Adaptive Data Management for Self-Protecting Objects in Cloud Computing Systems

Anna Cinzia Squicciarini[1], Giuseppe Petracca[1], Elisa Bertino[2]
1 College of Information Sciences and Technology, Pennsylvania State University
2 Computer Science Department, Purdue University

***Abstract*—While Cloud data services are a growing successful business and computing paradigm, data privacy and security are major concerns. One critical problem is to ensure that data owners' policies are honored, regardless of where the data is physically stored and how often it is accessed, and modified. This scenario calls for an important requirement to be satisfied. Data should be managed in accordance to owners' preferences, Cloud providers service agreements, and the local regulations that may apply. In this work we propose innovative policy enforcement techniques for adaptive sharing of users' outsourced data. We introduce the notion of autonomous security-aware objects, that by means of object-oriented programming techniques, encapsulate sensitive resources and assure their protection. Such protection is afforded through the provision of adaptive security policies of various granularity. Our evaluation demonstrates that our approach is effective and efficient.**

I. Introduction

Cloud computing services enable individuals and organizations to outsource data management with ease and at low cost. However, while Cloud data services are a growing successful business and computing paradigm, data privacy and security are major concerns. Data stored in the Cloud often encodes sensitive information and should be protected as mandated by organizational policies and legal regulations. A commonly adopted approach is to encrypt the data when uploading them to the Cloud. Encryption alone however is not sufficient as often organizations have to access, modify and selectively share data with other parties [18], [19]. Policies must be associated with the data to specify which party can access the data for performing which actions. Such policies can be quite complex as they typically encode different access constraints. Furthermore, a Cloud secure data management system should be able to demonstrate ongoing conformity with their customer's business compliance and regulatory requirements (depending on industry and jurisdiction). Maintaining separate compliance efforts for different regulations and standards in an automated fashion is knowingly a major challenge in the Cloud domain. This is especially challenging since users' data may be stored in multiple locations, based on Cloud Service Providers (CSPs) internal scheduling and management processes. Therefore, access restrictions for sensitive data have to adapt not only to changes by the content originators (or owners), but also to privacy and auditing regulations that are in place in the location where data is stored and managed. Furthermore, depending on the location, different architectures and standards may affect data representation and management, resulting in compatibility issues.

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 security-aware objects* (SAOs), that by means of object-oriented programming techniques, encapsulate sensitive resources and assure their protection. Such protection is afforded through the provision of adaptive security policies of various granularity. The resources protected by SAOs may be files of different kind, including images, text, and media content. The security policies can include constructs to specify access control, authentication and usage controls. Furthermore, they are enforced according to contextual criteria to ensure compliance with location-specific regulations and service level agreements. Through careful design and deployment, SAOs provide strong security guarantees, and adapt to local compliance requirements. As part of our solution, we show how SAOs can either use locally pre-loaded policies or securely accept new policies from trusted authorities.

We have implemented a running prototype of the proposed approach using Java™-based technologies [1], [16]. We selected Java over other object-oriented environments because of its unique benefits, such as code portability, ease-of-use, richness, and wide adoption. Yet, our design is language-independent, and can be prototyped using other languages that support an object-oriented encoding, such as C++, or .Net. Java also offers important security features that are leveraged with advanced cryptographic protocols and cross-checking techniques to guarantee acceptable trustworthiness of the SAOs. We have extensively tested our prototype, and used the Amazon Web Services (AWS) APIs to demonstrate how our solution can be integrated with actual CSPs.

The rest of the paper is organized as follows. Next section provides an overview of the notion of SAO. Section III discusses the security policies and their enforcement. Section IV presents the deployment of our architecture and discusses the context retrieval process. Section V discusses implementation and security of our solution. Section VI reports experimental results. Section VII discusses related work and Section VIII concludes the paper.

II. Overview of the SAO-based solution

A. Overview of Secure-Aware Objects

Any object-oriented approach designed to achieve adaptive data protection has to address the following requirements: protection of encapsulated data through enforcement of security

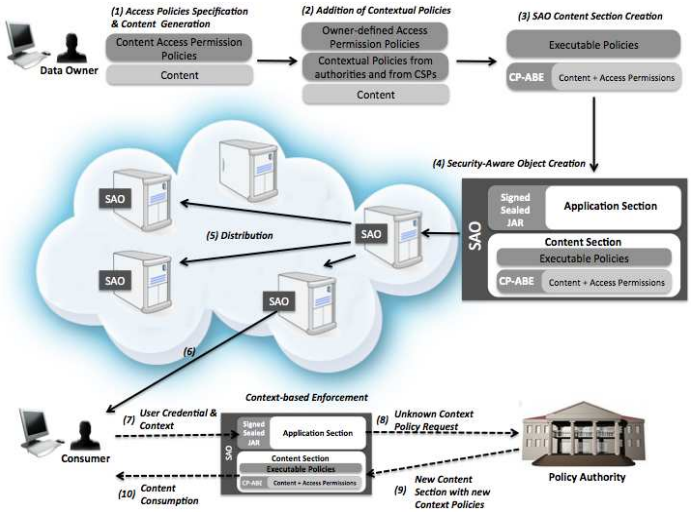

Fig. 1. Overall Approach

policies associated with the data, interoperability, portability, and object security. Our SAO-based approach addresses the first requirement by provisioning owner-specified security policies that are tightly bound with the content, and executable within the SAO. To ensure interoperability, our SAOs are self-contained, and do not require any dedicated software to execute, other than the Java running environment. Precisely, each SAO includes five key components: (1) authentication and authorization tools; (2) self-enforcement policy engine; (3) security policies in executable form; (4) secure connections manager; and (5) protected file(s);

Self-containment ensures portability, in that SAOs may be moved and replicated without the need of installing dedicated software. We employ nested Java Archives (JARs) [16] to achieve both the portability of the protected content and the portability of the modules required to access and use the content as intended by its originator.

In order to guarantee adaptivity to the context from where the content is accessed, the policies embedded in the SAO 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 [5] and Oblivious Hashing for program tracing [10], combined with extended and advanced object-oriented coding techniques. We leverage built-in Java constructs, like Java policies and Java Authentication and Authorization Service, to implement the object's content protection mechanisms. As elaborated in Section IV-A and V, each component is designed to be as resilient as possible to the numerous threats against the SAO itself and its protected resources.

B. Interaction Flow

Our SAO-centered architecture deals with four different classes of stakeholders, that is, content owners, government authorities, cloud service providers (CSPs), and content consumers. These entities are in charge of authoring policies for the protected content, and/or accessing the content upon need. In particular, government authorities issue regulations concerning content sharing and protection, and require that these regulations be enforced by organizations managing data; whereas CSPs may enforce rules reflecting their terms of service or to facilitate data processing. Content owners may have their own personal policies concerning the access and use of their own content. Finally, content consumers access content, their accesses must comply with all the access restrictions imposed on the content by the other stakeholders.

In what follows, we assume that government authorities and service providers are represented by a trusted third-party, referred to as *Policy Authority* (PA), which is in charge of storing high-level regulations and service level agreements, and translate them in policy rules expressed in a machine interpretable form. The generated policy rules can thus be automatically enforced by a computer system.

Figure 1 shows the interplay among the four stakeholders during generation and consumption of SAO. In detail, a SAO is generated by a content originator subscribed to a CSP. The generation of a SAO includes three major phases, reported on top in the figure. The first phase is the specification by the content originator of policy rules, for controlling the sensitive content (e.g. text, video or image file) to be stored at the CSP (step 1). The second phase is the specification by the policy authorities of policy rules whose enforcement assures compliance with security and data disclosure regulations (e.g. auditing requirements), as well as with SLAs specified by cloud service providers (step 2). The third phase is the compilation of all these policy rules into executable policies. The executable policies are then encrypted and sealed with the content using CP-ABE primitives (step 3). After these three phases, the generation of the SAO is completed.

Once created, the content stored in each SAO may be accessed according to its security policy. SAO consumers may include authorized users, CSPs for processing and analysis, and auditors/authorities to meet compliance requirements. Each time access to the SAO content is attempted, its policy rules are evaluated for applicability, according to the requested subject credentials, and contextual features (step 7).

Notice that as SAOs are moved across locations, the SAOs must always comply with the regulations in place at the SAO consumption location. For example, if customer data is maintained within SAOs, and the SAOs are stored in a EU server, data disclosure laws apply, whereas different auditing rules apply in the USA[1]. To address this requirement, the policy protecting the content stored within the SAO is organized in rules of varying priority, and each rule may be context-specific. As the SAO changes location, high-priority rules may be added to the SAO, using a third-party invocation protocol if the rules are not stored within the SAO already. Specifically, each policy rules set generated by a local authority is stored in a repository managed by the local PA, as shown in steps 8 and 9 of Figure 1. The PA guarantees that each policy rule contained in the set is within the scope of the authority jurisdiction, and that it is applicable only for SAO stored within the authority's domain.

[1]We focus on auditing requirements, and do not address the case where data is to be stored or encrypted according to a specific format.

III. SAO POLICIES

In this section, we first introduce an abstract representation of the security policies supported by our system. Next, we discuss how these are translated and stored within the SAO.

A. Policy Specification

We define a security policy as a collection of declarative 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 entity (e.g. user, service provider, government authority). It may either target a single content item f (e.g. file) stored within the SAO, or if the SAO stores multiple content elements (i.e. files), it may refer to all of them.

A rule r is a tuple composed by the following elements: $<$ PolicyL:Effect, Act, Subj, Obj, Location, Time $>$. Besides the PolicyL : Effect, and the Obj, the remaining components are defined in terms of boolean conditions against a set of pre-defined variables. Boolean conditions take the form $(\mathtt{a}\ \mathtt{op}\ \mathtt{a}')$, where a is the attribute name, $\mathtt{a}'$ the attribute value, and op is a basic atomic operator, such as $>, <, =, \geq$, and $\leq$. We assume that boolean conditions are atomic. We briefly discuss each component in what follows:

- PolicyL denotes the policy level and can take one of the following values: Auth, CSP, or Owner; whereas Effect is a boolean variable that represents an access $Deny$ (0) or a $Grant$ (1). The PolicyL is assigned according to the policy author, and it dictates rules' hierarchical order. Auth refers to rules specified by government authorities, and take priority over rules defined by other parties. CSP refers to rules that can be defined by CSPs for managing the user's content. This is a secondary priority level. Owner denotes the lowest priority level and it is used by data owners to control access of their own content.
- Act denotes the admitted action, and is therefore content-specific. Act is instantiated if the rule effect is set to 1.
- Subj is the targeted entity. A subject can be identified by means of a combination of one or more joint (disjoint) boolean conditions against the attributes of the credential used for authentication purposes at the SAO. For example, if X.509 certificates are used, any condition against any of the attributes included in the credential to qualify the subject (e.g. Country, Name, and Age) can be used. In addition, the policy author can constrain the class of entities the rule applies to, by specifying a boolean condition against a special Type attribute, if not supported directly by the credential. This type refers to the entity performing the access, which could be one of the three stakeholders: Auth, CSP, or User. In particular, Auth refers to (regulatory/legal) authorities, CSP refers to cloud service providers, and User refers to end consumers.
- Obj refers to the target of the rule. It can refer to an individual file within the SAO, or the whole object, in which case Obj assumes the default value SAO.
- Location refers to conditions related to the location of the data consumption. Location-based conditions can refer to both geographical and cyber locations. They are both expressed at various levels of granularity, from a general country or state, to the specific machine where the object can be executed. In the current version of our policy language, locations can be specified using boolean conditions against predefined attributes. Examples include Country, State, City, IPaddress. To ensure correct evaluation of location information, we request that if City $\neq$ NULL $\vee$ State $\neq$ NULL, then Country $\neq$ NULL.
- Time is used to represent any possible time constraints. Temporal conditions are represented by means of finite intervals [3] $[\mathtt{t_b}, \mathtt{t_e}]$, where $\mathtt{t_b}$ and $\mathtt{t_e}$ denote the lower and upper bound of the interval, respectively. Open ended intervals are also supported (e.g. $\mathtt{t_b} > 12/12/2014$).

Example 1: We now provide some simple examples of policy rules. We use the wild card "NONE" to denote an element is empty, and the dot notation to indicate rule's components. The following policy rule, specified by the authority, allows access to the SAO data by a USA federal agent:

$\mathtt{r_1} =<$ Auth : 1, Read, (Subj.Type = Auth, Subj.Name = USAFA), ANY, (Location.Country = USA), NONE $>$

An example of a rule authored by the CSP is specified below:

$\mathtt{r_2} =<$ CSP : 1, Read, (Subj.Type = CSP, Subj.Name = AmazonAdm), ANY, (Location.Country = USA, Location.Cyber = Amazon), NONE $>$

Finally, the rule below is an example of a negative rule against a user, specified by a content owner.

$\mathtt{r_3} =<$ Owner : 0, NONE, (Subj.Type = User, Subj.Name = Sue Yoy), SAO, NONE, $[12/10/2012, 12/01/2013] >$.

B. Policy Rules Selection and Translation

When a new SAO is created or a SAO is moved to a new location, the policy composition and translation process is activated. This process is executed so as to ensure that only relevant and applicable rules are included in the SAO. Simply storing all the rules in the SAO may be impractical and not scalable. The policy translation process is preceded by a selection of the applicable rules, followed by a static ordering of applicability.

1) Policy rules selection: This first step identifies a set of applicable rules. Applicable rules are identified using current contextual information, without consideration of the subject requirements, as if subject information were not available at this point of time. Precisely, a policy rule r =<PolicyL : Effect, Act, Subj, Obj, Location, Time> is applicable to the current context c =< Country, State, City, IP, Timestamp>, if and only if: (a) the rule location conditions match the contextual information, that is c.Country = r.Country or c.State = r.State or c.City = r.City; (b) r refers to a file f witch matches or is included in the Obj component of the rule, i.e. Obj = f or Obj = SAO; (c) temporal restrictions are not obsolete, that is Timestamp $<$ r.Time.$\mathtt{t_e}$.

Next, the applicable rules with highest priority are selected. To ensure correctness of execution, the rules are organized into classes according to the PolicyL values (i.e. Auth, CSP, Owner). The obtained sorted list is then used to ensure a deterministic order of evaluation of the policies, to be reflected at the time of policy translation.

2) Policy rules translation: The sorted list of rules are next translated into Java policies and access structures. Access structures are defined according to the specification of the CP-ABE encryption schema [5]. CP-ABE is in fact adopted as it supports the notion of attribute-based policies as a criteria for encryption, and is complementary to the Java policies. The access structures are embedded as part of the encrypted content, as by the CP-ABE construction. Which party is entitled to decrypt the content and under which context is addressed by means of the CP-ABE boolean access structure entries. Conversely, the Java policies are stored in the data container, i.e. the SAO, and specify which party can access a specific resource (code, files), and how. For example, the Java policies help render the content according to the specific access privilege.

A local Trusted Authority (TA) connected with the PA acts as key management authority, and is the only entity which knows and manages the cryptographic key needed for the encryption of data and the Master key needed for the creation of CP-ABE decryption keys associated with the users. We specifically employ the *Encrypt(PK, T, f)* primitive of the CP-ABE protocol for encryption, where T is the access structure representing attribute-based conditions (in terms of conjunctions and disjunctions), f is the file being encrypted, while PK is a set of public parameters.

T is implicitly contained in the ciphertext. T takes the form of a directed and acyclic graph. Each graph has five special nodes, denoting the level of the policy authors (*AUTH, CSP* and *OWNER*), and the possible effects of the rules evaluation (*GRANT, DENY*). The effects are stored in two terminal nodes, denoting access *DENY* and $GRANT$, respectively. If the $DENY$ node is reached, the evaluation fails, and decryption is not allowed. Each non-special node includes boolean conditions to be evaluated, reflecting conditions against subject, object, location, and time components of the rules. Edges logically connect boolean conditions with the effect of evaluation. The detailed algorithm for building access structure T is reported in Appendix.

Once this basic structure T is created, it is decomposed in Disjunctive Normal Form (DNF), and used for encryption. The $DENY$ node is translated into a DNF atomic negation. Upon evaluation T is traversed in depth, to ensure correct hierarchical ordering: the root node is $AUTH$, and the node $OWNER$ is last child node in the subtree with root *CSP*.

The special nodes introduced in the access structure T do not affect the security of the CP-ABE schema. The first three special nodes are simple pass-by nodes, used to discriminate the policy rules priority level in our hierarchical model. The two single leaf nodes are used to re-define the definition of access structure satisfaction. In the classical model an access structure is satisfied if and only if all the leaf nodes are verified as true, in our model instead the access structure *T* is satisfied if, and only if, the leaf *GRANT* is reached.

Example 2: Figure 2 shows the translation of the policy rules of Example 1, in Java Policies and in the Access Structure T respectively. The access structure T derived from P is then used as input for the data encryption.

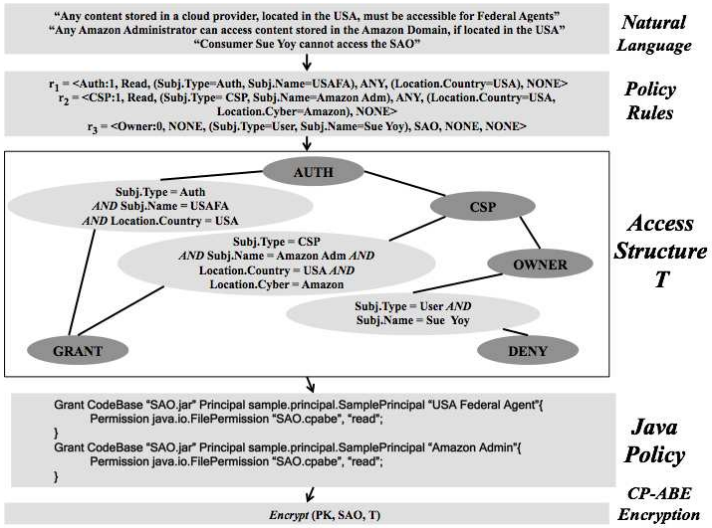

Fig. 2. Policy Rules Translation and Data Encryption

C. Access of SAO protected content

In order to access the content stored in a SAO, the authentication and authorization procedure must be executed first. A SAO exploits the JAAS technology to enforce authentication and authorization mechanisms, by using standard certificates and Java Policies files. For authentication and authorization purposes each user is uniquely identified by means of a certificate, released by the TA. The JAAS technology uses the attributes of the certificate in order to evaluate the Java policies file. The Java policy file in turn dictates if the user has grant to access the content first and, if access is permitted, what kind of actions he/she can perform on it.

Only if the authentication and authorization procedure ends with a grant privilege, the SAO proceeds toward the decryption of the content. To decrypt a SAO protected content, a user needs to exhibit some set of attributes (and his/her associated secret key) which need to satisfy the SAO content access policy. The decryption function, denoted as *Decrypt(PK, EC, SK)*, takes as input the public parameters PK, the encrypted data EC, which contains the access structure T representing the policy rules set, and the private key SK, which is a private key associated with the user for a set of attributes representing user's credentials. By construction of the CP-ABE primitive, if the set of user's credentials (attributes) satisfies the access structure T, then the algorithm will decrypt the EC.

In our case, the attributes satisfying the policy are retrieved from the subject's credential presented upon access request, and the contextual information. These attributes are used as input of the CP-ABE decryption primitive. In our system we need to take into account also of the context of the data consumption, viz. space and time of attempted access.

To this end, we extend the CP-ABE decryption primitive to include contextual information. The extended decryption primitive is *ExtDecrypt(PK, EC, SK, SPACE, TIME)*. The two values `SPACE` and `TIME` are obtained from secure channels, as discussed in Section IV-B. These are evaluated in the access structure T in conjunction with the attributes used for decryption, although not directly used to generate the secret

key SK. As before, these changes to the original decryption primitive do not undermine the security properties of the classical CP-ABE schema. The two contextual attributes are used in the decryption algorithm equally as the attributes mathematically incorporated into the key SK. Furthermore, this contextual information is retrieved from secure channels, ensuring their validity and integrity. If the set of attributes in the SK, combined with the attributed contained in the contextual information, satisfies the access structure T, the algorithm will decrypt EC.

IV. SAO DEPLOYMENT AND CONTEXT RETRIEVAL

We now present the SAO architecture and the interaction protocols from SAO to the local Trusted Authority to obtain and securely enforce context-specific policies.

A. SAO components and implementation

Architecturally, a SAO includes three components: the Application Section, the Content Section, and the Policy Section, wrapped into an external JAR.

Application Section: a set of unmodifiable software modules, signed and sealed by the PA to ensure integrity of the application. The Application Section is itself deployed in a nested JAR, and organized into four components: Authentication and Access Control Component (AAC), Action Control Component (ACC), Content Protection Component (CPC), and the Connection Manager Component (CMC).

The AAC implements the authentication and authorization mechanisms, by exploiting the services offered by Java Authentication and Authorization Services (JAAS) [1] and Security Manager. By means of JAAS primitives, the SAO accepts X.509 certificates.

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 CPC manages the protected content stored in the Content Section, as well as the protocols to re-deploy the new Content Section upon context change or time expiration. Content encryption and access control enforcement are integrated by means of the CP-ABE scheme [5].

The CMC handles all processes for creating secure connections toward the PA to receive the Content Section and toward the WorldWide Space and Time Services to retrieve the context information. The CMC component implements a Challenge-Handshake Authentication Protocol and exploits SSL connections to ensure private connections.

Content Section: stores the protected content items, the Java policy file that records the policies implementing the security rules, and an record with the last context in which the content has been accessed. Both Java policy file and the last context record are signed with the TA private key ($TA_{PrivKey}$) for integrity purposes. The Content Section represents the dynamic part of the SAO since it replaced by the TA, each time that the SAO is accessed in a new context or after a certain time period, to ensure compliance with more recent policy rules.

Policy Section: a collection of `owner` level policy rules, i.e., the set of policy rules specified by the data owner for the SAO. This section is encrypted with the user SK, thus only the TA is able to decrypt it.

B. Context Retrieval for policy evaluation

Every time the SAO 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 (Country, State/Province, City/Town) is generated by the GeoIP service [6] that uses as input only the IP address of the machine hosting the SAO. Temporal information (i.e. 10:11 AM Tuesday, March 06, 2012, Eastern Standard Time (EST) -05.00 UTC) is generated by using the WorldTime-Server service [6] that uses as input the current geographical location of the SAO.

Notice that depending on where and how the SAO is accessed, obtaining this information may require different approaches. In general, data stored on CSPs 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 CSP (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 object (in our case SAO) is downloaded from the virtual space on the user's machine. The context information is retrieved directly by the SAO that wraps the content, running on the local machine, by exploiting the Global Services [6].

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

We provide a concrete example of how the file is generated in the context of Amazon Web Services (AWS) APIs in the code snippet below.

```
AmazonS3 s3 = new AmazonS3Client(new
PropertiesCredentials(S3Sample.class.getResourceAsStream
("AwsCredentials.properties")));
s3.createBucket(bucketName);
s3.putObject(new PutObjectRequest(bucketName, key,
SAO));
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 SAO in the AmazonS3 storage service, and retrieve the content from the SAO 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 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 SAO not accessible.

C. Context Adaptiveness

If the SAO moves to an unexpected context, it forwards a policy request to the PA. The PA retrieves high-priority rules applicable to the current context, which are translated and made available by the TA. We assume that the TA is responsible of collecting low-level contextual policies, which implement local laws or regulations, as well as CSP location-specific rules. Thus, the TA updates the SAO Content Section, by adhering to the protocol sketched below:

1 The SAO sends a request to the TA, through an encrypted channel by implementing a Challenge-Handshake Authentication Protocol. In the request the SAO sends the current Content Section (containing the last context record), the current context of consumption, and the policy rules specified by the content owner (Policy Section).
2 The TA after having verified the request's authenticity, determines whether any policy rules applicable to the context specified by the SAO exist.
3 If no new policy rules exist for the new context, the TA sends an ACK OK message and the protocol ends.
4 If a new set of policy rules applies to the SAO for the new context, the TA creates a new Content Section. The creation of the new Content Section requires to re-execute the policy rules translation process and the content encryption with the new assess structure T, as presented in Section III B. During this step a new Java Policy file and a new last context record are created and signed with $TA_{PrivKey}$. Once the new Content Section is ready, the TA sends it to the SAO
5 The SAO replaces the old Content Section with the newly verified one, upon validating the signature of the TA.

V. Security Features

We discuss possible attacks to our SAO-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 by the TA. An attacker may try to access information directly from the SAO, or try to disassemble it to gain access to the protected content. We only consider attacks to the SAO. Attacks attempting to exploit the communication between SAOs and the PA are prevented by means of encrypted authenticated sessions and use of challenge response protocols.

Disassembling and Reverse Engineering: By disassembling the SAO outer JAR, an attacker will be able to get the internal elements of a SAO. The disassembled 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 [12]. 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. In fact, attributes and keys used for decrypting the content, under the CP-ABE scheme, are never stored but instead retrieved by the user's certificate when needed, at the SAO execution time. Instead, in the Content Section the Java Policy file and the Last Context Record are signed by using the $TA_{PrivKey}$ of 2048 bits, and therefore resistant against brute force attacks. For the EC we rely on the CP-ABE schema that is, by construction, resistant to collusion and chosen plaintext attacks. Finally, the Policy Section is encrypted by using the content owner SK of 2048 bits.

Security Policy modification: A malicious user might attempt to tamper with the SAO in order to change the security policy and gain access to protected content. Even if the attacker reads the Java Policy file or decrypts the Policy Section, he/she does not obtain enough information to decrypt the EC. Our architecture leverages the properties of the CP-ABE encryption schema rendering this attack not feasible. In the CP-ABE the access policies are directly and implicitly contained within the encrypted content: a user is be able to decrypt the protected content only if his/her attributes satisfy the access structure T used for content encryption. Then, a malicious user could try to build a set of attributes to satisfy the access structure T for the encrypted content. Such attack would fail, in that the attacker would need to provide a valid certificate with those attributes, for which he/she would need the CP-ABE master key known only by the TA.

Bypassing Authentication: An attacker could try to bypass the authentication process required by our application 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 due to the digital signature and the sealing, which guarantee the JAR file integrity. Alternatively, the attacker may 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 inject malicious code to bypass the authentication step so to jump to a different point of the execution. To address this problem, we exploit program tracing techniques [10], by inserting in the application code a set of checkpoints. Each checkpoint controls the current 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 application 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 or between classes. An attacker cannot eliminate the checkpoints used for program tracing because these checkpoints are directly inserted in the signed code. Any change would be therefore detected.

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 encrypted content or the Policy Section. To quickly detect a corrupted JRE we tie together various control mechanisms, and continually do crosschecks and stop any execution as soon as one of crosscheck fails. These checks include integrity checks on the system environment, validating manifest of sensitive directories, validating the runtime JRE, and checking that the Java 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 decryption, and creation of secure connections among SAO and the TA.

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 to improve the level of the operational integrity of the Java application. This approach consists of extensions to the Java 2 security system [20]. The JRE extended components can be directly contained in our SAO, and ready to be loaded at each execution. The SAO will run only if all the extended components are properly loaded, so as to ensure the proper functioning of all the Java security services. More details are reported in Appendix.

VI. Experimental Evaluation

Our evaluation testbed includes a client machine, a Mac Book Pro (with a 2.2 GHz Intel Core i7 process, and 4 GB 1333MHz DDR3 memory); and a server machine, an iMac (with a 3.06 GHz Core 2 Duo E8435 processor, and 8 GB 1333MHz DDR3 memory). The client machine acts as a content consumer, whereas the server machine hosts the TA. The client machine connects to the Amazon Web Services. In particular, we have used Amazon S3, Amazon EC2 and Amazon Cloud Drive for the storage and computing services.

We conducted several tests to evaluate the performance of our architecture. First, we evaluated the total time of

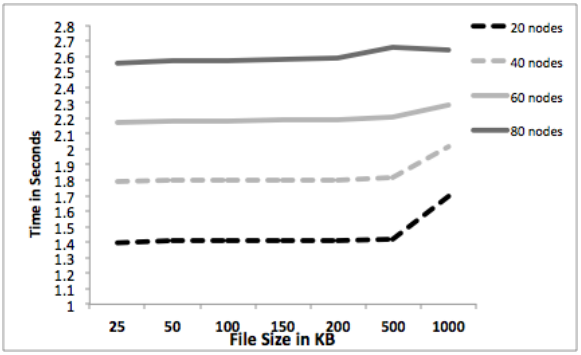

Fig. 3. Time required to reconstruct the SAO

reconstructing a SAO upon changing the context. We tested the reconstruction time by varying both the size of the SAO, and the size of the policy. We measured the size of the SAO in KB, ranging from 25 to 1000, and we measured the complexity of the policy in terms of size of the access structure, which complexity is defined in terms of nodes in the access structure. The results are reported in Figure 3. As shown, the time is not significantly affected by the file size, while there is an increase upon augmenting the size of the access structure. For a very complex policy, with 80 nodes, the reconstruction time is 2.6 seconds.

As a second experiment, we compared the delay in accessing a file with and without the SAO architecture. A file encrypted with the CP-ABE scheme (access structure with 25 nodes) is decrypted in 0.68 seconds. If the file is hosted by the SAO, the authentication, the context retrieval and the rendering procedures, added to the file, bring the access time to 1.31 seconds. A change of context adds further 1.0784 seconds to the earlier time, because in this case a communication and data exchange with the TA is required.

Next, to better investigate the cause of the overhead added by using the SAO, we studied the time consumption of the single steps characterizing the SAO execution. We repeated these tests for SAO of various sizes and access structures of increasing complexity. We omit detailed results for lack of space. On average, for an access structure with 40 nodes and a file of size 500K it takes less than 3 seconds to complete access, including accessing contextual information. The authentication and authorization steps only take 0.065 seconds. As expected, context retrieval from the Global Services is the most time consuming operation, and takes about 1.272 seconds. This time represent the time needed to the SAO to contact the two Global Services and retrieve the information to generate the actual context of consumption. The data exchange with the TA only takes 0.096 seconds, whereas the policy rules translation and evaluation takes 0.9824 seconds. Notice that these two operations are executed only if the context of consumption is changed. In case the context is not changed, the time of access drastically decreases (less than 1.3 seconds).

Finally, we tested the overall size of the SAO. In details, the size of the Application Section is fixed to 350 KB. The Content Section includes one Java Policy file, which is usually around 5 KB and the Last Context Record with a size of 1.5

KB. The size of the Java Policy file may changes slightly depending upon the number of grant and deny represented. At the end, the size of the content file is to be added also. With encryption, the size of the content increases of a few bytes.

In summary, our solution does not present significant overhead in terms of space. Yet, the time overhead could be improved. We plan to explore how to optimize the SAO reconstruction, context retrieval, and policy rules translation steps, to decrease the time overhead.

VII. Related Works

Access control [18], [21] and data management outsourcing techniques targeting the Cloud have been recently proposed [2], [9], [13]. Further, Cloud-specific cryptographic-based approaches for ensuring remote data integrity have been developed [19]. Subsequently, Wang and et al. [19] proposed a data outsourcing protocol specific to the Cloud. Wang's work is focused on auditing of stored data from trusted parties.

The notion of SAO is corroborated by previous projects [11], [7] and our own results [17], [15]. Our approaches are closely related to self-defending objects (SDO) [7] 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. As a result, adaptiveness issues are not considered. Related to the idea of self-protecting data is also the work on sticky polices [11] 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. In previous work, we proposed an approach for accountability [15] in the Cloud. This work could be integrated with the framework currently proposed, in that it builds on a related architecture.

The notion of policies used in our framework is inspired by existing work on access control policies [4], as well as secure data provenance. However, previous approaches do not ensure that policies are securely coupled with the protected data when data dynamically moves across a distributed system. 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 [8], [14]. These approaches may be suitable for secure content rendering from the SAO, in that they address confidentiality, but not distributed management.

VIII. Conclusion

We presented an approach for secure and distributed data management in the Clouds. The idea behind our solution is to protect the data by means of wrappers applied to the targeted content file(s) which are "security-aware" in that they protect the content as it travels across domains by locally enforcing security policies. The policies may be dynamic, and adaptable according to the location and other contextual dimensions that may affect the enforcement process. In the future, we plan to further develop the policy language and support more articulated access rights and contexts.

References

[1] http://java.sun.com/products/archive/jaas/.

[2] 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.

[3] E. Bertino, P. Bonatti, and E. Ferrari. Trbac: A temporal role-based access control model. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):191–233, 2001.

[4] E. Bertino, G. Ghinita, and A. Kamra. Access control for databases: Concepts and systems. *Foundations and Trends in Databases*, 3(1-2):1–148, 2011.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] A. C. Squicciarini, S. Sundareswaran, and D. Lin. Preventing Information Leakage from Indexing in the Cloud. In *3rd IEEE International Conference on Cloud Computing*, 2010.

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

[17] S. Sundareswaran, A. C. Squicciarini, D. Lin, and S. Huang. Promoting distributed accountability in the cloud. In *4th IEEE International Conference on Cloud Computing*, 2010.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

Algorithm 1 Access Structure construction

Input: Sorted rules.
Output: Access structure T
Let Auth_Rules $=\{r_1^a, \ldots, r_n^a\}$, CSP_Rules $=\{r_1^{CSP}, \ldots, r_n^{CSP}\}$, Owner_Rules $=\{r_1^o, \ldots, r_n^o\}$ be the set of applicable rules authored by the authority, CSP and consumers, respectively.
let $T = \{N, E\}$ be a directed empty graph
if $Auth\_Rules \neq \emptyset$ **then**
  $n_0 = Auth\ Root$
  $n_a =$Allow *Special Terminal Node*
  $n_d =$Deny *Special Terminal Node*
  $N = N \cup n_0 \cup n_a \cup n_d$
end if
for $i = 1$ to n **do**
  $n = (r_i^a.Subj, r_i^a.Loc)$
  $e = \{n_0, n\}$
  if $r_a^i.Effect = 1$ **then**
    $e = \{n, n_a\}$
  else
    $e = \{n, n_d\}$
  end if
  $E = E \cup e$
end for
if $CSP\_Rules \neq \emptyset$ **then**
  $n_1 = CSP$
  $N = N \cup n_1$
  $e = n_0, n_1$
end if
for $i = 1$ to n **do**
  $n = (r_i^{CSP}.Subj, r_i^{CSP}.Loc)$
  $e = \{n_1, n\}$
  if $r_i^{CSP}.Effect = 1$ **then**
    $e = \{n, n_a\}$
  else
    $e = \{n, n_d\}$
  end if
  $E = E \cup e$
end for
if $Owner\_Rules \neq \emptyset$ **then**
  $n_1 = (Owner)$
  $N = N \cup n_2$
  $e = \{n_1, n_2\}$
end if
for $i = 1$ to n **do**
  $n = (r_i^O.Subj, r_i^O.Loc)$
  $e = \{n_2, n\}$
  if $r_i^O.Effect = 1$ **then**
    $e = \{n, n_a\}$
  else
    $e = \{n, n_d\}$
  end if
  $E = E \cup e$
end for

APPENDIX

Access Tree Construction Algorithm 1 presents the steps taken to construct an access tree to be used as input of our SAO encryption.

Preventing corruption of SAOs through tampered Java Running Environments

The Java-based security extensions proposed in Section VI B 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 [20] 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 SAO, and ready to be loaded at each execution.

These extensions allow to 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.

The code snippet below shows the controls placed inside the application to check the JRE and the Vendor used during the application execution

```
if (System.getSecurityManger()==null)
System.exit(-1);

if(!System.getSecurityManger().getClass().
getName().equals(myPackage.
 ExtendedSecurityManager))
System.exit(-1);

if(!System.getClassLoader().getClass().
getName().equals(myPackage. ExtendedClassLoader))
System.exit(-1);
```

These system integrity checks are repeated multiple times 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 systems 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. The extended Security Manager generates a security exception. The extended Class Loader refuses to load any classes by throwing a security exception. This approach guarantees that whenever the system is operating, it must be operating correctly, because the extended Class Loader, the extended Policy Manager and the extended Security Manager are loaded and started at the beginning of the application execution and are continually checked. Only approved class can run in the system because the extended Class Loader loads all non-JVM classes, and the extended Policy Manager and the extended Security Manager validate digitally signed manifest of all application directory, JAR files, and configuration files.