POSTER: RIA – an Audition-based Method to Protect the Runtime Integrity of MapReduce Applications

Yongzhi Wang
School of Computer Science and Technology,
Xidian University, P.R. China
Key Laboratory of Grain Information Processing
and Control (Henan University of Technology),
Ministry of Education, P.R. China
yzwang@xidian.edu.cn

Yulong Shen
School of Computer Science and Technology
Xidian University, P.R. China
ylshen@mail.xidian.edu.cn

ABSTRACT

Public cloud vendors have been offering varies big data computing services. However, runtime integrity is one of the major concerns that hinders the adoption of those services. In this paper, we focus on MapReduce, a popular big data computing framework, propose the runtime integrity audition (RIA), a solution to verify 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 the performance with multiple Hadoop applications. Our experimental results showed that MR Auditor is an efficient tool to detect runtime integrity violation and incurs a moderate performance overhead.

Keywords

Computation Integrity, Remote Verification, MapReduce

1. INTRODUCTION

MapReduce, the fundamental framework supporting various big data applications, has been delivered as a computation service by public cloud vendors. However, in this model, since computations are performed remotely, security is beyond the control of customers. Security breach incidents reported by mass media [1] and vulnerabilities discovered by researchers [3] have obstructed the wide adoption of such a service. Among varies security issues, runtime integrity is one of the most critical ones that is closely related to the MapReduce service. Existing works relying on specific hardware environment [4], task redundancy [5], or cryptographic construction [2], either lose flexibility or suffer from high performance overhead. Therefore, a generalized method that incurs moderate performance overhead is needed.

In the paper, we propose *runtime integrity audition*, (or *RIA* for short), a hybrid cloud-based method, to protect the runtime integrity of MapReduce applications executed on the untrusted public cloud. By inserting logging statements to MapReduce programs, RIA records the *execution traces* of the MapReduce applications executed on the public cloud. Using the application's program and the application input data as the baseline, RIA audits the execution traces against those baseline on the private cloud. Intuitively, by auditing the integrity of the input data, the control flow, the data flow and the output data of each MapReduce phase, we ensure the integrity of that phase. By ensuring the integrity of all phases, we ensure the integrity of the final result of the MapReduce application.

The audition on each phase consists of two parts, i.e., the input audition and the execution audition. The input audition uses the output of the last phase as a baseline and checks the input of the current phase against the baseline. The execution audition is performed on each execution of a function. While auditing a function execution, RIA derives mathematical constraints among variables and checks the consistency between the execution traces and the mathematical constraints to determine the runtime integrity.

The novelty of RIA is that it extracts mathematical constraints from the programs and uses them to audit the runtime integrity. The design of execution auditions can be performed independently on each function execution, thus can be applied in a sampling-based manner, which provides users a flexibility to make a trade-off between the audition accuracy and the audition time.

Based on RIA, we implemented *MR Auditor*, a prototype system, that can perform the runtime integrity audition on Hadoop applications. Our experimental results showed that MR Auditor can efficiently audit the runtime integrity of Hadoop applications and incurs a moderate performance overhead (with a 0.18% to 34% of extra execution time with an unoptimized implementation).

2. SYSTEM DESIGN

2.1 System Model and Security Assumptions

We define that an ordinary MapReduce application consists of n phases, marked as p_1, p_2, ..., p_n, where $n/2$ phases are map phases and the other $n/2$ are reduce phases, executed in an interleaved sequence. Each phase p_i consists of q_i tasks, marked as t_{i1}, t_{i2}, ..., t_{iq_i}. The application input I is originally stored in the private cloud and then transmitted to the public cloud. The output of the last job, O, is the output of the entire application, which will be sent back to the private cloud after the entire application completes.

CCS'16 October 24-28, 2016, Vienna, Austria

ACM ISBN 978-1-4503-4139-4/16/10.

DOI: http://dx.doi.org/10.1145/2976749.2989042

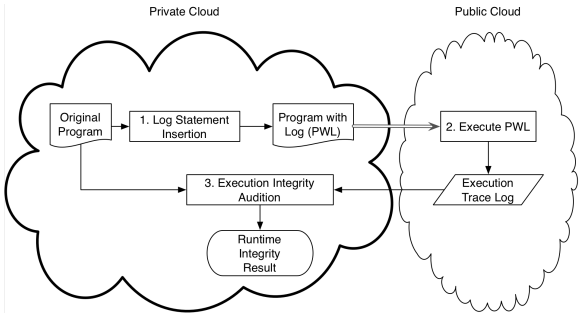

Figure 1: The architecture of RIA

In the hybrid cloud architecture, we assume that the public cloud is untrusted and the private cloud is trusted. In the audition procedure performed on the private cloud, we assume the application program and the application input I are trusted. The execution traces and the application output O are untrusted, and thus needs to be audited.

2.2 System Overview

The runtime integrity audition (RIA) is performed on a hybrid cloud environment, shown in Fig. 1. On the private cloud, RIA inserts logging statements to the original program, generating *PWL* (in step 1). PWLs are then sent to the public cloud to execute (in step 2). During the execution, PWLs generate *execution trace logs*, which reflect the statements execution sequence and the runtime variable values. When the application execution completes, the execution trace logs will be transferred back to the private cloud to be audited. (in step 3). The audition mainly checks the data integrity of the input, the runtime integrity of each task, and the data integrity of the output on each phase.

The audition is shown in Algorithm 1. The numberings of phases and tasks in the algorithm are defined in Section 2.1. The algorithm uses the application input I and the application program as the trusted anchor and audits the integrity of each phase one by one. In each phase, it first audits the input data of the current phase by comparing it against the output of the previous phase (or against I if the current phase is the first phase) (line 6-7). Then, it audits the executions of each task of the current phase (line 8-10). The above two steps ensure the integrity of the output data in the current phase. The output data of the current phase is collected (line 11) and will be used as a baseline for auditing the input of the next phase. If the output of the last phase passes the audition, it will be used as a baseline to audit the application output O (line 13). During the audition, any inconsistency indicates a violation of the runtime integrity.

We elaborate the technical details in the succeeding sections, including the PWL generation (step 1 in Fig. 1), the input audition (line 6-7 of Algorithm 1) and the execution audition (line 9 of Algorithm 1).

2.3 The Execution Trace Logs Generation

We insert logging statements to the original program, so that the execution trace logs will be generated while the program is executed on the public cloud. The execution trace logs are in the format of `<variable name, variable value>` (see example in Table 1), recording the runtime variable values.

Algorithm 1 Runtime Integrity Audition

1: $inputBaseLine \leftarrow I$
2: **for** $i = 1$ to n **do**
3: **for** $j = 1$ to q_i **do**
4: $ETL_{ij} \leftarrow retrieveExecutionTraceLog(t_{ij})$
5: **end for**
6: $phaseInput \leftarrow extractInput(\{ETL_{i1}, \cdots, ETL_{iq_i}\})$
7: $inputAudition(phaseInput, inputBaseLine)$
8: **for** $j = 1$ to q_i **do**
9: $executionAudition(ETL_{ij})$
10: **end for**
11: $inputBaseline \leftarrow extractOutput(\{ETL_{i1}, \cdots, ETL_{iq_i}\})$
12: **end for**
13: $inputAudition(O, inputBaseline)$

```
1 void func(int x, int y){ //invoke func(6,0)
2     while(x>5){
3         y+=x;
4         x--;
5     }
6     print(y);
7 }
```

Figure 2: A sample function

We insert 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 in by the task. In the MapReduce case, the logging statements are inserted to record parameter values of the `map` or `reduce` function. The output log records the data written out by the task. In the MapReduce case, the logging statements are inserted to record the parameters in the invocation of function `context.write(outKey, outValue)`. The branch log records the variable values involved in the predicate of the branch statement. Therefore logging statements are inserted before the branch statement to record variable values involved in each branch condition. Each invoke log consists of two parts, inserted before and after each function invocation. The former is called the *pre-invoke log*, recording the values of parameters before the function was invoked. The latter is called the *post-invoke log*, recording the values of the parameters after the function was invoked, as well as its return value.

2.4 The Input Audition

The input audition is performed on a MapReduce phase basis. For each phase, the input audition is performed based on the input logs of the current phase and the output logs of the previous phase. Since the output records of a task in the previous phase can be distributed to multiple tasks in the current phase, we collect the output logs of all tasks in the previous phase into the *output data set*, collect the input logs of all tasks in the current phase into the *input data set*, and perform the equality test for the two sets. The efficiency of the equality test can be improved by employing the Counting Bloom Filter.

2.5 The Execution Audition

The execution audition can be performed on each executions of the functions implemented by the customer, including the MapReduce interface functions (e.g., `map`, `reduce`

Table 1: The execution audition details of invoking `func(6, 0)`

Line #	Input	Action	Constraints (C)
1	pre:$\{x = 6, y = 0\}$	update(C,pre)	$x = 6 \wedge y = 0$
2	bra:$\{x = 6\}$; stmt:$\{x > 5\}$	check(C,bra); update(C,stmt)	$x = 6 \wedge y = 0 \wedge x > 5$
3	stmt:$\{y = y + x\}$	update(C,stmt)	$x = 6 \wedge y - x = 0 \wedge x > 5$
4	stmt:$\{x = x - 1\}$	update(C,stmt)	$x + 1 = 6 \wedge y - (x + 1) = 0 \wedge x + 1 > 5$
2	bra:$\{x = 5\}$; stmt:$\{x <= 5\}$	check(C,bra); update(C,stmt)	$x + 1 = 6 \wedge y - (x + 1) = 0 \wedge x + 1 > 5 \wedge x \leq 5$
6	pre:$\{y = 6\}$; post:$\{y = 6\}$	check(C,pre); update(C,post)	$x+1 = 6 \wedge y-(x+1) = 0 \wedge x+1 > 5 \wedge x \leq 5 \wedge y = 6$

, and `partition`) and other functions recursively invoked by those interface functions. When performing the execution audition on one function, we skip the audition of the functions invoked by the current function and postpone its audition after the current audition completes. Such a design removes the cohesion between the caller and the callee, making the audition of each execution independently. As a result, execution auditions can be performed in a sampling-based manner, thus improving the audition efficiency.

To audit the execution of a function, we generate the *control flow graph (CFG)* of that function based on the application's program and simulate the execution based on the CFG and its trace logs. With the branch log, the simulation can derive which branch was taken while executed on the public cloud, and thus to reproduce the runtime control flow. During the simulation, we also derive the mathematical constraints among runtime variable values based on the semantics of each simulated statement and the runtime variable values recorded in certain types of trace logs, including the input logs and the post-invoke logs. The runtime variable values recorded in other trace logs, including the output logs, the branch logs and the pre-invoke logs, will be checked against the derived mathematical constraints. During the simulation, any check inconsistency indicates a runtime integrity violation on the public cloud.

As a concrete example, Table 1 shows the execution audition details on the invocation of `func(6, 0)`. The implementation of function `func` is shown in Fig. 2. The first column in the table tracks the statements been executed during the simulation. The second column records the input information used for deriving mathematical constraints, including the pre-invoke logs (*pre*), the branch logs (*bra*), the post-invoke logs (*post*) and the executed statement (*stmt*). Based on the input, the audition performs the corresponding actions, shown in the third column. The last column indicates the resulting constraints after the action. For example, when an *if* [1] statement is executed, (i.e., line 2), we first check the branch logs against the constraints. If the check is passed, we update the constraints by adding the predicate of the *if* statement.

3. IMPLEMENTATION AND EXPERIMENTS

We developed a prototype system called *MR Auditor* to perform runtime integrity audition on Apache Hadoop (a mainstream MapReduce implementation) applications. Specifically, we use *Soot*, an open source Java-based compiler tool, to perform program analysis and transformation. We use *Symja*, a computer algebra system, to perform constraints generation and the integrity verification.

[1]The *while* statement is usually implemented as a combination of *if* and *goto* statements

We set up a hybrid cloud environment and performed a set of experiments to evaluate MR Auditor. Our results showed that MR Auditor can be applied to Hadoop applications directly and incurs a moderate performance overhead. We selected four Hadoop applications, including Word Count, Pi, Terasort and Pegasus PageRank, to evaluate MR Auditor on Hadoop 1.0.4. The experimental results showed that MR Auditor can be applied to all the four applications successfully. The application execution overhead on the public cloud is 9.89% on average. The average efficiency for input audition is *584,100 records/second*. The average efficiency for the function audition is *53.8 functions/second*.

4. CONCLUDING REMARKS

If the malicious cloud executes a tampered function, it will generate incorrect trace logs that are inconsistent with the application's input and the original program. If the attacker returns correct trace logs with incorrect application output, the output will be inconsistent to the trace logs, the application's input and the original program. Therefore, RIA is a secure method to protect the runtime integrity of MapReduce applications. Our experiments indicate that RIA incurs a modest performance overhead on the application execution and therefore a promising method.

Acknowledgments

This paper is supported in part by the Open Fund of the Chinese Key Laboratory of the Grain Information Processing and Control (No. KFJJ-2015-202), the Fundamental Research Funds for the Central Universities (No. XJS16042, JB160312 and BDY131419), and the NSFC (No. U1536202, 61571352, 61373173, 61602364 and 61602365).

5. REFERENCES

[1] Top 10 cloud fiascos. http://www.networkcomputing.com/cloud/top-10-cloud-fiascos/96279858. Accessed: 2016-05-21.

[2] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*. ACM, 2013.

[3] S. Bugiel, S. Nürnberger, T. Pöppelmann, A.-R. Sadeghi, and T. Schneider. Amazonia: when elasticity snaps back. In *CCS*. ACM, 2011.

[4] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *S&P '15*. IEEE, 2015.

[5] W. Wei, J. Du, T. Yu, and X. Gu. Securemr: A service integrity assurance framework for mapreduce. In *ACSAC*, 2009.