Practical Verifiable Computation –A MapReduce Case Study

Yongzhi Wang, Yulong Shen, and Xiaohong Jiang, *Senior Member, IEEE*

***Abstract*— Public cloud vendors have been offering a variety of big data computing services on their clouds. However, runtime integrity is one of the major security concerns that hinder the wide adoption of those services. In this paper, we focus on MapReduce, a popular big data computing framework, and propose the runtime integrity audition (RIA), a solution that remotely verifies the runtime integrity of MapReduce applications. RIA records the runtime variable values of the MapReduce application on the public cloud and checks those values against the application's code on the private cloud. By doing so, RIA protects the runtime integrity of MapReduce applications. Based on the idea of RIA, we developed a prototype system, called MR Auditor, and tested its applicability and performance with several Hadoop applications. Our experimental results showed that MR Auditor is a general tool that can efficiently audit the runtime integrity of all the MapReduce applications that we tested. In addition, MR Auditor incurs a moderate performance overhead. For example, when verifying the Word Count application, a proper parameter setting of MR Auditor incurs 1% of extra execution time on the public cloud and 14% of extra execution time on the private cloud.**

***Index Terms*— Runtime integrity, remote verification, cloud computing, mapreduce.**

I. Introduction

MAPREDUCE applications usually will be executed on the cluster consisting of a number of hosts. For many users who cannot afford a dedicated cluster, public clouds offer a viable solution, in which users can rent clusters on demand and only pay for the rented resources. However, security breaches [1] and vulnerabilities [2], [3] frequently reported by mass media and researchers keep on reminding us that protecting the security of public cloud is not so easy. Among multiple security problems, runtime integrity is one of the most critical issues that hinder the wide adoption of public clouds. For MapReduce applications, when the computation job is outsourced to the untrusted public cloud, ensuring the computation correctness is still a challenge that needs to be addressed.

Existing works either do not offer data integrity guarantees or incur high performance overheads. For instance, program anomaly detection systems build models on the program control flow [4]–[6] or system call parameter values patterns [7], [8]. They can protect the control flow integrity of remote computation by remotely checking the execution trace against the built model. However, this class of work cannot detect the tampering of variables that are not involved in the control flow, thus cannot protect the integrity of data computation. Task assignment-based solutions [9]–[11] leverage task replication, task verification and trust management to achieve the optimal trade-off between the result integrity and the performance overhead. Such a class of works needs to perform case-by-case system modifications and is effective only when the number of tasks is large. Proof-based verifiable computation [12]–[14] enable programs to generate *proofs* while they are executed on the remote host. By employing cryptographic schemes and computing complexity theorems, the local host is able to verify the correctness of the remote computation. Such a class of works checks the execution of each statement, thus providing a thorough integrity protection. However, as of this writing, existing works still incur a significant performance overhead.

Under such a situation, a practical solution is needed to protect the result integrity of big data computations performed on the public cloud. In this paper, we use MapReduce as a study case and propose *runtime integrity audition*, (or *RIA* for short), a hybrid clouds-based remote verification method. RIA can be used to verify the runtime integrity of MapReduce applications executed on the untrusted public cloud. RIA transforms the MapReduce program by inserting logging statements, so that while the program is executed on the public cloud, *execution logs* are generated to reflect runtime variable values. On the private cloud, by performing the *integrity audition* based on the execution logs, RIA is able to verify the runtime integrity of each MapReduce phase. As a result, RIA protects the runtime integrity of MapReduce applications.

The integrity audition includes the *input audition* and the *execution audition*. In each phase, the input audition verifies

Manuscript received April 30, 2017; revised September 18, 2017 and December 15, 2017; accepted December 17, 2017. Date of publication December 29, 2017; date of current version February 7, 2018. This work was supported in part by the National Natural Science Foundation of China under Grant 61602364, in part by the Natural Science Foundation of Shaanxi Province under Grant 2017JM6083, in part by the Open Fund of the Chinese Key Laboratory of the Grain Information Processing and Control under Grant KFJJ-2015-202, and in part by the Natural Science Foundation of China under Grant 61373173, Grant U1536202, and Grant 61602365. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Mauro Conti. *(Corresponding author: Yongzhi Wang.)*

Y. Wang is with the School of Computer Science and Technology, Xidian University, Xi'an 710071, China, and also with the Key Laboratory of Grain Information Processing and Control, Henan University of Technology, Ministry of Education, Zhengzhou 450066, China (e-mail: yzwang@xidian.edu.cn).

Y. Shen is with the School of Computer Science and Technology, Xidian University, Xi'an 710071, China (e-mail: ylshen@mail.xidian.edu.cn).

X. Jiang is with the School of Systems Information Science, Future University Hakodate, Hakodate 041-8655, Japan, and also with the School of Computer Science and Technology, Xidian University, Xi'an 710071, China (e-mail: jiang@fun.ac.jp).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2017.2787993

the input integrity. The *execution audition* verifies the integrity of task executions. In the input audition, a *CBF-based set compare* technique is proposed to speed up the input data comparison. In the *execution audition*, we introduce the *function execution audition* (*FEA*) technique to audit the integrity of each function execution. FEA removes the cohesion between the function caller and the function callee, thus enabling users to audition function executions in a sampling manner. To support the sampling-based FEA, RIA introduces the *log retrieve protocol* to keep all the execution logs on the public cloud, and enable the private cloud to safely retrieve the execution logs on demand.

On top of the basic solution, we proposed extended solutions to address two related security problems, namely, how to protect the the integrity audition on the untrusted private cloud, and how to protect the data confidentiality on the public cloud.

Based on the design of RIA, we implemented a prototype system, called *MR Auditor*, to protect the result integrity of Hadoop applications. The experimental results showed that MR Auditor is a generalized tool to protect Hadoop applications. During the program execution and audition, it introduced a modest performance overhead, however incurred non-neglectable storage and transmission loads of execution logs. For example, to process 1GB of texts with Word Count application, the transformed program introduced 1% of extra execution time but generated 22 GB of uncompressed execution logs on the public cloud. During the audition of the above application, MR Auditor introduced 14% of extra execution time on the private cloud and 3.6 GB of logs transmitted between clouds with a proper parameter setting. (see Section VI-A3)

Compared to the program anomaly detection solutions [4]–[8], RIA can detect not only the control flow tampering, but also the data tampering. Compared to the task assignment-based solution [9]–[11], RIA does not have restrictions on the task number and is more general. Compared to the proof-based verification computation [12]–[14], RIA has shown a much smaller performance overhead (see Section VI-C).

We denote that the contribution of RIA is not only restricted to MapReduce computations. The execution log insertion and the execution audition proposed in Section III-C and Section III-E are applicable to general computations.

The paper is organized as follows. We introduce the preliminary knowledge of MapReduce and describe the security model in Section II. We introduce the design details of RIA in Section III. We perform the security analysis of RIA in Section IV. We discuss two extended solutions to address two related security problems in Section V. We present the experiments and evaluation results in Section VI. We discuss the related works in Section VII. We conclude the paper and discuss the future work in Section VIII.

II. Preliminaries and Security Model

A. MapReduce

MapReduce [15] 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,

```
1  void map(int key, int value, Context context){
2      logInput(key, value);
3      int counter = value;
4      int sum = 0;
5      int fib1 = key;
6      int fib2 = fib1+1;
7      while(logBranch(counter)&&counter>0){
8          sum = fib1+fib2;
9          fib1 = fib2;
10         fib2 = sum;
11         counter−−;
12     }
13     logPreInvoke(sum);
14     boolean shouldOutput = isPrime(sum);
15     logPostInvoke(shouldOutput);
16     if(logBranch(shouldOutput)&&shouldOutput){
17         logOutput(key, sum);
18         context.write(key, sum);
19     }
20 }
```

Listing 1. The PWL code of the sample `map` function.

each computation request issued by the user is called a *job*. Each job is usually broken down into multiple *tasks*. The traditional architecture of MapReduce consists of one *master* and multiple *workers*. The master controls the entire computation, while workers contribute computation resources to execute tasks assigned by the master.

Each MapReduce job consists of three consecutive phases: the *map phase*, the *shuffle phase* and the *reduce phase*. In each phase, data are stored and processed in records, where each record is in the format of <*key, value*> pairs. The computation infrastructure is managed by the MapReduce framework. The developer only needs to implement the `map`, `reduce`, and `partition` function. The MapReduce framework will generate the map and the reduce tasks based on the implemented functions. Listing 1 shows the typical processing routine of a MapReduce function: read the data in the format of `<key, value>` pair, process the data, and output the data in the format of `<key, value>` pair. The types of `key` and `value` can be any object instead of only being integers. Notice in Listing 1, the original map function does not include the code in highlights. The highlighted code is the logging statement inserted by RIA (see Section III-C).

B. Security Model

In our paper, we focus on protecting the *result integrity* of the MapReduce applications. For a MapReduce application, the result integrity is preserved if and only if its result is correct.

In the hybrid clouds architecture that is employed in this paper, we assume the public cloud is not trusted. Malicious public cloud employees or hackers who compromise the public cloud might want to gain profits by injecting errors into application results. For instance, when a recommendation system is outsourced to the public cloud, a malicious company might want to increase the recommendation frequency of its product by compromising the program. Different from the lazy cheater model, where the malicious public cloud commits cheats only to save computation energy, in this paper, we use a stronger adversary model, in which the malicious public cloud

is willing to perform extra computations if this can help to pass the check. For example, a compromised public cloud can first compute a job faithfully. After sending the correct execution logs to the integrity audition, it will tamper the job results and feed the tampered results as the input to the next job.

In our basic solution, we assume that the private cloud is trusted, given the fact that private clouds are usually deployed in the user's organization, or on a dedicated infrastructure protected by large cloud vendors. For the cases that the private cloud can be compromised, we propose two alternative solutions (in Section V-A) to protect the integrity audition performed on the private cloud.

III. System Design

In this section, we discuss the design of RIA. We firstly define the MapReduce model and introduce our design motivation. After that, we give a design overview, followed by the detailed description.

A. The MapReduce Model and the Design Intuition

Without losing generality, a MapReduce application consists of a sequence of map or reduce phases, executed in an interleaved sequence. Since our method can be directly applied to both the map task and the reduce task, we do not distinguish the map and the reduce phase in the following discussions. We define the MapReduce application model as follows. The application input I is originally stored on the private cloud and then transmitted to the public cloud. A MapReduce application consists of n phases, marked as p_1, p_2,..., p_n, respectively. For each phase p_i, it consists of q_i tasks, marked as t_{i1}, t_{i2},..., t_{iq_i}. The input and the output of task t_{ij} are I_{ij} and O_{ij}, respectively. The output of the last job is the output of the entire application, marked as O. O will be sent back to the private cloud after the entire application completes.

The high level idea of RIA is as follows. For a MapReduce application consisting of n phases, if we have the input data set, the execution logs and the output data set of each phase, we can audit the result integrity of that application in a chain. Since the application input I is maintained in the private cloud, by using it as a trusted anchor, we can check the integrity of the input set in phase 1. When the input set in phase 1 passes the check, we can use it to verify the execution integrity of the tasks in phase 1. When the execution integrity of phase 1 passes the check, we can use the output of phase 1 to check the input set in phase 2, and so on... By auditing the runtime integrity of each phase sequentially, we are able to verify the integrity of the last phase's output set, i.e., O. If the audition fails at any point in the chain, we simply reject O; otherwise, O is accepted.

B. System Overview

1) System Architecture: The runtime integrity audition (RIA) is performed on a hybrid clouds environment, where the private cloud is trusted and the public cloud is untrusted (extended solutions are provided in Section V-A to address the scenario where the private cloud is untrusted). As shown in Fig. 1, on the private cloud, RIA transforms the original programs by inserting logging statements. The resulting

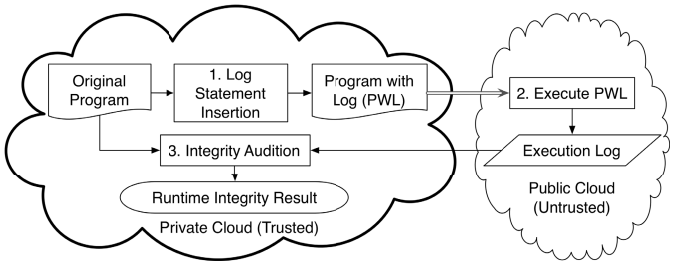


Fig. 1. The architecture of Runtime Integrity Audition.

Algorithm 1 Integrity Audition

Require: The application model is defined in Section III-A. t_{ij}: task j in phase i. L_{ij}: the execution log of task t_{ij}.

```
1: inputBaseLine ← I
2: for i = 1 to n do
3:   for j = 1 to q_i do
4:     L_ij ← retrieveExecutionLog(t_ij)
5:   end for
6:   phaseInput ← extractInput({L_i1, ..., L_iq_i})
7:   inputAudition(phaseInput, inputBaseLine)
8:   for j = 1 to q_i do
9:     executionAudition(L_ij)
10:  end for
11:  inputBaseline ← extractOutput({L_i1, ..., L_iq_i})
12: end for
13: inputAudition(O, inputBaseline)
```

program is called the *Program with Log*, or *PWL* for short (in step 1). The PWLs are then sent to the public cloud to execute (in step 2). During the execution, the PWLs generate *execution logs*, which reflect the statements execution sequence and the runtime variable values of the program. When the execution completes, the execution logs will be transferred back to the private cloud. On the private cloud, integrity audition is performed based on the execution logs and the original program. (in step 3). The integrity audition is performed on a MapReduce phase basis. In each phase, it performs the *input audition* and the *execution audition*.

The integrity audition is described in Algorithm 1. The algorithm uses the application input I as the trusted anchor and audits the integrity of the phase inputs and task executions phase by phase. If a phase passes the audition, its output becomes the baseline and will be used to audit the integrity of its succeeding phase. If the output of the last phase passes the audition, it will be used to audit the application's output O. During the execution of the algorithm, any inconsistency indicates a violation of the runtime integrity.

C. The Execution Log Generation

We transform the MapReduce application by inserting logging statements to its original program, so that the execution logs will be generated while the program is executed on the public cloud. We generate four types of logs: the *input log*, the *output log*, the *branch log* and the *invoke log*.

The input log records the input data read from the program. In MapReduce, task input is read in as the parameters of the `map` or `reduce` function, in the format

TABLE I
THE EXECUTION LOGS OF EXECUTING `Map(3, 5, Context)`

log file name	log content
input log of the current task	$< 3, 5 >$
output log of the current task	$< 3, 47 >$
branch log of function `map`	5, 4, 3, 2, 1, 0, true
invoke log of function `isPrime`	$< pre : \{47\}, post : true >$

of `<key, value>` pair. Therefore, the logging statements are inserted at the beginning of the `map` or `reduce` function, recording values of the `<key, value>` pairs. The output log records the data written to the DFS or the local storage. In MapReduce, task output is written with an invocation of `context.write(outKey, outValue)`, where `<outKey, outValue>` is one record of the task output. Therefore, the values of `<outKey, outValue>` in function `map` or `reduce` will be logged. The branch log records the variable values involved in the condition of the branch statement. The invoke log records the parameters and the return value of each function call. For each function call, statements are inserted before and after it, respectively. The one inserted before the function invocation is called the *pre-invoke log*, which records the values of parameters in the function invocation. The one inserted after the function invocation is called the *post-invoke log*, which records the return value.[1]

Listing 1 indicates a PWL program of a `map` function. The code in highlights is the logging statement inserted by RIA. Note that Listing 1 only shows the equivalent code after the transformation. In the real implementation, the program code is analyzed and transformed with *Jimple*, a three-address intermediate representation introduced in *Soot* framework. In each phase, the input and output logs are stored in an input log file and an output log file, respectively; the branch logs reflecting the execution of the same function are stored in the same branch log file; the invoke logs for invoking the same function are stored in the same invoke log file. As a concrete example, Table I shows the execution logs when executing `map(3, 5, context)`.

The execution logs will be sent to the private cloud to perform the integrity audition, including the input audition and the execution audition, which are elaborated next.

D. The Input Audition

1) The Basic Algorithm: For each phase i, the input audition constructs the *input data set* I_i from the input log of phase i and the *output data set* O_{i-1} from the output log of phase $i-1$ (or the application input I, if $i = 0$). Since the output records of a task in phase $i-1$ can be distributed to multiple tasks in phase i, I_i should contain the input records from all the tasks in phase i, and O_{i-1} should contain the output records from all the tasks in phase $i-1$. After that, the audition compares I_i against O_{i-1}. The input integrity of phase i is assured when the following tests are passed.

$$|I_i| = |O_{i-1}| \tag{1}$$

$$\forall e \in O_{i-1}, \quad COUNT(e, I_i) = COUNT(e, O_{i-1}) \tag{2}$$

The function $COUNT(e, S)$ returns the number of e in set S. Test (1) ensures that the sizes of I_i and O_{i-1} are equivalent. Test (2) ensures that if any record e appears in O_{i-1}, the number of its occurrence in O_{i-1} is equal to its counterpart in I_i. The fulfillment of the two tests ensures that $O_{i-1} = I_i$.

2) Probabilistic Input Audition: The number of input/output records in each phase is usually significantly large. We introduce the *Counting Bloom Filter (CBF) based set compare* to speed up the input audition. Specifically, we construct a Counting Bloom Filter $CBFI_i$ based on I_i and a Counting Bloom Filter $CBFO_{i-1}$ based on O_{i-1}. While constructing $CBFI_i$ and $CBFO_{i-1}$, we test the equality of $|I_i|$ and $|O_{i-1}|$, i.e., the test (1). If test (1) is passed, we perform test (2). For each record r from set O_{i-1}, with a certain probability t_r, we query $CBFI_i$ and $CBFO_{i-1}$ for the occurrence of records with the same value and test their equality. The inequality occurred on any selected record indicates the violation of the input data integrity in phase i. We call the probability t_r as the *CBF test probability*. We defer the security analysis of the probabilistic input audition in Section IV-A.

E. The Execution Audition

Having verified the correctness of the input log for a phase, the next step is to check whether the tasks in that phase are executed faithfully. To achieve this goal, we introduce the *function execution audition* (*FEA*) algorithm to audit the execution integrity of a given function invocation. The functions to be audited include the `map` and `reduce` function and other functions invoked direct or indirectly by the `map` or `reduce` function. The FEA algorithm removes the function invocation nesting, making sure that the audition of one function execution will not be interrupted by another function call invoked inside of that execution. As a result, each function execution can be audited independently. Directly applying the FEA algorithm on each function will incur a significant performance overhead. In addition, it requires all the execution logs to be transmitted from the public cloud to the private cloud. Such a cost would be significantly higher than executing the application on the private cloud directly. To avoid such a case, we introduce the *log retrieve protocol*, which enables the private cloud to safely retrieve the execution logs on demand. With the support of the log retrieve protocol, we introduce the *probabilistic execution audition* to perform the FEA in a sampling manner.

1) The Function Execution Audition: In the function execution audition algorithm, we generate the *control flow graph (CFG)* of that function and simulate the execution based on its CFG and its execution logs. With the branch log, the simulation can derive which branch was taken while executed on the public cloud, and thus can reproduce the runtime control flow. During the simulation, the mathematical relationships among runtime variable values can be derived

[1]Our implementation targets on Java language. The parameters of the function call are always unchanged after the function call. Therefore we do not log the parameter values after the invocation.

based on the semantics of each simulated statement. The derived mathematical relationships are stored in a *constraint set*, marked as C. The runtime variable values in the execution log will be checked against C during the simulation. Any violation of the constraints indicates the runtime integrity is violated on the public cloud.

During the simulation, for each statement s that is traversed, C will be updated based on the semantics of s. If s has a corresponding log record r, it either will be used to update C or will be checked against C, depending on the semantics of s. The FEA technique is shown in Algorithm 2. We explain the algorithm as follows case by case:

Case 1: If s is an input statement, i.e., the entry point of a `map` or `reduce` function, its corresponding log record r is an input log record. The integrity of r is verified by the input audition of the current phase, we therefore trust it and use it in the constraint set. Thus, r is converted into a constraint and added to C.

Case 2: If s is an output statement, i.e., it is an invocation of the function `context.write`, its corresponding log record r is an output log record. We need to verify the integrity of such a log record. Thus, r is checked against the current C.

Case 3: If s is a branch statement, i.e., it is in a form of `if(condition)`, its corresponding log record r is a branch log record. We first verify the integrity of the log record r by converting r to a constraint and check it against the current C. After that, the condition is evaluated according to the values in r, deciding the runtime branch that is simulated. Based on the condition and its decided branch, we generate a new constraint and add it to C. Since r is verified, the runtime branch in the simulation should be consistent to the execution on the public cloud.

Case 4: If s is a function invocation statement, and r_{pre} and r_{post} are a pre-invoke and a post-invoke log record corresponding to the function invocation, we first check r_{pre} against C to ensure the integrity of the parameters passed to the function. Then, we convert r_{post} into a constraint and add it into C. The idea of this design is that, we temporarily trust the integrity of r_{post} and use it to verify the integrity of the execution after the invocation statement. We defer the verification of r_{post} to the time when the invoked function execution is audited. To achieve that, we need to maintain a connection between the caller and the callee, which is described in case 5 and 6.

Case 5: If s is the entry point of a function that is neither `map` nor `reduce`, r_{pre} is the corresponding pre-invoke log record, we initialize C as an empty set, convert r_{pre} into a constraint and add it to C. Since r_{pre} is verified in case 4), we can trust its integrity and use it to audit the current function execution.

Case 6: If s is a return statement of a function that is neither `map` nor `reduce`, r_{post} is its corresponding post-invoke log record, we check r_{post} against C. Checking the integrity of r_{post} makes its caller function ensured that the post-invoke log is trusted. It can be used to audit the integrity of the remaining executions of the caller function.

Case 7: If s is an assignment statement where the right hand side is primitive operations on variables, no log record corresponds to such a statement. However, C has to be updated due to the possible change of variable values.

(a) If s is in the format of $y = \lambda(\cdots)$, where λ is primitive operations on variables that do not include y, the value

Algorithm 2 Function_Execution_Audition (Function f, ExecutionLog log)

Require: f is the function to be audited, log contains the execution logs corresponding to the execution of function f.

```
1: C ← {∅}
2: s ← f.nextStatement()
3: while s! = End_Of_Function do
4:   switch (s)
5:   case 1:
6:     (s is the entry point of function map/reduce)
7:     r ← log.getInputLogRecord()
8:     C ← C ∪ r.toConstraint()
9:   case 2:
10:    (s is an invocation of function context.write)
11:    r ← log.getOutputLogRecord()
12:    Check(C, r.toConstraint())
13:  case 3:
14:    (s is a branch statement)
15:    r ← log.getBranchLogRecord()
16:    check(C, r.toConstraint())
17:    branch ← evaluate(s, r)
18:    c ← s.getCondition(branch)
19:    C ← C ∪ c.toConstraint()
20:  case 4:
21:    (s is an invocation of other functions)
22:    r_pre ← log.getPreInvokeLogRecord()
23:    r_post ← log.getPostInvokeLogRecord()
24:    check(C, r_pre.toConstraint())
25:    C ← C ∪ r_post.toConstraint()
26:  case 5:
27:    (s is the entry point of a non-map/reduce function)
28:    r_pre ← log.getPreInvokeLogRecord()
29:    C ← C ∪ r_pre.toConstraint()
30:  case 6:
31:    (s is a return statement of a non-map/reduce function)
32:    r_post ← log.getPostInvokeLogRecord()
33:    check(C, r_post.toConstraint()
34:  case 7:
35:    (s is an assignment statement)
36:    if s ∼ y = λ(X), y ∉ X then # case 7(a)
37:      Def_y ← C.getDefinition(y)
38:      C ← C − {y = Def_y}
39:      C ← C ∪ s.toConstraint()
40:    else if s ∼ y = λ(X, y) then # case 7(b)
41:      y_old ← λ_y^{-1}(X, y)
42:      C.replace(y, y_old)
43:    end if
44:  end switch
45:  s ← f.nextStatement()
46: end while
```

TABLE II
THE EXECUTION AUDITION DETAILS OF INVOKING `Map(3, 5, Context)`

No.	Input	Action	Constraints [1]
1	in:{$key = 3, value = 5$}	C1	$k = 3 \wedge v = 5$
2	stmt:{counter = value;}	C7(c)	$k = 3 \wedge v = 5 \wedge c = v$
3	stmt:{sum = 0;}	C7(c)	$k = 3 \wedge v = 5 \wedge c = v \wedge s = 0$
4	stmt:{fib1 = key;}	C7(c)	$k = 3 \wedge v = 5 \wedge c = v \wedge s = 0 \wedge f1 = k$
5	stmt:{fib2 = fib1+1;}	C7(c)	$k = 3 \wedge v = 5 \wedge c = v \wedge s = 0 \wedge f1 = k \wedge f2 = f1 + 1$
6	bra:{$counter = 5$}, stmt:{$if(counter > 0)$}	C3	$k = 3 \wedge v = 5 \wedge c = v \wedge s = 0 \wedge f1 = k \wedge f2 = f1 + 1 \wedge c > 0$
7	stmt:{$sum = fib1 + fib2$;}	C7(a)	$k = 3 \wedge v = 5 \wedge c = v \wedge f1 = k \wedge f2 = f1 + 1 \wedge c > 0 \wedge s = f1 + f2$
8	stmt:{$fib1 = fib2$;}	C7(a)	$k = 3 \wedge v = 5 \wedge c = v \wedge f2 = k + 1 \wedge c > 0 \wedge s = k + f2 \wedge f1 = f2$
9	stmt:{$fib2 = sum$;}	C7(a)	$k = 3 \wedge v = 5 \wedge c = v \wedge c > 0 \wedge s = 2k + 1 \wedge f1 = k + 1 \wedge f2 = s$
10	stmt:{$counter = counter - 1$;}	C7(b)	$k = 3 \wedge v = 5 \wedge c + 1 = v \wedge c + 1 > 0 \wedge s = 2k + 1 \wedge f1 = k + 1 \wedge f2 = s$
11	bra:{$counter = 4$}, stmt:{$if(counter > 0)$}	C3	$k = 3 \wedge v = 5 \wedge c + 1 = v \wedge s = 2k + 1 \wedge f1 = k + 1 \wedge f2 = s \wedge c > 0$
12	stmt:{$sum = fib1 + fib2$;}	C7(a)	$k = 3 \wedge v = 5 \wedge c + 1 = v \wedge f1 = k + 1 \wedge f2 = 2k + 1 \wedge c > 0 \wedge s = f1 + f2$
13	stmt:{$fib1 = fib2$;}	C7(a)	$k = 3 \wedge v = 5 \wedge c + 1 = v \wedge f2 = 2k + 1 \wedge c > 0 \wedge s = k + 1 + f2 \wedge f1 = f2$
14	stmt:{$fib2 = sum$;}	C7(a)	$k = 3 \wedge v = 5 \wedge c + 1 = v \wedge c > 0 \wedge s = 3k + 2 \wedge f1 = 2k + 1 \wedge f2 = s$
15	stmt:{$counter = counter - 1$;}	C7(b)	$k = 3 \wedge v = 5 \wedge c + 2 = v \wedge c + 1 > 0 \wedge s = 3k + 2 \wedge f1 = 2k + 1 \wedge f2 = s$
16	bra:{$counter = 3$}, stmt:{$if(counter > 0)$}	C3	$k = 3 \wedge v = 5 \wedge c + 2 = v \wedge s = 3k + 2 \wedge f1 = 2k + 1 \wedge f2 = s \wedge c > 0$
17	stmt:{$sum = fib1 + fib2$;}	C7(a)	$k = 3 \wedge v = 5 \wedge c + 2 = v \wedge f1 = 2k + 1 \wedge f2 = 3k + 2 \wedge c > 0 \wedge s = f1 + f2$
18	stmt:{$fib1 = fib2$;}	C7(a)	$k = 3 \wedge v = 5 \wedge c + 2 = v \wedge f2 = 3k + 2 \wedge c > 0 \wedge s = 2k + 1 + f2 \wedge f1 = f2$
19	stmt:{$fib2 = sum$;}	C7(a)	$k = 3 \wedge v = 5 \wedge c + 2 = v \wedge c > 0 \wedge s = 5k + 3 \wedge f1 = 3k + 2 \wedge f2 = s$
20	stmt:{$counter = counter - 1$;}	C7(b)	$k = 3 \wedge v = 5 \wedge c + 3 = v \wedge c + 1 > 0 \wedge s = 5k + 3 \wedge f1 = 3k + 2 \wedge f2 = s$
21	bra:{counter=2}, stmt:{$if(counter > 0)$}	C3	$k = 3 \wedge v = 5 \wedge c + 3 = v \wedge s = 5k + 3 \wedge f1 = 3k + 2 \wedge f2 = s \wedge c > 0$
22	stmt:{$sum = fib1 + fib2$;}	C7(a)	$k = 3 \wedge v = 5 \wedge c + 3 = v \wedge f1 = 3k + 2 \wedge f2 = 5k + 3 \wedge c > 0 \wedge s = f1 + f2$
23	stmt:{$fib1 = fib2$;}	C7(a)	$k = 3 \wedge v = 5 \wedge c + 3 = v \wedge f2 = 5k + 3 \wedge c > 0 \wedge s = 3k + 2 + f2 \wedge f1 = f2$
24	stmt:{$fib2 = sum$;}	C7(a)	$k = 3 \wedge v = 5 \wedge c + 3 = v \wedge c > 0 \wedge s = 8k + 5 \wedge f1 = 5k + 3 \wedge f2 = s$
25	stmt:{$counter = counter - 1$;}	C7(b)	$k = 3 \wedge v = 5 \wedge c + 4 = v \wedge c + 1 > 0 \wedge s = 8k + 5 \wedge f1 = 5k + 3 \wedge f2 = s$
26	bra:{$counter = 1$}, stmt:{$if(counter > 0)$}	C3	$k = 3 \wedge v = 5 \wedge c + 4 = v \wedge s = 8k + 5 \wedge f1 = 5k + 3 \wedge f2 = s \wedge c > 0$
27	stmt:{$sum = fib1 + fib2$;}	C7(a)	$k = 3 \wedge v = 5 \wedge c + 4 = v \wedge f1 = 5k + 3 \wedge f2 = 8k + 5 \wedge c > 0 \wedge s = f1 + f2$
28	stmt:{$fib1 = fib2$;}	C7(a)	$k = 3 \wedge v = 5 \wedge c + 4 = v \wedge f2 = 8k + 5 \wedge c > 0 \wedge s = 5k + 3 + f2 \wedge f1 = f2$
29	stmt:{$fib2 = sum$;}	C7(a)	$k = 3 \wedge v = 5 \wedge c + 4 = v \wedge c > 0 \wedge s = 13k + 8 \wedge f1 = 8k + 5 \wedge f2 = s$
30	stmt:{$counter = counter - 1$;}	C7(b)	$k = 3 \wedge v = 5 \wedge c + 5 = v \wedge c + 1 > 0 \wedge s = 13k + 8 \wedge f1 = 8k + 5 \wedge f2 = s$
31	bra:{counter=0}, stmt:{$if(counter > 0)$}	C3	$k = 3 \wedge v = 5 \wedge c + 5 = v \wedge s = 13k + 8 \wedge f1 = 8k + 5 \wedge f2 = s \wedge c <= 0$
32	invoke:{pre: sum=47, post: {shouldOutput=true, sum=47}}	C4	$k = 3 \wedge v = 5 \wedge c + 5 = v \wedge f1 = 8k + 5 \wedge f2 = 13k + 8 \wedge c <= 0 \wedge so = 1 \wedge s = 47$
33	bra:{$shouldOutput = true$}, stmt:{$if(shouldOutput)$}	C3	$k = 3 \wedge v = 5 \wedge c + 5 = v \wedge f1 = 8k + 5 \wedge f2 = 13k + 8 \wedge c <= 0 \wedge s = 47 \wedge so = 1$
34	out:{key=3, value=47}	C2	$k = 3 \wedge v = 5 \wedge c + 5 = v \wedge f1 = 8k + 5 \wedge f2 = 13k + 8 \wedge c <= 0 \wedge s = 47 \wedge so = 1$

[1] Variables used in constraints are abbreviated from the original program code. The correspondence are as follows: $k : key; v : value; c : counter; s : sum; f1 : fib1; f2 : fib2; shouldOutput : so$.

of y will be changed after the execution of s. Therefore, any constraints in C pertaining to y will not hold. Thus the algorithm replaces those constraints with the equality constraints generated from the statement s.

(b) If s is in the format of $y = \lambda(\cdots, y)$, where the value of y is updated after s is executed, the algorithm will first solve the "reverse expression of y", i.e., the expression that represents the LHS y with the RHS y. Then, it will replace all the y in C with the reverse expression of y. For instance, if s is `y=y+x`, for each constraint containing y in C, it will replace y with $y - x$. It is because at this moment, the old y is not available. The existing relations containing old y needs to be replaced with an expression using new y. If the operation λ on y is irreversible, we only remove constraints that contain y from the $constraintSet$.

As a concrete example, Table II shows the details of the FEA on executing `map(3, 5, context)`. The implementation of function `map` is shown in Listing 1. The execution log is listed in Table I. The second column in the table tracks the statement to be simulated (started with *stmt*), along with the corresponding log records, including input logs (*in*), output logs (*out*), invoke logs (*invoke*), branch logs (*bra*), etc. Based on the input, the audition determines the case that applies and performs the corresponding actions, shown in the third column. The fourth column indicates the resulting constraints after the action.

2) The Log Retrieve Protocol: Our study showed that, in the four types of logs generated on the public cloud, the invoke log and the branch log took the majority portion. For instance, in the 22 GB of logs generated in the Word Count application in Table V, the invoke logs took 16 GB and the branch logs took 3.4 GB. We introduce the *Log Retrieve Protocol*, a commitment-based protocol, to enable the private cloud safely retrieving parts of logs on demand. The protocol guarantees that the retrieved logs are consistent to the previous commitment made by the public cloud, thus are untampered. The protocol proceeds as follows.

TABLE III
PARAMETERS USED IN MODELING THE SYSTEM SECURITY

Notation	Name	Explanation
M_r	tampered record number	The number of input records tampered by malicious workers in a phase.
t_r	CBF test probability	The probability of performing test (2) on each distinct input record in input audition.
f_b	CBF false positive rate	The false positive rate of the employed CBF.
$P_{in}(M)$	input tampering detection rate	The detection probability in a phase when malicious workers tampered with M records.
t_f	function audition probability	The probability of performing FEA on each function execution in probabilistic execution audition.
M_f	tampered function execution number	The number of function executions tampered by malicious workers in a phase.
$P_{out}(M)$	output tampering detection rate	The detection probability in a phase when malicious workers tampered with M function executions.

a) The commitment step: According to Section III-C, the execution logs for the same function in the same task are stored in the same invoke/branch log file. Execution logs of different functions are stored in different files. We mark all the invoke log files as $F_v = \{v_1, \ldots, v_n\}$, and mark all the branch log files as $F_b = \{b_1, \ldots, b_n\}$, where v_i and b_i corresponds to function i. Based on F_v and F_b, we build a Merkle trees T [16] consisting of n leaves, where each leaf L_i corresponds to the $\{v_i, b_i\}$ pair. The value of leaf L_i is defined as $\Phi(L_i) = hash(i||hash(v_i)||hash(b_i))$, where $||$ is the concatenation and $hash$ is the Hash one-way function. Following the definition of Merkle tree, the value of each internal node Z is defined as $\Phi(Z) = hash(\Phi(Z_{left})||\Phi(Z_{right}))$, where Z_{left} and Z_{right} are the left and right child of Z. Recursively, the value of the root node R is defined as $\Phi(R) = hash(\Phi(R_{left})||\Phi(R_{right}))$. According to the definition of the Merkle tree, for each leaf node L_i in T, there exists a *proving path*, marked as $\lambda_i = \{\lambda_{i1}, \ldots, \lambda_{ik}\}$, where $k = log(n)$, such that $\Lambda(\Phi(L_i), \lambda_i) = hash(\cdots hash(\Phi(L_i), \lambda_{i1}), \ldots, \lambda_{ik}) = \Phi(R)$. As a *commitment*, the public cloud submits $\Phi(R)$ and n to the private cloud.

b) The challenge and verify step: If the private cloud needs to verify the execution of function j, it will send j to the public cloud as a *challenge*. The public cloud, upon receiving j, will return the private cloud the corresponding file v_j and b_j, as well as the proving path λ_j. To *verify*, the private cloud computes $\Lambda(hash(j||hash(v_j)||hash(b_j)), \lambda_j)$ and checks the result against $\Phi(R)$. If the result is the same as $\Phi(R)$, it will accept v_j and b_j.

Due to the property of the Merkle tree, if the invoke or branch log file received from the public cloud is different from the one used in the commitment step, it is computational infeasible for the public cloud to generate a valid proving path. As a result, the invoke log file and branch log file received during the challenge step are the same as they were committed. Therefore, the invoke log files and branch log files can be stored on the public cloud. The private cloud can request the logs on demand safely.

3) Probabilistic Execution Audition: With the support of the log retrieve protocol, we propose the probabilistic execution audition so that the private cloud can perform the FEA in a sampling manner. To perform the execution audition, the private cloud first performs the commitment step of the log retrieve protocol, receiving $\Phi(R)$ and n. For each function j, with the *function audition probability* t_f, the private cloud retrieves its invoke log file v_j and the branch log file b_j by going through the challenge and verify step of the log retrieve protocol. For the passed function j, the private cloud performs the FEA to audit its integrity. The security analysis of the execution audition is performed in Section IV-B.

IV. SECURITY ANALYSIS

In this section, we first perform security analysis on the input audition and the execution audition. After that, we discuss the security guarantee of the entire MapReduce application. To facilitate the discussion, we list all the parameters used in our analysis in Table III.

A. Security of the Input Audition

For a MapReduce phase k, we need to perform input audition on the phase input I_k, using phase output O_{k-1} as the baseline. Since the CBF-based set compare is performed on each record with probability t_r, the malicious worker does not know which record will be tested. Therefore, we model the malicious worker's behavior as randomly choosing records in I_k to tamper with their values. We define the probability of detecting tampered records in a phase as the *input tampering detection rate*, marked as P_{in}. We mark the input tampering detection rate as $P_{in}(M)$ when the malicious worker randomly chooses M input records to tamper with.

Theorem 1 (Phase Input Integrity Under Probabilistic Input Audition): Suppose the CBF test probability is t_r, the CBF false positive rate is f_b, and the number of records tampered by the malicious worker is M_r, the input tampering detection rate in a phase is

$$P_{in}(M_r) = 1 - \frac{(1 - t_r + t_r f_b)^{M_r}}{M_r!} \tag{3}$$

Proof: Suppose the number of input records in a phase is N. If the malicious worker randomly chooses one record to tamper with, the probability of not detecting this tampering is $1 - t_r + t_r f_b$. If a malicious worker randomly chooses M out of N records to tamper with, in this case, the probability that the input audition does not detect the tampering is

$$\begin{aligned} P_{(one\ case)} &= \prod_{i=0}^{M-1} \left(\frac{1}{N-i}(1 - t_r + t_r f_b) \right) \\ &= (1 - t_r + t_r f_b)^{M_r} \cdot \frac{(N_r - M_r)!}{N_r!} \end{aligned}$$

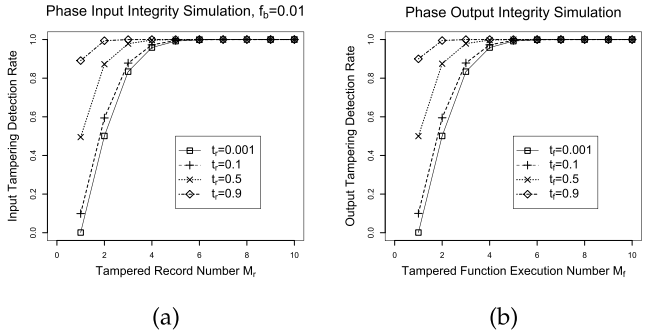


Fig. 2. Theoretical simulations. a) A simulation of the input tampering detection rates with different tampered record numbers and different CBF test probabilities. b) A simulation of the output tampering detection rates with different tampered function execution numbers and different function audition probabilities.

The number of combinations for the malicious worker to choose M_r out of N_r records is $\binom{N_r}{M_r}$. Thus in the current phase, the total probability for the malicious worker to avoid the detection of the input audition is

$$P_{(avoid\ detection)} = \binom{N_r}{M_r} \cdot P_{(one\ case)} = \frac{(1 - t_r + t_r f_b)^{M_r}}{M_r!}$$

Therefore, if the malicious worker tampers with M_r records, the input tampering detection rate is

$$P_{in}(M_r) = 1 - P_{(avoid\ detection)} = 1 - \frac{(1 - t_r + t_r f_b)^{M_r}}{M_r!}$$

□

We simulate the input tampering detection rate $P_{in}(M_r)$ with different tampered record numbers M_r, shown in Fig. 2a). In the simulation, we set the CBF false positive rate f_b as 0.01. As the figure shows, when more than five records are tampered, a very low CBF test probability t_r (such as 0.001) would result in a very high input tampering detection rate (close to 1.0). In other words, the input audition can limit the number of tampered records to a very small value. However, a smaller value of t_r would results in a low $P_{in}(M_r)$ if the tampered record number is less than five. For example, when only one record is tampered, setting t_r to 0.001 would make the detection rate $P_{in}(M_r)$ drop to 0.001. In this case, increasing t_r can result in a higher phase input integrity. According to the figure, setting t_r as 0.1, 0.5 and 0.9, respectively, would obtain a detection rate of 0.1, 0.5 and 0.9, respectively, even if only one record is tampered, with a cost of an increased performance overhead.

B. Security of the Execution Audition

The malicious worker can tamper with the values of variables in function executions to undermine the integrity. For each function execution, the possible targets to be tampered can be any variable in that function. We first discuss the security property of the function execution audition. After that, we reason about the integrity of each phase by considering the probabilistic execution audition introduced.

1) Security of the Function Execution Audition:

Theorem 2 (Function Execution Integrity): Suppose the function execution audition (FEA) is performed on an execution of a `map/reduce` *function and the functions directly or indirectly invoked by that* `map/reduce` *function. If the audition is passed, the statement simulation sequence of each function execution is consistent to the program's control flow, and the constraint set C during each FEA always reflects (i.e., contains and only contains) the correct runtime mathematical relationships among variables.*

Due to the space limit, we informally discuss the proof of Theorem 2. The rigorous proofs will be provided upon request. Since all the execution auditions have passed the FEA, we can imagine that the sequence of the simulated statements in all FEAs is the same as they were executed on the public cloud. This will not affect the audition effect. With such a sequence, we can prove the correctness of the control flow and the constraint set by induction. At the beginning of the simulation, the correctness of the control flow and the constraint set C is trivially true. At any point of the simulation, the previous checks performed in the FEA ensure that the existing runtime control flow and the already generated constraint set C are consistent to the original program. We can prove that, for each case in algorithm 2, after simulating the next statement, the runtime control flow and C will still be consistent to the original program. By induction, the runtime control flow and the constraint set always reflects the original program.

Corollary 1 (Result Integrity): Suppose l is an output log record. Any tampering that results in a change of value in l will be detected by the function execution audition algorithm (FEA).

Proof: According to theorem 2, when statement s is simulated in the FEA algorithm, the corresponding constraint C will reflect the correct runtime mathematical relationships. According to case 2 of algorithm 2, the output log record l will be checked against the constraint set C when s is simulated. Thus tampering variables that results in a change of output log record l will fail the audition. □

Corollary 2 (Phase Output Log Integrity): Suppose the input log of a phase preserves the computation integrity. If function execution auditions (FEA) are performed on each function execution of that phase, the output log of that phase preserves the computation integrity if all the FEAs are passed.

Proof: The computation in a phase consists of multiple function executions. Suppose the input log in a phase is correct, according to Corollary 1, passing the function execution audition of each function execution indicates correct output logs. □

If a malicious worker tampers with the output records in phase i, but keeps the output logs corresponding to those records correct, the tampered output records will be used as the input for phase $i+1$, which will generate tampered input logs. The input audition in phase $i+1$ will detect the inconsistency between the tampered input logs in phase $i+1$ and the correct output logs of phase i, thus detecting the tampering.

2) Security of Probabilistic Execution Audition: We define the probability of detecting the tampered output logs in a MapReduce phase as the *output tampering detection rate*,

marked as P_{out}. We mark the output tampering detection rate as $P_{out}(M_f)$ when the malicious worker randomly chooses M_f function executions to tamper.

Theorem 3 (Phase Output Integrity Under Probabilistic Execution Audition): Suppose the function audition probability is t_f, and the number of function executions tampered by the malicious worker is M_f, the output tampering detection rate is

$$P_{out}(M_f) = 1 - \frac{(1-t_f)^{M_f}}{M_f!} \tag{4}$$

The proof of Theorem 3 is similar to that of Theorem 1. We therefore omit its proof. We simulate the output tampering detection rate $P_{out}(M_f)$ with different tampered function execution numbers M_f, shown in Fig. 2b). In the simulation, we set the function audition probability t_f as 0.001, 0.1, 0.5, and 0.9, respectively, and increase the value of M_f. The simulation result is similar to that of the phase input integrity (Fig. 2a)): setting the function audition probability t_f as low as 0.001 can detect more than five function execution tampering with a very high probability (close to 1.0). When M_f is less than five, the detection rate would be very low. However, increasing t_f to 0.1, 0.5 or 0.9, respectively, can detect even one function execution tampering with a detection rate of 0.1, 0.5 or 0.9, respectively. Yet a higher value of t_f would incur a higher performance overhead.

C. Security of the MapReduce Application

In this section, we analyze the security of an entire MapReduce application when using RIA.

1) Using Basic Input Audition and Execution Audition:

Theorem 4 (Application Output Integrity): Suppose the basic input audition and the basic execution audition are employed. If all the auditions performed on each phase have passed, the application output will be correct.

Theorem 4 can be proved by induction. We skip its proof due to space limit.

2) Using Probabilistic Input Audition and Probabilistic Execution Audition:

Theorem 5 (Application Output Integrity under Probabilistic Checks): Suppose a MapReduce application consists of n phases. In a phase i ($1 \leq i \leq n$), if the tampered input records number is M_{r_i}, the tampered function executions number is M_{f_i}, the CBF test probability in the input audition is t_{r_i}, the CBF false positive rate is f_{b_i} and the function audition probability is t_{f_i}, the probability of detecting the output error in the entire application is

$$P_{app} = 1 - \prod_{i=1}^{n} \left(\frac{(1 - t_{r_i} + t_{r_i} f_{b_i})^{M_{r_i}}}{M_{r_i}!} \cdot \frac{(1-t_{f_i})^{M_{f_i}}}{M_{f_i}!} \right) \tag{5}$$

Theorem 5 can be derived through a straightforward probabilistic analysis, we skip its proof due to the space limit.

V. Discussions

In this section, we discuss the extended solutions to address two related security problems, namely the private cloud security and the data confidentiality.

A. Protecting Integrity on the Private Cloud

In the case where the private cloud is not trusted, extended solutions are needed to protect the integrity of the integrity audition performed on the private cloud. To achieve this goal, we redefine the system architecture by introducing the trusted user host. In this architecture, we assume that both the private cloud and the public cloud are untrusted. We assume the user host is trusted. However, it has a smaller computing capacity compared to the private cloud. Based on this architecture, we propose two alternative solutions, the hardware-based solution and the sampling-based solution.

1) The SGX Based Solution: Our first solution is based on *Intel SGX*, a *Trust Execution Environment* (*TEE*) technology supported since the 6th generation of Intel CPU. Intel SGX [17], [18] allows application to set up protected execution environments (called *enclaves*) without requiring trust in anything but the processor. Enclaves are protected by the processor: the processor controls access to enclave memory. Instructions that attempt to read or write the memory of a running enclave from the outside of the enclave will fail. SGX supports remote attestation. It enables a remote system to verify cryptographically that the specific application has been loaded within an enclave. If the enclave passes the remote attestation, a shared secret will be established, allowing the remote system to bootstrap an end-to-end encrypted channel with the enclave.

Based on Intel SGX technique, we introduce the *Verification Protocol* to ensure that the integrity audition is faithfully performed and the input/output of the audition is untampered. The protocol, shown in Fig. 3, works as follows.

Step 0: (the preparation step): After the original program P was transformed to PWL and executed on the public cloud, the execution logs L were generated. The public cloud generates a public key pair, K_{pr} and K_{pu}, and publishes K_{pu}. The private cloud creates an enclave, called the *Audition Enclave* (*AE*), and loads the *audition program*, i.e., the program of the input audition and the execution audition, in to AE.

Step 1: The user host executes the SGX *Remote Attestation* (*RA*) protocol to verify the authenticity of the audition program loaded in the AE. The protocol is defined in SGX specification [19]. At the end of the remote attestation, a shared key SK between the user host and the AE will be generated. At this time, the user host assures that the AE is loading the correct audition program, and SK is securely shared between the user host and the AE.

Step 2: The user host sends the *program message* $P||MAC_{SK}(P)$ to the AE. $MAC_{SK}(P)$ is the message authentication code of P, using key SK.

Step 3: The AE, upon receiving the program message, verifies the integrity of the message using SK. If the verification fails, the AE knows that program P is tampered and terminates the protocol; otherwise, continue.

Step 4: The public cloud sends the *log message* $L||SIG_{K_{pr}}(L)$, to AE, where L is the execution log and $SIG_{K_{pr}}(L)$ is the digital signature of L, signed with the private key K_{pr}.

Step 5: The AE, upon receiving the log message, verifies the integrity of L using K_{pu}. If the verification fails, the AE knows

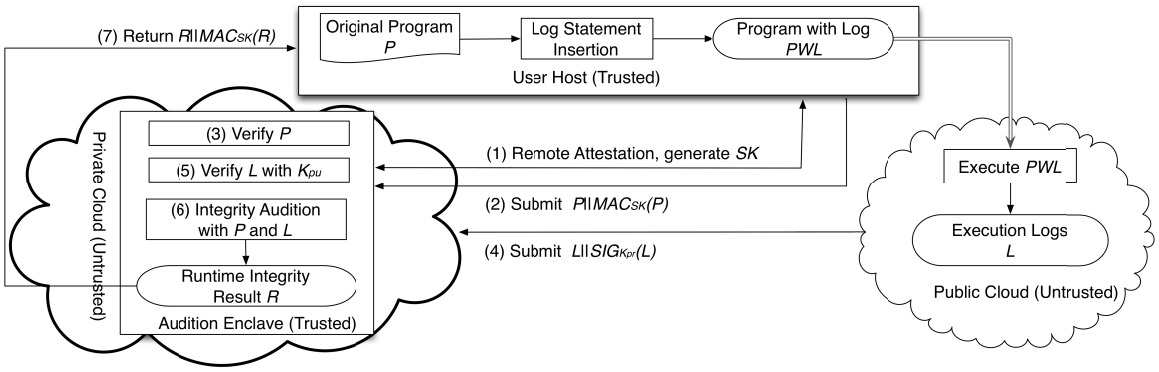


Fig. 3. The verification protocol.

that L is tampered and terminates the protocol; otherwise, continue.

Step 6: The AE executes the audition program with received P and L, and generates the Runtime Integrity Result R.

Step 7: The AE sends the *result message* $R||MAC_{SK}(R)$, to the user host. The user host verifies the integrity of the result message. If the verification is passed, the user host accepts the result.

In the verification protocol, the remote attestation assures that the audition program loaded by the AE is untampered. The SGX technology ensures that the memory of the audition program is protected and cannot be tampered by outside attackers. The integrities of the original program P, the execution log L, and the output R are protected by the MAC or the digital signature. As a result, the integrity of the integrity audition is protected.

2) The Commit-and-Sampling Based Solution: For the private cloud that does not support SGX technology, we further extend the log retrieve protocol introduced in Section III-E2, and propose the *commit-and-sampling protocol* to verify the integrity audition performed on the private cloud.

The protocol is shown in Fig. 4. As MapReduce phase i is executed on the public cloud, the public cloud constructs the Merkle tree $T(L)$ based on the generated invoke logs and the branch logs (Step 0). Similar to the definition of the log retrieve protocol, in $T(L)$, a leaf node L_k is the hash value of all the invoke logs υ_k and branch logs b_k related to the execution of function k. The root value of $T(L)$ is marked as $R(T(L))$ and the proving path for L_k is marked as λ_k. After completing the phase execution, the user host downloads the input and output logs and performs the input audition to ensure the input log of phase i is consistent to the output log of phase $i-1$ (Step 1 and 2). After that, the public cloud commits the leaf node number N and the root value $R(T(L))$ of $T(L)$ to the user host (Step 3).

The user host now can perform multiple rounds of challenges and verifies the responded logs. Specifically, for each round of challenge, the user host randomly selects $k \in [1, N]$ and sends it to the private cloud (Step 4). The private cloud thus performs the execution audition on function k and returns the audition result to the user host (Step 5 and 6). Upon receiving the audition result of function k, the user host sends the sampling request with a certain probability to verify the execution audition performed on the private cloud (Step 7). The private cloud, upon receiving the sampling request, sends the branch logs b_k and invoke logs υ_k to the user host, along with the proving path of L_k, namely λ_k (Step 8). After receiving the response, the user host first validates the integrity of b_k and υ_k by regenerating $R(T(L))$ (Step 9). If the generated $R(T(L))$ is the same as the previously committed value, the user host believes that b_k and υ_k are untampered since its commitment in Step 3. Then, the user host performs the execution audition on b_k and υ_k to determine whether the audition of k on the private cloud is performed faithfully (Step 10).

Notice the challenge process (Step 4 to Step 10) needs to be performed for multiple times. During the protocol execution, failing to pass any check in Step 9 or Step 10 indicates that the audition result returned in Step 6 is incorrect. When the private cloud responds the challenges with incorrect audition results, the probability for the user host detecting the cheating is similar to the output tampering detection rate specified in Theorem 3: suppose the private cloud returns M incorrect audition results, if the user host samples the audition result with probability p, the probability of detecting the cheating is $P_{sampling}(M) = 1 - \frac{(1-p)^M}{M!}$. The simulation of $P_{sampling}(M)$ is similar to Fig. 2b). We skip the discussion due to the page limit. When the result sampling probability (step 7) is set to p, the workload ratio of the user host to the private cloud is p.

B. Protecting Data Confidentiality on the Public Cloud

The solutions discussed so far only address computation integrity. A related but important problem is to protect the confidentiality of sensitive data processed in MapReduce. While the program is executed on the public cloud, the confidentiality of all the sensitive data should be protected. The sensitive data not only includes the sensitive input marked by the programer or user, but also the variables processing the sensitive input and the output generated based on the sensitive input. Constructing the privacy preserving program in untrusted public cloud setting requires non-trivial work. In this section, we only discuss the general solution of constructing privacy preserving programs. Our main focus is how to integrate such a solution into RIA so that confidentiality and integrity can be protected together.

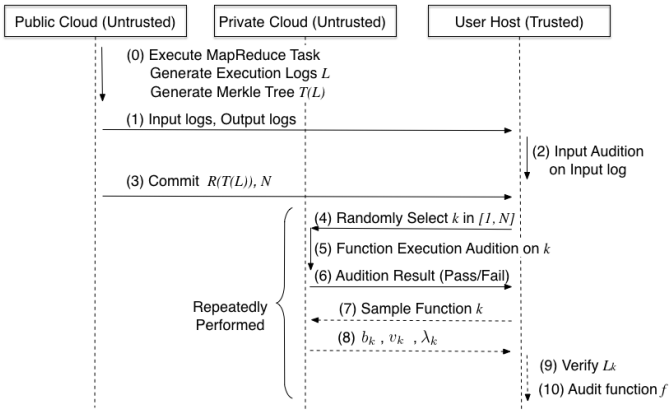


Fig. 4. The commit-and-sampling protocol.

```
1  void map(String key, String values, Context context){
2  //Each token in values: DET
3      String[] words = values.splitToken(System.deliminator);
4      for(String word: words){
5          context.write(word, 1); //word: DET; 1:AH
6      }
7  }
8
9  void reduce(String key, int[] values, Context context){
10 //key: DET, values[]: AH
11     int sum = 0; //sum: AH
12     for(int i=0; i<values.length; i++){
13         sum = sum + values[i];
14         //replaced with sum=AH_add(sum, values[i]);
15     }
16     context.write(key, sum);
17 }
```

Listing 2. The code of Word Count. Comments indicate the encryption and transformation plan.

To protect data confidentiality on MapReduce programs, we can transform the program into a privacy preserving format where sensitive variables are computed in ciphertexts. Which encryption scheme is used for each variable depends on the operation the variable will perform. For instance, variables involved in equality tests can be encrypted with DET (deterministic) [20]; variables involved in comparisons can be encrypted with OP (order-preserving) [21], [22]; variables involved in additions can be encrypted with AH (additive homomorphic) [23], etc. With this idea, we can re-write the program by defining encrypted variables and replacing statements with certain operations with invocations of encryption functions.

As a concrete example, we show how to transform the Word Count application (the first application in Table IV) into the privacy preserving format. The code of Word Count is shown in Listing 2. The comments in the code indicates the encryption and transformation plan. In this application, we assume each word in the input and its count are sensitive. Thus, in the `map` function, each token in the parameter `values` is encrypted with DET. To protect the count, we encrypt the constant `1` with AH (line 5). Encrypting each word with DET ensures that, in the sort and shuffle phase, counts for the same word will be grouped together. In the `reduce` function, we encrypt the variables related to the count in AH, including variable `sum`, and elements in `values`. In line 13, we replace the add operation with `sum=AH_add(sum, values[i])`, where function `AH_add` adds two parameters in AH ciphertexts. As a result, each word is protected in DET and its count is protected in AH.

Automatically transforming a MapReduce program to the privacy preserving form is out of the scope of this paper. Readers can refer to existing solutions, such as [24] and [25]. In this paper, we mainly focus on how RIA can be applied to audit such a program. To generate the execution logs, we use the same method to insert logging statements, as introduced in Section III-C. Since sensitive variables are in ciphertexts, the execution logs record the ciphertexts of those variables. During the audition, the input audition and the execution audition can be performed on the execution logs without any change. One improvement is that we do not have to audit the executions of encryption functions. When the encryption functions, such as `sum = AH_add(sum, values[i])`, are executed on the public cloud, we only generate the invocation logs for the function calls, and do not generate the branch logs and invoke logs inside of the encryption function. To audit the executions of encryption functions, we simply execute those functions and compare the function return values against the post-invoke log.

Combining encryption schemes with RIA is a valid resort in protecting both program integrity and confidentiality. However, it is still necessary to point out that, compared to its original version, the privacy preserving version of a program will introduce non-trivial execution overhead in addition to the overhead incurred by RIA. In Section VI-A, we showed the performance details of RIA when it is used to protect the privacy preserving MapReduce application.

VI. Experiments and Evaluation

We have developed a prototype system, called *MR Auditor*, to perform integrity audition on Apache Hadoop (a mainstream MapReduce implementation) applications. Our experiments showed that MR Auditor can be applied to Hadoop applications directly.

We implemented MR Auditor, with *Soot*, an open source Java-based compiler tool, to perform program analysis and transformation. We used *Symja*, a computer algebra system, to generate constraints and verify the integrity. MR Auditor consists of two parts. The first part, *Log Inserter*, directly analyzes original program's *.class* files and generates the PWL with the *.class* format. The second part, *Integrity Checker*, performs the input audition and the execution audition based on the original program's *.class* file and the execution logs retrieved from the public cloud.

A. Experiments

We performed a set of experiments on Apache Hadoop 1.0.4 to evaluate MR Auditor. The evaluation was measured with two groups of metrics to evaluate the Log Inserter and the Integrity Checker, respectively.

TABLE IV
SPECIFICATIONS OF HADOOP APPLICATIONS

Name	Description
Word Count	Count the number of words from 1 GB of texts. The job consists of 1024 map tasks and 1 reduce task.
Pi	Estimate π using the Monte-Carlo method The job consists of 100 map tasks, where each task generates 100,000 samples.
Tera Sort	Sort 1GB of text strings in the alphabetical order. The job consists of 1024 map tasks and 1 reduce task.
Page Rank	Rank 500,000 web pages by performing the page rank algorithm implemented in *Pegasus*, a Hadoop-based graph mining system. Application consists of six jobs. Each job contains up to 96 map tasks and 48 reduce tasks.
PPWC	Privacy Preserving Word Count. It performs the same job as application one. However, data are processed in ciphertext, as described in Section 5.2.

TABLE V
THE EXECUTION TIME AND THE EXECUTION LOG SIZE OF EACH APPLICATION

Application	Original Program Exec. Time (s)	PWL Exec. Time (s)	Performance Overhead (%)	Execution Log Size (GB)
Word Count	3100	3131	1.00	22.0
Pi	327	333	0.18	3.4
Tera Sort	2926	3044	4.03	5.7
Page Rank	3284	4413	34.38	56
PPWC	7863	10563	34.34	46.3

1) Experiment Setup: In the experiment, we used the Dragon Stack, a cloud server hosted in Xidian University as the public cloud. It was responsible for executing the Hadoop applications (the PWL) and collecting the execution logs. The cloud server was set up with two Intel Xeon E5 CPU (6 cores on each CPU), 128 GB of memory and 6 TB of hard disk storage. We chose XenServer 6.2 as the cloud hypervisor. On this server, we deployed a Hadoop cluster consisting of five Linux virtual machines. Each virtual machine used Ubuntu 14.04.1 and was configured with one core of CPU, 2 GB of memory and 128 GB of storage. We also set up a private cloud to perform the execution audition. The private cloud server was set up with two Intel Xeon E5 CPU (8 cores on each CPU), 128 GB of memory and 3 TB of hard disk storage. We chose KVM 2.0.0 as the cloud hypervisor. On the private cloud, we set up one Linux virtual machine, configured as 8 cores of CPU, 8 GB of memory and 512 GB of storage.

We selected five Hadoop applications, listed in Table IV, to evaluate MR Auditor. The first three applications were selected from the Hadoop 1.0.4 release package. The fourth application was selected from the Hadoop benchmark suite HiBench 3.0.0 [26]. The last application is the privacy preserving version of Word Count (the first application), as introduced in Section V-B.

2) The Evaluation of the Log Inserter: We transformed each original application program into the PWL with the Log Inserter. For each Hadoop application, we measured the execution time of the original application and the PWL, as well as the size of execution logs generated by Log Inserter, as shown in Table V.

Table V showed that when processing a relatively large Hadoop application where the map/reduce task number is significantly large (e.g., more than 1,000 map task) and the computing logic is fairly complex (e.g., the Pi and the Page Rank application), the performance overheads for the tested applications are moderate (ranging from 0.18% to 34.48%). However, the size of the execution logs is fairly large (ranging from 3.4 GB to 56 GB), introducing a large storage overhead.

In this set of experiments, PPWC processes the same data as Word Count application. However, data are processed in ciphertexts. Due to the repeated execution of encryption function (such as `AH_add` in Listing 2), the execution time of original PPWC is 2.54 times of Word Count. When the PWL of PPWC was executed on the public cloud, it incurred 34.34% of performance overhead compared to the original PPWC, and generates 46.3 GB of logs, in which 14 GB are the invocation logs for the encryption functions.

The size of the generated execution logs can be further reduced. For instance, compressing the log files generated in Word Count application can reduce the log size from 22 GB to 4.23 GB, obtaining 81% of size reduction. In addition, as shown in Section VI-A3, the log retrieve protocol will significantly reduce the size of logs transmitted between clouds.

Since the execution of each PWL is executed independently, the execution time on the public cloud will grow linearly with the number of computation requests, if the computation loads of all requests are the same. However, one potential bottleneck of Log Inserter is the storage capacity. Notice that each application in our experiments generated several Gigabytes of data even after the compression, when more applications are executed on the public cloud, the space to store the execution logs will be very high, which could affect the scalability.

3) The Evaluation of the Integrity Checker: We evaluated the performance of the Integrity Checker with two applications, Word Count and PPWC. The performance about Word Count is listed in Table IV. Specifically, we performed the input audition and the execution audition on the execution logs generated by the Word Count application and recorded their execution times.

When evaluating the input audition, we recorded the execution time of the input audition under different CBF test probability t_r. We listed the execution times under each t_r in Table VI.

The result indicated that the efficiency of the input audition was fairly high. For instance, when t_r is set to 100%, the

TABLE VI
THE INPUT AUDITION TIME FOR WORD COUNT

CBF Test Probability t_r (%)	Execution Time (s)
1	239
10	244
50	309
100	384

TABLE VII
THE EXECUTION AUDITION DETAILS FOR WORD COUNT

Func. Audition Prob. t_f (%)	Audition Time (s)	Audited Func. No.	Retrieved Branch & Invoke Log Size (MB)
0.1	192	3949	4.50
1	1571	55775	43.57
10	11723	411500	477.01

audition speed is $95792543/384 \approx 249\ k/s$. We also observed that the execution time of the input audition increases as the test probability t_r increases. The data shown in Table VI indicates that, compared to the execution time (i.e., 239 seconds) when t_r is 1%, when t_r increases from 50% to 100%, the increased execution times are 70 seconds and 145 seconds, respectively. The increase rate of the execution time is 107%, which is close to the increase rate of t_r (i.e., 102%).

We further evaluated the performance of the execution audition and listed the details in Table VII. We changed the function audition probability t_f to measure the time of the execution audition, listed on the second column. We also counted the number of function executions audited in each setting of t_f, listed on the third column. In the last column, we listed the size of execution log files retrieved from the public cloud.

Data in Table VII indicates that the audition time is generally proportional to the audited function number. On average, the number of function execution auditions performed in each second is $29.98/s$. The sizes of the retrieved branch and invoke log files are very small compared to the execution log size in Table V. They are in general proportional to t_f.

When t_f was set to 0.1% (as the first row of Table VII), the execution audition can be completed in 192 seconds, retrieving 4.5 MB of invoke and branch log files and 3.6 GB of input and output log files (In this experiment, the size of the input and output logs are 3.6 GB. In our design, all the input and output log files has to be retrieved from the public cloud.). The first row of Table VI indicates that, when setting t_r as 1%, the input audition can be completed in 239 seconds. Thus, the entire verification can be completed in 431 seconds. Combining Table VII, Table V and Table VI, we summarize that, to verify the Word Count application (setting t_r as 1% and t_f as 0.1%), RIA incurs $431/3100 = 14\%$ of extra time of execution on the private cloud, 1% of extra time of execution on the public cloud, an extra 22 GB of data stored on the public cloud and 3.6 GB of data transmitted from the public cloud to the private cloud. Comparing with the data in Table V, only 16% of generated logs are transmitted across the cloud. The size of the transmitted logs can be further reduced with optimizations. For instance, simply compressing the transmitted log can reduce the size from 3.6 GB to 712 MB, further obtaining 81% of size reduction.

TABLE VIII
THE EXECUTION AUDITION DETAILS FOR PPWC

t_f (%)	Audition Time (s)	Audited Other Func. No.	Audited Encrypt Func. No.	Audition Time for Enc. Func. (s)
0.1	463	7443	106292	188
1	3068	73845	1065015	222

Table VIII listed the performance of Integrity Checker when auditing PPWC. When the function audition probability t_f is set as 0.1%, the audition time is 463 seconds, which is 2.4 times of Word Count (the first row of Table VII). The extra audition time is caused by the extra audited function executions, including 106,292 encryption functions and 3,494 non-encryption functions. We also observed that compared to auditing the non-encryption functions, auditing the encryption function is much faster. According to the last two columns of the table, when only auditing encryption functions, auditing an extra $958,723$ encryption functions only took 34 seconds, the function audition efficiency is $28k/s$.

The integrity audition is performed phase by phase. Thus the audition time will grow linearly with the phase number. In each phase, the private cloud needs to download the entire input and output logs and parts of the branch and invoke logs from the public cloud. The log transmission between the clouds may become a bottleneck. For instance, processing 1 GB of data with Word Count requires to transmit 712 MB of compressed logs. When the size of the processed data increases, the transmission time would increase and may affect the scalability.

B. Financial Evaluation

RIA is useful only when the financial cost of using RIA to be cheaper than directly executing applications on the private cloud. In this section, we evaluate the financial costs. Based on the costs, we derive the feasibility condition in which employing RIA is economically cheaper than directly using the private cloud. Specifically, we build two models for the two cases above. The parameters used in the models are listed in Table IX.

Assuming the computing capacity on the private could and the public cloud is the same, we obtain the cost for running the application directly on the private cloud as $C_{private} = T_{app} \cdot P_{prv}$ and the cost for using RIA as $C_{RIA} = T_{pub} \cdot P_{pub} + T_{prv} \cdot P_{prv} + S_{proof} \cdot P_{net}$.

To better facilitate the understanding, we give a concrete value for each parameter so that we can obtain the real cost for $C_{private}$ and C_{RIA}. For the application execution time, the public execution time, the private execution time and the proof size, we use the data in the Word Count experiment, where t_r and t_f are set as 1% and 0.1%, respectively. We normalize the computing capacity for both the public cloud and the private cloud, according to the configurations depicted in Section VI-A1. For instance, the public cloud consisting of five instances of one-core CPU and 2GB memory

TABLE IX
PARAMETERS FOR FINANCIAL EVALUATION MODEL

Notation	Name	Explanation	Value (Word Count, $t_r = 1\%, t_f = 0.1\%$)
T_{app}	application execution time	The time spent in executing the original application when using a single instance.	3100 s/instance* 5 instances
T_{pub}	public execution time	The time spent on the public cloud for running the application with RIA when using a single instance.	3131 s/instance * 5 instances
T_{prv}	private execution time	The time spent on the private cloud for running the application with RIA when using a single instance.	431 s/instance * 8 instances
S_{proof}	proof size	The size of execution logs transmitted from the public cloud to the private cloud.	3.6 GB
P_{pub}	public cloud price	The price of renting one computing unit for one unit time on the public cloud.	\$0.023/hour/instance (AWS t2.small)
P_{prv}	private cloud price	The price of using one computing unit for one unit time on the private cloud.	Not avaliable
P_{net}	network price	The price of transmitting one unit of data from the public cloud to the private cloud.	\$0.09/GB (AWS to Internet)

is considered as five instances. The private cloud consisting of one instance of eight-core CPU and 8GB of memory is considered as 8 instances. For the public cloud price and the network price, we referred to the Amazon AWS public cloud pricing webpage [27] and selected the price of the t2.small instance, which has the same configuration as the public cloud instance in our experiment. The private cloud price is not available on the Internet. However, we can derive a feasible private cloud price by solving the inequality $C_{private} > C_{RIA}$. As a result, we have

$$P_{prv} > (T_{pub} \cdot P_{pub} + S_{proof} \cdot P_{net})/(T_{app} - T_{prv}) \quad (6)$$

Replacing the parameters with values in Table IX, we have $P_{prv} > \$0.13/hour/instance$. In other words, using the values listed in Table IX, RIA will incur less cost than directly using the private cloud if the private cloud price is greater than $\$0.13/hour/instance$.

C. Compare With Proof-Based Verifiable Computation

Proof-based verifiable computation [28] is a class of solutions that verifies the integrity of the remote computation. In this class of works, the verifier (the private cloud in our setting) will generate a set of constraints C about the runtime values of variables in the program. During the execution, the prover (the public cloud in our setting) will generate the proof of the computation π based on the runtime value of variables. The verifier then will perform a set of tests on the proof π based on C. If the program was executed faithfully, the test will pass; otherwise, the test will fail, except for a very small probability. Systems following this direction include Pepper [29], Buffet [14], TinyRAM [30], etc. Compared to RIA, this class of works has strong limitation in terms of language expressiveness, performance, and flexibility.

In this set of solutions, Buffet [14] achieves the best mix of performance and generality in the literature. Even for Buffet, it can support only a subset of C language (disallowing function pointer and `goto` statement). TinyRAM [30] can support all the C language. But it incurs an expensive overhead in both the verifier and the prover. As of this writing, existing solutions in this direction can only support simple programs. In contrast, RIA supports real Java language and can be used on the real applications, such as Hadoop MapReduce.

TABLE X
PERFORMANCE COMPARISON BETWEEN PANTRY AND RIA ON THE MAP PHASE OF *Dot Product* TEST CASE (M = 20K)

	Pantry	RIA
Baseline	10ms	51s
constraint size	0.15MB	80MB
Prover's cost	780s	66s
Prover's cost / Baseline	$7.8 * 10^4$	1.29
Verifier's cost	324s	111s
Verifier's cost / Baseline	$3.24 * 10^4$	2.18

From the performance's perspective, such a class of solution usually has a significant performance overhead on the prover and in the set up phase of the verifier. Since Pantry [12] is the only system claiming to support MapReduce, we compared RIA with Pantry. Our comparison chose the dot product test case used in [12, Fig. 10]. The MapReduce applications in Pantry is implemented in C language with Open-MPI. Since RIA only supports Java, we implemented the same algorithm in Hadoop MapReduce and applied RIA on the implemented program. In RIA, we set both t_r and t_f as 100%. In this test, we executed the application on a cluster consisting only one virtual machine instead of five. The configuration of the virtual machine on the private cloud is the same as our previous experiment. Since the data in [12, Fig. 10] only reflects the map phase, our experiment also only focused on the map phase. The comparison data is listed in Table X.

The Pantry column lists the data appeared in [12, Fig. 10]. The constraint size is calculated by assuming 12 constraints constitute one byte (according to [12, Sec.8.1]). The prover's cost is the total time required for the prover to complete the map phase of one job. The verifier's cost is the total time required for the verifier to setup and to verify the map phase of one job. The RIA column lists the fact of the map phase of one dot product job. Since Hadoop is a full-fledged MapReduce framework implemented in Java, it takes a longer time (51s) to complete the map phase. The constraint size is the size of the execution log. The Prover's cost is the execution time of all the PWL map tasks executed on the public cloud. The verifier's cost is the time took for the integrity audition. The figures indicate that RIA has a much better performance than Pantry, however incurs a higher constraint size. Compared to

the baseline, RIA only incurs 29% of overhead on the prover and 118% of overhead on the verifier. In contrast, the execution times of Pantry on the prover and on the verifier are both 10^4 time higher than the baseline.

In addition to the expressiveness and the performance advantages, RIA is able to check the execution logs with a probabilistic manner. Such a feature enables the audition of scaled-up computation, making RIA a practical solution.

VII. Related Works

Secure co-processor [31] provided a hardware-based solution towards the trusted outsourced computing. The Trusted Platform Modules (TPMs) protects the result correctness by verifying whether the worker's environment complies to the specification. Such a solution has a strict requirement on worker's environment, thus lacks the flexibility. The Trusted Execution Environment technology, such as Intel SGX [17], [18] offers an alternative option, which allows trusted software to be executed in a trusted environment. Using such a technology, Schuster proposed VC3 [32] to protect the execution integrity of MapReduce. This solution still depends on a specific hardware configuration. In addition, the trusting base of such a system usually is carefully tailored, which makes this method difficult to be generalized.

Anomaly detection-based solution works on the code level. By statically modeling the program, collecting the program execution trace, and comparing the consistency between the model and the program execution trace, this type of method can verify the runtime integrity. Previous works, such as [4] and [5], focus on the integrity of the program's control flow while ignoring the integrity of its data flow. [7] and [8] proposed methods to cover the data flow integrity of the system call parameters. However, they cannot protect the tampering of variables that are not involved in system calls. Shu *et al.* [6] proposed to combine dynamic tracing and machine learning to detect abnormal assemble language instruction sequence. However, such a method has certain false positive and false negative rate.

Practical solutions towards such a problem fall to the direction of computing replication [33], [34], computing verification [35], trust management [36], and the combined method [11]. The main challenge in this direction is to achieve security guarantee while minimizing the cost. Such a class of works covers various environment settings, including grid computing [37], Cloud Computing [9], [10], [38], Fog Computing [39], etc.

Proof-based verifiable computation [28] searches for theoretical solutions, utilizing cryptography and computing complexity theorem. The framework of this direction is similar to our work. Despite a few systems are proposed to realize such a framework, including Pantry [12], Pinocchio [13], and Buffet [14], they are not practical due to the lack of program expressiveness, performance, and flexibility. The detailed discussion can be found in Section VI-C.

An orthogonal research direction is the control flow integrity [40], [41]. Such a class of works transforms the binary code to enforce control flow checks during the execution. This direction, only focusing on local control flow hijackings, cannot be applied to the remote computation. Even for the local control flow hijacking, existing works only raises the bar for such an attack, and cannot defeat complicated attacks, such as Control-Flow Bending [42] and Conti *et al.*'s work [43].

The earlier research outcomes of this paper have been published in [44]. This paper has improved the previous work from multiple aspects, which includes reducing the log size, improving security from different aspects, performing more thorough evaluations, etc.

VIII. Conclusion and Future Work

In this paper, we propose the Runtime Integrity Audition (RIA) technique, a method to remotely verify the runtime integrity of MapReduce applications. We developed a prototype system called MR Auditor based on the idea of RIA, and tested its applicability and the performance with several Hadoop applications. Our experimental results showed that our tool is compatible with all the applications that we tested and incurs a moderate performance overhead.

RIA has shown a promising prospect. However, we believe that refining the design will further improve the integrity guarantee and reduce the performance overhead. From the integrity's perspective, detecting a small number of tampering with the sampling-based solution would be an interesting topic to work on. From the performance's perspective, further reducing the execution log size on the public cloud and during the cross-cloud transmission would further reduce the performance overhead.

Acknowledgments

The authors thank Jiawen Ma and Cuicui Su for their contributions on performing benchmark experiments. We also thank anonymous reviewers for their valuable comments and suggestions.

References

[1] *Top 10 Cloud Fiascos*. Accessed: May 21, 2016. [Online]. Available: http://www.networkcomputing.com/cloud/top-10-cloud-fiascos/96279858

[2] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, 2009, pp. 199–212.

[3] S. Bugiel, S. Nürnberger, T. Pöppelmann, A.-R. Sadeghi, and T. Schneider, "AmazonIA: When elasticity snaps back," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 389–400.

[4] J. T. Giffin, S. Jha, and B. P. Miller, "Efficient context-sensitive intrusion detection," in *Proc. NDSS*, 2004.

[5] J. T. Giffin, S. Jha, and B. P. Miller, "Detecting manipulated remote call streams," in *Proc. USENIX Secur. Symp.*, 2002, pp. 61–79.

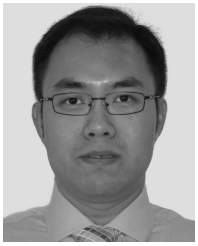
[6] X. Shu, D. Yao, and N. Ramakrishnan, "Unearthing stealthy program attacks buried in extremely long execution paths," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 401–413.

[7] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow anomaly detection," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2006, pp. 1–15.

[8] P. Li, H. Park, D. Gao, and J. Fu, "Bridging the gap between data-flow and control-flow analysis for anomaly detection," in *Proc. Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Dec. 2008, pp. 392–401.

[9] W. Wei, J. Du, T. Yu, and X. Gu, "SecureMR: A service integrity assurance framework for MapReduce," in *Proc. Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Dec. 2009, pp. 73–82.

[10] Y. Wang and J. Wei, "VIAF: Verification-based integrity assurance framework for MapReduce," in *Proc. IEEE CLOUD*, Jul. 2011, pp. 300–307.
[11] Y. Wang, J. Wei, S. Ren, and Y. Shen, "Toward integrity assurance of outsourced computing—A game theoretic perspective," *Future Generat. Comput. Syst.*, vol. 55, pp. 87–100, Feb. 2016.
[12] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish, "Verifying computations with state," in *Proc. 24th ACM Symp. Oper. Syst. Principles (SOSP)*, 2013, pp. 341–357.
[13] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 238–252.
[14] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish, "Efficient RAM and control flow in verifiable outsourced computation," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2015, pp. 1–25.
[15] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[16] R. C. Merkle, *A Digital Signature Based on a Conventional Encryption Function*. Berlin, Germany: Springer, 1988, pp. 369–378.
[17] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proc. 2nd Int. Workshop Hardw. Archit. Support Secur. Privacy*, vol. 13. 2013.
[18] F. McKeen *et al.*, "Innovative instructions and software model for isolated execution," in *Proc. HASP ISCA*, 2013, pp. 1–8.
[19] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel software guard extensions: EPID provisioning and attestation services," Intel Corp., Santa Clara, CA, USA, White Paper, 2016.
[20] S. Bruce, *Applied Cryptography*, 2nd ed. New York, NY, USA: Wiley, 1996.
[21] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-preserving symmetric encryption," in *Advances in Cryptology—EUROCRYPT*. Cham, Switzerland: Springer, 2009, pp. 224–241.
[22] A. Boldyreva, N. Chenette, and A. O'Neill, "Order-preserving encryption revisited: Improved security analysis and alternative solutions," in *Advances in Cryptology—CRYPTO*. Cham, Switzerland: Springer, 2011, pp. 578–595.
[23] P. Paillier *et al.*, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology—EUROCRYPT*. Cham, Switzerland: Springer, 1999, pp. 223–238.
[24] Y. Dong, A. Milanova, and J. Dolby, "JCrypt: Towards computation over encrypted data," in *Proc. 13th Int. Conf. Principles Practices Program. Java Platform, Virtual Mach., Lang., Tools (PPPJ)*, 2016, pp. 8:1–8:12.
[25] S. D. Tetali, M. Lesani, R. Majumdar, and T. Millstein, "MrCrypt: Static analysis for secure cloud computations," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA)*, New York, NY, USA, 2013, pp. 271–286.
[26] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proc. IEEE 26th Int. Conf. Data Eng. Workshops (ICDEW)*, Mar. 2010, pp. 41–51.
[27] *Cloud Services Pricing*. Accessed: Sep. 13, 2017. [Online]. Available: https://aws.amazon.com/pricing/services/
[28] M. Walfish and A. J. Blumberg, "Verifying computations without reexecuting them," *Commun. ACM*, vol. 58, no. 2, pp. 74–84, Jan. 2015.
[29] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish, "Making argument systems for outsourced computation practical (sometimes)," in *Proc. NDSS*, 2012, vol. 1. no. 9, p. 17.
[30] . Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, *SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge*. Berlin, Germany: Springer, 2013, pp. 90–108.
[31] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proc. 13th Conf. USENIX Secur. Symp. (SSYM)*, vol. 13. 2004, pp. 223–238.
[32] F. Schuster *et al.*, "VC3: Trustworthy data analytics in the cloud using SGX," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2015, pp. 38–54.
[33] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982.
[34] M. Castro *et al.*, "Practical Byzantine fault tolerance," in *Proc. OSDI*, vol. 99. 1999, pp. 173–186.
[35] P. Golle and I. Mironov, "Uncheatable distributed computations," in *Topics in Cryptology—CT-RSA*. Cham, Switzerland: Springer, 2001, pp. 425–440.
[36] S. Javanmardi, M. Shojafar, S. Shariatmadari, and S. S. Ahrabi, "FR trust: A fuzzy reputation–based model for trust management in semantic P2P grids," *Int. J. Grid Utility Comput.*, vol. 6, no. 1, pp. 57–66, 2014.
[37] W. Du, J. Jia, and M. Murugesan, "Uncheatable grid computing," in *Proc. 24th Int. Conf. Distrib. Comput. Syst.*, Jan. 2004, pp. 4–11.
[38] Y. Wang, J. Wei, and M. Srivatsa, "Result integrity check for MapReduce computation on hybrid clouds," in *Proc. IEEE CLOUD*, Jun./Jul. 2013, pp. 847–854.
[39] M. Shojafar, N. Cordeschi, and E. Baccarelli, "Energy-efficient adaptive resource management for real-time vehicular cloud services," *IEEE Trans. Cloud Comput.*, to be published.
[40] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 4:1–4:40, Oct. 2009.
[41] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2015, pp. 914–926.
[42] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proc. USENIX Secur. Symp.*, 2015, pp. 161–176.
[43] M. Conti *et al.*, "Losing Control: On the effectiveness of control-flow integrity under stack attacks," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2015, pp. 952–963.
[44] Y. Wang and Y. Shen, "POSTER: RIA: An audition-based method to protect the runtime integrity of MapReduce applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2016, pp. 1799–1801.



Yongzhi Wang received the B.S. and M.S. degrees in computer science from Xidian University, China, in 2004 and 2007, respectively, and the Ph.D. degree in computer science from Florida International University, Miami, FL, USA, in 2015. Since 2015, he has been an Assistant Professor with Xidian University. His research interests include big data, cloud computing security, and outsourced computing security.

Yulong Shen received the B.S. and M.S. degrees in computer science and the Ph.D. degree in cryptography from Xidian University, Xian, China, in 2002, 2005, and 2008, respectively. He is currently a Professor with the School of Computer Science and Technology, Xidian University, and also an Associate Director of the Shaanxi Key Laboratory of Network and System Security. He has also served on the technical program committees of several international conferences, including the ICEBE, the INCoS, the CIS, and the SOWN. His research interests include wireless network security and cloud computing security.

Xiaohong Jiang (SM'09) received the B.S., M.S., and Ph.D. degrees from Xidian University, China, in 1989, 1992, and 1999, respectively. He was an Associate Professor with Tohoku University from 2005 to 2010. He is currently a Full Professor with Future University Hakodate, Japan. He has published over 260 technical papers at premium international journals and conferences, which include over 50 papers published in top IEEE journals and top IEEE conferences, such as the IEEE/ACM TRANSACTIONS ON NETWORKING, the IEEE JOURNAL OF SELECTED AREAS ON COMMUNICATIONS, the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, and IEEE INFOCOM. His research interests include computer communications networks, wireless networks and optical networks, network security, and routers/switches design. He is a member of the ACM and the IEICE. He was a recipient of the Best Paper Award at the IEEE HPCC 2014, the IEEE WCNC 2012, the IEEE WCNC 2008, the IEEE ICC 2005-Optical Networking Symposium, and the IEEE/IEICE HPSR 2002.