ABSTRACT

DU, JUAN. Service Integrity Assurance in Large-Scale Cloud Systems. (Under the direction of Dr. Xiaohui (Helen) Gu and Dr. Douglas S. Reeves.)

Internet has evolved into an important service delivery infrastructure instead of merely providing host connectivity. With rapid adoption of the concepts of Software as a Service (SaaS), Service Oriented Architecture (SOA), and Cloud Computing, service oriented cloud systems have emerged as cost-effective platforms for users to access various software applications as services via Internet.

However, cloud systems are often shared by multiple tenants from different security domains, which makes them vulnerable to various malicious attacks. Moreover, cloud systems often host long-running applications such as massive data processing, which provides more opportunities for attackers to exploit the system vulnerability and perform strategic attacks.

This dissertation focuses on securing data processing applications in large-scale multi-tenant cloud systems. It includes three studies on service integrity assurance for data processing applications in cloud systems.

The first study designs, implements, and evaluates RunTest, a scalable runtime integrity attestation framework. RunTest provides light-weight application-level attestation to dynamically verify the integrity of data processing services and pinpoint malicious service providers in cloud systems. RunTest validates service integrity by aggregating and analyzing result consistency information and utilizes a clique based attestation graph analysis algorithm to pinpoint malicious service providers and recognize colluding attack patterns.

The second study designs, implements, and evaluates IntTest, an integrated service integrity attestation framework that can efficiently verify the integrity of dataflow processing services and quickly pinpoint malicious service providers within a large-scale cloud infrastructure. In contrast to RunTest, IntTest can effectively detect more challenging colluding attacks and mitigate false alarms with more relaxed assumption than RunTest. Furthermore, we have investigated stateful dataflow processing and have provided service integrity attestation schemes supporting stateful services.

The third study designs, implements, and evaluates AdapTest, an adaptive continuous service integrity attestation framework for large-scale cloud systems. Building on top of RunTest, AdapTest can (1) significantly reduce attestation overhead and shorten detection delay; and (2) automatically detect and correct corrupted data processing results produced by the cloud system.

All of the three systems have been implemented on top of the IBM System S streaming processing system and tested on the virtual computing lab (VCL), a production virtualized
computing cluster that operates in a similar way as Amazon EC2. Our experimental results show that our schemes are effective and impose low performance impact for data processing in the cloud system.

We have also identified two other security threats toward dataflow processing applications in cloud systems, including data attacks and dataflow topology attacks. We have provided efficient and effective countermeasures to mitigate such attacks.
Service Integrity Assurance in Large-Scale Cloud Systems

by

Juan Du

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2011

APPROVED BY:

Dr. Munindar P. Singh
Dr. Ting Yu

Dr. Douglas S. Reeves
Co-chair of Advisory Committee

Dr. Xiaohui (Helen) Gu
Chair of Advisory Committee
DEDICATION

To my family and friends.
BIOGRAPHY

Juan Du received her BS and MS degrees in Computer Science from Tianjin University in 2001 and 2004, respectively. At North Carolina State University, her research is motivated by the emergence of cloud systems as promising service provisioning platforms and new security challenges imposed on the application services running inside cloud. Her research has focused on advancing the state of the art of cloud system and service security and developing a set of practical integrity assurance schemes to address the new challenges.
I would like to thank all those who have supported and encouraged me during my PhD study. First, I would like very much to thank my co-advisors, Dr. Xiaohui (Helen) Gu and Dr. Douglas Reeves, for all of their help. Their aid in directing the research has been invaluable, and I have learned a great deal from them about how to do research, how to write a good research paper, and so on.

Second, I would like to thank other members of my advisory committee, Dr. Munindar P. Singh and Dr. Ting Yu, for their valuable feedback and comments on my research. I would also like to thank Dr. Yu for his time to participate in brainstorming and many helpful insights offered during our project meetings.

Third, I would like to give my thanks to two of my friends and collaborators, Wei Wei and Nidhi Shah. They have been very supportive and they were a big help to make the papers published.

Fourth, I would like to thank all my friends for their help during my PhD study: Wei Wei, Nidhi Shah, Yongmin Tan, Zhenhuan Gong, Zhiming Shen, Hiep Nguyen, Kamal Kc, Sethuraman Subbiah, Vinay Venkatesh, Daniel Dean, An Liu, Pan Wang, Dingbang Xu, Qinghua Zhang, Youngjune Pyun, Kyuyong Shin, Jungki So, YoungHee Park, Wu Zhou, Zhi Wang, Yao Liu, and so on.

Fifth, I would like to thank National Science Foundation (NSF) and U.S. Army Research Office (ARO) for their funding support under grants CNS0915567, CNS0915861 and W911NF-10-1-0273, respectively.

Finally, I would like to give my special thanks to my family for their encouragement and support, without whom this would have been much more difficult. My daughter Diana always makes me believe I am the happiest person in the world.
# TABLE OF CONTENTS

List of Tables .............................................................................................................. vii

List of Figures ................................................................................................................. viii

Chapter 1 Introduction ...................................................................................................... 1
  1.1 Motivation ................................................................................................................ 2
  1.2 Summary of the State of the Art ............................................................................. 2
  1.3 Research Challenges ............................................................................................. 3
  1.4 Summary of Contributions ...................................................................................... 3
  1.5 Organization of Dissertation .................................................................................. 4

Chapter 2 Background ...................................................................................................... 6
  2.1 Multi-Tenant Cloud Computing Infrastructure ...................................................... 7
  2.2 Dataflow Processing Services ................................................................................ 8

Chapter 3 RunTest: Scalable Runtime Service Integrity Attestation Framework for Large-Scale Cloud Systems .......................................................................................... 10
  3.1 Attack Model and Assumptions ............................................................................. 10
  3.2 Design and Algorithms ......................................................................................... 12
    3.2.1 Approach Overview ......................................................................................... 12
    3.2.2 Integrity Attestation Graph ............................................................................. 13
    3.2.3 Pinpointing Malicious Service Providers ...................................................... 15
    3.2.4 Identifying Attack Patterns .......................................................................... 18
  3.3 Security Analysis ..................................................................................................... 22
    3.3.1 Security Properties ......................................................................................... 22
    3.3.2 Data Quality ................................................................................................. 23
    3.3.3 Detection Probability ..................................................................................... 24
  3.4 Implementation and Evaluation ............................................................................. 25
    3.4.1 System Implementation .................................................................................. 25
    3.4.2 Results and Analysis ...................................................................................... 27
  3.5 Summary ................................................................................................................ 33

Chapter 4 IntTest: Integrated Service Integrity Attestation Scheme for Large-Scale Cloud Systems .............................................................................................................. 35
  4.1 Assumptions ............................................................................................................ 36
  4.2 Design and Algorithms ......................................................................................... 36
    4.2.1 Replay-based Consistency Check ................................................................... 37
    4.2.2 Stateful Service Attestation ......................................................................... 37
    4.2.3 Attestation Graph Model ............................................................................... 39
    4.2.4 Consistency Graph Analysis .......................................................................... 41
    4.2.5 Inconsistency Graph Analysis ....................................................................... 41
    4.2.6 Integrated Attestation Graph Analysis .......................................................... 44
LIST OF TABLES

Table 2.1 Notations. ................................................. 6
Table 4.1 Graph analysis time in IntTest. ......................... 55
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Cloud-based data processing.</td>
<td>1</td>
</tr>
<tr>
<td>2.1</td>
<td>Multi-tenant cloud infrastructure.</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Dataflow processing application example.</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Dataflow processing application example.</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Integrity attestation graph example.</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>An example of finding maximal cliques.</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Consistency clique discovery algorithm.</td>
<td>19</td>
</tr>
<tr>
<td>3.5</td>
<td>Attestation graph motif A.</td>
<td>20</td>
</tr>
<tr>
<td>3.6</td>
<td>Attestation graph motif B.</td>
<td>20</td>
</tr>
<tr>
<td>3.7</td>
<td>Attestation graph motif C.</td>
<td>21</td>
</tr>
<tr>
<td>3.8</td>
<td>Attestation graph motif D.</td>
<td>21</td>
</tr>
<tr>
<td>3.9</td>
<td>Cloud Dataflow Integrity attack detection algorithm.</td>
<td>22</td>
</tr>
<tr>
<td>3.10</td>
<td>Dataflow processing application used for RunTest.</td>
<td>25</td>
</tr>
<tr>
<td>3.11</td>
<td>Service instance deployment for RunTest.</td>
<td>26</td>
</tr>
<tr>
<td>3.12</td>
<td>Service integrity attack detection rate under non-collusion scenario where $b_i = 1$ and $c_i = 0$.</td>
<td>28</td>
</tr>
<tr>
<td>3.13</td>
<td>Service integrity attack detection rate under non-collusion scenario where $b_i = 0.2$ and $c_i = 0$.</td>
<td>28</td>
</tr>
<tr>
<td>3.14</td>
<td>Service integrity attack detection rate under collusion scenario where $b_i = 1$ and $c_i = 1$.</td>
<td>29</td>
</tr>
<tr>
<td>3.15</td>
<td>Service integrity attack detection rate under collusion scenario where $0 &lt; b_i &lt; 1$ and $c_i = 1$.</td>
<td>29</td>
</tr>
<tr>
<td>3.16</td>
<td>Service integrity attack detection rate under collusion scenario where $0 &lt; b_i &lt; 1$ and $0 &lt; c_i &lt; 1$.</td>
<td>30</td>
</tr>
<tr>
<td>3.17</td>
<td>Detection rate under different attestation probability in $(b_i = 1, c_i = 1)$ scenario.</td>
<td>30</td>
</tr>
<tr>
<td>3.18</td>
<td>Detection rate under different misbehaving probability in $(0 &lt; b_i &lt; 1, c_i = 0)$ scenario. (attestation probability = 0.2)</td>
<td>31</td>
</tr>
<tr>
<td>3.19</td>
<td>Computing time of clique discovery algorithm.</td>
<td>31</td>
</tr>
<tr>
<td>3.20</td>
<td>Comparison of actual data quality and reported data quality.</td>
<td>32</td>
</tr>
<tr>
<td>3.21</td>
<td>Dataflow processing delay comparison.</td>
<td>32</td>
</tr>
<tr>
<td>3.22</td>
<td>Detection time comparison between our scheme and the full-time majority voting scheme.</td>
<td>33</td>
</tr>
<tr>
<td>3.23</td>
<td>Attestation overhead comparison between our scheme and the full-time majority voting scheme.</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Stateful functions.</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Indirect state recovery for stateful functions.</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>Per-function consistency graphs.</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>Global inconsistency graph.</td>
<td>40</td>
</tr>
</tbody>
</table>
Figure 4.5  Isolated benign service providers in the global inconsistency graph. ........................................ 42
Figure 4.6  Inconsistency graph G and its residual graph. ...................... 43
Figure 4.7  Malicious service provider pinpointing algorithm. .............. 45
Figure 4.8  An example for integrated graph analysis. ......................... 46
Figure 4.9  Another example for integrated graph analysis. .......... 46
Figure 4.10  Dataflow processing application used for IntTest. ............. 48
Figure 4.11  Service instance deployment for IntTest. .......................... 49
Figure 4.12  Detection rate ($A_D$) and false alarm rate ($A_F$) comparison with 20% aggressive malicious attackers. .................. 50
Figure 4.13  Detection rate ($A_D$) and false alarm rate ($A_F$) with 20% conservative malicious attackers. .................. 51
Figure 4.14  Detection rate ($A_D$) and false alarm rate ($A_F$) with 40% aggressive malicious attackers. .................. 52
Figure 4.15  Detection rate ($A_D$) and false alarm rate ($A_F$) with 40% conservative malicious attackers. .................. 53
Figure 4.16  Detection time comparison. ........................................ 53
Figure 4.17  Attestation overhead comparison. .................................. 54
Figure 4.18  Detection rate of IntTest under different attestation probability. 54
Figure 4.19  Detection rate of IntTest under different misbehaving probability. 55
Figure 4.20  Average dataflow processing delay under different data rate. 56
Figure 4.21  Average dataflow processing delay under different service hops. 57
Figure 4.22  Comparison of actual data quality and reported data quality. 57

Figure 5.1  Data-driven service integrity attestation. .......................... 60
Figure 5.2  Weighted attestation graph. ......................................... 62
Figure 5.3  Per-hop adaptive attestation. ....................................... 64
Figure 5.4  Multi-hop adaptive attestation. ..................................... 65
Figure 5.5  Automatic data correction using attestation data processing results. ................................................. 66
Figure 5.6  Dataflow processing application used for AdapTest. .......... 67
Figure 5.7  Service instance deployment for AdapTest. ....................... 68
Figure 5.8  Detection time comparison under colluding attacks with node misbehaving probability = 0.2. (40% malicious nodes) ....... 69
Figure 5.9  Detection time comparison under colluding attacks with node misbehaving probability = 0.4. (40% malicious nodes) ....... 69
Figure 5.10  Attestation overhead comparison under colluding attacks. (40% malicious nodes) ........................................ 70
Figure 5.11  Detection time comparison under non-colluding attacks with node misbehaving probability = 0.2. (40% malicious nodes) ....... 70
Figure 5.12  Detection time comparison under non-colluding attacks with node misbehaving probability = 0.4. (40% malicious nodes) ....... 71
Figure 5.13  Attestation overhead comparison under non-colluding attacks. (40% malicious nodes) ........................................ 71
Figure 5.14  Attestation overhead comparison under non-colluding attacks. (20% malicious nodes) ........................................... 72
Figure 5.15  Detection time comparison between RunTest, AdapTest and the full time majority voting scheme. .......................... 73
Figure 5.16  Attestation overhead comparison between probabilistic attestation schemes and the full time majority voting scheme. .... 73
Figure 5.17  Result quality detection and auto-correction performance under non-colluding attacks. (20% malicious nodes) ............ 74
Figure 5.18  Result quality detection and auto-correction performance under colluding attacks. (40% malicious nodes) ............... 74
Figure 5.19  Data processing delay comparison .................................................. 75
Figure 6.1  Dataflow processing with carry-on data provenance evidence. . 78
Figure 6.2  Cascading dataflow topology encryption. ................................. 79
Figure 6.3  Dataflow processing delay under different service hops. .......... 82
Figure 6.4  Dataflow processing delay under different data size. ............... 82
Figure 6.5  Dataflow processing delay under dropping attack. ................. 83
Figure 6.6  Dataflow processing delay with varying number of service hops under dropping attack. ....................................... 84
Chapter 1

Introduction

With rapid adoption of the concepts of Software as a Service (SaaS) [4] and Service Oriented Architecture (SOA) [11, 31], cloud computing infrastructures [1] have emerged as promising service provisioning platforms. Cloud systems allow different users, called tenants, to lease computing resources on-demand to avoid the expensive cost of owning and maintaining dedicated computing infrastructures. Thus, application service providers can conveniently use the cloud infrastructure to provide their software as services in a cost-effective way.

Cloud systems are particularly amenable for data processing services [2, 25, 36, 48], which are often extremely resource-intensive. In particular, our work focuses on dataflow processing systems [7, 39, 49, 60] that have many real world applications such as security surveillance, scientific study, and business intelligence. As shown by Figure 1.1, users can feed data from certain data sources into the cloud system to perform various data processing functions such as correlation, filtering, or any other stream mining services, and receive final data processing results from the cloud. The data source could be either public or private data set produced by various sources.

Figure 1.1: Cloud-based data processing.
1.1 Motivation

Cloud systems are often shared by multiple tenants that belong to different security domains, which makes them vulnerable to various malicious attacks. Moreover, data processing services are often long-running, which provides more opportunities for attackers to exploit the system vulnerability and perform strategic colluding attacks. For example, attackers can pretend to be legitimate service providers to provide fake service components and the service components provided by benign service providers may include security holes that can be exploited by attackers. Although virtualization ensures certain isolation between users, malicious attackers can still leverage the shared hardware to launch attacks [13, 58] from the virtual machines (VMs) they own or by compromising VMs of benign users.

One of the key security concerns for cloud-based data processing is the result integrity, no matter whether private or public data are processed by the cloud system. In open cloud systems consisting of a large number of service providers from different security domains, we can no longer assume all service providers are trustworthy and strive to provide the promised services. A malicious service provider can perform an arbitrary service function instead of the advertised one. We call such an attack service integrity attack. For example, an online shopping service can be provided by several service providers, each providing a different service function. The service provider responsible for credit checking may skip the credit checking algorithm by reporting all credit claims as qualified. Although confidentiality and privacy protection problems have been extensively studied by previous research [10, 19, 47, 64, 81, 82], the service integrity problem has not been properly addressed.

1.2 Summary of the State of the Art

Previous work on distributed dataflow processing mainly focuses on resource and performance management issues, such as selecting optimal dataflow composition based on the user’s quality-of-service (QoS) requirements and load balancing objectives [7, 39, 44, 49, 50, 60]. It usually assumes that all data processing components are trustworthy. This assumption generally holds for small-scale closed cluster systems, where data processing providers and users are from the same security domain or from collaborative domains with strong pre-existing trust. In open cloud systems, security measurements have to be used to provide service integrity assurance.

Although previous work has proposed various remote integrity attestation techniques [16, 33, 35, 51, 65, 67–69], existing solutions often require trusted hardware or secure kernel to coexist with the remote computing platform, which are difficult to be deployed in cloud systems. Traditional Byzantine Fault Tolerance (BFT) techniques (e.g., [8, 20, 23, 24, 46, 54, 55]) can detect arbitrary Byzantine faults or integrity violation using replicated services, however, those
techniques often incur high overhead and impose certain agreement protocol over all replicas.

1.3 Research Challenges

In summary, it is challenging to provide service integrity assurance for dataflow processing applications in large-scale cloud systems. We identify the following four key research challenges for performing service integrity attestation in large-scale multi-tenant cloud systems:

- **Scalability:** Dataflow processing applications often demand real-time high performance data processing, which requires service integrity assurance schemes to be light-weight. The integrity assurance schemes should run continuously in order to timely detect malicious service integrity attacks and minimize the negative impact on data processing results.

- **Robustness:** Malicious or compromised service providers may control multiple service components that provide a data processing service. Service integrity assurance schemes should be effective even in case of strategic colluding attacks.

- **Transparency:** Ideally, service integrity assurance schemes should be transparent to service providers so that malicious service providers can gain little knowledge of any information that may help them escape detection.

- **Applicability:** Service integrity assurance schemes should avoid modifying the underlining cloud software and hardware platform as well as the service provider software, since it is often infeasible to get access or authorization to modify them. Ideally, integrity assurance schemes should be non-intrusive and treat service providers as black boxes.

To this end, this dissertation explores a set of scalable data-driven integrity attestation approaches to detect malicious attacks and pinpoint malicious service providers, which are completely transparent to the attested services without imposing any special software or hardware requirements.

1.4 Summary of Contributions

The contributions of this dissertation are summarized as follows:

- **RunTest** [30, 77]: We developed RunTest, a scalable runtime integrity attestation framework that provides light-weight application-level attestation to dynamically verify the integrity of dataflow processing services and pinpoint malicious service providers in multi-tenant cloud systems. We proposed a novel randomized service integrity attestation
scheme that validates service integrity by aggregating and analyzing result consistency information and a *clique based attestation graph analysis* algorithm to pinpoint malicious service providers and recognize colluding attack patterns.

- **IntTest [26,27]**: We developed IntTest, an integrated service integrity attestation scheme that can efficiently verify the integrity of dataflow processing services and quickly pinpoint malicious service providers within a large-scale cloud infrastructure. In contrast to RunTest, IntTest can effectively detect more challenging colluding attacks and mitigate false alarms with more relaxed assumption than RunTest. We have also investigated stateful dataflow processing services and provided replay-based integrity attestation scheme that supports stateful service attestation.

- **AdapTest [28]**: We developed AdapTest, an adaptive data-driven service integrity attestation scheme for multi-tenant cloud systems. Building on top of RunTest, AdapTest can (1) significantly reduce attestation overhead and shorten detection delay; (2) automatically detect and correct corrupted data processing results produced by the cloud system. AdapTest achieves the above by dynamically evaluating the trustworthiness of different services based on previous attestation results, adaptively selecting services to attest, and providing optimized attestation for multi-hop data processing services.

- We have implemented RunTest, IntTest and AdapTest on top of the IBM System S stream processing system and tested them on the NCSU virtual computing lab. Our experimental results show that these service integrity attestation schemes are effective and impose low performance impact for dataflow processing in cloud systems. Our schemes can reduce attestation overhead by one order of magnitude compared to Byzantine Fault Tolerant schemes. Specifically, compared to RunTest, AdapTest can further reduce attestation overhead by up to 60% and shorten the detection delay by up to 40%.

- We have identified another two security attacks besides service integrity attack toward dataflow processing in cloud systems [29], including data attacks and topology attacks. We have provided efficient and effective countermeasures to protect data and topology and mitigate malicious attacks.

### 1.5 Organization of Dissertation

The organization of this dissertation is as follows. Chapter 2 gives a background overview on multi-tenant cloud systems and dataflow processing services. Chapter 3 presents the design, implementation, and evaluation of RunTest, a scalable runtime service integrity attestation framework that provides light-weight application-level attestation for data processing services
in multi-tenant cloud systems. Chapter 4 presents the design, implementation, and evaluation of IntTest, an integrated service integrity attestation scheme for dataflow processing services in large-scale cloud systems. Chapter 5 presents the design, implementation, and evaluation of AdapTest, an adaptive data-driven service integrity attestation scheme for multi-tenant cloud systems. Chapter 6 presents other security threats including data and topology attacks and provides our countermeasures to achieve secure dataflow processing. Chapter 7 discusses the related work. Chapter 8 concludes this dissertation and discusses the future work.
Chapter 2

Background

In this chapter, we provide a background overview about the multi-tenant cloud computing infrastructure and dataflow processing services. Table 2.1 summarizes all the notations used in this dissertation.

<table>
<thead>
<tr>
<th>notation</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$</td>
<td>service provider</td>
</tr>
<tr>
<td>$f_i$</td>
<td>service function</td>
</tr>
<tr>
<td>$s_i$</td>
<td>dataflow service component</td>
</tr>
<tr>
<td>$d_i$</td>
<td>application data tuple</td>
</tr>
<tr>
<td>$m_i$</td>
<td>malicious service component</td>
</tr>
<tr>
<td>$A$</td>
<td>the set of malicious service components</td>
</tr>
<tr>
<td>$b_i$</td>
<td>the probability that $m_i$ misbehaves on a data tuple</td>
</tr>
<tr>
<td>$c_i$</td>
<td>the probability that $m_i$ colludes with its colluders</td>
</tr>
<tr>
<td>$P_u$</td>
<td>attestation probability</td>
</tr>
<tr>
<td>$r$</td>
<td>number of copies for a tuple</td>
</tr>
<tr>
<td>$K$</td>
<td>maximum number of malicious service providers</td>
</tr>
<tr>
<td>$C_G$</td>
<td>minimum vertex cover of graph $G$</td>
</tr>
<tr>
<td>$N_p$</td>
<td>the neighbor set of node $p$</td>
</tr>
<tr>
<td>$G'_p$</td>
<td>the residual graph of $G$</td>
</tr>
<tr>
<td>$\Omega$</td>
<td>the set of malicious service providers identified by the global inconsistency graph</td>
</tr>
<tr>
<td>$M_i$</td>
<td>the set of malicious service providers identified by consistency graph in service function $f_i$</td>
</tr>
</tbody>
</table>
2.1 Multi-Tenant Cloud Computing Infrastructure

Multi-tenant cloud infrastructures provide a promising platform for different service providers to deliver their software services in an economical way, as illustrated by Figure 2.1. Cloud infrastructures [1, 3] allow users to lease computing resources from the cloud to run their applications without maintaining complex physical infrastructures. Application service providers can conveniently use the cloud infrastructure to provide their software as services. Each service provider \( p_i \) can lease a set of virtual machines (VMs) to host their services and may provide multiple service functions.

Each data processing service component, denoted by \( s_i \), is a self-contained software unit that provides a specific data processing function, denoted by \( f_i \), such as sorting, filtering, correlation, or data mining utilities. Each service component can have one or more input ports for receiving input application data tuples, denoted by \( d_i \), and one or more output ports to emit output tuples. In a large-scale cloud system, multiple service components can be functionally-equivalent, providing the same service function but running within different VMs on different hosts. These redundant service components can be owned by different service providers and may have different internal implementations. Such redundancy property exists due to several reasons: i) service providers may create replicated service components for load balancing and fault tolerance purposes; and ii) popular services may attract different service providers for
Figure 2.2: Dataflow processing application example.

profit. Note that we treat service instances as black boxes. We do not impose any special hardware or software requirements on either application services or the cloud platform.

A multi-party service provisioning infrastructure usually employs some portal nodes [40, 61] to aggregate different service components into composite services based on the user’s requirements. The portal node interacts with users directly by forwarding input data from users to different service components and then forwarding the final processing results to users. For security protection, the portal node can perform authentication on users to prevent malicious users from disturbing normal service provisioning.

2.2 Dataflow Processing Services

Data processing systems (e.g., Google’s MapReduce [25], Yahoo’s Hadoop [2], IBM System S [36], Microsoft Dryad [48]) have become increasingly popular with applications in many domains such as business intelligence, security surveillance, and scientific computing. In particular, our work focuses on continuous dataflow processing applications [36, 39], which supports high performance in-memory data processing. For example, Figure 2.2 shows an example of dataflow processing application. The application reads data from NOAA (the National Oceanic and Atmospheric Administration) that reports weather conditions from weather stations throughout the world, and generates the most recent weather information for different locations where the fruit suppliers are located. Produced results will help in making decisions on whether to purchase fruit from a supplier. The raw data tuples are first converted into a set of structured streams of weather condition information, and then are filtered to keep data only for some specific locations. Then the next step is to perform some conversions and calculations on the weather data for different locations respectively. The weather data will then be joined with the supplier location data to generate the required output.

A dataflow processing service is often provided through a set of data processing functions.
To access dataflow processing services from the cloud infrastructure, users can make a request through portal nodes. In the request, users specify the required service functions and inter-function dependencies as well as desired quality of service (QoS) levels. Based on the user’s service function specification, the portal node selects service components and aggregates into composite services [61], illustrated by Figure 2.1. The portal node accepts input data from the user and delivers final results to the user. Each input tuple gets computed by the service functions along the composed service path. Different from previous work, which employs such portal nodes for load balancing and QoS management [40], our work leverages the portal node to perform service integrity attestation.
Chapter 3

RunTest: Scalable Runtime Service Integrity Attestation Framework for Large-Scale Cloud Systems

In this chapter, we present RunTest, a scalable runtime service integrity attestation framework that provides light-weight application-level attestation to dynamically verify the integrity of dataflow processing services and pinpoint malicious service providers in multi-tenant cloud systems. We first describe the attack model and our assumptions. Then, we present the design details of the RunTest system. We then describe our implementation and discuss evaluation results. Finally, we summarize the work.

3.1 Attack Model and Assumptions

Malicious attackers can launch security attacks to a cloud infrastructure by pretending to be legitimate service providers or taking control of vulnerable service providers. Our work focuses on detecting integrity attacks to cloud dataflow processing where a malicious (or compromised) service node gives untruthful data processing results. Compared to user data privacy and confidentiality, integrity assurance is the most prevalent, which is needed no matter whether public or private data are processed by the cloud.

Unlike standalone web servers or clusters, cloud infrastructures comprise a large number of distributed service provisioning nodes. We must consider colluding attack scenarios when multiple malicious attackers collude or multiple service sites are simultaneously compromised and controlled by a single malicious attacker. We assume that malicious nodes have no knowledge of other nodes except those they interact with for receiving and forwarding data or their colluding parties. However, attackers can communicate with their colluders in any arbitrary...
Given a specific data processing function, suppose there are $w$ malicious service components $A = \{m_1, m_2, ..., m_w\}$. For each attacker $m_i$, it has a set of colluders, which is a subset of $A$, denoted by $L$. Given an input tuple, if $m_i$ ($1 \leq i \leq k$) is the first attacker among its colluders to receive this tuple, $m_i$ has $b_i$ ($0 < b_i \leq 1$) probability to misbehave on this tuple. If $m_i$ is not the first one, it has $c_i$ ($0 \leq c_i \leq 1$) probability to collude with its colluders by giving the same results with what their colluders have given, and $1 - c_i$ to behave independently. We define that two malicious components are non-concluding if they give inconsistent results on the same input. Further, if a malicious service component always gives correct data processing results (i.e., misbehaving probability $b_i = 0$), we say that the component’s integrity attack fails since the accuracy of data processing results is not impacted. Thus, we can use parameters $b_i$ and $c_i$ to represent an attacker’s behavior.

Malicious components have various strategies to choose from: they can always misbehave or probabilistically misbehave; they can collude with their colluders in different degrees and in different ways. We characterize all possible attack scenarios using different combinations of parameters ($b_i, c_i$) and classify those attacks into five attack patterns.

- **Non-Collusion Always Misbehave (NCAM).** Malicious components always act independently and always give incorrect results. It corresponds to $b_i = 1$ and $c_i = 0$.

- **Non-Collusion Probabilistically Misbehave (NCPM).** Malicious components always act independently and give incorrect results probabilistically with probability less than 1. Different malicious components may have different misbehaving probability $b_i$. It corresponds to $0 < b_i < 1$ and $c_i = 0$.

- **Full Time Full Collusion (FTFC).** Malicious components always collude and always give the same incorrect results, corresponding to $b_i = 1$, and $c_i = 1$.

- **Partial Time Full Collusion (PTFC).** Malicious components always collude and give the same incorrect results on selected tuples, corresponding to $0 < b_i < 1$ and $c_i = 1$. Malicious components cannot arbitrarily select tuples to misbehave, since they only have a local view and do not know if the selected tuple has already passed through a benign component or not. The portal is bound to detect the existence of malicious behavior if it receives different results for the same tuple. In order to reduce the risk of being detected, malicious components can select tuples to misbehave based on a pre-agreement. For example, colluding malicious components may choose to misbehave if the received tuple has an even number sequence number.

- **Partial Time Partial Collusion (PTPC).** Malicious components sometimes collude and sometimes act independently. It corresponds to $0 < b_i < 1$ and $0 < c_i < 1$. 


We assume that dataflow processing services are input-deterministic, that is, given the same input, a benign node always produces the same output. We also assume data processing services are stateless. Many dataflow processing functions fall into this category, such as selection, filtering, and mapping [2, 36].

3.2 Design and Algorithms

In this section, we present the details of our RunTest System. We first give an overview of our approach. Then, we describe the integrity attestation graph model that serves as the basis for our integrity attack analysis. We then describe a clique-based malicious node pinpointing algorithm followed by the attack pattern recognition algorithm for identifying colluding attack behavior in large-scale cloud infrastructures.

3.2.1 Approach Overview

The design objectives of the RunTest system are to identify untruthful dataflow processing results, pinpointing malicious data processing service providers, and detecting colluding attack behavior. To achieve these goals, we propose a new integrity attestation graph model to capture aggregated cross-node integrity attestation results, which includes the statistical output consistency/inconsistency information from different dataflow processing nodes. By analyzing the features of weighted attestation graphs, we can comprehensively verify the integrity of different dataflow processing results produced by the cloud infrastructure and pinpoint malicious service nodes in the cloud.

The RunTest system dynamically induces the weighted integrity attestation graph through randomized data attestation. When a tuple $d$ first enters the cloud, the portal node first sends the data to a pre-defined dataflow path $p_1 \rightarrow p_2 \ldots \rightarrow p_l$ providing functions $f_1 \rightarrow f_2 \ldots \rightarrow f_l$. After the portal receives the processing result for $d$, the portal may decide to perform integrity attestation with a certain probability $p_u$. The portal performs integrity attestation by sending a duplicate of $d$ to an alternative dataflow paths such as $p'_1 \rightarrow p'_2 \ldots \rightarrow p'_l$, where $p'_i$ provides the same data processing function $f_i$ as $p_i$. The portal may send multiple duplicates of $d$ to perform concurrent attestation. For distributed dataflow processing applications, we verify both intermediate and final processing results to pinpoint malicious processing nodes. In order to achieve non-repudiation, each service provider is required to keep the secure hash values of its input and output as evidence and sign the hash values with its private key [29].

We intentionally decouple the normal data processing phase from the attestation phase to prevent malicious attackers from detecting and escaping our attestation scheme. If we send attestation data together with the original data $d$, the malicious attacker can decide to cheat when it is sure that no attestation is triggered for $d$ or when all attestation data are sent to its
colluders. As a result, as long as the attestation is not employed for every tuple all the time, the malicious attackers can compromise the integrity of dataflow processing without being detected. In contrast, under our decoupled attestation scheme, the malicious attackers cannot avoid the risk of being detected when they produce false results on the original data $d$. Although the decoupled attestation scheme may cause delay in a single tuple processing, we can overlap the attestation and normal processing of consecutive tuples in the dataflow to hide the attestation delay from the cloud user.

After receiving the attestation results, the portal compares each intermediate result between pairs of functionally equivalent nodes $p_i$ and $p'_i$. If $p_i$ and $p'_i$ receive the same input data but produce different output results, we say that $p_i$ and $p'_i$ are inconsistent with regard to function $f_i$. Otherwise, we say that $p_i$ and $p'_i$ are consistent with regard to function $f_i$. For each pair, the portal maintains counters of consistencies and inconsistencies in order to compute the weight of the corresponding edge in the attestation graph. The portal updates the counters and the weights each time when it receives attestation data. The attestation graph captures aggregated attestation results over a period of time.

The RunTest system then performs comprehensive analysis over the attestation graphs to pinpoint malicious service providers and identify untruthful results. Particularly, we look for attestation graph patterns (e.g., number of cliques, weights of non-clique edges) to identify malicious service providers and their collusion behavior. Furthermore, our scheme can also give accurate estimation about the quality of data (QoD) provided by the cloud dataflow processing services.

3.2.2 Integrity Attestation Graph

In order to detect service integrity attack, we employ data attestation on data processing service nodes. In contrast to code attestation schemes (e.g., [38,67,73,74]), which often require special hardware/software to verify the code running on a system, data attestation verifies the integrity of service function by feeding same input data into redundant service nodes and comparing output data. The results of data attestation are in the form of consistency or inconsistency relationships, based on the comparison of output data on the same input data. We use the integrity attestation graph to aggregate individual data attestation results. Thus, we can achieve more efficient and scalable attack detection for dataflow processing in large-scale cloud infrastructures. We formally define the integrity attestation graph as follows,

\textbf{Definition 1:} An \textit{Integrity Attestation Graph} is a weighted undirected graph, with all the attested service providers (or cloud nodes) as the vertex set and consistency/inconsistency relationships as the edges. Each edge in the attestation graph is associated with a weight. The Weight on an attestation graph edge, denoted as $w$ ($0 \leq w \leq 1$), is the fraction of consistent
output out of all output data between two service providers given the same input data.

Thus, if \( w = 1 \), it means that the two service providers always agree with each other. If \( w = 0 \), it means the two service providers never agree with each other. In particular, we define consistency pairs and inconsistency pairs as follows,

**Definition 2**: A *Consistency Pair* is a pair of service providers, which always give the same output on the same input data. An *Inconsistency Pair* is a pair of service providers, which give at least one inconsistent output on the same input data. Thus, the edge between a consistency pair has weight \( w = 1 \). The edge between an inconsistency pair has weight \( w < 1 \).

Figure 3.1 and Figure 3.2 show an example dataflow application and its corresponding attestation graph. Note that in Figure 3.1, only a small subset of input data will undergo the attestation process. Each attestation process only involves a subset of service providers. However, over a period of time, all service providers will be attested for at least once. Hence, our scheme can detect malicious attacks to dataflow processing in the cloud without losing scalability. For example, in Figure 3.1, \( d_1 \) is selected for attestation, and the attestation data \( d'_1 \) is sent to \( p_2 \). Similarly, \( d_3 \) is selected for attestation and the attestation data \( d'_3 \) is sent to \( p_3 \). Thus, \( p_1, p_2 \) and \( p_3 \) are all attested through different tuples. Figure 3.2 shows the
generated attestation graphs. We use solid lines to represent consistency pairs, and dotted lines for inconsistency pairs in the attestation graph. For function \( f_1 \), the attestation graph contains one consistency pair \((p_1, p_2)\) as well as two inconsistency pairs \((p_1, p_3)\) and \((p_2, p_3)\). For function \( f_2 \), the attestation graph contains one consistency pair \((p_4, p_5)\) as well as two inconsistency pairs \((p_3, p_4)\) and \((p_3, p_5)\). Note that service provider \( p_3 \) provides both functions \( f_1 \) and \( f_2 \). We can also generate an integrated attestation graph by combining all per-function attestation graphs, as shown by the rightmost graph. In the rest of the section, we use attestation graph to refer to per-function attestation graph.

We propose to perform comprehensive analysis over the attestation graph to detect malicious service providers and identify untruthful data processing results. We would like to identify those representative attestation graph features, called *attestation graph motifs*, which can distinguish benign service providers from malicious attackers. Particularly, we observe that any two benign service providers always give the same results for the same input. In the attestation graph, this is expressed by an edge between them with weight one. We further define consistency clique as follows,

**Definition 3:** A *Consistency Clique* is a complete subgraph of an attestation graph such that
1) it has at least two nodes; 2) the weights of all edges are one (i.e., \( w = 1 \)); and 3) it is maximal, that is, it is not contained in any other complete subgraph where the weights of all edges are one.

If all pairs of service providers provisioning a particular data processing function have been attested, we can make the following proposition:

**Proposition 1:** All benign service providers always form a consistency clique in the integrity attestation graph.

**Proof:** we prove the above proposition by contradiction. Suppose there is a benign service node \( p_{i+1} \) that is not in the consistency clique formed by other benign service nodes \( p_1, p_2, ..., p_i \). Then there must be at least one weight 1 edge missing from \( p_{i+1} \) to one of the benign nodes in the clique, say \( p_j \) (\( 1 \leq j \leq i \)). Since pair \((p_{i+1}, p_j)\) has been attested together and they both give correct results, they must have given the same results on all the common tuples. Thus, there should be a solid line edge of weight 1 between them. This contradicts with the previous assumption that there is no weight 1 edge between \( p_{i+1} \) and \( p_j \).

### 3.2.3 Pinpointing Malicious Service Providers

Although our attestation scheme randomly attests a subset of all providers at a time, randomized attestation over a stream of data can cover all service providers over a period of time. In a large-scale cloud system, it is reasonable to assume that for any given data processing function,
the number of benign providers is larger than that of malicious ones. Suppose there are \( k \) nodes in the attestation graph. Since all benign service providers always form a clique in the attestation graph (Proposition 1), we can claim that, at any time, a benign node must be in at least one clique with size larger than \( \lfloor k/2 \rfloor \). Therefore, we can make the following proposition to pinpoint malicious nodes.

**Proposition 2:** Any node that is outside of all maximal cliques of size larger than \( \lfloor k/2 \rfloor \) in a per-function attestation graph must be a malicious node.

**Proof:** we prove the above proposition by contradiction. Suppose there is a benign node \( p_{i+1} \) that is not in any of the maximal cliques of size larger than \( \lfloor k/2 \rfloor \). According to Proposition 1, the benign node forms a clique with all other benign nodes. Since the number of benign nodes is larger than \( \lfloor k/2 \rfloor \), node \( p_{i+1} \) is in a clique of size larger than \( \lfloor k/2 \rfloor \). So, it is in a maximal clique of size larger than \( \lfloor k/2 \rfloor \). This contradicts with the assumption.

Initially, all nodes are treated as benign nodes and stay in a single consistency clique. As a malicious node keeps misbehaving, it will produce inconsistent results with that of benign nodes, which can be captured by the attestation. Thus, the malicious node will be excluded from the clique it stayed before. It either remains in a downsized clique or becomes an isolated node. When the malicious node is pushed out of any of the cliques with size larger than \( \lfloor k/2 \rfloor \), it will be pinpointed as malicious. When all malicious nodes are pinpointed, there will be only one clique with size larger than \( \lfloor k/2 \rfloor \) in the per-function integrity attestation graph, which is formed by all benign nodes. This clique is the maximum clique in the attestation graph. All other cliques, if there are any, should have size less than \( \lceil k/2 \rceil \).

Thus, pinpointing malicious nodes becomes the problem of finding consistency cliques in the attestation graph. We adapt the well-known Bron-Kerbosch (BK) clique finding algorithm \([18, 22, 53]\) for finding consistency cliques in the attestation graph. We maintain three disjoint sets of nodes \( R, P, \) and \( X \):

- The set \( R \) stands for the currently growing clique, i.e. the set to be extended by a new node or shrunk by one node on traveling along a branch of the backtracking tree, and \( R \) is initialized to be \( \emptyset \);
- The set \( P \) stands for prospective nodes which are connected to all nodes in \( R \) and using which \( R \) can be expanded, and \( P \) is initialized to contain all nodes;
- The set \( X \) contains nodes already processed, i.e. the nodes that were previously in \( P \) and hence all maximal cliques containing them have already been reported, and \( X \) is initialized to be \( \emptyset \). Maintaining set \( X \) is necessary is because \( X \) being empty is one of the conditions to claim the current \( R \) contains a maximal clique. If \( X \) is not empty, \( R \) may be extended by considering nodes in \( X \).

Note that all nodes that are connected to every node of \( R \) are either in \( P \) or \( X \). The algorithm
Figure 3.3: An example of finding maximal cliques.

runs as traversing the recursion tree by moving nodes from $P$ to $R$ and updating the $R$, $P$, $X$ sets recursively. A maximal clique is reported when both $P$ and $X$ are empty. The heuristic of the pivot selection is based on the identification and elimination of equal sub-trees appearing in different branches of the recursion tree which lead to the formation of non-maximal cliques.

We use an example to explain the algorithm shown by Figure 3.3. There are three benign nodes $\{p_1, p_2, p_3\}$ and two malicious nodes $\{p_4, p_5\}$ in the attestation graph. Based on our consistency clique definition, we only count consistency links as connections between any two nodes. Initially, the current clique node set $R = \emptyset$, and the candidate set $P = \{p_1, p_2, p_3, p_4, p_5\}$. We randomly take one of the nodes in $P$, say $p_4$, as the pivot node. Thus, we can move those nodes that are not the neighbors of $p_4$ from $P$ to $R$. In this example, we can move the candidate nodes $p_1, p_2, p_3$ but not $p_5$ since $p_5$ is the neighbor of the pivot node $p_4$. Suppose we first move $p_1$ to $R$. Then, $R = \{p_1\}$, and we update the candidate set $P$ to include the nodes that are connected to all node(s) in $R$, which should be $P = \{p_2, p_3\}$. $X$ is the set containing already processed nodes with regard to the node currently under consideration. So the update of $X$ is to shrink it so that it only contains nodes that have connections with $p_1$. In this case, $X$ is $\emptyset$ and does not need to be shrunk. Next, we use the new $R, P, X$ in recursion to explore a maximal clique. We move another node $p_2$ from $P$ to $R$ to expand the clique, then updating $R, P$, and $X$ into $R = \{p_1, p_2\}$, $P = \{p_3\}$, and $X = \emptyset$. Similarly, we move $p_3$ from $P$ to $R$, and have $R = \{p_1, p_2, p_3\}$, $P = \emptyset$, and $X = \emptyset$. By now, we have identified a clique $R = \{p_1, p_2, p_3\}$ since both $P$ and $X$ become empty. After returning from the recursions, $p_3, p_2, p_1$ will all be added to $X$ respectively, since they are processed nodes. Note that the usefulness of $X$ would be clearer when there are other nodes connected to only a part of the currently forming clique, which is not presented in this simple example.

Now, we start from a different pivot node, say $p_1$. We have $R = \emptyset$, $P = \{p_1, p_2, p_3, p_4, p_5\}$, $X = \{p_1, p_2, p_3\}$. We can move those nodes that are not the neighbors of $p_1$ from $P$ to $R$, which include $p_4$ and $p_5$. Suppose we first move $p_4$ to $R$ and then update $R, P$, and $X$ into $R = \{p_4\}$, $P = \{p_5\}$, and $X = \emptyset$. Next, we can move $p_5$ from $P$ to $R$ since it is not the neighbor of the pivot $p_1$. Thus, we have $R = \{p_4, p_5\}$, $P = \emptyset$,
and $X = \emptyset$, so that we identify another clique \{p_4, p_5\}.

Generally, a maximal clique is a complete subgraph that is not contained in any other complete subgraph. Among all cliques, the largest one is the maximum clique. The maximum clique problem is one of the canonical NP-complete problems, while the problem of enumerating all cliques in a graph is NP-hard. The complexity of the BK algorithm increases with the number of cliques in the graph. However, in practice, the number of cliques in the attestation graph is very small. Furthermore, RunTest adds two requirements for cliques: 1) The clique contains at least two nodes; and 2) weights of all edges are one. These two features can help us eliminate some nodes that do not satisfy the criteria in $O(n + e)$ time, with $e$ being number of edges. Thus, we extend the BK algorithm by first reducing the attestation graph through eliminating a subset of nodes. Nodes without edges of weight 1 cannot be in a clique. By going through each node in the graph at the beginning, such type of nodes can be eliminated, and the complexity of the BK algorithm is reduced. Figure 3.4 shows the pseudo-code of our consistency clique finding algorithm. The algorithm takes the attestation graph $G$ as input, and returns all maximal cliques.

### 3.2.4 Identifying Attack Patterns

A large-scale cloud computing infrastructure often consists of many distributed service provisioning nodes. When multiple malicious attackers collude or multiple cloud nodes are compromised and controlled by a single malicious attacker, the cloud infrastructure will be exposed to collusive integrity attacks. Thus, in addition to pinpointing those malicious nodes, it is also important to discover colluding behavior among different nodes, which can assist in cloud security diagnosis. A nice feature of our integrity attestation graph is that it can not only expose malicious nodes but also reveal collusive attack patterns. We summarize the features of the corresponding attestation graphs for each attack pattern, and express them using attestation graph motifs. When all the possible inconsistency relationships are exposed by the attestation, the attestation graph motifs of all possible integrity attack patterns can be described as follows, which are shown in Figures 3.5 - 3.8. Note that we begin to analyze attack patterns after we find only one clique of size larger than $\lfloor k/2 \rfloor$.

**Attestation graph motif A**: If malicious nodes are not in collusion and always misbehave, the portal would find different results from different malicious nodes. In the attestation graph, malicious nodes would be isolated nodes in terms of consistency relationship. And in terms of inconsistency relationship, they are the nodes that have the maximal degrees. In other words, the weights on all the edges involving malicious nodes are all zero. Figure 3.5 shows an example of attestation graph motif A. This scenario corresponds to the Non-Collusion Always Misbehave (NCAM) attack pattern.
AdaptiveBK($G$)
1. Initialization 1: Mark any two nodes with $w < 1$ edge as unconnected, and with $w = 1$ edge as connected;
2. Initialization 2: Eliminate nodes that do not have any edge of $w = 1$
3. FindConsistencyClique($\emptyset, V(G), \emptyset$), where $V(G)$ is the node set of $G$

FindConsistencyClique($R, P, X$)
1. if ($P == \emptyset$ and $X == \emptyset$ and size of $R > 1$)
2. Report $R$ as a maximal clique
3. else
4. Let $u_p$ be the pivot node
5. Assume $P = u_1, u_2, ..., u_k$
6. for $i = 1$ to $k$
7. if $u_i$ is not a neighbor of $u_p$
8. $P = P - u_i$
9. $R_{new} = R \cup u_i$
10. $P_{new} = P \cap N[u_i]$, where $N[u_i]$ is neighbor set of $u_i$
11. $X_{new} = X \cap N[u_i]$
12. FindConsistencyClique($R_{new}, P_{new}, X_{new}$)
13. $X = X \cup u_i$

Figure 3.4: Consistency clique discovery algorithm.

**Attestation graph motif B:** If malicious nodes misbehave probabilistically and independently, they would agree with benign nodes and even well-behaved malicious nodes on some attestation data. This scenario corresponds to the Non-Collusion Probabilistically Misbehave (NCPM) attack pattern. Alternatively, malicious nodes may collude sometimes and act independently for the rest of time, which corresponds to Partial Time Partial Collusion (PTPC) attack pattern. In both cases, the weights on the attestation graph edges involving malicious nodes appear to be randomized. Figure 3.6 shows an example of the attestation graph motif B.

**Attestation graph motif C:** If all malicious nodes collude together and always give the same incorrect results, malicious nodes also form a consistency clique among themselves in the attestation graph. However, the weights on the edges involving one malicious node and one benign node are all zero. Figure 3.7 shows an example the attestation graph motif C. This attestation graph motif can capture the Full Time Full Collusion (FTFC) attack pattern.
**Attestation graph motif D:** If all malicious nodes collude together but only on selected tuples based on a pre-agreement, malicious nodes also form a clique. Each malicious node connects to benign nodes with the same weight of $w$. Figure 3.8 shows an example attestation graph motif D. This scenario corresponds to the Partial Time Full Collusion (PTFC) attack pattern.

From the attestation graph motifs, it can be observed that different attack patterns can be distinguished according to two criteria: (a) number of cliques; and (b) the weight patterns of non-clique edges. Thus, we can identify attack patterns using the following rules: (1) Depending on number of cliques in an attestation graph, we can divide the attack patterns into two groups. One group includes NCAM, NCPM, and PTPC, which contains only one clique. And the other group includes FTFC and PTFC. (2) Depending on the weights on non-clique edges, we can further identify individual attack patterns within each group. For example, in the first group, if weights are all zero, the attack can only be NCAM. However, in the second group, if weights are all zero, attack pattern is FTFC. Note that the algorithm cannot distinguish NCPM and PTPC, since their attestation graphs have the same features. This is acceptable because NCPM is a special case of PTPC, where the parameter controlling collusion goes from sometimes to
Figure 3.7: Attestation graph motif C.

Figure 3.8: Attestation graph motif D.

Figure 3.9 shows the pseudo code of our algorithm to identify attack patterns and malicious service nodes. The algorithm takes the attestation graph $G$ as input, and outputs suspected malicious nodes. First, the algorithm finds all cliques in the attestation graph using the adapted BK algorithm. Second, it checks nodes against the identified cliques to pinpoint malicious service nodes. Finally, when there is only one clique of size larger than half of the total number of nodes, the algorithm identifies attack patterns. By checking the number of cliques in the attestation graph and the weights on non-clique edges one by one, the algorithm identifies specific attack patterns. Note that the algorithm needs to be started when all pairs of functionally equivalent nodes have been attested in order to assure that all benign nodes have showed up in the maximum clique.
IdentifyMaliciousNodes($G$)
1. find all maximal cliques $CL_i$ ($1 \leq i < k$) in $G$ using adapted Bron-Kerbosch algorithm with pivot selection
2. in $CL_i$, find those maximal cliques with size larger than $\lfloor k/2 \rfloor$, $CL_b$ ($b \leq i$), where $k$ is the total number of nodes in $G$
3. check all nodes $Node_j$ in $G$ against nodes in all $CL_b$
4. if ($Node_j$ is not in any of $CL_b$)
5. $Node_j$ is malicious
6. if (only one maximal clique in $CL_b$, i.e., the maximum clique)
7. if (numCliques == 1)
8. nodes in the clique is identified as $N_i$
9. if (weights of edges from nodes in $N_i$ to rest of nodes not in $N_i$ are all 0s)
10. attack model NCAM
11. else
12. attack model NCPM or PTPC
13. if (numCliques $\geq$ 2)
14. nodes in the maximum clique are $N_i$, in the rest cliques are $N_{1i}$, $N_{2i}$, ...
15. check each clique other than the maximum clique
16. if (weights from $N_i$ to any of $N_{1i}$, $N_{2i}$, ... are all 0s)
17. attack model FTFC
18. else if (all links between $N_i$ and $N_{ji}$ have same weight)
19. attack model PTFC

Figure 3.9: Cloud Dataflow Integrity attack detection algorithm.

3.3 Security Analysis
3.3.1 Security Properties

Our scheme of pinpointing malicious service providers is based on attestation graph analysis. We claim that the scheme preserves the following properties:

Property 1: No false positive: a benign service provider will not be pinpointed as malicious.

Proof: As proved for our Proposition 2, a benign node always stays in at least one clique of size larger than $\lfloor k/2 \rfloor$. Therefore, a benign node will not be treated as malicious since our algorithm only pinpoints a node when the node is outside of any maximal cliques with size larger than $\lfloor k/2 \rfloor$.

Property 2: Non-Repudiation: for any pinpointed malicious service provider, the trusted
portal node can present evidence to prove it is malicious.

**Proof:** The data received by the portal contains the signed hash value of intermediate processing results provided by service providers. The portal can present proof that shows the pinpointed node is inconsistent with more than \( \lfloor k/2 \rfloor \) of its functionally equivalent nodes. According to our algorithm, a pinpointed node is not in any of the maximal cliques with size larger than \( \lfloor k/2 \rfloor \), which means it has inconsistent results with more than \( \lfloor k/2 \rfloor \) nodes. Since the number of malicious nodes is less than or equal to \( \lfloor k/2 \rfloor \), the pinpointed node must have been inconsistent with a benign node. Thus, it must be malicious.

Note that our scheme has false negative, since the randomized data attestation cannot capture the misbehavior of malicious service providers if they accidentally only misbehave on non-duplicated tuples or all our attestation are conducted on collusive service providers. However, since our scheme ensures that malicious nodes cannot predict when they will be attested, the probability of detecting misbehavior through observing inconsistent data results will increase accordingly as more attestations are performed. Thus, our runtime attestation scheme can identify all misbehaving malicious nodes as data tuples continuously flow into the system and the inconsistency links are added to push the nodes outside of any cliques of size larger than \( \lfloor k/2 \rfloor \).

### 3.3.2 Data Quality

We define data quality as the percentage of processed data with correct results. Our scheme can detect tampered data results probabilistically and report data quality close to actual data quality.

Suppose the total number of unique data is \( n \), and the number of tampered unique data is \( s \). Thus, the actual data quality, denoted by \( Q_a \), equals to \( 1 - s/n \). In contrast, our reported data quality, denoted by \( Q_r \), can be computed as the percentage of data that we believe has correct results. For non-duplicated data, we count them as non-tampered since we are not sure whether it is tampered. For duplicated data, we also count it as non-tampered if we get consistent results. If we get inconsistent results, we count it as tampered. The expected number of unique data that gets duplicated for attestation is \( np_u \). We suppose the number of data tuples that gets duplicated but with inconsistent results is \( c \). Thus, the reported data quality becomes \((n - np_u) + (np_u - c))/n\), which turns out to be \( 1 - c/n \). That is,

\[
Q_r = 1 - \frac{c}{n} \quad \tag{3.1}
\]

where \( c \) can be observed and recorded by the portal through data attestation. It can be seen that \( c < s \). The data quality reporting of our scheme has false negatives but does not have false
positives.

We can introduce auto-correction mechanism to correct tampered data results. By checking visited components against the pinpointed malicious component set, we can either select data result from existing results processed by only benign components or process the data again through only benign components. After pinpointing a malicious component, we can also eliminate it from the system and thus improve data quality extensively.

3.3.3 Detection Probability

We quantify the detection capability of our randomized service integrity attestation scheme. We define detection probability $P_d$ as $1 - P_e$, where $P_e$ is the escaping rate denoting the probability that malicious components can escape from being detected within a time period $T$ because the final results of all testing data are consistent. There are two dimensions of factors that affect the escaping rate.

First, a malicious service component may not misbehave all the time. It can escape detection if not cheating on any of the testing data. Suppose a malicious service component misbehaves on a data item with a fixed probability $p_m$. With $x$ testing data a malicious component may receive during a time period $T$, the service component gives false result on none of the testing data items with a probability of $(1 - p_m)^x$. This shows that increasing testing data set size reduces escaping rate and helps expose those untruthful service components that selectively misbehave. Second, whether malicious components collude with each other is another factor. Suppose out of $k$ components at most $m$ are malicious, and $r$ components are selected for redundant processing. Also suppose $m \geq r$. Otherwise, misbehavior can be detected no matter malicious components collude or not. If there is no collusion, we can naturally assume different misbehaving components produce different results on the same input data. Thus, a malicious component can escape from being detected if and only if it behaves benignly on all testing data. The average number of testing data a component receives is $rl/k$. Therefore, the average escaping rate of a single malicious component is $(1 - p_m)^{rl/k}$. However, in the worst case, where all $m$ components are in collusion, a malicious component may cheat on a data item as long as it sees other malicious component(s) receiving the same data item cheated. We define $P$ as the probability that malicious components are not detected on a single testing data, then escaping rate $P_e$ is $(P)^l$, given a set of $l$ testing data.

To compute $P$, we suppose for a single testing data, when having $i$ malicious ones out of the selected $r$ components, the probability of malicious components not being detected is $P_i$. If $k - m \geq r$, $i$ ranges from 0 to $r$. When $i = 0$, obviously, malicious components cannot be detected. When $i = r$, since malicious components work in collusion, they can escape no matter they misbehave or not. However, when $0 < i < r$, they need to behave benignly to escape.
Note that once any of the $i$ malicious components first decides to misbehave on a data item, the rest $i-1$ malicious components have no choice but to follow the decision. Thus, $P$ equals to

$$P = \sum_{i=0}^{r} P_i = \binom{k-m}{r} + \sum_{i=1}^{r-1} \binom{m}{i} \binom{k-m}{r-i} (1-p_m) + \binom{m}{r}$$

(3.2)

If $k - m < r$, $i$ ranges from $r - (k - m)$ to $r$. Then, $P$ equals to

$$P = \sum_{i=r-k+m}^{r} P_i = \sum_{i=r-k+m}^{r-1} \binom{m}{i} \binom{k-m}{r-i} (1-p_m) + \binom{m}{r}$$

(3.3)

The number of candidate components $k$, the size of testing data set $l$ and the redundancy degree $r$ denote the tradeoff between the overhead and the escaping rate. Intuitively, the larger the duplicated component number and testing data size, the more likely the portal can catch service integrity attacks.

### 3.4 Implementation and Evaluation

#### 3.4.1 System Implementation

We have implemented our runtime integrity attestation scheme in C++ within IBM System S stream processing system [36]. The stream processing system is a high-performance continuous stream processing platform that can handle high-rate data streams and scale to hundreds of processing components. Our experiment is conducted on NCSU virtual computing lab [6] consisting of several hundreds of blade servers, which provides similar virtual cloud resources as Amazon EC2 [1]. We use about 10 blade servers in our experiments. Each host runs CentOS 5.2 64-bit with Xen 3.0.3 for managing virtual machines.

The dataflow processing application we use is extracted from the sample application provided by the IBM System S stream processing system, illustrated by Figure 3.10. The application takes real data about weather information as input, performs conversions and calculations on the weather data, and generates the most recent weather information for different fruit suppliers. We perform attestation on three functions. Each function is provisioned by five different
service providers. We have a trusted portal node that accepts a stream of application data tuples, launches randomized data attestation, and collects attestation results. For each function, the portal constructs an attestation graph for pinpointing malicious service providers and identifying attack patterns. The input data rate is 100 tuples per second. Figure 3.11 shows the service instances deployed in the system. Note that the service instances are distributed on 10 different hosts. The System S controls which host each service instance is assigned to.

For comparison, we have also implemented the common consensus-based integrity verification scheme *full-time majority voting* that employs all functionally redundant service providers all the time to cross validate each other. The scheme determines which service providers are malicious through majority voting. It can detect any service integrity attack behavior immediately when receiving inconsistent results with the assumption that the number of benign service providers is larger than that of malicious ones. Our scheme relies on the same assumption to pinpoint malicious service providers.

We evaluate our scheme using two important metrics: *detection rate* and *attestation overhead*. The detection rate is calculated by the number of pinpointed malicious service providers
over the total number of malicious service providers that have misbehaved at least once during one experiment run. The attestation overhead is calculated by the number of duplicated data tuples that are redundantly processed for integrity attestation and the extra dataflow processing time incurred by the integrity attestation.

We evaluate the proposed scheme in three steps: 1) we evaluate the effectiveness of our service integrity verification scheme in terms of detecting malicious service providers under different attack strategies; 2) we investigate the sensitivity of our scheme to system parameter variations; 3) we compare our scheme with the full-time majority voting scheme. We show that our scheme can achieve similar attack detection performance as the full-time majority voting scheme while imposing much lower overhead.

3.4.2 Results and Analysis

We first evaluate the attack detection effectiveness of our scheme. We vary the percentage of malicious service providers from 20% to 40% but guarantee that for each function, the number of benign service providers is larger than that of malicious ones. After the portal receives the processing result of a new data tuple, it randomly decides whether to perform data attestation. Each tuple has 0.2 probability of getting attested (i.e., attestation probability $p_u = 0.2$), and only one attestation data is used (i.e., number of total data copies $r = 2$). Figure 3.12 through Figure 3.16 show the detection rate under different integrity attack scenarios. For each attack scenario, we plot the detection rates in the presence of different percentage of malicious service providers. The X axis shows the number of total attestation data used up to that point. We observe that our scheme can detect all malicious service providers using a small number of attestation data. Generally speaking, we need to use more attestation data to detect all malicious service providers under collusion attack patterns.

For non-collusion attacks, we test $(b_i = 1, c_i = 0)$ and $(0 < b_i < 1, c_i = 0)$ scenarios respectively. Our algorithm can correctly identify $(b_i = 1, c_i = 0)$ as NCAM attack pattern, and identify $(0 < b_i < 1, c_i = 0)$ as either NCPM or PTPC attack pattern. Note that our algorithm cannot distinguish NCPM and PTPC because they share the same attestation graph motif. Figure 3.12 is the NCAM scenario, where malicious service providers misbehave all the time and independently. As we can see, the detection rate reaches 100% earlier when there are less malicious service providers. However, the intermediate detection rate depends on the time when malicious service providers begin to misbehave as well as the time that misbehavior is captured through attestation. Figure 3.13 shows the NCPM scenario, where malicious service providers misbehave independently but probabilistically. They have 0.2 probability of misbehaving on a tuple (misbehaving probability $b_i = 0.2$). Compared with NCAM, the NCPM scenario needs longer time to reach 100% detection rate. This is because malicious service providers misbehave
probabilistically, which makes it harder to catch their malicious behavior immediately.

For collusion attacks, we test \((b_i = 1, c_i = 1)\), \((0 < b_i < 1, c_i = 1)\), and \((0 < b_i < 1, 0 < c_i < 1)\) scenarios respectively. Our algorithm correctly identifies \((b_i = 1, c_i = 1)\) as FTFC, \((0 < b_i < 1, c_i = 1)\) as PTFC, and \((0 < b_i < 1, 0 < c_i < 1)\) as either PTPC or NCPM. Figure 3.14 shows the FTFC scenario, where all malicious service providers form a group and launch group colluding attacks all the time. They give the same output on a data tuple. Even though our scheme randomly selects service providers for attestation, it can still achieve 100% detection rate under such colluding attacks. Figure 3.15 shows a more intelligent attack scenario of PTFC, where all malicious service providers are in collusion and they have pre-agreement specifying on which tuples to misbehave. They only misbehave on tuples with even-numbered
sequence number, so \( b_i = 0.5 \). Figure 3.16 is the PTPC scenario, where attackers may form collusion sometimes and act independently otherwise. In the experiments, each attacker relies on a random number generator to decide whether to collude. And if they collude, they give the same results on the same input data. Our scheme can still capture such attacks and pinpoint malicious service components under such scenario.

Secondly, we investigate the impact of system parameters, such as attestation probability and malicious service providers misbehaving probability, on the effectiveness of our algorithm. We fix the percentage of malicious service providers at 20%. Figure 3.17 shows the detection rate under different attestation probability in \((b_i = 1, c_i = 1)\) scenario. By increasing attestation probability, attestation traffic has more opportunities to cover malicious service providers and capture their misbehavior. Thus, our scheme can reach a higher detection rate earlier. However,
the system overhead, in terms of attestation traffic would increase accordingly since we duplicate more tuples for attestation. Figure 3.18 shows the detection rate under different misbehaving probability of malicious service providers in \((0 < b_i < 1, c_i = 0)\) scenario. As we can see, the more frequently malicious service providers misbehave, the less attestation traffic and less time we use to detect them. In other words, our scheme forces attackers to slow down and reduce their attacks. Note that even with low misbehaving probability, e.g. 0.2, our scheme can pinpoint those attackers with limited attestation traffic. As long as attackers keep misbehaving, no matter how infrequently, our scheme can detect them when the inconsistency links push the malicious nodes outside of all cliques of size larger than \(\lfloor k/2 \rfloor\), where \(k\) is the total number of service providers serving the function.

We also measure the computation time of the clique discovery algorithm. Figure 3.19 shows
Figure 3.18: Detection rate under different misbehaving probability in \((0 < b_i < 1, c_i = 0)\) scenario. (attestation probability = 0.2)

Figure 3.19: Computing time of clique discovery algorithm.

the total clique enumeration time of attestation graphs with 1, 2, till 5 cliques. We test it with different number of service providers in the system. It shows that given a small number of cliques, the number of cliques in the graph does not have much impact on clique enumeration time. Even with 100 service providers serving the same function, clique enumeration time measurements are within two milliseconds.

We compare actual data quality with reported data quality to evaluate whether our scheme can accurately capture data quality. Actual data quality is defined as the percentage of processed data with correct results. The reported data quality is calculated using equation 3.1 by the portal. Figure 3.20 shows the data quality comparison under different attack patterns. It shows that for each scenario, our scheme can give a very close estimate of actual data quality.

We compare the no attestation and with the attestation \((r = 2)\) schemes in terms of the
average dataflow processing delay. The delay is measured by the average per-tuple turnaround time (i.e., the duration between the time when the first dataflow tuple enters the system and the time when the last dataflow tuple leaves the system over the total number of tuples processed). Figure 3.21 shows the comparison in cases of different data rates. We can see that our scheme had little delay overhead compared with the no attestation scheme.

Thirdly, we compare our scheme with a common existing scheme, full-time majority voting. Figure 3.22 compares the detection time of our scheme with the full-time majority voting scheme in the \((b_i = 1, c_i = 0)\) attack scenario (identified as NCAM). Our scheme was run with different attestation probabilities. Each time, two functionally equivalent service providers are selected, while in the full-time majority voting scheme, it sends duplicated tuples to all the five service components. The full-time majority voting scheme has the shortest detection time. Our
scheme can achieve 100% detection with a short delay, which can be shortened by increasing the duplication probability. Note that such a short delay is acceptable by dataflow processing applications, where we can mark a set of result data as tentative and commit the final correct results after the attestation [14]. Figure 3.23 compares the attestation overhead in terms of number of redundant tuple duplicates processed. Our scheme outperforms the naive full-time majority voting scheme with the limited overhead at the expense of only a little delay toward detecting all malicious service providers.

3.5 Summary

In this chapter, we have presented the design and implementation of RunTest, a service integrity attestation system for verifying the integrity of dataflow processing in multi-tenant cloud in-
frastructures. RunTest employs application-level randomized data attestation for pinpointing malicious dataflow processing service providers in large-scale cloud infrastructures. We propose a new integrity attestation graph model to capture aggregated data processing integrity attestation results. By analyzing the integrity attestation graph, RunTest can i) pinpoint malicious service providers, ii) identify untruthful data processing results, and iii) discovering colluding attack patterns in the large-scale cloud infrastructure. We have implemented our service integrity attestation scheme within IBM System S dataflow processing system and tested it on NCSU virtual computing lab. Our initial experimental results show that the proposed scheme is effective and imposes a low performance impact for dataflow processing in cloud infrastructures.
Chapter 4

IntTest: Integrated Service Integrity Attestation Scheme for Large-Scale Cloud Systems

In this chapter, we present the design, implementation and evaluation of IntTest, a novel integrated service integrity attestation scheme that can efficiently verify the integrity of dataflow processing services and quickly pinpoint malicious service providers within a large-scale cloud infrastructure.

IntTest is built on top of our previous work RunTest [30], which employs probabilistic replay-based consistency check to detect malicious service providers. RunTest can achieve the same detection accuracy with full time majority voting based schemes but with much less overhead. However, RunTest needs to assume that benign service providers always take majority in every service function. In a large-scale cloud system, multiple malicious attackers may launch colluding attacks to invalidate the assumption. When attackers always give consistent wrong results and form a majority clique in a specific service function, they may trick the RunTest system to label benign service providers as malicious, yielding inaccurate integrity check results. To address the challenge, IntTest takes a holistic approach by examining all consistency and inconsistency relationships among different service providers within the whole cloud system.

IntTest performs comprehensive integrity attestation by examining both per-function consistency graphs and the global inconsistency graph. The per-function consistency graph analysis scheme can limit the scope of damage caused by colluding attackers since they have to form majority in every attacked service function to escape our attestation scheme. The global inconsistency graph analysis scheme can effectively expose those attackers that attack many different service functions. Thus, by intelligently combining the attestation results from both consistency and inconsistency graphs, IntTest can mitigate false alarms and detect more malicious compo-
ments than previous schemes that only examine individual service function separately.

In the following, we first give our assumptions and then describe the design details, followed by implementation and experimental results.

4.1 Assumptions

We first assume that the total number of malicious service components is less than that of benign ones in the entire cloud system. Without this assumption, it would be very hard, if not totally impossible, for any attack detection scheme to work, given no reference service components or comparable ground truth processing results. But we do not assume benign service components have to be the majority for every service function as in RunTest [30].

A malicious attacker can pretend to be a legitimate service provider or take control of vulnerable service providers to provide untruthful dataflow processing results. Malicious attackers can be stealthy, which means they can misbehave on a selective subset of input data or service functions while pretending to be benign service providers on other input data or functions. The stealthy behavior makes detection more challenging due to the following reasons: 1) the detection scheme needs to be hidden from the attackers to prevent attackers from gaining knowledge on the set of data processing results that will be verified and therefore easily escaping detection; 2) the detection scheme needs to be scalable while being able to capture misbehavior that may be both unpredictable and occasional. We assume that result inconsistency caused by hardware or software faults can be marked by some fault detection schemes [42,43] and are excluded from our malicious attack detection.

In a large-scale cloud system, we need to consider colluding attack scenarios where multiple malicious attackers collude or multiple service sites are simultaneously compromised and controlled by a single malicious attacker. Attackers could sporadically collude, which means an attacker can collude with an arbitrary subset of its colluders at any time. We assume that malicious nodes have no knowledge of other nodes except those they interact with for data receiving and forwarding. However, attackers can communicate with their colluders in an arbitrary way. Attackers can also change their attacking and colluding strategies arbitrarily.

4.2 Design and Algorithms

Our algorithm pinpoints malicious service providers based on the consistency / inconsistency relationships between service providers. It includes three parts: 1) A runtime attestation scheme, called replay-based consistency check, to derive the consistency / inconsistency relationships between functionally equivalent service providers; 2) Attestation Graph Model model that comprises of both consistency graphs and the integrated inconsistency graph to aggregate attestation
results; and 3) Attestation graph-based pinpointing algorithm that takes the attestation graphs as input and outputs malicious service providers.

In this section, we first give an overview of the replay-based consistency check scheme. Then we present how the replay-based consistency check scheme supports stateful service attestation. We then introduce the integrity attestation graph model that comprises of both consistency graphs and the integrated inconsistency graph. Then we describe our algorithms that employ the per-function consistency graphs and the integrated inconsistency graph to pinpoint malicious service providers in large-scale cloud infrastructures.

4.2.1 Replay-based Consistency Check

The basic idea is to feed same input data into functionally equivalent service components and compare output results to find out consistency/inconsistency relationships between service providers. Two service providers have consistency relationship if they always give consistent output results on all input data, or have inconsistency relationship if they give inconsistent outputs on at least one input data. Result consistency is defined as either result equality, or the distance between the results according to some distance function falling within a threshold. Note that we perform integrity attestation by replaying a subset of original input data at a later time. Thus, the malicious attackers cannot avoid the risk of being detected when they produce false results on the original data.

4.2.2 Stateful Service Attestation

A dataflow processing service is often provided through a set of data processing functions. The data processing functions could be either stateless or stateful. A service function is stateless if the processing result for one input data item does not depend on the state of the service function (e.g., previously received data). Streaming operators selection and projection are typical stateless functions. In contrast, if the processing result for one input data item does depend on the state of the service function, then the function is stateful. Window based streaming operators, such as aggregation and join, are examples of stateful functions. Stateful functions maintain their states during runtime. The state is the data that must be remembered by the operator to complete its operation. For window-based streaming operators, the state is a window of tuples. For example, in Figure 4.1, when a new tuple $d_4$ arrives, some old tuples (e.g. $d_1$) get purged out of the window depending on the types of windows. The operator then processes the new window $d_4, d_3, d_2$ as a unit and generates the sum of the tuples as output, i.e., $f(d_4) = d_4 + d_3 + d_2$.

For stateless functions, given the same input data, two benign service providers will always return consistent output results independent of the input data they have processed before.
However, it is challenging to perform replay-based consistency check on stateful functions, such as windowed aggregation and join [36]. Even if the same input data is used for attestation, two benign service providers at different states may produce different results. Thus, for stateful functions, both input data and the states have to be replayed. We propose two methods to attest stateful functions. One method is called indirect state recovery, which relies on replaying a sequence of historic input data to indirectly bring back the state. For window based streaming operators, we can record the data tuples sent to one service provider and resend them to another service provider to form the exact same window. The other method is called difference check, which derives consistency relationship between two stateful service components by comparing result difference produced by two consecutive input data. For example, the state can be a counter to count the number of received tuples. Even though the counters of two service components may have different values, they both will increase by one when accepting a new input data.

Here we focus on window-based stateful service functions, which is adopted by IBM System S [36]. Since the state of a service component is a window of tuples, in order to perform consistency check, we need to replay the window of tuples on other service components. We assume the window size is known to the portal node. We adopt a master-slave mechanism for data attestation. Given a set of functionally equivalent service providers, we randomly designate a service provider as the master and the rest as slaves. The portal node sends all data tuples to the master, and sends only attestation data to the slaves. The portal randomly selects a subset of the tuples for attestation. Specifically, when receiving a data tuple from the user, the portal decides to perform attestation on this tuple with a probability $P_u$. As Figure 4.2 shows, if data tuple $d_6$ is selected for attestation, the portal buffers a window size $w$ ($w = 3$) of continuous tuples $d_6, d_5, d_4$, which serve as the state on the master service provider $p_1$. After the portal receives the processing result of $d_6$ from the master, it randomly selects one slave service providers, $p_2$, for attestation. The portal sends a window of buffered tuples that are not present at $p_2$, $d_6, d_5$ as attestation data to $p_2$. Results from the master and the slave service provider $p_2$ can be then compared to detect any inconsistency.

Note that when window size is very large, replaying an entire window of tuples may degrade
dataflow processing performance. By predicting the window status and sending only the missing tuples, the portal can reduce attestation traffic and delay overhead. In order to predict the window status of a slave service provider, the portal needs to remember the sequence number $q$ of the last tuple sent to the service provider. Since the portal only sends windows of consecutive tuples as attestation data to slave service providers, the portal can conclude that the current window on the service provider has sequence numbers $[q, ..., q - w + 1]$. Thus, if the new attestation window has overlaps with the previous one, the portal only needs to fill in the missing tuples.

Note that the portal periodically switches master service provider and slave service providers. Suppose the portal decides to make the switch when it receives $d_i$ from the user. The portal then sends a window of tuples starting from $d_i$ to both the current master and a randomly selected slave. The window of tuples include $d_i + w - 1, ..., d_{i+1}, d_i$. The randomly selected slave becomes the new master. All the tuples after $d_{i+w-1}$ will be sent to the new master only. Since an entire window is sent to both the old master and the new master, the scheme makes a smooth switch without losing any data.

### 4.2.3 Attestation Graph Model

With replay-based consistency check, we can test functionally equivalent service providers and obtain their consistency and inconsistency relationships. We employ both consistency graph and inconsistency graph to aggregate pair-wise attestation results for further analysis. The graphs reflect the consistency/inconsistency relationships across multiple service providers over a period of time. Before introducing the attestation graphs, we first define consistency links and inconsistency links.

**Definition 2**: A consistency link exists between two service providers who always give consis-
tent output for the same input data during attestation. An inconsistency link exists between two service providers who give at least one inconsistent output for the same input data during attestation.

We then construct consistency graphs for each function to capture consistency relationships among the service providers provisioning the same function. Figure 4.3 shows the consistency graphs for two functions. Note that two service providers that are consistent for one function are not necessarily consistent for another function. This is the reason why we confine consistency graphs within individual functions.

**Definition 3**: A per-function consistency graph is an undirected graph, with all the attested service providers that provide the same service function as the vertices and consistency links as the edges.

We use a global inconsistency graph to capture inconsistency relationships among all service providers. Two service providers are said to be inconsistent as long as they disagree in any function. Thus, we can derive more comprehensive inconsistency relationships by integrating inconsistency links across functions. Figure 4.4 shows an example of the global inconsistency graph whose corresponding consistency graphs are illustrated by Figure 4.3. Note that service provider $p_2$ provides both functions $f_1$ and $f_2$. In the inconsistency graph, there is a single node $p_2$ with its links reflecting inconsistency relationships in both functions $f_1$ and $f_2$.

**Definition 4**: The global inconsistency graph is an undirected graph, with all the attested service providers in the system as the vertex set and inconsistency links as the edges.

The portal node is responsible for constructing and maintaining both per-function consistency graphs and the global inconsistency graph. In order to generate these graphs, the portal maintains counters for the number of consistency results and counters for the total number of
attestation data between each pair of service providers. The portal updates the counters each time when it receives attestation results. At any time, if the counter for consistency results has the same value with that for the total attestation data, there is a consistency link between this pair of service providers. Otherwise, there is an inconsistency link between them.

4.2.4 Consistency Graph Analysis

We first examine per-function consistency graphs to pinpoint suspicious service providers. The consistency links in per-function consistency graphs can tell which set of service providers keep consistent with each other on a specific service function. Given any service function, since benign service providers always keep consistent with each other, benign service providers will form a clique in terms of consistency links. For example, in Figure 4.3, \( p_1, p_3 \) and \( p_4 \) are benign service providers and they always form a consistency clique.

In our previous work [30], we have developed a clique-based algorithm to pinpoint malicious service providers. If we assume that the number of benign service providers is larger than that of malicious ones, a benign node will always stay in a clique formed by all benign nodes, which has size larger than \( \lceil k/2 \rceil \). Thus, we can pinpoint malicious nodes by identifying nodes that are outside of all maximal cliques of size larger than \( \lceil k/2 \rceil \). For example, in Figure 4.3, \( p_2 \) and \( p_5 \) are identified as malicious because they are excluded from the maximal clique of size 3.

However, malicious service providers can avoid being detected by trying to form a majority in a specific service function. After removing the assumption that the benign service providers are always the majority, the consistency graph analysis algorithm might produce false alarms by labeling benign service providers as malicious. Thus, we need to integrate the consistency graph analysis with the inconsistency graph analysis to achieve more robust integrity attestation. However, note that with the limited number of malicious service providers, the number of service functions in which they can form a majority is also limited. We will present the formal analysis on the damage degree in section 4.2.7.

4.2.5 Inconsistency Graph Analysis

We now present the inconsistency graph based pinpointing algorithm. Intuitively, given two service providers connected by an inconsistency link, we can say that at least one of them is malicious since any two benign service providers should always agree with each other. Thus, we can derive the lower bound about the number of malicious service providers by examining the minimum vertex cover of the inconsistency graph. The minimum vertex cover of a graph is a minimum set of vertices such that each edge of the graph is incident to at least one vertex of the set. For example, in Figure 4.4, \( p_2 \) and \( p_5 \) form the minimum vertex cover.

**Proposition 1**: Given an inconsistency graph \( G \), let \( C_G \) be a minimum vertex cover of \( G \).
Then the number of malicious service providers is no less than $|C_G|$. 

**Proof:** We can prove this by contradiction. Suppose the number of malicious service providers is less than $|C_G|$. Then the graph formed by malicious nodes cannot cover the entire graph, which means there exists one edge that is not incident to any of the malicious nodes. Thus, the edge must be incident to two benign nodes. Since two benign nodes always agree with each other, this contradicts with the existence of an inconsistency link between them.

Note that benign service providers that do not serve same functions with malicious ones will be isolated nodes in the inconsistency graph, since they will not be involved in any inconsistency links. For example, in Figure 4.5, nodes $p_4$, $p_5$, $p_6$ and $p_7$ are isolated nodes since they are not associated with any inconsistency links in the global inconsistency graph. Thus, we can remove these nodes from the inconsistency graph without affecting the computation of the minimum vertex cover. Since the majority of the service providers in the whole system are benign and can therefore be removed from the inconsistency graph, the size of the inconsistency graph could be greatly reduced. Although the minimum vertex cover problem is a NP-hard problem [34], the overhead of our scheme is small in practice, which will be shown in Section 4.

Given an inconsistency graph containing only the inconsistency links, there may exist different possible combinations of the benign node set and the malicious node set. However, if we assume that the total number of malicious service providers in the whole system is no more than $K$, we can pinpoint a subset of malicious service providers. Intuitively, if a node is inconsistent with more than $K$ nodes, then that node must be malicious since it must be inconsistent with at least one benign node. In fact, we can do better in pinpointing malicious service providers by examining each individual node in the inconsistency graph.

For example, in Figure 4.4, suppose we know the number of malicious service providers is no more than 2. We examine $p_2$ first. If $p_2$ is benign, then all of its neighbors should be malicious.
The overall malicious nodes can be composed of two sets. One set is the neighbor set of $p_2$. In this case, $p_2$ has 3 neighbors. The other set is the residual graph after removing $p_2$, illustrated by Figure 4.6. The minimum vertex cover of the residual graph is $C_G = \{p_5\}$. According to proposition 1, the residual graph after removing $p_2$ contains at least $|C_G|=1$ malicious service provider. Thus, if $p_2$ is benign, the total number of malicious service providers is at least $3+|C_G|=4$, which contradicts with our assumption that there are no more than 2 malicious nodes. Thus, $p_2$ must be malicious. Similarly, when we examine $p_1$, we find the total number of malicious service providers is at least 2, which is consistent with our assumption. In this case, we cannot draw any conclusion on whether $p_1$ is benign or malicious. We generalize the idea into the following proposition.

**Proposition 2**: Given an integrated inconsistency graph $G$ and the upper bound of the number of malicious service providers $K$, a node $p$ must be a malicious service provider if and only if

$$|N_p| + |C_{G_p}'| > K$$

(4.1)

where $|N_p|$ is the neighbor size of $p$, and $|C_{G_p}'|$ is the size of the minimum vertex cover of the residual graph after removing $p$ and its neighbors from $G$.

**Proof**: We can prove it by contradiction. Suppose there exists a benign service provider $p$ that satisfies $|N_p| + |C_{G_p}'| > K$. Since $p$ is inconsistent with its neighbors, the neighbors must be malicious. Then the total number of malicious service providers can be calculated by adding the number of $p$’s neighbors and the number of malicious service providers in the residual graph. According to proposition 1, we can use the size of a minimum vertex cover to serve as the lower bound number of malicious service providers in the residual graph $|C_{G_p}'|$. Since the total number of malicious service providers is no more than $K$, $|N_p| + |C_{G_p}'| \leq K$, which contradicts with the assumption $|N_p| + |C_{G_p}'| > K$. 

\[\text{Figure 4.6: Inconsistency graph } G \text{ and its residual graph.}\]
In order to escape detection, attackers need to limit the number of inconsistency links (i.e., the neighbor size) exposed in the inconsistency graph. Thus, the number of service functions that malicious service providers can attack is constrained, since the more functions they attack, the more benign service providers they may be inconsistent with. Similarly, our scheme also discourages non-colluding attacks, which may also increase the number of inconsistency links connected to the attackers.

Let \( N \) denote the total number of service providers in the system. Since we assume that the total number of malicious service providers is less than that of benign ones, the number of malicious service providers should be no more than \( \lfloor N/2 \rfloor \). According to Proposition 1, the number of malicious service providers should be no less than the size of the minimum vertex cover \( |C_G| \) of the global inconsistency graph. Thus, \( K \) is bounded by its lower bound \( |C_G| \) and upper bound \( \lfloor N/2 \rfloor \).

We use an iterative algorithm to obtain a tight estimation of \( K \). We start from the lower bound of \( K \), and compute the set of malicious nodes according to Proposition 2, denoted by \( \Omega \). Then we gradually increase \( K \) by one each time. For each specific value of \( K \), we can get a set of malicious nodes. With a larger \( K \), the number of nodes that can satisfy \( |N_s| + |C_{G_s}'| > K \) becomes less, which causes the set \( \Omega \) to be reduced. When \( \Omega = \emptyset \), we stop increasing \( K \), since any larger \( K \) cannot give more malicious nodes. Intuitively, when \( K \) is large, fewer nodes may satisfy Equation 4.1. Thus, we may only identify a small subset of malicious nodes. In contrast, when \( K \) is small, more nodes may satisfy Equation 4.1, which may mistakenly pinpoint benign nodes as malicious. To avoid false positives, we want to pick a large enough \( K \), which can pinpoint a set of true malicious service providers.

### 4.2.6 Integrated Attestation Graph Analysis

We now present our integrated attestation graph analysis algorithm. Let \( G_i \) be the consistency graph generated for service function \( f_i \), and \( G \) be the global inconsistency graph. Let \( M_i \) denote the list of suspicious nodes by analyzing per-function consistency graph \( G_i \) (i.e., nodes belonging to minority cliques), and \( \Omega \) denotes the list of suspicious nodes by analyzing the global inconsistency graph \( G \), given a particular upper bound of the number of malicious nodes \( K \). We examine per-function consistency graphs one by one. Let \( \Omega_i \) denote the subset of \( \Omega \) that serves function \( f_i \). If \( \Omega_i \cap M_i \neq \emptyset \), we add nodes in \( M_i \) to the identified malicious node set. The idea is that since the majority of nodes serving function \( f_i \) have successfully excluded malicious nodes in \( \Omega_i \), we could trust their decision on proposing \( M_i \) as malicious nodes. Figure 4.7 shows the pseudo-code of our integrated attestation graph analysis algorithm, where \( R \) is the final set of malicious service providers.

For example, Figure 4.8 shows both the per-function consistency graphs and the global
PinpointMaliciousSPs(G, G_i)
1. for every $K \in \lceil |C_G|, \lfloor N/2 \rfloor \rceil$
2. $\Omega = \emptyset$, $R = \emptyset$
3. for every node $p$ in $G$
4. compute $|N_p| + |C_{G_p}|$
5. if $(|N_p| + |C_{G_p}| > K)$
6. $\Omega = \Omega \cup \{p\}$
7. final malicious node set $R = R \cup \Omega$
8. if $R = \emptyset$
9. continue
10. else
11. for every $G_i$
12. compute $M_i$
13. set $\Omega_i$ to the subset of $\Omega$ appearing in $G_i$
14. if $(\Omega_i \cap M_i \neq \emptyset)$
15. $R = R \cup M_i$
16. return all sets of $R$

Figure 4.7: Malicious service provider pinpointing algorithm.

Inconsistency graph. If the upper bound of the malicious nodes $K$ is set to 4, the malicious node set identified through inconsistency graph analysis $\Omega = \{p_9\}$, but the actual malicious node set also includes $p_8$. The inconsistency graph analysis algorithm may not be able to capture $p_8$ because $p_8$ only participates in one function and exposes limited inconsistency links. However, by checking function $f_1$, we can find $\Omega_1 = \{p_8\}$ has overlap with the minority clique $M_1 = \{p_8, p_9\}$. We can then infer $p_8$ to be malicious too.

Note that even if we have an accurate estimation of the number of malicious nodes, the inconsistency graph analysis result may not identify all malicious nodes. However, our integrated algorithm can pinpoint more malicious nodes than the inconsistency graph based algorithm. For example, in Figure 4.9, the true malicious nodes are $\{p_7, p_8, p_9, p_{10}\}$. If we set $K = 4$, the inconsistency graph analysis returns $\Omega = \{p_7\}$. However, by checking function $f_3$, we can find $\Omega_3 = \{p_7\}$ has overlap with the minority clique $M_3 = \{p_7, p_{10}\}$. We can then infer $p_{10}$ to be malicious too.

In summary, the consistency graph based pinpointing method forces malicious attackers to form majority in every service function they attack, while the inconsistency graph based pinpointing method limits the number of functions malicious service providers can attack. By considering both per-function consistency graphs and the global inconsistency graph, we can limit the damage of malicious attacks and detect malicious service providers with a high prob-
4.2.7 Security Analysis

Our malicious service provider pinpointing algorithm has the following property.

**Proposition 3:** Given an accurate upper bound of the number of malicious service providers $K$, if malicious service providers always collude together, our algorithm does not have false positives.

*Proof:* According to Proposition 2, any node identified through inconsistency graph must be malicious. Therefore, any node in the subset of malicious nodes identified through inconsistency graph, e.g., $\Omega_i$, must be malicious. If, for all functions, $\Omega_i \cap M_i = \emptyset$, our algorithm returns set $\Omega$ as malicious set, which contains only malicious service providers. Otherwise, there exists some function $f_i$ such that $\Omega_i \cap M_i \neq \emptyset$. For any $p_b \in (\Omega_i \cap M_i)$, $p_b$ must be malicious because it belongs to $\Omega_i$. Suppose our algorithm has false positives, which means there exists a benign node $p_g$, where $p_g \in M_i$ but $p_g \notin \Omega_i$. Since $p_g \in M_i$, $p_g$ must be outside of the maximum clique. Thus, the maximum clique must be formed by malicious nodes. This indicates that malicious...
node $p_b$ must disagree with at least one of the malicious nodes in the maximum clique, which contradicts with our assumption that attackers always collude together as a single group.

Although our algorithm cannot guarantee zero false positive when there are multiple independent colluding groups, it is difficult for attackers to escape inconsistency graph based detection, since they will have inconsistency links not only with benign nodes but also with other groups of malicious nodes. Thus, our algorithm can achieve high detection rate with a small number of false positives under those attack scenarios.

We now quantify the damage degree collusive attackers can make without being detected. We assume that collusive attackers are intelligent in that they can select service functions to attack together in order to maximize the damage they can bring to the system. The damage is defined as follows.

**Definition 5**: The Damage Degree, denoted by $D$, is the number of the service functions on which malicious service providers misbehave without being detected.

Since attackers can escape detection by forming the majority in per-function consistency graphs, attackers can select service functions that have less benign service providers. Suppose there are $n$ service functions, $f_1, ..., f_n$, ranked in the descending order of the number of benign service providers participating in the function. The number of benign and malicious service providers participating in the function $f_i$ are $g_i$ and $b_i$, respectively. The total number of malicious service providers are $b$, where $b = b_1 + ... + b_n$. Thus, in order to escape detection in per-function consistency graphs, attackers need to take majority in all attacked functions, which means they can attack up to $k$ functions at different time, where $g_k \leq b \leq g_{k+1}$. That is, attackers can only attack functions $f_1, ..., f_k$. For functions $f_{k+1}, ..., f_n$, attackers cannot form a majority so that any misbehavior on these functions will be detected by our algorithm.

However, attackers cannot form majority in all the $k$ functions at the same time. The number of functions that attackers can attack simultaneously is significantly limited. If $b$ satisfies the following equation,

$$
\sum_{i=1}^{m} g_i \leq b \leq \sum_{i=1}^{m+1} g_i,
$$

then attackers cannot attack more than $m$ functions at the same time, which means the damage degree $D = m$.

Moreover, a single attacker cannot participate in unlimited number of service functions. First, the cloud infrastructure can impose certain monetary cost for provisioning a service function. Second, by providing untruthful results, the attacker increases its inconsistency links with all benign service providers provisioning $f_i$, with the number of inconsistency links equal to the number of benign service providers in that function, $g_i$. Suppose an attacker $p_b$ provides functions $f_i, f_{i+1}, ..., f_{i+k}$. In the global inconsistency graph, in order to escape detection, every
single attacker $p_b$ needs to limit the number of inconsistency links. Since malicious service providers have no idea of which functions a benign service provider serves simultaneously, a conservative strategy is to serve only on one function. If attackers are too greedy to participate in more service functions or attack functions that they cannot form a majority, they will get detected by our algorithm.

### 4.3 Implementation and Evaluation

In this section, we present the experimental evaluation of the IntTest system. We first describe our experimental setup. We then present and analyze the experimental results.

#### 4.3.1 Experiment Setup

We have implemented a prototype of the IntTest system in C++ on top of the IBM System S stream processing platform [36,49] and tested it in the NCSU virtual computing lab (VCL) [6], which is a production virtualized resource provisioning infrastructure operating in a similar way as Amazon EC2 [1]. In our experiments, we used 10 blade servers in VCL, which runs CentOS 5.2 64-bit and a set of VMs with Xen 3.0.3 for managing virtual machines.

The dataflow processing application we use in our experiments is adapted from the sample applications provided by System S, illustrated by Figure 4.10. This application takes stock information as input, performs windowed aggregation on the input stream according to the specified company name and then performs a series of calculations on the stock data. We use a trusted portal node to accept the input stream, perform comprehensive integrity attestation on the processing components and analyze the attestation results. The portal node constructs one consistency graph for each service function and one global inconsistency graph across all service providers in the system.

Figure 4.11 shows the service instances deployed in the system. The service instances are distributed on 10 different hosts. It is the System S who controls which host each service instance is assigned to. Note that the number of service instances deployed for each service function is different. The purpose is to evaluate the scenarios where attackers can take majority in some of the service functions by attacking those functions that have less number of service
instances with less efforts. In the experiments, the data rate of the input stream is 300 tuples per second.

For comparison, we have also implemented three alternative integrity attestation schemes: 1) the Full-Time Majority Voting (FTMV) scheme that employs all redundant service providers at all time for attestation and determines malicious service providers through majority voting on the processing results; 2) the Part-Time Majority Voting (PTMV) scheme that also employs all redundant service providers for attestation and determines malicious service providers using majority voting in a similar way as FTMV. However, the difference is that the part-time majority voting scheme attests all service providers probabilistically on a subset of input data; and 3) the RunTest scheme [30] that pinpoints malicious service providers by analyzing only the per-function consistency graphs, which labels those service providers that are outside of all maximal cliques of size larger than \( \lceil k/2 \rceil \) as malicious, where \( k \) is the number of service providers that take part in this service function.

Three major metrics for evaluating our scheme are detection rate, false alarm rate, and attestation overhead. We calculate the detection rate, denoted by \( A_p \), as the number of pinpointed
malicious service providers over the total number of malicious service providers that have misbehaved at least once during the experiment. During runtime, the detection rate should start from zero and increase as more malicious service providers are detected. False alarm rate $A_F$ is defined as $N_{fp}/(N_{fp} + N_{tn})$, where $N_{fp}$ denotes false alarms corresponding to the number of benign service providers that are incorrectly identified as malicious; $N_{tn}$ denotes true negatives corresponding to the number of benign service providers that are correctly identified as benign.

The attestation overhead is evaluated by both the number of duplicated data tuples that are redundantly processed for service integrity attestation and the extra dataflow processing time incurred by the integrity attestation.

We assume that the colluding attackers know our attestation scheme and take the best strategy while evaluating the IntTest system. According to the security analysis in Section 4.2.7, in order to escape detection, the best practice for attackers is to attack as a colluding group. Colluding attackers can take different strategies. They may *conservatively attack* by first attacking those service functions with less number of service providers where they can easily take majority, assuming they know the number of participating service providers for each service function. Alternatively, they may *aggressively attack* by attacking service functions randomly, assuming they do not know the number of participating service providers. We investigate the impact of these attack strategies on our scheme in terms of both detection rate and false alarm rate.
Figure 4.13: Detection rate ($A_D$) and false alarm rate ($A_F$) with 20% conservative malicious attackers.

4.3.2 Results and Analysis

We first investigate the accuracy of our scheme in pinpointing malicious service providers. Figure 4.12 compares our scheme with the other alternative schemes (i.e., FTMV, PTMV, RunTest) when malicious service providers aggressively attack different number of service functions. In this set of experiments, we have 10 service functions and 30 service providers. We set 20% of service providers as malicious. The number of service providers in each service function randomly range in [1, 8]. Each benign service provider provides two randomly selected service functions. After the portal receives the processing result of a new data tuple, it randomly decides whether to perform data attestation. Each tuple has 0.2 probability of getting attested (i.e., attestation probability $P_u = 0.2$), and two attestation data replicas are used (i.e., number of total data copies including the original data $r = 3$). Each experiment is repeated three times. We report the average detection rate and false alarm rate achieved by different schemes. Note that RunTest can achieve the same detection accuracy results as the majority voting based schemes after the randomized probabilistic attestation covers all attested service providers and discovers the majority clique [30]. In contrast, IntTest comprehensively examines both per-function consistency graphs and the global inconsistency graph to make the final pinpointing decision. We observe that IntTest can achieve much higher detection rate and lower false alarm rate than other alternatives. Moreover, IntTest can achieve better detection accuracy when malicious service providers attack more functions. We also observe that when malicious service providers attack aggressively, our scheme can detect them even though they attack a very low percentage of service functions.
Figure 4.13 shows the malicious service provider detection accuracy results under the conservative attack scenarios. All the other experiment parameters are kept the same as the previous experiments. The results show that IntTest can consistently achieve higher detection rate and lower false alarm rate than the other alternatives. In the conservative attack scenario, as shown by Figure 4.13, the false alarm rate of IntTest first increases when a small percentage of service functions are attacked and then drops to zero quickly as more service functions are attacked. This is because when attackers only attack a few service functions where they can take majority, they can hide themselves from our detection scheme while tricking our algorithm into labeling benign service providers as malicious. However, if they attack more service functions, they can be detected since they incur more inconsistency links with benign service providers in the global inconsistency graph. Note that majority voting based schemes can also detect malicious attackers if attackers fail to take majority in the attacked service function. However, majority voting based schemes have high false alarms since attacks can always trick the schemes to label benign service providers as malicious as long as attackers can take majority in each individual service function.

We then increase the percentage of malicious service providers to 40% and repeat the above two sets of experiments. Figure 4.14 shows the comparison results under the aggressive attack scenarios while Figure 4.15 shows the comparison results under the conservative attack scenarios. The results show that IntTest still achieves better detection accuracy than the other alternatives. Note that when there are a high percentage of malicious attackers, majority voting based schemes may fail to identify any attacker, while our scheme can still detect all attackers if the attackers attack many service functions.

Figure 4.16 compares IntTest with FTMV, PTMV, and RunTest in terms of detection time.
Malicious service providers has 0.2 probability to misbehave on an incoming data tuple. For probabilistic attestation schemes such as IntTest, PTMV, and RunTest, the attestation probability is 0.2. For IntTest and RunTest, two attestation data replicas are used ($r = 3$). Here, the attackers may attack different service functions with different subset of their colluders. As expected, the FTMV scheme needs the least time to detect malicious service providers because it attests all service components all the time. Although PTMV has the same attestation probability with IntTest and RunTest, it has shorter detection time since it uses all service components for each attestation data. IntTest can achieve shorter detection time than RunTest. Similar to previous experiments, IntTest achieves the highest detection rate among all algorithms. RunTest, FTMV and PTMV cannot achieve 100% detection rate since they cannot detect those attackers that only misbehave in service functions where they can take the
Figure 4.17: **Attestation overhead comparison.**

![Attestation overhead comparison graph](image)

Figure 4.18: **Detection rate of IntTest under different attestation probability.**

![Detection rate graph](image)

majority.

Figure 4.17 compares the overhead of the four schemes in terms of the attestation traffic, that is the total number of duplicated data tuples used for attestation. The data rate is 300 tuples per second. We send 20,000 data tuples for processing for each scheme. IntTest and RunTest save more than half attestation traffic than PTMV, and incur an order of magnitude less attestation overhead than FTMV.

We now evaluate the impact of various system parameters on the effectiveness of our algorithm. Figure 4.18 shows the time to detect each malicious service provider under different attestation probability $P_u$. With higher attestation probability, IntTest has more opportunities to capture the sneaky occasional misbehavior of attackers. Thus, with a higher attestation probability, we can detect malicious service providers earlier. However, the system overhead, in terms of attestation traffic, would increase accordingly since IntTest performs attestation
on more data. Figure 4.19 shows the detection rate under different misbehaving probabilities, where attestation probability is fixed at 0.2. The more frequently malicious service providers misbehave, the more opportunities are given to our scheme to capture the misbehavior. Therefore, it takes less time to detect malicious service providers with a higher misbehaving probability.

We also measure the computation overhead for the graph analysis. Table 4.1 shows the graph analysis time for both consistency graphs and inconsistency graph including both the mean and standard deviation values, where the number of service providers varies from 200 to 1000. The analysis time for consistency graphs is the sum of per-function analysis time. As the table shows, the total time for both consistency and inconsistency graph analysis is less than 140 milliseconds given 1000 service providers and 2000 service components in the system. Note that we start from a complete graph connected solely by consistency links. IntTest only triggers the graph analysis algorithm when any new inconsistency links are captured by the probabilistic attestation.

---

Table 4.1: Graph analysis time in IntTest.

<table>
<thead>
<tr>
<th># of providers</th>
<th>Consistency graph</th>
<th>Inconsistency graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>4.22 ± 0.018 ms</td>
<td>1.64 ± 0.001 ms</td>
</tr>
<tr>
<td>400</td>
<td>15.89 ± 0.013 ms</td>
<td>6.52 ± 0.004 ms</td>
</tr>
<tr>
<td>600</td>
<td>35.29 ± 0.095 ms</td>
<td>15.11 ± 0.015 ms</td>
</tr>
<tr>
<td>800</td>
<td>62.62 ± 0.021 ms</td>
<td>26.18 ± 0.350 ms</td>
</tr>
<tr>
<td>1000</td>
<td>100.00 ± 0.434 ms</td>
<td>39.98 ± 0.179 ms</td>
</tr>
</tbody>
</table>
We now evaluate the impact of our integrity attestation scheme on the data processing delay, an important performance metric for data processing systems. The data processing delay is measured as the average tuple turnaround time, which is the duration between the time when the first data tuple enters the system and the time when the last data tuple leaves the system over the total number of tuples processed. Figure 4.20 shows the average per-tuple processing delay under different data rate. The results show that IntTest imposes little overhead to the dataflow processing delay. Note that the processing delay is lower under a higher data rate. This is because the turnaround time is smaller when more data are sent into the system before the system reaches its maximum capacity. Figure 4.21 shows the average per-tuple processing delay under different numbers of service hops. The results show that our scheme only imposes about tens of microseconds extra delay. Note that our experiments are conducted in a production cluster system with high speed networks. Thus, the processing delay does not increase too much when we increase the number of service hops.

We finally compare the actual data quality with the reported data quality to evaluate the ability of our scheme in capturing data quality. Data quality is an important evaluation criteria for data processing system. We define data quality as the percentage of correctly processed data. The reported data quality is calculated as the fraction of data that the integrity attestation system believes is correctly processed. The attestation system can detect tampered data results by pinpointing malicious service components. Our system treats both non-duplicated data and duplicated data with consistent results as correctly processed data. Figure 4.22 shows the data quality comparison under different misbehaving probability. It shows that no matter how frequently malicious service provider misbehaves, our scheme can give a close estimation of the
Figure 4.21: **Average dataflow processing delay under different service hops.**

![Average tuple processing delay](image)

Figure 4.22: **Comparison of actual data quality and reported data quality.**

![Data quality comparison](image)

actual data quality. Since both IntTest and PTMV perform probabilistic attestation schemes, they may not capture all misbehavior. Thus, the reported data quality is always a bit higher than the actual data quality. However, the data quality reported by IntTest is almost the same as that of PTMV, although PTMV spends more attestation traffic than IntTest.

### 4.4 Summary

In this chapter, we have presented the design and implementation of IntTest, a novel integrated service integrity attestation scheme for processing dataflow applications in cloud systems. IntTest employs replay-based consistency check to efficiently verify the integrity of dataflow processing service components and pinpoint malicious service providers. For challenging windowed stateful services, IntTest adopts indirect state recovery to bring two services to the same state...
on demand. IntTest performs integrated analysis over both per-function consistency graphs and global inconsistency graph to effectively pinpoint colluding attackers. We have implemented IntTest on top of the IBM System S stream processing system and tested it on the NCSU virtual cloud computing lab. Our experimental results show that IntTest is effective and imposes low performance impact for dataflow processing in cloud infrastructures.
Chapter 5

AdapTest: Adaptive Data-Driven Service Integrity Attestation Scheme for Multi-Tenant Cloud Systems

In this chapter, we present the design, implementation and evaluation of AdapTest, a novel adaptive data-driven service integrity attestation scheme for multi-tenant cloud systems.

AdapTest builds on top of the RunTest [30] system that performs randomized probabilistic attestation and employs a clique-based algorithm to pinpoint malicious nodes. However, randomized attestation still imposes significant overhead for high-throughput multi-hop data processing services. In contrast, AdapTest dynamically evaluates the trustworthiness of different services based on previous attestation results and adaptively selects attested services during attestation. Thus, AdapTest can significantly reduce the attestation overhead and shorten the detection delay. In addition, AdapTest not only detects corrupted data processing results by pinpointing malicious attackers but also can automatically enhance data quality by replacing bad results produced by malicious attackers with good results produced by benign service providers.

In the following sections, we present the design details of AdapTest by first giving an overview of our approach, then we present the evaluation results.

5.1 Design and Algorithms

AdapTest is a novel adaptive multi-hop integrity attestation scheme based on a new weighted attestation graph model. In this section, we first give an overview of our approach, then we present the new weighted attestation graph model and our approach to employ trust management to achieve efficient service integrity attestation.
5.1.1 Approach Overview

AdapTest has three major design goals: 1) support runtime continuous attestation with low overhead; 2) pinpoint malicious (or compromised) service instances among a large number of interacted service instances without assuming any prior knowledge about which service instances are trusted; and 3) identify corrupted data processing results and perform auto-correction to enhance result quality. AdapTest adopts a data-driven approach to achieve the above design goals without imposing any special hardware or software requirements over remote attested services, illustrated by Figure 5.1. AdapTest leverages the portal node to perform service integrity attestation. To achieve non-repudiation, each service instance is required to produce a receipt for each data it receives and sign the data it has processed [29].

AdapTest performs attack detection using replay-based consistency check [30]. The basic idea is to duplicate some original inputs and re-send them as attestation data to different service instances for consistency check. Note that attestation data and original data are made indistinguishable to service instances. Moreover, our attestation scheme does not affect the original data processing. In other words, original data can be routed as before to different service instances for processing based on certain load balancing and quality-of-service (QoS) management objectives. The attestation data are replayed after the portal receives the original data processing results rather than being sent concurrently with the original data. Thus, we can prevent two colluding attackers from detecting attestation by comparing their received
data and thus escaping detection. Although the replay scheme may cause delay in a single data item processing, we can overlap the attestation and normal processing of consecutive data items to hide the attestation delay from the user. AdapTest leverages our previously developed clique-based algorithm [30] to pinpoint malicious nodes, and shares the same assumptions with RunTest.

AdapTest performs adaptive attestation to quickly expose malicious nodes. We make three key observations. First, we should attest suspicious nodes more often in order to capture selective cheating with minimum attestation data. Second, in order to quickly pinpoint malicious nodes, we need to expose as many inconsistency links as possible. Therefore, AdapTest dynamically derives a set of trust scores for each node based on previous attestation results and use those trust scores to guide future attestation. Specifically, AdapTest attests the nodes with lower trust scores with higher probability, and gives priority to those node pairs that have not been attested before or have been consistent before. Third, attesting multi-hop data processing services requires additional consideration since inconsistent intermediate processing results from upstream hops will invalidate attestation for all downstream hops. To address the problem, AdapTest intentionally picks good nodes based on previous attestation results for upstream hops in order to effectively attest downstream hops.

Note that AdapTest does not use the trust scores to directly pinpoint malicious nodes. Without assuming the trustworthiness of any nodes, the trust scores only represent the relative goodness of different nodes. The trust score of a specific node can dynamically change after the node is attested with different nodes. Even if the node trust scores are stabilized, it is very difficult, if not impossible, to pre-define a proper trust score threshold to separate the malicious and benign nodes. Such threshold depends on a set of unknown factors such as the percentage of malicious nodes and the misbehaving probability of those malicious nodes. Thus, AdapTest only uses trust scores to guide attestation but still uses the clique-based malicious node pinpointing algorithm to guarantee zero false positive [30].

5.1.2 Weighted Attestation Graph

AdapTest strives to pinpoint malicious service instances without making any prior assumption about the trustworthiness of any service instance. Moreover, malicious attackers can perform selective cheating during long-running data processing services, which means the trust score of a service instance must be continuously monitored and updated. Thus, AdapTest employs a weighted attestation graph to aggregate previous attestation results and dynamically derives a set of trust scores for each service instance, illustrated by Figure 5.2. We formally define the weighted attestation graph as follows.

Definition 1: A weighted attestation graph is an undirected complete graph consisting of all
functionally equivalent service instances as nodes. The weight of each edge consists of a pair of counters denoting the number of inconsistent results and the number of consistent results respectively.

For example, in Figure 5.2, $s_1$ produces three inconsistent results and two consistent results with $s_5$. Two nodes are connected by a consistent link only if they have zero inconsistent result. We can derive a node trust score and a set of pair-wise trust scores for each node from the weighted attestation graph. The node trust score denotes how trustworthy a node is and the pair-wise trust score denotes how well two nodes trust each other. We formally define both the node trust score and the pairwise trust score as follows.

**Definition 2**: The trust score of the node $s_i$, denoted by $\alpha_i$, is defined as the fraction of consistent results returned by the node $s_i$ when attested with all the other nodes. Node trust scores range within $[0,1]$, and are initialized to be 1.

**Definition 3**: The pairwise trust score between two service instances $s_i$ and $s_j$, denoted by $\beta_{i,j}$, is calculated by the fraction of consistent results when $s_i$ is attested against $s_j$. The pairwise trust score ranges within $[0, 1]$, and are initialized to be -1, which means that $s_i$ and $s_j$ have not been attested with each other yet.

The trust score of a node takes the consistency relationships between this node with all the other nodes into consideration. For example, in Figure 5.2, $s_1$ has a node trust score of 0.5 since it has total 10 consistent results and 10 inconsistent results with $\{s_2, s_3, s_4, s_5\}$. Intuitively, malicious nodes should have higher probabilities than benign ones to be inconsistent with the other nodes given that benign nodes are the majority. Thus, we assign node trust scores
according to how consistent a node is with the other nodes. Nodes that are more consistent with the others have higher trust scores and are considered to be more trustworthy. The node trust scores can be affected by two factors. The trust score of a node $s_i$ decreases if i) the node $s_i$ is inconsistent with more nodes; or ii) the node $s_i$ is inconsistent with other nodes more frequently.

The pairwise trust scores reflect how consistent two nodes are and therefore how trustworthy they think each other. The more frequently two nodes give inconsistent results, the less pairwise trust score between them. Note that we initialize pairwise trust scores with -1 to indicate that the two nodes have not been attested together before. For example, in Figure 5.2, if the pairwise trust score between two nodes equals to 1, we draw a solid line between them. Otherwise, if two nodes do not always agree with each other, we use a dashed line to represent the inconsistency relationship. The pairwise trust score between $s_1$ and $s_2$ is $2/(4 + 2) = 0.33$ since they produce 4 inconsistent results and 2 consistent results.

### 5.1.3 Per-Hop Adaptive Attestation

AdapTest leverages dynamically derived trust scores to intelligently guide probabilistic service attestation. The goal of our adaptive attestation scheme is to expose malicious nodes faster. We achieve the goal by capturing more inconsistency relationships for malicious nodes so that they can be pushed out from the maximum consistency clique.

AdapTest expedites the exposure of inconsistency relationships and therefore shorten detection time using two adaptive node selection schemes. First, AdapTest selects suspicious nodes that have low trust scores and attests those suspicious nodes more frequently. The rationale is that the nodes that have already delivered more inconsistent results have the potential to deliver even more inconsistent results in the future attestation. By intensively attesting suspicious service nodes, we may have higher probabilities to find inconsistency results. Second, AdapTest strives to attest suspicious nodes together with benign nodes since two colluding malicious nodes will try to avoid producing inconsistent results with each other. Attesting a suspicious node together with a benign one is more effective in producing inconsistent results.

For scalability, AdapTest performs probabilistic attestation by randomly selecting a subset of input data for consistency check. When an input data item is selected for attestation by the portal, AdapTest first identifies a pool of suspicious nodes based on node trust scores and randomly selects a suspicious node from this pool to attest. Given the assumption that malicious nodes are no more than half of total nodes, we rank all nodes in an increasing order of trust scores, and mark the first half nodes as the suspicious node pool $B$ and the rest of nodes as the benign node pool $G$. We then randomly pick one node from the suspicious node pool, excluding the node processing the original data, for attestation by comparing the result of the
attestation data with that of the original data. Note that we do not want to always attest the node with the lowest trust score for maintaining attestation coverage and tolerating imprecise trust scores. Moreover, we want to avoid alerting the malicious node by continuously attesting it. At the beginning, since all nodes have the same initial trust scores, AdapTest will randomly pick nodes from the whole node pool to attest.

AdapTest may send multiple attestation data to attest different nodes concurrently. To maximize the chance of capturing inconsistent results, we want to attest a suspicious node with a benign node together. Thus, after picking a suspicious node \( s_i \) from \( B \), AdapTest picks the other attested node from the benign node pool \( G \) using the following rules. First, if there are benign nodes that have not been attested with \( s_i \) before (i.e., pairwise trust score equals to -1 in the weighted attestation graph), we randomly pick one from them. Second, if all nodes in the benign set have been attested with \( s_i \), we randomly pick one from \( G \) that have always been consistent with \( s_i \). If all nodes in \( G \) are inconsistent with \( s_i \), we pick the one that has the highest pair-wise trust score with \( s_i \). Our scheme can achieve both good coverage and avoid wasting attestation traffic on those node pairs that have already presented inconsistency relationships.

Figure 5.3 shows an example of adaptive per-hop attestation. The number associated with each protocol step indicates the execution order. If two steps have the same number, it means that the two steps are executed concurrently. The portal first sends the original data \( d_1 \) to \( s_1 \) for processing. After the portal receives the result from \( s_1 \), it decides to perform attestation by replaying \( d_1 \) on two service instances. The portal first randomly picks one node from the suspicious node set \( \{s_3, s_4\} \) to attest, say, \( s_3 \). Then the portal picks the node with the highest
pairwise trust score with $s_3$ from the benign set $\{s_1, s_2, s_3\}$, say $s_5$.

### 5.1.4 Multi-Hop Adaptive Attestation

Complicated data processing services often comprise multiple data processing functions called service hops. Malicious attackers can attack any of the service hops to compromise the final data processing results. Suppose a data processing service consists of total $n$ hops and an input data item is selected for attestation through two service paths, $s_1 \rightarrow ..., s_i \rightarrow ... \rightarrow s_n$ and $s'_1 \rightarrow ..., s'_i \rightarrow ... \rightarrow s'_n$, respectively. If the intermediate processing results begin to become inconsistent at the hop $s_i$, then the attestation for all service hops after $s_i$ becomes invalid since all downstream node pairs, such as $s_{i+1}$ and $s'_{i+1}$, would receive different input data. In this case, attestation data cannot be efficiently utilized. More attestation data are required to attest downstream service hops, which will result in extended detection time in multi-hop attestation.

AdapTest provides adaptive multi-hop attestation by intentionally picking benign upstream nodes based on the node trust scores in order to efficiently attest a downstream node, illustrated by Figure 5.4. Specifically, during a $n$-hop service attestation, we first randomly select a service hop, say the $i$th hop, as the target attestation hop. For each service hop before the $i$th hop, we intentionally select nodes that have high trust scores. For the $i$th to the $n$th service hops, we follow the same per-hop attestation node selection scheme described in the previous subsection. Thus, AdapTest maintains high probabilities for the nodes at the $i$th hop to receive consistent
input to perform valid attestation.

Figure 5.4 shows an example of multi-hop adaptive attestation. We target to attest the second service hop. Thus, we select only benign nodes \{s_1, s_3, s_5\} for the first service hop. For the second service hop, AdapTest randomly selects \(s_8\) from the suspicious set, and selects \(s_{10}\) from the benign set for concurrent attestation. Similarly, for the third service hop, a suspicious node \(s_{12}\) and a benign node \(s_{13}\) are selected for attestation.

5.1.5 Result Auto-Correction

AdapTest can automatically correct corrupted data processing results to improve the result quality of the cloud data processing service, illustrated by Figure 5.5. Without our attestation scheme, once an original data item is manipulated by any malicious node, the processing result of this data item can be corrupted, which will result in degraded result quality. AdapTest leverages the attestation data and the malicious node pinpointing results to detect and correct compromised data processing results.

Specifically, after the portal node receives the result \(f(d)\) of the original data \(d\), the portal node checks whether the data \(d\) has been processed by any malicious node that has been pinpointed by our algorithm. We label the result \(f(d)\) as a “suspicious result” if \(d\) has been processed by any pinpointed malicious node. Next, the portal node checks whether \(d\) has been chosen for attestation. If \(d\) is selected for attestation, we check whether the attestation copy
of $d$ only traverses good nodes. If it is true, we will use the result of the attestation data to replace $f(d)$. For example, in Figure 5.5, the original data $d$ is processed by the pinpointed malicious node $s_6$ while one of its attestation data $d''$ is only processed by benign nodes. The portal node will use the attestation data result $f(d'')$ to replace the original result that can be corrupted if $s_6$ cheated on $d$.

5.2 Experimental Evaluation

In this section, we first describe our experiment setup, the schemes used for comparison as well as the evaluation metrics. We then present the experimental results in detail.

5.2.1 Experiment Setup

We have implemented AdapTest in c++ on top of the IBM System S stream processing system [36]. We have deployed and tested the AdapTest prototype on a subset nodes of the NCSU virtual computing lab (VCL) [6], which is a virtualized computing cluster similar to Amazon EC2. We used 10 blade servers in VCL, each of which runs CentOS 5.2 64-bit and a set of VMs with Xen 3.0.3 hypervisor.

The data processing application we use is extracted from the sample application provided by IBM System S stream processing system, illustrated by Figure 5.6. The application takes real weather data from weather stations as input, performs conversions and calculations, and generates the most recent weather information for different locations where the fruit suppliers are located. The results help in making decisions on whether to purchase fruit from a supplier. We perform attestation on three service hops, with five service instances at each hop. The input data rate is 300 tuples per second. We have one trusted portal node that accepts input data streams and constructs the weighted attestation graphs for each service function based on adaptive attestation results. The service instances are deployed in a similar way as in RunTest, illustrated by Figure 5.7.

We first compare AdapTest with RunTest, both of which perform attestation probabilistically and employ only a subset of nodes at a time for attestation. Note that RunTest randomly selects service instances at every hop independently for attestation. For both schemes, when the
Byzantine Fault Tolerance techniques [55]. The full time majority voting scheme performs in-
puting the data to \( r \) copies to launch service attestation, where \( p_u \) is called attestation probability
and \( r \) is called redundancy degree. In this experiment study, both schemes leverage the portal
node to select service paths for attestation, which strives to avoid selecting the same node on
different attestation paths for good attestation coverage\(^1\).

We also compare AdapTest with the full time majority voting scheme used by traditional
Byzantine Fault Tolerance techniques [55]. The full time majority voting scheme performs in-
tegrity attestation for all input data using all service instances all the time. When inconsistency
happens, the scheme relies on majority voting to detect which instances are faulty.

We evaluate AdapTest using three major metrics: detection time, attestation overhead, and

\(^1\)Our original RunTest system implementation makes the portal select attestation nodes for different at-
testation paths independently, which often involves the same node on different attestation paths. Thus, the
performance of RunTest reported in this section is already much improved compared to [30].
result quality. The detection time is the time duration that is needed to detect all malicious nodes. Early detection is desired so that the system can make proper actions to prevent malicious nodes from compromising more data processing results. The attestation overhead is calculated as the number of attestation data that are needed to detect all malicious nodes. The result quality is the percentage of data processing results that are truthful.

5.2.2 Results and Analysis

We first evaluate the detection time and start with the most challenging case. Figure 5.8 and Figure 5.9 compare the detection time in collusion scenarios, with 40% malicious nodes and node misbehaving probability of 0.2 and 0.4 respectively. Here, the detection rate is calculated as the number of pinpointed malicious nodes over the total number of malicious nodes that have misbehaved at least once during the experiment. The detection rate starts at zero and
Figure 5.10: **Attestation overhead comparison under colluding attacks. (40% malicious nodes)**

![Graph showing attestation overhead comparison.]

Figure 5.11: **Detection time comparison under non-colluding attacks with node misbehaving probability = 0.2. (40% malicious nodes)**

![Graph showing detection time comparison.]

keeps increasing as more malicious nodes are pinpointed. For both algorithms, the attestation probability $p_u$ is 0.2. That is, the portal node randomly selects 20% of original input data for attestation. Each time, the portal node uses two duplicated attestation data items to perform two-way concurrent attestation. Our results show that our adaptive attestation scheme achieves much shorter detection time than the random attestation scheme.

We also evaluate the attestation overhead. Figure 5.10 compares AdapTest and RunTest under collusion scenarios, with 40% malicious nodes. The misbehaving probability of all malicious nodes varies from 0.2 to 1. When they misbehave, all malicious nodes give the same incorrect processing results. Our results show AdapTest can consistently achieve lower attestation overhead than RunTest.

We now evaluate our algorithms under non-colluding attack cases. Figure 5.11 and Figure 5.12 show the time to detect each of the malicious nodes with 40% malicious nodes under node
Figure 5.12: Detection time comparison under non-colluding attacks with node misbehaving probability = 0.4. (40% malicious nodes)

Figure 5.13: Attestation overhead comparison under non-colluding attacks. (40% malicious nodes)

misbehaving probability of 0.2 and 0.4, respectively. We again observe that in both scenarios AdapTest achieves 100% detection rate much earlier than RunTest. Figure 5.13 shows the attestation overhead comparison under non-colluding attack scenarios, with 40% malicious nodes. Each malicious node misbehaves independently with the misbehaving probability varying from 0.2 to 0.8. The results show that AdapTest consistently incurs less attestation overhead to achieve 100% detection rate, with up to 60% less attestation overhead than RunTest. Note that both schemes need less attestation traffic to detect all malicious nodes when malicious nodes misbehave more frequently. This is because the schemes have more opportunities to catch inconsistency results and derive inconsistency relationships between the nodes. We observe that AdapTest can achieve significant cost reduction under different node misbehaving probabilities.

Comparing Figure 5.10 and Figure 5.13, we can see that the attestation overhead for non-
collusion scenarios is generally lower than that in collusion scenarios. This is because the inconsistency links of malicious nodes increase much faster in non-collusion scenarios, since malicious nodes produce inconsistent results with both benign nodes and other malicious nodes.

For sensitivity study, we also evaluate the attestation overhead for non-collusion scenarios with 20% malicious nodes, as shown in Figure 5.14. Again, AdapTest also incurs the lower overhead among all the probabilistic attestation schemes. We did not evaluate collusion scenarios with 20% malicious nodes because we would have only one malicious node per hop, which cannot form collusion. Comparing Figure 5.14 and Figure 5.13, we can see that the attestation overhead that is needed to detect all malicious nodes is lower under 20% malicious nodes than that under 40% malicious nodes. This is because we have less malicious nodes to detect.

We also compare the probabilistic attestation schemes with the full time majority voting scheme. Figure 5.15 compares the detection time with 95% confidence for RunTest, AdapTest and the full time majority voting scheme under challenging collusion scenarios with 40% malicious nodes, where attestation probability is 0.2. Since the full time majority voting scheme employs all nodes all the time, it can detect malicious nodes in the shortest time. However, as Figure 5.16 shows, it has much higher overhead than the probabilistic attestation schemes. In contrast, RunTest and AdapTest tradeoff a short detection delay for a much lower attestation overhead. We observe that our adaptive attestation scheme can significantly reduce the detection delay by up to 40% compared to the random attestation scheme. Note that AdapTest and RunTest have the same continuous attestation overhead because they generate the same amount of attestation data given the same attestation probability and the same number of duplicates per attestation.

Figure 5.14: Attestation overhead comparison under non-colluding attacks. (20% malicious nodes)
We evaluate AdapTest in terms of the result quality improvement. We compare the result quality without auto-correction and with auto-correction, and also investigate the impact of the attestation probability. Figure 5.17 and Figure 5.18 show the result quality under non-colluding attacks with 20% malicious nodes and colluding attacks with 40% malicious nodes respectively. We vary the attestation probability from 0.2 to 0.4. In both scenarios, AdapTest can achieve significant result quality improvement without incurring any extra overhead other than the attestation overhead. AdapTest can achieve higher result quality improvement under higher node misbehaving probability. This is because AdapTest can detect the malicious nodes earlier so that it can correct more compromised data using the attestation data.

We evaluate the overhead imposed on the data processing delay for AdapTest. We compare the average data processing delay with and without attestation under different data rates.
Figure 5.17: Result quality detection and auto-correction performance under non-colluding attacks. (20% malicious nodes)

Figure 5.18: Result quality detection and auto-correction performance under colluding attacks. (40% malicious nodes)

Figure 5.19 shows AdapTest imposes little delay overhead on the stream processing.

5.3 Summary

In this chapter, we have presented AdapTest, a novel adaptive runtime service integrity attestation scheme for large-scale multi-tenant cloud systems. AdapTest employs a data-driven intelligent integrity attestation approach, which employs historical attestation results to guide future attestation. Particularly, AdapTest dynamically derives both node trust scores and pairwise trust scores, and provides differentiated probabilistic attestation. We have implemented AdapTest on top of the IBM System S stream processing system and tested it on the NCSU virtual computing lab. Our prototype implementation indicates that AdapTest is feasible and
Data processing delay comparison.

Figure 5.19: Data processing delay comparison.

efficient for real cloud systems. The experimental results show that AdapTest can reduce attestation overhead by up to 60% and shorten detection delays by up to 40% compared to previous approaches.
Chapter 6

Other Security Vulnerabilities and Countermeasures

Besides service integrity attacks, where a malicious service provider may perform an arbitrary data processing function instead of its advertised one, we have also identified two other major security attacks that can compromise the integrity of dataflow processing in open cloud systems.

By examining the systems in different aspects including protocol layer, communication layer and application layer, we consider the following security threats, which, to the best of our knowledge, have not been addressed by existing security management schemes: 1) Data attacks where a malicious service provider may alter input or output data tuples. For example, in Figure 2.1, a malicious service provider may alter $d_1$ or drop $d_1$. Other data handling attacks include dropping output data tuples, substituting correct data tuples with fake ones, injecting bogus data tuples, and replaying old data tuples; and 2) Dataflow topology attacks where a malicious service provider may change the topology of a dataflow processing application. For example, in Figure 2.1, service component $s_4$ may insert an additional hop to the topology by forwarding its output data tuples to its colluder $s_3$.

As countermeasures, we have developed a provenance-based data protection scheme that enforces processing service components to provide “receipts” for each input data tuple they receive and keep “evidence” for each data tuple they produce. This protocol can effectively counter data attacks in distributed dataflow processing. We have also designed a cascading dataflow topology encryption scheme to protect both confidentiality and integrity of dataflow topologies. Our topology encryption scheme assures that each processing service component knows nothing about the whole dataflow topology except its upstream and downstream components and no one can change the topology without being detected.

The organization of this chapter is as follows. First, we describe our provenance-based data protection scheme to counter data attacks such as data dropping, insertion, substitution, and
data replay. Then, we present the cascading topology protection scheme.

6.1 Provenance-based Data Protection

In this section, we describe a provenance-based data protection scheme to protect data tuples. We assume that both the portal and service components have public/private key pairs bound to themselves, with which they can encrypt, decrypt, and sign data. A party cannot forge others’ signatures or decrypt data encrypted using others’ public keys.

A malicious service component may drop an input data tuple and claim that the upstream component fails in sending the data tuple. Similarly it may also claim that it receives an intermediate data tuple $d'$ from an upstream component even though the upstream component actually sends $d$. To counter such attacks, we require each service component send a “receipt” for each data tuple it receives to its upstream component. The receipt is used to resolve the dispute between two interacting service components. The receipt includes the sequence number and session ID of the data tuple for which the receipt acknowledges, and the hash value of the received data tuple. In order to achieve integrity and non-repudiation, the receipt message is signed with the downstream service component’s private key. When an upstream service component receives a receipt, it can verify the receipt and make sure that its downstream did receive what it sent before.

When an upstream service component does not receive any receipt from its downstream component, it will ask the portal to forward the data tuple to its downstream component. If the portal does not receive a receipt from the downstream component, we can conclude that the downstream component is either unreliable or untrustworthy. Otherwise, after the portal receives the receipt, the portal forwards it to the upstream component as evidence that the downstream component receives its output data tuple. However, if the portal keeps receiving such requests from the same component, it is reasonable for the portal to suspect that either the upstream component or the downstream component is malicious, and either one or both of them try to launch a denial-of-service attack and keep the portal busy. In this case, the portal will mark those two service components suspicious and employ other service components. If the underlying communication channel guarantees in-order message delivery, each service component only needs to store the last receipt for each session. Otherwise, the service component needs to employ a sliding-window mechanism to record possibly out-of-order receipts.

A malicious service component may also drop output data tuples, substitute correct data tuples, or inject bogus data tuples. To counter those attacks, we require each service component to create the hash values for both the input data tuple it receives and the output data tuple it generates to form a data provenance evidence. The evidence can be either stored on
different distributed components as cached evidence or carried with the result data tuple as carry-on evidence. Compared to carry-on evidence, cached evidence induces smaller overhead to the dataflow processing system. However, the portal needs to dynamically request evidence from distributed service components to perform hop-by-hop service attestation and pinpoint malicious service components when the portal detects inconsistency among final data processing results. In contrast, the carry-on evidence allows the portal to perform immediate security diagnosis.

When the portal receives the final result data tuple, it can verify the consistency between different service components based on either cached data provenance evidence or carry-on data provenance evidence provided by different service components. For example, let us consider a dataflow portal → s₁ → s₂ → portal, provisioning functions f₁ → f₂. Let d denote the source data tuple received by the composite dataflow application. Figure 6.1 shows the dataflow processing with carry-on provenance evidence. The input and output of the first hop service component s₁ is (d, f₁(d)). The component s₁ signs this information and then encrypts it with the portal’s public key key_c to get [[d, f₁(d)]sigₛ₁]key_c. If every component is honest, the input of one component should be equal to the output of its preceding component. In addition, since each data tuple has a sequence number and session ID, it can be uniquely identified by each component and the portal. Thus, the malicious service component cannot launch a successful replay attack by sending old data tuples to legitimate service components.

Figure 6.1: Dataflow processing with carry-on data provenance evidence.
6.2 Dataflow Topology Protection

Our dataflow topology protection scheme has two objectives: i) any service component cannot change the dataflow topology; and ii) each participating service component only knows a minimum part of the dataflow topology (e.g., its previous-hop and next-hop components). We provide a cascading topology encryption (CTE) scheme to achieve the topology protection objectives. We would like to explain the CTE scheme using an example. Let us consider a dataflow topology $\Omega: C \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow C$, where $C$ denotes the portal and $s_i$ denotes the $i$th data processing component. After applying the CTE scheme, the encrypted dataflow topology $\{\Omega\}_E$ becomes $C \rightarrow [s_1]_{\text{sig}_C} \rightarrow [[s_2]_{\text{sig}_C} \rightarrow [[C]_{\text{sig}_C} \rightarrow [s_3]_{\text{sig}_C} \rightarrow [s_4]_{\text{sig}_C} \rightarrow \text{key}_{s_1}]_{\text{key}_{s_3}}$, where $\text{sig}_C$ denotes the signature of the portal and $\text{key}_{s_i}$ denotes the public key of the service component $s_i$. Figure 6.2 shows how CTE scheme encrypts a two-hop dataflow topology. Our cascading technology is similar to onion routing [37]. The difference is we also authenticate the topology to ensure that no one can change it.

To prevent malicious service components from tampering the dataflow topology, the portal signs each-hop in the dataflow topology using its private key. Thus, it is impossible for malicious service components to forge any data processing hop. It is easy to see that our CTE scheme only allows each service component to know its previous-hop and next-hop in the dataflow. For example, when the first-hop service component $s_1$ receives the topology information $\Omega$: $[[s_2]_{\text{sig}_C} \rightarrow [[s_3]_{\text{sig}_C} \rightarrow [[C]_{\text{sig}_C} \rightarrow [s_4]_{\text{sig}_C} \rightarrow \text{key}_{s_3}]_{\text{key}_{s_2}}]_{\text{key}_{s_1}}$, it can only obtain the next-hop component information $s_2$ using its own private key. Moreover, $s_1$ knows its previous hop in order to implement the receipt-based communication protocol described in section 6.1. However, $s_1$ cannot acquire the third hop $s_3$ since the information is encrypted using $s_2$’s public key that can only be decrypted by $s_2$. 

Figure 6.2: Cascading dataflow topology encryption.
However, if multiple malicious service components collude, our topology protection scheme needs to integrate with the data protection scheme to detect the malicious behavior. Let us consider two collusive malicious components $s_1$ and $s'_1$. The malicious component $s_1$ can forward its output data tuple to its colluder $s'_1$ who is not part of the dataflow. After $s'_1$ processes the data tuple, $s'_1$ sends it back to $s_1$, and then $s_1$ forwards it to $s_2$. In this case, $s_2$ cannot detect that the dataflow has been changed and may think the data tuple is routed from $s_1$ to itself directly. However, the data tuple received by $s_2$ may be tampered by either $s_1$ or $s'_1$. This case becomes a service integrity attack, which has been addressed by this dissertation. On the other hand, if $s'_1$ does not send the data tuple back to $s_1$ but to $s_2$, that means the dataflow topology is changed to $s_1 \rightarrow s'_1 \rightarrow s_2$. It can be detected when the portal receives the final result with carry-on evidence, because the portal can check if the result data tuple is processed by the right set of service components. However, if the final result data tuple does not contain carry-on evidence, the portal cannot verify the composition integrity immediately. Although the portal can acquire the cached evidence from all service components on the dataflow path to verify the dataflow topology later, it is hard for the portal to decide when to do so in order to achieve efficiency. Moreover, it is time-consuming to receive all the cached evidence from each service component. In order to handle this situation, we require that each data tuple carry a trace that records all service components where the data tuple is processed. Considering the original example, the output data tuple from $s_2$ will carry a trace: $[[s_1]_{sig_{s_1}}]_{key_c} \rightarrow [s_2]_{sig_{s_2}}$. And when $s_3$ receives the data tuple from $s_2$, $s_3$ will verify $[s_2]_{sig_{s_2}}$ using $key_{s_2}$ to make sure that the previous hop is the same as what it claims. Thus, the final result data tuple will include a trace: $[[[s_1]_{sig_{s_1}}]_{key_c} \rightarrow [s_2]_{sig_{s_2}}]_{key_c} \rightarrow [s_3]_{sig_{s_3}}$, and the portal can use the trace to verify if the data tuple has been processed as expected.

Although a malicious service component cannot forge a dataflow processing hop without the portal’s signatures, it can launch replay attacks by replacing the current encrypted dataflow topology with an old one that was signed by the portal. To remedy this problem, a unique session ID is attached to each hop and both of them are signed by the portal. For simplicity, session IDs are not shown in the encrypted dataflow topology.

The cascading topology encryption scheme can be extended to support graph-based service composition encryption by transforming the graph-based service composition into a tree-based service composition. Thus, the linear encryption scheme can be applied to each branch in the tree.
6.3 Experimental Evaluation

6.3.1 System Implementation

We have implemented a prototype of secure dataflow processing system, including the provenance-based data protection scheme, the cascading topology encryption scheme, and our replay-based attestation scheme [26,28,30]. Our prototype is built on top of SpiderNet [41], which is an integrated P2P service composition framework that provides scalable quality-aware service composition and proactive failure recovery in a fully distributed fashion. We have tested the system on the wide-area network testbed Planetlab [5]. Our experiments use more than 200 Planetlab hosts that are distributed all over the world. Each PlanetLab node represents one service provider that offers one or more data processing components. The portal is deployed at a predefined PlanetLab host, which is responsible for composing dataflow applications and verifying the integrity of the distributed dataflow processing. For simplicity, we only deploy one portal in our experiment. However, our design is readily applicable to large-scale open distributed systems that may require multiple portals. Our dataflow processing system closely follows the design of the IBM System S stream processing system [49].

For security protection, our dataflow processing system runs on top of a public key infrastructure that is deployed in advance. The portal and service components know each other’s public keys and use public/private key pairs to perform encryption, decryption, signature, and verification. Same as many implementation in practice, when encrypting a message, the upstream component generates a temporary secret key to encrypt the message and then uses the public key of a downstream service provider to encrypt the secret key. After the receiver gets the message with the encrypted key, it can first obtain the secret key using its private key and then decrypt the message.

For comparison, we have implemented three alternative dataflow processing schemes: i) insecure dataflow, ii) secure dataflow with cached evidence, and iii) secure dataflow with carry-on evidence. The insecure dataflow scheme does not provide any security protection and only considers the QoS and resource requirements of dataflow users. In contrast, the two secure dataflow schemes include the provenance-based data protection scheme, the cascading topology encryption scheme and the replay-based attestation scheme. The difference is that in the cached evidence scheme, the data provenance evidence is stored locally on different distributed service components. However, in the carry-on evidence scheme, each service component inserts the data provenance evidence into its output data tuples. When the portal receives the final result data tuple, which contains all the data provenance evidence, the portal can then perform an immediate security check.

In the following, we present our evaluation results on the performance impact of our schemes. The metric we use is dataflow processing delay, which is measured by the average per-tuple
turnaround time (i.e., the duration between the time when the first data tuple enters the system and the time when the last data tuple leaves the system over the total number of data tuples processed).

6.3.2 Results and Analysis

Figure 6.3 compares the insecure dataflow scheme, the secure dataflow with cached evidence scheme, and the secure dataflow with carry-on evidence scheme, in terms of the dataflow processing delay under different number of service hops in service composition. The data sending rate is 15 tuples per second and data tuple size is 100 bytes. We observe that the overhead of our security protection mechanism is about 20ms for both secure schemes, with the carry-on
scheme being a little more expensive, which, to a great extent, depends on the computation ability of a specific PlanetLab host. Note that the overhead showed here is less than the sum of the actual encryption/decryption processing delay at each service component. This is because the data processing time of consecutive data tuples has overlaps due to the continuous processing nature of streams. We measure the average per-hop processing time and the average per-hop encryption/decryption time to be 100ms and 15ms respectively, which means the per-hop overhead of security mechanism is about 15%.

Figure 6.4 compares the three schemes in terms of dataflow processing delay as a function of data size, where the number of service hops is 5 and data sending rate is 15 tuples/sec. The results show that the dataflow processing delay dramatically increases with the increase of data size, no matter which scheme is used. However, increasing data size shows negligible impact on the overhead of enforcing security mechanisms. Moreover, the fraction of the security overhead out of the total dataflow processing delay decreases as the size of data tuple increases. In addition, the dataflow processing delay of the carry-on evidence scheme is very close to that in the cached evidence scheme due to the small size of the hash values.

Figure 6.5 shows the impact of the provenance-based data protection scheme on the dataflow processing delay under dropping attacks. In this experiment, the number of service hops is 10, and data sending rate is 15 tuples per second. The data size is 100 bytes. The dataflow processing delay increases as either of the two parameters (i.e., the percentage of malicious parties, drop probability) goes up. This is because when either of the two parameters increases, the probability of getting the portal involved to forward data tuples to downstream components becomes higher. Figure 6.6 shows a sensitivity study on the performance impact under different number of service hops, different percentages of malicious nodes, and different dropping
probability, where the data sending rate is 15 tuples per second and data size is 100 bytes. It is clear that the average dataflow processing delay increases as the increase of the faction of malicious nodes and the dropping probability.

6.4 Summary

In this chapter, we have identified another two major security threats that can compromise the integrity of dataflow processing in open cloud systems, namely, data attacks and dataflow topology attacks. We then present scalable countermeasures for dataflow topology and inter-component communication to protect the integrity of composed dataflow processing applications and achieve trustworthy distributed dataflow processing. To the best of our knowledge, our work makes the first attempt to address the integrity of composed dataflow application delivery in open distributed systems. We have implemented a prototype of the secure distributed dataflow processing system and tested it on the wide-area network testbed PlanetLab. Our initial experimental results show that the proposed schemes are effective and impose low overhead to the distributed dataflow processing system.
Chapter 7

Related Work

In this chapter, we discuss previous work that is related to service integrity assurance. We classify the related work into the following categories.

7.1 Trust Management

Trust management has been exploited in multi-party systems to protect the interests of honest parties and expose malicious or dishonest parties. It has been studied in different application contexts [16,52,59,70,72]. Generally, users or components of a distributed system, are evaluated according to some trust metrics. A higher trust score is assigned to users or components who follow the rules honestly.

Trust management schemes usually rely on passively monitoring node interactions and obtaining node feedbacks to establish node trust scores. For example, the EigenTrust [52] algorithm aims to reduce the number of fake files in peer-to-peer (P2P) networks. It assigns each peer a unique trust score based on the peer’s history of uploading authentic files. It then identifies and isolates malicious peers by requiring peers to interact with each other based on the trust score. NetProbe [59] detects networks of fraudsters in online auction sites by analyzing user transactions and infers fraudsters by detecting suspicious patterns. Feedback-based trust management systems may be highly unreliable since malicious parties can misuse the reputation systems by submitting dishonest feedbacks. TrustGuard [72] targets at the vulnerabilities in a reputation system itself. It utilizes a feedback admission control mechanism and feedback credibility based algorithms to prevent malicious nodes from submitting dishonest feedbacks or flooding feedbacks with fake transactions.

In contrast, our service integrity attestation schemes evaluate different service instances by actively attesting them rather than merely performing passive monitoring. Our scheme relies on a trusted portal node to strategically decide when and whom to attest. Note that in our
AdapTest scheme, trust scores are employed for intelligently guiding the active attestation rather than pinpointing malicious nodes, which can be highly inaccurate, especially under colluding attacks. We still rely on the clique-based algorithms and the trusted portal node to achieve zero false positive.

7.2 Remote Attestation

Remote attestation techniques use a challenge-response paradigm for detecting malicious behavior. They are generally used to ensure that a remote software platform is running code that is not compromised or altered by attackers. The techniques can be classified into system-level and application-level attestation.

System-level attestation schemes are mostly based on a challenge-response paradigm performed on system resources, such as memory, to provide integrity assurance. A remote attester sends a challenge to the untrusted party, who converts the content of its system resources as the response. The remote attester can locally compute the answer to its challenge, and can thus verify the answer returned by the untrusted party. A variety of techniques [16,33,35,51,65,67–69] have been proposed in the literature. Some of the schemes require a trusted entity (e.g., trusted hardware or trusted secure kernel) to coexist with the software platform and rely on the trusted entity to provide integrity evidence by converting memory contents through some cryptographic means. Other schemes require the verification procedure to be pre-programmed into the system memory of the untrusted party or downloaded from the attester prior to verification.

For example, Sailer et al. designed an integrity measurement system for Linux, which relies on a piece of trusted hardware, the Trusted Platform Module (TPM), to verify the integrity of both system software and application software. SWATT [68] performs software-based attestation for embedded devices, which need to compute a checksum of the memory at random locations whenever receiving a challenge. Observing the great variability in software versions and configurations as well as the time-of-use and time-of-attestation discrepancy, Shi, et al. provided a fine-grained code attestation scheme [69] for distributed systems, attesting a piece of code immediately before it is executed. Alam et al. proposed a set of specification, hardware-based trust measurement, and verification schemes for attesting the behavior of business processes [9].

In multi-tenant cloud systems, it is impractical to assume the existence of a trusted entity at the third-party service provider site. It may also be impractical to require modifications to the third-party service providers to download verification procedure from the remote attester. In contrast, our approach relies on application-level function redundancy and data attestation, instead of low-level system attestation.

Application-level auditing protocols are mostly based on a challenge-response paradigm performed on application data to provide service integrity in distributed systems. They have been
proposed in different application contexts, such as peer-to-peer systems [45], publish-subscribe systems [71], cloud storage systems [17], distributed web applications [76], and database systems [80]. Such systems either construct integrity evidence through cryptographical transformation of application data [17, 45] or rely on emulation [76] to detect deviation from expected execution results. For example, HAIL [17] provides a distributed cryptographic system that allows a set of servers to prove to a client via a challenge-response protocol that a stored file is intact and retrievable. The response is computed using certain hash function families over the stored file. RIPLEY [76] can preserve the integrity of distributed computing by replicating client-side computation on a trusted server and verifying client execution through emulation. Generally, an auditor needs to challenge the untrusted party periodically. Auditors could be trusted parties [76], or a group of untrusted auditors [45, 57].

In contrast to other application-level attestation schemes, our approach supports black-box service integrity attestation, which is general to different application service functions. Note that Qing et al. proposed a similar compromised core algorithm with our inconsistency graph method in the IntTest system to identify malicious nodes in sensor networks [83]. However, the difference is that we employ both consistency and inconsistency relationships in the IntTest system to pinpoint malicious service providers. Moreover, unlike in sensor networks, different service providers provide non-uniform functions in cloud systems, which presents much complicated application scenarios.

7.3 Byzantine Fault Tolerance Techniques

Our work is also closely related to Byzantine Fault Tolerance (BFT) techniques [8, 20, 23, 24, 46, 54, 55, 57, 75], which can detect arbitrary faults in replicated systems. Given no more than one third out of a total of $n$ replicas being faulty, BFT can guarantee correct execution of requests and exposure of Byzantine faulty replicas through full time majority voting.

The goal of BFT is to achieve consensus on all good nodes by preserving the correct total ordering on the node replicas. BFT deals with stateful systems in that client requests may change the state of servers through write operations. BFT provides powerful fault detection capabilities by guaranteeing every client request get correct response. However, without a central trusted entity, it often requires all nodes communicate with each other and enforces a certain agreement-based protocol in replicated systems in order to achieve consensus, which incurs prohibitive overhead for large-scale systems. Although recent techniques have provided various performance improvement over the traditional scheme [20], they are still impractical for large-scale cloud systems, especially for high-throughput data processing applications.

BFT schemes can be roughly classified into two categories: replica-based and quorum-based. In replica-based approaches, clients contact a primary replica, who forwards client requests to
other replicas, and all replicas need to communicate with each other to agree on the total order of client requests. In quorum-based approaches, clients contact server replicas directly to execute operations.

Castro and Liskov proposed a three-phase commit protocol that is able to tolerate Byzantine faults in state machine replication systems [20], which belongs to replica-based approaches. The scheme requires all replicas process the requests and communicate with each other to achieve consensus and relies on digital signatures and message digests to prevent attacks such as spoofing and replay. The scheme can tolerate Byzantine faults given that no more than one third of the replicas are simultaneously faulty. Some optimizations are also provided to improve the performance. For example, in order to reduce the cost of communication, only one replica needs to send complete processing results back to the client while others just send message digests. In order to reduce message delays, clients can tentatively commit once receive $2f + 1$ tentative replies instead of waiting for all $3f + 1$ replies. It also improves performance of read-only requests by allowing replicas to execute read-only requests immediately in their tentative state.

In view of the unnecessary high overhead of enforcing the three-phase protocol over all client requests even when there is no faulty nodes, Abd-El-Malek, Ganger and et al. proposed a quorum-based Query/Update (Q/U) protocol that provides better throughput and fault-scalability than replicated state machines using agreement-based protocols [8]. The scheme avoids server-to-server broadcast communication and the requirement that all correct servers process every request. Clients send requests to all servers directly and servers reply to clients. If there is no fault, a single phase is enough. However, if there are faulty nodes, clients need additional repair phase and rely on object history set maintained at the clients to achieve ordering. The Q/U protocol requires $5f + 1$ servers to tolerate $f$ Byzantine faulty servers.

Cowling, Myers, and et al. proposed a hybrid Byzantine-fault-tolerant state machine replication protocol, called HQ replication [24], which combines quorum-based and replica-based approaches. The authors pointed out that Q/U performs poorly when there is contention among concurrent write operations because Q/U resorts to exponential back-off to resolve contention, leading to greatly reduced throughput. Thus, HQ uses quorum-based approach when there is no contention, while uses the three-phase protocol [21] to order the contending operations. HQ also adopts some optimizations such as using MACs to replace signatures to reduce computation delay and using TCP instead of UDP to provide reliability.

Zyzzyva [54] reduces the cost of BFT state machine replication by allowing replicas to speculatively execute requests. Replicas adopt the order proposed by the primary replica and respond to clients immediately. When there are faulty replicas, clients have to interact with all replicas again to determine the correct ordering. Zyzzyva also adopts some optimizations such as replacing signatures with MACs and caching out-of-order requests.
In contrast to the BFT schemes, the goal of our service integrity attestation schemes is to detect integrity attacks and pinpoint malicious nodes rather than achieve consensus among replicas. We employ the trusted portal node to perform consistency check on the data processing results. Thus, functionally equivalent service components do not need to communicate with each other. Our schemes perform probabilistic data attestation and comprehensive attestation graph analysis to achieve both scalability and efficiency, which can significantly reduce runtime integrity attestation overhead. Note that in order to achieve scalability, our schemes cannot have a strong guarantee that can always give correct response to every client request. However, our schemes can improve data quality by pinpointing malicious nodes and eliminate them in future data processing. Moreover, our integrity attestation schemes are completely transparent to service replicas and do not require any replica modification. The service instances attested together are not necessarily replicas, which can have different internal implementations.

7.4 Cloud Security

Security protection for cloud systems has recently received much attention [17, 32, 63, 78].

Ristenpart et al. explored the security holes of existing deployed cloud systems, and identified that current cloud deployments of Amazon EC2 are vulnerable to cross-VM side channel attacks [63]. Their experiments show that it is possible to map the internal cloud infrastructure and instantiate new virtual machines so that one virtual machine could be placed with a target virtual machine of particular interests. Attackers can then utilize side channels to learn information about the target virtual machine, such as detecting web traffic patterns and timing keystrokes.

Erway and et al. presented a framework for dynamic Provable Data Possession (PDP) for cloud storage system [32]. The framework extends the PDP model [12] that only supports static files to support provable updates to stored data.

Santos, Gummadi, and Rodrigues presented a trusted cloud computing platform (TCCP) that provides confidential execution of guest virtual machines [66]. Although encryption may be effective in providing data privacy before the data is used at the service providers, it cannot be used to protect the data in computation, since unencrypted data must reside in the memory of the host running the computation and it is hard to prevent privileged users from inspecting or modifying the data. TCCP secures the guest virtual machines by confining the execution of a VM to a trusted node and preserving the VM state privacy and integrity when it is in transit on the network.

Wood, Gerber and Ramakrishnan proposed CloudNet [79], which can seamlessly integrate virtual private networks (VPNs) with cloud computing platforms and make it easy for cloud customers to manage cloud resources. CloudNet adopts the idea of Virtual Private Clouds
(VPC), which are created by taking dynamically configurable pools of cloud resources and connecting them to users’ own infrastructure.

In comparison, our work focuses on a different aspect of security for cloud systems, which is to assure service integrity for dataflow processing applications.
Chapter 8

Conclusion and Future Work

This dissertation focuses on providing service integrity assurance for data processing applications in large-scale multi-tenant cloud systems. Service integrity is one of the top security concerns of cloud users, which threatens both public and private data processing.

We have presented three studies on runtime data-driven service integrity attestation schemes that can efficiently identify service integrity attacks and pinpoint malicious service providers in large-scale cloud systems. Compared to previous approaches such as code attestation and traditional Byzantine Fault Tolerance schemes, our approaches achieve both scalability and applicability, while being completely transparent to the attested services without imposing any special software or hardware requirements.

The organization of this chapter is as follows. First, we summarize our major contributions. Then, we briefly discuss possible future research directions.

8.1 Contributions

In an attempt to design and implement a scalable runtime service integrity attestation framework for data processing applications in large-scale cloud systems, this dissertation makes the following specific contributions:

• \textit{RunTest}: We developed RunTest, a scalable runtime integrity attestation framework that provides light-weight application-level attestation to dynamically verify the integrity of dataflow processing services and pinpoint malicious service providers in multi-tenant cloud systems. We proposed a novel \textit{randomized} service integrity attestation scheme that employs redundant functionally-equivalent service components and relies on result consistency check to validate service integrity. Furthermore, we proposed an \textit{attestation graph model} to aggregate result consistency information and a \textit{clique based attestation}
graph analysis algorithm to pinpoint malicious service providers and recognize colluding attack patterns.

- **IntTest**: We developed IntTest, an integrated service integrity attestation framework that can efficiently verify the integrity of dataflow processing services and pinpoint malicious service providers within a large-scale cloud infrastructure. Besides the per-function consistency information between different service providers, IntTest also collects the inconsistency relationships between service providers across different functions to expose malicious nodes more efficiently and effectively. Compared to RunTest, IntTest can effectively detect more challenging colluding attacks and mitigate false alarms with more relaxed assumption than RunTest.

- **AdapTest**: We developed AdapTest, an adaptive service integrity attestation scheme that relies on trust management to expedite the pinpointing of malicious service providers in multi-tenant cloud systems. AdapTest dynamically evaluates the trustworthiness of different services based on previous attestation results, adaptively selects services to attest, and provides optimized attestation for multi-hop data processing services. Furthermore, AdapTest can significantly reduce attestation overhead, shorten detection delay, and automatically detect and correct corrupted data processing results produced by the cloud system.

- We have implemented RunTest, IntTest and AdapTest on top of the IBM System S stream processing system, a production system that provides high-performance in-memory stream processing. We tested the systems on the NCSU virtual computing lab, a production virtualized computing cluster that operates in a similar way as Amazon EC2. We evaluated both the detection rate and the performance overhead, and compared the schemes with full-time majority voting schemes. Our experimental results show that these service integrity attestation schemes are effective and impose low performance overhead for dataflow processing in cloud systems.

- We have identified two other security attacks besides service integrity attack toward dataflow processing in cloud systems, namely, data attacks and dataflow topology attacks. We have provided efficient and effective countermeasures to protect data and topology and mitigate malicious attacks.

### 8.2 Future Work

Although many advances of security mechanisms for protecting service integrity in multi-tenant cloud systems have been achieved, open questions still exist and need further investigation. We
discuss several of the most interesting ones as follows.

- **Attestation for Stateful and Non-Deterministic Services**: Our service integrity attestation schemes have assumed that the attested services are stateless and deterministic. However, some data processing functions are stateful and/or non-deterministic. For example, the computing results of a window-based stream operator, such as aggregation and join, rely on the window status, essentially, the entire tuples within the window. Although we have provided solutions for attesting window-based stream operators, further investigation is still needed to search for more efficient and effective way to attest stateful and non-deterministic services.

- **Attestation for Web Services**: Our work has focused on service integrity attestation schemes for dataflow processing applications. However, there are some other services that have unique characteristics such that existing techniques cannot be simply applied. For example, web services, currently as the dominate services running on Amazon EC2, may involve both session states shared by different web server replicas and persistent states stored in the database. It is hard to role back database for attestation. Thus, it is non-trivial to apply attestation techniques to web services.

- **Profiling and Prediction for Service Integrity Attacks**: Our service integrity attestation schemes belong to reactive approaches in that they aim to detect service integrity attacks and pinpoint malicious service providers after the attack actually happens. It would be interesting to investigate proactive approaches, which can predict attacks or detect vulnerabilities before an actual attack is launched. There have been some related existing work that focuses on profiling malware or kernel rootkit behavior [15, 56, 62]. However, these approaches focus on a single computer system, without leveraging the homogeneity feature in cloud computing systems. Moreover, these approaches cannot predict future attacks.
REFERENCES


