# Adaptive Data Protection in Distributed Systems

Anna Cinzia Squicciarini
College of Information Sciences and Technology
Pennsylvania State University
University Park, PA 16802

Giuseppe Petracca
College of Information Sciences and Technology
Pennsylvania State University
University Park, PA 16802

Elisa Bertino
Computer Science Department
Purdue University
West Lafayette, IN 47906

## ABSTRACT

Security is an important barrier to wide adoption of distributed systems for sensitive data storage and management. In particular, one unsolved problem is to ensure that customers data protection policies are honored, regardless of where the data is physically stored and how often it is accessed, modified, and duplicated. This issue calls for two requirements to be satisfied. First, data should be managed in accordance to both owners' preferences and to the local regulations that may apply. Second, although multiple copies may exist, a consistent view across copies should be maintained. Toward addressing these issues, in this work we propose innovative policy enforcement techniques for adaptive sharing of users' outsourced data. We introduce the notion of autonomous self-controlling objects (SCO), that by means of object-oriented programming techniques, encapsulate sensitive resources and assure their protection by means of adaptive security policies of various granularity, and synchronization protocols. Through extensive evaluation, we show that our approach is effective and efficiently manages multiple data copies.

## Categories and Subject Descriptors

C.2 [**COMPUTER-COMMUNICATION NETWORKS** ]: Security and protection (e.g., firewalls)

## Keywords

Security, Distributed Systems

## 1. INTRODUCTION

A distributed system is by definition a collection of independent computers that appear to the users of the system as a single coherent system. With the advances of distributed technologies, including cloud computing, wireless networks, grid computing, distributed systems are nowadays proliferating. Data stored in these systems may often encode sensitive information, and if distributed across regions or nations it should be protected as mandated by organizational policies and legal regulations.

Ensuring that policies associated with data distributed across domain (regardless of where the data is physically stored and how often it is accessed, modified, and duplicated) are honored is an important challenge. Depending on the degree of distribution, and the size of the data consumers' base, users' data may be stored in multiple locations, based on providers internal scheduling and management processes. Therefore, as data travels across sites and is modified by multiple parties, it may be replicated and accessed from remote locations. In these cases, access restrictions have to apply not only to changes by the content originators (or content owners) but also to privacy and auditing regulations that are in place in the location where data is stored and managed.

We notice that data replication in distributed systems is the norm. In newer environments, such as cloud computing, data replication is anticipated to gain even more importance, as the reliability requirements of customers increase. It has been envisioned that replication technologies will become part of the storage foundation. Cloud providers are to leverage this technology to meet existing as well as evolving customer requirements. Replication will not only enable data recovery in the cloud, but server recovery in the cloud as well [4, 9].

Toward addressing these issues, in this work we propose innovative policy enforcement techniques for adaptive sharing of users' outsourced data. In particular, we introduce the notion of *self-controlling objects* (SCOs), that encapsulate sensitive resources and assure their protection through the provision of adaptive security policies.

SCO are movable data containers, generated by end users through automated wizards. A given SCO may be distributed across domains if the originator wishes, and content recipients may consume the encapsulated resource at any time. Access privileges will be verified at the time of consumption, based on both the recipient credentials and the context wherein access takes place. Accordingly, the content will be decrypted and accessed on the fly. No centralized policy enforcement engine or decision point is required in order to enforce the security requirements, since the core security modules belong to the object itself.

SCOs provide strong security guarantees, even in presence of multiple copies of the same SCO disseminated across the cloud, and adapt to local compliance requirements. SCOs support a synchronization and update protocol that relies on a subset of the SCO themselves. As part of our solution, we also show how SCOs can either use locally pre-loaded policies or securely accept new policies from trusted authorities.

To effectively accomplish adaptive data protection our design satisfies the following unique properties: strong binding between security policies and data, interoperability, portability, and object security. Protection of encapsulated data is achieved by provisioning owner-specified security policies that are tightly bound with the content, and executable within the SCO. To ensure interoperability, our SCOs are self-contained, and do not require any dedicated software to execute, other than the Java running environment. Self-containment ensures portability, in that SCOs may be moved and replicated without the need of installing dedicated software. To guarantee adaptivity, the policies embedded in the SCO are sensitive to the location and other contextual dimensions that may affect the enforcement process. Finally, object security is guaranteed by advanced cryptographic primitives, such as Ciphertext-Policy Attribute-Based Encryption [2] and Oblivious Hashing for program tracing [14], combined with extended and advanced object-oriented coding techniques. We have implemented a running prototype of the proposed architecture and protocols using Java$^{TM}$-based technologies [11, 21] along with advanced cryptographic protocols and cross-checking techniques to guarantee acceptable trustworthiness of the SCOs. In the paper, we discuss the results of our experiments on a distributed system testbed which includes several physical and virtual nodes installed. Our results demonstrate the resiliency of our architecture even in case of multiple SCO nodes faults.

The rest of the paper is organized as follows. Next section provides background notions. Next, in Section 3, we provide an overview of the notion of SCO. Section 4 discusses the security policies. Section 5 introduces our synchronization algorithms. Section 6 discusses the most important properties achieved by our solution, whereas Section 7 discusses implementation and security of our solution. Section 8 reports experimental results. Section 9 discusses related work and Section 10 concludes the paper.

## 2. CIPHERTEXT-POLICY ATTRIBUTE- BASED ENCRYPTION

In this section, we briefly discuss the basic cryptographic protocol underlying our solution. The Ciphertext-Policy Attribute-Based Encryption (CP-ABE) [2] is a type of public-key encryption scheme.In the CP-ABE scheme, there exists a trusted party that generates a public key which can be used by any entity to encrypt a message irrespective of its recipients. The message sender encrypts the message using the public key. A trusted party computes the private key. Only entities satisfying the rules are able to compute the private key. The schema includes four phases:

- *Setup*: The trusted third party takes as input implicit security parameters and outputs the public key $PK_{ABE}$ of ABE protocol and a master key $MK$. The master key is only known to the trusted party who generated it.

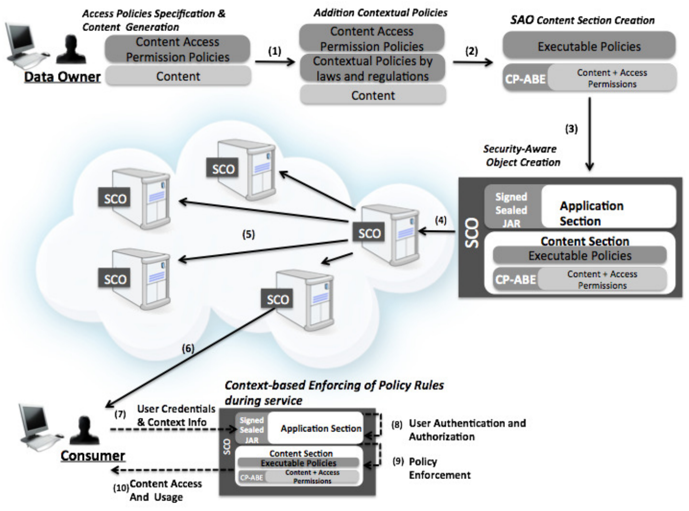

Access Policies Specification & Content Generation
Content Access Permission Policies
Content
Data Owner
(1)
Addition Contextual Policies
Content Access Permission Policies
Contextual Policies by laws and regulations
Content
(2)
SAO Content Section Creation
Executable Policies
CP-ABE
Content + Access Permissions
(3)
Security-Aware Object Creation
SCO
Signed Sealed JAR
Application Section
Content Section
Executable Policies
CP-ABE
Content + Access Permissions
(4)
SCO
SCO
SCO
SCO
SCO
(5)
(6)
Context-based Enforcing of Policy Rules during service
(7) User Credentials & Context Info
Consumer
SCO
Signed Sealed JAR
Application Section
Content Section
Executable Policies
CP-ABE
Content + Access Permissions
(8) User Authentication and Authorization
(9) Policy Enforcement
(10) Content Access And Usage

Figure 1: Overall Approach

- *Encrypt(PK, M, A)*: The encryption algorithm takes as input the public key $PK$, a message $M$, and an access structure $A$ over the universe of desired attributes. The algorithm encrypts $M$ and produces a ciphertext $M'$ which implicitly contains the access structure $A$. $M'$ can only be decrypted by the entity which possesses a set of attributes that satisfy the access structure.
- *KeyGen(MK,S)*: The private key generation algorithm is carried out by the trusted entity. The algorithm takes as input the master key $MK$ and a set of attribute values, $S$ on which access control would be achieved. The outcome of $KeyGen(MK, S)$ is a private key $SK_S$.
- *Decrypt(M', SK)*: The decryption algorithm takes as input the ciphertext $M'$, and the private key $SK_S$. The output is the original message $M$.

## 3. THE SCO-BASED SOLUTION

We illustrate the functionality offered by our policy-protection framework through a use-case (see Figure 1). We particularly focus on a cloud computing domain, wherein data is replicated to facilitate users' access across location. A SCO is generated by a content originator subscribed to a cloud provider. The content originator triggers the SCO generation process as he places sensitive content (e.g. text, video or image file) on the Cloud Service Provider he is subscribed to. SCOs may protect files of different kind, including images, text, and media content.

The originator indicates the file to protect as well as its security policies (step 1), which are encoded and embedded in the SCO. Notice that the security policies can include constructs to specify access control, authentication and usage controls. Furthermore, they are enforced according to contextual criteria defined by the content owners, to ensure, for example, compliance with location-specific regulations and policies. In addition to the content owner's rules, prior to being signed and sealed, the SCO is loaded with rules whose enforcement guarantees compliance with security and data disclosure regulations (e.g. auditing requirements), as shown in step 3 and 4. The SCO is then stored at a single service provider (SP), and possibly duplicated for greater dependability and availability. The content stored in each SCO may be accessed at time by authorized users, SPs for processing and analysis, and auditors, to meet compliance requirements. Each time access to the SCO protected content is attempted, its policy is evaluated according to the requestor's credential and location (step 5). If permitted by the policy, the content is then rendered according to the granted privilege.

Notice that as SCO instances are accessed, replicated and moved across different locations, two requirements must be always addressed: (1) The policy enforced by the SCO must comply with the legal regulations of the SCO storage location. For example, the SCO is stored in a EU server, and therefore, data disclosure laws apply. (2) Modifications to the SCO content file must be propagated to all SCO copies, for obvious consistency reasons. To address the first requirement, the policy in the SCO is organized in rules of varying priority, and may include specific applicability requirements, defined in terms of pre-defined context-specific rules. New high-priority rules may be added to the SCO, using a third-party invocation protocol.To address the second requirement, the SCO applies a quick synchronization protocol to all the available copies. The update process entails a coordinated cooperative protocol among one or multiple copies of the same SCO and ensures that the most up to date content version is rendered to the SCO being accessed.

## 4. POLICIES

We now introduce a simple representation of the security policies supported by our protection mechanism and their enforcement process. Our representation allows to enforce simple access control and usage requirements. The design of both the language and the resolution algorithm is partly inspired by the XACML language, which represents the defacto standard in terms of access control.

### 4.1 Policy Specification

We define a security policy as a collection of rules, i.e. $p = \{r_1, \ldots, r_n\}, n \geq 1$. Each rule controls access and usage of a given content by a certain user (or collection of users). It may either target a single content item $f$ (e.g. file) stored within the SCO, or if the SCO stores multiple content elements (i.e. files), it may refer to all the files in the SCO. For simplicity, we assume that the policy rules within a same SCO are atomic, do not overlap, and do not conflict among one another.

Definition 4.1 (Policy Rule). *A policy rule $r$ is a tuple* $\langle \texttt{s}, \texttt{o}, \texttt{effect:act}, \texttt{cond}||\texttt{Cont}\rangle$*, where:*

- `s` *is the subject element. It can be the name of a user, or of any collection of users, e.g. subscriber or Amazon SP2, or a special user collection anyuser representing all users.*
- `o` *is a content item or a set of content items.*
- `effect:act` *denotes the effect of the rule in case of truthful evaluation and impacts the selection of the rule when multiple rules exist in the same policy. The* `effect` *takes one of the following values:* `AbsolutePermit`, `AbsoluteDeny`, `FinalPermit`. `act` *denotes the privilege applied to the content, in terms of its access and consumption. It is left null in case of* `AbsoluteDeny` *effect.*
- `cond` *is an optional element that further refines the applicability of the rule to the subject in the rule through the specification of predicates on the subject attributes (combined with anyuser) and/or the object attributes.*
- `cont` *is a set of boolean conjunctive conditions defined against a set of pre-defined contextual variables.*

The context component of our rule model is the key to support adaptive SCO, as this component specifies the contextual conditions controlling the activation of rules in the security policy. In the current version of the language, we support two types of contexts: the *Environment* context, that depends on the characteristics of the host system, and the *Location* context that depends on the subject location. Each context type consists of a set of pre-defined variables. Examples include *IP address* for location conditions, and *OS* or *System Version* for the environment conditions.

### 4.2 Rules Enforcement Process

The rules enforcement process consists two steps. The first step identifies a set of applicable rules, as not all rules are always applicable to all contexts. The second step enforces the highest priority rule among the set of applicable rules identified by the first step.

*Rules Applicability* A rule `r` in a SCO policy is applicable to an access request if and only if: (a) the user meets the subject specification in (`r.s`); and (b) the user's and objects attributes verify true the condition set `r.cond`, and (c) the context conditions `r.Cont`, if present.

To verify the applicability of a given rule, at the time of an access request, the policy enforcement engine at the SCO obtains the user credential and verifies its validity. It further verifies whether the user attributes satisfy the access conditions. Next, the SCO collects information required in order to establish if the context definitions are satisfied. To achieve this, various actions are undertaken, depending upon the specific context conditions to be verified. Context primitives deployed within the SCO have direct access to the host system (e.g. a global clock to check temporal conditions in the Environment context) or they use metadata carried by the actions. For example, the location of the SP can be determined using the IP address. The SCO performs an IP lookup and uses the range of the IP address to find the most probable location of the SP, so as to match it against the location defined in the condition constraint.

Notice that we assume that each SCO copy is always able to gather the location information (i.e. the IP address and a free port number) of the machine on which it is running. This information is needed also to ensure the capability for a SCO copy to communicate with other copies. Even in an environment where the applications execution is controlled, these two pieces of information must always be available. For this purpose, protected APIs can be used (see Section 7.1), to ensure the correct use of such sensitive information solely for the purpose of of communications among SCOs. If other contextual information is necessary for the evaluation of policy rules is not available to the SCO, the policy rules will be defined as indeterminate. This will block the use of the SCO copy for security reasons.

*Rule Selection* Upon determining the set of applicable rules, the rule-selection algorithm is executed (see Appendix A for a detailed flow). By definition, each rule can only have one of three effects: `AbsolutePermit, AbsoluteDeny, FinalPermit`.

1. An applicable rule whose effect is `AbsolutePermit` has the highest priority, that is, access is granted regardless of the effects of other applicable rules. The motivation for the `AbsolutePermit` rule is that access requests required by law enforcement institutions or national security agencies should always be able to circumvent the access restrictions specified by the owner and organizational policies. We assume that `AbsolutePermit` rules are always pre-loaded within the SCO, and defined to meet the regulatory requirements imposed by countries and jurisdictional locations where the content may move to. Additional `AbsolutePermit` rules may also be dynamically added to the SCO, as it moves in unforeseen locations. In such case, rules are obtained from a third party (see Section 7.1). The presence of context in rules with an `AbsolutePermit` effect has the ability to confine the geographic locations under which a given regulation is enforced. For example, the rule $\langle \mathtt{anyuser}, \mathtt{f}_1, \mathtt{AbsolutePermit}:\mathtt{read}, \mathtt{subject.CountryOfOrigin} =' \mathtt{France}', \mathtt{role} = \mathtt{Auditor}|| \mathtt{location} = \mathtt{France}\rangle$ specifies that any French auditor can read file $f_1$ when the file in located in France.
2. If no `AbsolutePermit` rule is applicable, the `AbsoluteDeny` rules are considered. If one `AbsoluteDeny` is applicable, access is denied. For example, a rule may specify that accesses must blocked from certain geographical areas, considered unsafe or legally prohibited (e.g. for copyright agreements). An example of rule with such effect is as follows: $\langle \mathtt{anyuser}, \mathtt{f}_1, \mathtt{AbsoluteDeny}: \mathtt{read}, \mathtt{subject.location} = \mathtt{China}||\mathtt{location} = \mathtt{China}\rangle$.
3. If neither `AbsolutePermit` rules or `AbsoluteDeny` rules are applicable, the most specific applicable rule with a `FinalPermit` is evaluated. The effect of such rules is to represent criteria that are preferred by content owner and that are to be met in order to obtain access. These rules may also include context conditions.
4. Finally, if no applicable rule exists, access is denied; the "deny takes precedence" principle is adopted.

Upon a positive access control decision (i.e. an `AbsolutePermit` or an `FinalPermit` decision) by a rule $r$, the access is granted according to the specified action set in `r.act`. The action set can vary according to the specific content type hosted within the SCO. For example, for image files, allowed actions are *view*, *edit*, *download*.

Architecturally, rules are translated into executable Java policy files and access structures for encryption (see Section 7.1), according to the cryptographic protocol used for content protection and access. Possibly applicable rules are pre-loaded in each SCO as it is created, based on the specific SP's servers' locations and anticipated location access. We assume that rules with a `FinalPermit` effect entered by the content originator do not change, but additional `AbsolutePermit` and `AbsoluteDeny` rules may need to be added to the SCO as it is accessed from unknown locations whereby `AbsolutePermit` and `AbsoluteDeny` rules are applicable.

## 5. SYNCHRONIZATION AND VERSIONING

SCOs may be replicated for a number of reasons, including workload distribution, disaster recovery, efficiency. Given the unique nature of SCOs, as SCO copies are generated, it is crucial to guarantee that the same policies and usage conditions apply for the object replica.

By replicating a SCO, a network of SCOs is inherently created (referred to as SCON), whereby nodes are SCO copies and edges their connections. To ensure consistency and proper policy enforcement, a distributed synchronization mechanism is to be deployed. The synchronization mechanism should be scalable, and incur in a low-traffic network overhead. A simple coverage algorithm can synchronize the content of all the SCON nodes upon change. However, if the number of nodes in the SCON increases drastically, the synchronization may incur in poor performance. Conversely, a fully centralized system may not scale effectively, if SCOs are spread across different locations. Therefore, we need an efficient structure to guarantee full synchronization of the SCO copies while at the same time reducing the number of messages exchanged among copies. Our solution lies in the notion of *SCON View*, representing an elastic view $V$ of nodes in the SCON. Nodes in $V$ are SCOs, and are referred to as *Master* nodes, in that they are always synchronized. The remaining nodes in the network are *Slave* nodes and are instead synchronized only after contacting a Master node.

### 5.1 SCO Network Creation

Starting from the generic SCON of identical copies, we build a network's view as the basis for our synchronization protocol. First, notice that SCOs keep track of how many copies exist in the SCON, by maintaining a shared counter of the known SCO copies in a local and encrypted registry *Network Management Information (NMI)*, which is updated each time a new copy is created and later synchronized. To build the network, we apply the following procedure. As mentioned, each SCO can either be a Slave node or a Master node. At the time of creation of the SCO, the first $\alpha$ of its identical SCO copies are by default Master nodes. Starting from the $\alpha+1$th copy, the subsequently generated copies will be Slave nodes up until the $\omega\alpha$ threshold is reached. For example, let $\alpha$ be equal to 5, and assume that 6 copies of the same SCO exist. Let $\omega$ be equal to 2. Regardless of the actual synchronization mechanism, copies from 1 to 5 are synchronized at each change, as they are Master copies, while the copy number 6 in order to be synchronized must communicate with one of the Master copies. As the number of SCO copies grows we continue to maintain an acceptable ratio between overall SCO copies and Master copies, in order to ensure good performance of the synchronization system. Suppose that we need $\alpha$ Master copies every $\omega\alpha$ SCO copies. Let $n$ be the current total number of SCO copies. Let the nodes be sorted by the time of creation, by means of a timestamp. Upon creating a new instance of the SCO, the following *node-ratio rule* applies:

$$\begin{cases} k(\omega\alpha) \le n \le k(\omega\alpha) + \alpha & n_{th}\ \textit{node is Master} \\ k(\omega\alpha) + \alpha < n < k(\omega\alpha) + \omega\alpha & n_{th}\ \textit{node is Slave} \end{cases} \tag{1}$$

$k$ represents the Master/Slave ratio according to the nodes' number. It is incremented each time $n = k(\omega\alpha) + \omega\alpha$. Upon generating a SCO instance, its connection to existing SCOs are to be established. SCO Masters and Slaves each maintain information about some of the nodes in the SCON. Precisely, each time a Master is generated, it stores its current local position (IP and port number) and uses the originally available position and port number to connect with its original version. As the new copy informs its parent node, the SCO updates its NMI registry by adding a new record, up until a set threshold of known copies, discussed next. It then propagates the information to the Master nodes merging the list with the new records stored at each Master. Until there are less than $\omega\alpha$ Master nodes in the SCON, all nodes are completely connected. From the $\alpha+1$ Master, each node, whether Master or Slave, must be connected to Master nodes with a degree $\gamma$.

Definition 5.1 (Nodes Connectivity). *Let $N$ be the set of copies of a same SCO, distributed across multiple SPs. Let $M$ be the set of Master nodes such that $M \subseteq N$ and $e = (n, n')$ be a connection between two nodes. For each node $n \in N$ the following rule applies:*

$$\begin{cases} \exists e = (n, n_i) \forall n, n_i \in N & |N| \leq \alpha \\ \exists e_1, \ldots, e_\gamma, e_i = (n, n_i) \wedge n_i \in M, \forall i \in [1, \gamma] & |N| > \alpha \end{cases}$$

By satisfying this simple rule, our system is robust against $\gamma-1$ simultaneous faults of Master nodes. In other words, if we set $\gamma = 3$ and two simultaneous Master nodes faults occur during synchronization, no Master node will be isolated in the SCON. Further, any Slave node will still be able to synchronize by querying the surviving Master node. If a Master is determined to be faulty, the other Master nodes will update their connections in order to meet again the connectivity rule. If the failed Master node recovers, it is then treated like a new node, and it will be a Master or a Slave following the rule of Eq. 1.

Example 5.1. *Assume: $\omega = 2, \alpha = 5, \gamma = 3$. An example of SCON with $n = 8$ is shown in Figure 2 (top). By deleting any $\gamma - 1$ nodes together with their connections, all the remaining nodes in the graph SCON are still connected, satisfying the connectivity rule. Hence, erasing 2 Master nodes (or two connections) leaves the network connected.*

Each connection also has a communication cost $\phi$, estimated by the SCOs by computing the distance with the other connected SCO (through IP lookup). The actual value of $\gamma$, as well as $\alpha$ and $\omega$ can be set empirically. As we show in our experimental evaluation, changes in these parameters only slightly affect the performance of the overall system.
We are now ready to define the SCON. The algorithm for the SCON construction is reported in Appendix A.

Definition 5.2 (SCO Network). *$G = (N, E, W)$ is called SCO network (SCON), with $N$ the set of nodes and $E$ the set of arcs, such that the elements of $E$ are pairs of elements of $N$ ($|E| \subseteq |N|^2$); and $W : E \times \Phi \rightarrow w \in \Re^+$ is the edge labeling function. $W$ maps the communication cost $\phi$ to an edge weight $w$. Each node $n \in V$ is an instance of a SCO, which is abstracted by a tuple of $n =< n_{ID}, m, NMI, l, CS >$. Where $n_{ID}$ is the id, $m \in \{M, S\}$ indicates where the node is a Slave or a Master, $NMI$ is the information needed for network management. Finally, $l$ is the current location (IP, port) of the node, and $CS$ represents the SCO content, i.e. the protected content and policies included in the SCO. $G$ satisfies the following properties:*

- *The $CS$ component is identical upon synchronization across all nodes in $N$.*
- *$\forall n \in N$, $m = M$ ($n$ is a Master) or $m = S$ ($n$ is a slave), according to the node-ratio rule of Equation (1)*
- *$\forall$ $n \in N$ satisfies the connectivity rule of Def. 5.1.*
- *$\forall$ $e_{ij}, \in E$ exists a corresponding weight $w_{ij}$ according to the weight function $W$.*

We denote the components of nodes using the dot notation.
The cost of traversing the *SCON* is $\Theta(SCON) = \sum_{i=0}^{|E|} W(e_i, \Phi_i) = \sum_{i=0}^{|E|} w_i$. For example, $W$ may assign weights according to the following mapping. Let $e_{i,j} = (n_i, n_j)$ be the edge linking $n_i, n_j$. Let $l.IP$ denote the machine's IP address.

$$w_{ij} = \begin{cases} 0 & n_i.m = S \wedge n_j.m = M \\ 1 & n_i.m, n_j.m = M \wedge n_j.l.IP = n_i.l.IP \\ 2 & n_i.m, n_j.m = M \wedge n_j.l.IP \neq n_i.l.IP \end{cases} \quad (2)$$

As we discuss next, the weight of each edge is used by the synchronization algorithm for optimization purposes.

## 5.2 Synchronization Algorithms

The synchronization process varies depending on whether the node to be synchronized is a Master or a Slave. It entails a collaborative protocol among Master nodes while it is a 2-party protocol for SCO slaves.
**Master Synchronization**: The Master node synchronization process is triggered upon receiving an update to the content or to the policy rules of one of its SCOs, or when new network management information is to be exchanged (e.g. a new Master is added). The update is a collaborative process that maps to the problem of finding the minimum spanning tree (MST) of the SCON View. For this purpose, we adopt a parallel search algorithm [7]. The nodes obey the same algorithm and exchange *update* messages with neighbors until the MST is constructed, traversing low-cost connections first. The algorithm determines fragments of the MST and connects them progressively, until one MST if found. At the end of the algorithm execution, each node knows its precedent and successor in the MST, and therefore it is able to pass updates along. The algorithm reaches the optimal solution with a complexity of $O(\sqrt{N} \log^* N)$, where $N$ is the number of nodes.

Example 5.2. *In our example, assume that the updated Master node is the node with number 5 and that the weights of the arcs are assigned according to the rule of Equation 2. Figure 2 shows one possible construction of the MST.*

**Slave Node Synchronization:** Synchronization can occur in two directions: from a Slave node to a Master node and vice versa. In the first case, the Slave node contacts the Master node to communicate the changes occurred at the Slave's content or its policy rules (by an authorized user), so as to allow the Master node to propagate the update among the nodes of the view $V$. In the second case, a Slave node needs to check for updates within the view $V$ of Master nodes to update its local Content Section. This second type of synchronization can occur either periodically, or at each access, to ensure a tighter control. When synchronization is requested, the Slave node connects to any of the Master nodes among the set of adjacent-known Master nodes. The choice of such Master node is random, so as to ensure a fair distribution of workload on Master nodes. We now elaborate on the synchronization processes for the two directions.
*Slave Update:* As a Slave obtains an access request from a third party it may contact a Master node to check for

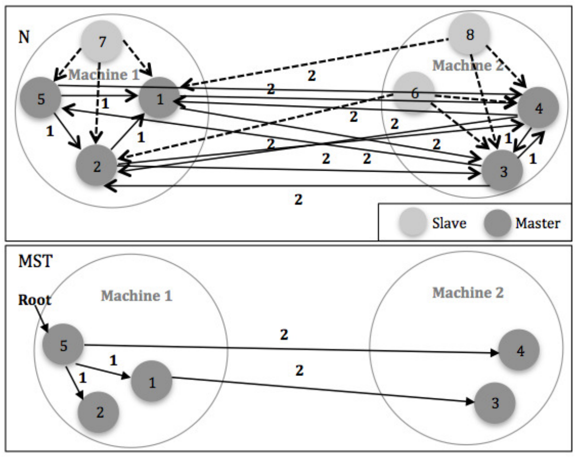


**Figure 2: The two steps of synchronization: MST construction and updates.**

updates in the SCON. Upon opening a secure connection, the Slave sends the digest of its current CS, so as to allow the Master to check if the received digest matches the current digest within its CS. If the two digests are the same, the update will not be necessary. Otherwise, the SCO Slave obtains the new CS and NMI. The redeployment aspects of the SCO are discussed in Sec. 7.1.
*Master Update:* When a Slave node contacts a Master to send an updated version of the managed content, the Slave node sends to the Master the SCO updated content or network information (denoted as CS and NMI respectively in Def. 5.2). We notice that this case presents an interesting transaction aspect: if two content files on SCO instances were updated simultaneously, multiple inconsistent versions of the same SCO may exist. To address this issue, two alternative solutions are possible. The Master node may apply one of the following two strategies (according to its configuration): (1) If one of the SCO instances was modified under an `AbsolutePermit` rule, it is used for propagation. Other conflicting copies are detached, that is they are handled as independent SCO objects, unrelated from the SCON. (2) Locking mechanisms may be enforced [15], which block multiple accesses in write mode at the same time.

## 6. PROPERTIES

We briefly discuss relevant properties of our solution, related to correctness, robustness, and consistency.

PROPERTY 6.1 (ENFORCEMENT CORRECTNESS). *Given a SCO, if a rule with* `AbsolutePermit` *or* `AbsoluteDeny` *is applicable to a request q, it is always enforced.*

This property is guaranteed by the combined effect of the enforcement algorithm and of the synchronization process. The enforcement algorithm guarantees that rules with `AbsolutePermit` and `AbsoluteDeny` effects are given highest priority. If the rule is not local to the SCO, the synchronization process guarantees that the applicable rule is passed along to the SCO.
Robustness here refers to the ability of the SCO to withstand crashes of one or multiple machines hosting SCOs.

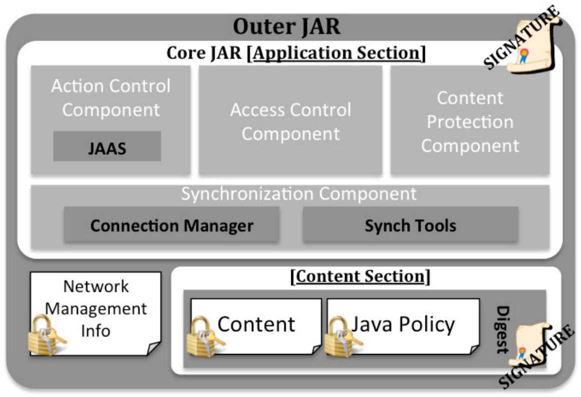


**Figure 3: SCO Architecture**

PROPERTY 6.2 (ROBUSTNESS). *The* SCON *is resilient to up to $\gamma - 1$ multiple Master nodes failure.*

This property is the result of the connectivity rule (see Definition 3). Each node, whether Master or Slave, will always be connected to at least $\gamma$ Masters, and therefore resistant to up to $\gamma - 1$ simultaneous failures. Consistency of SCON is defined as follows.

DEFINITION 6.1 (SCON CONSISTENCY). *Let SCON be the SCO Network constructed according to Definition 5.2. The set of SCON is in a consistent state if and only if: 1) each Master node carries the most up to date content and policy version; 2) upon an access request q, each Slave node enforces the highest priority policies rule that applies to q.*

PROPERTY 6.3 (CONSISTENCY). *The* SCON *satisfies the consistency definition.*

This property can be demonstrated by analyzing the SCON, and the guarantees provided by the synchronization algorithms. Precisely, the requirement (a) of Def. 6.1 is satisfied by construction of the SCON. The SCON defines a set of connected masters (see Algorithm in Appendix A), whereas the synchronization process ensures that each Master is timely informed about the latest version of the SCO's content. Condition (b) is a consequence of the enforcement correctness property (Property 1): every SCO applies the policy enforcement algorithm, which selects the highest priority rule.

## 7. ARCHITECTURE

We now present the implementation details of the SCO architecture, along with a discussion on the SCO security.

### 7.1 Implementation of SCO

Each SCO is constructed and accessed using several cryptographic keys. One key set is used for encryption and decryption of sensitive content using the Cyphertext-Policy Based Attribute scheme (CP-ABE) [2], whereas the second key pair is associated with the owners' identity, and is used for signature purposes. Further, some of the low sensitive information, such as the SCON information uses a SCO specific key shared among the SCO copies.
**Core Components**: Architecturally, the SCO includes two components and the Network Management Information (NMI),

all wrapped into an external JAR (see Figure 6). The first component, referred to as *Application Section*, is a set of unmodifiable software modules, signed and sealed to ensure integrity of the application. The Application Section is itself deployed in a nested JAR. The second component, referred to as *Content Section*, stores the protected content items, the Java policy file that records the policies implementing the security rules. This section and the NMI represent the dynamic part of the SCO and is modified upon updates. Since the Content Section stores confidential data, a signed digest of the content and of the Java policy file is added at each update, using the key of the user committing the update. The NMI, instead, stores the shared counter and the address known neighbors. NMI is also encrypted using the SCO specific key. The Application Section is further organized into four components: Authentication and Access Control Component (AAC), Action Control Component (ACC), Content Protection Component (CPC), and Synchronization Component (SYC). The AAC implements the authentication and authorization mechanisms, by exploiting the services offered by Java Authentication and Authorization Services (JAAS) [11] and Security Manager. By means of JAAS primitives, the SCO accepts X.509 certificates[1]. The ACC implements mechanisms for managing access to protected content, according to the outcome of the authentication and authorization process. This component accesses the Java policy file, which dictates what portions of the code to execute upon verification of identities and of authorization. The ACC also deals with the NMI. The CPC manages the protected content stored in the Content Section, as well as the protocols to re-deploy the new Content Section upon updates. Content encryption and access control enforcement are integrated by the use of the CP-ABE scheme [2].

**Policies Implementation:** The security policies are translated into Java policies (stored into the Java policy file) and access structures that are the input for CP-ABE encryption. CP-ABE supports the notion of attribute-based policies as a criteria for encryption and it is complementary to the Java policy. The access structures are embedded as part of the encrypted content, as by the CP-ABE construction, and therefore not separately stored. These keys are known to the owners and the certification authorities only, and always extracted by valid certificates used for authentication. Conversely, the Java policy file is stored in the Content Section. Each Java Policy specifies which party can access a specific resource (code, file) and how, while which subject is entitled to decrypt -and therefore view- the protected content is addressed by means of the CP-ABE boolean access structure entries. Content is encrypted using the *Encrypt(PK, T, f)* primitive, where $T$ is the access structure representing the attribute-based conditions (in terms of conjunctions and disjunctions) and $f$ is the file being encrypted, while $PK$ is the encryption key of the owner. We assume that CP-ABE keys are managed by the certificate authorities issuing the attribute certificates used for attributes verification.

**Copy management:** Copy management and communication protocols are managed by the SYC component.The SYC handles secure communication protocols and traces the copies, to support secure versioning. It is organized into two sub-components: the Connection Manager and the Synch Tools. The Connection Manager handles all processes for creating a connection to another SCO copy or accepting a connection request sent by another SCO copy, besides implementing the Challenge-Handshake Authentication Protocol. The Synch Tools implement all the mechanisms for synchronization between copies, whether Masters or Slaves. Both components ensure secure connections through SSL.

**Context Retrieval for policy evaluation** Every time the SCO moves in a new location (geographical or cyber), contextual information is used in order to verify if new secure policy rules apply to the specific location of consumption. To guarantee this high level of adaptivity, the system must be able to retrieve, at any single access attempt, all the information needed to define the current context, both in term of time and physical and/or cyber space. Geographical information is generated by the GeoIP service [8] that uses as input only the IP address of the machine hosting the SCO. Temporal information (i.e. 10:11 AM Tuesday, March 06, 2012, Eastern Standard Time (EST) -05.00 UTC) is generated by using the WorldTimeServer service [8] that uses as input the current geographical location of the SCO.

Depending on where and how the SCO is accessed, obtaining this information may require different approaches. For example, data stored on cloud service providers can be accessed in two different ways, as simple documents stored in a virtual drive (for example by using Amazon Cloud Drive, Dropbox, or Windows Live Sky Drive), or as application data accessible by applications built and hosted on the SP (for example data stored in Amazon S3 used by application hosted in Amazon EC2). These two ways of data consumption are substantially different. In the case of virtual drives, a copy of the SCO is downloaded from the virtual space on the user's machine. The context information is retrieved directly by the SCO that wraps the content, running on the local machine, by exploiting the Global Services [8].

In case an application accesses data stored by the provider (e.g. an application built on Amazon EC2 accesses data stored on Amazon S3), the context information must be provided by the provider hosting the data and the application. The SP generates and makes available a context file, using the provider's APIs. The context file includes the current time and geographical location of consumption, and an identifier of the `Cyber Location` (e.g., Windows Azure Domain) where the information was captured. If the SCO is located on a public domain or on a public network, location and temporal information are retrieved by the SCO through the context information file. We provide a concrete example in the context of Amazon Web Services APIs in the code snippet below.

```
AmazonS3 s3 = new AmazonS3Client(new PropertiesCre-
dentials(S3Sample.class.getResourceAsStream
("AwsCredentials.properties")));
s3.createBucket(bucketName);
s3.putObject(new PutObjectRequest(bucketName, key, SCO));
S3Object object = s3.getObject(new
GetObjectRequest(bucketName, key));
AmazonEC2 ec2 = new AmazonEC2Client(credentials);
AllocateAddressResult aar = ec2.allocateAddress();
aar.getPublicIP();
```

The example shows how to store the SCO in the AmazonS3 storage service, and retrieve the content from the SCO using Amazon EC2. The AmazonEC2 instance can retrieve the information needed to create the current context of data consumption and create the context file. The

[1]The SCO may be configured to interpret any other authentication token - such as XACML, SAML, X.509.

actual location is defined by the public IP address (retrieved by the `allocateAddress`() function), where the AmazonEC2 instance is actually running. The same approach could be adopted by other Cloud providers, like Windows Azure or Rackspace.
If it is not possible to retrieve all the information needed to define the current context, the context will be defined as indeterminate. This will render the SCO not accessible.

**Adaptiveness:** If the policy enforcement mechanism determines the lack of a rule applicable to the current context, the SCO before enforcing the `FinalPermit` rules forwards a policy request to retrieve high-priority rules applicable to the current context, if any. We assume that a trusted authority (CA) managed by the SP is responsible collecting contextual policies that implement local laws or regulations. Thus, the SP updates the SCO policy rules, whenever the SCO is in an unknown and unexpected context. The execution flow is as follows. The SCO sends a request to the CA, through an encrypted channel requesting for high priority rules. The CA after having verified the request's authenticity, determines whether any policy rules applicable to the context specified by the SCO exist. If some policy rules are found, the CA executes the policies translation and redefines a new Content Section with a new encrypted content and a Java policy (without modifying the location information). The CA calculates the new digest and signs it with its private key, to ensure integrity. Once the new content section is ready, the CA sends it to the SCO that replaces its current content section and activates the synchronization process, to disseminate the newly acquired rules. If no policy rule referred to the specified context is found by the CA in its policy rules set, the CA returns no output, and the `FinalPermit` rule is applied.

## 7.2 Security Features

We discuss possible attacks to our SCO-based framework. We assume that a content originator does not release sensitive information to unauthorized parties, and secret keys used for signature generation and content encryption are kept secret. Conversely, an attacker may try to access information directly from the SCO that is disseminated in the network or try to disassemble the SCO to gain access to the protected content. We only consider attacks to the SCOs. Attacks attempting to exploit the communication between SCO are prevented by means of encrypted authenticated sessions and use of challenge response protocols.

**Disassembling and Reverse Engineering:** By disassembling the SCO outer JAR, an attacker will be able to get the internal elements of a SCO. The disassembled Core JAR will show the attacker all .class files. The class files once extracted can easily be decompiled into the original source code using Java decompiler tools. To mitigate this risk, we adopt Java-specific obfuscation techniques, which leverage polymorphism and exception mechanism to drastically reduce the precision of points-to-point analysis of the programs [17]. Regarding the Application Section, the attacker cannot change the content after disassembling the Section, nor can he reassemble it with modified classes, files, or packages, because of the protection offered by the JAR seal and signature techniques. As a result of the disassembling, no sensitive information can be obtained from reverse engineering since this information is not within the code but is retrieved by the originator's certificate, at the SCO execution time. The attributes and keys used for decrypting the content under the CP-ABE scheme are never stored, but retrieved when needed. Instead, in the Content Section the Java Policy file is encrypted using the RSA encryption algorithm with the owner's secret key of 2048 bits, and therefore resistant against brute force attacks. Even if the attacker obtains the decryption key used to read the Java Policy file, it will never be able to alter the content, if the CP-ABE key used for content encryption remains secret. CP-ABE is by construction resistant to collusion and chosen plaintext attacks. The only possibly useful information stored in the Application Section is the key encrypting the NMI. Since we rely on advanced obfuscation techniques and store the key in randomized fashion, it is hard to retrieve it. Even in case the key is obtained, the attacker only learns the connections of the SCO and the size of the SCON.

**Security Policy modification:** A malicious user might attempt to tamper with the SCO in order to change the security policy and gain access to protected content. Our architecture leverages the properties of the CP-ABE encryption schema rendering this attack not feasible. In the CP-ABE the access permission policy is directly and implicitly contained within the protected content. There is no actual policy file that an attacker can modify to change the access decision. Even if the attacker decrypts and reads the Java Policy file, the file does not provide information about the actual authorization and privileges in terms of content access. A user will be able to decrypt the protected content only if her attributes satisfy the attribute-based policy used for content encryption. A malicious user could overcome this control by trying to build a set of attributes to satisfy the access structure for the encrypted content. Such attack would fail during the authentication phase, in that the attacker would need to provide a valid certificate with those attributes, for which he would need the CP-ABE master key stored at the authority.

**Unauthorized copy of protected content:** A malicious user with right to access the encrypted protected content can generate another possible attack. Once the user is granted access to the protected content, the user attempts to copy it, and save it in a decrypted form in order to allow unauthorized users to read the content. We mitigate this attack by decrypting the content always on the fly and transferring it directly to the content rendering application, therefore preventing the attacker from intercepting temporary files. The content rendering application (such as the Java application window) is then in charge of securely rendering the content.

**Bypassing Authentication:** An attacker could try to bypass the authentication process and try to directly access the protected content. A way for the attacker to do so is to modify the code within the Application Section. This approach would not work since both the digital signature and the sealing guarantee the JAR file integrity. This ensures that the code within classes and packages cannot be changed. Alternatively, the attacker may try to change the behavior of our application. The attacker can try to exploit malicious external code, able to communicate with our internal code, thus altering the proper functioning of the application. For example, an attacker could insert malicious code and bypass the authentication step so to jump directly to a different point of the execution. To solve this problem, we exploit program tracing techniques [14], by inserting in the application code a set of checkpoints. Each checkpoint controls the current

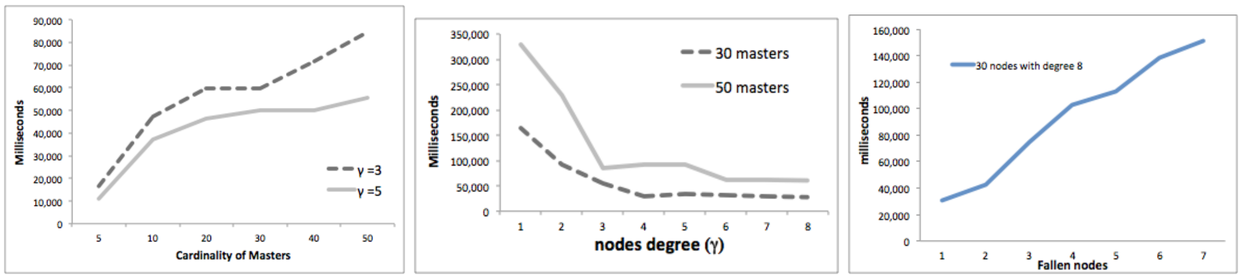


**Figure 4: (a) Synchronization for networks of increasing size and $\gamma$, (b) Synchronization for networks of same size and different $\gamma$ (c) Robustness of SCON**

value of a set of global oblivious hash variables whose value is continuously changed during the application execution using the oblivious hash technique. During the execution each checkpoint controls the value contained in these hash variables. If the value of a hash variable is different from what it would be obtained from a proper code execution, the checkpoint fails and the application is forced to end. This technique allows us to force a particular sequence of instruction execution within methods, and a certain sequence of method calls within a class or between classes. An attacker cannot eliminate the checkpoints used for program tracing because these checkpoints are directly inserted in the signed code. Any such change would be detected by the verification of the code signature.

**Java Corruption:** Undoubtedly, the most challenging attack to our architecture is corruption through compromised Java environments. With a corrupted Java Running Environment (JRE) the attacker may overcome all the Java-based security controls (signature, sealing, authentication). Hence, the attacker may authenticate, read and modify the JAR. However, unless the attacker has a valid cryptographic key, the attacker cannot access the content or the policy. Henceforth, while the synchronization mechanism may fail, the confidentiality and integrity of the content are guaranteed, since the keys are never stored in the application. To quickly detect a corrupted JRE we tie together various control mechanisms, and continually do crosschecks and stop any execution as soon as any crosscheck fails. These checks include integrity checks on the system environment, validating manifest of sensitive directories, validating the runtime JAR, and checking that the extended Policy Manager is operating. These system integrity checks are repeated more times during the execution of the application, before critical points. Examples of critical points are: permission check, policy enforcement, protected content encryption/decryption, and creation of secure connections between SCOs. A stronger solution consists of deploying a Java Security Extension for the JRE, that provides assurance of correct system operation and integrity even in presence of successful attacks on the underlying operating system. The primary objective of this security extension is improving the level of the operational integrity of the Java application. This approach, originally proposed by Wheeler [24] consists of extensions to the Java 2 security system. The JRE extended components can be directly contained in our SCO, and ready to be loaded at each execution. The SCO will run only if all the extended components are properly loaded, to ensure proper functioning of all the Java security services. More details are reported in the Appendix.

## 8. EXPERIMENTAL EVALUATION

We tested our framework on the Emulab testbed. The test environment consists of several OpenSSL-enabled servers: one head node which is the certificate authority, and several computing nodes. Each of the servers is installed with the Eucalyptus middleware [6]. We used Linux-based servers running Fedora5 OS. Each server has a a 64-bit Intel Quad Core Xeon E5530 processor, 4 GB RAM, and a 500 GB Hard Drive, and is equipped to run the JRE 6. We conducted several tests to evaluate the performance of the two key operations supported by our framework: (1) node synchronization; (2) Content Section update.

Concerning the synchronization algorithms, we tested the overall time required to complete synchronization processes. We considered networks of Master nodes only. As shown in Figure 4(a), we varied the number of Master nodes from 5 to 50. We run two tests, with SCON of $\gamma$ equal to 3 and 5, respectively. With a higher connectivity ($\gamma = 5$) the synchronization is consistently faster, for every SCON size. We notice that even if the synchronization times for 50 Master nodes appear relatively high, this synchronization may help to quickly update several hundreds nodes. For example, we estimated that 50 Master nodes with $\gamma = 3$ may tolerate about 400 Slave nodes. To obtain such estimate, in a separate test we measured the time for synchronizing a Slave node by obtaining the Content Section from a Master node. The overall synchronization time consists of: (1) the connection and authentication time; and (2) the Content Section update time. Using a file of average size 20Mb, and a connection with a speed of 100Mb/sec the overall synchronization time is 240 ms. This implies that even in absence of parallelization, a Slave node may have to wait up to 240ms to be updated. Within 2 seconds, each Master node can serve 8 Slave nodes for a total of 400 Slave nodes.

Next, to better investigate how to reduce the synchronization time, we tested the effect of the Master nodes degree, $\gamma$. We tested the synchronization times for values of $\gamma$ ranging from 1 to 10. We repeated these tests for different network sizes. As reported in Figure 4 (b), the overall trend is the same: the times for synchronization drop drastically after a certain value of $\gamma$. For values higher than such optimal value, the times remain seemingly constant. The optimal value appears to be approximately at $\gamma = \frac{|M|}{10}$, where $|M|$ is the Masters' cardinality. Based on these two experiments, we conclude that efficient synchronizations can be

achieved with acceptable tolerance without requiring highly connected networks. Additionally, we tested the synchronization time in case of failed nodes. The results are reported in Figure 4 (c). The test considers a SCON with 30 Master nodes, and $\gamma = 8$. Each test considers an increasing number of failed nodes from 1 to all but one (7). Clearly, the times increase significantly, due to the time spent by a node waiting for the response. Yet, the synchronization can be completed within 150 sec.
To evaluate content update times, we tracked the time required for constructing a new Content Section (see Sect. 7). For a file of size equal to 1.2 MB, the overall update time is equal to 850ms. Most of the update time is due to the cryptographic operations. Encryption time under CP-ABE is proportional to the complexity of the policy. For this experiment, we considered an access tree with 20 leaves, resulting in a content encryption time of 640ms. Our experiments with files of different size reflect the complexity of the original CP-ABE protocol [2]. The other operations required to complete the update, i.e. content replacement, digest creation signature are in the order of few milliseconds, with the exception of the digest creation time, which takes 160ms.
Finally, we tested the overall size of the SCO. The Application Section is fixed, and it has a size of 350 KB. The Content Section includes one Java Policy file, which is about 5 KB and the content. This Policy file size changes slightly depending upon the number of grant and deny in the Java policy. The NMI is also small, $< 20KB$ and its size depends on $\gamma$ (# of known Master nodes). The size of the protected content file is to be added also. With the encryption, the NMI and Content Section files increase of a few bytes.

## 9. RELATED WORK

Access control [22, 25] and data management outsourcing techniques targeting the cloud have been recently proposed [1, 13, 18]. Further, cloud-specific cryptographic-based approaches for ensuring remote data integrity have been developed [23]. Ateniese et al. [1] proposed the Provable Data Possession model for ensuring possession of files on untrusted storages, as well as a publicly verifiable version, which allows anyone to challenge the server for data possession [5]. These schemes are effective with static data, but are not suitable for dynamic scenarios where data may change and be used within and outside the cloud domain. Subsequently, Wang and et al. [23] proposed a data outsourcing protocol specific to the cloud. Wang's work is focused on auditing of stored data from trusted parties, and does not deal with access policies, nor does it provide aspects of data management.
The notion of SCO is corroborated by previous projects [10, 16] and our own results [20]. Our approaches are closely related to self-defending objects (SDO) [10] and self-protecting objects. SDOs are Java-based solutions for persistent protection to certain objects. SPOs are software components hosted in federated databases. The objective of SDOs is to ensure that all the policies related to any given object are enforced irrespective of which distributed database the object has been migrated to. SDOs depend on a Trusted Common Core for authentication and authorization, and therefore are not applicable in distributed systems. In fact, the realm of application of both SDOs and SPOs is restricted to centrally controlled systems that support trusted cores and federations. As a result, adaptiveness, synchronization and versioning issues are not considered. Related to the idea of self-protecting data is also the work on sticky polices [16] that focuses on portability of disclosure policies by means of declarative policies tightly coupled to sensitive data by means of cryptographic algorithms. However, these policies are designed for federated organizations, and therefore lack adaptiveness to the domain of application.
Furthermore, enterprise Java Beans (EJB), Microsoft's Component Object Model (COM), and the Common Object Request Broker Architecture (CORBA) are all object-oriented frameworks for building three-tiered applications. While they differ in design and exact functionality, all are based on distributed objects, location transparency, declarative transaction processing, and integrated security. They share SCO's goals of offering object-oriented technologies to facilitate distributed transactions and programming in distributed systems, but target trusted enterprises. Similarly, Orleans [3] offers a software framework focusing on synchronization of distributed programming objects and recovery on the cloud. Despite these similarities it fundamentally differs from our approach in that it focuses on programming environments and does not involve any protection of the objects, nor it is concerned about controlling accesses or usage.
Distributed synchronization protocols have also been widely investigated in the past [15]. However, these protocols are based on the use of software agents external to the objects (e.g. a lock manager), whereas in our case the SCO encapsulates all the software needed for an update and only the availability of a Java environment is required. Also those protocols do not include any protection mechanism for the object, nor do they support policies based on contexts or other properties of the access requestor. Yet, implementing a locking mechanism for our SCO-based architecture is part of our future work.
Other well-known cryptographic primitives have been developed not only to protect remote data from attacks to confidentiality and integrity, but also to support condition evaluation on encrypted data [12,19]. These approaches may be suitable for secure content rendering from the SCO, but are complementary to our architecture, in that they address confidentiality, rather than distributed management.

## 10. CONCLUSION

We presented an approach for secure and distributed data management. The idea behind our solution is to protect the data by means of wrappers applied to the targeted content file(s) which protect the content as it travels across domains by locally enforcing security policies. The policies may be dynamic, and adapt to the location and other contextual dimensions. We plan to further develop the policy language and support more articulated contexts. We will also investigate how to support distributed locking systems.

## 11. ACKNOWLEDGEMENT

Portion of the work from Dr. Squicciarini was funded by the National Science Foundation under Award #1250319 "Collaborative Research: Brokerage Services for the Next Generation Cloud". Portions of the work from Dr. Bertino was funded by National Science Foundation under Award # 1111512 "Privacy-Enhanced Secure Data Provenance".

## 12. REFERENCES

[1] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *ACM conference on Computer and Communications Security*, pages 598–609, 2007.

[2] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *2007 IEEE Symposium on Security and Privacy*, SP '07, pages 321–334, Washington, DC, USA, 2007.

[3] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 16:1–16:14, New York, NY, USA, 2011. ACM.

[4] R. Crozer. Mr rental trials azure as telstra cloud redundancy. http://www.itnews.com.au/News/306871,mr-rental-trials-azure-as-telstra-cloud-redundancy.aspx.

[5] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. MR-PDP: Multiple-replica provable data possession. In *The 28th International Conference on Distributed Computing Systems*, pages 411 –420, June 2008.

[6] Eucalyptus Systems. http://www.eucalyptus.com/.

[7] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5:66–77, January 1983.

[8] GeoIP Web Service. http://www.webservicex.net/geoipservice.asmx.

[9] Gordon's Notes. Reliability and the cloud - redundancy required. http://notes.kateva.org/2011/05/reliability-and-cloud-redundancy.html.

[10] J. W. Holford, W. J. Caelli, and A. W. Rhodes. Using self-defending objects to develop security aware applications in java. In *27th Australasian conference on Computer science - Volume 26*, ACSC '04, pages 341–349, Darlinghurst, Australia, Australia, 2004.

[11] Java Authentication and Authorization Services. http://java.sun.com/products/archive/jaas/.

[12] J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *27th annual international conference on Advances in cryptology*, pages 146–162, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative internet backup scheme. In *USENIX Annual Technical Conference*, pages 29–41, 2003.

[14] D. Liu and S. Xu. MuTT: A Multi-Threaded Tracer for Java Programs. In *8th IEEE/ACIS International Conference on Computer and Information Science*, pages 949–954, 2009.

[15] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems, 2/E.* Prentice Hall, 1999.

[16] H. C. Pöhls. Verifiable and revocable expression of consent to processing of aggregated personal data. In *International Conference on Information and Communications Security (ICICS)*, pages 279–293, 2008.

[17] Y. Sakabe, M. Soshi, and A. Miyaji. Java obfuscation with a theoretical basis for building secure mobile agents. In A. Lioy and D. Mazzocchi, editors, *Communications and Multimedia Security. Advanced Techniques for Network and Data Protection*, volume 2828 of *Lecture Notes in Computer Science*, pages 89–103. 2003.

[18] T. J. E. Schwarz and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *IEEE International Conference on Distributed Systems*, page 12, 2006.

[19] N. Smart and F. Vercauteren. Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes. In *Public Key Cryptography â PKC 2010*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. 2010.

[20] A. Squicciarini, G. Petracca, and E. Bertino. Adaptive data management for self-protecting objects in cloud computing systems. In *The International Conference on Network and Service Management (CNSM)*, 2012.

[21] Sun. Lesson: Packaging programs in jar files. http://java.sun.com/docs/books/tutorial/deployment/jar/.

[22] Q. Wang and H. Jin. Data leakage mitigation for discretionary access control in collaboration clouds. In *16th ACM symposium on Access control models and technologies*, SACMAT '11, pages 103–112, New York, NY, USA, 2011. ACM.

[23] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *ESORICS*, pages 355–370, 2009.

[24] D. M. Wheeler, A. Conyers, J. Luo, and A. Xiong. Java security extensions for a java server in a hostile environment. In *ACSAC*, pages 64–73, 2001.

[25] S. Yu, C. Wang, K. Ren, and W. Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *Proceedings IEEE INFOCOM*, pages 1–9, 2010.

## APPENDIX

### A: Algorithms

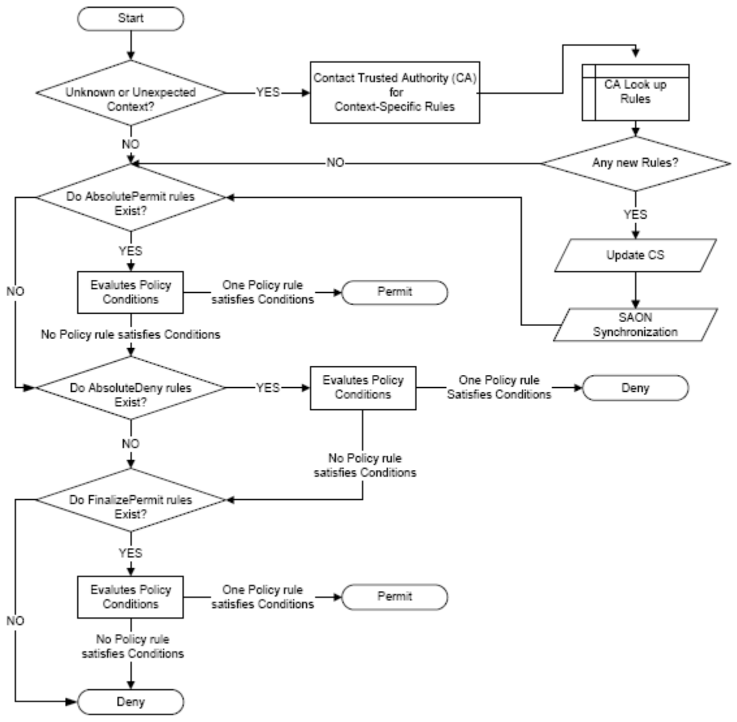
In this appendix we provide the main algorithms presented in the paper. First, we present the operational flow of the policy enforcement algorithm. The algorithm is displayed in Figure 5. Algorithm 1 reports the node creation process.

**Algorithm 1:** SCO Network node addition

```
Require: $\omega$ Ratio
  $\alpha \geq 5$ nodes ratio
  $\gamma \geq 2$ Redundancy Threshold
  $SelectedMs$ Set of Masters to be contacted during synchronization
  $NMI$ Network Management Information
  $selectRandomM(SelectedMs)$ Selects a new Master from NMI that it is not in SelectMs set
  Begin
  t = NMI.getNumCopies()
  SelectedMs = {}
  if ($t = k(\omega\alpha) + \omega\alpha$) then
    k++
  end if
  if ($k(\omega\alpha) \leq t \leq k(\omega\alpha) + \alpha$) then
    %Master
    if ($t \leq 2$) then
      for (count := t; count > 1; count –) do
        SelectedMs := NMI.selectRandomM(SelectedM)
      end for
    else
      if $\gamma > t - 1$ then
        count:= $t - 1$
      else
        count:= $\gamma$
      end if
      while count$\neq$ 0 do
        SelectedMs$\leq$ NMI.selectRandomM(SelectedM)
        count$--$
      end while
    end if
    PORT := n.getFreePort()
    NMI.addNewMaster(IP,PORT)
    NMI.incNumCopies() startSynch(NMI)
  else
    %Slave
    count := $\gamma$
    while (count $\neq$ 0) do
      SelectedMs $\leq$ NMI.selectRandomM(SelectedM)
      count$--$
    end while
    NMI.incNumCopies()
    startSynch(NMI)
  end if
  End
```

### B: Security Extensions

The Java-based security extensions proposed in Section 7.2 provide assurance of correct system operation and integrity even in presence of successful attacks on the underlying operating system. The primary objective of this security extension is improving the level of the operational integrity of the Java application. The idea, originally proposed by Wheeler [24] relies on extensions to the Java 2 security system, including the Security Manager, the Class Loader and the Policy Manager. The JRE extended components can be directly contained in our SCO, and ready to be loaded at each execution. The extensions create a strong dependency between these components. For example, the Class Loader will not operate without the extended Security Manager. The extended Security Manager performs system integrity checks during startup and during normal permission checks. These checks include integrity checks on the system environment, validating manifest of sensitive directories, validating the run-time Jar (rt.Jar), and checking that the extended Policy Manager is operating.



Start
Unknown or Unexpected Context?
YES
Contact Trusted Authority (CA) for Context-Specific Rules
CA Look up Rules
NO
Any new Rules?
NO
YES
Update CS
SAON Synchronization
Do AbsolutePermit rules Exist?
YES
Evalutes Policy Conditions
One Policy rule satisfies Conditions
Permit
No Policy rule satisfies Conditions
NO
Do AbsoluteDeny rules Exist?
YES
Evalutes Policy Conditions
One Policy rule Satisfies Conditions
Deny
No Policy rule satisfies Conditions
NO
Do FinalizePermit rules Exist?
YES
Evalutes Policy Conditions
One Policy rule satisfies Conditions
Permit
No Policy rule satisfies Conditions
NO
Deny

**Figure 5: Policy Evaluation and Enforcement**

These system integrity checks are repeated during the execution of the application, before critical points. Examples of critical points of the execution are: permission check, policy enforcement, protected content encryption/decryption, and creation of secure connections between SPCs and Synchronization Managers. The extended policy manager checks that the system's configuration (.policy and .conf files) are not modified, by verifying theirs digital signature. Before the Class Loader loads any classes, its loads the extended Security Manager, which checks the system and validates that the extended Policy Manager is loaded. If any of the integrity checks fails, each extended component notifies the other components of the failure, and attempts to shut down the system operation. Furthermore, to prevent any sensitive operation from being performed for newly loaded classes, the extended Policy Manager denies access to all services and resources by returning a null permission in response to any query.