3-27-2015

Constructing Secure MapReduce Framework in Cloud-based Environment

Yongzhi Wang

School of Computing and Information Sciences, ywang032@cis.fiu.edu

DOI: 10.25148/etd.FIDC000061

Follow this and additional works at: <https://digitalcommons.fiu.edu/etd>

Part of the [Information Security Commons](#)

Recommended Citation

Wang, Yongzhi, "Constructing Secure MapReduce Framework in Cloud-based Environment" (2015). *FIU Electronic Theses and Dissertations*. 2238.

<https://digitalcommons.fiu.edu/etd/2238>

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact dcc@fiu.edu.

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

CONSTRUCTING SECURE MAPREDUCE FRAMEWORK IN CLOUD-BASED
ENVIRONMENT

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Yongzhi Wang

2015

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Yongzhi Wang, and entitled Constructing Secure MapReduce Framework in Cloud-based Environment, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Xudong He

Geoffrey Smith

Bogdan Carbunar

Gang Quan

Jinpeng Wei, Major Professor

Date of Defense: March 27, 2015

The dissertation of Yongzhi Wang is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2015

© Copyright 2015 by Yongzhi Wang

All rights reserved.

DEDICATION

To my family.

ACKNOWLEDGMENTS

I would like to express my sincerest gratitude to all those who contributed to the success of my PhD study and the completion of this dissertation. Special thanks to the members of my committee, Drs. Xudong He, Geoffrey Smith, Bogdan Carbunar and Gang Quan for their insights and rigorous questioning that has resulted in the focus that resulted in this work. I especially would like to express my foremost gratitude to Dr. Jinpeng Wei who guided me through this process, going above and beyond in due diligence and patience.

I own a great debt to Dr. Sundaraja Sitharama Iyengar for his strongest support in the most difficult time during my PhD study, which helped me to go through the hardness. I am also grateful to Ms. Olga Carbonell for her consistent help and support in providing suggestions and preparing all the required paperwork.

Finally, I would like to thank my family. I thank my parents, Mr. Junjun Wang and Ms. Yanyun Dong, for their endless loves and supports. I also thank my wife, Ms. Lan Yu, for all her patience, understanding and sacrificing. Last but not least, I also thank my son, Qiaoze Wang, who gives me numerous moments of joy and the strength required to finish the research and to write this dissertation.

This dissertation is partially funded by the Florida International University Dissertation Year Fellowship.

ABSTRACT OF THE DISSERTATION

CONSTRUCTING SECURE MAPREDUCE FRAMEWORK IN CLOUD-BASED
ENVIRONMENT

by

Yongzhi Wang

Florida International University, 2015

Miami, Florida

Professor Jinpeng Wei, Major Professor

MapReduce, a parallel computing paradigm, has been gaining popularity in recent years as cloud vendors offer MapReduce computation services on their public clouds. However, companies are still reluctant to move their computations to the public cloud due to the following reason: In the current business model, the entire MapReduce cluster is deployed on the public cloud. If the public cloud is not properly protected, the integrity and the confidentiality of MapReduce applications can be compromised by attacks inside or outside of the public cloud. From the result integrity's perspective, if any computation nodes on the public cloud are compromised, those nodes can return incorrect task results and therefore render the final job result inaccurate. From the algorithmic confidentiality's perspective, when more and more companies devise innovative algorithms and deploy them to the public cloud, malicious attackers can reverse engineer those programs to detect the algorithmic details and, therefore, compromise the intellectual property of those companies.

In this dissertation, we propose to use the hybrid cloud architecture to defeat the above two threats. Based on the hybrid cloud architecture, we propose separate solutions to

address the result integrity and the algorithmic confidentiality problems. To address the result integrity problem, we propose the Integrity Assurance MapReduce (IAMR) framework. IAMR performs the result checking technique to guarantee high result accuracy of MapReduce jobs, even if the computation is executed on an untrusted public cloud. We implemented a prototype system for a real hybrid cloud environment and performed a series of experiments. Our theoretical simulations and experimental results show that IAMR can guarantee a very low job error rate, while maintaining a moderate performance overhead. To address the algorithmic confidentiality problem, we focus on the program control flow and propose the Confidentiality Assurance MapReduce (CAMR) framework. CAMR performs the Runtime Control Flow Obfuscation (RCFO) technique to protect the predicates of MapReduce jobs. We implemented a prototype system for a real hybrid cloud environment. The security analysis and experimental results show that CAMR defeats static analysis-based reverse engineering attacks, raises the bar for the dynamic analysis-based reverse engineering attacks, and incurs a modest performance overhead.

TABLE OF CONTENTS

CHAPTER	PAGE
1. INTRODUCTION	1
1.1 Motivation and Contribution	1
1.2 Outline	6
1.3 Publications	6
2. PRELIMINARY KNOWLEDGE.....	7
2.1 MapReduce.....	7
2.2 Software Obfuscation	10
3. INTEGRITY ASSURANCE MAPREDUCE.....	11
3.1 System Motivation.....	11
3.2 System Architecture and Design Overview.....	12
3.3 System Assumption and the Attacker Model	13
3.3.1 System Assumption.....	13
3.3.2 Attacker Model.....	15
3.4 System Design.....	15
3.4.1 Map Phase Integrity Check	16
3.4.2 Reduce Phase Integrity Check.....	20
3.5 Theoretical Study.....	26
3.5.1 Quantitative Analysis	26
3.5.2 Simulation Result	37
3.6 Optimal Parameter Searching Algorithms.....	40
3.6.1 Model the System as a Two-player Zero-sum Game	40
3.6.2 Interactive Gradient Descent Algorithm	42
3.6.3 Tiered Interactive Gradient Descent Algorithm.....	50
3.7 Experimental Results.....	57
3.7.1 Experiments on the Result Checking Technique	58
3.7.2 Experiments on the Parameter Searching Algorithm.....	63
3.8 Related Work.....	66
3.9 Summary.....	69
4. CONFIDENTIALITY ASSURANCE MAPREDUCE.....	71
4.1 System Motivation.....	71
4.2 Architecture and System Overview	76
4.3 System Assumptions and Attacker Model.....	82
4.4 Design Details	84
4.4.1 Transforming the Original Condition.....	84
4.4.2 Inserting Fake Branch Statements.....	88
4.4.3 Speed up Control Flow Queries with the Continuous Cache.....	95
4.4.4 CFQ Function Encryption	97
4.4.5 Loop Transformation.....	98
4.5 Security Discussion	106

4.6	System Implementation	109
4.6.1	Condition Transformation	109
4.6.2	Fake Branch Statement Insertion	110
4.6.3	Private Program Implementation	111
4.6.4	Hadoop Job Obfuscation	111
4.6.5	CFQ Function Encryption	112
4.6.6	Loop Transformation.....	113
4.6.7	Obfuscation Case Study	113
4.7	Experiments	116
4.7.1	Performance Test.....	117
4.7.2	Scalability Test.....	119
4.7.3	Discussion	122
4.8	Related Work.....	123
4.9	Summary.....	126
5	DISCUSSION.....	128
5.1	Integrity Assurance MapReduce	128
5.2	Confidentiality Assurance MapReduce	129
6	CONCLUSION.....	131
	REFERENCES	133
	VITA.....	140

LIST OF TABLES

TABLE	PAGE
TABLE 1 IAMR System Modeling Parameters and Measurement Metrics	29
TABLE 2 Categories of Tasks Based on Assignment Conditions	34
TABLE 3 The Probability, Workload and Verifier Overhead of Each Category	35
TABLE 4 IAMR System Modeling Parameters for Reduce Phase	36
TABLE 5 The Job Error Rate, Overhead and Verifier Overhead of Wordcount Application with Map Phase Integrity Check	59
TABLE 6 The Performance of Map Phase IAMR	61
TABLE 7 The Performance of Reduce Phase IAMR	62
TABLE 8 Parameter Settings in TIGD Experiments	64
TABLE 9 Results in TIGD Experiments	64
TABLE 10 Hadoop Jobs Used in the Experiments	117
TABLE 11 CFQ Function Invocation Statistics and Cache-hit Rates	119
TABLE 12 Scalability Test with Word Count	120
TABLE 13 Scalability Test with Pi	121
TABLE 14 Scalability Test with TeraSort	122

LIST OF FIGURES

FIGURE	PAGE
Figure 1 MapReduce Example: Word Count.....	9
Figure 2 The Architecture of Integrity Assurance MapReduce.....	13
Figure 3 The Workflow of Integrity Assurance MapReduce	19
Figure 4 Locating Map Output Between Key 2 and Key 9 using Key Table.....	24
Figure 5 Workflow of IAMR in the Reduce Phase.....	25
Figure 6 Simulation of IAMR Analysis.....	38
Figure 7 Interactive Gradient Descent Algorithm	45
Figure 8 Minimize U_{IAMR} with Gradient Descent on T	47
Figure 9 Minimize U_{ATT} with Gradient Descent on c	47
Figure 10 Tiered Interactive Gradient Descent Algorithm	53
Figure 11 Minimize U_{IAMR} with Restricted Gradient Descent on T	54
Figure 12 Malicious Worker Elimination Progress	60
Figure 13 TIGD and IGD algorithm execution details	66
Figure 14 The Amazon Elastic MapReduce Use Case and the Attacker Model	72
Figure 15 Implementation of Canny Edge Detector Program	74
Figure 16 Condition Transformation of the Canny Edge Detector Implementation in Figure 15.....	78
Figure 17 Execution Scenario of Confidentiality Assurance MapReduce	82
Figure 18 Condition Normalization Rule in Original Branch Statement Transformation	85
Figure 19 Example of Transforming Normalized Condition.....	87
Figure 20 Variable Range Evaluation Algorithm	90
Figure 21 Example of Inserting Fake Condition.....	95
Figure 22 The Design of Continuous Cache.....	96

Figure 23 CFQ Function Encryption	98
Figure 24 Example of Loop Transformation	100
Figure 25 Obfuscation of SortGenMapper.addKey in TeraGen job.....	115
Figure 26 Execution Times of MapReduce Jobs with Different Obfuscation Degree d	118

ABBREVIATIONS AND ACRONYMS

AST	Abstract Syntax Tree
AWS	Amazon Web Services
CAMR	Confidentiality Assurance MapReduce
CFQ	Control Flow Query
DFS	Distributed File System
EC2	Elastic Compute Cloud
EMR	Elastic MapReduce
IAMR	Integrity Assurance MapReduce
IGD	Interactive Gradient Descent
LHS	Left Hand Side
LTPG	Loop Transformation Parameter Generation
RCFO	Runtime Control Flow Obfuscation
RHS	Right Hand Side
S3	Simple Storage Service
TIGD	Tiered Interactive Gradient Descent
TPM	Trusted Platform Module

1. INTRODUCTION

1.1 Motivation and Contribution

Big Data becomes a buzzword in recent years when sizes of computing jobs scale up rapidly. The traditional computing model can no longer satisfy the speed requirement of such large-scale jobs. One direction to address such a challenge is to process computing jobs in a parallel approach. MapReduce is a generalized parallel computing paradigm that is widely used in various applications, including web search engine, geo-informatics system, and so on. MapReduce applications normally are deployed to a cluster consisting of hundreds or thousands of computation nodes to achieve high parallelism. However, most MapReduce customers cannot afford or do not want to invest in computer clusters of such a large scale. The emergence of Cloud Computing provides an economical alternative for those users to deploy arbitrary large-scale clusters on demand. With the public cloud, MapReduce users can deploy virtualized clusters on the public cloud based on their demands. They only need to pay the cloud vendor based on the resources they used. Such a new model not only reduces users' IT infrastructure cost, but also boost their productivities.

Today, many cloud vendors are offering MapReduce computation services (e.g., *Amazon Elastic MapReduce (EMR)* and *Microsoft Daytona*) on their public clouds. However, a lot of companies or individuals are still reluctant to move their computations to the cloud due to security concerns. Indeed, those worries are necessary. For instance, [1] pointed out a security vulnerability that *Amazon Elastic Compute Cloud (EC2)* suffers from: some members of the EC2 community can create and upload malicious *Amazon Machine Images (AMIs)*, which, if widely used, can flood the EC2 cloud with malicious

virtual machines that contain compromised applications, including MapReduce. Even worse, since the cloud service is opaque to the cloud user, the service providers and the government authorities might abuse their privileges to violate the security guarantee of the public cloud without notifying the cloud customer. For example, the 2013 global surveillance disclosure [18] revealed that the National Security Agency of the US government spies on the online activities of the Internet users with or without the knowledge of Internet corporations. Under such a situation, a service that can outsource computation without outsourcing security is needed to earn customers' confidences.

The above vulnerabilities exist due to the reason that the entire service is deployed to the public cloud. In the case of MapReduce computing, the entire computation is performed on the public cloud. If the computation node in the public cloud is compromised, the computation integrity and the confidentiality can be compromised. In this dissertation, we specifically explore the following two security problems.

1. *The result integrity problem.*

The MapReduce job is usually executed on a cluster, which consists of a number of computing nodes. The task result of each node will contribute to the accuracy of the final job result. If any computation node on the public cloud is compromised, this node can return incorrect task results and therefore render the final job result inaccurate. In this situation, a solution is needed to protect the integrity of task results and the entire job result.

2. *The algorithmic confidentiality problem.*

Nowadays, more and more companies are devising their innovative algorithms to provide various information services, including business analytics, geospatial

mapping/searching, bioinformatics analysis, image processing, etc. When those companies deploy their services to the public cloud [33][34][35][36], they hope that the algorithms in their services are not revealed to others, especially to their business competitors. However, if the hacker is able to compromise the public cloud, the hacker can perform reverse engineering attacks on the company's program to find out its algorithmic details. By doing so, the hacker can offer a similar service with the same level of service quality. Under this situation, a solution is needed to protect the confidentiality of the algorithm details in the outsourced program.

In this dissertation, we propose to use a unified architecture, the hybrid cloud architecture, to address the above two problems. The hybrid cloud architecture consist of a trusted private cloud and an untrusted public cloud. The intuition of our solution is as follows. Since the private cloud is trusted, we can leverage its trustworthiness to obtain the control of the computation; meanwhile, we utilize the public cloud to perform the majority of computations. Since the private cloud can be deployed in users' premises, users control the security.

Based on the hybrid cloud architecture, we propose separate solutions to address the result integrity problem and the algorithmic confidentiality problem. To address the result integrity problem, we propose a novel MapReduce framework, called *Integrity Assurance MapReduce (IAMR)*. IAMR can guarantee high result integrity of MapReduce computations, even if the computation is executed on a public cloud that contains malicious computing nodes. IAMR overlays MapReduce computations on the hybrid cloud. It performs a proposed high accuracy assurance scheme called the *result checking*

technique to assure high result integrity of MapReduce computations. The result checking technique performs probabilistic task replication on the public cloud, performs probabilistic task verification on the private cloud, and maintains trust management for each worker to ensure the participating computing node are benign.

Our theoretical analysis shows that IAMR can guarantee a very low job error rate. For example, when we set the batch size as 50 and the replication probability as 0.5, we can guarantee less than 1% of job error rate when half of workers on the public cloud are malicious. Based on our theoretical analysis, we model the IAMR and the attacker as a two-player zero sum game and propose two algorithms to search for optimal parameter values for IAMR: *Interactive Gradient Descent (IGD)* algorithm and *Tiered Interactive Gradient Descent (TIGD)* algorithm. The IGD algorithm can find the parameter settings for IAMR based on the user's accuracy requirement. The TIGD algorithm can find the parameter settings not only based on the user's accuracy requirement, but also on user's system restriction settings.

We implemented a prototype system that caters to a real hybrid cloud environment (a hybrid cloud consisting of a local private cloud and Amazon public cloud¹) and performed a series of experiments to evaluate its performance. Our experimental results show that IAMR can guarantee a very low job error rate and maintain a moderate performance overhead. For instance, it introduces 19% to 83% of delay in the map phase depending on the replication probability, and 29% of delay in the reduce phase.

In order to address the algorithmic confidentiality problem, we propose the *Confidentiality Assurance MapReduce (CAMR)* platform. CAMR performs a novel control

¹ Amazon Elastic Compute Cloud (EC2), <http://aws.amazon.com/ec2/>

flow obfuscation technique called *Runtime Control Flow Obfuscation (RCFO)* on outsourced MapReduce jobs to hide information on program predicates and complicate the control flow graph. RCFO transforms each original MapReduce job program into two programs: the public program and the private program. The private program is executed on the trusted private cloud. It maintains the information of program predicates and determines the runtime control flow. The public program executes the majority computation on the public cloud, but it needs to query the private program to determine the runtime control flow. In order to further protect the control flow, RCFO also performs the *fake branch statements insertion* method, the *CFQ function encryption* scheme and the *loop transformation* technique to further raise the bar for reverse engineering attacks. To reduce the performance overhead, RCFO maintains a *continuous cache* to reduce the cross-cloud communication.

We developed a prototype system that caters to a real hybrid cloud environment (a hybrid cloud consisting of a local private cloud and Amazon EMR²). Our prototype system can automatically apply RCFO on MapReduce job programs and run the obfuscated job on the hybrid cloud environment. Our experimental results show that the average performance overhead ranges from 14.9% to 33.2% when the obfuscation degree increases from 0 to 1.0.

We declare that our solutions in this dissertation are not only restricted to MapReduce itself. Since MapReduce represents a wide class of outsourced computing paradigms, our solutions can be applied to a wide range of outsourced computing scenarios without major modification. The detailed discussion will be performed in Chapter 5.

² Amazon Elastic MapReduce (EMR), <http://aws.amazon.com/elasticmapreduce/>

1.2 Outline

The rest of this dissertation is organized as follows: Chapter 2 introduces the background knowledge involved in this dissertation. Chapter 3 presents the motivation, assumption, design details, experimental results, and the related work of the IAMR framework. Chapter 4 presents the motivation, assumption, design details, implementation details, experimental results, and the related work of CAMR framework. Chapter 5 discusses the generalization of our solutions to other outsourced computing scenarios. Chapter 6 makes the conclusion of this dissertation.

1.3 Publications

Based on our works in this dissertation, we published the following papers in peer-reviewed journals and conferences:

- Yongzhi Wang, Jinpeng Wei, "Toward Protecting Control Flow Confidentiality in Cloud-based Computation", Elsevier Journal of Computers and Security. Volume 52, pp. 106-127, July 2015. Doi: 10.1016/j.cose.2015.04.005.
- Yongzhi Wang, Jinpeng Wei, Yucong Duan "Securing MapReduce Result Integrity via Verification-based Integrity Assurance Framework". International Journal of Grid and Distributed Computing (IJGDC). Vol. 7, No. 6 (2014), pp.53-70
- Yongzhi Wang, Jinpeng Wei, Mudhakar Srivatsa "Cross Cloud MapReduce: A Result Integrity Check Framework on Hybrid Clouds". International Journal of Cloud Computing (IJCC), Vol. 1, No. 1, 2013.
- Yongzhi Wang, Jinpeng Wei, Mudhakar Srivatsa, Yucong Duan, and Wencai Du. "IntegrityMR: Integrity Assurance Framework for Big Data Analytics and

Management Applications". The First International Workshop on Knowledge Management and Big Data Analytics (KMBA), Oct 6 - 9, 2013, Santa Clara, CA, USA.

- Yongzhi Wang, Jinpeng Wei, and Mudhakar Srivatsa. "Result Integrity Check for MapReduce Computation on Hybrid Clouds". The 6th IEEE International Conference on Cloud Computing (IEEE CLOUD 2013), June 27 - July 2, 2013, Santa Clara, CA.
- Yongzhi Wang, Jinpeng Wei, Mudhakar Srivatsa. "Cross Cloud MapReduce: an Uncheatable MapReduce". The 33rd IEEE Symposium on Security and Privacy (IEEE S&P), San Francisco, CA, May 20-23, 2012. Poster paper.
- Yongzhi Wang, Jinpeng Wei. "VIAF: Verification-based Integrity Assurance Framework for MapReduce". The Fourth IEEE International Conference on Cloud Computing (CLOUD 2011), July 4-9, 2011, Washington DC.

2 PRELIMINARY KNOWLEDGE

In this chapter, we introduce the background knowledge involved in this dissertation. Specifically, we will briefly introduce MapReduce and discuss basic concepts about software obfuscation.

2.1 MapReduce

MapReduce [5] is a parallel programming model for large-scale dataset processing. It usually has been implemented as a distributed system deployed on a cluster. In MapReduce, each computation request issued by the user is called a *job*. Each job is usually broken down into several *tasks*. The traditional architecture of MapReduce consists of one *master*

and a number of *workers*. The master controls the entire computation by managing jobs, scheduling tasks, and maintaining load balance, etc. Workers are hosts that contribute computation resources to execute tasks assigned by the master. In MapReduce, data are stored in the *Distributed File System (DFS)*. Each job usually retrieves input data from the DFS and stored the result back to DFS when the job is finished.

Each MapReduce job consists of two consecutive phases: the *map phase* and the *reduce phase*. The workers executing tasks in the map phase are called *mappers*. The tasks executed in the map phase are called *map tasks*. The workers executing tasks in the reduce phase are called *reducers*. The tasks executed in the reduce phase are called *reduce tasks*. In the map phase, the input data are fetched from the DFS and are divided into several chunks. Each data chunk will be assigned to one map task and processed independently. Each map task result consists of a collection of tuples with the format of $\langle key, value \rangle$. The tuples in each task result are sorted by *key*. Each task result is stored in its worker's local storage, and will be used as input in the reduce phase. In the reduce phase, each reduce task only processes the map task output tuples with certain *keys*. Therefore a special function called *partition* determines to which reduce task a map task output tuple should be assigned based on the content of the “*key*” in that tuple. Such a process is called *shuffle*. After the shuffle process, each reduce task will execute the *reduce* function to aggregate assigned tuples and generate the job output.

Figure 1 shows the workflow of a MapReduce application, Word Count. Word Count computes the number of occurrences of each word in the provided input text files. In the example, the input data are stored in the DFS and are broken down into three chunks. Each chunk is sent to a mapper (M1, M2 or M3). Each map task computes the occurrence of each

input chunk and stores the result in its local storage. After that, the map task output goes through the shuffle function *partition* and sends the tuples with key “Hello” to reducer R1 and sends other tuples to reducer R2. Each reduce task aggregates tuples with the same key (i.e., accumulate the occurrences of each distinct key) and outputs its result tuples into DFS. The generated result indicates the occurrences of each word in the input text files.

Figure 1 MapReduce Example: Word Count

MapReduce is usually deployed on a computer cluster in order to process computing tasks in a parallel manner. However, many users such as small companies and individuals do not want to or cannot afford investing a dedicated cluster to process MapReduce jobs. Fortunately, public cloud vendors offer services that enable users to set up virtualized clusters on demand and execute their MapReduce jobs on those clusters. As a return, the cloud vendor charges the cloud user based on the cluster size and the use time.

For example, Amazon EC2 and *Windows Azure* allow users to rent virtualized computers and run their applications on it. By using such a service, a MapReduce user can set up her own cluster based on the computation size and pay the fee only based on the resources she uses. Such a service still requires the cloud user to setup a cluster. Cloud vendors also offer MapReduce computing services (e.g., Amazon Elastic MapReduce),

which automatically set up MapReduce clusters based on user's requirement. With such services, users do not have to setup cluster by themselves. They only need to upload their jobs and data to the public cloud and initiate the computation. The cloud service will take care of other details.

2.2 Software Obfuscation

Software Obfuscation generally refers to a process of transforming a program P through an obfuscator O into a program $O(P)$ such that $O(P)$ has the equivalent functionality of P (correctness), is resilient against a reverse engineering attack (resilience) and is not significantly slower than P (efficiency). Collberg et al. [17] classify obfuscation methods into three categories: lexical obfuscation, data obfuscation and control flow obfuscation. Lexical obfuscation typically scrambles the program identifiers to confuse the attacker. Data obfuscation modifies the data structure to hide the information. Control flow obfuscation transforms the program to make the control flow unintelligible. Because control flow obfuscation is the most direct way of protecting the confidentiality of the program algorithm, in this dissertation, we explore that direction to protect the algorithmic confidentiality.

3 INTEGRITY ASSURANCE MAPREDUCE

In this chapter, we present the design of the Integrity Assurance MapReduce. We start with the system motivation. Following the motivation, we describe our system architecture and the design overview, define the system assumption and the attacker model, present the design details, and perform the theoretical analysis. Based on the theoretical analysis result, we present algorithms to search for the optimal system parameter values. We present the experimental results and discuss the related work in the last part of this chapter.

3.1 System Motivation

MapReduce usually requires to be deployed to a large computer cluster. Since many small companies or individuals cannot afford or do not want to invest in computer clusters of such a large scale, they usually will choose to outsource their computation to public computing service platforms such as Volunteer Computing [45][37] and Cloud Computing [38][39]. Volunteer Computing platforms have long been blamed as unaccountable due to the existence of malicious nodes [40][41]. Cloud Computing platforms have better security guarantee than volunteer computing platforms. However, it is still possible for a skillful hacker to breach the security of commercial clouds. For instance, [42] pointed out a security vulnerability that Amazon EC2 suffers from: some members of the Amazon EC2 community can create and upload malicious Amazon Machine Images which, if widely used, can flood the community with hundreds of infected virtual machine instances. Along the same line, Bugiel et al. [2] perform a systematic study of the security status of public AMIs and report various vulnerabilities of Amazon EC2.

The above threats put MapReduce customers in a dilemma: using public clouds has economic advantage but incurs the risk of getting incorrect computation results; on the

other hand, avoiding the public cloud completely (i.e., running everything “in house” or in the private cloud) can guarantee result accuracy, but there will be less economic benefit. Under this situation, an economical solution that can guarantee high result integrity is needed.

3.2 System Architecture and Design Overview

Based on the above motivations, we propose the Integrity Assurance MapReduce (IAMR). IAMR employs a hybrid cloud architecture, which consists of a private cloud and a public cloud. The master that is in control of the entire computation runs on a private cloud. Workers run on the untrusted public cloud. Since the public cloud is untrusted, the workers deployed to it can be malicious. We introduce a special type of workers, called *verifiers*, and deploy them on the private cloud. The verifier is trusted. It can be used to detect malicious workers on the public cloud. The key rationale of our solution is to retain control and the trust “at home”, while delegating the more resource-intensive computations to the public cloud. Since our research focuses on the integrity of computation, we assume the DFS is trusted. Therefore, we deploy the DFS on the public cloud (See Section 3.3) to reduce cross-cloud data transmission. The architecture of IAMR is shown in Figure 2.

Based on this architecture, we leverage the trustworthiness of the master and the verifiers to propose the result checking technique to enhance the result integrity. The result checking technique performs probabilistic task replication on the public cloud, performs probabilistic task verification on the private cloud, and maintains a trust management mechanism for each worker to distinguish malicious workers from benign workers. Due to the different characteristics of the map phase and the reduce phase, IAMR performs the result checking technique on different objects. In the map phase, the technique is

performed on map tasks. In the reduce phase, IAMR factors each reduce task into multiple *sub-tasks* and applies the technique on sub-tasks.

Figure 2 The Architecture of Integrity Assurance MapReduce

3.3 System Assumption and the Attacker Model

3.3.1 System Assumption

We make the following assumptions based on our proposed architecture. We assume that the private cloud is deployed within the user’s organization, and thus is trusted. In other words, the master and the verifiers are trusted. On the public cloud, we assume the infrastructure provided by the cloud provider, such as the virtualized hardware and the network, is trusted. However, we assume virtual images used by the cloud user are untrusted. That makes the workers running on the public cloud untrusted. Since our system only focuses on the MapReduce computation, we assume the Distributed File System (DFS) is trusted. For example, the integrity of DFS can be guaranteed by the storage integrity assurance techniques, such as [3] and [4].

In IAMR, the master requires that each worker who runs a task/sub-task to submit the hash value of the task/sub-task result to the master. We assume the submitted hash value to be consistent with the actual task/sub-task output. Such an assumption can be realized by applying the commitment-based protocol proposed in [8] (i.e., the previous worker commits the task output to the master by the hash value. The later worker who takes the previous task’s output as input will calculate the hash value of input, and send the

calculated hash value to the master. The master compares the two hash values to ensure the previous worker does not lie).

Finally, we assume that the MapReduce job has the following properties:

Property 1. The tasks/sub-tasks of that job should be deterministic. In other words, multiple executions of the same task/sub-task by honest workers should return the same result.

Property 2. The job should be big enough so that it will be broken down into a big number of tasks. For those jobs, each task's result should equally contribute to the entire job's result. We also assume that accepting a small number of incorrect task results should only affect the job's precision, without affecting its correctness. Therefore, the portion of tasks that return correct results determines the precision of a job result.

We argue that Property 2 is a reasonable assumption. Firstly, one primary reason for the cloud user to use public clouds is that the computation workload is significantly large. Otherwise, the user would perform the computation locally. Secondly, high parallelism is a necessary factor to address the challenge of computing load scaling up. In parallel computations, each task result usually is equally important to the job result. In fact, such classes of jobs widely exist in the field of statistics, machine learning, and data mining. For some computations whose accuracy can be affected by any incorrect task results, that computation usually can be broken down into multiple jobs. For the jobs whose accuracy is not sensitive, we can perform our proposed solution on those jobs. For other jobs, we can compute the job on the trusted private cloud. For example, some big data analytics computation is performed in a chain of MapReduce jobs (e.g., Apache Pig³, Apache

³ <http://pig.apache.org>

Mahout⁴, etc.). Our solution can be applied to the jobs that satisfy Property 2. For jobs that do not satisfy Property 2, we can compute those jobs on the private cloud directly.

3.3.2 Attacker Model

We model the attacker as an intelligent adversary that controls the malicious nodes on the public cloud. It receives and correlates information collected by malicious nodes and coordinates them to cheat at the best time in order to introduce as many errors as possible to the final result without being detected. For example, if the master replicates the same task on two malicious workers, the adversary can instruct them to return the same erroneous results (i.e., to collude), so that simply comparing the results cannot detect that error. We call such malicious workers *collusive workers*.

3.4 System Design

In both the map phase and the reduce phase, IAMR defines three types of tasks: the *original task*, the *replication task*, and the *verification task*. The original task and the replication task are executed on the public cloud. The verification task is executed in the verifier on the private cloud. The replication task repeats the original task's work to validate the original task's result. Since replication tasks are executed on untrusted workers, the verification tasks are launched non-deterministically to verify replication tasks' results in order to detect collusive workers.

In the map phase, the replication and verification task completely repeat the original map task's work. In the reduce phase, each original reduce task is broken down into a number of small parts (see Section 3.4.2). Each replication/verification reduce task repeats only one part of the original reduce task. Each part of a reduce task is called a *sub-task*. In both the

⁴ <http://mahout.apache.org/>

Map phase and the Reduce phase, IAMR performs a probabilistic *two-layer check* technique on each returned original task/sub-task result. Specifically, IAMR performs probabilistic task replications on the public cloud and performs probabilistic task verifications on the private cloud. In addition, IAMR maintains trust management for each worker. The master only accepts a worker’s task/sub-task results in a batch when all the tasks/sub-tasks in that batch do not fail on probabilistic two-layer checks.

The task/sub-task assignment in IAMR differs from the original MapReduce. Rather than passively waiting for the worker to ask for task/sub-task, the master of IAMR randomly selects a worker to assign a certain task/sub-task. When a worker finishes its assigned task/sub-task, IAMR requires the worker to return the result. In order to reduce the communication cost, the worker only returns the result’s hash value to the master. Since the replication task and the verification task are only used to evaluate their original task’s correctness, the actual result of the replication/verification task/sub-tasks will not be stored back to the DFS.

Since map and reduce phases have different characteristics (i.e., some jobs may contain a small number of reduce tasks. Yet each reduce task contains a significant workload), we propose different designs for the two phases. We present the system design for the map phase in Section 3.4.1 and the design for the reduce phase in Section 3.4.2.

3.4.1 Map Phase Integrity Check

IAMR applies the *two-layer check* on each returned map task result. In the first-layer, IAMR creates a replication task and assigns the task to another randomly selected worker. The replication task is assigned with a rule called *hold-and-test*. The hold-and-test rule is that a replicated task can be assigned to a worker only when the master receives its original

task result. By doing so, even if an original task is assigned to a malicious worker, the malicious worker cannot safely commit a cheat. This is because by the time the malicious worker receives a task, it cannot tell whether a replicated task corresponding to the assigned task will be generated, and whether this replicated task will be assigned to another collusive worker. If that malicious worker cheats and a replicated task is generated and assigned to a benign worker, the cheat will be detected (For the detailed analysis of the hold-and-test technique, please refer to Section 3.5.1). When a worker returns the replication task result, IAMR compares the original task result with the replication task result. If the two results are not consistent, at least one of the two workers is cheating. IAMR will generate a verification task and assign it to a verifier to determine the malicious worker(s). If the two results are consistent, IAMR will perform the second-layer check. In the second-layer check, IAMR creates a verification task and assigns it to a verifier to verify the consistent results. If the consistent results are different from the verification task result, we know that the two mappers are both malicious. The goal of the second-layer check is to detect collusive workers.

Performing two-layer check on each task will incur significant performance overhead. To reduce the overhead, IAMR creates the replication task and the verification task with a certain probability. Specifically, for each original map task, IAMR creates the replication task with a *replication probability*, and creates a verification task with a *verification probability*.

Since the two-layer check is performed in a probabilistic manner, there is a possibility that some incorrect results can evade the check. In order to overcome this drawback, IAMR performs a trust management technique to improve the job accuracy. The idea is that the

master performs the probabilistic two-layer check on each mapper for a period of time. The master accepts a mapper's original task results in a batch when that mapper did not fail any two-layer check during that period. The number of original tasks on which the master performs the probabilistic two-layer check before accepting their results is called the *batch size*. Notice that since the two-layer check is performed in a probabilistic manner, the original task that is not replicated or verified is also counted for the batch size. However, the replicated task will not be counted. The intuition of this mechanism is that observing a mapper for a longer period without detecting misbehavior gives the master more confidence that the mapper is benign.

In order to achieve the above mechanism, the master maintains a history cache for each mapper to record the original tasks (not the replicated task) that this mapper has completed and passed the probabilistic two-layer check. Specifically, the history cache for each mapper records the task id and the hash value of the task results of the original map tasks that mapper has computed. Note that the actual task result is buffered in the mapper's local storage. When a mapper accumulates batch size of tasks in its history cache without failing any two-layer check, the mapper becomes trusted temporarily. The master thus accepts all the task results buffered in the mapper's local storage and empties its history cache. After that, this mapper becomes untrusted again. The mapper has to accumulate another batch size of tasks on its history cache without failing the two-layer check in order to submit the next batch results to the master. If a mapper fails any two-layer check before submitting batch size of tasks, the mapper is determined to be malicious and is added to a black list. Meanwhile, the task results buffered in this mapper's local storage are discarded, and the corresponding tasks cached in the history cache will be rescheduled.

Figure 3 presents the workflow of IAMR. In this figure, $W1$ and $W2$ are two workers randomly chosen from the public cloud. The “*Arbitrate/Verify task*” step is a step executed by the verifier on the private cloud. The remaining components in the figure are all performed on the master. Notice that in this figure, IAMR applies the hold-and-test rule to assign the original and the replicated tasks: the “*replication*” decision (step 3) is made after $W1$ returns the original task result $R2$ (step 2). This assignment sequence makes it difficult for malicious workers to collude because they cannot predict whether the replication task will be assigned to another collusive worker. A detailed discussion about this rule is presented in Section 3.5.1.

Figure 3 The Workflow of Integrity Assurance MapReduce

If the total number of original map tasks in a job is less than the batch size, IAMR will directly assign all tasks to verifiers, since no mapper can accumulate batch size of tasks in its history cache and thus guarantee expected accuracy. This will not incur significant workload to the verifier if the batch size is a reasonable number. Meanwhile, the accuracy is still guaranteed. If the number of the original map tasks is large enough, a higher batch size will guarantee a higher accuracy, as shown in our simulation in Section 3.5.2.

3.4.2 Reduce Phase Integrity Check

In the reduce phase, the approach presented in Section 3.4.1 can be directly applied only if the number of reduce tasks is large enough (i.e., bigger than the batch size). However, in some applications, the reduce task number is smaller than the batch size, even though the computing workload for each reduce task is significant. For instance, the word count application with the default partition function contains only one original reduce task. However, this single task will process intermediate results generated from each map task, which can be a heavy workload. For example, our experiments showed that in a word count job processing a file with 628 Megabyte of text, the single reduce task will process 2.7M of records (1.07GB of data) in the input and generate 598K of records in the output, and take 262 seconds to complete. In this case, directly verifying the entire reduce task is expensive in terms of computation and communication cost. Therefore, we propose to break down each original reduce task into many sub-tasks and apply the two-layer check technique on each sub-task. We present the technical details in Section 3.4.2.1. Based on the basic reduce phase integrity check technique, we present the *request bucketing* technique in Section 3.4.2.2 to limit the sub-task number into a small value without sacrificing the result accuracy.

3.4.2.1 Break Down Reduce Tasks into Sub-tasks

Our reduce phase integrity check design is based on the following intuition. Remember that each record in MapReduce computation is in a format of $\langle key, value \rangle$ tuple. We observe that both the map task and the reduce task outputs are sorted by key. For each key in the reduce output, if we can precisely pinpoint the map output tuples that are related to that key, we can reproduce the portion of the reduce task that is related to that key. We call

each portion of reduce task as sub-task. Therefore, each original reduce task can be divided into multiple sub-tasks, each of which is related to one key. By applying two-layer check to each sub-task, we can guarantee high accuracy of the original reduce task. Here, we temporarily define each sub-task to cover one key. We will extend this concept later for practicality reason.

Our reduce phase integrity check uses the same high-level ideas as the map phase. Each original sub-task returns its result to the master in the form of a hash value (we call each returned sub-task result as a *report*). The master applies the first-layer (i.e., replication) and second-layer (i.e., verification) check on each report with replication probability and verification probability, respectively. The generation of a replication sub-task is decided after the report of an original sub-task is returned to the master (i.e., hold-and-test). We regulate that an original reduce task result is accepted by the master only when all its sub-tasks pass the probabilistic two-layer check, which is essentially a trust management mechanism. The batch size is therefore the number of sub-tasks in the original reduce task. If the number of sub-tasks in an original reduce task is smaller than the batch size, IAMR directly generates a verification reduce task to verify the entire original reduce task.

The above idea only works conceptually. In order to make it practical, we need to extend the concept of sub-tasks and reports, and overcome the following three challenges.

- 1) Creating a sub-task for each key would incur significant overhead. It is because in many cases, the amount of keys in a reduce task can be huge (e.g., a word count job with 628 Mega bytes of input text file will generate more than 598,000 keys).
- 2) The accuracy only relies on the two-layer checks of the sub-task reports submitted by the reducer. If a malicious reducer cheats on some sub-tasks but does not send

these reports to the master, the master would have no way to detect the error.

- 3) The replication and verification sub-tasks should efficiently locate the portion of map task output with the key they are interested in.

We address the first challenge by extending the concept of report and sub-task to cover a range of consecutive keys, instead of just one key. With this improvement, the number of sub-tasks will be reduced.

For the second challenge, IAMR requires that consecutive reports in the original reduce task must overlap in one key. In addition, the first and last report in each original reduce task should cover the first and the last key of the task output, respectively. The master will check those requirements when it receives reports. Since the reduce task result is sorted by key, this requirement ensures that no key in the output is skipped in the reports. In case that the master does not know the first and the last key in the original reduce task output, the master can insert dummy records in the job input data, which will generate reduce result tuples with predictable smallest and largest keys. For instance, when the type of the key in a job is integer, the master can insert special records with their keys as *Integer.MIN_VALUE* and *Integer.MAX_VALUE*, respectively. When the reduce task is complete, the job output can be sanitized by removing the output records related to those dummy input records.

For the third challenge, each map task in IAMR builds a *key table* to facilitate the record look up in the map task output, as shown in Figure 4. When a sub-task wants to fetch the map output within a certain key range (e.g., *Key 2* to *Key 9*) from the map task output file, it will locate the position through the key table of that task. In the original MapReduce, each map task stores the map output file locally, which consists of key-value pairs sorted

by key. Usually, the lengths of keys and values vary. Notice that multiple key-value pairs can have the same key (e.g., *Key 3*). For brevity, we call consecutive key-value pairs in the map output file with the same key as a *block*. Each record in the key table corresponds to a block in the map output file. It has three fields, indicating the start position of the block, the length of the key, and the length of the block. Since the lengths of keys and values vary, each access to a key on the map output file needs to go through the key table, which has the fixed record length. The key table records are sorted by key, IAMR can apply binary search to look up the records within the request key range. However, each key comparison in the binary search needs to fetch the key in the map output file through the key table. When the binary search finds the keys (e.g., *Key 3* through *Key 8*) between the request range (e.g., *Key 2* to *Key 9*), the key position and the record length direct the reducer to fetch the portion of map output.

Since it is the mapper who creates the key table, a malicious mapper can manipulate the content in its key table to fool IAMR. As a defense, IAMR requires each map task to submit the hash value of its key table to the master along with that of task result. Therefore the key table has the same accuracy as the map task result. The consistency between the hash value and the actual key table can be guaranteed by the commitment-based protocol [8].

Figure 4 Locating Map Output Between Key 2 and Key 9 using Key Table

Figure 5 shows the workflow of IAMR in the reduce phase when executing the word count application. Remember the word count application calculates the frequency of each word appeared in a collection of text files. For simplicity, our example only has two map tasks (*map 0* and *map 1*) and one original reduce task (*reduce 0*). As Figure 5 shows, each map task creates a key table (Step 1). When the original reduce task (*reduce 0*) starts to output (step 3), sub-task reports (e.g., *Report 1* and *Report 2*) are sent to the master sequentially. The report format is *<start key, end key, hash value of the output records covered in the key range>* (Step 4). Since consecutive reports must overlap in one key (According to the solution of challenge 2), the key “Driver” appears in both *report 1* and *report 2*. When the master receives *report 1* (Step 5), it launches the first-layer check by initiating a replication sub-task (with replication probability). The replication sub-task fetches input with a key range of (*Apple, Driver*) from each map task (*map 0* and *map 1*) through key tables (Step 7). When it completes reducing (Step 8), the replication sub-task sends a report to the master (Step 9), and the master compares the report with the original sub-task report (Step 10). If they are consistent, a second-layer check is performed to verify

the consistent results. The verification sub-task will be created if necessary (Step 11).

Figure 5 Workflow of IAMR in the Reduce Phase

3.4.2.2 Request Bucketing to Reduce Sub-task Number

One drawback of reduce phase design in Section 3.4.2.1 is that in order to achieve high accuracy, IAMR will generate a large number of sub-tasks. Each sub-task will be executed as an individual reduce task. Such a reduce task needs to connect to each map tasks, locate the map output position, and fetch only a small portion of data. In addition, setting up and tearing down each sub-task consume a certain amount of resource. As a result, a big number of sub-tasks can introduce a big performance delay. We propose the *request bucketing* technique to merge multiple sub-tasks into one reduce task to improve performance, without sacrificing the accuracy.

The idea of request bucketing is as follows. We create several *replication buckets* ready to receive replication sub-task requests. Each bucket has a limited capacity. Whenever the master receives a sub-task report from original reduce task and decides to generate a

replication sub-task (first-layer check), instead of generating a reduce task just for that single sub-task, the master *randomly* chooses a bucket and stores the sub-task request to this bucket. When a bucket achieves its capacity, the master generates a replication reduce task to compute all the sub-task requests in that bucket. Meanwhile, the bucket is emptied. The master will *randomly* choose a worker to execute this replication task. The replication task will send multiple replication sub-task reports back to the master, each of which will correspond to one original sub-task report. When a report passes the first-layer check, the master will apply the second-layer check to those sub-tasks. Similar to the replication buckets, the master also creates several verification buckets to accumulate the verification sub-task requests. When a verification bucket is full, the master will generate a verification reduce task to compute all the requests in that bucket.

Although the request bucketing technique merges multiple sub-tasks into one reduce task, it does not undermine the accuracy with the following reasons. First, the two-layer check is still performed on each original reduce sub-task. Second, the replication and verification sub-tasks are assigned to workers randomly.

3.5 Theoretical Study

3.5.1 Quantitative Analysis

Since in the map phase and the reduce phase, IAMR performs the same two-layer check technique (only on different objects), we can use a similar model to analyze both phases. Even though the request bucketing technique merges sub-tasks to execute, the assignment of sub-task requests to the buckets and the assignment of buckets to the workers are all randomized. This is equivalent to the effect of assigning each sub-task request to workers randomly.

Based on the above reason, we perform a unified analysis on both the map phase and the reduce phase in Section 3.5.1.1. The only difference is that in the map phase, our analysis focuses on “tasks”, while in the reduce phase, our analysis focuses on “sub-tasks”. To simplify our expression, we use the term “task” at some places. However, the reader should understand that it refers to the “sub-task” if it is discussing the reduce phase. We discuss security analysis specific for the request bucketing technique in Section 3.5.1.2

3.5.1.1 *Unified Analysis on the Map Phase and the Reduce Phase*

We first analyze the adversary strategy of malicious workers. Based on that, we perform quantitative analysis on accuracy and overhead.

Adversary Strategy: We denote the *malicious worker fraction* on the public cloud as m . We assume that the adversary controls all malicious workers. In other words, all malicious workers are collusive, and there exists only one collusive group. Assuming that the goal of the adversary is to inject as many errors as possible and yet not to reveal the malicious workers, we analyze the strategy of the adversary under IAMR as follows. Suppose a task is assigned to a malicious worker, two cases are possible for the adversary.

Case 1 If the adversary has not seen a similar task (i.e., the one with the same input) before, it has to make a decision on whether to cheat, and remembers the decision, the current task and the returned result. Due to the existence of hold-and-test rule, the adversary is not allowed to defer the decision to the time that it sees the replica of the current task. If the decision is not to cheat, the worker is obviously safe (i.e., not to be caught). If the decision is to cheat, the malicious worker can survive the first-layer check only when either the current task is not replicated, or the replica of the current task is assigned to another malicious worker.

Case 2 If the adversary has seen a similar task before, it is assured that the current task is a replication task. It can simply ask the worker to take the same action for the current task as the one it has seen before. In this case, it is guaranteed that the malicious worker will survive the first-layer check.

Since in Case 2 the adversary just follows its decision made previously in Case 1, the risk of revealing a malicious worker is essentially determined by the adversary’s decision in Case 1. Because the master controls task assignment and replication in a randomized manner, the adversary in Case 1 cannot predict whether cheating at the current task is safe or not. On the other hand, since the master constantly applies the two-layer check on tasks in a randomized manner, the adversary cannot tell whether cheating at the current task has a smaller chance of detection than cheating at other tasks. Therefore, the only thing the adversary can do in Case 1 is to make a random guess/predict in terms of whether the cheat can be detected or not. We model the adversary’s decision making behavior in Case 1 as a random variable, *cheat probability*, marked as c . Note that adversaries who cheat rarely (e.g., only cheat once in hundreds of tasks) can still fit in our model because we can set c as a small value close to 0.

We study the accuracy and the overhead of IAMR by modeling the system and performing the probability analysis. We model the system as two participating parties: the malicious worker, modeled with the malicious worker fraction (m) and the cheat probability (c); the IAMR, modeled with the replication probability (r), the verification probability (v) and the batch size (T). The system modeling parameters and the measurement metrics are shown in TABLE 1. We perform a series of probabilistic analysis and present our analysis result in Theorem 1.

Item	Name	Definition
m	Malicious worker fraction	The fraction of malicious workers on the public cloud.
c	Cheat probability	The probability that the adversary decides to cheat when he is not sure whether it is safe to cheat.
r	Replication probability	The probability that an original task/sub-task is replicated.
v	Verification probability	The probability that consistent task/sub-task results are verified.
T	Batch size	The number of tasks/sub-tasks a mapper/reducer has executed by passing the probabilistic two-layer check to make its batch of results to be accepted by the master.
L	Survival length	The expected number of batches a malicious worker can submit to the master before it is detected.
J	Job error rate	The ratio of incorrect results number to the total results number in one job.
O	Overhead	The expected number of extra executions for each task/sub-task performed on the public cloud.
V	Verifier overhead	The expected number of extra executions for each task/sub-task performed on the private cloud.

TABLE 1 IAMR System Modeling Parameters and Measurement Metrics

Theorem 1. Assuming that the assignment of tasks/sub-tasks is uniformly distributed across all workers on the public cloud, and the detected malicious workers are *not* added to the black list, the probability for a malicious mapper/reducer to survive after executing n original tasks is

$$S_n = (1 - cr + crm(1 - v))^n \quad (1)$$

The survival length of a malicious mapper/reducer is

$$L = S_T / (1 - S_T) \quad (2)$$

The job error rate is

$$J = m(c(1 - r) + crm(1 - v))(1 - cr + crm(1 - v))^{T-1} \quad (3)$$

Let

$$U = 1 - (1 - rc - rv + rcv + rcm - rcmv)rc(1 - m + mv)m \sum_{k=0}^{T-2} S_k$$

The *overhead* for each task/sub-task is

$$O = (m + r - mr - m(1-r)S_{T-1} + mr(1-v)(1-c+cm)(rc - rcm + rcmv) \sum_{k=0}^{T-2} S_k) / U \quad (4)$$

The *verifier overhead* for each task/sub-task is

$$V = ((1-m)r(cm + v - vmc) + mr(c + v - vc - cm + cmv)S_{T-1} + rcm(1-m+mv)(1+rv-rvc) \sum_{k=0}^{T-2} S_k) / U \quad (5)$$

Proof: S_n , the probability of a malicious mapper/reducer to survive after executing n original tasks/sub-tasks is the probability summation of all permutations on n independent events. Each event should fall into one of the below three cases.

1. The worker does not cheat. The probability in this case is $(1-c)$;
2. The worker cheats, but the task/sub-task is not replicated. The probability in this case is $c(1-r)$;
3. The worker cheats, and the task is replicated. However, the other worker executing the replication task/sub-task can collude with it. When the consistent results pass the first-layer check, the task result is not verified. The probability in this case is $crm(1-v)$.

By summing up the probability of different permutations of above three cases on n independent events, we have

$$S_n = \sum_{i=0}^n \sum_{j=0}^{n-i} \binom{n}{i} \binom{n-i}{j} (1-c)^i (c(1-r))^j (crm(1-v))^{n-i-j}$$

By applying the multinomial theorem, we have

$$S_n = (1 - cr + crm(1-v))^n$$

Setting $n=T$, we have S_T , the probability that a malicious mapper/reducer submit a batch of task/sub-task to a master. The probability that a malicious worker can submit exactly k batches to the master without detection (i.e., not detected in first k batches, but detected on

the $(k+1)^{th}$ batch) is $(S_T)^k \cdot (1 - S_T)$.

The expected number of batches a malicious worker can submit (*survival length*) is therefore.

$$L = \sum_{k=0}^{\infty} k \cdot (S_T)^k (1 - S_T) = \frac{S_T}{1 - S_T}$$

We are now deriving the value of J . Suppose the batch size is T , the scenario that the master accepts exactly k out of T incorrect task results consists of $(T-k)$ events that the worker does not cheat, and k events that the worker cheats but undetected. That is

$$\Delta_k = \sum_{i=0}^k \binom{T}{T-k} \binom{k}{i} (1-c)^{T-k} (c(1-r))^i (crm(1-v))^{k-i}$$

The expected number of tasks returning incorrect results in a batch (i.e. in T tasks) is

$$E = \sum_{k=0}^T (k \times \Delta_k) = \sum_{k=0}^T (k \times \sum_{i=0}^k \binom{T}{T-k} \binom{k}{i} (1-c)^{T-k} (c(1-r))^i (crm(1-v))^{k-i})$$

The *batch error rate* of T tasks is therefore

$$e = E / T$$

Since the assignment of tasks/sub-tasks on the public cloud is uniformly distributed on all workers and the malicious worker fraction m stays constant, we have the job error rate

$$J = m * e + (1 - m) * 0$$

$$= m * e$$

$$= \frac{m}{T} \sum_{k=0}^T (k \binom{T}{T-k} (1-c)^{T-k} (c(1-r))^i (crm(1-v))^{k-i})$$

We are now simplifying the equation J . By reorganizing equation J and applying multinomial theorem, we have

$$\begin{aligned}
J &= \frac{m}{T} \sum_{k=0}^T \binom{T}{T-k} (1-c)^{T-k} \sum_{i=0}^k \binom{k}{i} (c(1-r))^i (crm(1-v))^{k-i} \\
&= \frac{m}{T} \sum_{k=0}^T \binom{T}{T-k} (1-c)^{T-k} (c(1-r) + crm(1-v))^k
\end{aligned}$$

To simplify the formula representation, we define

$$A = 1 - c$$

$$B = crm(1 - v)$$

By replacing k with $T-l$, we have

$$J = \frac{m}{T} \sum_{l=0}^T ((T-l) \binom{T}{l} A^l B^{T-l})$$

Expanding $\binom{T}{l}$, we have

$$\begin{aligned}
J &= m \sum_{l=0}^T \left(\frac{(T-1)!}{l! (T-l-1)!} A^l B^{T-l} \right) \\
&= m \sum_{l=0}^T \binom{T-1}{l} A^l B^{T-l} \\
&= m \left(\sum_{l=0}^{T-1} \binom{T-1}{l} A^l B^{T-l} + \binom{T-1}{T} A^T \right)
\end{aligned}$$

By binomial definition, we have $\binom{T-1}{T} = \frac{(T-1)!}{T!(-1)!}$. By factorial definition

$n! = (n+1)! / (n+1)$, we have $0! = 1! / 1 = 1$ and $(-1)! = 0! / 0 = 1/0$. By replacing $\binom{T-1}{T}$ with

factorial form and replacing $(-1)!$ With $1/0$, we have

$$\begin{aligned}
J &= m \left(\sum_{l=0}^{T-1} \binom{T-1}{l} A^l B^{T-l} + \frac{(T-1)!}{T!(-1)!} A^T \right) \\
&= m \left(\sum_{l=0}^{T-1} \binom{T-1}{l} A^l B^{T-l} + \frac{1}{T \binom{1}{0}} A^T \right) \\
&= m(B(A+B)^{T-1} + 0) \\
&= m(c(1-r) + crm(1-v))(1-cr + crm(1-v))^{T-1}
\end{aligned}$$

In order to calculate the overhead and the verifier overhead, we divide the original tasks/sub-tasks into 11 different categories based on different conditions the task/sub-task may encounter, as shown in TABLE 2.

We summarize the probability, workload and verifier overhead of each category in TABLE 3. Here we define *workload* as the number of executions each task/sub-task has to run on the public cloud, marked as W . It includes the original task/sub-task execution and the task/sub-task overhead. Therefore, we have $W=I+O$, where I corresponds to the original task/sub-task and O corresponds to the overhead.

Since the categories summarized in TABLE 3 are mutual exclusive and exhaustive, we have $\sum_{i=1}^{11} P_i = 1$.

We calculate the expected workload by combining the probability and workload under each category.

$$W = P_1 + P_{11} + (1+W)P_4 + (2+W)(P_5 + P_6) + 2(P_2 + P_3 + P_7 + P_8 + P_9 + P_{10})$$

Condition1	Condition 2, given condition 1 is satisfied	Condition 3, given condition 2 is satisfied	Category Number
The task/ sub-task t is assigned to a malicious worker z , which survived in the current batch	--	t is not replicated.	1
		t is replicated, but z does not cheat.	2
		t is replicated but not verified. z cheats. But the replication task/sub-task is executed by another malicious worker.	3
The task/ sub-task t is assigned to a malicious worker z , which does not survive in the current batch	t is not the last task/sub-task in the current batch(i.e., t is the one being detected)	t is not replicated.	4
		t is replicated. z does not cheat. The results are not verified.	5
		t is replicated. z cheats. The error is not detected.	6
		t is replicated. z does not cheat. The results are verified.	7
	t is the last task/sub-task in the current batch (i.e., the one being detected).	--	8
The task/ sub-task t is assigned to a benign worker z .	t is replicated.	The corresponding replication task is assigned to a malicious worker. The malicious worker cheats.	9
		The corresponding replication task returns the same result as the original one.	10
	t is not replicated	--	11

TABLE 2 Categories of Tasks Based on Assignment Conditions

Category Number	Probability	Workload W	Verifier overhead V
1	$P_1 = m \binom{T}{1} \cdot \frac{1}{T} \cdot S_{T-1}(1-r)$	1	0
2	$P_2 = m \binom{T}{1} \cdot \frac{1}{T} \cdot S_{T-1}r(1-c)$	2	v
3	$P_3 = m \binom{T}{1} \cdot \frac{1}{T} \cdot S_{T-1}rcm(1-v)$	2	0
4	$P_4 = m \sum_{j=0}^1 \sum_{i=1}^{T-1} \binom{i}{1} \frac{1}{i} S_{i-1}(1-r)(rc(1-m))^j (rcmv)^{1-j}$	$1+W$	V
5	$P_5 = m \sum_{j=0}^1 \sum_{i=1}^{T-1} \binom{i}{1} \frac{1}{i} S_{i-1}r(1-c)(1-v)(rc(1-m))^j (rcmv)^{1-j}$	$2+W$	V
6	$P_6 = m \sum_{j=0}^1 \sum_{i=1}^{T-1} \binom{i}{1} \frac{1}{i} S_{i-1}rcm(1-v)(rc(1-m))^j (rcmv)^{1-j}$	$2+W$	V
7	$P_7 = m \sum_{j=0}^1 \sum_{i=1}^{T-1} \binom{i}{1} \frac{1}{i} S_{i-1}r(1-c)v(rc(1-m))^j (rcmv)^{1-j}$	2	1
8	$P_8 = m \sum_{j=0}^1 \sum_{i=0}^{T-1} S_i (rc(1-m))^j (rcmv)^{1-j}$	2	1
9	$P_9 = (1-m)rmc$	2	1
10	$P_{10} = (1-m)r(1-mc)$	2	v
11	$P_{11} = (1-m)(1-r)$	1	0

TABLE 3 The Probability, Workload and Verifier Overhead of Each Category

We calculate the expected verifier overhead by combining the probability and verifier overhead under each category.

$$V = vP_2 + VP_4 + VP_5 + P_7 + P_8 + P_9 + vP_{10}$$

By reorganizing the formula and replacing P_i with the value in TABLE 3, we have,

$$O = (m + r - mr - m(1 - r)S_{T-1} + mr(1 - v)(1 - c + cm)(rc - rcm + rcmv) \sum_{k=0}^{T-2} S_k) / U$$

$$V = ((1 - m)r(cm + v - vmc) + mr(c + v - vc - cm + cmv)S_{T-1} + rcm(1 - m + mv)(1 + rv - rvc) \sum_{k=0}^{T-2} S_k) / U$$

$$, \text{ where } U = 1 - (1 - rc - rv + rcv + rcm - rcmv)rc(1 - m + mv)m \sum_{k=0}^{T-2} S_k \quad \blacksquare$$

3.5.1.2 Analysis on Request Bucketing Technique

We introduce four parameters specific for the request bucketing technique, shown in TABLE 4.

Item	Name	Definition
K	task key number	The number of keys (records) generated by an original reduce task.
S	sub-task key number	The number of keys covered by a sub-task.
R	report number	The number of reports an original reduce task sends to the master.
B	bucket size	The maximum number of sub-task requests contained in a bucket.

TABLE 4 IAMR System Modeling Parameters for Reduce Phase

The total number of reports a master receives from one original reduce task is $R = \lceil K / S \rceil$.

In our design, an original reduce task result is accepted by the master when its submitted task number achieves the batch size. In other words, we can adjust the parameters to have the batch size T to be equal to the report number (i.e., $T=R$). When K is big enough, we can set T to a big value by adjusting S to ensure high accuracy. For example, in the word count application in Section 3.7.1.2, the single original reduce task output contains 598,000 keys.

In this case, in order to set T to 600 to ensure high accuracy, we can set S as $\lceil K / R \rceil = 997$.

When request bucketing is applied, the two-layer check is still applied to each sub-task. Also, the assignment of sub-tasks is still random. Therefore, request bucketing will not undermine the IAMR's accuracy. In addition, the number of sub-tasks is not reduced with the introduction of request bucketing. Only the replication/verification reduce task number is affected. Suppose the number of replication/verification reduce sub-tasks generated in a job is $\bar{R}$, without request bucketing, the replication/verification reduce task number is $\bar{R}$, since each sub-task will be executed in an independent task. However, with request bucketing, the number of replication/verification reduce tasks will be $\lceil \bar{R} / B \rceil$.

3.5.2 Simulation Result

We present several simulation results based on Figure 6 to analyze the relationships among accuracy, overhead, and other system parameters.

We first simulate the job error rate under different system parameters in Figure 6(a). The four curves show that when other parameters (c , v , r and m) are fixed, increasing batch size T will reduce the job error rate J , and when T is greater than 200, J is close to 0 for any parameter combinations in the figure. Moreover, when T and other parameters are fixed, J will be increased if malicious worker fraction m is increased or the replication probability r is decreased. For example, when T is 50 and r is 0.5, J increases from near 0 to 0.06 when m increases from 0.5 to 1.0; when T is 50 and m is 1.0, J increases from 0.06 to 0.15 when r drops from 0.5 to 0.3.

Figure 6 Simulation of IAMR Analysis

Figure 6(b) shows the relationship between cheat probability c and job error rate J with fixed T and v . According to the simulation, when T is 50, r is 0.5, and m is 0.5, the maximum J an adversary can achieve is less than 0.01. When m is 1.0, setting r as 0.5 can limit J to less than 0.09. The simulation also shows an interesting tradeoff between c and J : if c is too big, the malicious worker would be detected easily and thus its injected errors are rejected, resulting in a smaller J ; if c is too small, the number of injected errors is reduced, which also results in a smaller J .

Figure 6(c) shows the relationship between c and J when T is 600, v is 0.07, and r is 0.16. With this configuration, even if when m is 1.0, the maximum J the adversary can achieve is less than 0.06; when m is no larger than 0.5, the maximum J is close to 0.

Figure 6(d) shows the relationship between cheat probability c and survival length L when T is 50 and v is 0.15. We can see that L is generally very small when c is bigger than 0.02, which means that a malicious worker cannot survive IAMR checks for a very long time. Our experimental results in Figure 12 confirm this observation. However, L increases exponentially when c decreases from 0.02 to 0, which suggests that IAMR cannot remove very low-profile malicious workers (those that rarely cheat) quickly, but since such workers inject very few errors at the same time, IAMR can still guarantee very low job error rate in that case.

Figure 6(e) shows the tradeoff between job error rate J and overhead O , and Figure 6(f) shows the tradeoff between job error rate J and verifier overhead V , given different batch size T . For each curve in the figures, the top-left most point corresponds to the setting where T is 0, and the bottom-right most point corresponds to the case where T is 600. The difference of T values between adjacent points on each curve is 50. The figures show that when T is small (e.g., 50), a higher value of r results in a lower job error rate and higher overhead and verifier overhead. When T is big enough (e.g., greater than 200), different values of r do not make much difference in job error rate. However, a smaller value of r would bring a smaller overhead and verifier overhead limit. We find that on each curve, the points become denser with the increase of T and eventually concentrate to their outmost limits. This suggests that when T is big enough (e.g., bigger than 200), increasing T further would bring neither additional accuracy benefit, nor additional overhead or verifier overhead cost.

Note that Figure 6 assumes that m is constant, i.e., detected malicious workers are not eliminated. Yet in our implementation, detected malicious workers are eliminated, which

reduces errors. As a result, task/sub-task reschedule will be reduced, and the overhead and verifier overhead should be lower than the simulation result.

In summary, the simulation result shows that when setting T as a reasonably big number, IAMR can achieve a very low job error rate even if the malicious worker dominates the public cloud environment. Meanwhile, by increasing the batch size T , the job error rate decreases with the increase of overhead and verification overhead. However, both the overhead and the verification overhead are bounded when T increases.

3.6 Optimal Parameter Searching Algorithms

Simulations in Section 3.5.2 indicate that the job error rate is determined by several parameters. How to find an appropriate parameter setting to satisfy the user's requirement in the adversarial environment is a challenging problem. In this section, we model the interaction between the IAMR system and the attacker as a two-player zero-sum game, and explore the method to search for optimal parameter values that can satisfy the user's accuracy requirement and restriction settings.

3.6.1 Model the System as a Two-player Zero-sum Game

By studying the monotonicity of the job error rate formula (3) in Figure 6, we have the following theorem.

Theorem 2. When other parameters are fixed, the job error rate J will

- a) Monotonically decrease if the batch size T increases.
- b) Monotonically decrease if the replication probability r increases.
- c) Monotonically decrease if the verification probability v increases.
- d) Monotonically increase if the malicious worker fraction m increases.
- e) First increase and then decrease when the cheat probability c increases.

It is easy to prove the monotonicity of J by analyzing the sign of the partial derivative of (3) on T, r, v, m and c . Thus we skip the proof in this dissertation.

We also study the monotonicity of the job error rate when the attacker always chooses the best cheat probability.

Theorem 3. When other parameters are fixed and the attacker always chooses the best cheat probability, the job error rate J will

- a) Monotonically decrease if the batch size T is greater than 1 and increases.
- b) Monotonically decrease if the replication probability r increases.
- c) Monotonically decrease if the verification probability v increases.

Proof. Since J is a convex function of variable c , the attacker can search for the optimal value by solving the equation $\frac{\partial J}{\partial c} = 0$. By solving this equation, we have

$$c = \frac{1}{rT(1-m(1-v))} \quad (6)$$

When the attacker sets c as $\frac{1}{rT(1-m(1-v))}$, it can achieve the highest job error rate. By substituting c with (6) on (3), we have

$$J = m \frac{(T-1)^{T-1}}{T^T} \left(\frac{1}{r(1-m+mv)} - 1 \right) \quad (7)$$

According to (7), J is monotonically decreasing, when $T > 1$ is increasing, r is increasing or v is increasing. ■

Theorem 2 suggests that the IAMR system and the attacker determine the job error rate together. From the IAMR system's perspective, the job error rate will decrease if it

- a) Increases the batch size T , or
- b) Increases the replication probability r , or

- c) Increases the verification probability v .

From the Attacker's perspective, the job error rate will increase if it

- a) Increases the malicious worker fraction m , or $\square$
- b) Sets the cheat probability c closer to the optimal value maximize the job error rate.

This system forms a two-player zero-sum game. We therefore propose the *Interactive Gradient Descent (IGD)* algorithm to find the system parameter setting that makes the system achieve Nash Equilibrium, meanwhile guaranteeing the job error rate close enough to the user's accuracy requirement. Based on the IGD algorithm, we propose the *Tiered Interactive Gradient Descent (TIGD)* algorithm, which not only achieves the goal of IGD, but also considers user's restriction settings.

3.6.2 Interactive Gradient Descent Algorithm

In this section, we introduce the *Interactive Gradient Descent (IGD)* Algorithm, an algorithm that searches for optimal value of T to ensure the resulting job error rate close enough to user's requirement. In this algorithm, we assume the value of r and v are fixed value preset by the user. We present the Tiered Interactive Gradient Descent algorithm in Section 3.6.3 to remove this assumption and further improve the algorithm effect.

According to Theorem 2, from IAMR's perspective, setting T as large as possible surely can minimize the job error rate. However, setting T as a significantly large number requires a significantly large number of tasks in order to be accepted in a batch, which is not practical in real situations. On the other hand, the effect of decreasing the job error rate is weakening when T is increasing. Therefore, if a user can set a reasonable small value as the target job error rate and if IAMR can achieve that value, we assume the user is satisfied.

We call the user's target job error rate as the *acceptable job error rate*, marked as J_0 . Our goal is to find a value T that makes the job error rate close enough to J_0 no matter how the malicious worker behaves.

In order to achieve our goal, we analyze the behaviors of both IAMR and the attacker. Since IAMR and the attacker form a two-player zero-sum game, the two players have conflicting interests. For the attacker, she will maximize the job error rate J (i.e., to make J as close to 1.0 as possible). For IAMR, she will make the job error rate close to the acceptable job error rate J_0 as much as possible. We formalize the two players' interests by defining their utility functions. Finding a parameter setting that minimizes a player's utility function is equivalent to achieving that player's goal. In other words, for each player, in order to achieve her goal, she will search for a parameter setting that minimizes her utility function.

For IAMR, we define her utility function as follows.

$$U_{IAMR} = (J - J_0)^2$$

By substituting J with (3), we have

$$U_{IAMR}(T, r, v, m, c, J_0) = (m(c(1-r) + crm(1-v))(1-cr + crm(1-v))^{T-1} - J_0)^2 \quad (8)$$

The square operation in (8) ensures that as J approaches J_0 , the utility function becomes smaller. In other words, if IAMR can minimize its utility function, she can make J as close to J_0 as possible, which achieves her goal. Since we assume the value of r and v are fixed, to minimize the utility, IAMR needs to search for an appropriate value for T .

On the other hand, when IAMR chooses a value for T , r and v , the attacker will need to search for appropriate values for m and c to maximize the job error rate, and thereby to

achieve her goal. For the value of m , the attacker only needs to set it as large as possible, since a larger value of m will incur a higher job error rate. Therefore, we assume m to be the highest fraction of malicious workers the attacker can inject to the public cloud. In the worst case that the malicious worker dominates the public cloud, we assume m as 1.0. For the value of c , the attacker needs to find a proper value between 0 and 1 to maximize the job error rate (according to property e) in Theorem 2). We define the attacker's utility function as follows:

$$U_{ATT} = (1 - J)^2$$

By substituting J with (3), we have

$$U_{ATT}(T, r, v, m, c) = (1 - m(c(1 - r) + crm(1 - v))(1 - cr + crm(1 - v))^{T-1})^2 \quad (9)$$

The square operation in (9) ensures that as J approaches 1.0, the utility function becomes smaller. In other words, if the attacker can minimize its utility function, she can make J as large as possible, which achieves her goal. In order to minimize the utility function, the attacker has to find a suitable value for c .

Both IAMR and the attacker need to minimize their own utilities. However, the value of the utility function for each player is determined by the parameters controlled not only by herself but also by her opponent. We therefore propose the Interactive Gradient Descent algorithm that can help IAMR search for a parameter setting, which makes both parties have no incentive to deviate. In other words, the output of the algorithm achieves Nash Equilibrium in the two-player game. The algorithm is shown in Figure 7. The idea of the algorithm is to perform *fictitious play* [70] between the IAMR and the attacker, in which two players take turns to adjust the parameter they can control to minimize their own utility function value. Theorem 5 shows that after limited iterations, the algorithm will terminate.

Theorem 6 shows that the algorithm result ensures that the game achieves Nash Equilibrium. In other words, both parties have no incentive to deviate from the algorithm suggested parameter values. Theorem 7 shows that the algorithm returned job error rate will be close enough to the acceptable job error rate J_0 .

```

Interactive_Gradient_Descent(m, r, v, J_0, T_init, c_init){
1   T = T_init;
2   c = c_init;
3   J_IAMR = 0;
4   J_ATT = 1;
5   while(|J_IAMR-J_ATT|<δ){
6       result = minimizeUiamr(U_IAMR, T, r, v, m, c, J_0);
7       J_IAMR = result.getNewJ();
8       T = result.getNewT();
9       result = minimizeUatt(U_ATT, T, r, v, m, c);
10      J_ATT = result.getNewJ();
11      c = result.getNewC();
12  }
13  T_opt = T;
14  c_opt = c;
15  J_opt = J_ATT;
16  return T_opt, c_opt, J_opt;
}

```

Figure 7 Interactive Gradient Descent Algorithm

As shown in Figure 7, the IGD algorithm takes accepted job error rate J_0 , m , r , v , the initial value of T (T_{init}) and the initial value of c (c_{init}) as arguments and returns a suggested value of T (T_{opt}), which guarantees that the job error rate does not exceed J_{opt} , no matter what cheat probability the attacker chooses. In fact, the optimal value of c the attacker can choose to maximum the job error rate is the suggested c_{opt} . Note that in this algorithm, m , r and v are fixed values, which do not change during the algorithm execution.

The algorithm performs multiple iterations until converge. In each iteration, it first searches for the value T that minimizes the utility function U_{IAMR} on variable T , with the current value c fixed (In the function *minimizeUiamr* in line 6 of Figure 7). This step simulates the situation that when the attacker sets a value for c , IAMR searches for the optimal value of T that makes J close to J_0 as much as possible. After this step, the

corresponding value of J is recorded in J_IAMR , and the value of T is updated with the new value (line 7 and 8). After that, it invokes the function *minimizeUatt* (line 9) to search for a value of c that minimizes the attacker's utility function U_{ATT} . This step simulates the situation that when IAMR sets a value for T , the attacker searches for the optimal value of c that makes J as high as possible (i.e., as close to 1 as possible). After this step, the corresponding value of J is recorded in J_ATT , and the value of c is updated with the new value (line 10 and 11). After each iteration, the algorithm checks if the difference between J_IAMR and J_ATT are reduced to a small value (i.e., the variable δ). If true, the iteration terminates and the latest value of T , c and J_ATT are returned; otherwise, the algorithm proceeds to the next iteration.

Since U_{IAMR} is differentiable on variable T , we can perform gradient descent on variable T in function *minimizeUiamr* to minimize U_{IAMR} . Since U_{ATT} is differentiable on variable c , we can perform gradient descent on variable c in function *minimizeUatt* to minimize U_{ATT} . Gradient descent algorithm can search for the local minimum value for a given function. Since the two utility functions are both convex functions, whose local minimum value is also the global minimum value, the return of the gradient descent algorithm will always be the optimal parameters that minimize the utility. The two function implementations are standard gradient descent procedure.

The pseudo code of *minimizeUiamr* is in Figure 8. The algorithm adjusts the value of T in the negative direction of the utility function gradient in multiple iterations until the utility function value converges. The implementation of *minimizeUatt* is similar to Figure 8.

```

0  minimizeUiamr( $U_{IAMR}$ , init_T, init_r, init_v, init_m, init_c, J_0){
1      old_utility =  $U_{IAMR}$ (init_T, init_r, init_v, init_m, init_c, J_0);
2      old_T = init_T;
3      do{
4          old_utility = new_utility;
```

```

5       $T = T - \gamma^* \frac{\partial U_{IAMR}}{\partial T};$ 
6      new_utility =  $U_{IAMR}(T, \text{init}_r, \text{init}_v, \text{init}_m, \text{init}_c, J_0)$ ;
7      }while(|old_utility-new_utility|> $\epsilon$ )
8      return { $J(T, \text{init}_r, \text{init}_v, \text{init}_m, \text{init}_c)$ ,  $T$ };
9  }
```

Figure 8 Minimize U_{IAMR} with Gradient Descent on T

```

0  minimizeUatt( $U_{ATT}$ , init_T, init_c, init_r, init_v, init_m){
1      old_utility =  $U_{ATT}(\text{init}_T, \text{init}_r, \text{init}_v, \text{init}_m, \text{init}_c)$ ;
2      old_c = init_c;
3      do{
4          old_utility = new_utility;
5           $c = c - \gamma^* \frac{\partial U_{attacker}}{\partial c};$ 
6          new_utility =  $U_{ATT}(\text{init}_T, \text{init}_r, \text{init}_v, \text{init}_m, c)$ ;
7      }while(|old_utility-new_utility|> $\epsilon$ )
8      return { $J(\text{init}_T, \text{init}_r, \text{init}_v, \text{init}_m, c)$ ,  $c$ };
9  }
```

Figure 9 Minimize U_{ATT} with Gradient Descent on c

The only difference is that the partial derivative is performed on function U_{ATT} and on variable c . We show its pseudo code in Figure 9. The accuracy and the efficiency of the gradient descent depend on two parameters: the step size γ and the convergence threshold ϵ . If the step size γ is too small, the convergence of gradient descent algorithm will be very slow. If the step size γ is too large, convergence will initially be very fast, but the algorithm will oscillate about the optimum. Ideally, the convergence threshold ϵ in Figure 8 and Figure 9 and δ in Figure 7 should be set as small as possible. However, considering the algorithm execution speed, the value should be set as practically small. We will discuss the choice of all the parameters in Section 3.7.2.

The following theorems show that, the IGD algorithm in Figure 7 will converge after a limited number of iterations; the return parameter values guarantee Nash Equilibrium and the J_{opt} returned from the algorithm is close enough to J_0 .

Theorem 4. For any two consecutive iterations in IGD algorithm, suppose the algorithm generates J_{IAMR_i} and J_{ATT_i} in the former iteration, and generates $J_{IAMR_{i+1}}$ and $J_{ATT_{i+1}}$ in the later iteration. We have

$$0 < J_IAMR_{i+1} < J_ATT_{i+1} < J_ATT_i \quad (8)$$

Proof: It is trivially to see that $0 < J_IAMR_{i+1} < J_ATT_{i+1}$. It is easily to prove that in (6), when J_0 is small enough, U_{IAMR} is monotonically decreasing with the increase of T . Consider any two consecutive iterations in Figure 7, namely iteration i and iteration $i+1$. We mark the resulting J_IAMR (line 7), J_ATT (line 10), T (line 8) and c (line 11) in iteration i as J_IAMR_i, J_ATT_i, T_i and c_i . In order for the iteration $i+1$ to receive a minimum job error rate after calling *minimizeUiamr* at line 6 of the IGD Algorithm in Figure 7, T will be increased. In other words, $T_{i+1} > T_i$. Since J is monotonically decreasing with the increase of T (according to Theorem 2), which makes the minimum of U_{ATT} monotonically increasing with the increase of T , we have

$$\text{Min}(U_{ATT_i}) < \text{Min}(U_{ATT_{i+1}}).$$

Therefore, $J_ATT_{i+1} < J_ATT_i$. ■

Theorem 5. The IGD algorithm will terminate after a limited number of iterations.

Proof: According to Theorem 4, J_ATT is monotonically decreasing in each iteration. We prove that after limited number of iterations, $|J_IAMR - J_ATT|$ will be no larger than an arbitrarily small positive number δ . By contradiction, we assume $|J_IAMR - J_ATT|$ is always larger than a small value δ , since J_ATT is monotonically decreasing, suppose after k iterations, J_ATT decreases to $J_ATT_k < \delta$, since $J_IAMR_k < J_ATT_k$ and $|J_IAMR_k - J_ATT_k| < \delta$, we have $J_IAMR_k < 0$, which contradicts with Theorem 4. Therefore, setting the threshold in the IGD algorithm in Figure 7 as δ will make the algorithm terminate. Since δ is an arbitrarily small positive number, setting threshold as any positive value will guarantee the termination of the IGD algorithm. ■

Theorem 6. The IGD algorithm returns a set of parameter setting that form Nash

Equilibrium.

Proof: Firstly, since the utility functions U_{IAMR} and U_{ATT} are convex functions. They both have no more than one minimum value. Therefore, the local minimum values generated by the gradient descent algorithm are global minimum values.

Suppose the IGD algorithm returns J_{opt} , c_{opt} and T_{opt} . According to the definition of the algorithm, T_{opt} ensures that the IAMR system's utility function U_{IAMR} achieves the global minimum value, when the attacker sets the cheat probability as c_{opt} . Also, c_{opt} ensures that the attacker's utility function U_{ATT} achieves the global minimum value, when IAMR system sets the batch size as T_{opt} . Therefore, for IAMR system, changing T from T_{opt} to any other value T' will increase the U_{IAMR} , if the attacker keeps the setting of c as c_{opt} . The IAMR system therefore has no incentive to deviate from T_{opt} . Similarly, for the attacker, changing c from c_{opt} to any other value c' will increase U_{ATT} , if IAMR system keeps the setting of T as T_{opt} . The attacker therefore has no incentive to deviate from c_{opt} . Since both players have no incentive to deviate from the IGD algorithm returned parameter values, Nash Equilibrium is achieved. ■

Theorem 7. The IGD algorithm returned job error rate J_{opt} satisfies user's accuracy requirement. In other words, J_{opt} is close enough to J_0 .

Proof: Suppose the IGD algorithm in Figure 7 terminates when the value of J_{IAMR} is close enough to J_{ATT} at iteration i , we have $|J_{IAMR_i} - J_{ATT_i}| < \delta$. According to Theorem 2, J is monotonically decreasing when T increases. It is also easy to prove that $\lim_{T \rightarrow \infty} J = 0$. Therefore, for any J_0 that is close to 0, there exists a T that makes $J = J_0$. In other words, the minimum of U_{IAMR} is 0. A gradient descent algorithm performed in Figure 8 can find a local

minimum of U_{IAMR} , which is a global minimum due to the fact that U_{IAMR} is a convex function. Therefore the invocation of $minimizeUiamr()$ at line 6 of the IGD algorithm in Figure 7 returns an J_{IAMR_i} , which makes $|J_{IAMR_i} - J_0| < \zeta$, where ζ is an arbitrary small positive number. Since $J_{opt} = J_{ATT_i}$, we have $|J_{opt} - J_0| < \zeta + \delta$. Since ζ and δ are both arbitrary small values, J_{opt} is close enough to J_0 ■

Since the IGD algorithm returned job error rate is close enough to the accepted job error rate and the attacker cannot further increase the job error rate, the parameter setting returned from the IGD algorithm is the optimal setting. Therefore, by setting T with the IGD returned T_{opt} , IAMR can achieve the lowest job error rate upper bound J_{opt} .

3.6.3 Tiered Interactive Gradient Descent Algorithm

According to IGD algorithm, when m is large, if J_0 is set to a very small value, the algorithm will return a large value of T . For example, according to TABLE 9, when J_0 is set to 0.001, m is 1.0, the IGD algorithm will return a value of T as large as 27,796. It is because the effect of decreasing job error rate is weakened when T increases. For example, according to Figure 6 (a), if r is set to 0.5, m is set to 1.0, when the batch size T increases from 0 to 50, the job error rate J drops from 0.47 to 0.07; when the batch size T further increases from 50 to 100, the job error rate J only drops from 0.07 to 0.01. On the other hand, we also observe that increasing the value of v or r can effectively decrease the job error rate, especially when the value of v or r is small. Hence, in order to achieve a small job error rate, instead of unlimitedly increasing T , increasing v or r will be a good option.

On the other hand, the characteristics of IAMR and the cloud-based MapReduce service determine that guaranteeing high result integrity in MapReduce computation is not free. A user usually has to consider the several system restrictions when using IAMR to compute

their jobs. Specifically, the restrictions include the following aspects:

1. The total number of tasks in a job. Even if increasing the batch size T can help decreasing the job error rate, the IAMR system requires the total number of tasks no less than the value of T . Hence the batch size T should be no greater than the total task number. It should have a limit. We call the maximum value of T that IAMR can set as the *maximum batch size*. We marked the maximum batch size as T_{max} .
2. The number of replicated tasks. Since the computing service provider usually charges the user by the time of computing, assuming each task takes the similar amount of time to be executed, the extra financial cost is therefore proportional to the number of replicated tasks. Hence the replication probability r should have a proper upper limit to ensure the extra financial cost paid to the public cloud within user's budget. We call the maximum value of r that IAMR can set as the *maximum replication probability*, marked as r_{max} .
3. The number of verified tasks. Verifying task requires users to consume their own computing resources, which may not be free (e.g., it requires users to invest on their own IT infrastructures) or time consuming (e.g., it needs to download data from computing service providers). Hence the verification probability v should also be bounded. We call the maximum value of v that IAMR can set as the *maximum verification probability*, marked as v_{max} .

We propose the Tiered Interactive Gradient Descent (TIGD) algorithm, which ensures that the IAMR's system setting is within the user's system restriction settings (T_{max} , r_{max} and v_{max}), meanwhile returning a job error rate as close to J_0 as possible.

This TIGD algorithm is based on the IGD algorithm. It performs three sets of interactive gradient descent algorithms to adjust T , r , and v , respectively. The sequence of the three interactive gradient descents is determined by the importance of the three system restriction aspects. The interactive gradient descent on T and c is firstly performed because T can be increased “financially free”, as long as T does not exceed T_{max} . The interactive gradient descent on r and c is performed before the interactive gradient descent on v and c because the task replication is performed on the public cloud, and the task verification is performed on the private cloud. Increasing task verification probability means putting more computing workload on the private cloud, which increases IT infrastructure investment and involve more cross-cloud communication (It is because that DFS is deployed on the public cloud). However, increasing replication probability only increases the workload on the public cloud, which is economically more practical.

The TIGD algorithm is shown in Figure 10. The algorithm accepts the following parameters: the user’s evaluation on m (m_0), accepted job error rate (J_0); the initial replication probability and the maximum replication probability (r_{init} and r_{max}); the initial verification probability and the maximum verification probability (v_{init} and v_{max}); the initial batch size and the maximum batch size (T_{init} and T_{max}); and the initial value of cheat probability (c_{init}). The algorithm returns a suggested parameter setting, T_{opt} , r_{opt} , v_{opt} , which guarantees that the job error rate does not exceed J_{opt} , regardless of the cheat probability the attacker chooses (In fact, the optimal value of c the attacker can choose to maximum the job error rate is c_{opt}).

```

Tiered_Interactive_Gradient_Descent(m_0, J_0, r_init, r_max, v_init, v_max, T_init,
1 T_max, c_init){
2     T = T_init;
3     c = c_init;
4     r = r_init;
5     v = v_init;
6     J_IAMR = 0;
7     J_ATT = 1;
8     while(|J_IAMR-J_ATT|<δ){
9         gdResult = minimizeUiamrOnT(U_IAMR, T, r, v, m, c, J_0, T_max);
10        J_IAMR = gdResult.getNewJ();
11        T = gdResult.getNewT();
12        gdResult = minimizeUatt(U_ATT, T, r, v, m, c);
13        J_ATT = gdResult.getNewJ();
14        c = gdResult.getNewC();
15    }
16    if(|J_ATT-J_0|<δ){
17        J_IAMR = 0;
18        J_ATT = 1;
19        while(|J_IAMR-J_ATT|<δ){
20            gdResult = minimizeUiamrOnR(U_IAMR, T, r, v, m, c, J_0, r_max);
21            J_IAMR = gdResult.getNewJ();
22            r = gdResult.getNewR();
23            gdResult = minimizeUatt(U_ATT, c, T, r, v, m);
24            J_ATT = gdResult.getNewJ();
25            c = gdResult.getNewC();
26        }
27    }
28    if(|J_ATT-J_0|<δ){
29        J_IAMR = 0;
30        J_ATT = 1;
31        while(|J_IAMR-J_ATT|<δ){
32            gdResult = minimizeUiamrOnV(U_IAMR, T, r, v, m, c, J_0, v_max);
33            J_IAMR = gdResult.getNewJ();
34            v = gdResult.getNewV();
35            gdResult = minimizeUatt(U_ATT, c, T, r, v, m);
36            J_ATT = gdResult.getNewJ();
37            c = gdResult.getNewC();
38        }
39    }
40    T_opt = T;
41    r_opt = r;
42    v_opt = v;
43    c_opt = c;
44    J_opt = J_ATT;
45    return T_opt, r_opt, v_opt, c_opt, J_opt;
46 }

```

Figure 10 Tiered Interactive Gradient Descent Algorithm

The algorithm performs three sets of fictitious plays between the IAMR and the attacker. Each set of fictitious play performs a modified IGD algorithm. In those modified IGD algorithms, the *restricted gradient descent algorithm*, instead of the standard gradient descent algorithm, is performed on the variables T , r , and v (in line 9, 20, 32 in Figure 10).

However, it still performs standard gradient descents on variable c because we assume the attacker does not have any restriction concerns.

The three IGDs in Figure 10 are performed on variables T and c , r and c , and v and c , sequentially. The restricted gradient descent on T is shown in Figure 11. In this algorithm, if the variable's value achieves its upper bound before the utility function converges, the algorithm will terminate. For instance, when the restricted gradient descent is performed on variable T in Figure 11, the gradient descent algorithm terminates either when the utility function converges (in line 9), or when T achieves T_max (in line 6), whichever is earlier. The restricted gradient descent on r and v are similar to Figure 11, except that the variables to perform compute are r and v . We skip the pseudo code in this dissertation.

```

0  minimizeUiamrOnT(U_IAMR, init_T, init_r, init_v, init_m, init_c, J_0, T_max){
1      old_utility = U_IAMR(init_T, init_r, init_v, init_m, init_c, J_0);
2      T = init_T;
3      do{
4          old_utility = new_utility;
5          T = T - γ *  $\frac{\partial U_{IAMR}}{\partial T}$ ;
6          if (T >= T_max)
7              break;
8          new_utility = U_IAMR(T, init_r, init_v, init_m, init_c, J_0);
9      }while(|old_utility - new_utility| > ε)
10     return T;
11 }

```

Figure 11 Minimize U_{IAMR} with Restricted Gradient Descent on T

The TIGD algorithm in Figure 10 first performs an interactive gradient descent on T and c (line 8-15). This iteration will try to make J as close to J_0 as possible, meanwhile guaranteeing T not exceeding T_max . If the generated job error rate is not close enough to J_0 , the algorithm will perform another interactive gradient descent on r and c (line 19-26). In this iteration, T is set as a fixed value generated from the last IGD. This iteration will make J further closer to J_0 , meanwhile guaranteeing r not exceeding r_max . If the resulting J is still not close enough to J_0 , a final round of interactive gradient descent will be

performed on v and c (line 31-38). The resulting J_{opt} in line 44 is the final job error rate that is closest to J_0 . Meanwhile, the resulting T_{opt} , r_{opt} and v_{opt} will not exceed the user's system restriction setting (i.e., T_{max} , r_{max} and v_{max}).

Notice that the TIGD algorithm does not guarantee that the resulting job error rate J_{opt} is arbitrarily close to J_0 . If the values of T_{max} , r_{max} and v_{max} are too low, the resulting J_{opt} may not be close enough to J_0 . In this situation, increasing T_{max} , r_{max} or v_{max} will further reduce the difference between J_{opt} and J_0 . However, we can prove that J_{opt} is the lowest job error rate that IAMR can achieve under current system restriction setting.

The following theorem proves that, the TIGD algorithm in Figure 10 will terminate after a limited number of iterations; the parameters suggested from the algorithm form Nash Equilibrium under the system restriction setting; the job error rate suggested from the algorithm is the lowest upper bound under the system restriction settings.

Theorem 8. The TIGD algorithm will terminate after a limited number of iterations.

Proof: In Theorem 5, we have proved that the IGD algorithm in Figure 7 will converge. With the similar technique, we can prove that the IGD on r and c and the IGD on v and c will converge, respectively. The difference between the TIGD algorithm and the IGD algorithm is that the former uses restricted gradient descents in each set of iterations. Since the restricted gradient descent only adds one more termination condition for the current set of iteration, it only makes the current iteration terminate before the utility function value converges. Hence, it will not hinder the convergence at any round of IGD in the TIGD algorithm. Therefore, the TIGD algorithm will terminate. ■

Theorem 9. The TIGD algorithm returned parameters, T_{opt} , r_{opt} , v_{opt} and c_{opt} , form Nash Equilibrium under the system restriction setting.

Proof: The TIGD algorithm in Figure 10 can terminate under three conditions:

- a. After the first IGD, the resulting job error rate is close enough to J_0 (i.e., the predicate $|J_ATT - J_0| < \delta$ in line 16 returns *false*);
- b. After the first IGD, the predicate in line 16 returns *true*. Also, after the second IGD, the resulting job error rate is close enough to J_0 (i.e., the predicate $|J_ATT - J_0| < \delta$ in line 28 returns *false*);
- c. The predicates in line 16 and line 28 both return *true* and the third IGD is performed.

From the attacker's perspective, in each case, the resulting value of c in the last IGD will be the optimal value, which makes the attacker's utility function achieve minimum. Therefore, the attacker has no incentive to deviate from this value. From IAMR's perspective, it will end up with two possibilities:

1. The resulting J_opt is close enough to J_0 . The possible terminate condition can be either a, b, or c.
2. The resulting J_opt is not close enough to J_0 . The possible terminate condition can only be c.

For possibility 1, the resulting utility function U_{IAMR} already achieves its minimum value. IAMR has not incentive to deviate from the parameters suggested from the TIGD algorithms. For possibility 2, the utility function U_{IAMR} does not achieve its minimum value. Increasing the value of T , r or v can further reduce the job error rate and thereby reducing U_{IAMR} . However, in this case, the algorithm returned T_opt , r_opt , and v_opt already achieve their maximum bounds, T_{max} , r_{max} and v_{max} , respectively. Further increasing those values will violate the system restriction setting. Therefore, IAMR still has no incentive to

deviate from T_{opt} , r_{opt} or v_{opt} . ■

Theorem 10. The resulting value J_{opt} returned from the TIGD algorithm either is close enough to the accepted job error rate J_0 or is the closest value to J_0 under the system restriction setting.

Proof: The algorithm will terminate with two possibilities:

1. The returned J_{opt} is close enough to J_0 . That is, $|J_{opt}-J_0|<\delta$.
2. The returned J_{opt} is not close enough to J_0 . That is, $|J_{opt}-J_0|\geq\delta$.

For case 1, the returned J_{opt} is close enough to J_0 . For case 2, the algorithm returned T_{opt} , r_{opt} and v_{opt} will achieve the maximum upper bounds T_{max} , r_{max} and v_{max} . By contradiction, if there exists another optimal job error rate $J_{opt}'<J_{opt}$, according to Theorem 3, J_{opt}' 's corresponding parameters T' , r' and v' will satisfy at least one of the following conditions: $T_{opt}'>T_{opt}$, $r_{opt}'>r_{opt}$, $v_{opt}'>v_{opt}$, which will violate the system restriction setting. Therefore, J_{opt} is closest value to the acceptable job error rate under the system restriction setting in case 2. ■

The TIGD algorithm can find parameter setting that either guarantees the resulting job error rate close enough to J_0 , or guarantees the resulting job error rate to be the lowest upper bound within the system restriction setting. Therefore, the TIGD algorithm returned parameter setting is an optimal setting. By setting system parameters with T_{opt} , r_{opt} and v_{opt} , IAMR can obtain the lowest job error rate upper bound.

3.7 Experimental Results

We performed two sets of experiments to evaluate the IAMR system. The first set is to measure the result checking technique described in Section 3.4. Specifically, our goal is to

verify the correctness of the theoretical analysis result and measure the performance overhead. The second set of experiments is to measure the optimal parameter searching algorithms described in Section 3.6. Since the IGD algorithm is a special case of TIGD algorithm, our experiments mainly focus on the TIGD algorithm.

3.7.1 Experiments on the Result Checking Technique

We implemented a prototype system based on Hadoop MapReduce and deployed it to a hybrid cloud consisting of a local private cloud and Amazon EC2. The experiment environment consists of two components: a Linux server (2.93 GHz, 8-core Intel Xeon CPU and 16 GiB of RAM) is deployed on a private cloud, running both the master and a verifier. Twelve Amazon EC2 micro instances are running as slave workers (Amazon Linux AMI 32-bit, 613 MB memory, Shared ECU, Low I/O performance).

We perform experiments on the map phase and the reduce phase separately to measure the job error rate, overhead, verifier overhead, and performance overhead. To compare the performance overhead, we set the baseline as a standard MapReduce cluster consisting of thirteen nodes deployed on Amazon EC2. Each node is a micro instance. Out of the 13 nodes, one is running as the master, and the other 12 nodes running as workers.

3.7.1.1 Map Phase

We measure the job error rate, overhead and verifier overhead of IAMR by running a word count MapReduce job (introduced in Figure 1) in an environment with malicious MapReduce workers. We simulate such malicious workers by implementing the adversary’s strategy described in Section 3.5.1. The word count job consists of 800 map tasks and one reduce task. We set T and v as fixed values and vary other parameters with different value combinations.

The experimental results in TABLE 5 indicate that in all parameter combinations, IAMR can keep a very low job error rate.

<i>System Setting</i>	T=50, v=0.15, m=0.5								
<i>c</i>	0.1			0.5			1.0		
<i>r</i>	0.3	0.5	1.0	0.3	0.5	1.0	0.3	0.5	1.0
$J_{map} (\%)$	1.31	0.58	0.14	1.08	0.28	0.05	2.25	0.76	0.0
$O_{map} (\%)$	46.3	66.3	126.7	47.5	70.1	122.6	51.4	74.6	118.8
$V_{map} (\%)$	4.9	8.3	18.1	6.3	11.5	21.5	9.5	15.4	24.5

<i>System Setting</i>	T=50, v=0.15, m=0.167								
<i>c</i>	0.1			0.5			1.0		
<i>r</i>	0.3	0.5	1.0	0.3	0.5	1.0	0.3	0.5	1.0
$J_{map} (\%)$	0.48	0.04	0.0	0.19	0.45	0.02	0.11	0.0	0.0
$O_{map} (\%)$	33.6	53.7	100.6	32.8	51.5	103.6	34.9	53.5	103.6
$V_{map} (\%)$	4.7	7.8	15.8	4.6	7.3	15.1	10.4	8.5	16.9

TABLE 5 The Job Error Rate, Overhead and Verifier Overhead of Wordcount Application with Map Phase Integrity Check

Overall, the maximum job error rate is 2.25% and the minimum is 0. The changing trend of the experimental results is consistent with the simulation result in Section 3.5.2. For example, when m and c are fixed (m is 0.167 and c is 0.1), the job error rate J drops from 0.48% to 0 when r increases from 0.3 to 1.0. When m and r are fixed (m is 0.5 and r is 1.0), the job error rate J drops from 0.14% to 0 when c increases from 0.1 to 1.0. On the other hand, a higher value of r incurs a higher overhead and a higher verifier overhead. For example, when r is 1.0, the average overhead is 112%, and when r is 0.3, the average overhead is 41%. We note that the experiment overhead and verifier overhead is lower than the simulation result in Figure 6(e) and (f), respectively. This fact confirms our prediction in Section 3.5.2: since Figure 6 assumes m as a constant value, its estimation on the overhead and the verifier overhead should be higher than the experiment result.

We observe that in each of the 18 parameter combinations, IAMR is able to eliminate all malicious workers during the execution of the word count job. In Figure 12, we use three

representative combinations to show how soon each malicious worker is detected and thus removed. We can see that under the first two combinations, IAMR can remove all malicious workers very quickly (within less than 15% of the total job execution time). Under the third combination, the malicious workers are very stealthy (cheat with a probability of 10%) and the replication frequency is low (30%), but IAMR can still remove all six malicious workers before 50% of the job has finished. Such observations suggest that IAMR is effective in detecting malicious workers even if the adversary implements its best strategy.

Figure 12 Malicious Worker Elimination Progress

We also measure the execution time delay introduced by IAMR in the map phase by running the same 800-map task word count job. Since here our focus is the delay in the map phase, we disable the reduce phase integrity check. In addition, we do not introduce malicious workers. The reason is that we believe the customer is willing to pay extra cost to detect malicious workers. However, they are reluctant to spend extra money for IAMR when there are no malicious workers. The experimental results are shown in TABLE 6. The data indicates that the extra running time compared to the baseline increases with the increase of r . When r increases from 0.3 to 1.0, the extra running time increases from 19.75% to 83.26%.

Configuration	Baseline	$v=0.15, T=50$		
		$r=0.3$	$r=0.5$	$r=1.0$

Running time(s)	1728	2069	2323	3167
Extra running time (%)	---	19.75	34.41	83.26

TABLE 6 The Performance of Map Phase IAMR

3.7.1.2 Reduce Phase

To measure the reduce phase, we set the replication probability r as 0.16, the verification probability v as 0.07, and the batch size T as 600. We set both the replication and verification bucket size B as 10 when applying the request bucketing technique.

Our accuracy test shows that such a configuration guarantees 0 job error rate when m is 0.5 and c changes among 0.1, 0.5 and 1.0, which is consistent to our simulation in Figure 6 (c). In this section, we focus on presenting the performance experiment results. We used two applications to measure the reduce phase performance overhead: the word count and the mahout twenty newsgroups classification⁵. For a similar reason as the map phase, we disabled the map phase integrity check and do not introduce malicious nodes.

To measure the performance benefits from the request bucketing technique, we perform two sets of experiments: the one without using the request bucketing technique and the one using such a technique. The experimental results are shown in TABLE 7.

The word count job was the same job described in Section 3.7.1.1, which consisted of 800 map tasks and one reduce task. We compared the running time of IAMR with the baseline. On average, IAMR without request bucketing took 1,398 seconds to finish the job. It produced 88 replication reduce tasks and six verification reduce tasks. Compared with the standard MapReduce, which took 979 seconds to finish the same job (we enabled the combine phase to accelerate the execution), the execution delay was 43%. When we applied the request bucketing to the reduce phase, the number of replication and

⁵ Twenty Newsgroups Classification Example <http://mahout.apache.org/users/classification/twenty-newsgroups.html>

verification tasks dropped to nine and one, respectively; the execution time was reduced to 1,263 seconds and the execution delay was reduced to 29%. We attribute the reduced execution delay to the fact that the number of replication/verification tasks is reduced.

App.	Job No.	Baseline	IAMR without request bucketing				IAMR with request bucketing			
		Exec. time (s)	Exec. time (s)	Delay (%)	Repl. Task Num.	Veri. Task Num.	Exec. time (s)	Delay (%)	Repl. Task Num.	Veri. Task Num.
Word Count	1	979	1398	43%	88	6	1263	29%	9	1
20 News-group Classification	1	517	983	90%	95	7	636	23%	10	1
	2	331	482	45%	108	8	465	40%	11	1
	3	210	221	5%	80	6	260	23%	8	1
	4	85	63	-25%	0	1	57	-32%	0	1
	5	161	143	-11%	0	1	138	-14%	0	1
Ave. Delay (%)*		----	46%				29%			

* Average delay does not count the direct verification job (i.e., job 4 and 5 in the 20 news letters application).

TABLE 7 The Performance of Reduce Phase IAMR

We also ran the Mahout twenty newsgroups classification example on IAMR. This application consisted of five jobs. Each of the first three jobs produced more than 100,000 keys. Hence, IAMR set the batch size T to 600. The last two jobs produced less than 600 keys, so IAMR directly generated a verification reduce task for each job. The total execution time under IAMR without request bucketing was 1,892 seconds. Compared to 1,304 seconds on the baseline, the execution delay was 45%. When applying request bucketing, the execution time was reduced to 1,556 seconds. The execution delay was reduced to 19%. It is interesting to notice that the IAMR execution times of the last two jobs were shorter than the baseline. This was because the master in IAMR was executed on the private cloud, which had more computation power than a micro instance on the public cloud. When evaluating the average slow down incurred by IAMR, we excluded these two special jobs.

Overall, the results show that the request bucketing technique played an important role in

boosting the performance. Based on the execution time of the word count and the first three jobs of 20 newsletters applications, the average execution delay of IAMR without request bucketing was 46%. When request bucketing was applied, the average execution delay was reduced to 29%.

3.7.2 Experiments on the Parameter Searching Algorithm

We mainly performed experiments on the TIGD algorithm. We treated IGD algorithm as a special case of TIGD algorithm, where T_{max} is unbounded, $v_{max} = v_{init}$ and $r_{max} = r_{init}$.

In this set of experiments, we evaluated the most severe situation by assuming m as 1.0. We changed the system restriction setting, including the max batch size (T_{max}), the max replication probability (r_{max}) and the max verification probability (v_{max}), and observed the TIGD suggested parameters and their corresponding optimal job error rates. In the experiments, we set other parameters as fixed values listed in TABLE 8.

The experimental results are shown in TABLE 9. It shows that in each system setting, the TIGD algorithm can find the optimal job error rate under user's system restriction setting.

For instance, when T_{max} was 10,000, r_{max} and v_{max} were 0.3, the TIGD algorithm can find the optimal job error rate as 0.107%, with T as 10,000, r as 0.3 and v as 0.111. When T_{max} was 1,000, r_{max} and v_{max} were 1.0, the algorithm can find the optimal job error rate as 0.122%, with T as 1,000, r as 1.0 and v as 0.231. When T_{max} was 1,000, r_{max} and v_{max} were both 0.3, the optimal value of J that the algorithm can find is 0.372%. Although such a job error rate was high, it was the best value that IAMR can achieve under the user's system restriction setting. In order to further reduce the job error rate, the user had to either increase T_{max} , r_{max} or v_{max} . The fourth row in the table indicates that when T_{max} was small (such as 100), the TIGD algorithm can still achieve a small value of J (0.124%), if r_{max} and

v_{max} were set to a large value.

Item	Explanation	Value
J_0	The acceptable job error rate	1E-3
γ_T	The step size in gradient descent on variable T . (the γ value in the implementation of <code>minimizeUiamrOnT</code> in Figure 10)	100
γ	The step size in gradient descent/restricted gradient descent on variable r , v and c . (The γ value in the implementation of <code>minimizeUiamrOnR</code> , <code>minimizeUiamrOnV</code> and <code>minimizeUatt</code> in Figure 10)	5E-4
ε	The converge threshold in each gradient descent/restricted gradient descent (the ε value in the implementation of each minimization function in Figure 10).	1E-15
δ	The converge threshold in the TIGD algorithm (the variable δ in Figure 10).	1E-5
T_{init}	The initial value of T	10
r_{init}	The initial value of r	0.1
v_{init}	The initial value of v	0.1
c_{init}	The initial value of c	0.1

TABLE 8 Parameter Settings in TIGD Experiments

m	T_{max}	r_{max}	v_{max}	T_{opt}	r_{opt}	v_{opt}	$J_{opt}(\%)$
1.0	10000	0.3	0.3	10000	0.3	0.111	0.107
1.0	1000	1.0	1.0	1000	1.0	0.231	0.122
1.0	1000	0.3	0.3	1000	0.3	0.3	0.372
1.0	100	1.0	1.0	100	1.0	0.752	0.124
1.0	$+\infty^*$	0.1	0.1	27796	0.1	0.1	0.114

*This setting is equivalent to executing IGD algorithm

TABLE 9 Results in TIGD Experiments

The configuration of the last row of the table was equivalent to execution an IGD algorithm. It set the value of r_{max} and v_{max} the same as their initial values ($r_{init}=0.1$ and $v_{init}=0.1$, respectively). T_{max} was set to infinity, which means that T can be increased unboundedly. This experiment was used to compare with the TIGD algorithm. The first row of this table had the same initial system configure and generated the same job error rate. We notice that, in order to achieve small optimal job error rate (0.114%), T has to be set to 27,796 in the IGD algorithm. Compared to the first row of TABLE 9, which achieved the equivalent job error rate, the resulting T that was generated by IGD algorithm is 178% higher than the one generated by TIGD algorithm. The result shows that in order to achieve a small job error rate, loosening r and v slightly (to 0.3) can significantly reduce the value of the batch size T .

To give readers an intuitive understanding of the working procedure of TIGD and IGD algorithm, we recorded the job error rate change during the algorithm executions. Figure 13 a) shows the execution details of the TIGD algorithm when setting system parameters as the first row of TABLE 9. The three disjoint curves reflect three iterations of the interactive gradient descent. In the first iteration, IAMR and the attacker took turn to adjust the value of T and c to minimize their utility function values. We observe that after three rounds of interactions, IAMR achieved the maximum batch size T_{max} (10,000). And the attacker found the optimal value of c that maximizes its utility function when IAMR set T to 10,000. (Note that even though in the figure the second and third round appears to be the same value of T and c , their values are actually different with a higher precision.) The second curve shows the second iteration of interactive gradient descent, where IAMR and the attacker adjusted r and c respectively to minimize their utility function values. Note that adjusting r has a significant effect in helping reducing the job error rate. After the first round of interactive gradient descent, the job error rate was 0.364%. After the second round of interactive gradient descent, the job error rate dropped to 0.119%. In the last iteration of interactive gradient descent, IAMR and the attacker adjusted the value of v and c respectively to minimize their utility function values. After this iteration, the job error rate finally dropped to 0.107%. The final suggested values for T , r , and v were 10,000, 0.3 and 0.111, respectively.

Figure 13 b) shows the execution details of the IGD algorithm when setting parameters as the last row of TABLE 9. We observe that after four rounds of interactive gradient descents, the algorithm can also achieve the job error rate (0.114%). However, it required a higher value of batch size (i.e. $T=27,796$).

Figure 13 TIGD and IGD algorithm execution details

The above experiments show that the TIGD algorithm is an effective method in finding the optimal job error rate under different system restriction settings when the malicious node dominates the environment.

3.8 Related Work

We present the related work about the IAMR in this section. Result Integrity Assurance is an old problem that has been studied for years. Researchers have proposed a wide range of solutions from different perspectives. The Secure co-processor [43] provides a hardware-based solution towards the trusted outsourced computing. The secure hardware chip Trusted Platform Modules (TPMs) installed on a computer can verify the integrity of the worker's software stack ranging from the bios configuration at the booting procedure to the application that executes the outsourced task. By verifying the computing system completely complies the software stack specification, such a technology can guarantee the correctness of task result [44]. However, this solution requires the configuration of the worker's machine to be exactly identical to the TPM's expectation. Such a strict requirement is only suitable for the dedicated worker, which is used for certain computation task.

The zero-knowledge proof [46][47][48] tries to address the problem via mathematical theoretical study. It requires the interaction between the master (the *verifier*) the worker (the *prover*). If the worker can submit a proof to the master and the master verifies that proof, the master believes the assigned task is faithfully executed and thus accepts the result. Gennaro et al. [49] proposed a zero-knowledge protocol that requires no interaction. Because such a class of solutions is only efficient on certain special computing applications, they are not suitable for the generalized computing platform such as MapReduce.

A class of practical solutions towards such a problem falls in the direction of task replication and verification. IAMR is following this direction. Golle et al. [7] proposed to guarantee correctness of the distributed computation result by duplicating computations. Zhao et al. [9] proposed a sampling based idea of inserting indistinguishable Quizzes to the task package, which is going to be executed by the untrusted worker, and verifying the returned result for those Quizzes. Their simulation results show that by combining reputation system, Quiz approach gains higher accuracy and lower overhead than replication-based approach. However, suggested by their simulation, the reputation accumulation is a long-term process so that in order to accumulate reliable reputation, it takes as many as 10^5 tasks, which is not a reasonable number for MapReduce job. Moreover, the attacker could pretend to be benign at the reputation accumulation phase and commit cheat when its reputation achieves a reliable value. Du et al. [6] proposed to sample tasks, and check the sampling task result using Merkle-tree based commitment technique. The Merkle-tree based commitment technique can effectively defeat the malicious worker who returns correct results only on sampled tasks. The analysis in the paper showed that to achieve certain accuracy, the number of samples increases with the decrease of attacker's

cheat probability. For example, in order to restrict the attacker's successful cheating probability to 0.0001, if the malicious worker cheats with probability 0.1, it takes more than 88 samples to be inserted to each worker; if the malicious worker cheats with probability 0.01, it takes 930 samples to be inserted! Our method, which introduces task replication with hold-and-test rule, can reduce the sampled task numbers (i.e., the verification task number) with the same accuracy expectation. In addition, we propose a different accuracy measurement, i.e., the job error rate, to better cater for the MapReduce (Big Data) computation characteristic. Wei et al. [8] explored the similar problem as our thesis, i.e., to construct an integrity assurance framework for MapReduce. They proposed SecureMR to enforce the *commitment protocol* and the *verification protocol*. SecureMR employs non-deterministic task duplication to defeat collusive workers. The design difference from our research is that the number of duplication task for each original task is non-deterministic. Such an approach guarantees 90% of detection rate in defeating periodical collusive attacker with 40% of duplication rate when the malicious worker fraction is below 0.15 and malicious cheat probability is 0.5. (According to (2) in [8]) However, (2) in [8] also shows that when malicious worker fraction is 0.5, malicious cheat probability is 0.1, 40% of duplication rate can achieve only 25% of detection rate. The maximum detection rate SecureMR can achieve under this environment setting is 80%, with a duplication rate more than 500%. In other words, SecureMR works well in an environment where the malicious node does not dominate the environment. However, it cannot guarantee high integrity in an uncontrolled environment like cloud cluster.

Some researchers combine the replication/verification technique with game theoretic approaches to incentivize workers behave honestly [67][68][69]. They usually design a

financial reward/punish mechanism, so that rational workers will behave honestly to maximize their profit. Those solutions assume that workers are lazy but rational entities. The workers' goals are to maximize the profit meanwhile using the least computing resources. Such a class of solution cannot be applied to the "adversary malicious" worker whose goal is to compromise the result integrity without considering the profit.

Fictitious play was introduced in 1951 [70][71] as a mechanism to compute Nash Equilibrium. Overviews on related lines of research are [72][73][74]. Fictitious play has been used in security of computer networks [76][77], intrusion detection systems [75], and cognitive radios [78]. Such approaches usually model the attacker and the defender as fictitious play games, prove the game convergence and provide guidelines for the player to maximize her gain. In our dissertation, we model the behaviors of IAMR and the attacker as a fictitious play. We prove that the fictitious play will converge and the resulting parameters can satisfy user's accuracy requirements and system restrictions.

3.9 Summary

In this chapter we proposed a hybrid cloud-based high result integrity assurance MapReduce framework, called Integrity Assurance MapReduce (IAMR). The IAMR system performs the result checking technique to guarantee high result integrity even if the public cloud contains malicious workers. We also proposed two game theory-based optimization algorithms, called Interactive Gradient Descent algorithm and Tiered Interactive Gradient Descent algorithm, respectively. The two algorithms can search for the optimal parameter setting that can guarantee the lowest job error rate upper bound, with considering user's system restriction setting, if available. We implemented a prototype system and deployed it onto a real hybrid cloud environment. Our theoretical analysis and

experimental results show that IAMR can guarantee high result integrity and incurs a modest performance overhead; the TIGD algorithm is effective in finding optimal parameter settings under different system restriction settings. Our works are published in multiple peer-reviewed journals and conferences [56][57][58][59][60][61].

4 CONFIDENTIALITY ASSURANCE MAPREDUCE

In this chapter, we will present the Confidentiality Assurance MapReduce (CAMR) system. We will start with the motivation of this system. Then we will introduce the system architecture and the system overview, define the system assumption and the attacker model, present the design details and the implementation details, and describe the experimental results. Finally, we will discuss the related work and conclude this chapter.

4.1 System Motivation

Today, many public cloud vendors support MapReduce computation services. For example, Amazon Web Service offers the *Elastic MapReduce (EMR)* service to compute MapReduce jobs on their virtualized Hadoop cluster. Figure 14 shows how a cloud customer uses EMR to compute a MapReduce job. To compute a MapReduce job, a user has to upload the MapReduce job and the input data to the *Amazon Simple Storage Service (S3)* storage system and issue a command to set up a virtualized cluster (EMR Cluster), which consists of multiple *Amazon Elastic Compute Cloud (EC2)* virtual machines. The virtualized cluster will load the MapReduce job from S3 storage and execute the job. Because the entire program is stored and executed on the public cloud, the open architecture of the public cloud gives attackers the opportunity to breach the security defense and compromise the confidentiality of the outsourced MapReduce job. For example, the attacker can perform the following attacks. (The detailed attacker model is described in Section 4.3).

1. Eavesdrop on the communication between the cloud user and the public cloud, shown as attack (1) in Figure 14; or
2. Access customers' storage on the public cloud to obtain the executable file of the

public program, shown as attack (2) in Figure 14; or

3. Eavesdrop on the data transfer between the storage node and the virtual machine, shown as attack (3) in Figure 14; or
4. Compromise the virtual machine to observe the execution of the public program, shown as attack (4) in Figure 14.

Figure 14 The Amazon Elastic MapReduce Use Case and the Attacker Model

Under this situation, an end-to-end protection on the outsourced MapReduce computation on the public cloud is needed.

Researchers have performed many studies on protecting the confidentiality of the data involved in outsourced computation [10][11][12][13][14]. Yet, a satisfying solution to protect the confidentiality of outsourced computation itself is still not available. However, the demand for such a solution is growing rapidly, as more and more innovative algorithmic computations are being deployed to the public cloud. These algorithmic computations involve important applications, such as business analytics [30], geospatial mapping/searching [31], bioinformatics analysis [29], and image processing [28]. When

companies or individuals outsource their programs to the public cloud [33][34][35][36], they wish the algorithms in their programs not to be revealed to others, especially to their competitors.

Control flow, which decides the sequence of instructions (e.g., statements) to be executed, directly reflects the algorithm of a program. The condition of the branch statement, which determines the runtime control flow, is a critical component of the control flow that needs to be protected foremost. As an illustration, imagine a company *A* offers an image processing service like [28] that implements several image manipulation algorithms, such as edge detection algorithm. A malicious company *B* wants to offer similar service with the same level of service quality. Company *B* thus hires a hacker to find out the edge detection algorithm that *A* is using. If company *A* outsources her service to the public cloud and the hacker is able to compromise the public cloud, the hacker can perform reverse engineering attacks on *A*'s program to find out details of *A*'s algorithm. It is interesting to note that a wide class of edge detection algorithms exist [62] and many of them have internal parameters that are important to the effectiveness of the algorithms. For example, the Canny edge detector [63] is a multi-stage algorithm that can detect a wide range of edges in images. Figure 15 shows two stages of that algorithm: generating Gaussian filter matrix (the function *generateGaussianKernel*) and determining edges via double threshold and hysteresis (the function *performHysteresis*).

```

// Two stages of the Canny edge detection algorithm implementation [32]:
// generating the Gaussian filter matrix (the function generateGaussianKernel) and
// determining edges via double threshold and hysteresis (the function performHysteresis).

// Constant variables
private float GAUSSIAN_CUT_OFF = 0.005f;
private int GAUSSIAN_KERNEL_WIDTH = 16;
private int LOW_THRESHOLD = 250;
private int HIGH_THRESHOLD = 750;
//other variables

public void generate_gaussian_kernel(){
    .....
    for (kwidth = 0; kwidth < GAUSSIAN_KERNEL_WIDTH ; kwidth++) {
        float g1 = gaussian(kwidth, kernelRadius);
        if (g1 <= GAUSSIAN_CUT_OFF && kwidth >= 2) break;
        float g2 = gaussian(kwidth - 0.5f, kernelRadius);
        float g3 = gaussian(kwidth + 0.5f, kernelRadius);
        kernel[kwidth] = (g1+g2+g3)/3f/(2f*(float)Math.PI*kernelRadius*kernelRadius);
        diffKernel[kwidth] = g3 - g2;
    }
}

private void performHysteresis(){
    Arrays.fill(data, 0);
    int offset = 0;
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            if (data[offset] == 0
                && magnitude[offset] >= HIGH_THRESHOLD) {
                follow(x, y, offset);
            }
            offset++;
        }
    }
}

private void follow(int x1, int y1, int i1) {
    .....
    data[i1] = magnitude[i1];
    for (int x = x0; x <= x2; x++) {
        for (int y = y0; y <= y2; y++) {
            int i2 = x + y * width;
            if ((y != y1 || x != x1)&& data[i2] == 0
                && magnitude[i2] >= LOW_THRESHOLD) {
                follow(x, y, i2);
                return;
            }
        }
    }
}
}

```

Figure 15 Implementation of Canny Edge Detector Program

In the Canny edge detector algorithm, the accuracy and the speed (i.e., the computation time) of detecting image edges are influenced by four parameters involved in the above two stages: the size and the cut-off value of the Gaussian filter matrix (represented as *GAUSSIAN_KERNEL_WIDTH* and *GAUSSIAN_CUT_OFF* in *generateGaussianKernel*, respectively) and the two threshold values (represented as *HIGH_THRESHOLD* in *performanceHysteresis* and *LOW_THRESHOLD* in *follow*, respectively). All these four

parameters are critical to the effectiveness and the speed of the edge detection service, so they should be kept confidential if company *A* uses the Canny edge detector algorithm. However, those parameters appear as plain text in branch statements of the program, so the hacker who has obtained the program can easily find their values by using reverse engineering attacks. In this scenario, hiding the branch statement conditions that involve those parameters will make it difficult for the attacker to extract values of the algorithmic parameters. In addition, hiding other branch statement conditions will further raise the bar for the attacker to understand the algorithm's details.

Note that in real life, company *A* may very likely invent its proprietary edge detection algorithm, and this is exactly the reason why company *A* wants to protect the confidentiality of its algorithm from company *B*. Unfortunately, we do not have access to any proprietary edge detection algorithm, so we have to use an existing edge detection algorithm like the Canny edge detector as an illustration. However, we believe that certain properties (e.g., the cut-off value of the Gaussian filter matrix) may well exist in company *A*'s algorithm. In other words, the implementation of company *A*'s algorithm may also contain such confidential information in its branch statement conditions, which makes a solution like ours useful.

Researchers have proposed quite a few solutions to protect the branch statement conditions. However, most of them [55][16][51][52][54][64] choose to submit the entire program to the adversarial environment in the obfuscated form. This cannot defeat a determined attacker as the entire program is exposed in the adversarial environment. The attacker can perform static and/or dynamic analysis at will. Other solutions [53][15] propose to break the control flow into two or multiple processes. The runtime control flow

is determined by the collaboration of those multiple processes. This offers better protection of the control flow, but there are well-documented performance shortcomings in those existing attempts. For example, [15] proposes a binary-code-level obfuscation based on a two-process model, where a *program process* executes the program, and a *monitor process* determines the control flow. The experimental results show that, when the shuffle rate is 5% (i.e., 5% of the control flow is hidden in the monitor process), the execution time of the obfuscated program is 1.08 to 3.37 times greater than that of the original program; when the shuffle rate is 20%, the execution time of the obfuscated program is 2.77 to 8.87 times greater than that of the original program. Such an overhead measurement is based on the fact that the two processes are executed on the same host, which enables an attacker to perform static and dynamic analysis on the two processes together to reverse engineer the control flow. We can imagine that, if we were to execute the two processes on separate hosts, the network communication involved would further aggravate the performance overhead.

4.2 Architecture and System Overview

In this chapter, we propose Confidentiality Assurance MapReduce (CAMR) framework to protect the control flow confidentiality of outsourced MapReduce jobs. CAMR use our proposed control flow obfuscation method called *runtime control flow obfuscation* (RCFO). RCFO resembles the two-process model in [15]: it breaks the outsourced program into two programs: the *public program* and the *private program*. The public program performs the computation, and the private program maintains the control flow. At runtime, the public program performs the computation and queries the private program to determine the execution path. Different from existing solutions such as [15], we transform the control

flow condition into a different format and extract the predicate secret, a constant number, from the branch statement condition and hide it in the private program. Based on our unique transformation, we propose the *continuous cache* on the public program to greatly reduce the number of control flow queries. Our experimental results show that our solution incurs 14.9% to 33.2% of performance overhead on average with different obfuscation degree settings.

Specifically, RCFO first analyzes the conditions of each branch statement in the original program and performs a mathematical transformation on it. (In this paper, the term *branch statement* refers only to *if* statements, as our method works directly at the lower level language representation, such as Java byte code, where *while* and *for* statements are represented by more primitive instructions such as *if* and *goto*.) This transforms each original predicate into a normalized inequality. The *left hand side (LHS)* of the inequality is a variable expression, called the *predicate expression* and the *right hand side (RHS)* is a random numeric value, called the *predicate secret*. Next, RCFO keeps the predicate expression in the public program while storing the inequality operator and the predicate secret to the private program. After mathematical transformation, the original program is transformed into a public program by replacing each predicate with an invocation of a *control flow query function (CFQ function)*. A CFQ function invocation takes the predicate expression as one of its parameters. The implementation of the CFQ function is in the private program. At runtime, the CFQ function returns a Boolean value based on the predicate expression, the inequality operator, and the predicate secret, to guide the program control flow.

<pre>//Constant variable declarations are removed. //Instead, their values are hidden in the private program after RCFO transformation.</pre>

```

public void generate_gaussian_kernel(){
    .....
    for(kwidth=0;CFQ(59-kwidth);kwidth++){//PredicateSecret:59+GAUSSIAN_KERNEL_WIDTH=75
    float g1 = gaussian(kwidth, kernelRadius);
    if (CFQ(0.01-g1) && kwidth>=2) break; //predicate secret: 0.01+GAUSSIAN_CUT_OFF=0.015
    float g2 = gaussian(kwidth - 0.5f, kernelRadius);
    float g3 = gaussian(kwidth + 0.5f, kernelRadius);
    kernel[kwidth] = (g1+g2+g3)/3f/(2f*(float)Math.PI * kernelRadius * kernelRadius);
    diffKernel[kwidth] = g3 - g2;
    }
}

private void performHysteresis(){
    Arrays.fill(data, 0);
    int offset = 0;
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            if (data[offset] == 0
                && CFQ(magnitude[offset]-54)) { //predicate secret: HIGH_THRESHOLD-54=696
                follow(x, y, offset);
            }
            offset++;
        }
    }
}

private void follow(int x1, int y1, int i1) {
    .....
    data[i1] = magnitude[i1];
    for (int x = x0; x <= x2; x++) {
        for (int y = y0; y <= y2; y++) {
            int i2 = x + y * width;
            if ((y != y1 || x != x1)&& data[i2] == 0
                && CFQ(magnitude[i2]+31)) { //predicate secret: LOW_THRESHOLD+31=281
                follow(x, y, i2);
                return;
            }
        }
    }
}
}

```

Figure 16 Condition Transformation of the Canny Edge Detector Implementation in Figure 15

Figure 16 shows the application of RCFO on the Canny edge detector example in Figure 15. As Figure 16 shows, the definition of the four critical parameter values (*GAUSSIAN_KERNEL_WIDTH*, *GAUSSIAN_CUT_OFF*, *LOW_THRESHOLD* and *HIGH_THRESHOLD*) are removed. Instead, RCFO directly transforms predicates that contain the four parameter values into CFQ function invocations and hides those values in the private program. For example, the original condition predicate *magnitude[offset]>=HIGH_THRESHOLD* is transformed into *CFQ(magnitude[offset]-54, L3)*. The corresponding predicate secret, 696 (*HIGH_THRESHOLD-54*), is hidden in the private program. Here, *L3* is a function call site identifier, which uniquely identifies a CFQ function invocation. Such a transformation hides critical parameter values into the private

program. This effectively defeats static analysis-based reverse engineering attacks and raises the bar for dynamic analysis-based reverse engineering attacks (see Section 4.5). For simplicity of exposition, here we only transform predicates that contain the algorithmic parameter values. In practice, RCFO can transform each predicate to further raise the bar for the attacker to understand the algorithm’s details.

In order to further protect the control flow graph, we propose the *fake branch statements insertion* method (see Section 4.4.2) to complicate the control flow graph. Specifically, we generate *indistinguishable fake branch statements* and insert them into the public program. Because each transformed condition, regardless of whether it is genuine or fake, has a piece of information (i.e., the predicate secret) hidden in the private cloud, it will be difficult for the attacker to distinguish between genuine branch statements and fake branch statements. As a result, it is difficult for the attacker to recognize the original algorithm from the transformed control flow graph.

Transforming the condition into a normalized inequality enables the public program to reuse existing control flow query results (i.e., return values of the CFQ function invocations). Based on this characteristic, we propose the *continuous cache* (See Section 4.4.3) to dramatically reduce the number of CFQ function invocations and thus reduce the performance overhead. Because the number of cross program queries is reduced, we can deploy the private program to a trusted private cloud without incurring significant performance overhead and therefore completely separate the private program from the attacker. To prevent the attacker from eavesdropping on the CFQ function invocation outside of the public program execution node, we propose the *CFQ function encryption* scheme (see Section 4.6.5). To prevent the attacker from passively observing the CFQ

function invocation and deriving the predicate secret inside of the public program execution node, we propose the *loop transformation* technique (see Section 4.6.6). We prove that, by selecting proper parameters, the loop transformation technique can prevent the attacker from distinguishing constant-iteration-number-loops from dynamic loops.

Compared to existing solutions such as [55][16][51][52][54][64], RCFO segregates critical control flow information on the trusted private cloud and raises the bar for static and dynamic reverse engineering attacks. Compared to existing distributed solutions [15][53], our proposed continuous cache reduces the cross-cloud communication incurred by control flow queries and thus reduces the performance overhead.

Based on the RCFO technique, we developed the system Confidentiality Assurance MapReduce (CAMR), which can be applied to existing cloud based MapReduce services, such as Amazon Elastic MapReduce (EMR). With CAMR, a user can apply RCFO on MapReduce jobs and execute obfuscated jobs on the public cloud. Our experimental results in Figure 26 show that the average performance overhead ranges from 14.9% to 33.2% when the obfuscation degree increases from 0 to 1.0.

It is important to note that the application of RCFO is not restricted to cloud-based MapReduce services. It is a generalized control flow obfuscation technique that can be applied to a wide range of remote execution architectures and different programming paradigms. We made further discussion in Section 5.2.

It is necessary to note that RCFO only targets the confidentiality of the control flow. Namely, RCFO does not protect the confidentiality of the data processed in the remote program execution. For example, in the statement sequence *temp=read(confidential); ...;*

print(temp);, confidential information is leaked through the data flow. Such a problem can be addressed by data flow protection techniques, which are of the scope of this paper. We refer readers to existing works such as [10][11][12][13][14].

Compared to IAMR, where the master has to be deployed on the private cloud, CAMR does not require the master to be deployed on the private cloud. In CAMR, RCFO is performed offline. After the obfuscation transformation, the user only needs to deploy the generated private program to the private cloud.

An ordinary execution scenario of CAMR is presented in Figure 17. For each original program that the cloud user wants to outsource to the cloud, the user first performs RCFO offline (step 1). The outcome is two programs: a public program and a private program. The public program and the private program are deployed to the public and private cloud, respectively (step 2). After that, the public program starts to execute (step 3). Whenever a CFQ function is invoked (step 4), the public program will first check the CFQ cache (step 5). If there is a cache hit, it will return the value from the CFQ cache. Otherwise, it will invoke the CFQ function (step 6) on the private cloud. The return value of the CFQ function invocation determines the control flow of the public program. Steps 3 through 6 may be executed repeatedly until the program finishes.

Figure 17 Execution Scenario of Confidentiality Assurance MapReduce

4.3 System Assumptions and Attacker Model

Based on the system architecture, we make the following system assumptions and declare the attacker model. We assume the program obfuscation is performed in a trusted environment and is completely opaque to the attacker. The generated private program will be sent to the private cloud through a secure channel, such that the private program is completely inaccessible to the attacker.

We assume the private cloud is trusted and well protected. Therefore, the attacker can neither access the private program statically nor observe the private program execution dynamically.

However, we assume that the public cloud is not trusted. We assume the worst-case situation that the attacker can completely compromise the public cloud. The attacker can be a malicious outsider who breaches the defense and breaks into the public cloud. The attacker can also be a malicious cloud operator, who abuses her administrative privilege to observe or tamper with the cloud user's program. Specifically, the attacker can perform the following attacks on the cloud-based MapReduce architecture depicted in Figure 14.

1. The attacker can eavesdrop on the communication between the cloud user and the

public cloud, shown as attack (1) in Figure 14.

2. The attacker can access customers' storage on the public cloud to obtain the executable file of the public program, shown as attack (2) in Figure 14.
3. The attacker can eavesdrop on the data transfer between the storage node and the virtual machine, shown as attack (3) in Figure 14.
4. The attacker can compromise the virtual machine to observe the execution of the public program, shown as attack (4) in Figure 14.

The goal of the attack is to understand the outsourced program's algorithm. By obtaining the public program and/or observing the program execution on the virtual machine, the attacker can perform the static analysis and the dynamic tracing to restore the outsourced program and, thus, to understand the program's algorithm.

In this paper, we focus on protecting the control flow confidentiality of the outsourced program. Hence, we assume that its control flow integrity is assured. We can employ existing techniques, such as [26][79] to protect the control flow integrity. The ideas of [26][79] are to have the remote program report its execution path to the trusted program. If the reported execution path deviates from the control flow graph, the trusted program can determine that the control flow integrity of the remote program execution is compromised. The reader can find further details in [26][79]. In RCFO, the public program invokes the CFQ function with the predicate expression value and the call site number. The private program can check the sequence of call site numbers against the control flow to determine if the public program has violated the control flow integrity. However, the attacker can construct a sequence of CFQ function calls that is consistent to the control flow to pass the check. To reduce such false negatives, the private program can check if the function

invocation's predicate expression value is within the expected range (i.e., legitimate range of the predicate expression). If the attacker uses different values to test a CFQ function, if the value is out of the range of the predicate expression, the private program can determine that the public program is compromised.

4.4 Design Details

The critical component of CAMR is the Runtime Control Flow Obfuscation technique. In this section, we will focus on the design detail of RCFO. We will first describe the transformation of the original condition, the construction of the fake branch statements, and the design of the continuous cache. After that, we will introduce the CFQ function encryption and loop transformation technique. Finally, we will discuss the security properties of the system design.

4.4.1 Transforming the Original Condition

We perform the original condition transformation with the following steps. We first extract the condition from each branch statement and transform it into a normalized inequality. Then we perform a mathematical transformation on the normalized inequality. Finally, we replace the original predicate with a CFQ function call and construct the CFQ function implementation in the private program. We discuss details for each step.

4.4.1.1 Condition Normalization

We extract the condition from each original branch statement and transform it into a *normal form*, which is one or multiple inequalities connected with the Boolean operator *AND* or *OR*. Each inequality has one of the following two relations: $>$ (greater than) or $\geq$ (greater or equal). The normalization rule is defined in Figure 18.

$$\boxed{\text{trans}(\alpha \text{ AND } \beta) \rightarrow \text{trans}(\alpha) \text{ AND } \text{trans}(\beta) \quad (1)}$$

$\text{trans}(\alpha \text{ OR } \beta) \rightarrow \text{trans}(\alpha) \text{ OR } \text{trans}(\beta)$	(2)
$\text{trans}(\delta) \rightarrow \text{trans}(\delta == \text{true})$	(3)
$\text{trans}(!\delta) \rightarrow \text{trans}(\delta == \text{false})$	(4)
$\text{trans}(\eta == \theta) \rightarrow \text{trans}((\eta >= \theta) \text{ AND } (\eta <= \theta))$	(5)
$\text{trans}(\eta != \theta) \rightarrow \text{trans}((\eta > \theta) \text{ OR } (\eta < \theta))$	(6)
$\text{trans}(\eta < \theta) \rightarrow (\theta > \eta)$	(7)
$\text{trans}(\eta <= \theta) \rightarrow (\theta >= \eta)$	(8)
$\text{trans}(\theta > \eta) \rightarrow (\theta > \eta)$	(9)
$\text{trans}(\theta >= \eta) \rightarrow (\theta >= \eta)$	(10)

Figure 18 Condition Normalization Rule in Original Branch Statement Transformation

Here we denote α and β as arbitrary expressions and δ , η and θ as “atomic” expressions that cannot be further broken down into expressions connected with Boolean operators (i.e., *AND* and *OR*) or relational operators (i.e., $==$, $!=$, $>$, $>=$, $<$, and $<=$). We define normalization rules as functions named *trans*. For each condition, the rules are recursively applied until no rule can be applied. The resulting expression complies with a normal form.

To perform the mathematical transformation, we replace the “*true*” and “*false*” in rules (3) and (4) into numerical values according to the specification of the programming language. For example, in the Java language, we transform “*true*” and “*false*” into “1” and “0”, respectively. Suppose an original condition is $((y < x) \text{ AND } \text{isAllowed}(x))$, where y and x are two variables and *isAllowed()* is a function returning a Boolean value. After transformation, it can be normalized into a condition $((x > y) \text{ AND } (\text{isAllowed}(x) >= 1) \text{ AND } (\text{isAllowed}(x) <= 1))$, via rules (1), (7), (3), (5), (8), (9) and (10).

Such a normalized inequality format lays the foundation for the mathematical transformation (see Section 4.4.1.2) and the continuous cache (see Section 4.4.3). Notice that the attacker can still restore the condition by applying simple *abstract syntax tree (AST)* analysis with the current format. However, the mathematical transformation introduced in Section 4.4.1.2 and the CFQ function transformation introduced in Section 4.4.1.3 make

such an attack difficult.

4.4.1.2 *Mathematical Transformation*

We next perform a mathematical transformation on each normalized inequality to generate a CFQ predicate. The transformation reorganizes the normalized inequality such that the RHS of this inequality only contains a randomly generated number and the other terms in the inequality are moved to the LHS. The reorganized inequality is mathematically equivalent to the original one. At this point, we have the inequality that the LHS is an expression containing program variables and the RHS is a constant value (i.e., the random number). We call the LHS expression the *predicate expression* and the RHS value the *predicate secret*. Due to normalization, the inequality can only have two possible relational operators: “>” or “>=”. The goal of such a mathematical transformation is to introduce a random number as a predicate secret that can be hidden in the private program.

We have to note that such a strategy also works for predicates that only contain variables or function calls. For example, suppose an original predicate is $x > y$; by generating the predicate secret as 5, we can transform the inequality into $x - y + 5 > 5$. In this case, although the predicate secret value, 5, is displayed in the CFQ predicate expression, the attacker cannot conclude that the exposed number is the predicate secret because she does not know the original predicate.

4.4.1.3 *CFQ Function Construction*

With the transformed inequality, we construct the CFQ function invocation and the CFQ function implementation as follows. We replace each inequality with a CFQ function invocation. It takes two parameters: the predicate expression and the call site identifier. The call site identifier is a number uniquely identifying the current CFQ function

invocation and is used to look up the predicate secret in the private program.

We replace each transformed inequality with the generated CFQ function invocation in the public program, and send the relational operator and the predicate secret to the private program to construct the CFQ function implementation. When the CFQ function is invoked, it looks up the relational operator and the predicate secret via the passed function call site identifier. The function returns *true* if the passed predicate expression value and the predicate secret satisfy the relational operator, and vice versa.

Because each transformed CFQ predicate is mathematically equivalent to the original predicate, the behavior of the transformed program is not changed. We show a transformation example in Figure 19.

Figure 19 Example of Transforming Normalized Condition

Suppose a normalized inequality is $x + 5 > y$, the transformation selects a random number as the predicate secret (i.e., 23) and generates the inequality $x - y + 28 > 23$. The normalized

inequality is replaced with a CFQ function invocation in the public program. The predicate secret 23 and the relational operator “>” are sent to the private program, indexed by the call site identifier LI . The invocation of the function $CFQ(pred_exp_val, call_site_id)$ passes the value of the predicate expression $x-y+28$ and the function call site identifier LI as parameters. When the CFQ function is invoked, the function looks up the predicate secret and the relational operator corresponding to $call_site_id$ and returns the comparison result.

4.4.2 Inserting Fake Branch Statements

To further obfuscate the original program, we also insert fake branch statements in the public program.

It takes three steps to complete the fake branch statement insertion. We first perform data flow analysis on the original program to evaluate the variable value range for each statement. After that, we construct fake branch statements based on the evaluated variable value range and insert the fake branch statements into the original program. Because the fake branch predicate is derived from the original branch predicate, we propose *expression relaxation* and *expression aggregation* to hide such a derivation relationship to confuse the attacker. Finally, we apply the condition transformation method discussed in Section 4.4.1 to transform the fake branch condition into the CFQ condition. In this section, we only discuss the first two steps because the last step was covered previously in Section 4.4.1.

4.4.2.1 Variable Range Evaluation

We perform an intra-procedural data flow analysis on each original program function to evaluate the variable ranges. We are interested in the question “*What are the ranges of variables in the current procedure when a statement is about to be executed?*” The evaluated variables in each statement are not restricted to the ones that appear in the current

statement; they can be any variable defined in the current procedure, as long as such the variable is within the scope of the current statement. We do not require the resulting variable ranges to be accurate. Yet we require the resulting variable range not violate its runtime value. In other words, the evaluated range can be gross. For example, suppose a variable x is evaluated as $x > 5$ in our evaluation. If its actual exact range is $x > 100$, we still think such evaluation is acceptable. This is because using $x > 5$ or $x > 100$ as the condition will return the same Boolean value at runtime. Therefore, it will not change the runtime control flow. In addition, $x > 5$ will not reveal the exact variable range, which is implied from the original control flow predicate. Therefore, a gross evaluation of variables helps the obfuscation.

The variable evaluation for each statement is represented in the form of a *range set*. Each element in the set is a *relation triple*, with the format $\langle \text{left expression}, \text{relational operator}, \text{right expression} \rangle$. The relation triple supports six relational operators: $>$, $>=$, $<$, $<=$, $==$, and $!=$. The left and right expressions can be any variable expression, including variables, function calls, and constants.

The variable evaluation is derived from the original branch conditions. We first perform normalization on the original branch condition to convert it into relation triples. The normalization is similar to the *trans* function defined in Figure 18. However because the relation triple can contain $<$, $<=$, $==$ and $!=$ (compared to the condition normalization, which only contains “ $>$ ” and “ $>=$ ”), this normalization only needs to apply rules (1) through (4) from Figure 18. Based on the relation triples, we perform a forward data flow analysis to propagate the relationship information to the succeeding statements and accumulate the range set for each statement. The analysis is complete when it reaches a

fixed point, i.e., the range set for each statement does not change. The analysis algorithm is shown in Figure 20.

```

//Init range set for each statement in procedure P
func range_set_init(proc P){
    for (stmt s in proc P){
        R_in[s] = {};
        R_out[s][ ] = {};
    }
}
//Range set accumulate for each statement s
func range_set_accumulate(stmt s, list<range_set> R_out_pred){
    R_in = intersection(R_out_pred);
    R_in[s] = union(R_in[s], R_in);
    R_out = prune(R_in[s], s);
    if(s is a branch statement){
        for(stmt o in s.next_stmt){
            R_out[s][o] = intersection(R_out, extract_range_set(s,o));
        }
    } else{
        o = s.next_stmt;
        R_out[s][o] = R_out;
    }
}
//Forward traverse to update the range set
func get_variable_range_set(proc P){
    range_set_init(P);
    it = P.get_forward_traverse_iterator();
    while(it.has_next()){
        s = it.next();
        R_in_prev = R_in[s];
        range_set_accumulate(s, R_out[s.pred_list]);
        if(R_in[s] == R_in_prev)
            break; //fix point reached
    }
}

```

Figure 20 Variable Range Evaluation Algorithm

We use two arrays to record the incoming and outgoing range set for each statement. $R_in[s]$ records the range set before s is executed, and $R_out[s][]$ records the range set after s is executed. Because a branch statement can have two or more successors and different successors can have different variable ranges, we use a two-dimensional array to record the outgoing range set. The entire analysis is controlled in the function *get_variable_range_set*. It accepts a procedure P as input and accumulates the evaluated variable ranges for each statement in P . It first calls the *range_set_init* function to initialize the range set of each statement as empty. After that, it traverses the program statements through its control flow. When a statement is visited, the function *range_set_accumulate* is

called to update its incoming and outgoing range set. When the analysis reaches a fixed point, it terminates.

The function *range_set_accumulate* takes the statement s and the outgoing range sets of its predecessors R_out_pred as arguments and updates the incoming and outgoing range sets of the statement s . It first merges the evaluations of its predecessors' outgoing range sets by calculating their intersection, which is stored in set R_in . After that, it updates the incoming range set of statement s by performing a union on the old incoming $R_in[s]$ and R_in . To compute the outgoing range set, it first prunes $R_in[s]$ if statement s may modify the value of the variables, which are evaluated in the range set. The pruning is conservative. It includes:

- If s is an assignment statement and the LHS of s is a variable in $R_in[s]$, the evaluation about the variable in $R_in[s]$ is pruned.
- If s contains a function call and a parameter of the function call is a variable in $R_in[s]$, the evaluation about the variable in $R_in[s]$ is pruned. This is because, inside the function, the value of the variable can be changed.

If s is not a branch statement, the pruning result will be the outgoing range set of s . Otherwise, the outgoing range set has to be updated by an intersection of the pruning result and the branch condition range evaluation, which is computed in the function *extract_range_set*. The *extract_range_set* function returns the relation triples based on the passed branch statement condition and the outgoing branch. For example, if a branch statement is *if*($x > 5$) $x++$; *else* $x--$;, and the relation triple is $x > 5$. The *extract_range_set* function returns $x > 5$ for the *true* branch and $x \leq 5$ for the *false* branch.

When the analysis completes, the range sets in the array of $R_in[]$ are the variable

evaluation results for each statement.

4.4.2.2 Fake Branch Statement Insertion

We can insert a fake branch statement before each original statement if its variable evaluation result is not empty. We call such original statements *insertion candidates*. However, we need to generate a fake condition for each insertion candidate first. To generate a fake condition for an insertion candidate, we first normalize each resulting relation triple in the variable evaluation into an expression with one of the following formats: $LHS > RHS$, $LHS \geq RHS$, $LHS == RHS$ or $LHS \neq RHS$ (here we only need to apply rules (7) and (8) from Figure 18 to normalize the $LHS < RHS$ and $LHS \leq RHS$ inequalities). After that, we perform the *expression aggregation* and the *expression relaxation* on those expressions to generate a series of indistinguishable fake predicates. Finally, we randomly select several fake predicates and connect them with the Boolean operator *AND* or *OR* to form a fake condition. To generate more genuine-seeming fake branch statements, we construct the fake branch statement using either the generated fake condition or the negation of the fake condition. If we use the generated fake condition, the *true* branch of the fake branch statement goes to the original next statement, and the *false* branch goes to a randomly chosen statement in the current procedure. If we use the negation of the fake condition, the *false* branch goes to the original next statement, and the *true* branch goes to a randomly chosen statement.

The expression aggregation technique can be performed on any two expressions with the format $LHS > RHS$, $LHS \geq RHS$ or $LHS == RHS$. With two such expressions, we perform addition on the LHS and RHS and adjust the relational operator according to the inequality addition properties. If one expression has the format $LHS == RHS$, the generated expression

will have the same relational operator as the other expression. If no expression is in the format $LHS == RHS$ and one expression is in the format $LHS > RHS$, the generated relational operator will be $>$. If both expressions are in the format $LHS >= RHS$, the generated expression will have the relational operator $>=$.

The expression relaxation technique can be performed on an expression with the format $LHS > RHS$ or $LHS >= RHS$ to further “relax” the expression relationship. For an expression with one of above formats, we “relax” the inequality by adding a random number to the LHS. We call the constant number the *relax value*. For the expression with the format $LHS > RHS$, we can also “relax” it to $LHS >= RHS$.

For each insertion candidate, we can apply expression aggregation and expression relaxation to any subset of its evaluation range set. The two techniques can also be applied repeatedly in an arbitrary order. As a result, we can generate a series of different fake predicates for each insertion candidate. For each insertion candidate, the obfuscator can insert more than one fake branch statements before it to further confuse the attacker.

To make a tradeoff between the obfuscation effect and the performance overhead, we insert a fake branch statement before each insertion candidate with a certain probability, which we call the *obfuscation degree*, denoted as d . The value of the obfuscation degree ranges from 0 to 1.0. A higher obfuscation degree means that a fake branch statement is inserted before an insertion candidate with a higher probability. For example, for the TeraSort application used in the experiment in Section 4.7, the number of original statements is 351. Out of the 351 original statements, 23 are branch statements. After variable range analysis, we can find 76 insertion candidates. When d is set to 0, no fake branch statements are inserted in the program. When d is set to 0.5, 38 fake branch

statements are inserted. When d is set to 1.0, 76 fake branch statements are inserted. Setting a higher obfuscation degree makes the program control flow graph more complicated, but it also incurs more CFQ function invocations on the fake branch statements. Fortunately, the continuous cache (introduced in Section 4.4.3) helps us effectively reduce the number of remote CFQ function invocations. Our experimental results in Figure 26 show that, when the obfuscation degree increases from 0 to 1.0, the performance overheads of the four applications do not increase significantly. Among the four applications, the maximum performance overhead increase occurs in the application TeraGen, whose performance overhead increases from 15.8% to 62.6%. The average performance overhead increase in the four applications is 18.3%.

After expression relaxation and aggregation, the variable range of the generated fake predicate is a superset of the actual runtime range for the variable expression. Therefore, the inserted fake condition will not alter the runtime control flow.

An example of fake branch statement insertion is shown in Figure 21. Suppose the range set for statement $L8$ is evaluated to be 1) $x > 6$ and 2) $45 > y$. By applying expression aggregation to 1) and 2) and applying expression relaxation with a relax value of 2, we obtain $x + 45 + 2 > y + 6$. By choosing the predicate secret as 4, we obtain $x - y + 45 > 4$. We transform the expression into the CFQ condition $CFQ(x - y + 45, L7)$, where $L7$ is the function call site identifier, and send $L7$, “>” and the predicate secret 4 to the private program. The obfuscator randomly chooses a statement $L2$ as the *false* branch statement target and constructs a false branch statement (e.g., *goto L2*). At runtime, the generated CFQ condition is always evaluated as *true*. Thus, the false branch will never be triggered.

<pre>//Original program L2: y=y-2;</pre>

	
	//Range evaluation: 1) x>6; 2) y<45
L8:	z = x+y;
	
	//Public program
L2:	y=y-2;
	
L7:	if (CFQ(x-y+45, L7))
L8:	z=x+y;
	else goto L2;

Figure 21 Example of Inserting Fake Condition

4.4.3 Speed up Control Flow Queries with the Continuous Cache

Because the predicate secret is stored in the private cloud, a straightforward implementation of the CFQ function that queries control flow cross-cloud incurs significant performance overhead. Hence, we design a *continuous cache* to mitigate such overhead. In our design, the public and the private programs both implement the CFQ function, resulting in the *public CFQ function* and the *private CFQ function*, respectively. The private CFQ function implements the actual control flow query, while the public CFQ function maintains a continuous cache to store the ranges of parameters that make the private CFQ function returns *true/false*. When the public program invokes the CFQ function, the public CFQ function is firstly invoked. It queries the continuous cache. If a cache hit occurs, the public CFQ function returns the cached value. Otherwise, it invokes the private CFQ function and updates the continuous cache.

Because the CFQ function only accepts two relational operators, “>” and “>=”, we design the continuous cache as follows. For each CFQ function call site, it records the ranges of the predicate expression values that make the CFQ function return *true/false*. Based on the relationship of the new predicate expression value and the cached predicate expression value range, the cache determines whether a private CFQ function call is needed. The design of the continuous cache is shown in Figure 22.

Figure 22 The Design of Continuous Cache

For each CFQ function call site, the continuous cache stores two numbers in the public program: the current smallest predicate expression value that makes the CFQ function return *true* (which we call the *true bound*), and the current greatest predicate expression value that makes the CFQ function return *false* (which we call the *false bound*). When a new predicate expression value is fed into a continuous cache to be looked up, it will be compared with the true bound and the false bound. If the new value is greater than the true bound, the continuous cache returns *true*. If the new value is smaller than the false bound, the cache returns *false*. If the new value is between the true bound and the false bound, the private CFQ function is invoked. If the private CFQ function returns *true*, the new predicate expression value becomes the smallest known value that makes the CFQ function return *true*. Therefore the true bound is updated with the new value. Otherwise, the false bound is updated with the new value.

Each CFQ function invocation only needs to keep two values (i.e., the *true bound* and the *false bound*) in the continuous cache. Suppose a transformed public program contains N branch statements; because each condition in the branch statement is transformed into a CFQ function invocation, the continuous cache only needs to maintain $N \times 2$ values in total. Therefore the space complexity of the continuous cache is $O(N)$.

We define the *cache-hit rate* of the continuous cache as the ratio of the private CFQ

function invocations to the public CFQ function invocations. When the CFQ expression value of a CFQ function falls between its true bound and false bound, a private CFQ function will be invoked. The invocation of the private CFQ function will update the true bound or the false bound, which reduces the range between the true bound and the false bound. If the sequence of the CFQ expression values is randomly distributed, as time goes by, the range between the true bound and the false bound will be reduced, and the probability of the next expression value falling into this range will also be reduced. Hence, the cache-hit rate is increasing. However, the sequence of the CFQ expression values is usually not randomly distributed. It is determined by the control flow of the outsourced program. Given the variety of the outsourced program control flow, making a quantitative analysis on the cache-hit rate is difficult. Therefore, we only measure the cache-hit rate through experiments. Our experiments in TABLE 11 counts the number of private and public CFQ function invocations and computes the cache-hit rate. It shows that the continuous cache dramatically reduces the number of private CFQ function invocation. According to TABLE 11, the lowest cache-hit rate the continuous cache experienced was 90.47%. In the four MapReduce applications, most tasks can achieve a cache-hit rate of over 99%.

4.4.4 CFQ Function Encryption

To prevent the attacker from wiretapping the private CFQ function invocation, we protect the parameters and the return values of CFQ function invocations with symmetric encryption. The symmetric encryption key is generated in the public program, encrypted by a public key encryption scheme and is sent to the private program. The protocol is

shown in Figure 23.

Figure 23 CFQ Function Encryption

In the offline obfuscation, RCFO generates an asymmetric public key pair K_{pub} and K_{priv} and sends them to the public and private programs, respectively (step 0). When the public program starts to execute, it first creates a symmetric key (K_{sym}) from a randomly generated number (step 1), encrypts it with K_{pub} , and sends it to the private program (step 2). The private program obtains K_{sym} by decrypting the received message with K_{priv} (step 3). At this point, the public program and the private program agree to encrypt the CFQ function call parameters and returns with the symmetric key K_{sym} . To prevent the known plaintext attack, we add a randomly generated nonce in each encryption of the CFQ function invocation and return. By doing so, the cypher text always appears different even if the parameter values (or the return values) are the same.

4.4.5 Loop Transformation

If the attacker is able to compromise the virtual machine, she can perform dynamic

tracing on the public program. When a loop in the public program contains a constant number of iterations and the loop variable increases or decreases continuously, the attacker can observe the CFQ function invocation and restore the control flow of the loop. For example, suppose an original program snippet is as shown in Figure 24 a); even though the program is transformed into the CFQ function invocation format, the attacker can still observe the value of variable i . When she observes that the CFQ function return changes from *true* to *false* when i increases from 2 to 3, she can infer that the original condition can be $i < 3$ or $i \leq 2$.

To mitigate such an attack, we propose the *loop transformation* technique. The intuition is as follows: instead of having the loop variable (i.e., i in Figure 24 a)) increase continuously (i.e., with an increasing step of 1), we set the loop variable increasing step to be a dynamically generated value. But we keep the loop iteration number the same as before. We give a concrete example later in this section. To hide the step generation procedure, we generate the loop variable increasing step through another loop and protect the loop condition with CFQ function invocation. To be clear, we call the original loop the *main loop*. We call the loop that generates the loop variable increasing step the *step generation loop*. The step generation loop repeatedly reduces a variable related to the input data by a constant value until the loop condition is no longer satisfied. Because we hide the loop condition with the CFQ function invocation, it is difficult for the attacker to derive the exact loop step value. Hence, it is difficult for the attacker to understand the original loop condition.

a)	<pre> //Original Program for(int i =0; i<3; i++){ print(i); } </pre>
----	---

b)	<pre> //Loop Transformation: select K=380, U=180, Δ=50, func(var)>180 int j = func(var); int x=0, p=0; while(j>180) j-=50; while(x<380){ x+=j; print(p); p++; } </pre>
c)	<pre> //Loop Transformation with CFQ function int j = func(var); int x=0, p=0; L1: while(CFQ(j-78, L1)) // predicate secret is 102 j-=Δ; L2: while(CFQ(125-x, L2)){//predicate secret is -255 x+=j; print(p); p++; } </pre>
d)	<pre> //Loop Transformation, assuming the loop iteration number is n //Generate random number Δ. (n+1)×Δ>U>n×Δ, and U×(n-1)<K<(U-Δ)×n. int j = func(var); //Step generation loop. while(j>U) j-=Δ; while(x<K){ x+=j; } </pre>

Figure 24 Example of Loop Transformation

The loop transformation procedure is shown in Figure 24 d). Suppose the original loop runs for a constant number of iterations, marked as n . We generate the constant numbers U , Δ and K as the loop transformation parameters and perform the transformation as follows. We introduce a variable j to hold the main loop variable increasing step. We initialize the value of j as a dynamically generated number. We insert a step generation loop before the main loop. The step generation loop reduces the value of variable j by Δ in each iteration, until the resulting variable is smaller than U . To be clear, we mark the initial value of j as j_0 and mark the resulting value of j after the step generation loop as j' . The resulting value of j' will be a dynamic number between U and $U-\Delta$. We introduce a new loop variable x in the main loop, initialize it to 0 and transform the main loop such that it increments x by j' in each iteration until the resulting variable is greater than K .

To prevent the attacker from performing pattern matching-based attacks (e.g., using the

pattern of decrementing j in the step generation loop as a hallmark to identify loop transformations), we can perform mathematical manipulations to hide the operation of decrementing variable j . We can also apply existing obfuscation techniques [22][23][24] to transform the step generation loop. For example, we can apply the variable encoding technique [23] to replace j with the negation of another variable, say l , and thus to replace the decrementing of j with the incrementing of l . Even further, we can replace the entire step generation loop with other implementations as long as the resulting value j' is within the range of U and $U-\Delta$.

To make the transformed loop preserve the original loop functionality, we have to select the values of j_0 , U , Δ , and K carefully. Specifically, we have the following theorem.

Theorem 11. To preserve the functionality of the original loop, we have to ensure that j_0 , j' , U , Δ and K satisfy the following formulas:

$$j_0 > U \quad (10)$$

$$U - \Delta < j' \leq U \quad (11)$$

$$U > \Delta \times n \quad (12)$$

$$U \times (n-1) < K \leq (U - \Delta) \times n \quad (13)$$

Proof: For the step generation loop, we need to ensure that j' is within the range $(U - \Delta, U]$ (i.e., formula (11)). To achieve that, we only need to ensure $j_0 > U$, i.e., the return value of function $func(var)$ is greater than U (formula (10)).

For the main loop, we need to set the condition such that the main loop returns *true* prior to the n^{th} iteration and returns *false* in the n^{th} iteration. For any value of j' , the value of the main loop iteration variable (i.e., x in Figure 24 d)) is $(n-1) \times j'$ after the $(n-1)^{th}$ iteration and

is $n \times j'$ after the n^{th} iteration. Therefore, by selecting K between $(n-1) \times j'$ and $n \times j'$ for any j' , we can preserve the functionality of the original loop. Because j' is within the range of $(U-\Delta, U]$, when j' is $U-\Delta$, the range of K is $((U-\Delta) \times (n-1), (U-\Delta) \times n]$; when j' is U , the range of K is $(U \times (n-1), U \times n]$. To ensure the value of K to be selected within the range of $((n-1) \times j', n \times j']$ for any j' , we have to ensure that the two ranges of K overlap, and K is selected from the overlapping range. I.e., $(U-\Delta) \times n > U \times (n-1)$ and $U \times (n-1) < K \leq (U-\Delta) \times n$. By simplifying the inequality, we obtain formulas (12) and (13). ■

From the attacker's perspective, she may find that one hallmark of the loop transformation is formula (12), i.e., $U > \Delta \times n$. The attacker may derive the range of U from dynamic tracing and verify if formula (12) is satisfied. If true, she can recognize the loop as an artifact of loop transformation, rather than an ordinary dynamic loop. However, we prove that, by carefully setting the transformation parameter U and Δ , we can defeat such an attack.

Theorem 12. By setting U and Δ to satisfy the following constraint, the attacker is unable to decide whether formula (12) is satisfied or not by analyzing the range of U . Therefore, we raise the bar for distinguishing loop transformations from regular dynamic loops.

$$U < \Delta \times (n+1) \quad (14)$$

Proof: From the attacker's perspective, if she can dynamically observe the program execution, she can observe the resulting main loop step value j' and count the iteration number n . Because the value of Δ is exposed in the public program, she can derive the approximate range of U based on j' and Δ . Given formula (11), the attacker has the range of U as

$$j' \leq U < j' + \Delta \quad (15)$$

The attacker can perform the following algorithm to determine if formula (12) is satisfied and thus determine whether the current loop is an artifact of the loop transformation:

- 1) For any $j' \in (U - \Delta, U]$, if $j' > \Delta \times n$, then $U \geq j' > \Delta \times n$ (by formula (15)) for all j' 's. The current loop is an artifact of the loop transformation (by formula (12)). In other words, for any j' , if formula (12) is satisfied, then the current loop is an artifact of the loop transformation.
- 2) If there exists a $j' \in (U - \Delta, U]$, such that $j' + \Delta \leq \Delta \times n$, then $U < j' + \Delta \leq \Delta \times n$ (by formula (15)) for some j' 's. The current loop is not an artifact of the loop transformation (by formula (12)). In other words, if there exists a j' that makes formula (12) unsatisfied, the current loop is not an artifact of the loop transformation.
- 3) Otherwise, there is an uncertainty. In other words, if the attacker finds $j' < \Delta \times n$ for some j' , she is not sure whether it satisfies formula (12) or not. Therefore, she is not sure whether the current loop is an artifact of the loop transformation or a real dynamic loop.

We can set the values of U and Δ properly to make the attacker fall into case 3). Because $U - \Delta < j' \leq U$ (formula (11)), by setting $U - \Delta < \Delta \times n$, we can make some $j' \in (U - \Delta, \Delta \times n)$ smaller than $\Delta \times n$. Simplifying it, we obtain $U < \Delta \times (n + 1)$. Therefore, by satisfying formula (14), we make the attacker unable to determine whether formula (12) is satisfied or not after analyzing the range of U . ■

By satisfying the constraints in Theorem 11 and Theorem 12 (formulas (10) through (14)), we ensure that the loop transformation preserves the original program's function and

make it difficult for the attacker to distinguish the loop transformation from an real dynamic loop. We propose the following algorithm to select the loop transformation parameters.

Loop Transformation Parameter Generation (LTPG) Algorithm: Suppose the original loop iteration number is a constant value n , we select the loop transformation parameters as follows to satisfy formulas (10) through (14).

- Step 1. Select a random positive number Δ .
- Step 2. Select a random positive number U such that $\Delta \times (n+1) > U > \Delta \times n$ (to satisfy formula (12) and (14)).
- Step 3. Construct a function $func(var)$, which takes an arbitrary non-zero-value variable in the current function as the argument and returns a number $j_0 > U$ (to satisfy formula (10)). If no variable is available for the argument of $func$, we generate a random number to feed the function $func(var)$. The implementation of $func(var)$ can be arbitrary, as long as the return value is greater than U . We describe our implementation in CAMR in Section 4.6.6.
- Step 4. Select a random number K within the range $(U \times (n-1), (U-\Delta) \times n]$ (to satisfy formula (13)).

Figure 24 b) shows the program after the loop transformation on Figure 24 a). Here, we choose Δ as 50, according to step 1) in the LTPG algorithm. Because $n=3$, we select U as 180, according to step 2). Accordingly, we select K as 380, according to step 4). Figure 24 c) shows the transformed program protected by the CFQ function. Because the original loop body contains a statement that involves the loop variable ($print(i);$), we introduce a

variable p and maintain it in the transformed main loop body to preserve the original program behavior.

Essentially, the format of the main loop condition after the transformation is unchanged: $x < K$ (in Figure 24 d)), where x is the loop variable, and K is a constant value. However, different obfuscations will result in different constant values of K . The transformation also makes the loop increasing step j' a dynamic value generated based on a variable value in the current function. From the appearance, because the loop increasing step j' is a dynamic value and K is a constant value, different values of j' may incur different iteration numbers. For example, suppose K is 380; $j'=10$ will incur 38 iterations, while $j'=38$ will incur 10 iterations. Therefore, from the appearance, the iteration number is determined dynamically, which makes the transformed loop appear to be a dynamic loop. At runtime, the main loop variable increasing step j' is dynamically generated based on the output of $func(var)$. Accordingly, the main loop variable value is dynamically computed. Take Figure 24 b) as an example; if $func(var)$ returns 300, after the step generation loop, the resulting j' will be 150. The value sequence of x in the main loop is 0, 150, 300, 450. If $func(var)$ returns 181, after the loop step generation loop, the resulting j' will be 131. The value sequence of x in the main loop is 0, 131, 262 and 393. Because K is set to 380, both main loop executions will iterate three times.

If the attacker performs taint analysis [25], she will find that the iteration number of the main loop is related to a variable in the current function, rather than a constant value. Therefore it is hard for her to connect such a dynamic loop with a constant-iteration-number-loop.

If the attacker performs pattern matching to discover the transformed loop, she will find

that the format of the step generation loop and the main loop are not significantly different from any common dynamic loops. Besides, the aforementioned mathematical and obfuscation skills will hide the pattern of step generation loop. Hence, it is hard for her to discover the transformed loop through static analysis.

If the attacker performs dynamic tracing, because the loop conditions (the step generation loop and the main loop) are protected by the CFQ function, she cannot tell whether the current loop is an artifact of the loop transformation by deriving the range of the loop transformation parameter, as proven in Theorem 12.

Our proposed solution is to transform loops whose loop variables monotonically increase. With minor modification, it can also be applied to loops where the loop variables monotonically decrease. Such a modification is straightforward and we skip the discussion due to the limited space.

We have to note that the loop transformation technique only works on loops with a constant number of iterations. It cannot protect dynamic loops because we can only hide a constant number in the private program. With a constant iteration number n , we can generate constant values of U and K and hide them in the private program. However, with a dynamic iteration number, we cannot generate constant values of U and K to preserve the original loop functionality.

4.5 Security Discussion

RCFO raises the bar for static analysis attacks. Firstly, CFQ predicates hide part of the control flow in the private cloud. When original conditions are transformed into CFQ conditions, predicate secrets (which determine the control flow) are stored in the private cloud. To the attacker who can access the public program, the control flow information is

not complete. Hence, static analysis on the public program alone does not give the attacker enough information about the control flow. Secondly, the inserted fake branch statements complicate the control flow graph and further confuse the attacker. The inserted fake branch statements are indistinguishable from the original branch statements and thus hide the real branch statements among the fake branch statements. From the appearance, a fake branch condition has the same CFQ condition format as that of the transformed original branch conditions. Although each fake branch condition is derived from the variable evaluation of the original condition, the expression aggregation and relaxation can reduce the correlation of the fake branch condition and the original branch condition. In addition, hiding predicate secrets in the private program makes it more difficult for the attacker to infer the who-derives-who relationship.

RCFO also raises the bar for dynamic analysis attacks. Firstly, the attacker cannot perform dynamic analysis offline. This is because, to observe the execution status of the public program, the private program has to collaborate with the public program. Because the private program is completely controlled by the cloud user, the attacker can only perform dynamic tracing when the cloud user sets up the private program.

Due to the CFQ function encryption, the attacker cannot eavesdrop on CFQ function invocations outside of virtual machines. This prevents the attacker outside of virtual machines from dynamically tracing the program. The attacker has to break into virtual machines to perform dynamic tracing attacks. However, breaking the public cloud security defense is not easy. Even if the attacker is able to compromise virtual machines and dynamically trace virtual machine executions, RCFO still offers multiple lines of defense.

If the attacker performs dynamic tracing attacks on the program, she can obtain

information by observing the arguments and the return values of CFQ function invocations. However, such tracing information can only give the attacker vague information about original predicates. Suppose the attacker observes that a CFQ function call $CFQ(x+9, LI)$ returns *true* when x is 19; the attacker can infer that the original predicate could be $x > n$, where $n \leq 19$ or $n < 19$. However, she cannot further understand the exact value of n without other information.

The attacker may perform brute force attacks. For instance, suppose a CFQ function call is $CFQ(x+9, LI)$. The attacker can repeatedly call this CFQ function with continuously increased values of x until the function's return value changes from *false* to *true*. In this case, the attacker can guess that the very value of x that triggers the change is related to the predicate secret. Because we assume the control flow integrity is protected by the remote attestation techniques in Section 4.3, such an attack will be detected if the call site sequence in the CFQ function invocations violates the control flow check in the private program.

The attacker may passively observe every CFQ function invocation. If she finds the expression variable value of a CFQ function increases/decreases continuously, she can derive the predicate secret, as described in Section 4.4.5. The loop transformation introduced in Section 4.4.5 transforms the loop with a constant number of iterations into a dynamic loop, so that the loop variable increasing step is dynamically generated. Because the conditions of step generation loops are protected by the CFQ function, by carefully selecting transformation parameters, it is difficult for the attacker to distinguish transformed loops from ordinary dynamic loops. The detailed analysis can be found in Section 4.4.5.

Finally, the attacker may perform dynamic tracing to mark unexecuted branches as fake branches. Such a method is risky to the attacker because it could incur false positives due to the incompleteness of dynamic analysis. As a result, the attacker may mistakenly categorize the un-triggered genuine branch as a fake branch.

4.6 System Implementation

We have developed a prototype system called *CAMR* to perform RCFO on Apache Hadoop (a popular MapReduce implementation) jobs. Our experiments show that CAMR can protect MapReduce jobs running on *Amazon Elastic MapReduce (Amazon EMR)*, a popular cloud-based MapReduce service.

To implement the obfuscation system, we use *Soot* [20], an open source Java-based compiler tool, to perform program analysis and program transformations. We use *Symja* [19], an open source Java-based computer algebra system, to perform mathematical transformation. CAMR reads Java *.class* files and generates obfuscated Java *.class* files. Hence, our tool can directly obfuscate Hadoop job files without the source code.

4.6.1 Condition Transformation

The transformation is performed in the *gb* phase of Soot analysis. To transform original conditions into CFQ conditions, we use Soot to extract the condition in each of the branch statement and perform condition normalization with the rule presented in Figure 18. The generated inequalities are represented in Soot expression objects. To manipulate the inequality, we translate Soot expression objects into string-based arithmetic expressions and send them to Symja. Because Symja can only perform arithmetic operations, we translate the arithmetic operator in the Soot expression object into a Symja operator and translate other operations that Symja does not recognize or will misunderstand into Symja

variables. For instance, when we have a Soot expression $x+func(y)$, we translate it into $x+funcLBRyRBR$. Here, “(” and “)” are keywords in Symja script. However, they are used in Symja to represent arithmetic operation priorities, rather than function calls. Thus we replace them with “LBR” and “RBR”, respectively.

Symja takes a string representing an arithmetic inequality as input and returns another string representing the reorganized inequality as output. The reorganization is described in Section 4.4.1.2. The LHS of the returned inequality is the predicate expression. The RHS is the predicate secret. From the RHS of the inequality, we extract the predicate secret and write it in a file, along with the inequality relational operator (i.e., “>” or “>=”). This file will be sent to the private cloud and loaded by the private program. From the LHS, we parse the string as an abstract syntax tree so that we can extract the predicate expression and convert it back to a Soot expression object.

After that, Soot replaces the original condition with the CFQ condition. Specifically, it replaces each inequality in the transformed predicate with an invocation to the CFQ function. The function call takes two parameters: the extracted predicate expression and the function call site identifier. The function call site identifier is a string consisting of the current class name, the current function name and a unique number in the current function to identify the function call. By using such a nomenclature for the function call site identifier, we guarantee that each function call site identifier is unique.

4.6.2 Fake Branch Statement Insertion

To insert fake branch statements, we use Soot to perform two rounds of program scans. In the first round, we perform intra-procedural forward data flow analysis on each function of the original program. The implementation is the same as the algorithm described in

Figure 20. The generated result is stored in an array. Each element in the array is the variable evaluation of a statement in the original program.

In the second round, we generate a fake branch predicate for each insertion candidate. For each insertion candidate, we pick the corresponding variable evaluation and perform expression relaxation and aggregation as described in Section 4.4.2.2. Because the variable evaluation is represented in Soot expression objects, we translate it into a string that applies Symjas syntax and performs expression relaxation and aggregation in Symja. After that, we translate the generated string back into a Soot expression object and construct the fake branch statement with the generated expression object.

4.6.3 Private Program Implementation

According to Section 4.6.1, each call site identifier contains the program class and function information. This makes the CFQ function invocation universally unique so that multiple public programs can share one private program. Therefore, our implementation only deploys one private program on the private cloud for all MapReduce jobs. Whenever a MapReduce job is obfuscated, the predicate secrets, the inequality operators and the call site identifiers are sent to the private program. The private program loads such information for the private CFQ function to look up. Whenever the private CFQ function is invoked, it first looks up the predicate secret and the inequality operator by the passed call site identifier. Then it compares the passed predicate expression value with the predicate secret via the inequality operator and returns the comparison result.

4.6.4 Hadoop Job Obfuscation

A Hadoop job is usually packaged into a *.jar* file, which contains multiple Java *.class* files. CAMR can directly obfuscate *.class* files. However, if we directly feed a *.class* file

from a Hadoop job into Soot to perform analysis, Soot will throw an exception. This is because each *.class* file references Hadoop framework classes and functions, which Soot cannot find. However, feeding the entire Hadoop framework into Soot is infeasible given the complexity of the Hadoop implementation. In this case, we only need to generate stub Hadoop framework classes and functions, which will be referenced by the job's *.class* files when performing RCFO. At runtime, the transformed job will reference the real framework classes and functions. Because the number of referenced framework classes and functions are limited and the implementation of the stub functions is trivial, such work can be done quickly and can be reused for all Hadoop jobs. For example, our implementation generates 1,502 lines of Java code, which covers all the classes and functions referenced in our experiments.

During obfuscation, we put the Hadoop framework stub files together with the original job *.class* files to perform program analysis and transformation. After obfuscation, we generate the job *.jar* file based on the transformed job *.class* files.

4.6.5 CFQ Function Encryption

We implemented a lightweight CFQ function encryption using the Java standard encryption library. We will leave the full-fledged implementation to future work. To implement CFQ function encryption, we generate K_{sym} in step 1 of Figure 23 through the Java random number generator and SHA-1 hash algorithm. We use RSA to implement the asymmetric key encryption scheme, which is performed in steps 0, 2 and 3. Each private CFQ function invocation (steps 4, 5 and 6) is encrypted with the symmetric encryption scheme AES. Because K_{sym} never leaks, all private CFQ function invocations in a task can share one K_{sym} . Therefore, the CFQ function encryption key establishment is only

performed once for each MapReduce task at the beginning of the task execution.

4.6.6 Loop Transformation

We perform the loop transformation technique on loops that contain a constant number of iterations. For each transformation, we choose the transformation parameters according to LTPG algorithm in Section 4.4.5. We first generate a random positive number Δ . For the constant loop iteration number n , we generate a random number U such that $\Delta \times (n+1) > U > n \times \Delta$. To generate j_0 , we insert a function call $func(var)$, which receives a non-zero numeric value as the argument and returns a value $j_0 > U$. If there is no variable defined before the loop, we generate a random number and feed it to $func(var)$. In the prototype implementation, we implemented $func(var)$ straightforwardly. It can be replaced with other more complicated implementations. In our implementation of $func(var)$, it first converts var into a number greater than 1: if var is a negative number, we update it by negating it; if the absolute value of var is smaller than 1, we update it with its reciprocal. After that, we keep on updating its value by performing square operations on the latest value, until the resulting value is greater than U . We return the final result as j_0 . With the value of n and the selected U and Δ , we generate a random number K between $U \times (n-1)$ and $(U-\Delta) \times n$. With the generated parameters, we transform the original loop according to Figure 24 d).

4.6.7 Obfuscation Case Study

To give readers an intuition about CAMR and RCFO, we show an obfuscation of the function *addKey* in the class *SortGenMapper* of the *TeraGen* application in Figure 25. The TeraGen application is available with Hadoop 0.20.205 (see TABLE 10). Here, we set the obfuscation degree to 0.5. We present the obfuscated public program in the format of

grimp, a representation of Java programs that is closer to the Java byte code. Notice that, in the grimp representation, the loop statements, such as *for* and *while*, are represented as more primitive statements, such as *if* and *goto*. According to Figure 25, each branch statement predicate is transformed into CFQ function invocation *CFQ(predicate expression, call site identifier)*. Here, we simplify the call site identifier as a number rather than the universal format just for clarity.

```
//Original Function
private void addKey() {
    for(int i=0; i<3; i++) {
        long temp = rand.next() / 52;
        keyBytes[3 + 4*i] = (byte) (' ' + (temp % 95));
        temp /= 95;
        keyBytes[2 + 4*i] = (byte) (' ' + (temp % 95));
        temp /= 95;
        keyBytes[1 + 4*i] = (byte) (' ' + (temp % 95));
        temp /= 95;
        keyBytes[4*i] = (byte) (' ' + (temp % 95));
    }
    key.set(keyBytes, 0, 10);
}
```

```
//Obfuscated Function, Grimp format
private void addKey(){
    org.apache.hadoop.examples.terasort.TeraGen$SortGenMapper r0;
    int i0, i1, i2;
    long l3, l11, l18, l25;
    r0 := @this;
    // Loop step generation
    i0 = r0.func((int) (java.lang.Math.random() * 100.0));
    i1 = 0;
    goto label2;
label0:
    if CFQ(i0 + -1.0 * 168, 123) goto label1;
label1:
    i0 = i0 + -50;
label2:
    if CFQ(i0 + -1.0 * 172, 124) goto label0;
    // End of loop step generation
    i2 = 0;
    if CFQ(-1.0 * i0 + 187, 125) goto label3;
    // start of original loop
label3:
    goto label13;
label4:
    if CFQ(-1.0 * i0 + 189, 126) goto label5;
label5:
    l3 = r0.rand.next() / 52L;
    if CFQ(-1.0 * i2 + 406, 127) goto label6;
label6:
    r0.keyBytes[3 + 4 * i1] = (byte) (int) (32L + l3 % 95L);
    if !CFQ(-1.0 * i0 + 184, 128) goto label5;
label7:
```



```

    l11 = 13 / 95L;
    r0.keyBytes[2 + 4 * i1] = (byte) (int) (32L + l11 % 95L);
    if CFQ(-1.0 * i0 + 192, 129) goto label8;
label8:
    l18 = 111 / 95L;
    if !CFQ(-1.0 * i0 + 194, 130) goto label10;
label9:
    r0.keyBytes[1 + 4 * i1] = (byte) (int) (32L + l18 % 95L);
    if CFQ(-1.0 * i2 + 404, 131) goto label10;
label10:
    l25 = 118 / 95L;
    if CFQ(-1.0 * i0 + 191, 132) goto label11;
label11:
    r0.keyBytes[4 * i1] = (byte) (int) (32L + l25 % 95L);
    i2 = i2 + i0;
    if !CFQ(-1.0 * i0 + 191, 133) goto label9;
label12:
    i1 = i1 + 1;
label13:
    if CFQ(-1.0 * i2 + 393, 134) goto label4;
//End of original loop
    if !CFQ(-1.0 * i0 + 191, 135) goto label11;
label14:
    r0.key.set(r0.keyBytes, 0, 10);
    if !CFQ(-1.0 * i0 + 192, 136) goto label4;
label15:
    return;
}

```

Figure 25 Obfuscation of SortGenMapper.addKey in TeraGen job

Figure 25 shows that the loop transformation is firstly performed on the original *for* loop. CAMR generates the random numbers K , U and Δ as 380, 180 and 50, respectively, to satisfy formulas (1) through (5). Because there is no variable available before the transformed loop, we generate a random number that is between 0 and 100 and feed it as an argument to the invocation of *func(var)*. Based on the argument *var*, the function *func(var)* generates a number $j_0 > U$. The step generation loop starts from the beginning of the function and ends after *label2*. It initializes the loop step i_0 by invoking the function *func(var)* (i.e., j_0) and continues decreasing i_0 by 50 (Δ) until i_0 is smaller than 180 (U). The resulting i_0 is the main loop increasing step. RCFO transforms the main loop (from *label3* to *label13*) by making it increase the loop variable i_2 by the loop step (i_0) repeatedly until i_2 exceeds K (in *label13*). The loop condition is transformed to CFQ function invocations to hide the transformation parameters K , U and Δ . In addition, CAMR inserts eleven fake branch statements to function *addKey*, and they are marked in *italic* font. Those fake branch statements complicate the control flow graph and make it difficult for

the attacker to distinguish original branch statements from fake branch statements.

4.7 Experiments

We performed a series of experiments to evaluate CAMR. Our experiments were launched on EMR. In EMR, the customer sets up a MapReduce cluster consisting of multiple EC2 instances and uploads the MapReduce job and the job input onto the Amazon public cloud. The MapReduce job is executed on the cluster, and the job result is stored on the public cloud. The detailed use case can be found in Figure 14. We used such an EMR service as a baseline and compared it against CAMR. CAMR requires a similar workflow. The difference is that the user has to perform RCFO on the MapReduce job and upload the public program onto the public cloud. In addition, the user needs to upload the private program onto the private cloud. The entire MapReduce job is executed via the coordination of the public program on EMR and the private program on the private cloud.

In our experiments, we used Hadoop 0.20.205.0 on EMR and set up different clusters for different experiment sets. We deployed the private cloud on a local Linux server, which was configured with a 2.93 GHz 8-core Intel Xeon CPU and 16 GiB of RAM. Because the security of RCFO was analyzed in Section 4.5, in this section, we only focus on the performance evaluation. We define the *performance overhead* as the percentage of the extra execution time incurred in CAMR, compared with the baseline. We measured the performance overhead in two sets of experiments. In the first set, we launched the performance test. We chose four Hadoop applications and observed the performance overhead by changing the obfuscation degree d . In the second set of experiments, we performed the scalability test. We chose three Hadoop applications from the previous experiment set, increased the computation load, and observed the changing trend of the

performance overhead. For the same Hadoop application with the same configuration, we repeated the job execution ten times and recorded the average execution time.

4.7.1 Performance Test

To launch the performance test, we set up an EMR cluster on the Amazon public cloud that consists of one small EC2 instance as the master and two small EC2 instances as workers (each small EC2 instance was configured to have one vCPU, 1.7 GiB of memory and 160 GB of storage). We set up the private cloud on a local Linux server, which was configured with a 2.93 GHz 8-core Intel Xeon CPU and 16 GiB of RAM. We chose four applications released for Hadoop framework 0.20.205, obfuscated the application with the obfuscation degrees 0, 0.5 and 1.0, and measured the performance overhead.

Job Name	Description	Parameters and Input Settings
Word Count	Count the occurrence of each word in the input text files.	100 text files downloaded from free ebook project website <i>Gutenberg</i> [27]. The total data size is 78 MB.
Pi	Compute the value of Pi with the Monte-Carlo method.	Generate 10^4 samples to calculate the value of Pi. Each task generates 1,000 samples.
TeraGen	Generate random text strings that can be used for sorting.	Randomly generate 10^6 text records.
TeraSort	Sort text strings in alphabetical order.	Sort the 10^6 records generated by TeraGen.

TABLE 10 Hadoop Jobs Used in the Experiments

The Hadoop applications used in this set of experiments are listed in TABLE 10. The results show that CAMR is able to obfuscate all four jobs. The execution time and the performance overhead of the four applications are shown in Figure 26.

Figure 26 Execution Times of MapReduce Jobs with Different Obfuscation Degree d

According to Figure 26, the performance overhead increases with the increase of the obfuscation degree d . However, for each job, the performance overhead is modest, even when d is increased to 1.0. According to the figure, when the obfuscation degree d is increased from 0 to 1.0, the performance overhead increases from 5.4% to 10.6% for Word Count, stays at 14.6% for Pi, increases from 15.8% to 62.6% for TeraGen, and increases from 23.8% to 44.9% for TeraSort. The results also show that CAMR incurs moderate performance overhead, even if the obfuscation degree d is set to 1.0, i.e., we insert a fake branch statement before each insertion candidate. We believe that the continuous cache effectively reduces the number of private CFQ function invocations and therefore reduces the performance overhead. To confirm our belief, we also counted the number of public and private CFQ function invocations in the above four jobs. For each job, we counted the number of public and private CFQ function calls in each map or reduce task and computed the average number for map tasks and reduce tasks, respectively. Because TeraGen and TeraSort do not invoke CFQ functions in their reduce tasks, we only counted the invocations in their map tasks. The average numbers of invocations and the cache-hit rates are shown in TABLE 11.

Job Name	M/R	d=0	d=0.5	d=1.0
----------	-----	-----	-------	-------

		pub	priv	cache hit rate(%)	pub	priv	cache hit rate(%)	pub	priv	cache hit rate(%)
Word Count	M	1.27E+6	6	99.9996	2.55E+6	13	99.9995	3.84E+6	18	99.9995
	R	3.04E+6	3	99.9999	6.08E+6	6	99.9999	1.03E+7	10	99.9999
Pi	M	1.55E+4	96	99.3806	4.26E+4	138	99.6761	6.31E+4	170	99.7306
	R	105	10	90.4762	116	10	91.3793	197	17	91.3706
TeraGen	M	7.50E+6	71	99.9991	1.80E+7	172	99.9990	3.90E+7	396	99.9990
TeraSort	M	1.50E+6	9	99.9994	2.00E+6	16	99.9992	4.00E+6	24	99.9994

TABLE 11 CFQ Function Invocation Statistics and Cache-hit Rates

The results show that the continuous cache greatly reduces the private CFQ function invocations and therefore reduces the cross-cloud network communication. For instance, in Word Count, when the obfuscation degree is 1.0, the average number of public CFQ function invocations in a map task is 3.84E+6, while the average number of private CFQ function invocations for each map task is only 18. The cache-hit rate in this setting reaches 99.9995%. TABLE 11 shows that the cache-hit rate stays above 90% for all experiments we performed. In the six different MapReduce tasks, five out of six tasks achieve more than 99% cache-hit rates.

4.7.2 Scalability Test

We observed that the performance overhead changes when the job computation load scales up. To speed up the experiment, we set up an EMR cluster consisting of one small EC2 instance running as master and four medium EC2 instances running as workers (each medium instance was configured with one vCPU, 3.75 GiB of memory and 410 GB of storage.). We continued to use the same private cloud configuration. In this set of experiments, we set d as 0.5 and used three applications: Word Count, Pi and TeraSort. For Word Count, we increased the computation load linearly. For Pi and TeraSort, we increased the computation load exponentially. All three applications showed no significant

performance overhead increase when the computation load scaled up.

For Word Count, we generated the job input with *RandomTextWriter*, a MapReduce job released in Hadoop 0.20.205. We linearly increased the job input size from 4 GB to 20 GB. Accordingly, the number of map tasks increased linearly. The execution time and the performance overhead are shown in TABLE 12.

Input Data Size	Map Task Number	Baseline Execution Time (s)	CAMR Execution Time (s) (d=0.5)	Performance overhead
4 GB	136	566	628	11.02%
8 GB	272	1019	1136	11.45%
12 GB	408	1518	1705	12.32%
16 GB	544	2010	2288	13.83%
20 GB	680	2476	2781	12.33%

TABLE 12 Scalability Test with Word Count

The results show that, when the input size increases from 4 GB to 20 GB, the performance overhead remains between 11.02% and 13.83%. We observe that the performance overhead increases slightly when the input data size grows from 4 GB to 16 GB. However, we also observe a performance overhead drop when the input data size increases from 16 GB to 20 GB. Hence, it is possible that the performance overhead increase from 4 GB to 16 GB is merely a reflection of the experiment's randomness. In general, no significant performance overhead increase is observed when the input data size grows linearly from 4 GB to 20 GB.

We further increased the computation load growth step to observe the change in performance overhead. This time, we used Pi and TeraSort and exponentially increased the computation load.

For Pi, we continued to set each map task to generate 1,000 samples. However, we increased the map task number from 10 to 10,000. By increasing the number of map tasks,

the job was able to take more samples to evaluate the value of Pi and was therefore able to generate a more accurate Pi value. We collected the execution time for each job and computed the average execution time for each type of job. The average execution time and the performance overhead are shown in TABLE 13.

Total Sample Number	Map Task Number	Baseline Execution Time (s)	CAMR Execution Time (s) (d=0.5)	Performance Overhead
10 ⁴	10	56	63	12.57%
10 ⁵	100	125	134	7.20%
10 ⁶	1000	831	890	7.10%
10 ⁷	10000	7812	8537	9.28%

TABLE 13 Scalability Test with Pi

These results show that when the number of map tasks is increased from 10 to 100, the CAMR incurs less performance overhead; i.e., the performance overhead drops from 12.57% to 7.20%. When the number of map tasks scales up from 100 to 1,000, the performance overhead is stable at 7.10%. When the number of map tasks scales up to 10,000, the performance overhead increases slightly to 9.28%. These results indicate that when the computation load exponentially increases, the CAMR does not incur significant increases in performance overhead.

For TeraSort, we exponentially increased the job input size. Specifically, we used TeraGen to generate input data containing 10 MB, 100 MB, 1 GB and 20 GB of records. Because each record requires 100 bytes, the input data sizes scaled up from 1 GB to 200 GB. The average execution times for the different input data sizes are shown in TABLE 14. When the input data size is 1 GB, the performance overhead of the CAMR is 22.22%. When the input data size scales up to 10 GB, the performance overhead drops to 5.45%. When the input data size further scales up, the performance overhead decreases slowly from 4.76% to 3.59%. The performance overhead changes are similar to those for Pi; when

the input data size scales up, CAMR does not incur significant increases in performance overhead.

Input Data Size	Map Task Number	Baseline Execution Time (s)	CAMR Execution Time (s) (d=0.5)	Performance Overhead
1 GB	16	108	132	22.22%
10 GB	80	716	755	5.45%
100 GB	752	7353	7703	4.76%
200 GB	1504	15415	15968	3.59%

TABLE 14 Scalability Test with TeraSort

4.7.3 Discussion

In summary, the experimental results show that the CAMR incurs modest performance overhead. Overall, for the same job, higher obfuscation degree values incur greater performance overheads. However, in the four applications that we tested, the performance overhead increases modestly even if the obfuscation degree increases to 100%. With the same obfuscation degree, when the job computation load is scaled up, the performance overhead does not significantly increase. Our experiments also show an interesting fact: When the computation load is exponentially scaled up, the performance overhead drops to a lower value and remains stable afterwards. This fact can potentially be explained as follows: when the computation load is small, establishing the communication connection for the private CFQ function call occupies a greater percentage of the overall job execution time. When the computation load is scaled up, the number of private CFQ function calls is also scaled up. Hence, the time used for the connection set up is amortized in the vast number of private CFQ function invocations.

We believe that we can further reduce the performance overhead by optimizing our implementation. Our current implementation requires each task to maintain a continuous cache that cannot be reused by other tasks. Different tasks have to call the private CFQ

function independently, even if the CFQ function call site and parameters are the same. In reality, all map (or reduce) tasks in a job execute the same code and can therefore share the continuous cache. By centrally managing the continuous cache on the master, we can expect a lower number of private CFQ function invocations when the task number scales up. This implementation requires a modification of the Hadoop framework. We were unable to realize this modification in our EMR-based experiments because EMR does not support the customized Hadoop framework. We expect this modification to be implemented via the collaboration of public cloud vendors.

4.8 Related Work

The confidentiality of cloud computing can be divided into two categories: The data confidentiality and the algorithmic confidentiality. The former problem is about how to protect the data manipulated in the computation program. The latter problem is about how to protect the algorithm of computation program. We discuss the related works for the two problems respectively.

Researchers have proposed a number of solutions to address the data confidentiality problem. Some researchers worked on encryption-based solutions for certain frequently used data operation such as SQL query [13], JOIN [12], SUM and AVG [14]. On the other hand, some researchers were seeking generalized solutions. For MapReduce computation, the Airavat platform [11] incorporates differential privacy mechanisms into the MapReduce execution framework by automatically adding noise to the output data. Such a class of solutions protects the privacy with the cost of losing result accuracy. Alternatively, Zhang et al. proposed the Sedic system [10] to use hybrids cloud architecture on the MapReduce computation. It pre-labels the input data and directs the sensitive input and

sensitive intermediate data to the trusted private cloud and finally merges the sensitive and the non-sensitive data on the private cloud.

Solutions toward algorithmic confidentiality problem lay in the direction of program obfuscation, a protection technique to make code unintelligible to reverse engineering attacks and thereby to hide the algorithm details. However, a generalized obfuscation solution is still lacking due to the great challenge of this problem.

Indeed, the study of Barak et al. [21] noted that every obfuscator will fail to *completely* obfuscate some programs. Garg et al. [66] proposed the first candidate cryptographic construction for a general-purpose obfuscator, an indistinguishable obfuscation scheme that can obfuscate all polynomial-size, log-depth circuit (i.e., NC^1). Many theoretical works followed this direction to explore new schemes [80][82] or improve the performance of existing solutions [81]. However, to this point, all such works are still limited in theoretical studies. The performance of such class of schemes is far from practical.

Another school tries to seek practical solutions that make the program transformation *difficult* rather than *impossible* to reverse engineer. Since the control flow, which decides the sequence of instructions to be executed, directly reflects the algorithm of a program, many researchers proposed solutions to obfuscate the program control flow. Our work falls in this direction.

Wang et al. [16] proposed a control-flow flattening technique that makes all basic blocks appear to have the same set of predecessors and successors. The actual control flow during execution is dynamically determined. The authors proved that the combination of control flow flattening and pointer alias construction makes the static analysis of the obfuscated

control flow a NP-complete problem. However, their obfuscation work cannot defend against dynamic analysis attacks. Udupa et al. [50] showed that in practice, Wang’s obfuscation can be defeated by a simple combination of static and dynamic analysis.

Collberg et al. [51] proposed to construct opaque predicates to impede the control flow de-obfuscation. An opaque predicate is a conditional expression whose value is known to the obfuscator, but is difficult for an adversary to deduce statically. Following Collberg’s work, several variants of the opaque predicate construction method were proposed to construct dynamic opaque predicates. [52] and [53] proposed dynamic and distributed opaque predicates, where values of the predicates can only be evaluated at runtime. To some extents, our obfuscation approach is inspired by opaque predicates. However, we construct the opaqueness by retrieving predicate secrets and hide them in the trusted private cloud. In addition, our solution (continuous cache) can limit the performance overhead to a more modest range.

Monirul et al. [54] proposed a control flow obfuscation technique that encrypts the code of specific trigger-based behavior with the trigger condition value and changes the trigger condition into the format that compares the hash value of variable expressions with a certain value. This method is effective in concealing trigger-based behaviors. However, because this technique can only handle the “equal” and “not equal” condition predicates, it is not suitable for generalized control flow obfuscation.

Wang et al. [55] proposed a linear obfuscation method that introduces unsolved mathematical conjectures into branch conditions to increase the difficulty of reasoning about the branch conditions using symbolic execution. However, the authors did not demonstrate the security of this approach against other analysis techniques. Additional,

their method can be defeated by pattern matching because the number of known conjectures is limited.

The above works enable the attacker to observe the entire program statically and dynamically and therefore provide the attacker the possibility of reverse engineering the control flow. Our work hides part of the control flow from the attacker and therefore increases the difficulty of reverse engineering attacks.

Ge et al. [15] worked at the executable binary level and proposed the extraction of the control flow into an independent process that can be deployed in a trusted environment. Their idea of hiding control branch information within the trusted entity is similar to ours. However, their control flow information is stored in a jump table, which can only record the static block predecessor-successor relationship and cannot handle dynamically computed control flow transfers. Additionally, their work involves a significant performance overhead. When two processes are executed on a single machine, and the shuffle rate is set to 5% (i.e., 5% of the control flow is hidden in the monitor process), their system incurs a 108% to 337% performance overhead. When the shuffle rate is 20%, their system incurs a 277% to 887% performance overhead.

4.9 Summary

In this chapter, we proposed Confidentiality Assurance MapReduce (CAMR), a hybrid cloud-based MapReduce framework to protect the control flow confidentiality. The CAMR framework performs the Runtime Control Flow Obfuscation technique to transform the original program into a public program and a private program, which will be deployed to the public cloud and the private cloud, respectively. The transformation keeps most original computation on the public program, and maintains the control flow

information on the private cloud to protect the control flow confidentiality. We implemented a prototype system and deployed it onto a real hybrid cloud environment. Our security analysis shows that CAMR can defeat static analysis-based reverse engineering attacks and raises the bar for dynamic analysis-based reverse engineering attacks. Our experimental results show that CAMR incurs modest performance overhead on each MapReduce jobs that we tested. Our work is published in a peer-reviewed journal [65].

5 DISCUSSION

We claim that even though CAMR and IAMR both focus on MapReduce computation, the techniques proposed in the two systems can be generalized to a wide class of outsourced computing scenarios. We will discuss the possible directions to extend our proposed techniques and their potential problems.

5.1 Integrity Assurance MapReduce

The result checking technique proposed in Chapter 3 can be applied to many other outsourced computing applications as long as that application complies with the assumption that we made in Section 3.3. We list several possible scenarios where the result checking technique can be applied.

1. Volunteer computing.

Volunteer computing is a type of distributed computing in which computer owners donate their computing resources (such as processing power and storage) to finish a certain big computing job. For example, the SETI@home project [37] utilizes the computing power of volunteer computers to perform observational analysis to detect intelligent life outside of Earth. Since anyone can contribute to the computation, malicious participant can also offer incorrect results to compromise the accuracy of the computation. For example, it has been reported that the SETI@Home project suffered from cheating behaviors from malicious volunteers who fake the number of work units completed in order to gain higher ranking on the website list of top donators [40].

Since tasks in volunteer computing are highly paralleled, the result checking technique presented in Section 3.4.1 can be performed on volunteer computing tasks.

2. Crowd computing

Crowd computing leverages the power of people on the Internet to complete tasks that are hard for individual users or computers to do alone. In this computing paradigm, the computing outsourcer categorizes similar tasks into batches and assigns tasks to humans on the Internet, which we call workers, to solve. The computing outsourcer receives task results from workers and pays each worker based on the returned tasks. Since crowd computing workers are rewarded according to their returned tasks, malicious workers may return trivial incorrect results to expedite their progress, and thereby to earn more money. Since tasks in crowd computing are highly parallel, we can also apply the result checking technique presented in Section 3.4.1 to achieve high result integrity.

5.2 Confidentiality Assurance MapReduce

The Runtime Control Flow Obfuscation (RCFO) technique proposed in Chapter 4 is a generalized technique that can be applied to most programming languages that support branch statement and network programming. For those programming languages, we can extract branch statement predicates, transform them with RCFO and deploy the transformed public programs and private programs onto the public cloud and private cloud, respectively.

One advantage of CAMR is that the program transformation does not need source code. For example, we can perform program analysis and transformations on java *.class* files by using Soot. To apply RCFO on other programs written with other programming languages, a similar tool that works on other programming languages is needed.

CAMR incurs modest performance overhead. It is because in CAMR, only the map and reduce functions in the job are obfuscated. The MapReduce framework is not obfuscated.

In other words, cross-cloud Control Flow Query invocation is performed only when the map and the reduce functions are invoked, which can greatly mitigate the performance delay. If we apply RCFO on other outsourced programs, protecting the entire program may incur high performance overhead. We may need to extract algorithmic-critical functions and apply RCFO on those functions only.

6 CONCLUSION

In this dissertation, we addressed two security problems for the cloud-based MapReduce service: the result integrity problem and the algorithmic confidentiality problem. We proposed a unified architecture, the hybrid cloud architecture, to address the two problems and proposed separate solutions. The principle of our solutions is to leverage the trustworthiness of the private cloud to retain the security at the cloud user's hand, meanwhile outsourcing majority computation to the public cloud.

To address the result integrity problem, we proposed the Integrity Assurance MapReduce (IAMR) framework to guarantee high result integrity of MapReduce computation, even if the computation is executed on a public cloud that contains malicious computing nodes. IAMR performs the result checking technique to reduce the job error rate. Specifically, the result checking technique performs probabilistic task/sub-task replication on the public cloud, performs probabilistic task/sub-task verification on the private cloud, and maintains a trust management mechanism for each worker to ensure the participating computing node are benign. We modeled the IAMR and the attacker as a two-player zero sum game and proposed two algorithms, the Interactive Gradient Descent (IGD) algorithm and the Tiered Interactive Gradient Descent (TIGD) algorithm, to search for optimal parameter values that can achieve lowest job error rate upper bound. The IGD algorithm can find the parameter settings for IAMR based on the user's accuracy requirement. The TIGD algorithm can find the parameter settings not only based on the user's accuracy requirement, but also on user's system restriction setting. Our theoretical analysis showed that our solution can guarantee a very low job error rate. For example, when we set the batch size as 50 and the replication probability as 0.5, we can guarantee

less than 1% of job error rate when half of workers on the public cloud are malicious. We implemented a prototype system for a real hybrid cloud environment (a hybrid cloud consisting of a local private cloud and Amazon EC2) and performed a series of experiments to evaluate its performance. Our experimental results showed that IAMR can guarantee a low job error rate and incurs moderate performance overhead. For instance, it incurs 19% to 83% of delay in the map phase and 29% of delay on in the reduce phase.

To address the algorithmic confidentiality problem, we focused on protecting the control flow confidentiality and proposed the Confidentiality Assurance MapReduce (CAMR) platform. We proposed a novel control flow obfuscation technique called Runtime Control Flow Obfuscation (RCFO) to hide information on program predicates and complicate the control flow graph. To further protect the control flow, RCFO also performs the fake branch statements insertion method, the CFQ function encryption scheme and the loop transformation technique to further raise the bar for reverse engineering attacks. To reduce the performance overhead, RCFO maintains a continuous cache to reduce the cross-cloud CFQ function invocation. We developed a prototype system for a real hybrid cloud environment (a hybrid cloud consisting of a local private cloud and Amazon EMR). Our system can apply RCFO on MapReduce job programs and run the obfuscated job on the public cloud. Our experimental results show that the average performance overhead ranges from 14.9% to 33.2% when the obfuscation degree increases from 0 to 1.0.

REFERENCES

- [1] “Cloud Security: Amazon’s EC2 serves up 'certified pre-owned' server images”
<http://dvlabs.tippingpoint.com/blog/2011/04/11/cloud-security-amazons-ec2-serves-up-certified-pre-owned-server-images>
- [2] Sven Bugiel, Stefan Nürnberger, Thomas Pöppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. “AmazonIA: when elasticity snaps back”. Proceedings of the 18th ACM conference on Computer and communications security. (2011), October, 17-21; Chicago, USA
- [3] Bowers, Kevin, et al. “HAIL: a high-availability and integrity layer for cloud storage.” Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009.
- [4] Popa, Raluca Ada, et al. “Enabling security in cloud storage SLAs with CloudProof.” Proceedings of 2011 USENIX Annual Technical Conference, Portland, OR. 2011.
- [5] Dean, Jeffrey, et al. “MapReduce: simplified data processing on large clusters”. Communications of the ACM, 51(1), 107-113.
- [6] Du, W., Jia, J., Mangal, M., and Murugesan, M. Uncheatable grid computing. International Conference on Distributed Computing Systems, (pp. 4-11), 2004.
- [7] Golle, P., & Stubblebine, S. “Secure distributed computing in a commercial environment.” Financial Cryptography (pp. 289-304). Springer Berlin Heidelberg, 2002.
- [8] Wei, W., Du, J., Yu, T., & Gu, X. (2009). “SecureMR: A service integrity assurance framework for MapReduce”. Computer Security Applications Conference (pp. 73-82). ACM.
- [9] Zhao, S., Lo, V., & Dickey, C. (2005). “Result verification and trust-based scheduling in peer-to-peer grids”. the 5th IEEE International Conference on Peer-to-Peer Computing (pp. 31-38). IEEE.
- [10] Zhang, Kehuan, Xiaoyong Zhou, Yangyi Chen, XiaoFeng Wang, and Yaoping Ruan. “Sedic: privacy-aware data intensive computing on hybrid clouds.” In Proceedings of the 18th ACM conference on Computer and communications security, pp. 515-526. ACM, 2011.
- [11] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. “Airavat: Security and privacy for mapreduce”. In NSDI, pages 297–312. USENIX Association, 2010.
- [12] Carbunar, Bogdan, and Radu Sion. “Toward private joins on outsourced data.” Knowledge and Data Engineering, IEEE Transactions on, 24, no. 9 (2012): 1699-1710.

- [13]H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. “Executing SQL over encrypted data in the database-service-provider model”. In Proceedings of the ACM SIGMOD international conference on Management of data, pages 216–227. ACM Press, 2002.
- [14]Tingjian Ge and Stan Zdonik. 2007. “Answering aggregation queries in a secure system model”. In Proceedings of the 33rd international conference on Very large databases (VLDB '07). VLDB Endowment 519-530.
- [15]Jun Ge, Soma Chaudhuri, and Akhilesh Tyagi. “Control flow based obfuscation.” Proceedings of the 5th ACM workshop on Digital Rights Management. ACM, 2005.
- [16]Wang, C., Hill, J., Knight, J.C., and Davidson, J.W. “Protection of software-based survivability mechanisms”. In Proceedings of the 2001 conference on Dependable Systems and Networks
- [17]Collberg, C.S., Thomborson, C., “Watermarking, tamper-proofing, and obfuscation - tools for software protection”. Software Engineering, IEEE Transactions on, vol.28, no.8, pp.735,746, Aug 2002
- [18]“2013 Global Surveillance Disclosures”
http://en.wikipedia.org/wiki/2013_global_surveillance_disclosures
- [19] “Symja, a java computer algebra system” <https://code.google.com/p/symja/>
- [20]Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. “Soot-a Java Optimization Framework”. In Proceedings of CASCON 1999
- [21]Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. 2012. “On the (im)possibility of obfuscating programs”. J. ACM 59, 2, Article 6 (May 2012).
- [22]Christian Collberg, Clark Thomborson, and Douglas Low. “A taxonomy of obfuscating transformations”. Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, July 1997.
- [23]Stephen Drape, Anirban Majumdar, and Clark Thomborson. “Slicing aided design of obfuscating transforms”. In Proceedings of the 6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007). pp.1019-1024, IEEE, 11-13 July 2007.
- [24]Levent Ertaul and Suma Venkatesh. “Novel obfuscation algorithms for software security”. In Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP 2005), vol. 5., pp.209-215. 2005.
- [25]Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have

- been afraid to ask)". In Proceedings of 2010 IEEE Symposium on Security and Privacy (S&P 2010). pp.317-331. IEEE, 2010.
- [26]Shuo Yang, Ali R. Butt, Y. Charlie Hu, and Samuel P. Midkiff. "Lightweight monitoring of the progress of remotely executing computations". In Languages and Compilers for Parallel Computing, pp.319-333. Springer Berlin Heidelberg, 2006.
- [27]"Free ebooks - Project Gutenberg", <http://www.gutenberg.org>, accessed on Jan 4, 2015.
- [28]"Cloudinary", http://cloudinary.com/features#image_manipulation, accessed on Jan 4, 2015.
- [29]"Bioproximity", <http://www.bioproximity.com>, accessed on Jan 4, 2015.
- [30]"Euclid", <http://euclidanalytics.com>, accessed on Jan 4, 2015.
- [31]"Nearmap", <http://us.nearmap.com>, accessed on Jan 4, 2015.
- [32]"Hadoop4Images",
<https://github.com/Aishraj/Hadoop4Images/blob/master/Hadoop4Images/src/canny/CannyEdgeDetector.java>, accessed on Jan 4, 2015.
- [33]"AWS Case Study: Euclid", <http://aws.amazon.com/solutions/case-studies/euclid/>, accessed on Jan 04, 2015.
- [34]"AWS Case Study: nearmap",
<http://aws.amazon.com/solutions/case-studies/nearmap/>, accessed on Jan 04, 2015.
- [35]"AWS Case Study: Bioproximity",
<http://aws.amazon.com/solutions/case-studies/bioproximity/>, accessed on Jan 04, 2015.
- [36]"AWS marketplace: Cloudinary – Image Management in the Cloud",
<https://aws.amazon.com/marketplace/pp/B00DQC6HWE?sr=0-2>, accessed on Jan 04, 2015.
- [37]"SETI@home". <http://setiathome.ssl.berkeley.edu/>
- [38]"Amazon Elastic Compute Cloud (Amazon EC2)". <http://aws.amazon.com/ec2/>
- [39]"Windows Azure Compute".
<https://www.windowsazure.com/en-us/home/features/compute>
- [40]Andrew Colley. "Cheats wreak havoc on seti@home: participants". ZDNet, Australia, (2002).

- [41]David Molnar. “The seti@home problem”. E-Commerce, (2000).
- [42]“Cloud Security: Amazon’s EC2 serves up 'certified pre-owned' server images”.
<http://dvlabs.tippingpoint.com/blog/2011/04/11/cloud-security-amazons-ec2-serves-up-certified-pre-owned-server-images>
- [43]R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, “Design and implementation of a tcb-based integrity measurement architecture,” in Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13, ser. SSYM’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 16–16. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1251375.1251391> □
- [44]A. Ruan and A. Martin, “Tmr: Towards a trusted mapreduce infrastructure,” in Services (SERVICES), 2012 IEEE Eighth World Congress on, June 2012, pp. 141–148. □
- [45]D. P. Anderson, “Boinc: a system for public-resource computing and storage,” in Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on, Nov 2004, pp. 4–10. □
- [46]S. Goldwasser, S. Micali, and C. Rackoff, “The knowledge complexity of interactive proof systems,” SIAM Journal on Computing, vol. 18, no. 1, pp. 186–208, 1989. [Online]. Available: <http://dx.doi.org/10.1137/0218012> □
- [47]O. Goldreich, S. Micali, and A. Wigderson, “Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems,” J. ACM, vol. 38, no. 3, pp. 690–728, Jul. 1991. [Online]. Available: <http://doi.acm.org/10.1145/116825.116852> □
- [48]M. Bellare and O. Goldreich, “On defining proofs of knowledge,” in Advances in Cryptology CRYPTO 92, ser. Lecture Notes in Computer Science, E. Brickell, Ed. Springer Berlin Heidelberg, 1993, vol. 740, pp. 390–420. [Online]. Available: http://dx.doi.org/10.1007/3-540-48071-4_28 □
- [49]R. Gennaro, C. Gentry, and B. Parno, “Non-interactive verifiable computing: Outsourcing computation to untrusted workers,” in Advances in Cryptology CRYPTO 2010, ser. Lecture Notes in Computer Science, T. Rabin, Ed. Springer Berlin Heidelberg, 2010, vol. 6223, pp. 465–482. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14623-7_25 □
- [50]Sharath K. Udupa, Saumya K. Debray, and Matias Madou. “Deobfuscation: reverse engineering obfuscated code”. In Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 2005). IEEE, 2005.
- [51]Christian Collberg, Clark Thomborson, and Douglas Low. “Manufacturing cheap, resilient, and stealthy opaque constructs”. In Proceedings of 1998 ACM

- SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL1998). ACM, 1998.
- [52]Jens Palsberg, Sowmya Krishnaswamy, Minseok Kwon, Di Ma, Qiuyun Shao, and Yi Zhang. “Experience with software watermarking”. In Proceedings of 16th IEEE Annual Computer Security Applications Conference (ACSAC 2000). pp.308-316, IEEE, 2000.
 - [53]Anirban Majumdar and Clark Thomborson. “Manufacturing opaque predicates in distributed systems for code obfuscation”. In Proceedings of the 29th Australasian Computer Science Conference (ACSC 2006). pp.187-196, Australian Computer Society Inc., 2006.
 - [54]Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. “Impeding malware analysis using conditional code obfuscation”. In Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS 2008).
 - [55]Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. “Linear obfuscation to combat symbolic execution”. In Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS 2011). pp.210-226, Springer Berlin Heidelberg, 2011.
 - [56]Yongzhi Wang, and Jinpeng Wei. “VIAF: Verification-Based integrity assurance framework for MapReduce” In Cloud Computing (CLOUD), 2011 IEEE International Conference on, pp. 300-307. IEEE, 2011.
 - [57]Yongzhi Wang, Jinpeng Wei, and Mudhakar Srivatsa. “Result Integrity Check for MapReduce Computation on Hybrid Clouds”. In Cloud Computing (CLOUD), 2013 IEEE International Conference on, IEEE, 2013.
 - [58]Yongzhi Wang, Jinpeng Wei, and Mudhakar Srivatsa. “Cross Cloud MapReduce: A Result Integrity Check Framework on Hybrid Clouds”. International Journal of Cloud Computing (IJCC), Vol. 1, No. 1, 2013.
 - [59]Yongzhi Wang, Jinpeng Wei, Mudhakar Srivatsa, Yucong Duan, and Wencai Du. “IntegrityMR: Integrity assurance framework for big data analytics and management applications.” In Big Data, 2013 IEEE International Conference on, pp. 33-40. IEEE, 2013.
 - [60]Yongzhi Wang, Jinpeng Wei, Mudhakar Srivatsa. “Cross Cloud MapReduce: an Uncheatable MapReduce”. The 33rd IEEE Symposium on Security and Privacy, San Francisco, CA, May 20-23, 2012. Poster paper.
 - [61]Yongzhi Wang, Jinpeng Wei, Yucong Duan “Securing MapReduce Result Integrity via Verification-based Integrity Assurance Framework”. International Journal of Grid and Distributed Computing (IJGDC). Vol. 7, No. 6 (2014), pp.53-70.

- [62]Djemel Ziou and Salvatore Tabbone. “Edge detection techniques: an overview”. Pattern Recognition And Image Analysis, vol. 8, no. 4, pp. 537-559. 1998.
- [63]John Canny. “A Computational Approach to Edge Detection”. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol.PAMI-8, no.6, pp.679-698, IEEE, Nov. 1986.
- [64]Sebastian Schrittwieser, Stefan Katzenbeisser. “Code obfuscation against static and dynamic reverse engineering”. In Proceedings of the 13th International Conference on Information Hiding. Volume 6958 of LNCS. (2011), pp.270–284
- [65]Yongzhi Wang, Jinpeng Wei. “Towards protecting control flow confidentiality in cloud-based computation”. Elsevier Journal of Computers and Security (Computers & Security). Volume 52, pp. 106-127, July 2015. doi: 10.1016/j.cose.2015.04.005
- [66]Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Roykova, Amit Sahai and Brent Waters. “Candidate indistinguishability obfuscation and functional encryption for all circuits”. In Proceedings of 2013 IEEE 54th Annual Symposium on Foundations of Computer Science (FOCS 2014), pp.40-49. IEEE, 2013.
- [67]Belenkiy, Mira, Melissa Chase, C. Chris Erway, John Jannotti, Alptekin Küpçü, and Anna Lysyanskaya. “Incentivizing outsourced computation.” In Proceedings of the 3rd international workshop on Economics of networked systems, pp. 85-90. ACM, 2008.
- [68]Nix, Robert, and Murat Kantarcioglu. “Contractual agreement design for enforcing honesty in cloud outsourcing.” Decision and Game Theory for Security. Springer Berlin Heidelberg, 2012. 296-308.
- [69]Khouzani, M. H. R., Viet Pham, and Carlos Cid. “Incentive engineering for outsourced computation in the face of collusion.” Proceedings of WEIS. 2014.
- [70]G. W. Brown. “Iterative solutions of games by fictitious play.” In T.C. Koopmans, editor, Activity Analysis of Production and Allocation, pages 374–376. Wiley, New York, 1951. □
- [71]J. Robinson. “An iterative method of solving a game”. Annual Mathematics, 54:296–301, 1951. □
- [72]D. Fudenberg and D.K. Levine. “The Theory of Learning in Games”. MIT Press, Cambridge, MA, 1998. □
- [73]J. Hofbauer and K. Sygmond. “Evolutionary Games and Population Dynamics”. Cambridge University Press, 1998.
- [74]J. W. Weibull. “Evolutionary Game Theory”. MIT Press, Cambridge, MA, 1995. □

- [75]Alpcan, Tansu, and Tamer Basar. "A game theoretic analysis of intrusion detection in access control systems." *Decision and Control*, 2004. CDC. 43rd IEEE Conference on. Vol. 2. IEEE, 2004.
- [76]Nguyen, Kien C., Tansu Alpcan, and Tamer Basar. "Security games with incomplete information." *Communications*, 2009. ICC'09. IEEE International Conference on. IEEE, 2009.
- [77]Nguyen, Kien C., Tansu Alpcan, and Tamer Basar. "Fictitious Play with Time-invariant Frequency Update for Network Security." *CCA*. 2010.
- [78]Dabcevic, Kresimir, et al. "A fictitious play-based game-theoretical approach to alleviating jamming attacks for cognitive radios." *Acoustics, Speech and Signal Processing (ICASSP)*, 2014 IEEE International Conference on. IEEE, 2014.
- [79]Giffin, Jonathon T., Somesh Jha, and Barton P. Miller. "Detecting Manipulated Remote Call Streams." *USENIX Security Symposium*. 2002.
- [80]Gentry, C.; Halevi, S.; Raykova, M.; Wichs, D., "Outsourcing Private RAM Computation," *Foundations of Computer Science (FOCS)*, 2014 IEEE 55th Annual Symposium on , vol., no., pp.404,413, 18-21 Oct. 2014
- [81]Prabhanjan Ananth, Divya Gupta, Yuval Ishai, and Amit Sahai. 2014. Optimizing Obfuscation: Avoiding Barrington's Theorem. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 646-658.
- [82]Amit Sahai and Brent Waters. 2014. How to use indistinguishability obfuscation: deniable encryption, and more. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC '14)*. ACM, New York, NY, USA, 475-484.

VITA
YONGZHI WANG

2000-2004	B.A., Computer Science Xidian University Xian, China
2004-2007	M.S., Computer Science Xidian University Xian, China
2007-2010	Staff Software Engineer IBM (SPSS) Xian, China
2010-2015	Doctoral Candidate Florida International University Miami, Florida, USA

PUBLICATIONS AND PRESENTATIONS

Yongzhi Wang, Jinpeng Wei. “*Towards Protecting Control Flow Confidentiality on Cloud Based Computation*” Elsevier Journal of Computers and Security (Computers & Security). Volume 52, pp. 106-127, July 2015. Doi: 10.1016/j.cose.2015.04.005.

Yongzhi Wang, Jinpeng Wei, Yucong Duan. “*Securing MapReduce Result Integrity via Verification-based Integrity Assurance Framework*” International Journal of Grid and Distributed Computing (IJGDC), Vol. 7, No. 6, 2014, pages 53-77.

Yongzhi Wang, Jinpeng Wei, Mudhakar Srivatsa. “*Cross Cloud MapReduce: A Result Integrity Check Framework on Hybrid Clouds*” International Journal of Cloud Computing (IJCC), Vol. 1, No. 1, 2013, pages 26-39.

Yucong Duan, Keman Huang, Dan Chen, Yongzhi Wang, Ajay Kattepur, Wencai Du, “*Service Value Broker Patterns: An Empirical Collection and Analysis*”. International Journal of Networked and Distributed Computing (IJNDC), Vol. 2, No. 1, 2013, pages 54-69.

Yongzhi Wang, Zhenguo Ding “*Research on the Protocol Analysis Module Based on the TCP Splicing Method*”. Science Technology and Engineering (Chinese Journal). Vol. 7, no. 5, 2007, pages 782-785.

Yucong Duan, Nanjangud Narendra, Wencai Du, Yongzhi Wang, Nianjun Zhou, “*Exploring Cloud Service Brokering from an Interface Perspective*” 21th IEEE

International Conference on Web Services (IEEE ICWS 2014), June 27 – July 2, 2014, Alaska, USA.

Yucong Duan, Yongzhi Wang, Jinpeng Wei, Ajay Kattapur and Wencai Du "*Value Added Modeling and Analysis on Service Value Brokerage*". The 1st International Workshop on Cloud Service Brokerage (CSB 2013), co-located with the 11th International Conference on Service Oriented Computing (ICSOC 2013), Berlin, Germany, December 2-5, 2013, pages 209-222.

Yongzhi Wang, Jinpeng Wei, Mudhakar Srivatsa, Yucong Duan, and Wencai Du. "*IntegrityMR: Integrity Assurance Framework for Big Data Analytics and Management Applications*". Proceedings of the IEEE International Conference on Big Data (IEEE Big Data 2013) Knowledge Management and Big Data Analytics Workshop, October 6-9, 2013, pages 17-24.

Yucong Duan, Yongzhi Wang, Jinpeng Wei, Ajay Kattapur, and Wencai Du. "*Constructing E-Tourism Platform Based on Service Value Broker: A Knowledge Management Perspective*". Proceedings of the IEEE International Conference on Big Data (IEEE Big Data 2013) Knowledge Management and Big Data Analytics Workshop, October 6-9, 2013, pages 33-40.

Yongzhi Wang, Jinpeng Wei, and Mudhakar Srivatsa. "*Result Integrity Check for MapReduce Computation on Hybrid Clouds*". Proceedings of the 6th IEEE International Conference on Cloud Computing (IEEE CLOUD 2013), IEEE Computer Society, Santa Clara, CA, June 27-July 2, 2013, pages 847-854.

Yongzhi Wang, Jinpeng Wei, Mudhakar Srivatsa. "*Cross Cloud MapReduce: an Uncheatable MapReduce*". The 33rd IEEE Symposium on Security and Privacy (IEEE S&P 2012), San Francisco, CA, May 20-23, 2012. Poster paper.

Yongzhi Wang, Jinpeng Wei. "*VIAF: Verification-based Integrity Assurance Framework for MapReduce*". Proceedings of the fourth IEEE International Conference on Cloud Computing (IEEE CLOUD 2011), IEEE Computer Society, Washington DC, July 4-9, 2011, pages 300-307.