E-GRANT: Enforcing Encrypted Dynamic Security Constraints in the Cloud

Muhammad Rizwan Asghar
Department of Computer Science
The University of Auckland
New Zealand
Email: r.asghar@auckland.ac.nz

Giovanni Russello
Department of Computer Science
The University of Auckland
New Zealand
Email: g.russello@auckland.ac.nz

Bruno Crispo
DistriNet
KU Leuven
Belgium
Email: bruno.crispo@kuleuven.be

***Abstract*—Cloud computing is an established paradigm that attracts enterprises for offsetting the cost to more competitive outsource data centres. Considering economic benefits offered by this paradigm, organisations could outsource data storage and computational services. However, data in the cloud environment is within easy reach of service providers. One of the strong obstacles in widespread adoption of the cloud is to preserve confidentiality of the data. Generally, confidentiality of the data could be guaranteed by employing existing encryption schemes. For regulating access to the data, organisations require access control mechanisms. Unfortunately, access policies in cleartext might leak information about the data they aim to protect. The major research challenge is to enforce dynamic access policies at runtime, *i.e.,* enforcement of dynamic security constraints (including dynamic separation of duties and Chinese wall) in the cloud. The main challenge lies in the fact that dynamic security constraints require notion of sessions for managing access histories that might leak information about the sensitive data if they are available as cleartext in the cloud. In this paper, we present E-GRANT: an architecture able to enforce dynamic security constraints without relying on a trusted infrastructure, which can be deployed as Software-as-a-Service (SaaS). In E-GRANT, sessions' access histories are encrypted in such a way that enforcement of constraints is still possible. As a proof-of-concept, we have implemented a prototype and provide a preliminary performance analysis showing a limited overhead, thus confirming the feasibility of our approach.**

I. Introduction

With its cost-effective model, cloud-based services are very attractive for enterprises and government sectors. Initially developed as a cheap storage solution (monthly $0.026/GB and $0.03/GB, as of March 2015, offered by Google [1] and Amazon [2], respectively), the cloud paradigm today is able to offer affordable software solutions. The term Software-as-a-Service (SaaS) is used to indicate software products offered as a service through the cloud. Several vendors have adopted this model to offer their products at a more affordable price. Classes of software products available as SaaS range from document management tools (such as Google Drive [3]) to image processing tools (such as Adobe Photoshop [4]).

Recently, Business Process Management (BPM) solutions have become available as SaaS from major players in this field, such SAP with its Business ByDesign [5]. One of the crucial aspects of such systems is the enforcement of access control decisions for assigning human resources to execute tasks. If this control is too restrictive then it could hamper the productivity. On the other hand, a very lax approach might undermine the confidentiality of sensitive data (when accessed by unauthorised users), resulting in serious consequences for the organisation. In such a system, the access control mechanism has to take into account business-related notions such as conflict-of-interests. Typical examples are that of an employee able to execute two tasks that might lead to fraudulent actions and that of an employee executing the same task over two different sets of data that could be in conflict with each other. Over the years, a huge amount of research effort has been put on this topic. The results have culminated in identifying and enforcing dynamic security constraints [6]–[10].

For correctly enforcing dynamic security constraints, the cloud server needs to maintain history of all actions executed by the entities that it controls, as well as contextual information of the requester (*e.g.,* time and location). When the server receives a new request, it checks whether allowing the current request violates any constraints in view of the earlier actions performed by the same (group of) requesters. State-of-the-art enforcement techniques [7], [11]–[13] rely on a trusted infrastructure, which expects information to be in cleartext. That is, the history of actions, contextual information, and constraints are all stored in cleartext to be readily accessible.

With the move towards outsourced solutions, the trust assumptions in the management of the infrastructure do not hold any longer. The cloud providers that have control over the hardware, where data and security constraints reside or are evaluated, could easily have access to them. The data can be protected using encryption techniques [14]–[16]; however, state-of-the-art enforcement techniques [7], [11]–[13] lack to preserve confidentiality of dynamic security constraints because they expect all information in cleartext at both deployment and enforcement times. The problem here is that learning about the security constraints might leak information about the data itself. There are some cryptographic techniques that can enforce static security constraints in outsourced environments [17]–[21]. Unfortunately, there is no cryptographic solution that can enforce dynamic security constraints in the cloud.

In this paper, we want to fill this gap and propose a novel solution for enforcing dynamic security constraints that can be offered either as a stand-alone SaaS solution or integrated with other SaaS products that require the enforcement of these constraints. Unlike state-of-the art solutions for enforcing only static constraints, our novel solution is capable of enforcing

dynamic security constraints without revealing sensitive information, such as the dynamic session information, to the untrusted infrastructure.

We named our solution *E-GRANT (EnforcinG encRypted dynAmic security constraiNts in The cloud)*. E-GRANT makes significant research contributions. Most importantly, E-GRANT can enforce constraints while taking into account contextual information (such as time and location of the user) without revealing any information to cloud providers. In our mechanism, an administrator (of the organisation that uses cloud services) can specify constraints with contextual conditions including non-monotonic boolean expressions and range queries. In E-GRANT, constraints as well as session information are encrypted. The encryption scheme we use is such that it does not require users to share any encryption keys. In case a user leaves the organisation, the system is still able to perform its operations without requiring re-encryption of constraints or access histories managed by the session. Finally, we have implemented a prototype of E-GRANT and analysed its performance to quantify the incurred overhead.

The rest of this paper is organised as follows. Section II provides an overview of the dynamic security constraints supported in E-GRANT. Section III describes the E-GRANT architecture. Section IV focuses on solution details. Section V provides security analysis. Section VI describes implementation details and analyses the performance overhead of the E-GRANT prototype. Section VII reviews the related work. Finally, Section VIII concludes this paper and gives directions for future work.

II. Dynamic Security Constraints in E-GRANT

E-GRANT focuses mainly on enforcing dynamic security constraints. There are two variants of dynamic security constraints: (i) Dynamic Separation of Duties (DSoD) [8], [9] and (ii) Chinese Wall (CW) [10]. Both DSoD and CW can be implemented by maintaining access history for each entity active in the system [22]. At each new request, the system has to check that none of the defined constraints are violated by granting the received request with respect to the earlier actions performed by the same (group of) requesters. With each variant of constraints, it is possible to specify contextual conditions, *i.e.,* enforcing constraints while taking into account contextual information, such as time and location of the requester. In the following, first we briefly explain both variants and then we describe contextual conditions.

A. Dynamic Separation of Duties

DSoD constraints [8], [9] aim at providing multi-user control over the resources when there is any conflict-of-interest for completing a business process. There are multiple categories of DSoD varying from coarse-grained to fine-grained levels, as discussed in [23]. In **Simple DSoD (SDSoD)**, *a user may be a member of two mutually exclusive roles but must not be active in both roles simultaneously.* In **Object-Based DSoD (ObDSoD)**, *a user may be active in mutually exclusive roles simultaneously, but must not act in both roles upon a single object.* In **Operational DSoD (OpDSoD)**, *a user may be active in mutually exclusive roles simultaneously, but must not get authorised to execute all actions of a business process.* In **History-Based DSoD (HBDSoD)**, *a user may be active in mutually exclusive roles simultaneously, but the user must not get authorised to execute all actions of a business process involving the same object.* For example, a user active in both clerk and manager roles can either issue or approve a particular instance of the *purchase order*. HBDSoD combines ideas behind ObDSoD and OpDSoD, requiring a detailed access history on each object. Thus, it is the most fine-grained category of DSoD.

B. Chinese Wall

A CW constraint [10] prevents users to access an object belonging to a domain which is in conflict-of-interest with other domain whose object is previously accessed by the same (group of) users. In other words, a CW constraint aims at providing confidentiality by preventing illegitimate information flow between domains that are in conflict-of-interest. For instance, let us consider the consultant organisation that provides services to companies that are in conflict-of-interest, say Google and Microsoft. The CW constraint will help the consultant organisation to enforce the policy that an employee can work at either Google or Microsoft but cannot work at both companies.

III. The E-GRANT Architecture

The E-GRANT architecture aims at enforcing dynamic security constraints in outsourced environments in such a way that contents of constraints, contextual conditions, session information for maintaining access histories and contents of the request are not revealed to cloud providers because they are encrypted. Therefore, the enforcement mechanism can be deployed in the cloud without the need of fully trusting administrators of cloud providers. Our main goal here is to protect the confidentiality of information used by the enforcement mechanism for taking its access control decisions. The rationale behind this is that even if the data is protected (*e.g.,* encrypted) a curious administrator might learn information about the data by inspecting the constraints and access histories that are typically deployed in cleartext.

A. The System Model

Let us assume an organisation ORG that uses cloud services and deploys E-GRANT for enforcing sensitive dynamic security constraints. There are the following entities in E-GRANT:

Admin User: An Admin User, who works for ORG, is responsible for deploying, updating and deleting dynamic security constraints.

Requester: A Requester, who also works for ORG, is a user that can make requests to access resources and execute actions in the system.

Outsourced Enforcement Module (OEM): It is responsible for storing and enforcing dynamic security constraints. In E-GRANT, the OEM is deployed as SaaS in the outsourced environment, managed by the cloud provider and paid by ORG.

Trusted Key Management Authority (TKMA): It is a trusted authority responsible for generating keys used for protecting

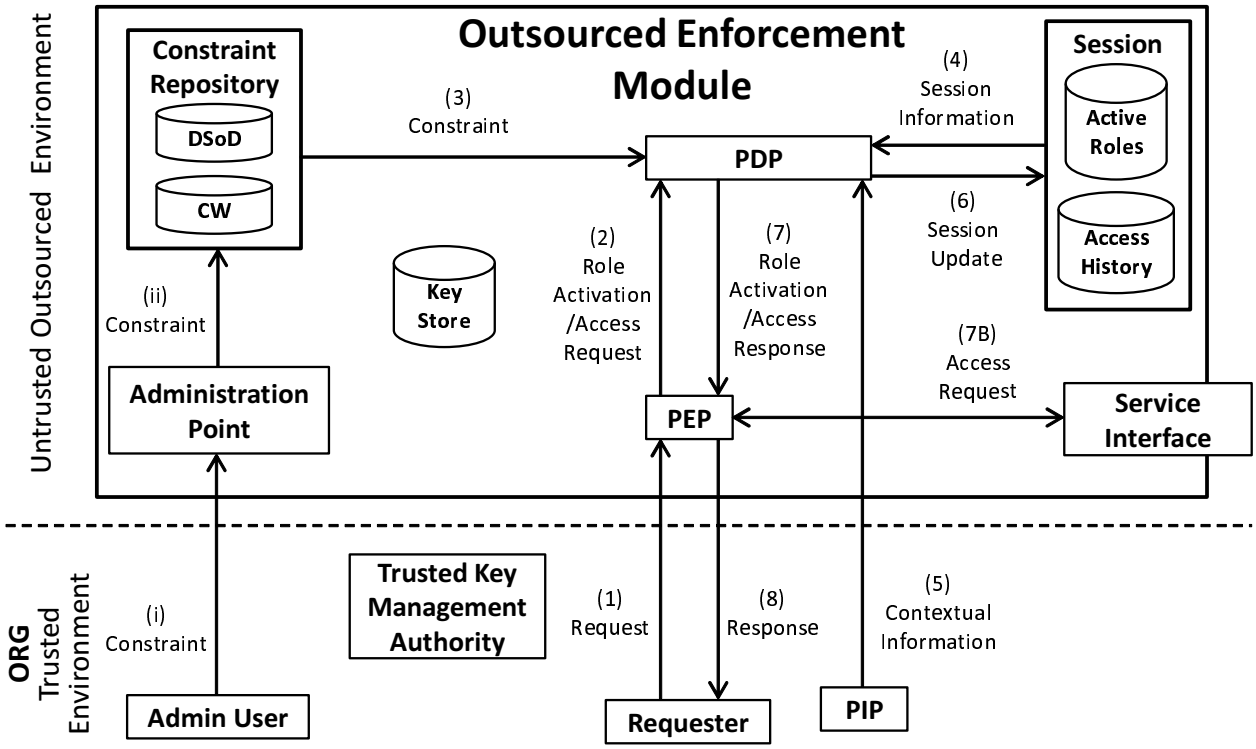


Fig. 1. The E-GRANT architecture for enforcing dynamic security constraints in outsourced environments.

data stored on the OEM. For each user (be it an Admin User or a Requester), the TKMA generates the client key set and the server key set that are sent to the user and the OEM, respectively. The OEM stores all server side key sets in the Key Store and is responsible for revoking users. The TKMA is only the minimal infrastructure that is run within a trusted environment. However, the TKMA can be kept offline because it generates the key only once when any user gets registered with the system.

The Threat Model. We assume that cloud provider is *honest-but-curious* (as assumed in [20], [21]): that is, it allows the components to follow the protocol for performing requested actions but curious to deduce information about contents of constraints, access histories and requests. We also assume that users (Admin Users and Requesters) may collude. Furthermore, we consider that the TKMA is fully trusted and plays a role at the time of system initialisation. Last but not least, we assume only passive adversaries and do not consider active adversaries that can manipulate the exchanged information.

B. Deployment of Constraints

In E-GRANT, an Admin User can deploy new constraints and update (or delete) existing constraints. For deploying new constraints, an Admin User sends the (i) Constraint to the OEM as shown in Figure 1. The Administration Point is a component of the OEM that receives (i) and then stores it in the Constraint Repository (ii), which is managed by the OEM.

C. Evaluation of Constraints

A Requester can send a (1) Request to the OEM as illustrated in Figure 1. The Policy Enforcement Point (PEP) receives (1) and then identifies whether (1) is a role activation request or an access request. The PEP forwards the (2) Role Activation/Access Request to the Policy Decision Point (PDP). The PDP is the core component that can grant the request after evaluating the deployed constraints. For evaluating constraints, the PDP fetches the (3) Constraint from the Constraint Repository and the (4) Session Information from the Session component of the OEM. The Session component maintains two repositories including Active Roles and the Access History. Active Roles is a repository that keeps record of roles that have been activated for a Requester while the Access History is a repository that maintains what information has been accessed by a Requester. The Session Information can include information about active roles or the access history; thus, it plays a vital role in evaluating the constraints.

The constraints could be enforced under some contextual conditions. A PDP evaluates contextual conditions after collecting contextual information, such as time and information about the Requester, *e.g.,* her location. The Policy Information Point (PIP) is a trusted entity that provides (5) Contextual Information to the PDP. The contextual information must satisfy contextual conditions for the successful enforcement of constraints.

After the evaluation, the PDP sends the (7) Role Activation/Access Response to the PEP. The response in (7) is either *allow* or *deny* depending on the PDP evaluation. In case of *allow*, the PDP updates the session with the role activation or

access information by sending the (6) Session Update message to the Session. The PDP forwards its decision to the PEP. If the decision is *allow*, the PEP forwards (7B) Access Request to the Service Interface. Finally, the PEP may send the (8) Response to the Requester.

IV. Solution Details

A. Representation of Constraints

For representing both DSoD and CW constraints[1], we extend the tree structure proposed by Bethencourt *et al.* in [24], which they used for representing Ciphertext-Policy Attribute-Based Encryption (CP-ABE) policies. Internal nodes of the tree represent AND, OR or threshold gates (*e.g.,* 2 out of 3) while leaf nodes represent values of the condition predicates of a constraint. We extend the tree structure by encrypting leave nodes in the tree that basically represent variable and values associated with conditions. Figure 2 illustrates an example of the HBDSoD constraint, where a Requester can execute either *issue* or *approve* but not both actions on the same instance of the *purchase order*.

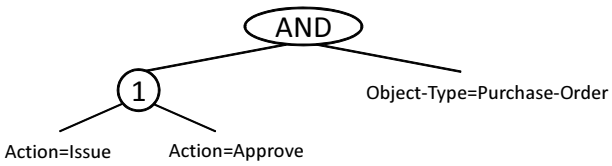


Fig. 2. An example of HBDSoD, where a Requester's *action can be 1-of-(Issue,Approve) AND Object-Type is Purchase-Order*.

B. Representation of a Request

The access request can be represented as a tuple $REQ = \langle R, A, O, I \rangle$, where R is role of the Requester, A indicates the action to be taken, O and I describe type of the object being accessed and its instance identifier, respectively. For instance, consider a Requester, active in a role *manager*, takes the *approve* action over the instance of a *purchase order*. The object type O may be a fully qualified name that may include the domain hierarchy an object type may belong to. For example, consider a CW constraint, where a Requester (employed by a consultant organisation) cannot work on instances belonging to both Google's marketing project and Microsoft's marketing project. Here, the object type O is *Project* while the domain hierarchy is: *Google/Marketing* and *Microsoft/Marketing*. In case, if it is the role activation request then a Requester just needs to send her role. Thus, the access request is more complex than the role activation request; therefore, we will focus more on the access request in rest of the paper.

C. Technical Details

The main idea behind E-GRANT is to employ the encryption scheme for protecting constraints and the sessions while delegating the enforcement mechanism to the OEM. Our proposed encryption scheme uses as a building block the proxy encryption proposed by Dong *et al.* [25]. The proxy encryption in [25] handles only a single keyword; whereas, our proposed scheme extends it by incorporating complex conditional expressions, such as range queries. The encryption scheme is multi-user, where users can write or read the data.

Initialisation: A TKMA generates two prime numbers p and q such that $q|p-1$. It generates g such that $\mathbb{G}$ is the unique order q subgroup of $\mathbb{Z}_p^*$. It chooses a random $x \in \mathbb{Z}_q^*$, which is a master secret key. It publishes public parameters including $\mathbb{G}$, g, q, $h = g^x$, a collision-resistant hash function H, a pseudorandom function f and a random key s for f.

Key Generation: In E-GRANT, each user (including an Admin User and a Requester) gets a client side key set, a random $x_{i1} \in \mathbb{Z}_q^*$ and s, from the TKMA while the OEM as a proxy server also receives a server side key set, $x_{i2} \leftarrow x - x_{i1}$, corresponding to that user i. The OEM maintains all these key sets in a Key Store, which can be accessed by different components of the OEM including the Administration Point, the PDP and the PEP.

Constraint Deployment: For deploying a constraint, an Admin User performs the first round of encryption using the client side key set. In this round of encryption (see **ClientEnc(.)**), each leaf node e of the constraint tree is encrypted while non-leaf nodes representing AND, OR or threshold gates are in cleartext. After the first round of encryption, constraints are protected but they cannot be enforced yet as they are not in common format. To convert constraints into a common format, the Administration Point of the OEM performs the second round of encryption (see **ServerReEnc(.)**) using the server side key set corresponding to the same Admin User who performed the first round of encryption. In fact, the second round of encryption by the Administration Point serves as a proxy encryption. The common format implies that the constraints get encrypted with the master secret key, which is known neither to any users nor to the OEM. Like the first round of encryption, each leaf node of the tree representing the security constraint is re-encrypted. Finally, the re-encrypted constraints are stored by the Constraint Repository.

- **ClientEnc**(e, x_{i1}, s)**:** The client chooses a random $r \in \mathbb{Z}_q^*$. It computes: $\sigma = f_s(e)$. Finally, it generates $c_i^*(e) = (\hat{c}_1, \hat{c}_2, \hat{c}_3)$, where $\hat{c}_1 = g^{r+\sigma}$, $\hat{c}_2 = \hat{c}_1^{x_{i1}}$ and $\hat{c}_3 = H(h^r)$.
- **ServerReEnc**$(c_i^*(e), i)$**:** The server retrieves from the Key Store the key x_{i2} corresponding to the user i. Next, it computes: $c(e) = (c_1, c_2)$, where $c_1 = (\hat{c}_1)^{x_{i2}}.\hat{c}_2 = \hat{c}_1^{x_{i1}+x_{i2}} = (g^{r+\sigma})^x = h^{r+\sigma}$ and $c_2 = \hat{c}_3 = H(h^r)$.

If an encrypted request satisfies any encrypted deployed constraint, then the session information is required to be matched against elements of the constraint. That is, the session information is matched with those elements of the constraint that are not present in the request. For example, let us consider the SDSoD constraint, where a user may be a member of two mutually exclusive roles clerk and manager but must not be active in both roles simultaneously. Let us assume that the requester's role is clerk. Since the requester's role is matched against the same role in the constraint, the OEM will consult the session to check if the same user is active in manager's

[1]For brevity reasons, we omit details of some operations (including enforcement of SDSoD, ObDSoD and OpDSoD) and cover the most complex operations offered by E-GRANT including enforcement of HBDSoD and CW.

role. For performing such a check, the OEM requires trapdoors of the constraint because only trapdoors could be matched with the encrypted information. That is why, trapdoors are stored along with the encrypted constraint at deployment time. For calculating these trapdoors, an Admin User performs the first round of trapdoor generation (see **ClientTD(.)**) using the client side key set for each leaf node e in the request while the OEM performs the second round of trapdoor generation (see **ServerTD(.)**) using the server side key set corresponding to that Admin User. The trapdoor representation does not leak any information.

- **ClientTD**(e, x_{i1}, s)**:** The client chooses a random $r \in \mathbb{Z}_q^*$. It computes: $\sigma = f_s(e)$. Finally, it generates $td_i^*(e) = (t_1, t_2)$, where $t_1 = g^{-r}g^{\sigma}$ and $t_2 = h^r g^{-x_{i1}r} g^{x_{i1}\sigma} = g^{x_{i2}r} g^{x_{i1}\sigma}$.
- **ServerTD**$(td_i^*(e), i)$**:** The server retrieves from the Key Store the key x_{i2} corresponding to the user i. Next, it computes: $td(e) = T = t_1^{x_{i2}}.t_2 = g^{x\sigma}$.

Request: For making a request, a Requester generates REQ and transforms it into trapdoors (see **ClientTD(.)**) using the client side key set for each element in the request. That is, there is a trapdoor for each element in REQ. Finally, REQ is sent over to the PEP of the OEM.

Constraint Evaluation: The deployed constraints are checked when the OEM receives a request from any Requester. The request is not in the common format yet and requires another round of the trapdoor generation. In the second round of trapdoor generation (see **ServerTD(.)**), the PEP generates the server side trapdoors for each element in REQ. Next, the PEP forwards the request to the PDP. The PDP fetches encrypted constraints from the Constraint Repository and matches it against the encrypted request. More specifically, each element in the constraint is matched against each element in the request (see **MatchElement(.):**). If a match is found then, in order to enforce the constraint, certain elements of the constraint (*i.e.,* all elements except one that is present in the request) are required to be matched against the session information. Recall that the constraint is represented as a tree, where each leaf is encrypted. In order to enforce the constraint, each leaf node is matched against elements in the session. After evaluating all leaf nodes, the OEM evaluates all internal nodes up to the root node. If the root node is matched, the request will be denied.

- **MatchElement**$(c(e), td(e))$**:** The server matches the encrypted element against the trapdoor: it evaluates $c_2 \stackrel{?}{=} H(c_1.T^{-1})$ and returns $true$ in case of a match and $false$ otherwise.

While performing the encrypted match between the encrypted session information and the encrypted constraint/request, the OEM does not reveal the content. If contextual information is required to be matched, it is matched in the same way as other elements of the constraint/request are matched against the session information. After checking the session information, if the constraint is not satisfied, the access is permitted and the role activation (or the access) response is sent from the PDP to the PEP as *allow*.

Session Update: If the evaluation is successful, the PDP updates the session to maintain the access history, as well as active roles. For updating the session, the PDP requires the request (and contextual information). The Requester may send encrypted request along with the trapdoors of the request. Alternatively, the PDP/PEP can collect this information after the PDP evaluation is succeeded. In both cases, the OEM performs the second round of encryption and finally updates the Session with the encrypted request. Finally, the PEP may send a response to the Requester.

User Revocation: In E-GRANT, users (both Admin Users and Requesters) do not share any keys and even if a compromised user is removed, there is no need to re-encrypt deployed constraints or re-distribute keys. For removing a user from the system, the Administration Point of the OEM takes the user identifier and then removes the server side key corresponding to that user from the Key Store.

V. Security Analysis

Definition 1 (Negligible Function)**.** *A function f is negligible if for each polynomial $p(.)$ there exists N such that for all integers $n > N$ it holds that $f(n) < \frac{1}{p(n)}$.*

Definition 2 (Decisional Diffie-Hellman (DDH) Assumption)**.** *The DDH problem is hard regarding a group $\mathbb{G}$ if for all Probabilistic Polynomial Time (PPT) adversaries $\mathcal{A}$, there exists a negligible function $negl$ such that $|Pr[\mathcal{A}(\mathbb{G}, q, g, g^{\alpha}, g^{\beta}, g^{\alpha\beta}) = 1] - Pr[\mathcal{A}(\mathbb{G}, q, g, g^{\alpha}, g^{\beta}, g^{\gamma}) = 1]| < negl(k)$ where $\mathbb{G}$ is a cyclic group of order q ($|q| = k$) and g is a generator of $\mathbb{G}$, and $\alpha, \beta, \gamma \in \mathbb{Z}_q$ are uniformly randomly chosen.*

Theorem 1. *If the DDH problem is hard relative to $\mathbb{G}$, then ClientGeneratedConstraint CGC is INDistinguishable under Chosen-Plantext Attack (IND-CPA) secure against the server S, i.e., for all PPT adversaries $\mathcal{A}$ there exists a negligible function $negl$ such that:*

$$Succ^{\mathcal{A}}_{CGC,S}(k) = Pr\left[b' = b \;\middle|\; \begin{array}{c} (param, msk) \leftarrow Init(1^k) \\ (K_u, K_s) \leftarrow KeyGen(msk, U) \\ w_0, w_1 \leftarrow \mathcal{A}^{CGC(K_u,\cdot)}(K_s) \\ b \stackrel{R}{\leftarrow} \{0,1\} \\ c_i^*(w_b) = CGC(x_{i1}, w_b) \\ b' \leftarrow \mathcal{A}^{CGC(K_u,\cdot)}(K_s, c_i^*(w_b)) \end{array} \right] < \frac{1}{2} + negl(k) \quad (1)$$

Proof. Recall that CGC encrypts each leaf node in the access tree using two variants of encryption: $ClientEnc$ and $ClientTD$. Therefore, the security of CGC boils down to the security of $ClientEnc$ and $ClientTD$. As proved in [25], both $ClientEnc$ and $ClientTD$ are INDistinguishable under Chosen Plaintext Attack (IND-CPA) secure under the assumption the DDH problem is hard relative to the group $\mathbb{G}$. Thus, CGC is also IND-CPA secure.

Theorem 2. *If the DDH problem is hard relative to $\mathbb{G}$, then ClientGeneratedRequest CGR is IND-CPA secure against the server S, i.e., for all PPT adversaries $\mathcal{A}$ there exists a negligible function $negl$ such that:*

$$Succ^{\mathcal{A}}_{CGR,S}(k) = Pr\left[b' = b \;\middle|\; \begin{array}{c} (param, msk) \leftarrow Init(1^k) \\ (K_u, K_s) \leftarrow KeyGen(msk, U) \\ w_0, w_1 \leftarrow \mathcal{A}^{CGR(K_u,\cdot)}(K_s) \\ b \xleftarrow{R} \{0,1\} \\ c_i^*(w_b) = CGR(x_{i1}, w_b) \\ b' \leftarrow \mathcal{A}^{CGR(K_u,\cdot)}(K_s, c_i^*(w_b)) \end{array} \right]$$
$$< \tfrac{1}{2} + negl(k) \tag{2}$$

Proof. Recall that CGR encrypts each element in the request using two variants of encryption: $ClientTD$ and $ClientEnc$. Like CGR, the security of CGR boils down to the security of $ClientTD$ and $ClientEnc$. As proved in [25], both $ClientTD$ and $ClientEnc$ are IND-CPA secure under the assumption the DDH problem is hard relative to the group $\mathbb{G}$. Thus, CGR is also IND-CPA secure. For further details, an interested reader is referred to [26].

VI. Performance Analysis

In this section, we quantify E-GRANT cryptographic operations performed at both the client and the server sides. During this performance evaluation, we are not taking into account the latency introduced by the network. In the following, we first describe implementation details of the prototype we have developed. Next, we show the performance evaluation of: (i) deploying dynamic security constraints, (ii) making a request, (iii) evaluating dynamic security constraints and (iv) finally updating session with the information within the request.

A. Implementation Details

We have developed a prototype of E-GRANT for enforcing dynamic security constraints. The prototype is implemented in Java 1.6. For this prototype, we have designed all the components of the architecture required for deploying and evaluating constraints.

The security parameter we have considered in our experimentation is of size 1024 bits. We have tested our E-GRANT prototype on a single node based on an Intel Core2 Duo 2.2 GHz processor with 2 GB of RAM, running Microsoft Windows XP Professional version 2002 Service Pack 3. The values of the execution time shown in the following graphs are averaged over 1000 iterations.

B. Performance Analysis of Deploying Dynamic Security Constraints

In this section, we analyse the performance of deploying dynamic security constraints. We measure the performance of deploying both types of security constraints including HBDSoD and CW. The simplest HBDSoD constraint is defined as *either of two actions*. For increasing complexity of the HBDSoD constraint, we can consider more than two actions using the following notation: $HBDSoD(Ya)$, where Y (≥ 2) denotes the number of actions in the constraint. Similarly, the simplest CW constraint is defined at the object level, meaning a user cannot access an instance of an object whose instance has already been accessed. In order to increase the complexity of the CW constraint, we can include the domain hierarchy. Generally, the CW constraint can be represented as:

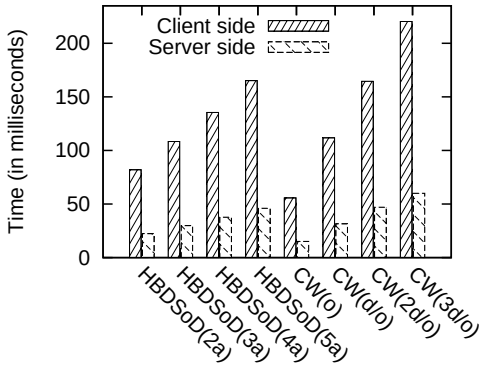


Fig. 3. Performance overhead of deploying dynamic security constraints.

$CW(Zd/o)$, where Z (≥ 0) denotes the number of domains that may be present in the domain hierarchy. If the constraint is at the object level, the value of Z will be 0 and constraint would become CW(o). However, if the constraint includes any domains, then the value of Z will be more than 0. For instance, if there is one domain then the constraint would be represented as CW(d/o). Similarly, if there are two domains (*i.e.,* one domain and one subdomain) in the domain hierarchy of an object then the constraint would be represented as CW(2d/o) and so on.

Figure 3 indicates the performance overhead incurred by deploying constraints on both the client and the server sides. During the performance evaluation, we consider both HBDSoD and CW constraints, each with varying level of complexity, where number of actions in the HBDSoD constraint are varied from 2 to 5 (with step size 1) and number of domains in the CW constraint are varied from 0 to 3 (with step size 1), respectively. As we can expect, the performance overhead of each type of constraint grows linearly if we gradually increase its complexity. Furthermore, we can observe that algorithms on the client side take more time as compared to that of the server side for deploying any type of constraints. This is mainly due to the fact the client side performs more complex cryptographic operations such as random number generations and hash calculations than the respective algorithms on the server side. However, these operations are executed only when the Admin User has to deploy a new constraint or update existing ones. On the other hand, constraints are evaluated every time a request is made. Thus, the performance of generating requests and evaluating constraints, which are measured in the following sections, is of great importance, considering it could impact the latency for providing access to the data.

C. Performance Analysis of Generating Requests

In this section, we analyse the performance of generating access requests on the Requester's client side. To make the access request, a Requester has to generate the $REQ = \langle R, A, O, I \rangle$ tuple representing that role R is requesting to perform action A on instance I of object type O. Each element of REQ is transformed into trapdoors, necessary for performing the match against encrypted HBDSoD or CW constraints deployed on the OEM. The trapdoor representation does not leak information on elements of REQ. Furthermore, each element of REQ is also encrypted, necessary for storing

the REQ tuple as encrypted in the session after REQ is granted. The time required to generate such a tuple is around 120 milliseconds as shown in the graph of Figure 4.

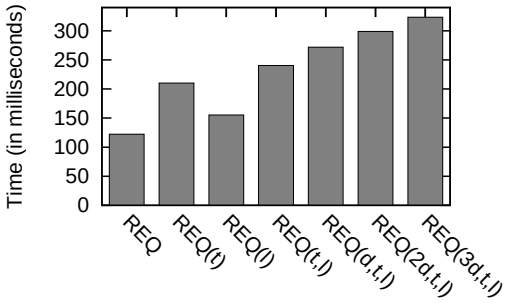


Fig. 4. Performance overhead of generating access requests on the Requester's client side.

In our experiments, we considered case in which the contextual information is included with every REQ tuple. We selected two types of contextual information: the time and the location of the Requester. As we explained in Section IV, the time t is represented as three elements indicating the office hour (from 9:00 to 17:00 hrs *i.e.,* 8 options) while the location l is represented as a single string element.

The graph in Figure 4 shows the performance overhead incurred at the Requester's client when the REQ tuple contains the value of time t ($REQ(t)$ in the graph) and location l ($REQ(l)$ in the graph). As can be seen in the graph, the time incurs higher overhead than the location because the time value t is represented as three elements, requiring generation of three trapdoors. On the other hand, the value l of the location is represented by just a single element, requiring generation of only a single trapdoor. We also measured the case in which both time and location trapdoors are generated with the REQ tuple and the overhead is a combination of two previous cases ($REQ(t, l)$ in the graph).

When CW constraints are enforced, it might be needed to include additional information about the target resource within the REQ tuple. This additional information is the domain hierarchy an object type may belong to. In the domain hierarchy, there may be multiple levels of domains. The trapdoors representing this information need also to be generated by the Requester's client. We performed experiments, considering combinations of time, location and domain information in the REQ tuple. Moreover, we also varied the depth of the domain hierarchy from one domain level (represented as $REQ(t, l, d)$) to three levels (represented as $REQ(t, l, 3d)$). The last three values in Figure 4 provide the measurements for these cases. As it is quite obvious, the performance overhead of generating these requests increases linearly with the increase in domains levels. However, it should be noticed that even in the worst case (where time, location and three domain levels are inserted in the REQ tuple), the average time for generating a request is still below 325 milliseconds.

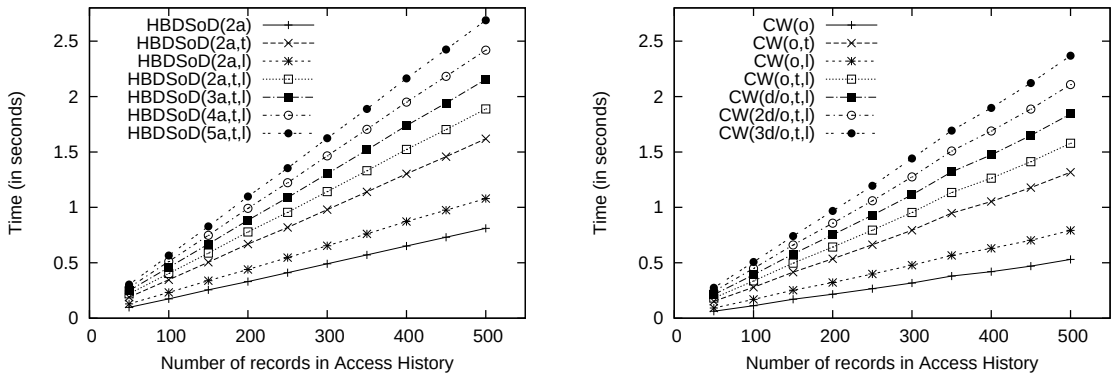
D. Performance Analysis of Evaluating Dynamic Security Constraints

HBDSoD: Let us assume that a Requester makes a request REQ for executing the action *approve* on the object type *purchase order*. As an example of a HBDSoD constraint, let us consider one that limits a Requester to execute only one out of the two actions *issue* and *approve* that can be executed on a particular instance of a *purchase order*. First, the PDP matches the object type in REQ with the object type of the deployed constraints in the Constraint Repository. If the match is successful, the PDP will match the action in REQ with one of the action specified in the HBDSoD constraint. On the second successful match, the PDP has to check that the Requester has not executed the *issue* action on this specific instance of *purchase order* in the past. To perform this check, the PDP searches in the Access History to find all records where the object type and instance match with that of REQ tuple. If such a record is found then the PDP checks if the action value in the records matches the k-out-of-n condition of the HBDSoD constraint. In the context of example we considered, it means the PDP searches in the Access History to find any records containing action *approve*. If this is the case, the constraint is violated and the PDP will not grant the action. Otherwise, the Requester can *issue* the *purchase order*.

From the above example, it is clear that the performance of enforcing a constraint depends on three main factors. The first factor is the number of constraints deployed in the Constraint Repository. When a request arrives, the PDP has to find in the repository a matching constraint. Finding a matching constraint clearly depends on the number of constraints in the repository. The second factor is the number of elements specified in the constraint. These elements can include two or more actions that could be executed only once by a Requester on a given instance of an object. Moreover, contextual information can be taken into account. Finally, the other major factor is the number of records in the Access History that the PDP has to search to check whether a given constraint is violated or not. Asymptotically, the enforcement of HBDSoD constraints is $O(Y \cdot c \cdot r)$, where Y is the number of actions specified in the constraints, c is the number of constraints deployed in the repository and r is the number of records in the Access History.

To measure the performance overhead, we performed the following experiments. We deployed 100 different HBDSoD constraints in the repository such that the one that matches the incoming request is the last one. This, of course, represents the worst case scenario. To study how the complexity of the constraint specification and number of records in the Access History affect the performance of the constraint evaluation, we execute several runs of our experiments varying the constraint complexity and number of records. Figure 5(a) shows the evaluation time in seconds in different settings. As we can observe in Figure 5(a), the evaluation time increases with the increase in the number of actions in the constraint (from 2 actions up to 5) and when contextual information such as time t and/or location l of the Requester are also considered. Similarly, the evaluation time increases with the increase in the number of records in the Access History.

CW: A CW constraint enforces that a Requester cannot gain access to two mutually exclusive objects. When a request



(a) (b)

Fig. 5. Performance overhead of evaluating (a) HBDSoD and (b) CW on the OEM.

REQ tuple is received, the PDP has to search the CW constraints relevant to the object type specified in the request tuple. Basically, the object type in the request tuple has to match one of the object types specified in a CW constraint. If a match is found, the PDP has to search in the Access History for a record containing the object type specified in the constraint that is not matched with that of the REQ tuple (and that is relevant to the Requester). If such a record is found, it means the constraint is violated; that is, the Requester has accessed in the past a object type that is in conflict with the one specified in the current request. In this case, the action in the request will not be permitted. The CW constraints can be specified at the level of object types. However, a fine-grained specification may be achieved if the domain hierarchy, objects may belong to, is also taken into account. In this case, we assume that REQ and records in the Access History repository have the domain information at the same level (where level indicates number of domains) as is present in the constraint, where each element of the domain information in REQ will be matched with the corresponding element in the constraint.

As for the HBDSoD constraints, the time for evaluating the CW constraints depends on the number of deployed constraints in the repository, the complexity of the constraint specification and the number of records in the Access History. Thus, the asymptotic complexity can be calculated as $O(Z \cdot c \cdot r)$ similarly to that of HBDSoD constraints. To measure the actual overhead, we performed a similar set of experiments as conducted for HBDSoD constraints. We deployed 100 different CW constraints and considered the worst case scenario. We then changed the number of elements in the constraint and the number of records in the Access History. The results are shown in Figure 5(b).

The above results clearly show that there is a penalty to be paid for the enforcement of encrypted constraints in outsourced environments. The execution time varies from 100 milliseconds to 2.5 seconds as number of records in the Access History increase from 100 to 500. To be fair, our experiments have been executed with very basic hardware. We expect that our solution would be able to perform better with more dedicated resources, such as servers deployed in a cloud infrastructure. Moreover, all the executions have been performed as a centralised solution. Clearly, having in these settings a single PEP and a single PDP to process all the incoming requests represent a bottleneck. To solve this problem, we are planning to develop a distributed version of our architecture that can be deployed on multiple nodes and adapted to the actual request demand.

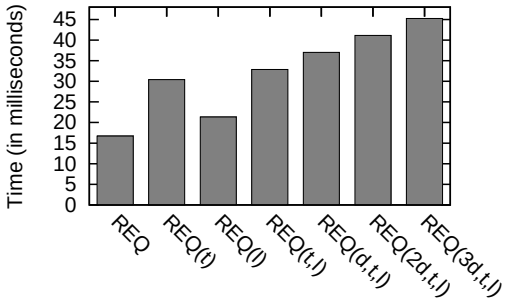


Fig. 6. Performance overhead of updating the Session with the request data.

E. Performance Evaluation of Session Update

After the PDP checks that the current request is not violating any deployed constraints and the request is granted, the Access History in the Session needs to be updated with the information in the executed request. The session update is managed by the PEP that executes the second round of encryption before storing the encrypted data in the Session. Figure 6 shows the performance overhead of encrypting the request for storing it in the Session. The graph shows the execution time of different formats of the REQ tuple: that is, from the basic format containing only subject, action and target information to more complex ones having time, location and a domain hierarchy of objects up to three levels.

VII. Related Work

There is a significant amount of research on enforcing dynamic security constraints including DSoD [6], [8], [9] and CW [10]. State of the art solutions including *GTRBAC* [11], *MFOTL* [27] and [7], [13] mainly focus on formally specifying the constraints. They assume a trusted infrastructure in order to enforce the constraints. There are some approaches that extend the enforcement mechanisms for taking into account contextual information such as time and location while making the access

decision [11], [28]. However, none of the existing approaches are applicable when the enforcement mechanism is delegated to a third party that is not trusted. These approaches operate on the constraints that are stored in cleartext. Unfortunately, these constraints may leak information about the internal policies of an organisation and can result in serious implications if not adequately protected.

There are some approaches for enforcing static security constraints in outsourced environments [17]–[21]. The idea of delegating the access control mechanism to an outsourced environment has initially been explored by De Capitani di Vimercati *et al.* in [21] and they extended it in [20]. Their proposed solution is based on the key derivation method [29], where each user has a key capable of decrypting resources she is authorised to access. The main drawback of this type of approaches is that they tightly couple security policies with the enforcement mechanism; therefore, any changes in the security policies require to generate new keys and to redistribute them to the users.

In [19], we propose *ESPOON*, where a data owner may attach an authorisation policy with her data while storing it on the server running in the outsourced environment. A data consumer may request for the data and get access if the authorisation policy is satisfied. *ESPOON* does not consider concept of roles. In [17], [18], we extend *ESPOON* for supporting an encrypted version of the RBAC model and propose $ESPOON_{ERBAC}$. In $ESPOON_{ERBAC}$, it is possible to enforce static security constraints, such as static separation of duties; however, it is not possible to delegate the enforcement of dynamic security constraints, such as HBDSoD and CW. The main issue is that the proposed architectures in [17], [18] lack to manage encrypted session management.

The security policy enforcement is mainly based on encrypted matching schemes in untrusted environments. There are number of schemes that address encrypted matching in outsourced environments [14]–[16], [24], [30], [31]. Song *et al.* [14] are the first to propose an encrypted matching scheme, where documents and requests are encrypted using symmetric keys. The main drawback of this scheme is that it is a single-user scheme. Multi-user Searchable Symmetric Encryption (MSSE) [16] is the first scheme to support encrypted matching in multi-user settings. In MSSE scheme, a data owner controls the search access by granting and revoking the search privileges to the users within her group by employing the symmetric encryption. The issue with scheme is that it requires redistribution of secret to all users once a user is revoked. Boneh *et al.* [15] are the first to propose the encrypted matching scheme in the public settings; however, it is not a multi-user scheme. Shao *et al.* [31] introduce Proxy Re-Encryption with keyword Search (PRES) scheme that is a combination of proxy encryption and PEKS. In PRES, a delegation key is generated for the target user. The target user re-encrypts the ciphertext with the delegated key. The re-encryption algorithm outputs another ciphertext corresponding to the public key of the target user. That is why, this scheme high performance overhead for re-encrypting ciphertext.

There are schemes based on ABE including CP-ABE [24] and Key-Policy ABE (KP-ABE) [30]. In CP-ABE, policies are attached with ciphertext; while, in KP-ABE, attributes are attached with ciphertext. The main issue is that both schemes leave policies and attributes in cleartext, respectively. Unfortunately, policies and attributes in cleartext may reveal private information about the encrypted data.

The homomorphic encryption schemes [32]–[34] allow untrusted parties to perform mathematical operations on encrypted data without compromising the encryption. There are a number of issues with these schemes. The major issue is scalability. Unfortunately, state-of-the-art schemes are not suitable in practice for processing a huge amount of data due to computational limitations. Another problem is the key management. These schemes consider a single user that can perform the decryption.

VIII. Conclusions and Future Work

In this paper, we have proposed E-GRANT, an architecture for enforcing dynamic security constraints as an outsourced service running in the cloud. The main contribution of E-GRANT is that it supports the enforcement of encrypted security constraints while maintaining encrypted session in the cloud. In this way, cloud providers learn neither about the information stored by the session nor about the content of security constraints being enforced. The approach provides a scalable key management, where users do not share any encryption keys. When users leave the organisations or keys get compromised, keys can be revoked without requiring the re-distribution of keys and the re-encryption of deployed constraints.

As future work, we are planning to explore ways of making the enforcement architecture accountable, thus preventing the cloud provider the possibility to repudiate the operations that have been performed. Another substantial part of our future research aims at re-engineering the architecture in a distributed manner in order to run several instances on multiple nodes of the cloud provider. One of the key aspects here is to adapt the number of instances to the actual request load in order to offer a reasonable Quality of Service (QoS) without over-provisioning the resources.

Acknowledgement

We would like to thank anonymous reviewers for providing their feedback for improving the quality of our work. The work of first and second authors was supported by the Security Technologies Returning Accountability, Transparency and User-centric Services in the Cloud (STRATUS) project, funded by the Ministry of Business, Innovation, and Employment (MBIE), New Zealand.

References

[1] Google, "Google cloud storage pricing," https://cloud.google.com/storage/#pricing, September 2014, last Accessed: March 9, 2015.

[2] Amazon, "Amazon s3 pricing," http://aws.amazon.com/s3/pricing/, September 2014, last Accessed: March 9, 2015.

[3] Google, "Introducing google drive... yes, really," http://googleblog.blogspot.it/2012/04/introducing-google-drive-yes-really.html, April 2012, google Official Blog, Last Accessed: September 9, 2014.

[4] P. Pehrson, "Adobe's new SAAS model," http://www.paulpehrson.com/2011/04/11/adobes-new-software-as-a-service-model/, April 2011, last Accessed: March 9, 2015.

[5] SAP, "SAP Business ByDesign," http://www.sap.com/pc/tech/cloud/software/business-management-bydesign/overview/index.html, last Accessed: March 9, 2015.

[6] D. Basin, S. J. Burri, and G. Karjoth, "Separation of duties as a service," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: ACM, 2011, pp. 423–429.

[7] J. Crampton and H. Khambhammettu, "A framework for enforcing constrained RBAC policies," in *Computational Science and Engineering, 2009. CSE '09. International Conference on*, vol. 3, August 2009, pp. 195–200.

[8] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *Computer*, vol. 29, pp. 38–47, February 1996.

[9] M. Nash and K. Poland, "Some conundrums concerning separation of duty," in *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, May 1990, pp. 201–207.

[10] D. Brewer and M. Nash, "The chinese wall security policy," in *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, May 1989, pp. 206–214.

[11] J. Joshi, E. Bertino, U. Latif, and A. Ghafoor, "A generalized temporal role-based access control model," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 17, no. 1, pp. 4–23, January 2005.

[12] G.-J. Ahn and R. Sandhu, "Role-based authorization constraints specification," *ACM Trans. Inf. Syst. Secur.*, vol. 3, pp. 207–226, November 2000.

[13] V. Gligor, S. Gavrila, and D. Ferraiolo, "On the formal definition of separation-of-duty policies and their composition," *Security and Privacy, IEEE Symposium on*, 1998.

[14] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Security and Privacy, 2000. S P 2000. Proceedings. 2000 IEEE Symposium on*, 2000, pp. 44–55.

[15] D. Boneh, G. Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in *Advances in Cryptology - EUROCRYPT 2004*, ser. Lecture Notes in Computer Science, C. Cachin and J. L. Camenisch, Eds. Springer Berlin Heidelberg, 2004, vol. 3027, pp. 506–522.

[16] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proceedings of the 13th ACM conference on Computer and communications security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 79–88.

[17] M. R. Asghar, M. Ion, G. Russello, and B. Crispo, "ESPOON$_{ERBAC}$: Enforcing security policies in outsourced environments," *Elsevier Computers & Security (COSE)*, vol. 35, pp. 2–24, 2013.

[18] M. R. Asghar, G. Russello, and B. Crispo, "Poster: $ESPOON_{ERBAC}$: Enforcing security policies in outsourced environments with encrypted RBAC," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 841–844.

[19] M. R. Asghar, M. Ion, G. Russello, and B. Crispo, "ESPOON: Enforcing encrypted security policies in outsourced environments," in *The Sixth International Conference on Availability, Reliability and Security*, ser. ARES'11, August 2011, pp. 99–108.

[20] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Encryption policies for regulating access to outsourced data," *ACM Trans. Database Syst.*, vol. 35, no. 2, pp. 12:1–12:46, May 2010.

[21] ——, "Over-encryption: management of access control evolution on outsourced data," in *Proceedings of the 33rd international conference on Very large data bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 123–134.

[22] V. C. Hu, D. F. Ferraiolo, and D. R. Kuhn, "Assessment of access control systems," National Institute of Standards and Technology, Tech. Rep., September 2006.

[23] A. Schaad, P. Spadone, and H. Weichsel, "A case study of separation of duty properties in the context of the austrian "elaw" process." in *Proceedings of the 2005 ACM symposium on Applied computing*, ser. SAC '05. New York, NY, USA: ACM, 2005, pp. 1328–1332.

[24] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *Security and Privacy, 2007. SP '07. IEEE Symposium on*, May 2007, pp. 321–334.

[25] C. Dong, G. Russello, and N. Dulay, "Shared and searchable encrypted data for untrusted servers," *Journal of Computer Security*, vol. 19, no. 3, pp. 367–397, 2011.

[26] M. R. Asghar, "Privacy preserving enforcement of sensitive policies in outsourced and distributed environments," Ph.D. dissertation, University of Trento, Trento, Italy, December 2013, http://eprints-phd.biblio.unitn.it/1124/.

[27] D. Basin, F. Klaedtke, and S. Müller, "Monitoring security policies with metric first-order temporal logic," in *Proceedings of the 15th ACM symposium on Access control models and technologies*, ser. SACMAT '10. New York, NY, USA: ACM, 2010, pp. 23–34.

[28] M. Strembeck and G. Neumann, "An integrated approach to engineer and enforce context constraints in RBAC environments," *ACM Trans. Inf. Syst. Secur.*, vol. 7, pp. 392–427, August 2004.

[29] M. J. Atallah, M. Blanton, N. Fazio, and K. B. Frikken, "Dynamic and efficient key management for access hierarchies," *ACM Trans. Inf. Syst. Secur.*, vol. 12, pp. 18:1–18:43, January 2009.

[30] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proceedings of the 13th ACM conference on Computer and communications security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 89–98.

[31] J. Shao, Z. Cao, X. Liang, and H. Lin, "Proxy re-encryption with keyword search," *Information Sciences*, vol. 180, no. 13, pp. 2576–2587, 2010.

[32] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 2009, aAI3382729.

[33] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," in *Advances in Cryptology - EUROCRYPT 2011*, ser. Lecture Notes in Computer Science, K. Paterson, Ed. Springer Berlin Heidelberg, 2011, vol. 6632, pp. 129–148.

[34] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology - EUROCRYPT 99*, ser. Lecture Notes in Computer Science, J. Stern, Ed. Springer Berlin Heidelberg, 1999, vol. 1592, pp. 223–238.