Managing Data Access on Clouds: A Generic Framework for Enforcing Security Policies

Cristina Basescu, Alexandra Carpen-Amarie, Catalin Leordeanu, Alexandru Costan, Gabriel Antoniu

Managing Data Access on Clouds: A Generic Framework for Enforcing Security Policies

Cristina Bǎsescu,
Cătălin Leordeanu, Alexandru Costan
Department of Computer Science,
University Politehnica of Bucharest
cristina.basescu@cti.pub.ro,
{catalin.leordeanu, alexandru.costan}@cs.pub.ro

Alexandra Carpen-Amarie,
Gabriel Antoniu
Centre Rennes - Bretagne Atlantique,
INRIA/IRISA, France
{alexandra.carpen-amarie,
gabriel.antoniu}@inria.fr

***Abstract*—Recently there has been a great need to provide an adequate security level in Cloud Environments, as they are vulnerable to various attacks. Malicious behaviors such as Denial of Service attacks, especially when targeting large-scale data management systems, cannot be detected by typical authentication mechanisms and are responsible for drastically degrading the overall performance of such systems. In this paper we propose a generic security management framework allowing providers of Cloud data management systems to define and enforce complex security policies. This security framework is designed to detect and stop a large array of attacks defined through an expressive policy description language and to be easily interfaced with various data management systems. We show that we can efficiently protect a data storage system, by evaluating our security framework on top of the BlobSeer data management platform. We evaluate the benefits of preventing a DoS attack targeted towards BlobSeer through experiments performed on the Grid'5000 testbed.**

***Keywords*-Cloud computing; Cloud storage service; security; policy enforcement; Denial of Service;**

I. Introduction

As Cloud computing [1] is emerging as a good means to leverage available remote resources in a flexible, scalable and cost-effective way thanks to a usage-based cost model, one of the critical concerns that directly impacts the adoption rate of the Cloud paradigm is security [2]. This currently motivates a large number of research efforts and collaborative projects on this subject. Even though Cloud computing is a relatively new field, some security mechanisms are already in place, most of which have been imported from the Grid computing area. However, simply translating Grid techniques into Clouds may not be enough, as Clouds introduce new assumptions and requirements: Cloud environments rely on *virtualization* and *isolation* of resources, which introduce a need for a different approach.

Let us consider the of case the Nimbus Cloud-Kit [3], which inherited the Grid Security Infrastructure (GSI) [4], widely used in Grids to ensure message integrity and authentication of the communicating entities. In this case, once mutual authentication is performed, a potential threat is that authenticated clients may behave in a malicious way, attempting to damage the system, consume bandwidth or decrease its overall performance by means of operations that they have the appropriate access rights to do. The focus of our research is the detection of such malicious clients that may be performing attacks [5] such as Denial of Service (DoS) attacks, flooding attacks or crawling, despite the typical security mechanisms.

Addressing such security vulnerabilities proves to be non-trivial. In order to minimize management costs and increase efficiency, Cloud providers could benefit from generic security management systems that meet two essential requirements: (1) they can be interfaced with any of the various Cloud systems that exhibit this type of security vulnerabilities and (2) they can handle and detect not only predefined attacks, but also those corresponding to customized security policies. This paper proposes such a generic security management framework, targeted at Cloud data storage systems, which allows providers of Cloud data management systems to define and enforce complex security policies. The generality of this approach comes from the flexibility both in terms of supporting custom security scenarios and interfacing with different Cloud storage systems.

In Section II we discuss the main security mechanisms used in current Cloud data management services. Section III describes two sample scenarios illustrating possible behaviors of malicious clients and presents a global overview of the generic security framework proposed in this paper. Section IV explains how a Cloud data service provider can use our framework to define a security policy and enforce it. As a case study, we show in Section V how the proposed framework has been applied to BlobSeer, a BLOB-

based data management system currently subject to integration efforts in existing Cloud environments. Some experiments are discussed in Section VI. Finally, Section VII concludes the paper and discusses future directions.

II. Related work

Whereas resource control in Grid environments is enforced by system administrators, the situation is different in the context of Clouds [6], where users have the control of the remote virtual resources. This raises some additional security concerns about control policies, as clients have to rely on the security tools of the Cloud service providers.

To take the example of Nimbus [3] again, GSI mechanisms are used to authenticate and authorize client requests, VM image files, resource requests, reservation and usage times for users. Authorization is done based on the role information contained in the issuer's Virtual Organization Membership Service credentials and attributes. This mechanism allows for simple group management, identity assignment, policies enforcement, setting reservation limits and path checks. Moreover, in [7], the authors extend this approach by encrypting the VM images on the client side, allowing the user to retain data control. However, the proposed remedy is only suitable for the storage of VMs, as their transfer is secured through GSI and the start-up relies on the not-always true assumption that involved systems can be trusted. More security mechanisms (e.g., intrusion detectors) are needed to protect the virtual host from attacks. From a more general perspective, there is a need to detect different types of malicious behavior through custom policy enforcement mechanisms.

Hadoop Distributed File System (HDFS) [8], the default back end for the Hadoop Map/Reduce framework [9], implements security as a rudimentary file and directory permission mechanism. Concerning authorization, the permission model is similar to other platforms such as Linux, each file and directory being associated with an owner and a group. Since both clients and servers need to be authenticated for keeping data secure from unauthorized access, HDFS relies on Kerberos [10] as the underlying authentication system. In contrast to Nimbus, which relies on the powerful features of GSI, the main security threats in HDFS arise from the lack of user-to-service authentication, service-to-service authentication and the lack of encryption when sending and storing data. Moreover, even if a typical user does not have full access to the filesystem, HDFS is vulnerable to various attacks that it cannot detect, such as Denial of Service.

In Amazon Simple Storage Service (S3) [11], the data storage and management infrastructure for Amazon's Elastic Compute Cloud [12], the users can decide how, when and to whom the information stored in Amazon Web Services is accessible. Amazon S3 API provides access control lists (ACLs) for write and delete permissions on both objects and objects containers, denoted buckets. Regarding data transfers, data in transit is protected from being intercepted, as the access is allowed only via SSL encrypted endpoints. Although S3 does not encrypt data when it is stored, as in the Nimbus approach, users may encrypt them before uploading so as to make sure the data are not tampered with. However, no high-level security mechanism is available to protect the environment from complex attacks, such as the ones that cannot be prevented by authentication mechanisms.

While all the projects described above rely heavily on authentication and authorization mechanisms, none of them is able to identify users who attempt to harm the system or to detect specific patterns of malicious behavior. We address precisely this goal: we propose a generic policy management system to protect Cloud services from complex attacks that may otherwise remain undetected and affect the overall performance perceived by the clients.

III. Overview

We aim to provide high-level security mechanisms for Cloud storage services, as data access operations are vulnerable against a wide range of security attacks prone to damage the system and to affect its overall data access performance and response time.

A. Detecting malicious access in Cloud storage systems: motivating scenarios

The following scenarios illustrate some representative applications for a Cloud storage platform and examine the inherent security threats of their usage patterns. These motivating scenarios highlight the benefits of complementing the typical Cloud security mechanisms with a security management framework that allows service providers to supervise user actions and restrict activities that fall outside the normal usage.

Cloud storage for video surveillance: Video surveillance cameras typically generate a continuous data flow that requires a large amount of storage space. The data will not be written to a single file, as video surveillance cameras usually store the recordings to different files according to their timestamps. A suitable storage system has to be able to scale to a large number of cameras, each of them concurrently writing huge amounts of data to different files. To leverage these needs for storage capacity, the data can be hosted directly on the Cloud.

In this scenario, an attacker might try a DoS attack on some of the storage nodes by sending a large number of write requests. This would lower the response time of the attacked data storage nodes, thus affecting the rate at which the data can be stored for the entire system. In order to maintain the overall performance at an acceptable level, these attackers must be quickly identified and blocked.

Storing medical records in the Cloud: In this scenario we consider a medical center which stores all the medical records for its patients in the Cloud. The employees have access to all the files, but each of them is supposed to access just the documents related to his work. The main security concern in this case is that we must protect the data from being accessed by unauthorized users. An attacker can impersonate an authorized user by stealing its credentials and then attempt to read all the stored files (crawling). This kind of unexpected behavior (reading all records in a short period of time) has to be detected as being suspect, since it can expose a compromised user. However, this is not a clear indication of an attack since an authorized user may also perform those actions. As a result, this behavior has to be labeled as suspicious, yet it will not result in a punishment for the client until it is correlated with other detected attacks.

Such threat scenarios represent complex attacks that are difficult to detect because they can take different forms, depending on each individual attack. To be able to identify any threat scenario, we have designed a flexible and extensible language to describe the access patterns specific for each type of attack. Moreover, we have developed a security management framework to detect and also block any client attempting an attack described by these patterns.

B. Global Architecture

In order to provide a high-level security mechanism for Cloud systems, we propose a generic framework for both security policies definition and enforcement. Figure 1 illustrates the modular architecture of our framework and the interactions between the components.

The Policy Management module represents the core of the framework, where security policies definition and enforcement takes place. This module is completely independent of the Cloud system, as its input only consists in user activity events monitored from the system.

The User Activity History module is a container for monitoring information describing users' actions. It collects data by employing monitoring mechanisms specific to each storage system and makes them available for the Policy Management module.

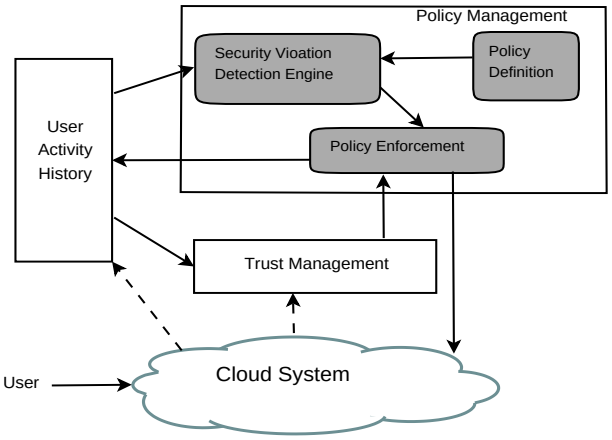


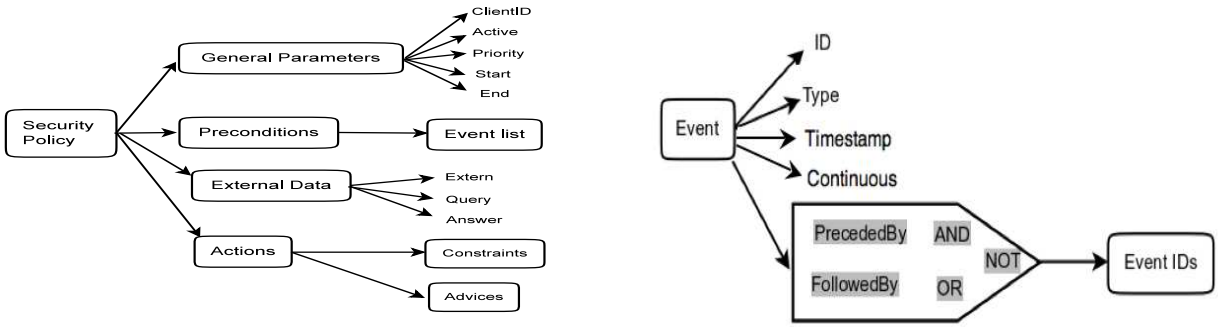
Figure 1. High-level architecture of the security management framework.

The Trust Management module incorporates data about the state of the Cloud system and provides a trust value for each user based on his past actions. The trust value identifies a user as a fair or a malicious one. Furthermore, the trust values enable the system to take custom actions for each detected policy violation, by taking into account the history of each user.

This paper focuses on the policy management core. In order to have an adequate malicious client detection level, we first have to define what kind of behavior is considered inappropriate or dangerous for the system. This is done through the *Policy Definition* component, which provides a generic and easily extensible framework for defining various types of security policies. This step is usually performed before the system starts, however the framework supports the addition of security policies on the fly. The *Security Violation Detection Engine* scans the *User Activity History* in order to find the malicious behavior patterns defined by the security policies. When such an attack is detected, the *Policy enforcement* component is notified and a set of possible feedback actions are forwarded to it. The *Policy enforcement* component is responsible for making a decision based on the state of the system and on the impact of the attempted attack on the typical performance of the system. Such decisions range from preventing the user from further accessing the system to logging the illegal usage into the activity history and decreasing the trust value corresponding to that user.

IV. ZOOM: MANAGING SECURITY POLICIES

In this section we show how we define templates for various attacks and how we map them into security policies and we give an insight on the mechanisms we designed to detect such attacks.



(a) High level representation of a security policy.

(b) Structure of an event.

Figure 2. Defining security policies

A. Defining security policies

In order to detect the various types of attacks that the user actions can expose, our policy management module has to meet a set of requirements:

- the format used to describe the security policies has to be flexible and expressive enough to allow the system administrator to translate any type of attack into a policy that can be understood by the Policy Management Module.
- the extensibility is an essential feature of the security policies, as specific attacks need an enriched policy format according to particular events collected by the user activity history.
- since writing policies is a tedious and error-prone task for administrators, this process has to be automated by means of an API that allows a straightforward definition of security policies compliant with our format.

We defined a hierarchical format for the security policies, so as to comply with the above requirements. On the one hand, each policy contains a set of template user actions that make up a pattern corresponding to a particular security attack. In addition, the policy can specify a set of thresholds that draw the limits between normal behavior that exhibits the same activity pattern and malicious user actions. In order for an attack to be detected, the policy has to be instantiated for a specific user, that is, the activity history of that user has to include recorded actions that match the template sequence provided by the policy. As an example, a DoS attack can be defined by a series of write operations that take place in a short period of time and are initiated by the same client. Therefore, the corresponding policy will describe a write operation as the needed pattern and will specify a duration and the maximum number of write operations considered normal for that duration.

On the other hand, a security policy has to specify a set of actions that are forwarded to the *Policy Enforcement* module when the policy is instantiated and thus a malicious user is identified. These actions range from feedback specific for the Cloud system used to recording the policy violation into the *User Activity History*.

Figure 2(a) illustrates the tree structure of a security policy, which consists of four elements:

The template set of user actions. The *Preconditions* element encloses the list of user actions that describe the pattern of an attack. Each user action is modeled by an *Event*, described through a set of attributes that identify a particular type of records in the *User Activity History*. To take the example of the DoS attack again, the *Preconditions* will contain only one event, whose *Type* attribute points to the list the recorded write operations in the *User Activity History*.

General Parameters. They are used to differentiate the policies (e.g., *Active*, *Priority*) and to enable the detection module to interpret the events describing the policy by specifying the *Start* and the *End* event.

Actions suggested when the policy is instantiated. The element *Actions* contains several actions made up of *Constraints* and *Advices*. When the sequence of events defined by the policy is matched, the *Security Violation Detection* module will select the *Action* whose *Constraints* are satisfied and propose the associated *Advices* to the *Policy Enforcement* module, which will be in charge of executing them. This approach allows us to define flexible policies that result in a customized feedback that depends on some given constraints.

Interaction with external modules. The element *External Data* allows the current policy to receive auxiliary input data from external modules, in addition to the *User Activity History*. For instance, a policy may need the user's ACL information to make a decision, but this data would be present in an external ACL module

and not in the *User Activity History*. This element enhances the extensibility of the policy format, allowing administrators to plug specific system building blocks to the *Policy management* module.

Figure 2(b) shows the structure of an *Event*. It includes a *Timestamp* that allows for the event's positioning in time with respect to one or more events in the same policy. To this end, the event includes as well *PrecededBy* or *FollowedBy* elements, which enclose references to other events' *ID* field. Moreover, in order to have a more flexible policy definition language, the referenced events can be grouped by means of logical operations such as *AND*, *OR* or *NOT*. In addition, the structure of an *Event* contains an element that models a sequence of user actions that have the same type; for instance, the *Continuous* attribute is used when modeling DoS attacks, for which the detection module has to look for a large number of similar write operations. Aside from these basic attributes, each event can be enriched with attributes containing specific information recorded in the *User Activity History* and with associated thresholds, which are enforced when the policy is instantiated.

B. Security Violation Detection Engine

The detection engine is able to handle any type of policy described using the format above, regardless of their complexity or targeted attacks. Its main goal is to search for recorded user actions that match the template events defined by the policy. The attributes are specific to each type of event and they allow the detection engine to identify the required user actions within the user history.

Until now we have described security policies from a static point of view. The *Security Violation Detection Engine* introduces the notion of *partially matched policy* as a policy for which some of the template events are instantiated with real attribute values found in the *User Activity History*.

The detection algorithm receives as input a list of static policies, each having a specific priority. The algorithm attempts to periodically detect attacks, according to the priority of each policy. For each static or partially matched policy, it builds a query to the *User Activity History*, attempting to instantiate the next template event in the policy's *Preconditions*. It adds to the list of partially instantiated policies all the possibilities for continuing the match, according to the query's results. The detection process is complete when all the events in a policy are instantiated, that is the history of the user's actions reflects a chain of events that are specific to the security attack described by the matched policy.

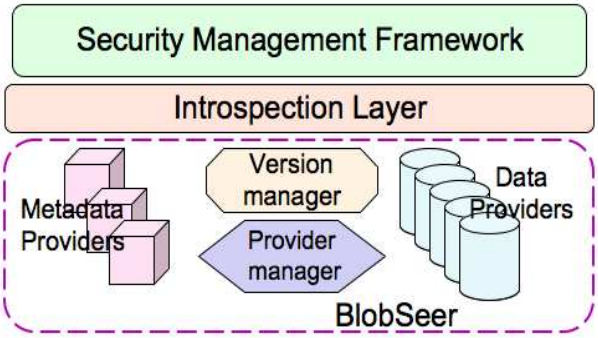


Figure 3. The architecture of the introspective BlobSeer.

V. Case Study: Malicious client detection in BlobSeer

To validate our approach we needed to see how it performs in large scale Cloud environments. Data-intensive applications can benefit from being executed in Cloud environments if the back-end storage services provide several important features, such as a scalable architecture, handling of massive unstructured data, high throughput for data accesses or data-location transparency.

We integrated the proposed Security Management Framework into BlobSeer [13], a data-management system designed for large-scale infrastructures, which addresses these requirements. To fully fit the prerequisites of a standalone storage service within a Cloud infrastructure, BlobSeer has to be able to handle malicious attacks and to isolate users that initialize them. Hence the need for configurable security policies, which can identify the malicious users that attempt to compromise the system, while sustaining the same level of performance for accesses to the stored data.

A. The introspective BlobSeer

BlobSeer is a data-sharing system that addresses the problem of efficiently storing massive data in large-scale distributed environments. It deals with large, unstructured data blocks called BLOBs. They are composed of flat sequences of bytes split into equally-sized strings referred to as *chunks* further in this paper. The design of the BlobSeer system enables two critical features for data-intensive applications. First, it allows fine-grained access to each BLOB's chunks and second, it provides an efficient versioning support that enables highly-concurrent access to data.

The architecture of BlobSeer is based on five actors. The *data providers* store the data chunks in a distributed manner, thus enhancing the support for a large number of concurrent operations. Each BLOB chunk is associated with some metadata, which are stored in a dis-

tributed fashion on the *metadata providers*. The *provider manager* keeps track of the existing data providers and implements the allocation strategies that map new chunks to available data providers. The *version manager* deals with the serialization of the concurrent requests and publishes a new BLOB version for each write operation.

The BlobSeer entity that exposes an interface to user applications is the *client*. It implements the client-side operations for each type of interaction with the BlobSeer system: create BLOBs, read a range of chunks from a BLOB, write or append data to a BLOB.

In [14], we proposed an introspection architecture on top of BlobSeer, which generates and analyses BlobSeer-specific data obtained by monitoring the activity of each of its actors. It was designed to provide support for a self-adaptive behavior of the BlobSeer system, so as to improve its performance and data availability. Its goal was to yield relevant data that can be fed to various self-* components, comprising informations about the state of the system, the state of the physical nodes where the storage providers are deployed and about BlobSeer-specific data that characterizes the stored BLOBs.

In order to provide security mechanisms specifically tuned for BlobSeer, we used the introspection layer to generate the *User Activity History*. The user actions are recorded into a database that includes both the clients' past activity and the information monitored from their current operations. The database represents the input data for the *Policy Management* module, as it exposes each event that occurs in the system, such as writing a chunk on a data provider or requesting the metadata associated with a BLOB.

B. Security Policies

In this section we present a sample security policy, which was represented through an XML language, using tags that follow the structure introduced in Figure 2. For this example we considered the *Cloud storage for video surveillance* scenario that we proposed in Section III.

A typical write operation in BlobSeer consists in (1) splitting the data to be written into chunks, (2) writing the chunks to the data providers and then (3) publishing the write as a new version of the BLOB on the version manager. Therefore, a security policy that detects DoS attacks involves limiting the amount of written data for each client before publishing it as a new version. Basically, the policy has to capture all writes on data providers that were performed in a specific time interval and that were not published by the end of the interval.

According to the structure, the top level XML element contains the *General Parameters*, in this case stating that this policy has a high priority, is active and will be applied to a certain client, identified at runtime. The *Preconditions* tag encloses a list of three event types that play a role in a DoS attack. We identify the start event *w1* that models a write operation, the event *p2* that denotes a publish operation (i.e. a write on the version manager, as indicated by its *type*) and the final event *c1*, which concludes the event sequence to be matched.

```
<securityPolicy id="1_25">
     <clientID rvalue="c" value="c"/>
     <active value="true"/>
     <priority value="1"/>
     <start value="w1"/>
     <end value="c1"/>
     <preconditions>
        <event id="w1" type="prov_write_summary">
        ...
        </event>
        ...
        <event id="p2" type="vman_write">
        ...
        </event>
        <event id="c1" type="check">
        ...
        </event>
     </preconditions>
...
</securityPolicy>
```

The listing below shows the contents of the start event *w1*, the one that identifies the write operations on the data providers.

```
<event id="w1" type="prov_write_summary">
     <blobId id="bId" rvalue="" value="b"/>
     <clientID rvalue="" value="c"/>
     <NoWritesCount id="wsc" rvalue=""/>
     <thresholdNoWrites id="tws" value="100"/>
     <supThresholdNoWrites id="stws" value="200"/>
     <firstDate id="fd" rvalue=""/>
     <lastDate id="ld" rvalue=""/>
     <distance id="dist" value="7000"/>
     <continuous>
        <refEvent value="bId"/>
        <refEvent value="wsc"/>
        <refEvent value="ld"/>
     </continuous>
     ...
     <neg>
        <followedBy>
             <refEvent value="p2"/>
             <count value="1"/>
             <distance value="<= fd + dist"/>
        </followedBy>
     </neg>
</event>
```

Event *w1* verifies the number of written data chunks by defining a specific counter (i.e. *NoWritesCount*) and thresholds (*thresholdNoWrites*) that set limits to the number of writes that can be recorded in a given period of time. In addition, the event specifies two time constraints using the *firstDate* and *lastDate* tags, and the maximum *duration* accepted between them. As the policy has to capture all client actions that match

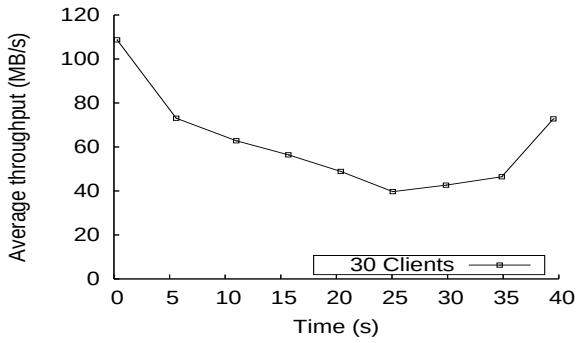


Figure 4. The evolution of the average throughput when 15 clients out of 30 perform malicious writes.

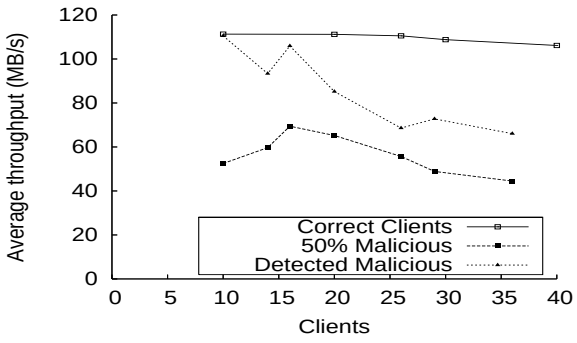


Figure 5. The average throughput under correct and malicious writes.

this event, the *continuous* tag is employed in the event's structure to specify which parameters may vary among the matched writes: the BLOB ID, the total write size and last event's timestamp. Moreover, the event listing clarifies the sequence needed for a match: it states that the write event must not be followed by a publish operation (modeled by the *p2* event) by the end of the time interval defined in the event parameters.

VI. Results

We evaluated the impact of enforcing security policies on top of the BlobSeer system and the performance of the policy management module through a series of large-scale experiments. They were performed on the Grid'5000 [15] testbed, an experimental grid platform comprising 9 geographically-distributed sites.

For all the experiments we employed the same deployment settings for the BlobSeer system. We used a typical configuration that enables the system to store massive amounts of data that can reach the order of TB. It consists of 50 data providers, 15 metadata providers, one provider manager and one version manager. In addition, we used 8 nodes for the monitoring services, which collect the user activity information and are based on the MonALISA grid monitoring system [16]. The *User Activity History* is stored on a dedicated node, which also hosts the *Policy Management* module. Each entity is deployed on a dedicated physical machine. Each experiment is composed of two phases. In the first phase, all BLOBs required by the experiment are created. The second phase of each experiment consists in write operations executed concurrently by the clients, each user generating data in a separate BLOB.

We focused on the video surveillance scenario described in Section III, in which a Cloud storage service is needed to host continuous flows of data recorded by the cameras. The video surveillance cameras are modeled as BlobSeer clients that perform a sequence of write operations. All clients run concurrently and each of them performs 10 writes to BlobSeer, each written string having a size of 256 MB. For our experiments, we assume that each client writes its data in 64 MB chunks, which ensures a constant sustained throughput by the storage system. In this context, we define a DoS attack as a write operation in which the number of chunks written before publishing is much larger than the number of chunks generated by a correct client for the same size of the write. As a consequence, we simulate the DoS attacks as malicious clients that write the same amount of data, i.e. 256 MB, but use a much smaller chunk size: 2 MB.

The first experiment shows the evolution in time of the average throughput of concurrent clients that write to BlobSeer when the system is subject to DoS attacks. For this test we used 30 concurrent clients, each of them performing 10 writes. Half of the clients behave as malicious clients that perform DoS attacks. To study the impact of our security framework, we defined a security policy that sets a limit on the number of chunks a client can write before publishing the full write and we enabled the *Policy Management* module for the experiment. Figure 4 shows that the initial average throughput has a sudden decrease when the malicious clients start attacking the system. As the *Policy Management* module detects the policy violations, it feeds back this information to BlobSeer, enabling it to block the malicious clients, when they issue requests for more data providers to write chunks on. As a consequence, the average throughput for the remaining clients increases back towards its initial value.

The goal of our second experiment is to assess the impact of concurrent DoS attacks on the performance of the storage system. Figure 5 shows the average throughput of concurrent clients that write to BlobSeer, when the number of clients ranges from 10 to 40. The results correspond to three different scenarios: (1) all the clients perform correct writes, (2) 50% of the clients have a malicious behavior and no security mechanism

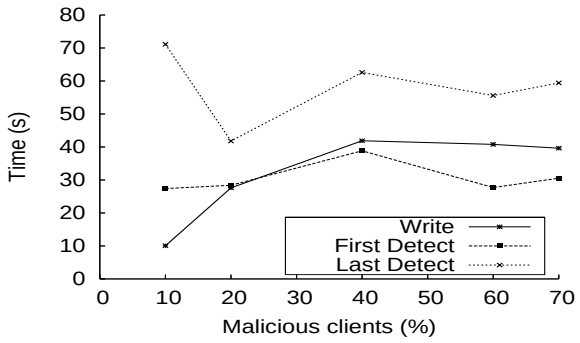


Figure 6. The write duration and the detection delay when 50 concurrent clients write to BlobSeer.

protects the system and (3) 50% of the clients have a malicious behavior and the *Policy Management* module is enabled. When all the concurrent writers act as correct clients, the system is able to maintain a constant average throughput for each client. However, when no security mechanism is employed and half of the clients attempt a DoS attack, the performance is drastically lowered for all the clients that access the system. Further, the results demonstrate that the throughput increases again, once the attackers are blocked by the *Policy Management* framework.

In order to efficiently protect BlobSeer against security threats, the *Policy Management* module has to expose attacks as fast as possible, so as to limit the damage inflicted to the system and to minimize the influence on the correct clients. To evaluate the performance of our policy violation detection component, we measured the detection delay when the percentage of malicious clients increases from 10% to 70% out of a total of 50 clients. For each percentage of malicious clients, Figure 6 displays the duration of the writes performed by all the clients (a sequence of 10 write operations of 256 MB each) and the delays between the beginning of the write operation and the moments when the first and the last malicious clients are detected, respectively. The results show that the time needed to detect and block the malicious clients is comparable to the time it takes to write the data into the system. Therefore, the system is able to promptly react when an attack is initiated and to restore its performance once the attackers are eliminated.

VII. Conclusions and Future work

The emergence of Cloud computing brings forward many challenges that may limit the adoption rate of the Cloud paradigm. In this paper, we addressed a series of security issues, which expose important vulnerabilities of Cloud platforms, and, more specifically, of Cloud data management services. We proposed a generic security management framework that enables Cloud storage providers to define and enforce flexible security policies. The Policy Management module we developed can be adapted to a wide range of Cloud systems, and can process any kind of policy that fits a given base format generated through the Policy Definition module.

As a case study, we applied the proposed framework to BlobSeer, a data management system that can serve as a Cloud storage service. We defined a specific policy to detect DoS attacks in BlobSeer and we evaluated the performance of our framework through large scale experiments. The results show that the Policy Management module meets the requirements of a data storage system in a large-scale deployment: it was able to deal with a large number of simultaneous attacks and to restore and preserve the performance of the target system.

Our future work will focus on more in-depth experiments involving the detection of various types of attacks in the same time. Moreover, we will investigate the limitations of our Security Management framework, with respect to the accuracy of the detection in the case of more complex policies, as well as the probability and the impact of obtaining false positive or false negative results. Another research direction is to further develop the Trust Management component of the security management framework and study the impact it has on the Policy Enforcement decisions for complex scenarios.

References

[1] K. Keahey, R. Figueiredo, J. Fortes *et al.*, "Science Clouds: Early experiences in cloud computing for scientific applications," *In Cloud Computing and Its Application 2008 (CCA -08) Chicago*, 2008.

[2] L. Vaquero, L. Rodero-Merino, J. Caceres *et al.*, "A break in the clouds: towards a cloud definition," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 50–55, 2009.

[3] K. Keahey, M. Tsugawa, A. Matsunaga, and J. Fortes, "Sky computing," *IEEE Internet Computing*, vol. 13, no. 5, pp. 43–51, 2009.

[4] V. Welch, F. Siebenlist, I. Foster *et al.*, "Security for grid services," *HPDC-12*, vol. 0, p. 48, 2003.

[5] M. Jensen, J. Schwenk, N. Gruschka *et al.*, "On technical security issues in cloud computing," in *CLOUD '09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 109–116.

[6] B. Sotomayor, R. S.Montero, I. M. Llorente *et al.*, "Virtual infrastructure management in private and hybrid clouds," *IEEE Internet Computing*, pp. 13(5):14–22, 2009.

[7] M. Descher, P. Masser, T. Feilhauer *et al.*, "Retaining data control to the client in infrastructure clouds," *International Conference on Availability, Reliability and Security*, vol. 0, pp. 9–16, 2009.

[8] "HDFS. the Hadoop distributed file system," http://hadoop.apache.org/common/docs/r0.20.1/hdfs_design.html.

[9] D. Borthakur, *The Hadoop Distributed File System: Architecture and Design*, The Apache Software Foundation, 2007.

[10] B. C. Neuman and T. Ts'o, "Kerberos: An authentication service for computer networks," *IEEE Communications*, vol. 32(9), pp. 33–38, September 1994.

[11] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/.

[12] Amazon Elastic Compute Cloud (EC2), http://aws.amazon.com/ec2/.

[13] B. Nicolae, G. Antoniu, and L. Bougé, "BlobSeer: How to enable efficient versioning for large object storage under heavy access concurrency," in *Data Management in Peer-to-Peer Systems*, St-Petersburg, Russia, 2009.

[14] A. Carpen-Amarie, J. Cai, A. Costan *et al.*, "Bringing introspection into the BlobSeer data-management system using the MonALISA distributed monitoring framework," in *International Workshop on Autonomic Distributed Systems*, Krakow, Poland, 2009.

[15] Y. Jégou, S. Lantéri, J. Leduc *et al.*, "Grid'5000: a large scale and highly reconfigurable experimental grid testbed." *Intl. Journal of High Performance Comp. Applications*, vol. 20, no. 4, pp. 481–494, 2006.

[16] I. Legrand, H. Newman, R. Voicu *et al.*, "MonALISA: An agent based, dynamic service system to monitor, control and optimize grid based applications," in *Computing for High Energy Physics*, Interlaken, Switzerland, 2004.