Towards Secure DataFlow Processing in Open Distributed Systems

Juan Du, Wei Wei, Xiaohui Gu, Ting Yu
Department of Computer Science
North Carolina State University
{jdu,wwei5}@ ncsu.edu, {gu,yu}@ csc.ncsu.edu

ABSTRACT

Open distributed systems such as service oriented architecture and cloud computing have emerged as promising platforms to deliver software as a service to users. However, for many security sensitive applications such as critical data processing, trust management poses significant challenges for migrating those critical applications into open distributed systems. In this paper, we present the design and implementation of a new secure dataflow processing system that aims at providing trustworthy continuous data processing in multi-party open distributed systems. We identify a set of major security attacks that can compromise the integrity of dataflow processing and provide effective protection mechanisms to counter those attacks. We have implemented a prototype of the secure dataflow processing framework and tested it on the PlanetLab testbed. Our experimental results show that our protection schemes are effective and impose low performance impact for dataflow processing in large-scale open distributed systems.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: General

General Terms

Security, Management, Verification

Keywords

Secure Data Processing, Secure Component Composition, Cloud Computing, Service Oriented Architecture

1. INTRODUCTION

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) [2], Service Oriented Architecture (SOA) [5, 8], and Cloud Computing [1], service oriented open distributed systems have emerged as cost-effective platforms for users to access various software applications as services via Internet. Users no longer need to maintain complicated hardware and software infrastructures but can tap into the open distributed system to access various software services in an on-demand fashion.

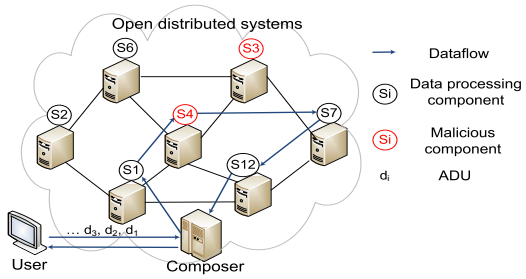

Figure 1: Dataflow processing in open distributed systems.

Many emerging applications, such as network traffic monitoring for intrusion detection, sensor data analysis, and audio/video surveillance, require sophisticated real-time processing over dataflows [4, 11, 15, 17]. Open distributed systems provide highly scalable and available infrastructures for running resource-intensive and quality-sensitive dataflow applications.

An open distributed system supporting dataflow processing applications often consists of many domain-specific data processing service providers, illustrated by Figure 1. Each service provider provides a set of data processing components. A component s_i is a self-contained software unit providing a certain dataflow processing function f_i. Each component can have one or more input ports for receiving input application data units (ADUs), denoted by d_i, and one or more output ports to emit output ADUs. The composer acts as a portal service provider, which interacts with dataflow application users directly and is responsible for dynamically selecting and composing processing components based on user's function and quality-of-service (QoS) requirements such as delay [13]. Users can push a stream of ADUs through the composer into the open distributed system and acquire a set of desired data processing functions such as correlation, filtering, or any other stream mining services [16]. The composer sends *source ADUs* received from the user to the first-hop component. Each component emits intermediate result ADUs called *derived ADUs*. Finally, the last-hop service component reports the final results to the composer that can then forward the final results to the user. Due to the natural redundancy of open distributed systems, there often exist multiple candidate composition plans for the same set of dataflow processing functions. For example, the same processing function can be offered by multiple service providers. The same dataflow processing application can also be delivered by multiple dataflow topologies because of exchangeable composition orders (e.g., $f_1 \rightarrow f_2 = f_2 \rightarrow f_1$), which is called polymorphic dataflow topology [12].

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-

STC'09, November 13, 2009, Chicago, Illinois, USA.

service (QoS) requirements and load balancing objectives [4, 11, 13, 15–17]. 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. However, in open distributed systems consisting of multi-party service providers, we can no longer assume that all processing components are trustworthy. For example, dataflow processing components may include security holes that can be exploited by attackers. Attackers can also pretend to be a legitimate service provider to compromise dataflow processing.

Trust management for distributed systems has been studied under different context, such as PeerReview [14] for distributed messaging systems, the suite of security guards proposed by Srivatsa and Liu for publish-subscribe systems [19], TVDc [7] for virtualized datacenters, and hierarchical Byzantine fault-tolerant replication architecture proposed by Amir et. al. for systems that span multiple wide area sites [6]. Different from previous work, our research focuses on providing efficient yet scalable trust management framework for processing dataflow applications in open distributed systems. Remote attestation techniques [18, 20], ensure that a remote software platform is running code that is not compromised and altered by attackers, which can be used to protect credentials of individual components and are complementary to our work.

We identify a set of major security attacks that can compromise the integrity of dataflow processing in open distributed 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) *ADU attacks* where a malicious component may alter input or output ADUs. For example, in Figure 1, a malicious component may alter d_1 or drop d_1. Other data handling attacks include dropping output ADUs, substituting correct ADUs with fake ones, injecting bogus ADUs, and replaying old ADUs; 2) *dataflow topology attacks* where a malicious component may change the topology of a dataflow application. For example, component s_4 may insert an additional hop to the topology by forwarding its output ADUs to its colluder s_3; and 3) *function integrity attacks* where a malicious component may perform an arbitrary data processing function instead of its advertised one. In composed dataflow applications, there can be multiple malicious components providing falsified dataflow processing functions concurrently.

As countermeasures to those attacks, we present a set of *data-centric* and *light-weight* protection schemes to achieve secure yet scalable dataflow processing. By "data-centric", we mean that our scheme focuses on protecting and verifying the authenticity of data processing results because dataflow users are only concerned about the correctness of final data processing results instead of the data processing procedure. By "light-weight", we mean that our scheme strives to impose minimum overhead since dataflow processing systems are often load-intensive. Specifically, we make the following major contributions.

- We present a *provenance-based ADU protection scheme* that enforces processing components to provide "receipts" for each input ADU they receive and keep "evidence" for each ADU they produce. This protocol can effectively counter ADU attacks in distributed dataflow processing.
- We describe a *cascading dataflow topology encryption* scheme to protect both confidentiality and integrity of dataflow topologies. Our topology encryption scheme assures that each processing component knows nothing about the whole dataflow topology except its upstream and downstream components and no one can change the topology without being detected.
- We present a *randomized consistency check* scheme to achieve scalable dataflow processing integrity verification. By randomly duplicating a subset of ADUs and processing them using randomly selected functionally equivalent components or dataflow topologies, we can detect malicious behavior through result consistency check with high detection probability and small overhead.

We conduct analytical study to quantify performance impact of our secure dataflow processing schemes. We further implement a prototype of the proposed secure dataflow processing system and test it on PlanetLab, a wide-area network testbed [3]. Our experimental results show that the proposed scheme is effective and imposes low performance overhead to the distributed dataflow processing system.

The rest of the paper is organized as follows. Section 2 presents the design of our secure dataflow processing schemes. Section 3 presents the prototype implementation and experimental results. Finally, the paper concludes in Section 4.

2. DESIGN AND ALGORITHMS

In this section, we present the design and algorithm details of our secure dataflow processing system including the provenance-based ADU protection scheme, the cascading dataflow topology encryption scheme, and the randomized consistency check scheme for detecting function integrity attacks. We assume that both the composer and component service providers have public/private key pairs bound to themselves. With their public/private key pairs, they can encrypt, decrypt, and sign data. A party cannot forge others' signatures or decrypt data encrypted using others' public keys.

2.1 Overview

Our security protection schemes provide confidentiality and integrity for dataflow processing applications by imposing light-weight processing on both composer and service components. The cascading dataflow topology encryption scheme requires the composer to encrypt the existing processing topology to preserve confidentiality. Each component needs to extract next-hop information by decrypting the topology and generate an execution trace for the later verification of actual service topology. The components also need to send a receipt packet to its upstream component as well as generate a provenance evidence to prevent ADU altering and dropping attacks as required by our provenance-based ADU protection scheme. Note that ADU data is encrypted with a shared secret key between interactive components, which is encrypted using the receiving party's public key and sent along with the ADU. Moreover, the composer also performs randomized data attestation to detect function integrity attack. We explain the details of the security mechanisms in the following sections.

2.2 Provenance-based ADU Protection

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

When an upstream component does not receive any receipt from its downstream component, it will ask the composer to forward the

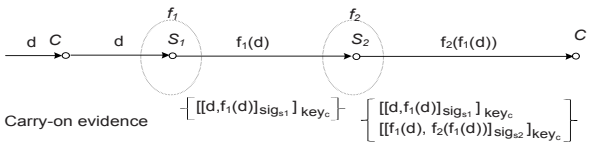

Figure 2: Dataflow processing with carry-on data provenance evidence.

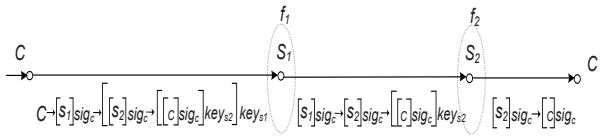

Figure 3: Cascading dataflow topology encryption.

ADU to its downstream component. If the composer does not receive a receipt from the downstream component, we can conclude that the downstream component is either unreliable or untrustworthy. Otherwise, after the composer receives the receipt, the composer forwards it to the upstream component as evidence that the downstream component receives its output ADU. However, if the composer keeps receiving such requests from the same component, it is reasonable for the composer 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 composer busy. In this case, the composer will mark those two components suspicious and employ other components. If the underlying communication channel guarantees in-order message delivery, each component only needs to store the last receipt for each session. Otherwise, the component needs to employ a sliding-window mechanism to record possibly out-of-order receipts.

A malicious component may also drop output ADUs, substitute correct ADUs, or inject bogus ADUs. To counter those attacks, we require each component to create the hash values for both the input ADU it receives and the output ADU 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 ADU as *carry-on evidence*. Compared to carry-on evidence, cached evidence induces smaller overhead to the dataflow processing system. However, the composer needs to dynamically request evidence from distributed components to pinpoint malicious components when the composer detects inconsistency among final data processing results. In contrast, the carry-on evidence allows the composer to perform immediate security diagnosis.

When the composer receives the final result ADU, it can verify the consistency between different components based on either cached data provenance evidence or carry-on data provenance evidence provided by different components. For example, let us consider a dataflow $C \rightarrow s_1 \rightarrow s_2 \rightarrow C$, where C denotes the composer, provisioning functions $f_1 \rightarrow f_2$. Let d denote the source ADU received by the composite dataflow application. Figure 2 shows the dataflow processing with carry-on provenance evidence. The input and output of the first hop component is $(d, f_1(d))$. The component s_1 signs this information and then encrypts it with the composer's public key key_c to get $[[d, f_1(d)]_{sig_{s_1}}]_{key_c}$. If every component is honest, the input of one component should be equal to the output of its preceding component. In addition, since each ADU has a sequence number and session ID, it can be uniquely identified by each component and the composer. Thus, the malicious component cannot launch a successful replay attack by sending old ADUs to legitimate components.

2.3 Dataflow Topology Protection

Our dataflow topology protection scheme has two objectives: i) any component cannot change the dataflow topology; and ii) each participating 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 Ω: $C \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow C$, where C denotes the composer and s_i denotes the ith data processing component. After applying the CTE scheme, the encrypted dataflow topology $\{\Omega\}_E$ becomes $C \rightarrow [s_1]_{sig_c} \rightarrow [[s_2]_{sig_c} \rightarrow [[s_3]_{sig_c} \rightarrow [[C]_{sig_c}]_{key_{s_3}}]_{key_{s_2}}]_{key_{s_1}}$, where sig_c denotes the signature of the composer and key_{s_i} denotes the public key of the component s_i. Figure 3 shows how CTE scheme encrypts a two-hop dataflow topology. Our cascading technology is similar to onion routing [10]. The difference is we also authenticate the topology to ensure that no one can change it.

To prevent malicious components from tampering the dataflow topology, the composer signs each-hop in the dataflow topology using its private key. Thus, it is impossible for malicious components to forge any data processing hop. It is easy to see that our CTE scheme only allows each component to know its previous-hop and next-hop in the dataflow. For example, when the first-hop service provider s_1 receives the topology information Ω: $[[s_2]_{sig_c} \rightarrow [[s_3]_{sig_c} \rightarrow [[C]_{sig_c}]_{key_{s_3}}]_{key_{s_2}}]_{key_{s_1}}$, 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 2.2. 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.

However, if multiple malicious components collude, our topology protection scheme needs to integrate with the ADU protection scheme and function attestation scheme to detect the malicious behavior. Let us consider two collusive malicious components s_1 and s_1'. The malicious component $s1$ can forward its output ADU to it colluder s_1' who is not part of the dataflow. After s_1' processes the ADU, s_1' sends the ADU 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 thought the ADU was routed from s_1 to itself directly as it should be. However, the ADU received by s_2 may be tampered by either s_1 or s_1'. In this case, this becomes a function attack, which will be addressed in section 2.4. On the other hand, if s_1' does not send the ADU 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 composer receives the final result ADU with carry-on evidence, because the composer can check if the result ADU is processed by the right set of components. However, if the final result ADU does not contains carry-on evidence, the composer cannot verify the composition integrity immediately. Although the composer can acquire the cached evidence from all components on the dataflow to verify the dataflow topology later, it is hard for the composer to decide when to do so. Moreover it is time-consuming to receive all the cached evidence from each component. In order to handle this situation, we require that each ADU carries a trace that records all service components where the ADU was processed. Considering the original example, the output ADU 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 ADU from s_2, s_3 will verify $[s_2]_{sig_{s_2}}$ using key_{s_2} to make sure that the previous hop is same as it claims. Thus, the final result ADU 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 composer can use the trace in the final result ADU to verify if the ADU has been processed as expected.

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

not shown in the encrypted dataflow topology.

The cascading topology encryption scheme in this paper 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. Due to space limit, we only focus on the linear composition topology case in this paper.

2.4 Randomized Dataflow Integrity Attestation

To achieve scalable function integrity attack detection in resource-intensive dataflow processing, we propose a randomized consistency check scheme. Suppose the system includes k components $\{s_1, s_2, ..., s_k\}$ that provide the same function f and the source ADU set includes n data items $\{d_1, ..., d_n\}$. During a time period T, the composer randomly duplicates each ADU with probability p_u into r ($r \leq k$) copies and sends all ADUs (i.e., original ADUs and duplicated ADUs) randomly to different functionally equivalent components. Thus, the expected size of the testing data set for consistency check l equals to p_u times the size of total source ADU set. Note that malicious components can hardly predict whether a duplicated ADU has been or will be processed by a truthful component. Thus, if not all functionally equivalent components are malicious and colluding, the composer can detect untruthful execution for f if the result ADUs are inconsistent for the duplicated testing data set $\{d_1{}', ..., d_l{}'\}$. The assumption here is that dataflow processing components are input-deterministic, that is, given the same input ADU or the same set of input ADUs, a truthful component always produce the same output. Many dataflow processing functions [9] fall into this category. The scheme achieves scalability and limited overhead by only duplicating a subset of ADUs and sending them to only a subset of the functionally equivalent components. Moreover, there is no communication overhead for leader election among different functionally equivalent components that is required by traditional Byzantine fault tolerance scheme.

When functionally equivalent components are not available, we propose to explore polymorphic composition topologies (because of the existence of exchangeable composition orders) to perform randomized consistency check. Suppose the dataflow processing application can be delivered through w functionally equivalent dataflow topologies $\{\Omega_1, ..., \Omega_w\}$ if all service providers provide truthful functions. Similar to the previous scheme, the composer randomly duplicates a subset of ADUs (say, $\{d_1{}', ..., d_l{}'\}$) by duplicating each ADU with probability p_u into r ($r \leq w$) copies and route all ADUs including the duplicated ones randomly using different composition topologies. Thus, the composer can still detect function integrity attack via inconsistent results between two functionally equivalent dataflow topologies.

We now quantify the function attack detection capability of our randomized consistency check 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 items are consistent. There are two dimensions of factors that affect the escaping rate.

First, a malicious component may not misbehave all the time. It can escape if not cheating on any of the testing data. Suppose a malicious 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 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 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 redundancy. 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 an ADU as long as it sees other malicious component(s) receiving the same ADU 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 l testing dataset.

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

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

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

$$P = \sum_{i=r-k+m}^{r} P_i = \sum_{i=r-k+m}^{r-1} \frac{\binom{m}{i}\binom{k-m}{r-i}}{\binom{k}{r}}(1 - p_m) + \frac{\binom{m}{r}}{\binom{k}{r}} \quad (2)$$

The number of candidate components k, the size of testing dataset 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 composer can catch function integrity attacks.

With redundant dataflow topologies, detecting function integrity attack is equivalent to finding inconsistent results from the r selected dataflow topologies. Similarly, the escaping rate in non-collusion scenario is $(1 - p_m)^{rl/w}$. In the collusion scenario, suppose there are at most m dataflow topologies that contain malicious components. When all malicious components collude, the escaping rate is similar to the above P_e, except that k is replaced with w.

3. EXPERIMENTAL EVALUATION

3.1 System Implementation

We have implemented a prototype of secure dataflow processing system and tested it on the wide-area network testbed Planetlab [3]. 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 composer is deployed at a pre-defined PlanetLab host, which is responsible for composing dataflow applications and verifying the integrity of the distributed dataflow processing. For simplicity, we only deploy one composer in our experiment. However, our design is readily applicable to large-scale open distributed systems that may require multiple composers. Our dataflow processing system closely follows the design of the IBM System S data stream processing system [15], a commercial dataflow processing system.

For security protection, our dataflow processing system runs on top of a public key infrastructure that is deployed in advance. The composer and component providers 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 gener-

ates 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 implement 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 protection mechanisms for ADU, dataflow topology, and function integrity verification. The difference is that the former stores the data provenance evidence locally on different distributed components, while the latter inserts the data provenance evidence into processed ADUs. When the composer receives the final result ADU, it contains all the data provenance evidence, which can be used for immediate security check.

To evaluate the performance and security of our schemes, we use the following two metrics: i) *dataflow processing delay* and *attack detection probability*. The dataflow processing delay is measured by the average per-ADU turnaround time (i.e., the duration between the time when the first dataflow ADU enters the system and the time when the last dataflow ADU leaves the system over the total number of ADUs processed). We define attack detection probability as the number of detected malicious components over number of all malicious components.

3.2 Results and Analysis

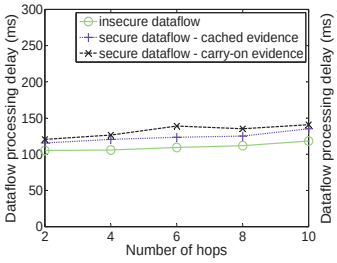

Figure 4: Dataflow processing delay vs. number of hops.

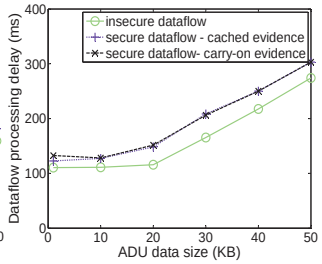

Figure 5: Dataflow processing delay vs. Data size.

Figure 4 shows dataflow processing delay versus the number of hops in service composition for three different approaches given that data sending rate is 15 ADUs per second and data size is 100 bytes. We observe that the overhead of security protection mechanism is about 20ms for both 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 data processing time of consecutive ADUs 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 5 shows dataflow processing delay as a function of ADU data size, where the number of hops is 5 and data sending rate is 15 ADUs/sec. As we can see, the increment of ADU data size dramatically increased the dataflow processing delay, no matter which scheme is used. However, the overhead of enforcing security mechanism shows negligible change with the increase of ADU data size. Moreover, the percentage of the security overhead out of the dataflow processing delay decreases as the size of ADU increases. In addition, since the carry-on evidence in the third scheme is generated and transmitted in the form of hash, the dataflow processing delay in the third scheme is very close to that in the second scheme.

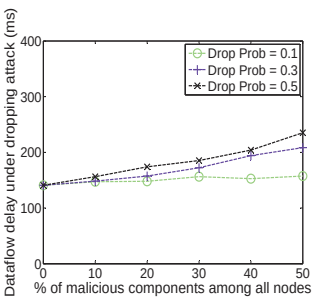

Figure 6: Dataflow processing delay under dropping attack.

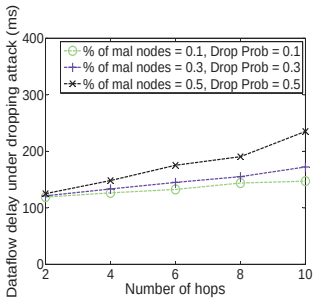

Figure 7: Dataflow processing delay vs. number of hops under dropping attack.

In order to counter ADU dropping attack, we develop the receipt-based communication protocol described in Section 2.2. Figure 6 shows the protocol impact on the dataflow processing delay when it is used to handle dropping attacks. In this experiment, we set the number of service hops to be 10, data sending rate to be 15 ADUs per second, and data size to be 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 composer involved to forward ADUs to downstream components becomes higher. Figure 7 shows the protocol performance impact when the number of service hops, the percentage of malicious nodes, and dropping probability change, where the data sending rate is 15 ADUs per second and data size is 100 bytes.

We now evaluate the efficiency of function integrity attestation scheme. Both non-collusion and collusion scenarios are investigated. In non-collusion scenario, we assume malicious components act on their own and give results independently. While in collusion scenario, the worst case is evaluated where all malicious components serve the same function. Since detection probability is not affected by the number of service hops, we use four hops for the non-collusion scenario and one hop for the collusion scenario. In all scenarios, the number of redundant components k equals to 5, and run time is 30 seconds. Data size is 1KB and data rate is 10 ADUs per second. Obviously, when r equals to 5, the scheme sends attestation traffic to all functionally equivalent components.

Figure 8 and Figure 9 show the percentage of detected malicious components in non-collusion scenario, given 10% and 30% malicious components randomly distributed in the system respectively. Each ADU has a duplication probability p_u of 0.05 and 0.1 respectively, which means the duplicated data set l being 0.05 or 0.1 times of total ADUs. Redundancy degree r varies from 2 to 4. It is obvious to note that even with small duplication probability, such as 0.1, malicious components can be captured well.

Figure 10 compares the percentage of detected malicious components in collusion and non-collusion scenarios, given 1 to 5 malicious components serving the same function respectively. Malicious components always misbehave, and ADU duplication probability is fixed at 0.01. Overall, detection probabilities are higher in non-collusion scenarios than in collusion ones. Detection probability does not change with the number of malicious components in non-collusion scenarios. This is because different malicious components give different results and inconsistencies can be detected as long as an ADU is duplicated. Note that in the non-collusion scenario when r equals to 3, the detection probability is less than 1.

This is because we only duplicate 1% of the ADUs for attestation so that attestation traffic may not pass through a malicious component in a short time. In the collusion scenario, detection probabilities may decrease with number of malicious components, and more redundancies generally produce higher detection probability.

Finally, we evaluate the performance impact of adopting redundant dataflow topologies for function integrity attestation. Figure 11 shows the average dataflow processing delay of using r different topologies, where the secure dataflow with cached evidence scheme is used. When r equals to one, the scheme does not employ any redundant components for consistency check. Our results show that the cost of using two dataflow topologies is very low. With the increase of redundant degree, more components get involved to process the same set of ADUs. Dataflow processing delay is affected by the extra delay of sub-optimal dataflow topologies.

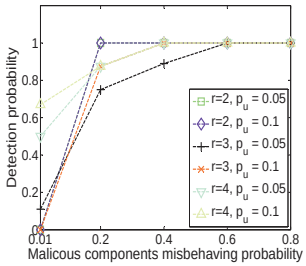

Figure 8: Non-collusion detection probability under 10% malicious nodes.

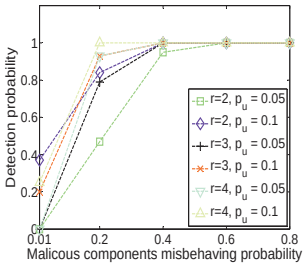

Figure 9: Non-collusion detection probability under 30% malicious nodes.

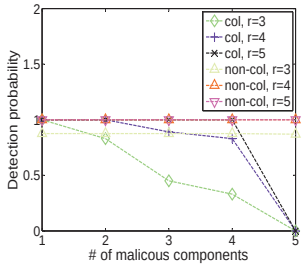

Figure 10: Collusion impact.

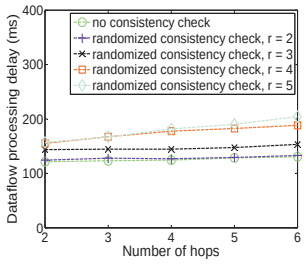

Figure 11: Randomized consistency check overhead.

4. CONCLUSION

In this paper, we have presented the design and implementation of a new security management framework to 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 identify several major security threats. We then present a set of scalable countermeasures to protect the integrity of composed dataflow application delivery including the dataflow topology, inter-component communication, and dataflow processing functions. 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 security management scheme can effectively detect malicious attacks and impose low overhead to the distributed dataflow processing system.

5. ACKNOWLEDGMENT

This work was sponsored in part by U.S. Army Research Office (ARO) under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI), NCSU startup funding and NSF IIS-0430166.

6. REFERENCES

[1] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[2] Software as a Service. *http://en.wikipedia.org/wiki/Software as a Service*.

[3] The PlanetLab. *http://www.planet-lab.org/*.

[4] Daniel J. Abadi and et al. The Design of the Borealis Stream Processing Engine. *Proc. of CIDR*, 2005.

[5] G. Alonso amd F. Casati, H. Kuno, and V. Machiraju. Web Services Concepts, Architectures and Applications Series: Data-Centric Systems and Applications. *Addison-Wesley Professional*, 2002.

[6] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling byzantine fault-tolerant systems to wide area networks. *Proc. of DSN*, pages 159–173, 2006.

[7] Stefan Berger, Ramon Caceres, Dimitrios Pendarakis, Reiner Sailer, Enriquillo Valdez, Ronald Perez, Wayne Schildhauer, and Deepa Srinivasan. Tvdc: Managing security in the trusted virtual datacenter. *ACM SIGOPS Operating Systems Review*, 42(1):40–47, 2008.

[8] T. Erl. Service-Oriented Architecture (SOA): Concepts, Technology, and Design. *Prentice Hall*, 2005.

[9] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: the system s declarative stream processing engine. *Proc. of SIGMOD*, April 2008.

[10] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing for anonymous and private internet connections. *Communications of the ACM*, 42:39–41, 1999.

[11] The STREAM Group. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin, 26(1):19-26*, 2003.

[12] X. Gu and K. Nahrstedt. Distributed multimedia service composition with statistical QoS assurances. *IEEE Transactions on Multimedia 8(1): 141-151*, 2006.

[13] X. Gu, P. S. Yu, and K. Nahrstedt. Optimal Component Composition for Scalable Stream Processing. *Proc. of ICDCS*, 2005.

[14] Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. *Proc. of SOSP*, 2007.

[15] N. Jain and et al. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. *Proc. of SIGMOD*, 2006.

[16] K.-L.Wu, P. S. Yu, B. Gedik, Ki. Hildrum, C. C. Aggarwal, E. Bouillet, W. Fan, D. George, X. Gu, G. Luo, and H. Wang. Challenges and Experience in Prototyping a Multi-Modal Stream Analytic and Monitoring Application on System S. *Proc. of VLDB, 1185-1196*, 2007.

[17] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. *Proc. of ICDE'06*, April 2006.

[18] Elaine Shi, Adrian Perrig, and Leendert Van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *Proceedings of SSP*, pages 154–168, 2005.

[19] Mudhakar Srivatsa and Ling Liu. Securing publish-subscribe overlay services with eventguard. *Proc. of ACM CCS*, 2005.

[20] Trusted computing group. https://www.trustedcomputinggroup.org/home.