations $r \cdot d + r'$ do not overflow the modulus. Typical bit length values are $l = 32$, $\sigma = 112$, and N is 2048 bits, at least [53]. Consider the case $d < 0$ (d is negative in the integers). In Paillier, d is represented by $N - d'$ where $d' = |d|$, and $r \cdot (N - d')\ mod\ N = -r \cdot d'$; given the large N, $-r \cdot d'$ is in the upper half of the modulus N. Since $r' < r$ and d' is a positive integer, it follows that $-r \cdot d' + r'$ is in the upper half of the modulus, i.e., it is a negative integer.

To prove security of $e(y - x; r, r')$ we again use the statistical parameter σ. $r \cdot d + r'$ reveals information to server B about the length of d only if the length of $r \cdot d + r'$ is less than $l + 1$. In order to have such a length, r must be 1 (and r' must be 0). The probability of selecting a random $r = 1$ is $\frac{1}{2^{\sigma + l}}$ which entails that hiding is at least statistically secure.

Encoding the rest of the comparison operations in terms of $x \leq y$ is straightforward:

$$\begin{aligned} x < y &\equiv \neg(y \leq x) \\ x == y &\equiv (x \leq y) \text{ AND } (y \leq x) \\ x \neq y &\equiv \neg(x \leq y) \text{ OR } \neg(y \leq x) \end{aligned}$$

3.3 Order Preserving Encryption (OPE)

The last cryptographic primitive that factors in our system and analysis is Order Preserving Encryption (OPE) [13, 14, 37]. An OPE scheme is a symmetric encryption scheme over a key space, message space M, and ciphertext space C. It is defined in terms of three algorithms:

- A key generation algorithm that returns a key K
- An encryption algorithm *Enc* that takes a key K and a plaintext m and returns the ciphertext c
- A decrpyption algorithm *Dec* that takes K and a ciphertext and returns a plaintext

The correctness property $Dec(K, Enc(K, m)) = m$ holds for every K in the key space, and $m \in M$. The order-preserving property, if $m_1 < m_2$ then $Enc(K, m_1) < Enc(K, m_2)$, holds for every K and $m_1, m_2 \in M$.

Boldyreva et al. [13, 14] study OPE and its security properties. It is well-known that since OPE schemes reveal order of plaintexts, their security is weaker than LHE or RE. Boldyreva et al. cast an OPE scheme in terms of a Random Order Preserving Function (ROPF), and describe the "ideal" behavior of an OPE scheme:

- Key generation picks a ROPF g
- *Enc* takes the key and a plaintext m and returns $g(m)$
- *Dec* takes the key and a ciphertext c and returns $g^{-1}(c)$

A secure OPE scheme, Boldyreva et al. argue, should closely "mimic" the behavior of a ROPF. In addition to the characterization of the "ideal" behavior, they propose an OPE scheme that is secure according to this definition.

Boldyreva et al. [14] give upper and lower bounds on *Window One-Wayness (WOW)*, a metric of the advantage of an adversary A trying to guess the plaintext values of OPE ciphertexts. Assuming an ideal OPE scheme, the upper bound is as follows:

$$\mathbf{Adv}^{1, z-WOW}(A) < \frac{4z}{\sqrt{M - z + 1}}$$

where M is the size of the domain of the plaintext, and z is the number of OPE challenge ciphertexts the adversary sees. $\mathbf{Adv}^{1, z-WOW}(A)$ in particular is the probability that A will find the exact value of at least one of the z challenge ciphertexts. Intuitively, the larger z is, i.e., the higher the number of *ordered* ciphertexts adversary A sees, the higher A's advantage, and hence the lower overall security.

Boldyreva et al. also present bounds on *Window Distance One-Wayness (WDOW)* a metric of the adversarial advantage of A in guessing the distance between two OPE ciphertexts. Again assuming an ideal OPE scheme the upper bond is defined as:

$$\mathbf{Adv}^{1, z-WDOW}(A) < \frac{4z(z - 1)}{\sqrt{M - z + 1}}$$

As shown, a larger z increases A's advantage by a factor of $(1 - z)$x over guessing individual values. While the *exact* distance value is secure the larger the size of the window (i.e. the number of ciphertexts seen) the smaller the domain of possible values becomes.

A key contribution of our work is to apply the upper bound on adversarial advantage on real-world MapReduce programs and study their OPE security, substantially expanding prior work [25, 52, 58, 61] that has used OPE (with weak security guarantees).

4 PROGRAMMING PRIMITIVES

This section describes the program execution semantics (Sect. 4.1) and static analysis (Sect. 4.2) we use to reason about the security

[1] The protocol assumes that x and y are integers, however, it is trivially adapted to work over fixpoint representation of real numbers as in the Java implementation of Paillier we use.

Unpublished working draft. Not for distribution.

$$
\begin{array}{lll}
s & ::= s; s & \\
 & \mid \mathsf{x} = \mathsf{y} & \\
 & \mid \mathsf{x} = \mathsf{y}\ aop\ \mathsf{z} & \\
 & \mid \text{while } (c)\ \{\ s\ \} & \\
 & \mid \text{if } (c)\ \{\ s\ \} \text{ else } \{\ s\ \} & \textit{statement} \\
c & ::= \neg c \mid c \wedge c \mid c \vee c \mid \mathsf{x}\ bop\ \mathsf{y} & \textit{logical formula} \\
aop & ::= + \mid - \mid * \mid / & \textit{arithmetic operator} \\
bop & ::= == \mid != \mid < \mid \leq & \textit{comparison operator}
\end{array}
$$

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.**

levels of MapReduce applications, and to help transform them to work on the cloud.

4.1 Program Execution Semantics

```
1  int b0, b1, b2, b3, b4, b5; // LHE−encrypted inputs (constants)
2  map(LongWritable key, Text value) {
3    int total = 0, sumRatings = 0, outValue = 0;
4    ...
5    while (token.hasMoreTokens()) {
6      int rating = token.nextToken(); // rating is LHE−encrypted input
7      sumRatings = sumRatings + rating;
8      total = total + 1;
9    }
10   if (total*b0 <= sumRatings && sumRatings <= total*b1)
11     outValue = 1;
12   else if (total*b1 <= sumRatings && sumRatings <= total*b2)
13     outValue = 2;
14   else if (total*b2 <= sumRatings && sumRatings <= total*b3)
15     outValue = 3;
16   else if (total*b3 <= sumRatings && sumRatings <= total*b4)
17     outValue = 4;
18   else if (total*b4 <= sumRatings && sumRatings <= total*b5)
19     outValue = 5;
20
21   output(outValue,1);
22 }
```

```
1  reduce(IntWritable 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 4: Histogram, adapted from PUMA, computes a histogram of ratings. The bucket interval boundaries b0, b1, ... **and the** rating **values are LHE-encrypted inputs.**

We assume program syntax as shown in Fig. 3. Next, we define the notions of the *AST value* and *program trace*; these are mostly standard concepts in programming languages, which we present in order to formalize our security model and security guarantees.

4.1.1 AST values. We denote values as $AST(v_1, \ldots, v_n)$, where $v_1, \ldots, v_n$ and $n \geq 1$ are *inputs*, which can be either (1) *program constants* or (2) *values read from MapReduce input files.* Inputs are plaintext or sensitive, where sensitive ones are LHE-encrypted by the client that outsources computation. For example, b0, b1, ... b5 in Fig. 4 are sensitive program constants, and rating's are sensitive values read from MapReduce input file. Inputs are the leaves of the Abstract Syntax Tree (AST) that computes the actual value. Our notation hints at the AST, however, we are interested in the inputs, and not the structure of the AST. For example, consider the value b0*total in line 13 in Fig. 4. Its AST notation will be AST(b0,1,...,1), as it is computed from inputs b0 and the constant 1 in line 10. As a remark $v_1, \ldots, v_n$ may repeat inputs, as in the example AST(b0,1,...,1).

AST($v_1, \ldots, v_n$) is plaintext if all inputs v_i's are plaintext; it is sensitive otherwise, i.e., there is at least one input that is sensitive. For example the AST of total, namely AST(1,...,1) is plaintext, and the AST of b0*total is sensitive. A sensitive AST must be LHE-encrypted on Server A, and blinded on Server B; this is the correctness invariant enforced by our system.

4.1.2 Traces. An execution *trace* is the sequence of statements (x = y, x = y aop z in Fig. 3) and tests (x bop y in Fig. 3)) that the program executes for a given input. For example, below is a trace for Fig. 4 that sums up the ratings in the first line of the input file and the average rating falls into the first bucket:

$$\mathsf{s1} = \mathsf{s0} + \mathsf{r}; \mathsf{t1} = \mathsf{t0} + 1; \ldots (\mathsf{b0} * \mathsf{tN} \leq \mathsf{sN}) \wedge (\mathsf{sN} \leq \mathsf{b1} * \mathsf{tN}); \mathsf{out} = 1;$$

Here s0, s1,... stand for the partial sumRatings and r0, r1,... stand for the sequence of rating inputs, and t0, t1,... stand for the partial total.

Consider the trace view of the program: the same trace is running on A and on B. All sensitive ASTs must be "encrypted", i.e., available as LHE ciphertexts on A, and as blinded plaintexts on B. The trace view makes no distinction between arithmetic operations and comparison operations—each operation in the trace sequence is either (1) locally computable on Server A, (2) RE-enabled through communication between A and B, or (3) "unknown". We elaborate on "unknown" operations in Sect. 5.

4.1.3 Path conditions. Note that a trace reveals the results of all comparisons to both servers A and B. The full sequence of comparison clauses in a trace T forms the *path condition* of T. More precisely, we consider atomic formulas are either $x \leq y$, which we denote by c, or $\neg(x \leq y)$, which we denote by $\neg c$. An individual comparison clause can be a conjunctions of disjunctions of c's and $\neg c'$, as shown in the grammar in Fig. 3. The path condition of a trace T for our purposes is the sequence of all comparisons. For example, the path condition for the running example in Fig. 4 and the trace we showed earlier is $(\mathsf{b0} * \mathsf{total} <= \mathsf{sumRatings}), (\mathsf{sumRatings} < \mathsf{b1} * \mathsf{total})$.

In Sect. 5 we use the notions of the trace view and path condition to reason about the security of programs with comparisons.

4.2 Taint Analysis

We describe the taint analysis that determines whether a variable x stores a plaintext AST or a sensitive AST. This is just a standard *may* dataflow analysis [2]. The simplicity of MapReduce programs allows us to largely elide issues like aliasing and carry out a highly precise flow- and context-sensitive taint analysis.

Recall that v_i denote the inputs, either (1) program constants, or (2) inputs read from data files. For each input there is a corresponding variable, which we also call v_i. For example, in Fig. 4 the

Unpublished working draft. Not for distribution.

inputs are b0, b1, b2, b3, b4, b5 and rating. The analysis tracks the flow of the inputs throughout the program, and computes, for each variable x the set of inputs I, such that at some point of program execution x may hold an AST(...v_i...) s.t. $v_i \in I$.

The analysis propagates maps $\sigma : x \in V \rightarrow 2^V$, where V is the set of all variables in the program. Sets for all *sensitive input* variables v_i are initializes to $\{v_i\}$, and sets for the remaining variables $x \in V$ are initialized to the empty set $\{\}$. The transfer functions are standard; they propagate the sensitive inputs v_i:

$$i: f_{x=y \text{ aop } z}(\sigma) = \sigma[x \leftarrow \sigma(y) \cup \sigma(z)]$$

What the rules entail is that if one of the operands is a sensitive AST(...,v_i,...), then x is a sensitive AST(...,v_i,...) as well. As it is standard, the merge operator at control merge nodes to set union: $\cup$. That is, per-variable sets are unioned pointwise.

The taint analysis determines whether x at program point p *may* hold an AST formed of sensitive inputs, i.e., whether x is a sensitive ASTs. We use the analysis to classify the corpus of benchmarks into security categories, depending on the kinds of operations performed on sensitive ASTs.

5 SECURITY ANALYSIS

This section presents the security analysis of SecureMCMR. We begin with stating our assumptions (Sect. 5.1). We describe the different kinds of program operations (Sect. 5.2) that give rise to the three security levels: Secure, OPE-secure, and Unknown (Sect. 5.3).

5.1 Assumptions

The goal of SecureMCMR is to distribute computation among two untrusted public clouds, cloud A (which runs the program over LHE-encrypted values) and cloud B (which helps run LHE-unsupported operations). Cloud A holds the public key pk as well as LHE-ciphertexts of sensitive ASTs. Cloud B holds the private key sk, however, it receives only statistically blinded values.

Our assumptions are:

1. Cloud A and cloud B are non-colluding, i.e., A has no access to B and vice versa.
2. We assume a passive adversary, e.g., an administrator at either cloud can monitor memory and traffic but would not attempt active attacks.
3. Each program input v_i is either a signed integer $-\frac{M}{2} \leq v_i \leq \frac{M}{2} - 1$, or unsigned integer $0 \leq v_i \leq M-1$, for some positive even integer M.
4. We assume that servers A and B consider M sufficiently large, i.e., bounds on adversary advantage due to order information are driven by z.

We allow that the program code is known to both cloud A and cloud B. We may state an assumption that the program is *not known* to cloud B, which will strengthen our security guarantees. (In this case we will be able to send *any* value from A to B, as B does not know anything about the structure of the AST.) In practice however, cloud B may infer the program from the set of LHE-unsupported operations it is asked to perform, particularly, given that MapReduce applications typically run well-known data analytics tasks. Allowing that B knows the program code, entails that B has the exact same trace view as A. Both A and B see the AST of each value.

In addition, since both A and B observe the execution trace, they observe the number of iterations each loop takes. Thus, they observe the size of container structures. As expected, knowledge of the exact number of executions a loop can create side-channel leaks when the loop-bound is input dependent. We examine such leaks in our benchmarks in detail.

5.2 Kinds of operations

We group operations on sensitive ASTs into 4 categories. Here x and y are ciphertexts and p is plaintext:

1. LHE-supported arithmetic (LHE): $x + y$ and $p \cdot x$,
2. RE-supported arithmetic (RE): $x \cdot y$
3. Comparison (CMP): $x \leq y$, and
4. Unsupported operations (UNK): any other operation, e.g., e^x.

LHE operations are carried out locally on Server A. RE ones are carried collaboratively by Servers A and B; as we argue in Sect. 3.2.2, RE operations do not leak additional information about the operands or result to either A or B. Comparisons are carried locally on A using OPE, or remotely using the encoding in Sect. 3.2.2. We classify a programs into one of three security levels, depending on the kinds of operations it executes. If the static analysis determines that sensitive inputs reach a LHE, RE, CMP, or UNK operation, then we say that the program executes LHE, RE, CMP, or UNK operations, respectively.

5.3 Security levels

5.3.1 Secure. Secure applications execute only LHE and RE operations. Server A sees only LHE-encrypted ciphertext of sensitive ASTs, and therefore SecureMCMR w.r.t. Server A is as strong as LHE encryption (i.e., we have semantic security). Server B sees statistically blinded sensitive ASTs, and thus, SecureMCMR w.r.t. Server B is at least statistically secure.

5.3.2 OPE-secure. OPE-secure applications execute CMP operations (i.e., comparisons) in addition to LHE and RE ones. Comparisons present the biggest challenge to security reasoning. There are two cases, a local CMP and a remote CMP. Local CMPs have two operands that are inputs v_i (rather than ASTs of two or more leaves). Remote CMPs are ones where at least one operand is a sensitive AST. In other words, local CMPs use inputs as is, while remote CMPs perform computation/transformation on inputs. For example, the comparison in line 3 in Fig. 5 is a local CMP, as both the entryDate column value, and the constant input date are inputs; the comparisons in lines 10,12,14,16, and 18 in Fig. 4 are remote CMPs.

CMP-local. We first discuss security of local CMPs, significantly expanding discussion of OPE compared to previous work [25, 52, 58, 61].

Our analysis applies Boldyreva et al's results to reason about OPE security (and order information more generally) in real programs. The actual bounds on adversary advantage depend on what order-preserving operations are executed in a program. If a program uses a large number of distinct OPE-encrypted ciphertexts, thus entailing a large z (recall Sect. 3.3), then the bound on adversary advantage is high and the program's OPE security is low. Conversely, if it uses

Unpublished working draft. Not for distribution.

Benchmark	z	Can re-key?	Additional Notes
L14	2	yes	character comparison
Benchmark3 Phase 1	4	yes	date comparison
Benchmark3 Phase 3	3	yes	
RankedInvertedIndex	O(n)	no	$sort()$ call
AdjList	O(n)	no	$sort()$ call
Mutual-Friends	O(m)	yes†	$sort()$ call
ConCmptIVGen	O(m)	yes†	bounded looping
ConCmpt	O(n)	no	$min()$ call
ConCmptBlock	O(n)	no	$min()$ call
q1	1	yes	date comparison
q3	2	yes	date comparison
q4	2	yes	date comparison
q5	2	yes	date comparison
q6	5	yes	date comparison
q7	2	yes	date comparison
q10	2	yes	date comparison
q12	4	yes	date comparison
q15	O(n)	no	$max()$ call
q20	2	yes	date comparison

Table 1: Concrete z values for CMP-local benchmarks. $O(n)$ means that the number of comparisons is of the order of the number of input values (m is the number of items in a line of input). Rekeying reduced z to a small constant (as in the example in Sect. 5.3.2). (†: Indicates that the problem can only be rekeyed with a *single* key per map or reduce call.)

Unpublished working draft.
Not for distribution.

```
1  int date; // OPE−encrypted input (constant)
2  Integer map(Integer entryDate , Integer caloriesBurnt) {
3    if ( entryDate > date)
4      return caloriesBurnt ;
5    else
6      return 0;
7  }
8  Integer reduce ( List<Integer> caloriesBurntList ) {
9    Integer sum = 0;
10   for ( Integer caloriesBurnt : caloriesBurntList )
11     sum += caloriesBurnt ;
12   return sum ;
13 }
```

Figure 5: A MapReduce-like program adapted from MrCrypt.

only a few OPE-encrypted ciphertexts, then the bound on adversary advantage is low and OPE security is high.

Consider Fig. 5. Applying an OPE encryption scheme entails that all v_i's in column entryDate are encrypted using a key K. In terms of Boldyreva's analysis K is, essentially, a randomly chosen Order Preserving Function (ROPF) (recall Sect. 3.3). Thus, the bounds on the adversary advantage, depend on the size of the input file. As input files are large, that means OPE security is low — the adversary can guess the plaintext of a ciphertext with probability very close to one.

A mitigation is to encrypt each record, i.e., each entryDate with a different random key, thus associating a different random OPF with each value (i.e., rekey). This necessitates encryption of the date constant with different keys as well, and the comparison becomes

$$\text{if}(\text{entryDate}_i > \text{date}[\text{key}_i])\ldots$$

```
1 int start_node = Integer.parseInt(line[0]);
2 int end_node = Integer.parseInt(line[1]);
3 for(i = start_node; i <= end_node; i++) {
4   from_node_int.set( i );
5   output.collect(from_node_int, new Text("v"+initial_weight));
6 }
```

Figure 6: Interval leak example.

z however is 2, since the adversary sees exactly 2 ciphertext per random function, and the bound becomes

$$\mathbf{Adv}^{1,z-WOW}(A) < \frac{4z}{\sqrt{M-z+1}} = \frac{8}{\sqrt{M-1}}$$

For the distance advantage we also get:

$$\mathbf{Adv}^{1,z-WDOW}(A) < \frac{4z(z-1)}{\sqrt{M-z+1}} = \frac{8}{\sqrt{M-1}}$$

In Section Sect. 6 we present analysis of the adversary advantage bounds in our corpus of benchmarks.

CMP-remote. Remote comparisons present a challenge because sequences of such comparisons combined with knowledge of the AST of operands can lead to side channel leaks. Recall that remote comparisons happen when at least one of the operands of the comparison is a result of a computation. Many existing works (e.g., MrCrypt and CryptDB) do not support such programs.

To reason about CMP-remote operations we define the notion of the *interval leak*:

Definition 1. *Consider an execution trace with corresponding path condition P. An interval leak occurs when $P \Rightarrow c_l \leq v_i$ or $P \Rightarrow v_i \leq c_u$, where c_l and c_u are plaintext constants, v_i is a sensitive input, and the implied interval is strictly included in v_i's interval.*

929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986

Fig. 6 shows an example of an interval leak. The path condition implied by the execution is

$$\text{start_node} \leq \text{end_node}, \ldots, \text{start_node} + n \leq \text{end_node}$$

where n = end_node − start_node. The condition implies a lower bound on end_node : $n \leq$ end_node and an upper bound on start_node : start_node $\leq M - n$. There is also an obvious leak of the difference between end_node and start_node, however, our analysis does not take such leaks into account. We are interested in interval leaks on input values rather than interval leaks on sensitive ASTs because, by definition, every remote comparison is an interval leak on a sensitive AST.

As another example, consider Fig. 4. If the first conditional fails, but the second one succeeds, the observed path condition implies b1*total < sumRatings <= b2*total, and therefore b1 $\leq M - 1$. If total = 1, the path condition restricts the interval of a rating input, which is an interval leak according to our definition (although by a small constant).

There is no interval leak, if no execution of the program leads to a path condition that constrains the intervals of its constituent inputs. Consider the problem of finding the minimal euclidian distance from a vector x to n other vectors $c_1, \ldots c_n$, which came up in several benchmarks. An execution may lead to path condition

$$\sum_{1 \leq i \leq k} (x_i - c_{1,i})^2 \leq \sum_{1 \leq i \leq k} (x_i - c_{2,i})^2 \ldots \leq \sum_{1 \leq i \leq k} (x_i - c_{n,i})^2$$

For every x_i there exist values $x_j, j \neq i$ and $c_{k,l}$ that satisfy the condition. $c_{k,l}$ remain unconstrained as well. Other executions (with corresponding path conditions) leave inputs x and $c_1, \ldots, c_n$ unconstrained as well.

In addition to interval leaks, we study adversarial advantage due to ordered sequences of sensitive ASTs. Let path condition P imply an ordered sequence of sensitive ASTs: $a \leq b \leq \ldots \leq c$. $a, b, \ldots$, and c can be treated as points in a Random Order Preserving Function (ROPF) $M \rightarrow N$, per Boldyreva et al. We can then make use of Boldyreva's framework to bound leakage due to such ordered sequences: each value (LHE ciphertext) in the sequence implied by the path condition is viewed as a challenge ciphertext. Thus, fixing z to be the number of ordered ciphertexts implied by the path condition, then fixing M, we have

$$\mathbf{Adv}^{1,z-WOW}(A_P) < \frac{4z}{\sqrt{M-z+1}}$$

to estimate the adversary advantage A_P implied by a path condition P. Consider Fig. 4 again. One can see that the longest chain of ordered values implied by a runtime path condition is 3. For example, if the first two arms of the if-then-else statement evaluate to false, but the third evaluates to true, the path condition implies b1*total < sumRatings <= b3*total and b2*total < sumRatings <= b3*total. b0, b1, and each rating are in the range [1..M]. Value total is a plaintext integer. Thus, b0*total, b1*total, and sumRatings are in the range [total..total*M], which maps to [1..M] (Recall that we made the simplifying assumption that M is known to be large to the adversary). The bound implied by this path condition is

$$\mathbf{Adv}^{1,z-WOW}(A_P) < \frac{12}{\sqrt{M-2}}$$

as the adversary sees a sequence of 3 ciphertexts. Therefore, when paths are relatively short, which tends to be the case in MapReduce programs, the adversary advantage due to order leaks is small. For the distance advantage we get an upperbound of

$$\mathbf{Adv}^{1,z-WDOW}(A_P) < \frac{24}{\sqrt{M-2}}$$

Showing that, in this case, the advesary has double the advantage when guessing distances between individual ciphertexts versus guessing the ciphertexts themselves.

In the euclidean distance example on the other hand, the worst case scenario implies an input-dependent, potentially large ordered sequence of sensitive values.

In summary, we reason about remote comparisons from two angles: (1) interval leaks on input values, and (2) ordered sequences of sensitive ASTs, applying Boldyreva et al.'s analysis on ordered sequences of sensitive ASTs implied by path conditions.

Problem	Notes
PageRankInitVector	start_node and end_node
HistogramMovies	b0...b5, rating

Table 2: Interval Leaks for OPE-Remote benchmarks.

Problem	z	Notes
PageRankInitVector	2	
HistogramMovies	3	
PageRankBlock	3	
Kmeans(Puma)	O(n)	$min()$ call
Kmeans(MLHadoop)	O(n)	$min()$ call
Classification	O(n)	$min()$ call
Recommendation	O(n)	$min()$ call
KNN	O(n)	$min()$ call
MarketBasketAnalysis	2	

Table 3: Sequence leaks for OPE-remote programs.

5.3.3 *Unknown.*

Our last category includes programs with unknown security guarantees. If there is an UNK operation whose operand is a sensitive value according to our static analysis, then the program is classified as unknown. When the sensitive value is an aggregate *agg* (e.g., the inner product of two sensitive vectors), our framework executes the operation by having A send *Enc*(*agg*) to B, where B decrypts it, runs the UNK operation over plaintext arguments, then encrypts and sends the result back to A. Our classification of Unknown means that the security guarantees of the framework are Unknown, in a sense that they depend on the kind of aggregation. B sees the aggregate sensitive value *agg*. In certain cases the programmer may deem safe to send the aggregate value to B; this can happen when the probability that *agg* leaks information about individual sensitive inputs is small (e.g., the inner product aggregate). However, in other cases, the programmer may deem leaking a sensitive aggregate unsafe. The degenerate case is when *agg* is a sensitive input itself (e.g., the maximal element in a sequence), in which case the framework is unsafe.

We note that any UNK function can be approximated by the first k terms of its Taylor series, and therefore, in terms of addition

987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044

Unpublished working draft. Not for distribution.

Secure	L1, L2, L3, L4, L5, L6, L7, L9, L10, L11, L13, L15, L16, L17
CMP-local	L14
CMP-remote	-
Unknown	L8, L12

Table 4: Pigmix 2 suite.

and multiplication. However, this will require multiple rounds of communication and we have not considered it in our framework.

6 EXPERIMENTAL RESULTS

We analyze MapReduce applications from six different benchmark suites. The Pigmix2 benchmark suite [50] is created by Apache to test the Apache Pig [8] applications. The Brown benchmark suite compares different MapReduce-like applications from industry. The Puma suite [1] specifically targets applications that are running the Apache Hadoop [7] implementation of MapReduce. The HiBench suite [34] is an industry standard for benchmarking "Big Data" frameworks. Additionally, we analyze the TCP-H [60] query suite from the Monomi [61] framework. This suite consists of SQL applications. The queries can be translated into MapReduce programs (similarly to the way the Pigmix2 suite was translated from SQL). In addition to the five suites described above we analyze implementations of machine learning applications in the MLHadoop GitHub repository [44]. Machine learning applications have been a prominent area of research in secure cloud computation [35, 39]. We found the repository by searching GitHub for "Hadoop machine learning" and MLHadoop was the first Java-based, non-framework, non-library result at the time. It is an open-source implementation of various machine learning applications using the Hadoop framework. We chose three basic but non-trivial machine learning algorithms to run on Google Cloud and AWS: LinearRegression, LogisticRegression and Kmeans. The results and details of our experiments are discussed in Sect. 6.3.

We address the following research questions:

- **RQ1** How applicable is our framework? Specifically, what percentage of applications are Secure, OPE-secure, and Unknown? Is the percentage of Unknown *low*?
- **RQ2** How secure are OPE-secure applications? Specifically, are bounds on lengths of ordered sequences and thus on adversary advantage *low*?
- **RQ3** How scalable is our framework? Is overhead of SecureMCMR over plaintext execution *low*?

Sect. 6.1 addresses **RQ1** and Sect. 6.2 addresses **RQ2**. Sect. 6.3 addresses **RQ3**; we describe experiments on Google Cloud and Amazon's AWS, and measure overhead over plaintext execution on Google Cloud.

Secure	Benchmark2, Benchmark4
CMP-local	Benchmark3 Phase1, Benchmark3 Phase3
CMP-remote	-
Unknown	Benchmark1, Sort

Table 5: Brown suite.

Secure	HistogramRatings, InvertedIndex, SelfJoin, SequenceCount, TermVectorPerHost
CMP-local	AdjList, RankedInvertedIndex
CMP-remote	Classification*, HistogramMovies*, Kmeans*
Unknown	Grep

Table 6: Puma suite. (*) indicates unsupported functions were replaced with semantically equivalent logic (see Sect. 6.2.2).

Secure	DecisionTree, LinearRegression*, MatrixMultiplication, NaiveBayes
CMP-local	Mutual-Friends
CMP-remote	KNN*, Kmeans*, MarketBasketAnalysis, Recommendation
Unknown	LogisticRegression

Table 7: MLHadoop suite. (*) indicates unsupported functions were replaced with semantically equivalent logic (see Sect. 6.2.2).

Secure	DegDist, JoinTablePegasus, MatvecNaive, PagerankPrep, RWRBlock, Saxpy, SaxpyBlock, SaxpyTextoutput, ScalarMult
CMP-local	ConCmptIVGen, ConCmpt, ConCmptBlock
CMP-remote	PagerankBlock*, PagerankInitVector
Unknown	HadiBlock, HadiIVGen, L1norm, L1normBlock, MatvecPrep, RWRNaive, PagerankNaive

Table 8: HiBench suite. (*) indicates unsupported functions were replaced with semantically equivalent logic (see Sect. 6.2.2).

Secure	q2, q8, q9, q11, q13, q16, q17, q18, q19, q22
CMP-local	q1, q3, q4, q5, q6, q7, q10, q12, q15, q20
CMP-remote	-
Unknown	q14, q21

Table 9: Monomi suite.

6.1 Applicability of SecureMCMR

We apply the analysis from Sect. 5 to classify the benchmarks into security levels as described in Sect. 5.3. The analysis from Sect. 4.2 is implemented in Soot and automatically runs on all but the Monomi benchmarks. We use manual inspection to examine path conditions and determine lengths of order sequences and interval leaks.

We present a classification of the benchmarks into one of the four categories introduced in Sect. 5.3: Secure, CMP-local, CMP-remote, and Unknown. Tab. 4, Tab. 5, Tab. 6, Tab. 7, Tab. 8, and Tab. 9 show the results for the six benchmark suites.

Based on our analysis, the majority of the analyzed benchmarks, 50.57% are Secure. 17.24% are Unkown. Tab. 10 summarizes the results. 50.0% of all benchmarks with comparisons exhibit short ordered sequences and no interval leaks. Our Unknown category includes Grep, HadiBlock, and LogisticRegression, which depend on operations that we could not represent using RE (e.g., LogisticRegression employs the sigmoid function).

Unpublished working draft. Not for distribution.

Secure	CMP-local	CMP-remote	Unknown
44	19	9	15
50.57%	21.83%	10.34%	17.24%

Table 10: Counts and percentages across all benchmarks for each category.

6.2 OPE Secure

For each benchmark in each suite we analyzed the severity of the leaks created by comparisons. We found that 63.16% of the CMP-local programs exhibit a low z value (or allow for rekeying which then entails a low z), and 22.22% of the CMP-remote programs exhibit short order sequences and no interval leaks. As mentioned earlier, this amounts to 50.0% of benchmarks with comparisons exhibiting short ordered sequences and no interval leaks.

6.2.1 CMP-local. We found that the majority of CMP-local tests involved a small number of comparisons for values that could be directly encrypted using OPE. Some benchmarks, for example L14 (Pigmix2), q3 (Monomi), entail an ordered sequence of 2, therefore z values are as low as 2. This allows for a larger number of records to be processed without giving an adversary a large advantage. In Tab. 1 we show concrete z values for all of the CMP-local classified benchmarks. If all values are encrypted with a single OPE key, every comparison adds to an attackers advantage with the worst case being $O(n)$ (or worse) where n is the number of records in the input file. However, as shown in Sect. 5.3.2, rekeying reduces this vulnerability. Some functions, including sorting, maximum, and minimum require a shared key and cannot benefit from rekeying. This is reflected in the classification of several benchmarks in Tab. 1, such as RankedInvertedIndex (Brown) and q15 (Monomi).

6.2.2 CMP-remote. With the exception of suites HiBench and ML-Hadoop (discussed below), we found that our hypothesis, which predicted low numbers of traces involving comparison of LHE-encrypted values, held. We found 3 out of the 57 programs in suites Pigmix, Brown, Puma, and Monomi that required remote comparison (Kmeans, HistogramMovies, Classification). As discussed in Sect. 5.3.2, there are two types of CMP-remote leaks:

(1) **Interval Leaks:** Leaks that are created due to the interaction of comparisons on LHE encrypted values with knowledge of AST structure.
(2) **Sequence Leaks:** Leaks due to path conditions revealing ordered sequences of values.

Tab. 2 quantifies interval leaks and Tab. 3 quantifies sequence leaks in our CMP-remote benchmarks.

The benchmarks marked with an asterisk involve functions that cannot be directly done due to unsupported operations, but can be rewritten to produce the exact same result. For example, Kmeans involves the sqrt function to determine the distance between points and cluster centroids. While sqrt cannot be done directly using RE, the distances can be ranked by using the sum of squares. This has been the standard approach in MPC, which does not support sqrt in biometric matching, a classical MPC benchmark [15, 22].

```
1 double coverage_threashold = 0.001
2 if( change_reported == 0 ) {
3   double diff = Math.abs(previous_rank−next_rank);
4   if( diff > converge_threshold ) {
5     ...
6   }
7 }
```

Figure 7: abs() calculation adapted from PagerankBlock.

```
1 double lower = ... // LHE−encrypted constant, e.g., −0.001
2 double upper = ... // LHE coverage_threshold, e.g., 0.001
3 if( change_reported == 0 ) {
4   double diff = previous_rank−next_rank;
5   if( lower < diff && diff < upper ) {
6     ...
7   }
8 }
```

Figure 8: Semantically equivalent transformation of Fig. 7.

The benchmark suites HiBench and MLHadoop provided a larger challenge. Many benchmarks are run in multiple phases, which created long traces for individual programs. Due to these long traces a variable would inevitably effect other variables increasing the possibility of comparisons that exhibit interval leaks and path conditions that entail long ordered sequences of LHE-encrypted values. Due to the length of variable life and a larger number of comparisons, 6 out of 32 benchmarks in these frameworks required remote comparisons. A relatively large percentage of benchmarks in HiBench are in the Unknown category due to unsupported operations, for example HadiBlock and HadiIVGen which utilize unsupported bitwise operations (e.g., AND, OR).

6.2.3 CMP-remote Examples. As stated earlier we transformed some functions into semantically equivalent ones. Fig. 7 shows use of the absolute value function. We cannot carry the LHE-unsupported abs(x) without leakage: if we send x to cloud B, this would leak sensitive x as B can decrypt the value; if we encode abs(x) as a remote comparison on Server A, this would leak the sign of x to A and B. Fortunately, we can obtain an equivalent function by bounding the negative and positive variants of the bound, as shown in Fig. 8. The resulting program is CMP-remote; it exhibits no interval leak and an order sequence of length 3.

6.3 Scalability of SecureMCMR

To test the scalability of SecureMCMR we transformed three MapReduce programs from MLHadoop, and studied their performance. Importantly, we used two *different* cloud providers to execute the programs.

We chose Machine Learning applications, which have been an important area of research in secure computation [28, 35, 39, 42, 46, 55, 56]. We chose three programs from MLHadoop: LinearRegression, LogisticRegression and Kmeans. They use LHE-unsupported operations that require remote communication such as RE-supported multiplication, and they are likely to exhibit slowdown with SecureMCMR. Neither of these applications is supported by existing frameworks for MapReduce such as MrCrypt or SecureMR. We

Unpublished working draft. Not for distribution.

	4 nodes	8 nodes	16 nodes
SecureMCMR	528 sec.	258 sec.	200 sec.
Slowdown	≈6.86x	≈3.35x	≈2.60x

Table 11: Run times: Linear Regression

note that there is a number of applications, MLHadoop ones included that can run in SecureMCMR without any remote communication, as every operation is supported by a single encryption scheme (e.g., NaiveBaise). The benchmarks span our classification—LinearRegression is Secure, Kmeans is CMP-remote, and LogisticRegression is Unknown. LogisticRegression is classified Unknown because it computes e^{-x} as part of the sigmoid function, where x is a sensitive aggregate value. We argue that users of SecureMCMR may determine that such an aggregate leak is acceptable for their purposes.

Overhead ranges from 3.8x for Kmeans to 1.6x for LinearRegression. Importantly, as the number of cloud B nodes (Amazon) increases, running time decreases.

6.3.1 Setup and Test Data. Google Cloud (cloud A) runs the MapReduce task and sends unsupported operations to AWS (cloud B). Google Cloud is hosted in the us-east-1b region while Amazon Web Services (AWS) in the N. Virginia region. Each cloud is configured with specific hardware to offer the lowest cost per run. More specifically, the Google side runs on 250 nodes, where each node has 1 thread and access to 3.75GB of RAM. The AWS side runs on 4, 8, or 16 t2.nano instances with 1 thread and access to 500MB of RAM. In terms of operating systems the Google Cloud side uses a variant of Debian 9, while the AWS side runs on Ubuntu Linux 18.04.1 LTS.

For Linear regression we used the Iris Data Set [26] as a base point to generate a 240,000 data points for training. For Logistic Regression we used a real world dataset used to classify potential pulsar stars; this dataset has 179,890 points with 8 features each [54]. For Kmeans classification we created a dataset of 100,000 data points in 2D euclidean space and grouped them into 4 clusters.

6.3.2 Results. We compared the running times of SecureMCMR, which uses both LHE and network communication for RE-supported operations, with the standard run on Google Cloud in plaintext. Tab. 11, Tab. 12, and Tab. 13 show the running times, slowdown and overhead. As expected, increasing the number of Amazon nodes decreases the overhead. We also found that fewer than 4 Amazon nodes results in 100% CPU utilization, stalls the experiment, and ultimately crashes the job.

In Fig. 9, Fig. 10, and Fig. 11 we also present three metrics for measuring the load over time on each individual Amazon instance. We measured 1) average CPU utilization, 2) bytes of data coming in, and 3) bytes of data going out for the LinearRegression benchmark. As expected, as we increase the number of t2.nano nodes, the workload is distributed among nodes and the average load per node decreases.

We view overhead as low as 160% using 16 very cheap Amazon nodes as promising (t2.nano node costs $0.0058 per hour or $4.23 per month as of November 26, 2019).

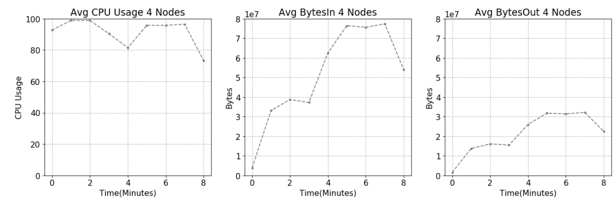

Figure 9: Linear Regression results for 250 Google cloud workers and 4 AWS t2.nano nodes. CPU usage measures the average CPU usage across all AWS instances. BytesIn and BytesOut show average amount of data flowing to and from the AWS instances.

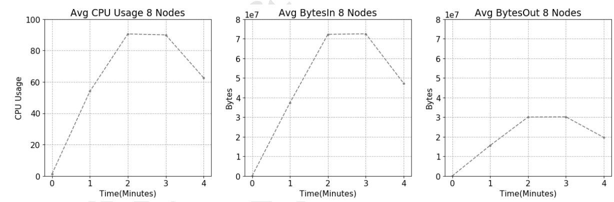

Figure 10: Linear Regression results for 250 Google cloud workers and 8 Amazon AWS t2.nano nodes.

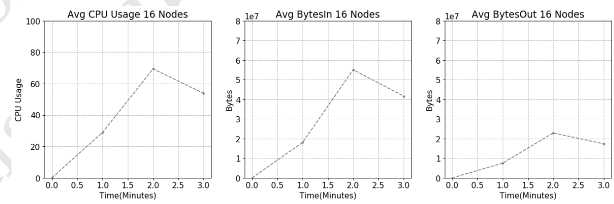

Figure 11: Linear Regression results for 250 Google cloud workers and 16 Amazon AWS t2.nano nodes.

	4 nodes	8 nodes	16 nodes
SecureMCMR	463 sec.	315 sec.	244 sec.
Slowdown	≈6.71x	≈4.57x	≈3.54x

Table 12: Run times: Logistic Regression

6.3.3 Discussion. Overall we believe that SecureMCMR is a promising direction. Not only do these benchmarks run on low cost cloud instances, but many can run with low overheads and minimal (if any) security leaks due to order preserving encryption schemes. Naturally, overhead is determined by the nature of the problem. Not only is each problem unique in the number of communications it requires, but it is also unique in the dependencies between subsequent operations or data points. These relationships determine whether operations can be parallelized and carried in a single communication round, a strategy that we found greatly improves efficiency.

Consider the calculation of *h_theta* from LinearRegression shown in Fig. 12. A naive approach will initiate x.length sequential remote

Unpublished working draft. Not for distribution.

	4 nodes	8 nodes	16 nodes
SecureMCMR	473 sec.	321 sec.	315 sec.
Slowdown	≈7.2x	≈4.89x	≈4.8x

Table 13: Run times: Kmeans Classification

```
int h_theta = 0;
...
for(int i=0; i < x.length; i++){
    h_theta += remote_mul(x[i], theta[i]);
}
...
```

Figure 12: Calculation of *h_theta* without parallelization.

```
int h_theta = 0;
...
for(int i=0; i < x.length; i++){
    queue.add(new remote_mul(x[i], theta[i]));
}
int[] answers = q.resolve();
for(int i = 0; i < answers.length; i++) {
    h_theta += answers[i];
}
...
```

Figure 13: Calculation of *h_theta* with parallelization

```
int foo(int start, int length) {
    int[] arr = new int[length];
    arr[0] = start;
    for(int i = 1; i < arr.length; i++) {
        arr[i] += arr[i−1] * random_integer(1, 5);
    }
    return arr[length − 1];
}
```

Figure 14: Loop carried dependencies that cannot be parallelized.

multiplication operations, maximizing the communication overhead. Using a simple threading pool all of the messages can be sent at once, or in large batches (see Fig. 13). This reduces the number of communication rounds and communication overhead. If the cardinality of x is low, this means a single communication round.

In contrast, in Fig. 14 each operation within the loop depends on the previous iteration. The loop cannot be parallelized due to the loop-carried dependencies, and it requires arr.length communication rounds.

Parallelization plays a key role in our strategy as we parallelize the inner product computation in LinearRegression, LogisticRegression, and Kmeans. By reducing the number of communication rounds we could consistently reduce overall runtimes. However, while parallelization decreases the number of messages, it also increases the size of each message. Improvement in runtime depends on the ability of cloud B nodes to process efficiently all operations they receive. Picking the optimal size of batches, and understanding of parallelization are important directions of future work.

7 CONCLUSIONS

We presented SecureMCMR, a novel framework for the analysis and execution of MapReduce programs using PHE, OPE, and multiple clouds. Using multiple untrusted clouds to execute MapReduce programs removes the bottleneck created by a single trusted machine. We also presented an analysis of MapReduce programs with comparisons, and the effect comparisons have on the security of the system. In the future we will continue to research MapReduce programs, parallelization, and optimization, in the context of SecureMCMR and MPC.

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] Aly, Cozzo, Keller, Orsini, Rotaru, Scholl, Smart, and Wood. 2019. SCALE–MAMBA v1.6 : Documentation. https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf

[4] Abdelrahaman Aly and Nigel P Smart. 2019. Benchmarking Privacy Preserving Scientific Operations. In *International Conference on Applied Cryptography and Network Security.* Springer, 509–529.

[5] N1 Analytics. 2017. javallier. https://github.com/n1analytics/javallier.

[6] Apache Software Foundation. 2005–2019. Apache CouchDB. https://couchdb.apache.org/.

[7] Apache Software Foundation. 2006–2018. Apache Hadoop. https://hadoop.apache.org/.

[8] Apache Software Foundation. 2008–2017. Apache Pig. https://pig.apache.org/.

[9] Benny Applebaum. 2017. Garbled Circuits as Randomized Encodings of Functions: a Primer. In *IACR Cryptology ePrint Archive.*

[10] Assaf Ben-David, Noam Nisan, and Benny Pinkas. 2008. FairplayMP: A System for Secure Multi-Party Computation. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08).* Association for Computing Machinery, New York, NY, USA, 257–266. https://doi.org/10.1145/1455770.1455804

[11] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-cryptographic Fault-tolerant Distributed Computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC '88).* ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/62212.62213

[12] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security (ESORICS '08).* Springer-Verlag, Berlin, Heidelberg, 192–206. https://doi.org/10.1007/978-3-540-88313-5_13

[13] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. 2009. Order-Preserving Symmetric Encryption. In *Advances in Cryptology - EUROCRYPT 2009*, Antoine Joux (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 224–241.

[14] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. 2011. Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *Advances in Cryptology – CRYPTO 2011*, Phillip Rogaway (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 578–595.

[15] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. 2018. HyCC: Compilation of Hybrid Protocols for Practical Secure Computation. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18).* ACM, New York, NY, USA, 847–861. https://doi.org/10.1145/3243734.3243786

[16] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. 2019. EzPC: Programmable, Efficient, and Scalable Secure Two-Party Computation for Machine Learning. In *IEEE European Symposium on Security and Privacy.* (IEEE EuroS&P 2019). https://www.microsoft.com/en-us/research/publication/ezpc-programmable-efficient-and-scalable-secure-two-party-computation-for-machine-learning/

[17] David Chaum, Claude Crépeau, and Ivan Damgard. 1988. Multiparty Unconditionally Secure Protocols. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC '88).* ACM, New York, NY, USA, 11–19. https://doi.org/10.1145/62212.62214

[18] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. 2007. Efficient and Secure Comparison for On-Line Auctions. In *Proceedings of the 12th Australasian Conference on Information Security and Privacy (ACISP'07).* Springer-Verlag, Berlin, Heidelberg, 416–430.

Unpublished working draft. Not for distribution.

[19] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. 2009. Asynchronous Multiparty Computation: Theory and Implementation. In *Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography: PKC '09 (Irvine)*. Springer-Verlag, Berlin, Heidelberg, 160–179. https://doi.org/10.1007/978-3-642-00468-1_10

[20] Ivan Damgård and Mats Jurik. 2001. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In *Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography (PKC '01)*. Springer-Verlag, Berlin, Heidelberg, 119–136.

[21] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 137–150.

[22] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*.

[23] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. 2015. M2R: Enabling Stronger Privacy in Mapreduce Computation. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, USA, 447–462.

[24] Disco Project. 2008–2019. disco: a Map/Reduce framework for distributed computing. https://github.com/discoproject/disco.

[25] Yao Dong, Ana Milanova, and Julian Dolby. 2018. SecureMR: Secure MapReduce computation using homomorphic encryption and program partitioning. In *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security, HoTSoS 2018, Raleigh, North Carolina, USA, April 10-11, 2018*. 4:1–4:13.

[26] R. A. Fisher. 1936. The Use of Multiple Measurements in Taxonomic Problems. *Annals of Eugenics* 7 (1936), 179–188.

[27] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. 2014. CBMC-GC: an ANSI C compiler for secure two-party computations. In *International Conference on Compiler Construction*. Springer, 244–249.

[28] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. 2017. Privacy-Preserving Distributed Linear Regression on High-Dimensional Data. *PoPETs* 2017 (2017), 345–364.

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

[30] Craig Gentry and Shai Halevi. 2011. Implementing Gentry's Fully-homomorphic Encryption Scheme. In *Proc. 30th Annu. Int. Conf. Theory and Appl. of Cryptographic Techn.: Advances in Cryptology (EUROCRYPT'11)*. 129–148.

[31] Irene Giacomelli, Somesh Jha, Marc Joye, C. David Page, and Kyonghwan Yoon. 2018. Privacy-Preserving Ridge Regression with only Linearly-Homomorphic Encryption. In *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings (Lecture Notes in Computer Science)*, Bart Preneel and Frederik Vercauteren (Eds.), Vol. 10892. Springer, 243–261. https://doi.org/10.1007/978-3-319-93387-0_13

[32] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87)*. ACM, New York, NY, USA, 218–229. https://doi.org/10.1145/28395.28420

[33] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. 2019. SoK: General Purpose Compilers for Secure Multi-Party Computation. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 479–496. https://doi.org/10.1109/SP.2019.00028

[34] Intel-bigdata. 2012-2017. HiBench Suite. https://github.com/Intel-bigdata/HiBench.

[35] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, Berkeley, CA, USA, 1651–1668. http://dl.acm.org/citation.cfm?id=3277203.3277326

[36] Marcel Keller, Valerio Pastro, and Dragos Rotaru. 2018. Overdrive: making SPDZ great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 158–189.

[37] Florian Kerschbaum. 2015. Frequency-Hiding Order-Preserving Encryption. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 656–667. https://doi.org/10.1145/2810103.2813629

[38] Florian Kerschbaum, Debmalya Biswas, and Sebastiaan de Hoogh. 2009. Performance Comparison of Secure Comparison Protocols. In *Proceedings of the 2009 20th International Workshop on Database and Expert Systems Application (DEXA '09)*. IEEE Computer Society, Washington, DC, USA, 133–136. https://doi.org/10.1109/DEXA.2009.37

[39] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. 2017. Oblivious Neural Network Predictions via MiniONN Transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 619–631. https://doi.org/10.1145/3133956.3134056

[40] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay—a Secure Two-Party Computation System. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, USA, 20.

[41] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 35–52. https://doi.org/10.1145/3243734.3243760

[42] P. Mohassel and Y. Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy (SP)*. 19–38. https://doi.org/10.1109/SP.2017.12

[43] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor. 2016. Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter for Secure Computation. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*. 112–127. https://doi.org/10.1109/EuroSP.2016.20

[44] Punit Naik. 2016-2018. MLHadoop. https://github.com/punit-naik/MLHadoop.

[45] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, USA, 377–394. https://doi.org/10.1109/SP.2015.30

[46] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. 2013. Privacy-Preserving Ridge Regression on Hundreds of Millions of Records. In *2013 IEEE Symposium on Security and Privacy*. 334–348. https://doi.org/10.1109/SP.2013.30

[47] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*. USENIX Association, USA, 619–636.

[48] 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)*. 223–238.

[49] Erman Pattuk, Murat Kantarcioglu, Huseyin Ulusoy, and Bradley Malin. 2016. CheapSMC: A Framework to Minimize SMC Cost in Cloud. *ArXiv* abs/1605.00300 (2016).

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

[51] Raluca Ada Popa et al. 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.

[52] 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. https://doi.org/10.1145/2043556.2043566

[53] Pille Pullonen, Dan Bogdanov, and Thomas Schneider. 2012. The Design and Implementation of a Two-Party Protocol Suite for SHAREMIND 3. http://tubiblio.ulb.tu-darmstadt.de/61259/

[54] Pavan Raj. 2018. Predicting a Pulsar Star. https://www.kaggle.com/pavanraj159/predicting-a-pulsar-star/metadata.

[55] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS '18)*. ACM, New York, NY, USA, 707–721. https://doi.org/10.1145/3196494.3196522

[56] Bita Darvish Rouhani, Siam Umar Hussain, Kristin Lauter, and Farinaz Koushanfar. 2018. ReDCrypt: Real-Time Privacy-Preserving Deep Learning Inference in Clouds Using FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 11, 3, Article 21 (Dec. 2018), 21 pages. https://doi.org/10.1145/3242899

[57] 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 Program. Lang. Researchers*.

[58] Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd Millstein. 2013. MrCrypt: Static Analysis for Secure Cloud Computations. In *Proc. 2013 ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. & Appl.* 271–286.

[59] Shruti Tople et al. 2013. AUTOCRYPT: Enabling Homomorphic Computation on Servers to Protect Sensitive Web Content. In *Proc. 2013 ACM SIGSAC Conf. Comput. & Commun. Security (CCS '13)*. 1297–1310.

[60] TPC. 2001-2019. TPC-H. http://www.tpc.org/tpch/.

[61] 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 (VLDB'13)*. 289–300.

[62] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. 2010. Fully Homomorphic Encryption over the Integers. In *Advances in Cryptology – EUROCRYPT 2010*, Henri Gilbert (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 24–43.

[63] A. C. Yao. 1982. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. 160–164.

Unpublished working draft. Not for distribution.